AN INVESTIGATION INTO THE PERFORMANCE OF INTEGRATION METHODOLOGIES FOR REAL TIME MASS-SPRING CLOTH SIMULATIONS

by

CHRISTOPHER PHILLIPS
ID: 15022229

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

MASTER OF SCIENCE IN COMPUTER GAMES PROGRAMMING

September 2016

Department of Computing
Staffordshire University
Stafford ST18 0AD

Supervised by: Christopher McCreadie

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

Christopher Phillips
September 2016

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

Literature Review

1.1 Cloth Simulation

The simulation of cloth is a relatively old and well studied field with applications in many different areas, including, but not limited to:

- Virtual Garment Design
- Virtual Fitting Rooms
- Films
- Video Games

Different use cases require different things from the cloth simulation. For example, in virtual garment design, the physical accuracy of the simulation is paramount whereas cloth simulation for video games prioritises real-time simulation, sacrificing accuracy. Hence, many different models for cloth simulation have been proposed.

Since this project is concerned with the real-time simulation of cloth, only those models appropriate for real-time simulations have been studied in detail. However, a brief overview of other techniques will be provided. For a more detailed overview of cloth simulation techniques, see Ng and Grimsdale (1996).

1.1.1 Cloth Properties

Cloth has several properties that should be considered for modelling.

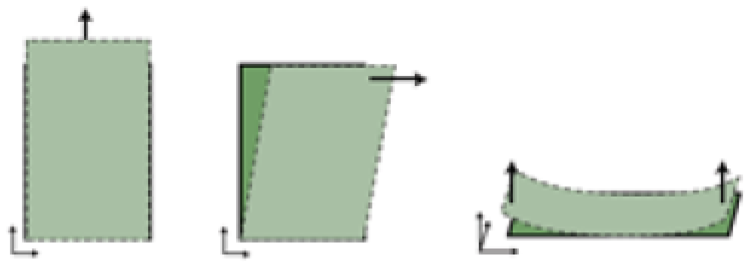


Figure 1.1: Mechanical properties of cloth (*Techniques for Animating Cloth*, p. 1)

1.1.1.1 Mechanical Properties

Cloth has three mechanical properties that control its behaviour; stretching, shearing and bending, fig. 1.1 shows how each property affects the cloth.
Stretching is the displacement of the cloth in either the horizontal or vertical direction. Most cloth has a high resistance to stretching and can typically only be stretched by 10% (*Techniques for Animating Cloth*, p. 1; Provot 1995, p. 4).
Shearing is the displacement of the cloth in a diagonal direction. Again, most cloths have high shearing resistance and this, coupled with high stretch resistance, makes cloth incompressible.
Finally, bending is the overall curvature of the cloth surface. Typically, cloth has low bending resistance and so is easily folded.

1.1.1.2 Visual Properties

The mechanical properties of cloth, discussed above, cause cloth to exhibit two visual properties. These properties arise from the fact that cloth is typically non-elastic, due to stretch and shear resistances, but highly flexible, due to low bend resistance.
Firstly, cloth will drape over objects and secondly the cloth will form many folds and wrinkles. Fig 1.2 demonstrates the visual properties of cloth.

1.1.2 Cloth Models

Techniques for modelling cloth are usually classified as either Geometric or Physically-based, and the choice of which modelling method to use depends on the use-case for the simulation.

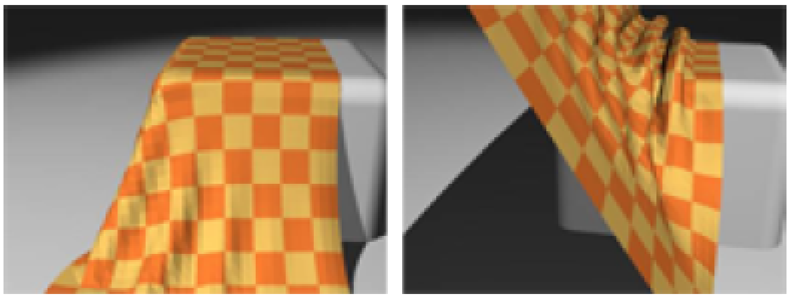


Figure 1.2: Visual properties of cloth (*Techniques for Animating Cloth*, p. 1)

1.1.2.1 Geometric Models

This family of techniques were the first models used to simulate cloth. They model the cloth using geometric equations and are especially good at modelling folds and wrinkles.
Weil was the first to propose a geometric model in 1986 and uses catenary curves to model the drape and folds of a hanging cloth. Following on from Weil, a number of other geometric models were proposed (see Ng and Grimsdale (1996) for more information).
All geometric techniques focus on simulating the appearance of cloth, rather than the physical properties. As such, geometric models are typically more computationally efficient than physically-based models, as there is no need to solve a series of complex equations. However, geometric techniques are unable to accurately simulate the motion of cloth, (Mongus et al. 2012, p. 1; Zhang and Yuen 2001, p. 2; Xinrong et al. 2009, pp. 1-2), and so are mostly useful for static cloth simulations.
As such, geometric models have not been considered for this project.

1.1.2.2 Physically-based Models

1.2 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.

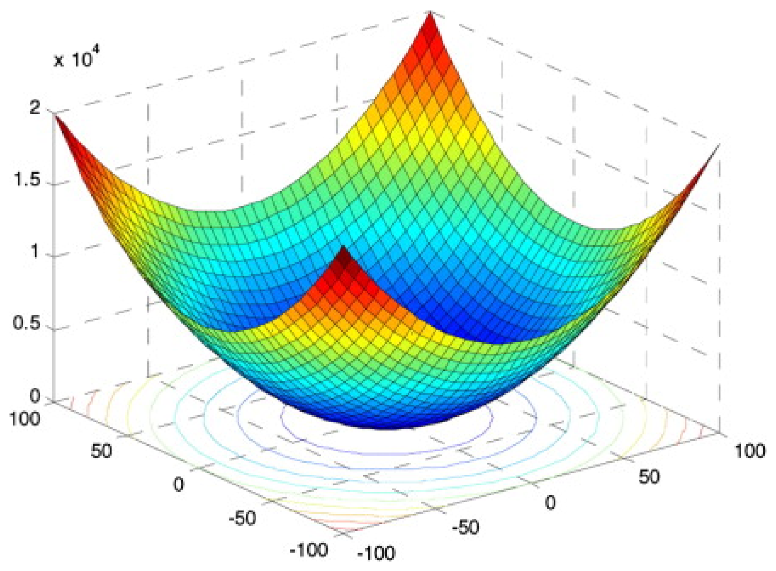


Figure 1.3: Fitness landscape for the 2D sphere function

Optimisation problems typically take the form**mathprog**

maximise (minimise) $f(x)$: x in X

subject to:

$X \in \mathbb{R}$

$g(x) < 0$

$h(x) = 0$

(1.1)

where:

X is the set of values for x; the domain of the function

$f(x)$ is the function being optimisied; the objective function

$g(x)$, $h(x)$ are constraints on x

The number of contraints depends on the optimisation problem

The goal of optimisation techniques is to "find the peak of the fitness landscape"**stoch** 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 1.3**fitland** for an example fitness landscape for the 2D sphere function.

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

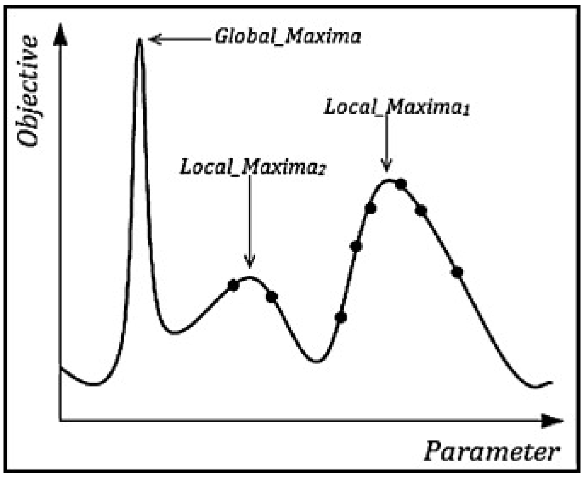


Figure 1.4: A simple fitness landscape showing 2 local optima

1.2.1 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 1.4**localo** 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.

1.3 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"**mccarthy** It is more commonly defined as "the study of the design of intelligent agents"**poole**
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.

1.4 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."**coello**

1.4.1 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.

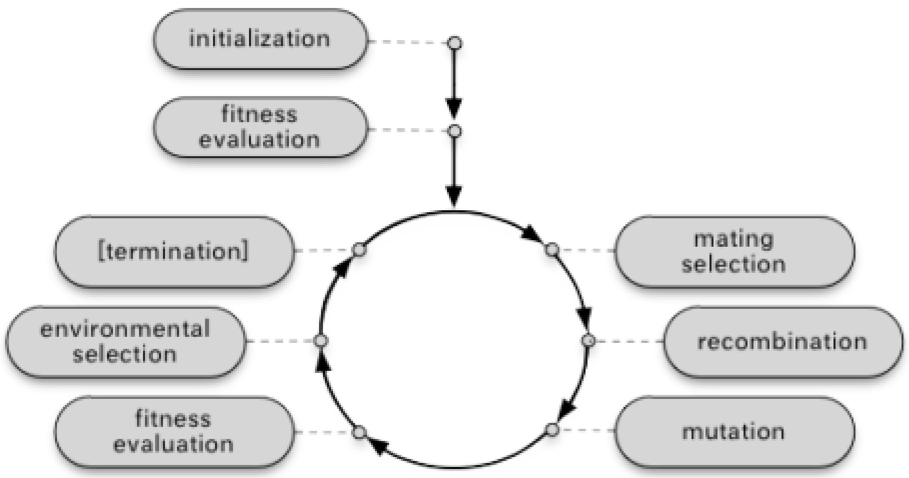
1.4.2 Operators

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

1.4.3 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 1.5 for a diagram**eadiag** and related pseudocode**eapseudo** 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.



(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 1.5: Diagram and pseudocode for the operation of a generic EA

1.4.4 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.

1.4.5 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 that interest in GAs took off, with John Holland's book *Adaptation in Natural and Artificial Systems***4-ga**; **gahist**; **pga1** GAs are designed to model genetic evolution and use the selection and recombination operators as the main evolutionary force**4-ga**

1.4.5.1 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 1.4.1).
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**4-en**

$$f_j = x_{min,j} + \frac{x_{max,j} - x_{min,j}}{2^{n_d} - 1}\left(\sum_{l=1}^{n_d-1} b_{j(n_d-l)} 2^l\right) \tag{1.2}$$

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 1.2, the equation for encoding individuals is obtained:

$$b_j = round\left(\frac{f_j - x_{min,j}}{\frac{x_{max,j} - x_{min,j}}{2^{n_d} - 1}}\right) \tag{1.3}$$

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"**4-en** 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

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 1.1: Differences between binary and grey coding

required to get to 8. Grey coding resolves this problem by ensuring that the Hamming distance between successive numbers is always 1, see table 1.1. Binary code is converted into grey code as follows**4-en**

$$\begin{aligned} g_1 &= b_1 \\ g_l &= b_{l-1}\bar{b}_l + \bar{b}_{l-1}b_l \end{aligned} \tag{1.4}$$

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 1.2**4-en**

$$f_j = x_{min,j} + \frac{x_{max,j} - x_{min,j}}{2^{n_d} - 1}\left(\sum_{l=1}^{n_d-1}\left(\sum_{q=1}^{n_d-l} b_{(j-1)n_d+q}\right) mod\ 2\right)2^l \tag{1.5}$$

Real value encoded GAs have also been proposed. The use of real values makes recombination and mutation more complicated, whilst making evaluation simpler. For this project, only binary or grey coded GAs were considered.

1.4.5.2 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

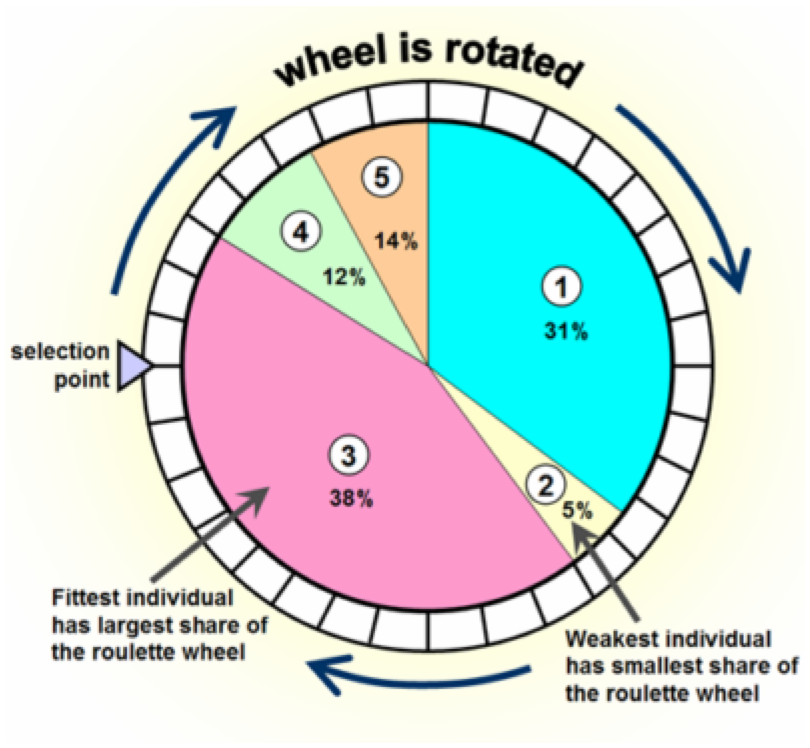

wheel is rotated
selection point
1
31%
2
5%
3
38%
4
12%
5
14%
Fittest individual has largest share of the roulette wheel
Weakest individual has smallest share of the roulette wheel

Figure 1.6: Representation of roulette wheel selection

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**4-se**; **eoselect**; **gaselect**; **gatsp**

$$P(x_i) = \frac{f(x_i)}{\sum_{j=1}^{n} f(x_j)} \tag{1.6}$$

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 1.6**rw**).

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**4-se**

$$scaledfitness(x_i) = f_{max} - f(x_i) \tag{1.7}$$

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**4-se**

$$scaledfitness(x_i) = \frac{1}{1 + f(x_i) - f_{min}(t)} \tag{1.8}$$

The advantage of proportional selection is that it preserves diversity by affording all individuals a probability of selection**gatsp**; **eoselect** 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**4-se**; **gatsp**

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)**gaselect** or rank 0**4-se** 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

$$\begin{aligned} P(x_i) &= \frac{1}{N}(\eta^- + (\eta^+ - \eta^-)\frac{i-1}{N-1})\textbf{gaselect} \\ or \\ P(x_i) &= \frac{1}{N}(\eta_{max} - 2(\eta_{max} - 1)\frac{i-1}{N-1}\textbf{eoselect} \end{aligned} \tag{1.9}$$

For this project however, the following method is used**l2**

$$P(x_i) = \frac{rank(x_i)}{\sum_{j=1}^{n} rank(x_j)} \tag{1.10}$$

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

```
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

may still lead to premature convergence as the best chromosomes are not so strongly differentiated**gatsp**

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**4-se**; **eoselect**; **gaselect**; **gatsp** 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**4-se**; **gatsp** A common tournament size is 2, and tournaments of this variety are called *binary tournaments***gaselect**; **gatsp**
For details on other selection methods, see **4-se** and **gaselect**

1.4.5.3 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**4-ga**). The purpose of a crossover operator is to compute the bit mask. Three operators have been developed for binary crossover, fig 1.7(a)**gacross**

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 1.7(b)**l2** 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 1.7(c)**l2** shows an example of uniform crossover with the bit mask

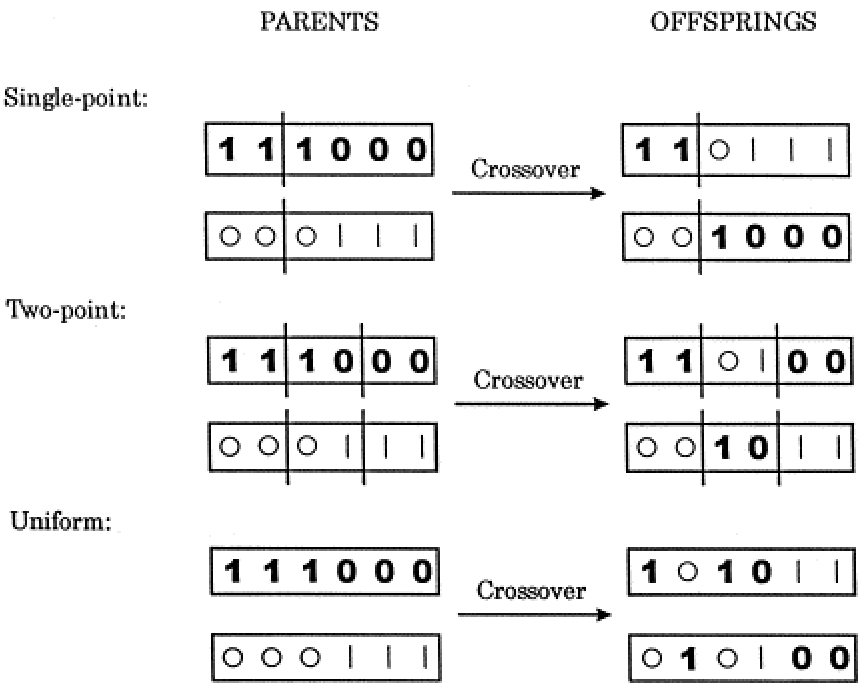
A configurable parameter controls crossover. Called the *crossover rate*, it is effectively the probability that crossover will take place. Generally, a high crossover rate is used; between 0.5 and 1**4-ga**; **garates**

1.4.5.4 Mutation

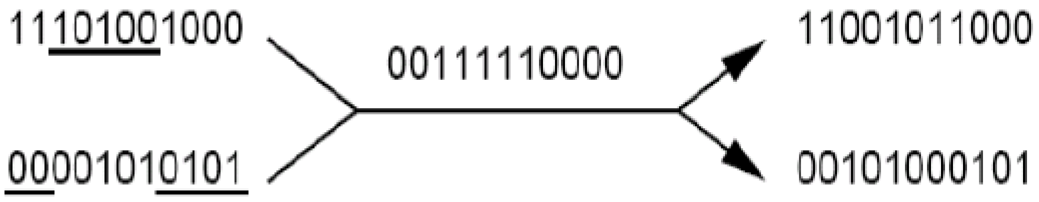
Mutation is applied to the offspring produced by crossover in order to add further genetic variation to the population. In binary or grey coded GAs, the process is very simple; the value of certain bits is inverted. The bits to invert are selected using one of two methods

- 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.

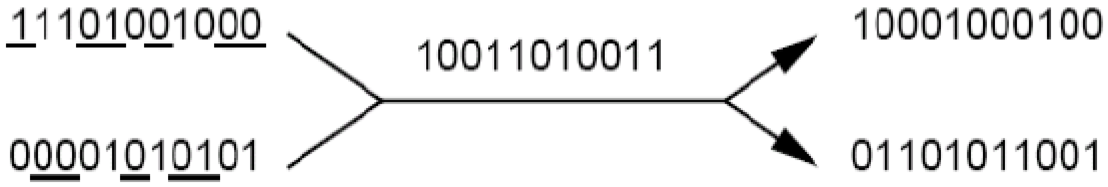
Gaussian mutation similar to that used in ES (see section 1.4.6.4) can be used for GAs where the phenotype is floating-point numbers. The genotype is first decoded into the phenotype, gaussian mutation is applied and then the result is converted back into the genotype.**4-ga**
As with crossover, mutation 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**4-ga**; **garates**



(a) Crossover operators



(b) Two-point crossover with bit mask



(c) Uniform crossover with bit mask

Figure 1.7: Summary showing the three sexual crossover operators (a) with bit masks (b), (c)

1.4.5.5 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**gap**; **iga**
The key parameters are

- Population size
- Number of generations
- 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

1.4.6 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"**4-es** 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**4-es**; **es** The first multimembered ES, $(\mu + 1)$, 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**4-es**; **es** Schwefel later introduced two more multimembered ES: (μ, λ) and $(\mu + \lambda)$**es** which use a population of μ parents and produce λ offspring. These types of ES are often shown as $(\mu \overset{+}{,} \lambda)$.

1.4.6.1 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**4-es**; **es** 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**es** The traditional way of mutating the strategy parameters was using the 1/5th-rule**es** 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.

1.4.6.2 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**4-es**; **es**; **esc** 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**4-es**

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**4-es**; **es**; **esc**

1.4.6.3 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**4-es** or multi**es** recombination. There are two simple methods of recombination, *discrete* and *intermediate*.

Discrete recombination is denoted as $(\mu/\rho_D \overset{+}{,} \lambda)$. 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 $r = (r_1, ..., r_n)$,**es**

$$\begin{aligned} r_k &= (a_p)_k \\ \text{where } p &= random[1, ..., \rho] \end{aligned} \tag{1.11}$$

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 $(\mu/\rho_I \overset{+}{,} \lambda)$ 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 $r = (r_1, ..., r_n)$,**es**

$$r_k = \frac{1}{\rho} \sum_{p=1}^{\rho} (a_p)_k \tag{1.12}$$

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

1.4.6.4 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**4-es**; **es**

$$\begin{aligned} y' &= y + z \\ \text{where } z &= \sigma(N_1(0,1), ..., N_n(0,1)) \end{aligned} \tag{1.13}$$

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**4-es**; **es**

$$\begin{aligned} y' &= y + z \\ \text{where } z &= (\sigma_1 N_1(0,1), ..., \sigma_n N_n(0,1)) \end{aligned} \tag{1.14}$$

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**4-es**; **es**

$$\begin{aligned} &\sigma' = \sigma exp(\tau N(0,1)) \\ &\text{where} \\ &\tau = \frac{1}{\sqrt{n}} \\ &n \text{ is the size of the search space, i.e. the number of decision variables} \end{aligned} \tag{1.15}$$

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**4-es**; **es**

$$\begin{aligned} &\sigma_i' = \sigma_i exp(\tau' N(0,1)) exp(\tau N_1(0,1)) \\ &\text{where} \\ &\tau' = \frac{1}{\sqrt{2n}} \\ &\tau = \frac{1}{\sqrt{2\sqrt{n}}} \\ &n \text{ is the size of the search space, i.e. the number of decision variables} \end{aligned} \tag{1.16}$$

1.4.6.5 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 1.8**constraints** 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 1.8, if the optimum was at d, the algorithm would have to move through the infeasible region to reach it.

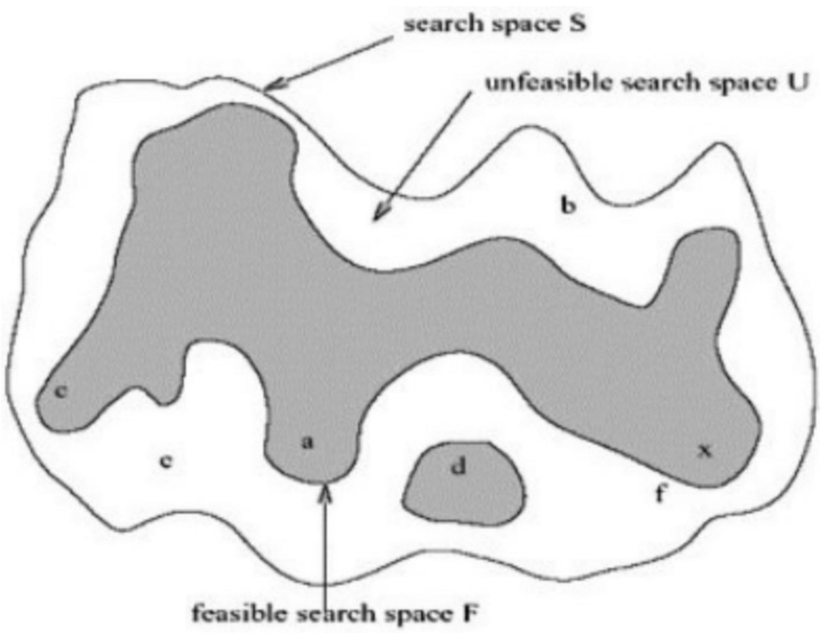


Figure 1.8: A search space, showing the feasible and infeasible regions

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"**gecco** For a constrained optimisation problem defined as in equation 1.1, a penalised fitness function is usually takes this form**4-ga**; **gecco**

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

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 $|h_j(x)|^\gamma$ (1.17)

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

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**4-ga**; **constraints**; **gecco**

$$f'(x) = f(x) + (\sum_{i=1}^{n} R_{j,i} G_i) \quad \text{where } R_{j,i} \text{ is the } j^{th} \text{ violation level for constraint } i \tag{1.18}$$

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**constraints**; **gecco**

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**4-ga**; **esc**; **constraints**; **gecco**

$$\begin{gathered} f'(x) = f(x) + (C.t)^{\alpha} (\sum_{i=1}^{n} G_i^{\beta}(x)) \\ \text{where } C,\ \alpha,\ \beta \text{ are user defined constants and } t \text{ is the generation number} \\ \text{typically } C = 0.5,\ \alpha = 1 \text{ or } 2 \text{ and } \beta = 1 \text{ or } 2 \end{gathered} \tag{1.19}$$

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**constraints**

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**esc**; **constraints**; **gecco**

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"**esrev** The repaired solution can then be used to evaluate its infeasible equivalent, or it can replace the infeasible solution in the population**constraints**; **esrev** The specifics of the repair algorithm are usually problem dependent, so there are few standard repair techniques**constraints**
Many other constraint handling mechanisms exist, see **constraints** for details.

1.4.6.6 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

1.5 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"**4-IV** 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***4-IV** 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).

1.5.1 Particle Swarm Optimisation

Developed in the 1990s by James Kennedy and Russell Eberhart, PSO attempts to model the behaviour of birds in a flock**4-pso**; **mopso** The complex behaviour displayed by PSO is that of "all individuals converging on the environment state that is best for all individuals"**4-IV** 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"**4-pso**; **mopso**

1.5.1.1 Basic Particle Swarm Optimisation Algorithm

PSO is much simpler than EAs, discussed in section 1.4. 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, $v_i(t)$**4-pso**; **mopso**; **psot**

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{1.20}$$

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*.

1.5.1.2 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**4-pso**; **psot**

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

where

c_1 and c_2 are user supplied acceleration constants, typically set to 2**psot**; **psoi** (1.21)

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.

1.5.1.3 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 a small neighbourhood is used now; the social component is called the *local best value*. The velocity for a particle is

```
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 1.21;
        update position vector using equation 1.20;
    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 1.22;
        update position vector using equation 1.20;
    end
end
```

Algorithm 3: lbest Particle Swarm Optimisation algorithm

updated by**4-pso**

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

where

c_1 and c_2 are user supplied acceleration constants, typically set to 2**psot**; **psoi**

r_1 and r_2 are fresh uniform random numbers (1.22)

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

$lbest_i(t)$ is the local best value of the neighbourhood of individual i

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

1.5.1.4 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**4-pso**; **mopso**; **psoi** Taking the gbest velocity equation (equation 1.21), adding an inertia weight changes the equation to the following

$$v_i(t+1) = \omega v_i(t) + c_1 r_1(pbest_i(t) - x_i(t)) + c_2 r_2(lbest_i(t) - x_i(t)) \tag{1.23}$$

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**psot** 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**4-pso**; **psoi** The inertia weight is updated using the following equation**4-pso**; **psoi**; **vi**

$$w(t) = \frac{(w_{max} - w_{min})(t_{max} - t)}{t_{max}} + w_{min} \tag{1.24}$$

where t_{max} is the maximum number of iterations

There are other methods of adjusting the inertia weight, see **4-pso** for details.

1.5.1.5 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}**4-pso**; **psot**; **psoi** In an optimisation problem bounded by $[-x_{max}, x_{max}]$, the value of v_{max} is**psot**

$$v_{max} = kx_{max} \tag{1.25}$$

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**4-pso**; **psot**

$$v_{max} = \frac{k(x_{max} - x_{min})}{2} \tag{1.26}$$

The value k has been shown to be problem dependent and should be defined in the range (0, 1]**4-pso**; **psot**

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 **4-pso**

1.5.1.6 Bound Handling

When dealing with constrained optimisation problems, many of the constraint handling methods used in EAs, see section 1.4.6.5, 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**4-pso** 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**psobh**

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**psobh**

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**psobh**

More details on these bound handling methods and a few others can be found in **psobh**

1.5.1.7 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

1.6 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"**connect4** 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 **connect4**

- 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**connect4**

1.7 Technologies

1.7.1 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.
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.

Library	Supported Algorithms
ECJ **ecj**	GAs, ES, PSO + more
cilib **cilib**	GAs, ES, PSO + more
JGAL **jgal**	GAs
JGAP **jgap**	GAs
JSwarm-PSO **jswarm**	PSO

Table 1.2: Java computational intelligence libraries

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

1.7.2 Computational Intelligence Libraries

There are a number of libraries providing CI algorithms, some of which are listed in table 1.2. 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 1.2 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**shark** was chosen. Shark release 2.3.4**sharksf** 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.

In 1988 C was totally awesome. (*Techniques for Animating Cloth*, p. 1; Provot 1995, p. 4). According to Miguel et al. (2012) C++ was even better.

Appendix A

Class Diagrams

A.1 Optimisation library Class Diagram

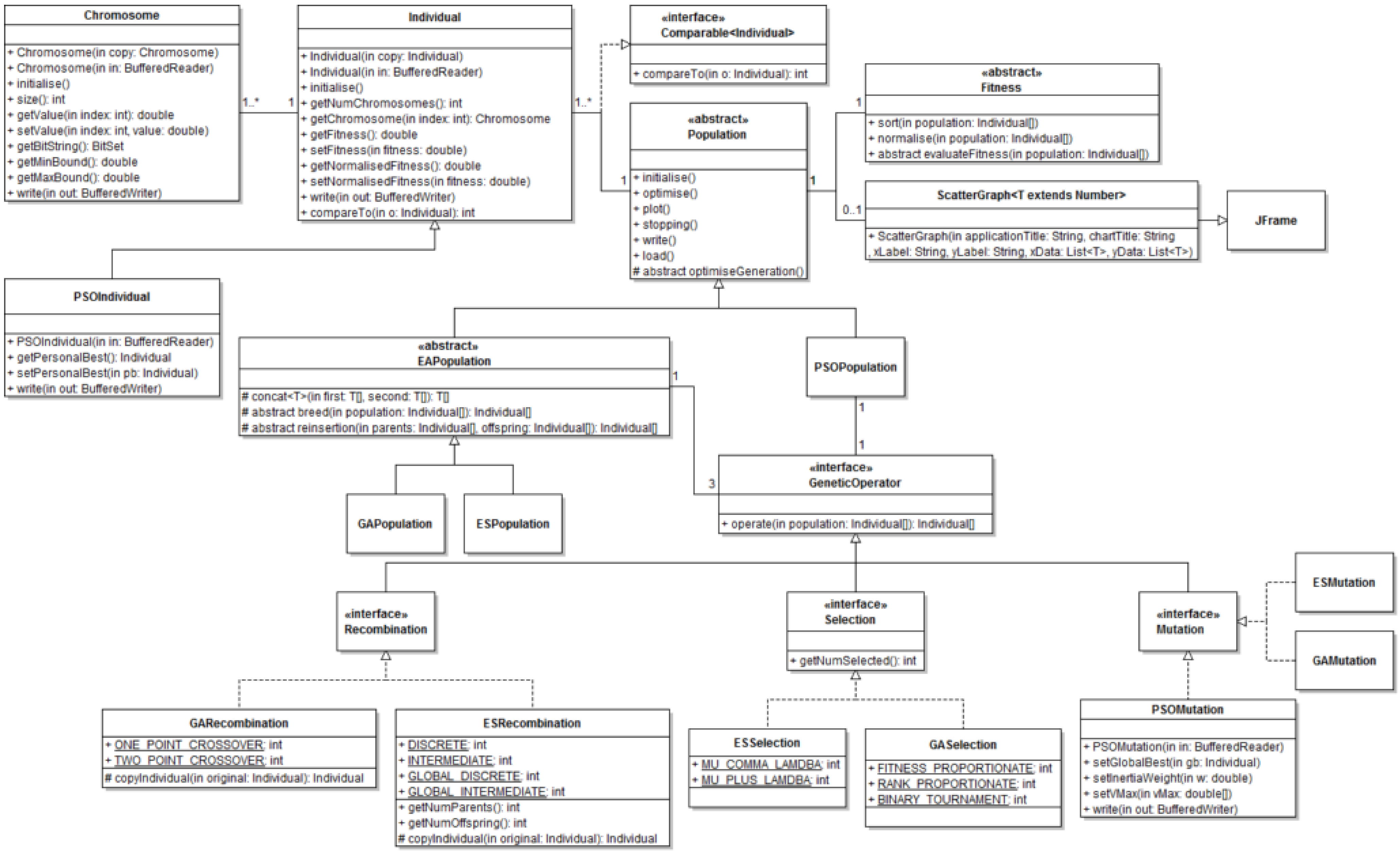

Chromosome
+ Chromosome(in copy: Chromosome)
+ Chromosome(in in: BufferedReader)
+ initialise()
+ size(): int
+ getValue(in index: int): double
+ setValue(in index: int, value: double)
+ getBitString(): BitSet
+ getMinBound(): double
+ getMaxBound(): double
+ write(in out: BufferedWriter)
1..*
1
Individual
+ Individual(in copy: Individual)
+ Individual(in in: BufferedReader)
+ initialise()
+ getNumChromosomes(): int
+ getChromosome(in index: int): Chromosome
+ getFitness(): double
+ setFitness(in fitness: double)
+ getNormalisedFitness(): double
+ setNormalisedFitness(in fitness: double)
+ write(in out: BufferedWriter)
+ compareTo(in o: Individual): int
1..*
«interface»
Comparable<Individual>
+ compareTo(in o: Individual): int
1
«abstract»
Population
+ initialise()
+ optimise()
+ plot()
+ stopping()
+ write()
+ load()
# abstract optimiseGeneration()
1
1
«abstract»
Fitness
+ sort(in population: Individual[])
+ normalise(in population: Individual[])
+ abstract evaluateFitness(in population: Individual[])
0..1
ScatterGraph<T extends Number>
+ ScatterGraph(in applicationTitle: String, chartTitle: String
, xLabel: String, yLabel: String, xData: List<T>, yData: List<T>)
JFrame
PSOIndividual
+ PSOIndividual(in in: BufferedReader)
+ getPersonalBest(): Individual
+ setPersonalBest(in pb: Individual)
+ write(in out: BufferedWriter)
«abstract»
EAPopulation
# concat<T>(in first: T[], second: T[]): T[]
# abstract breed(in population: Individual[]): Individual[]
# abstract reinsertion(in parents: Individual[], offspring: Individual[]): Individual[]
1
PSOPopulation
1
1
3
«interface»
GeneticOperator
+ operate(in population: Individual[]): Individual[]
GAPopulation
ESPopulation
«interface»
Recombination
«interface»
Selection
+ getNumSelected(): int
«interface»
Mutation
ESMutation
GAMutation
GARecombination
+ ONE_POINT_CROSSOVER: int
+ TWO_POINT_CROSSOVER: int
# copyIndividual(in original: Individual): Individual
ESRecombination
+ DISCRETE: int
+ INTERMEDIATE: int
+ GLOBAL_DISCRETE: int
+ GLOBAL_INTERMEDIATE: int
+ getNumParents(): int
+ getNumOffspring(): int
# copyIndividual(in original: Individual): Individual
ESSelection
+ MU_COMMA_LAMDBA: int
+ MU_PLUS_LAMDBA: int
GASelection
+ FITNESS_PROPORTIONATE: int
+ RANK_PROPORTIONATE: int
+ BINARY_TOURNAMENT: int
PSOMutation
+ PSOMutation(in in: BufferedReader)
+ setGlobalBest(in gb: Individual)
+ setInertiaWeight(in w: double)
+ setVMax(in vMax: double[])
+ write(in out: BufferedWriter)

Figure A.1: Class diagram for the optimisation library

A.2 The Arena Class Diagram

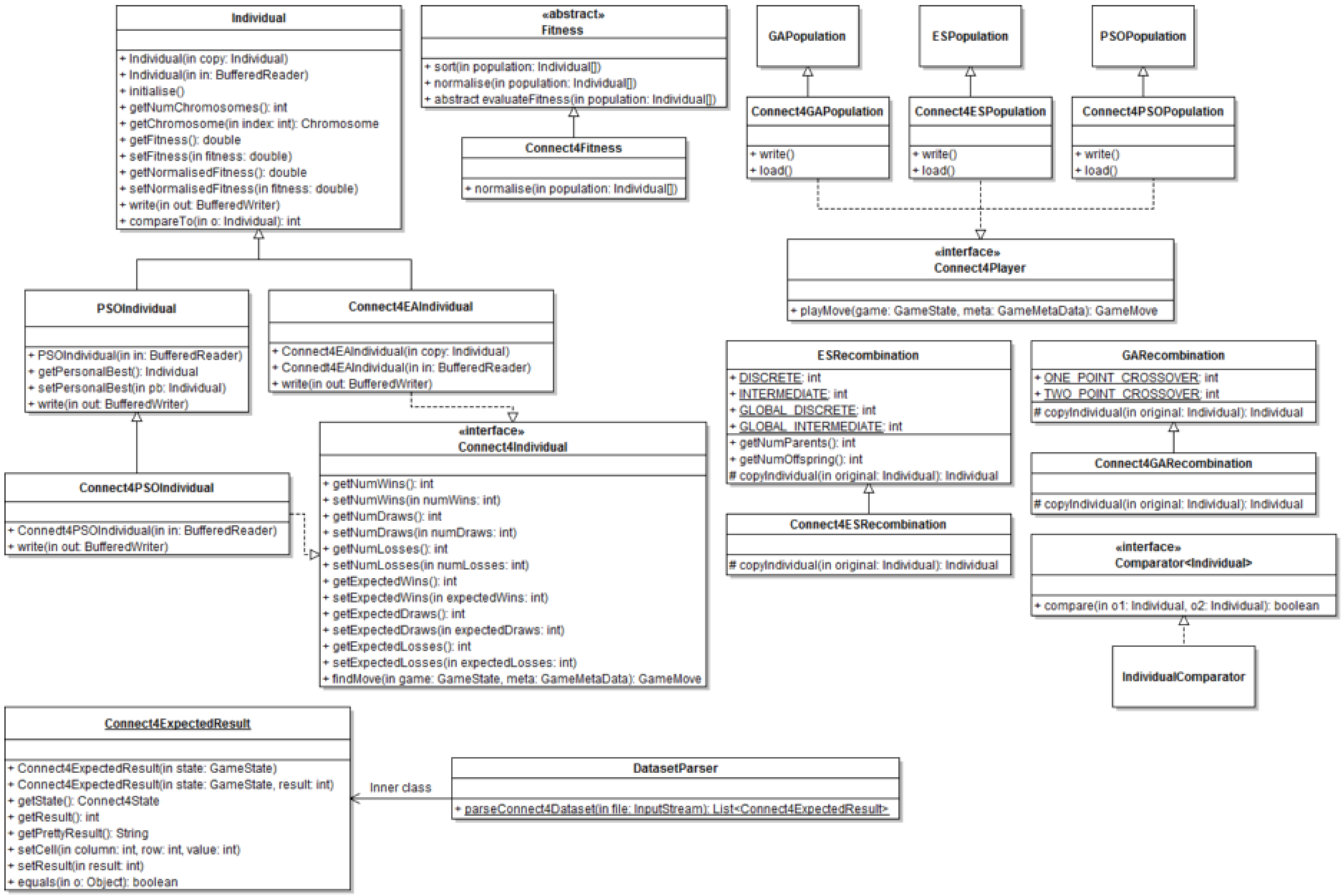

Individual
+ Individual(in copy: Individual)
+ Individual(in in: BufferedReader)
+ initialise()
+ getNumChromosomes(): int
+ getChromosome(in index: int): Chromosome
+ getFitness(): double
+ setFitness(in fitness: double)
+ getNormalisedFitness(): double
+ setNormalisedFitness(in fitness: double)
+ write(in out: BufferedWriter)
+ compareTo(in o: Individual): int
«abstract»
Fitness
+ sort(in population: Individual[])
+ normalise(in population: Individual[])
+ abstract evaluateFitness(in population: Individual[])
Connect4Fitness
+ normalise(in population: Individual[])
GAPopulation
ESPopulation
PSOPopulation
Connect4GAPopulation
+ write()
+ load()
Connect4ESPopulation
+ write()
+ load()
Connect4PSOPopulation
+ write()
+ load()
«interface»
Connect4Player
+ playMove(game: GameState, meta: GameMetaData): GameMove
PSOIndividual
+ PSOIndividual(in in: BufferedReader)
+ getPersonalBest(): Individual
+ setPersonalBest(in pb: Individual)
+ write(in out: BufferedWriter)
Connect4EAIndividual
+ Connect4EAIndividual(in copy: Individual)
+ Connect4EAIndividual(in in: BufferedReader)
+ write(in out: BufferedWriter)
ESRecombination
+ DISCRETE: int
+ INTERMEDIATE: int
+ GLOBAL_DISCRETE: int
+ GLOBAL_INTERMEDIATE: int
+ getNumParents(): int
+ getNumOffspring(): int
# copyIndividual(in original: Individual): Individual
GARecombination
+ ONE_POINT_CROSSOVER: int
+ TWO_POINT_CROSSOVER: int
# copyIndividual(in original: Individual): Individual
Connect4GARecombination
# copyIndividual(in original: Individual): Individual
Connect4ESRecombination
# copyIndividual(in original: Individual): Individual
«interface»
Connect4Individual
+ getNumWins(): int
+ setNumWins(in numWins: int)
+ getNumDraws(): int
+ setNumDraws(in numDraws: int)
+ getNumLosses(): int
+ setNumLosses(in numLosses: int)
+ getExpectedWins(): int
+ setExpectedWins(in expectedWins: int)
+ getExpectedDraws(): int
+ setExpectedDraws(in expectedDraws: int)
+ getExpectedLosses(): int
+ setExpectedLosses(in expectedLosses: int)
+ findMove(in game: GameState, meta: GameMetaData): GameMove
Connect4PSOIndividual
+ Connect4PSOIndividual(in in: BufferedReader)
+ write(in out: BufferedWriter)
«interface»
Comparator<Individual>
+ compare(in o1: Individual, o2: Individual): boolean
IndividualComparator
Connect4ExpectedResult
+ Connect4ExpectedResult(in state: GameState)
+ Connect4ExpectedResult(in state: GameState, result: int)
+ getState(): Connect4State
+ getResult(): int
+ getPrettyResult(): String
+ setCell(in column: int, row: int, value: int)
+ setResult(in result: int)
+ equals(in o: Object): boolean
Inner class
DatasetParser
+ parseConnect4Dataset(in file: InputStream): List<Connect4ExpectedResult>

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

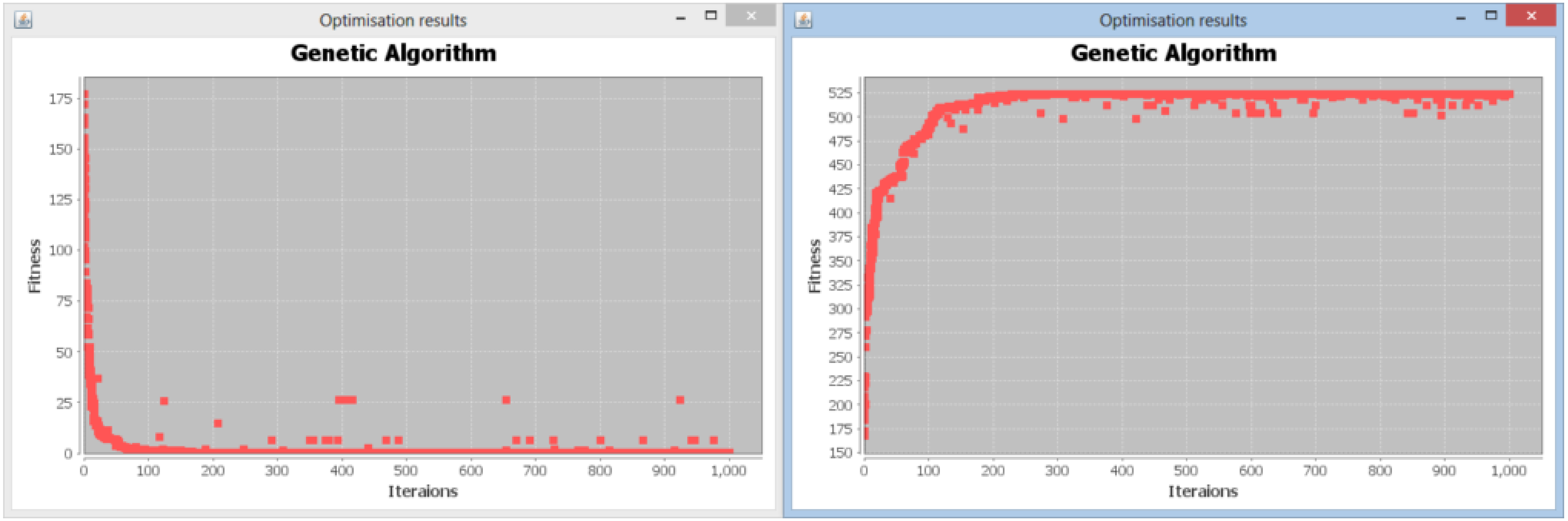


Figure B.1: Optimisation results of a genetic algorithm using binary tournament selection and two-point crossover

B.1.2 Binary Tournament Selection With One-point Crossover

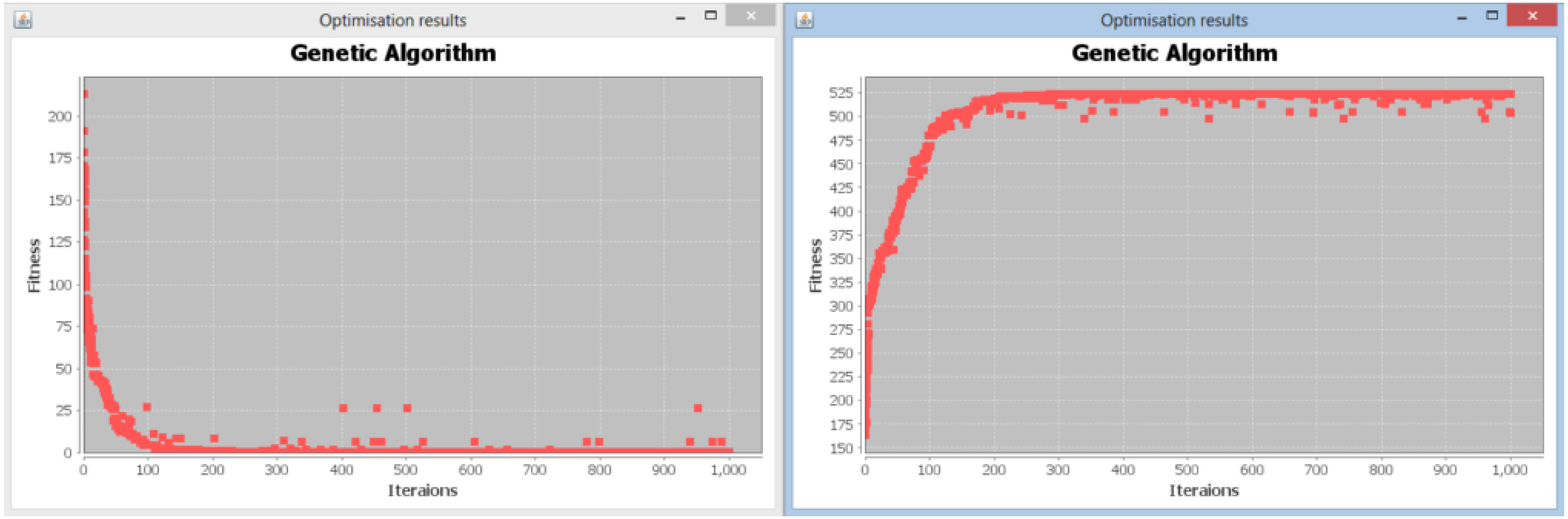
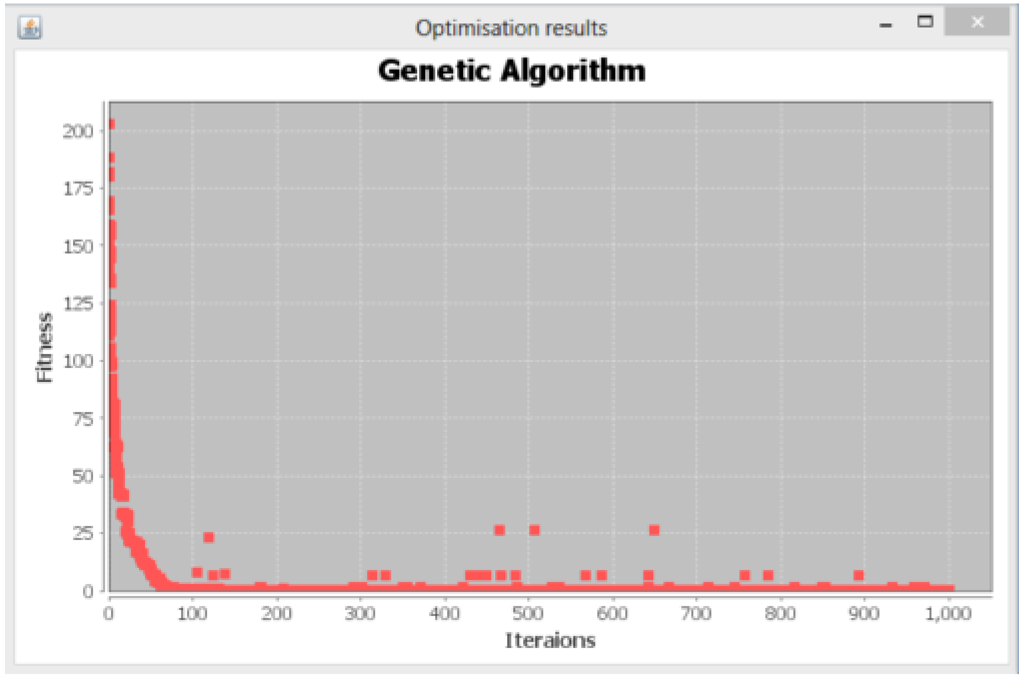


Figure B.2: Optimisation results of a genetic algorithm using binary tournament selection and one-point crossover

B.1.3 Rank Selection With Two-point Crossover



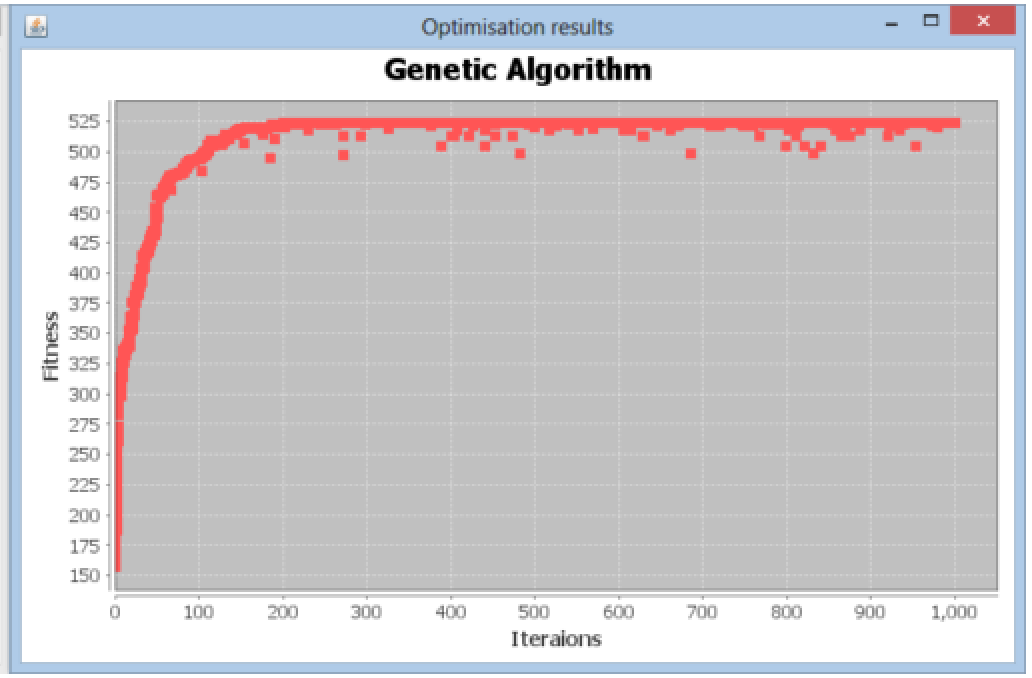


Figure B.3: Optimisation results of a genetic algorithm using rank selection and two-point crossover

B.1.4 Rank Selection With One-point Crossover

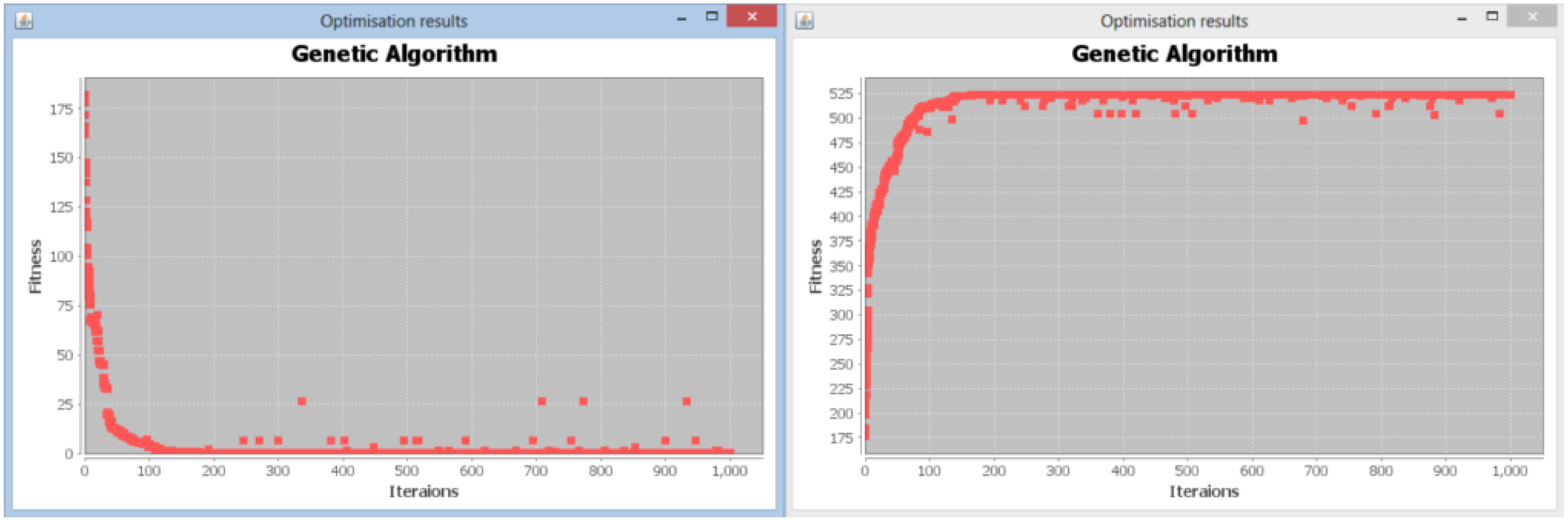


Figure B.4: Optimisation results of a genetic algorithm using rank selection and one-point crossover

B.1.5 Proportional Selection With Two-point Crossover

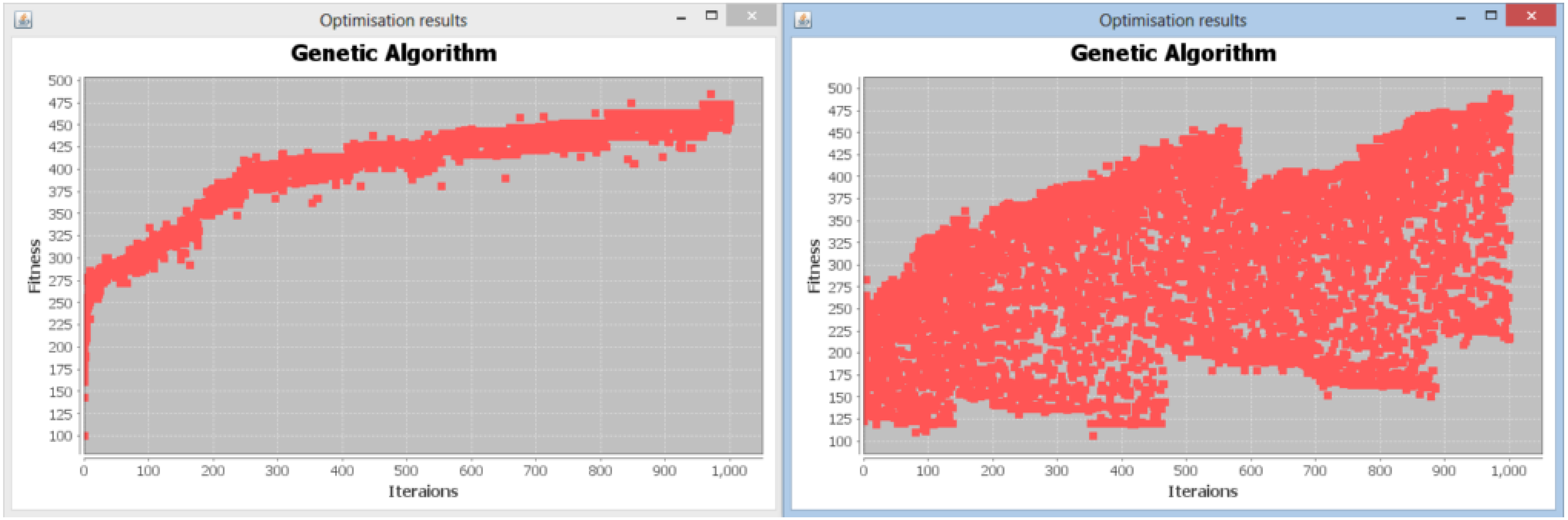


Figure B.5: Optimisation results of a genetic algorithm using proportional selection and two-point crossover

B.1.6 Proportional Selection With One-point Crossover

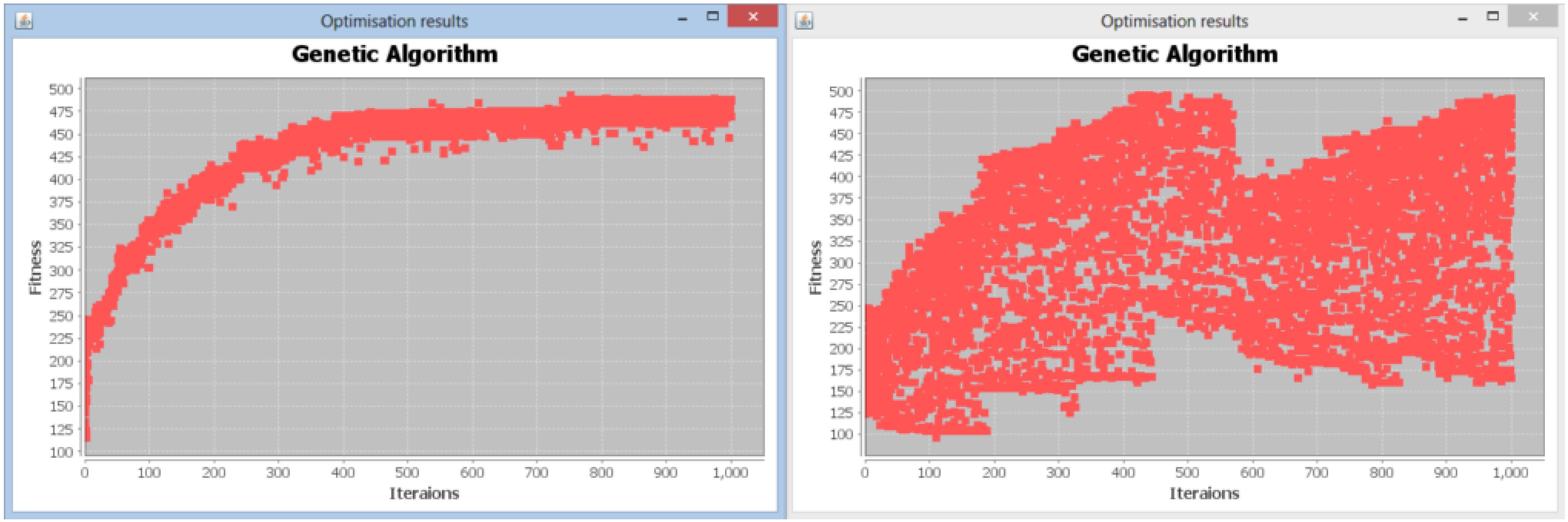


Figure B.6: Optimisation results of a genetic algorithm using proportional selection and one-point crossover

B.1.7 Proportional Selection With Adjusted Numbers of Parents

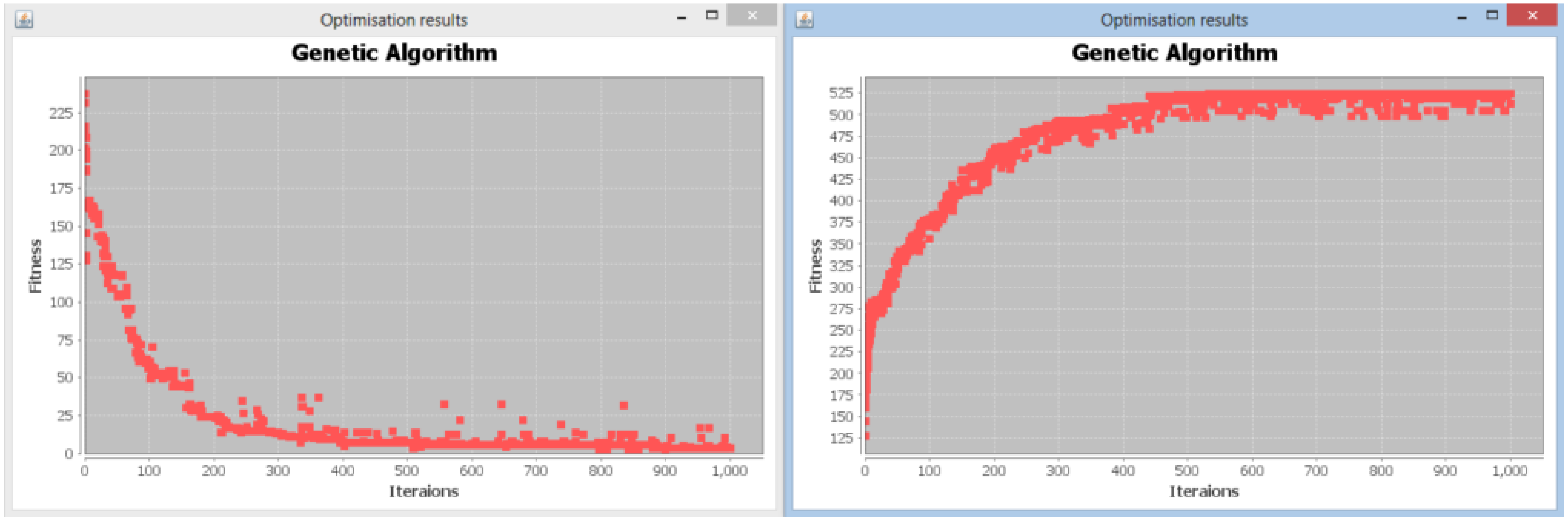


Figure B.7: Optimisation results of a genetic algorithm using proportional selection and two-point crossover. The number of parents has been adjusted to give better results.

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

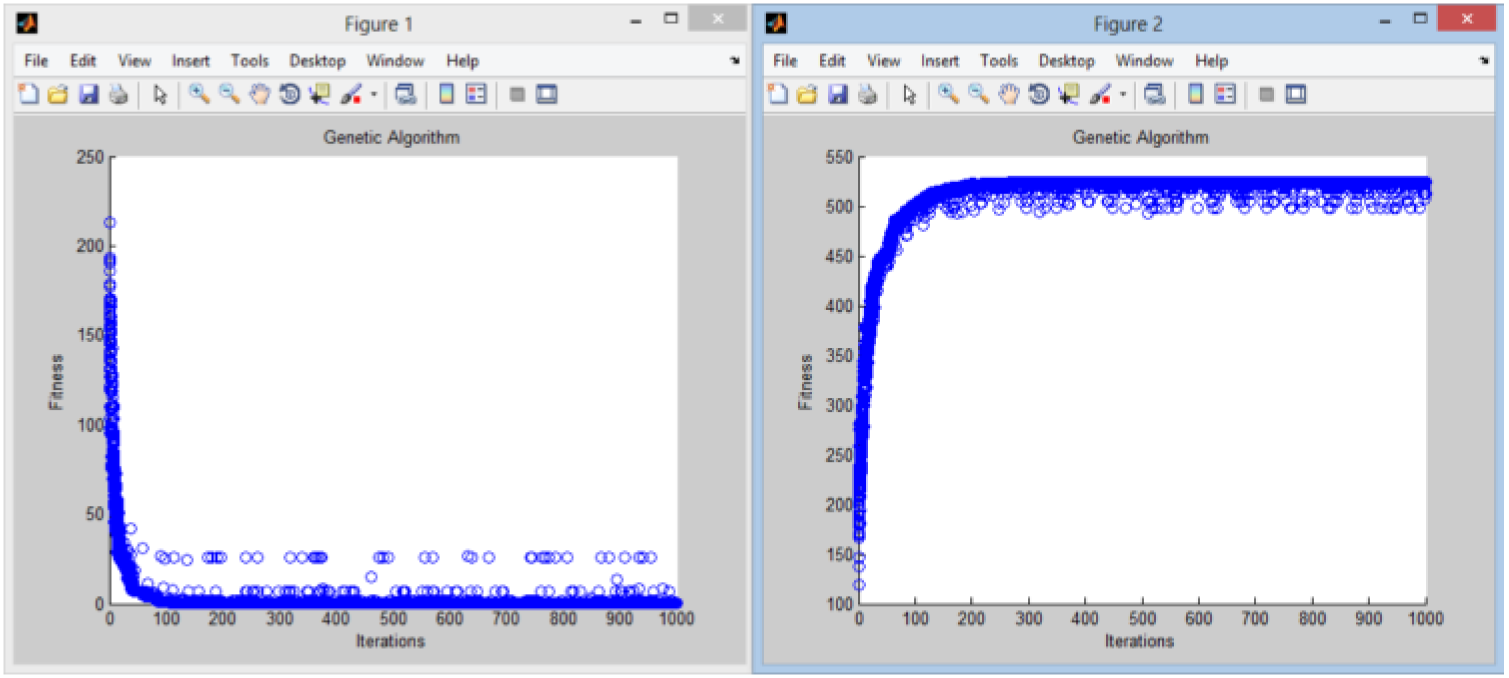


Figure B.8: Optimisation results of a genetic algorithm using binary tournament selection and two-point crossover implemented in Shark

B.2 Evolution Strategies

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

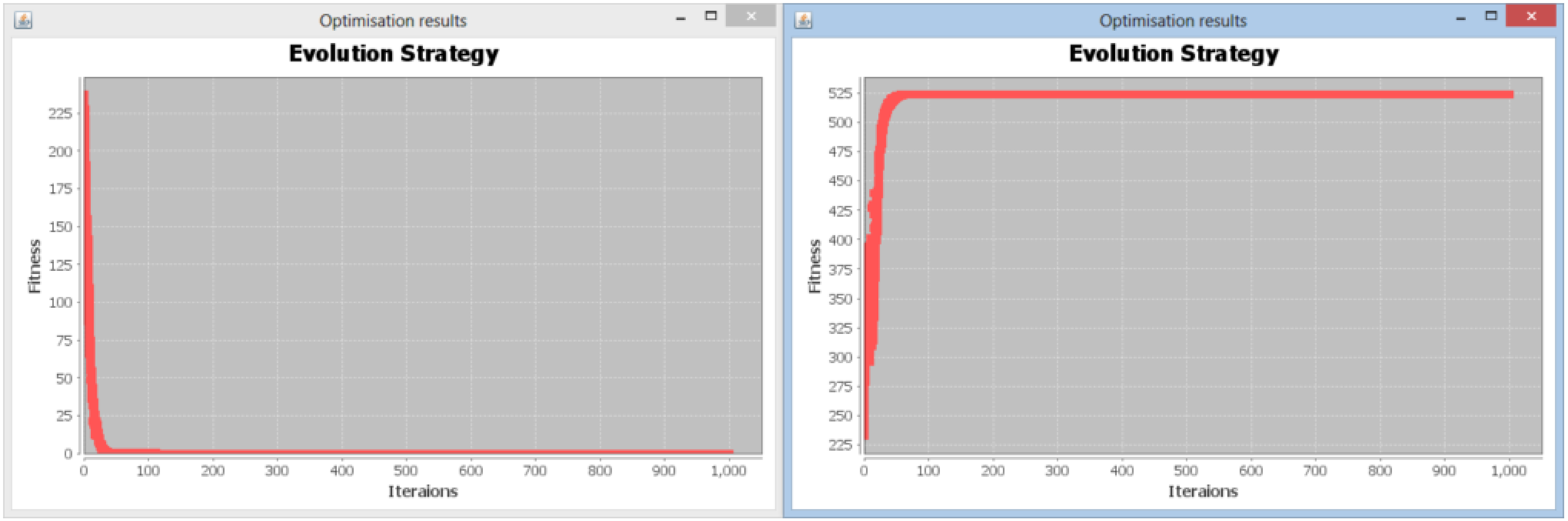


Figure B.9: Optimisation results of evolution strategies using (μ, λ) selection and discrete recombination

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

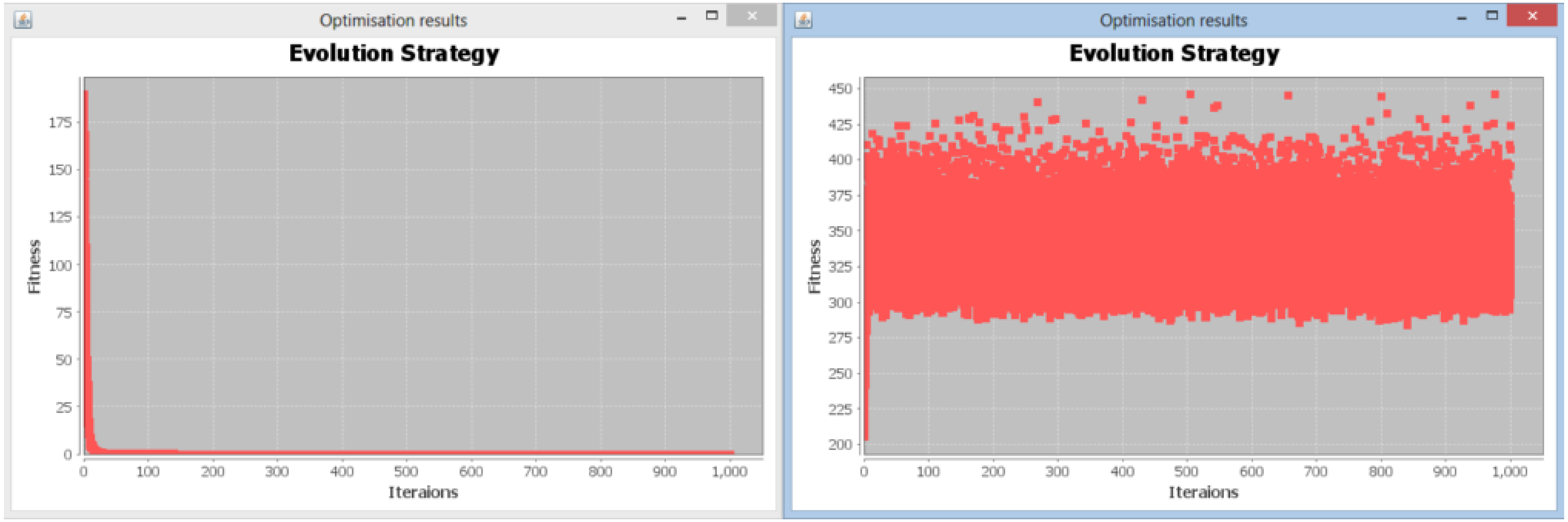
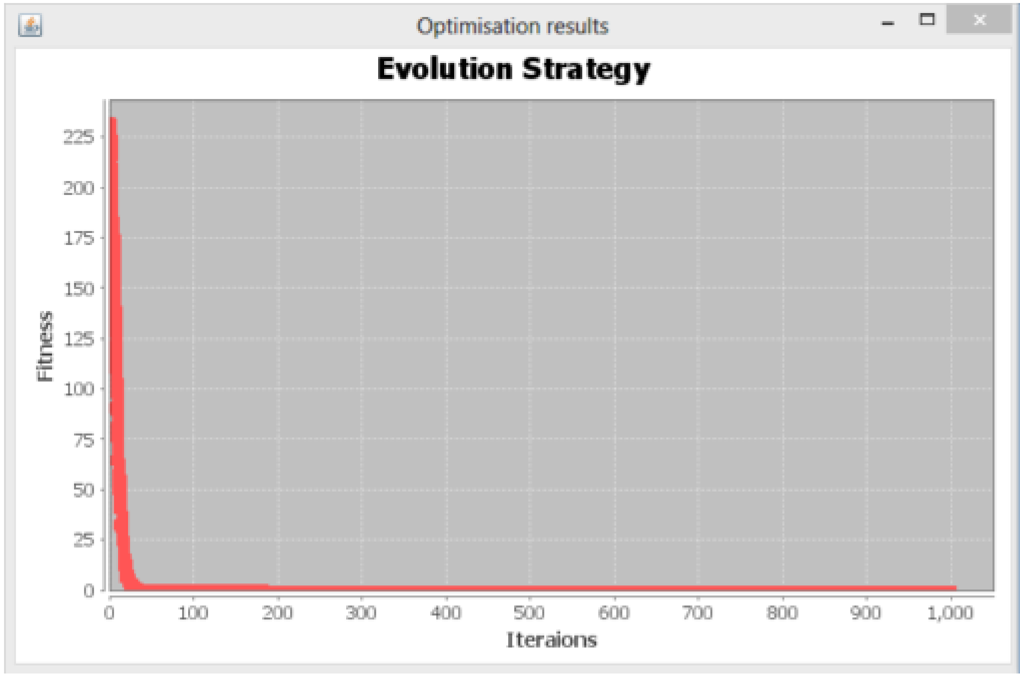


Figure B.10: Optimisation results of evolution strategies using (μ, λ) selection and intermediate recombination

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



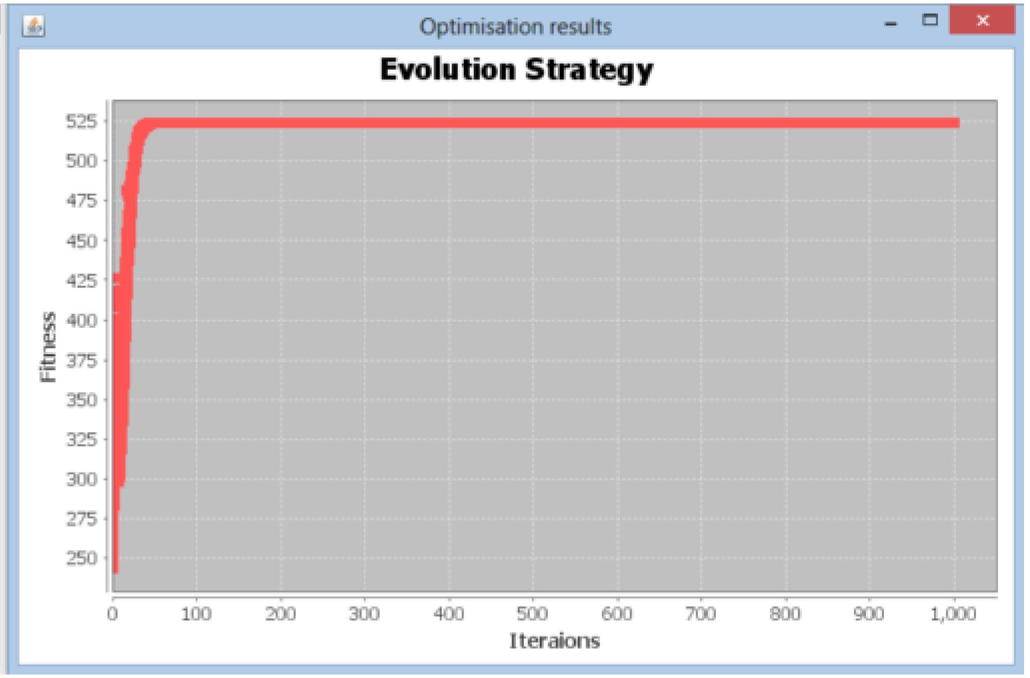


Figure B.11: Optimisation results of evolution strategies using (μ,λ) selection and global discrete recombination

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

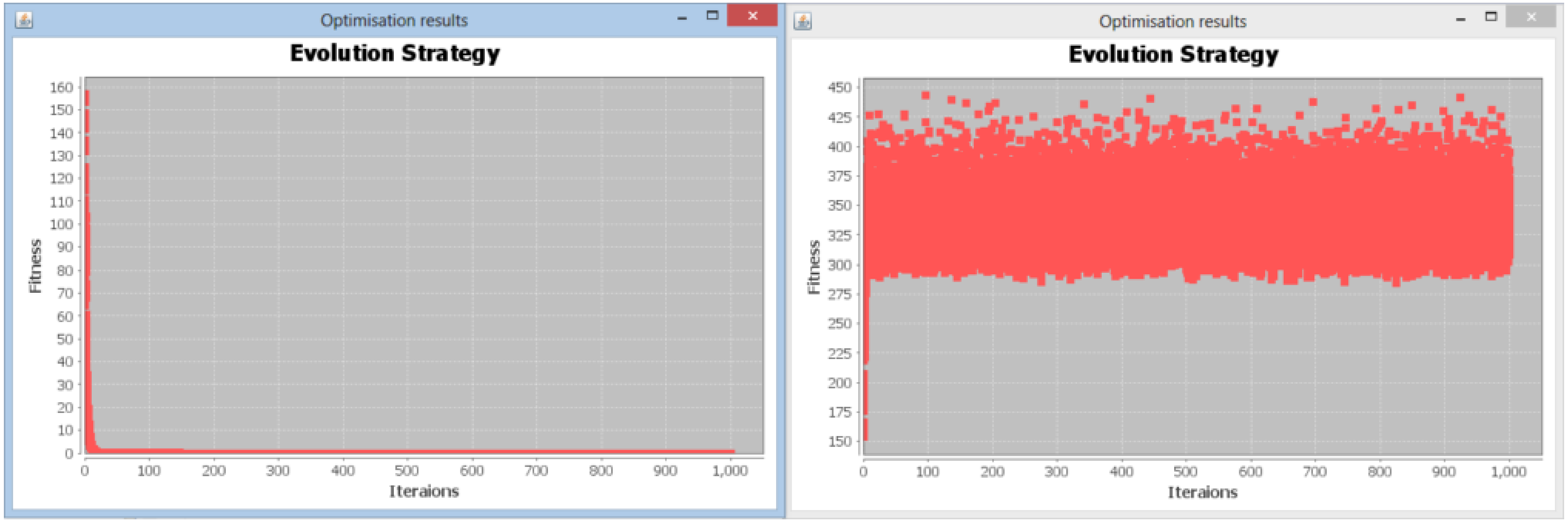


Figure B.12: Optimisation results of evolution strategies using (μ,λ) selection and global intermediate recombination

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

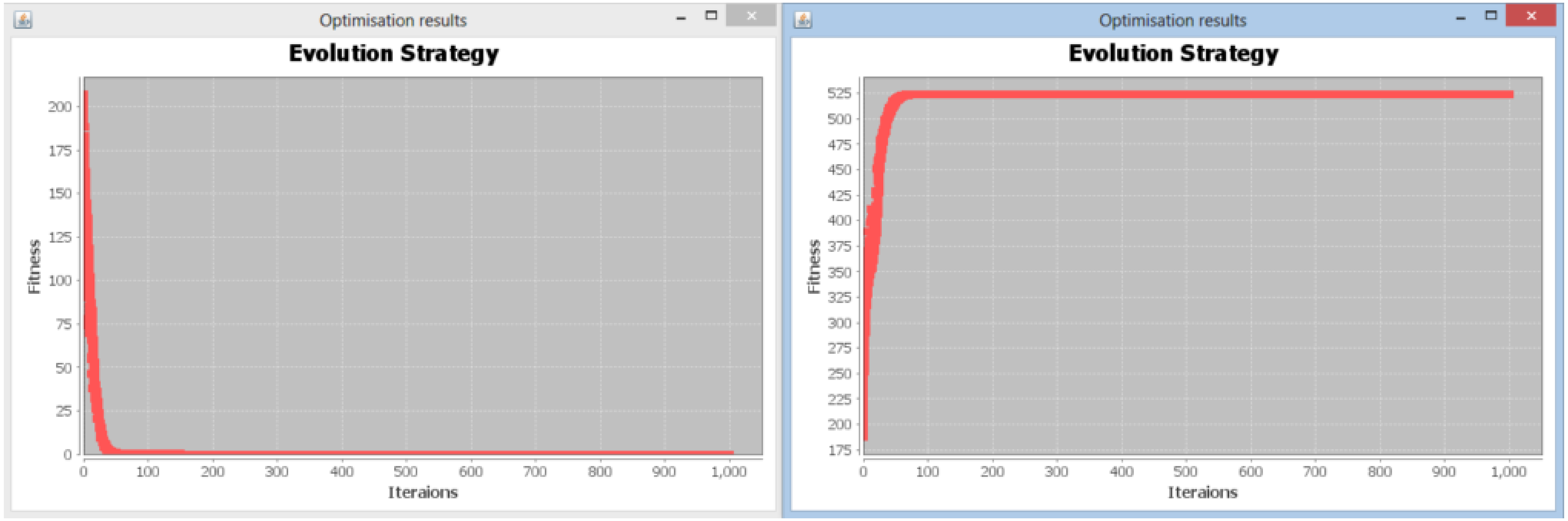


Figure B.13: Optimisation results of evolution strategies using $(\mu + \lambda)$ selection and discrete recombination

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

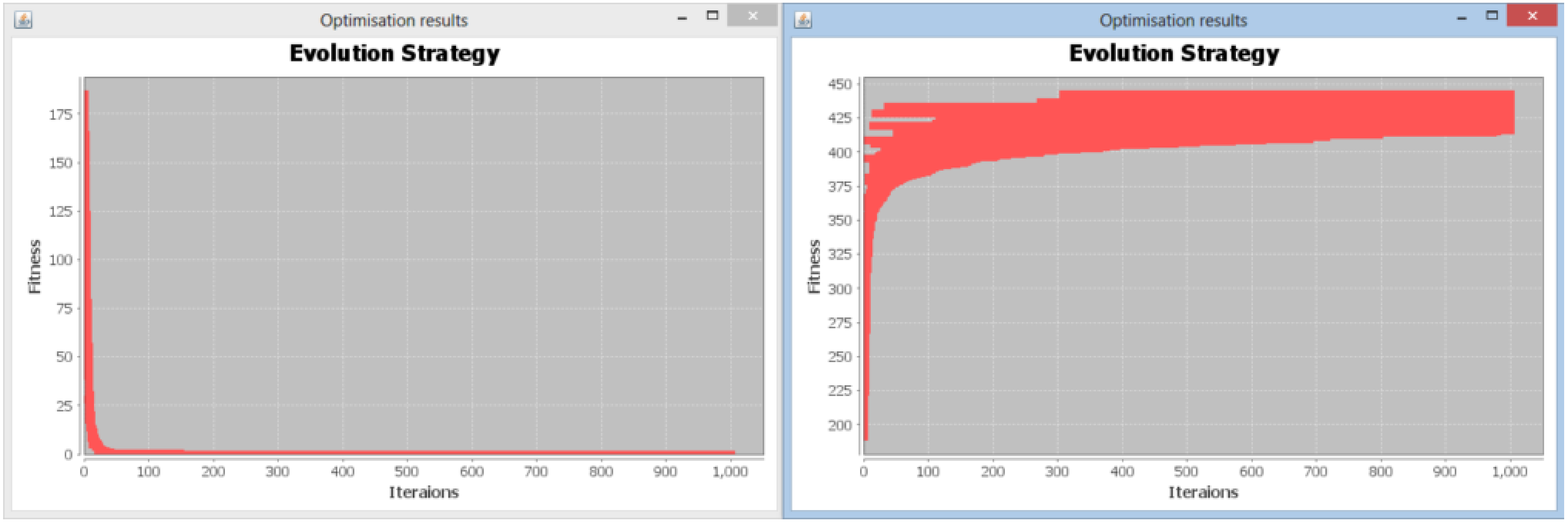


Figure B.14: Optimisation results of evolution strategies using $(\mu + \lambda)$ selection and intermediate recombination

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

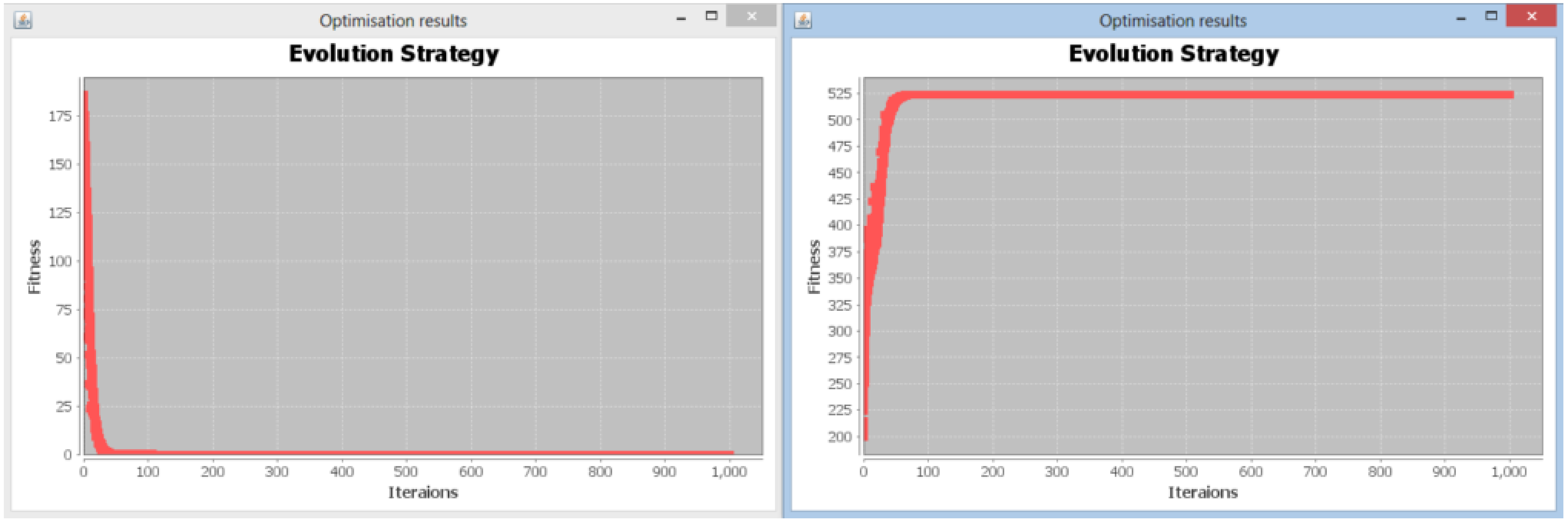


Figure B.15: Optimisation results of evolution strategies using $(\mu + \lambda)$ selection and global discrete recombination

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

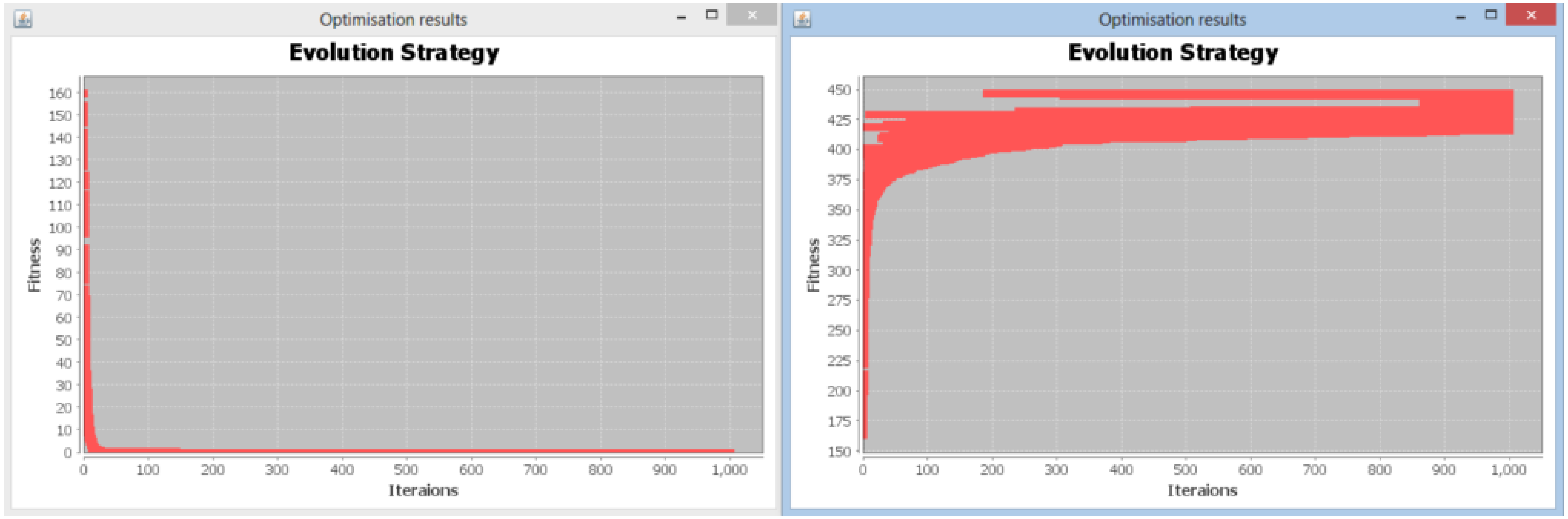


Figure B.16: Optimisation results of evolution strategies using $(\mu+\lambda)$ selection and global intermediate recombination

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

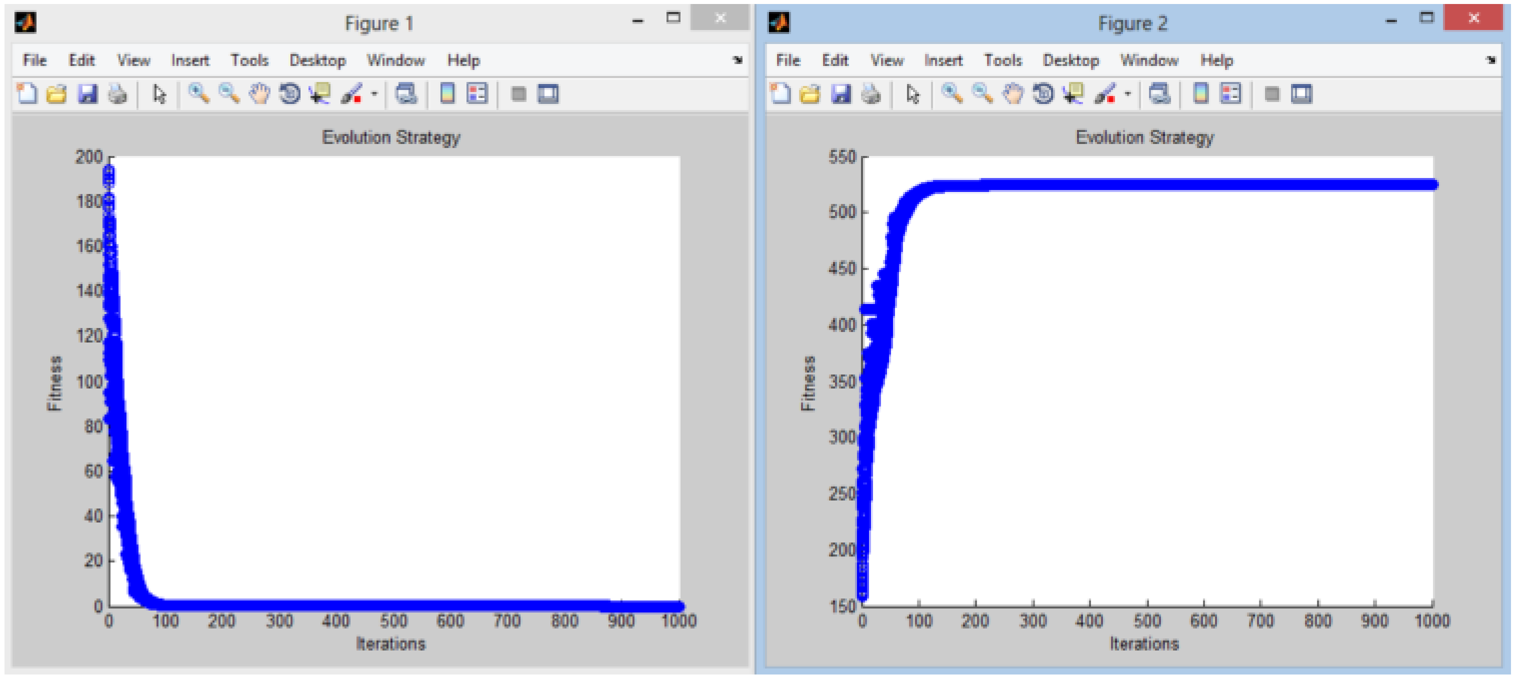


Figure B.17: Optimisation results of evolution strategies using $(\mu+\lambda)$ selection and discrete recombination implemented in Shark

B.3 Particle Swarm Optimisation

B.3.1 Optimisation Results

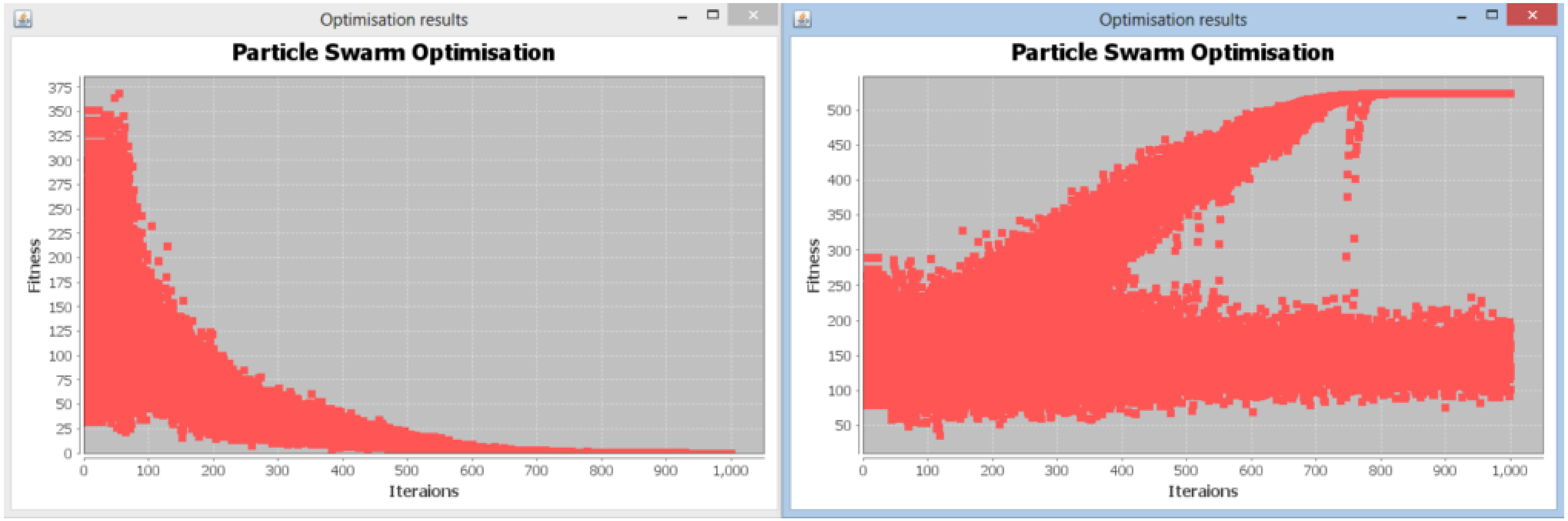


Figure B.18: Optimisation results of particle swarm optimisation

B.3.2 Optimisation Results in Shark

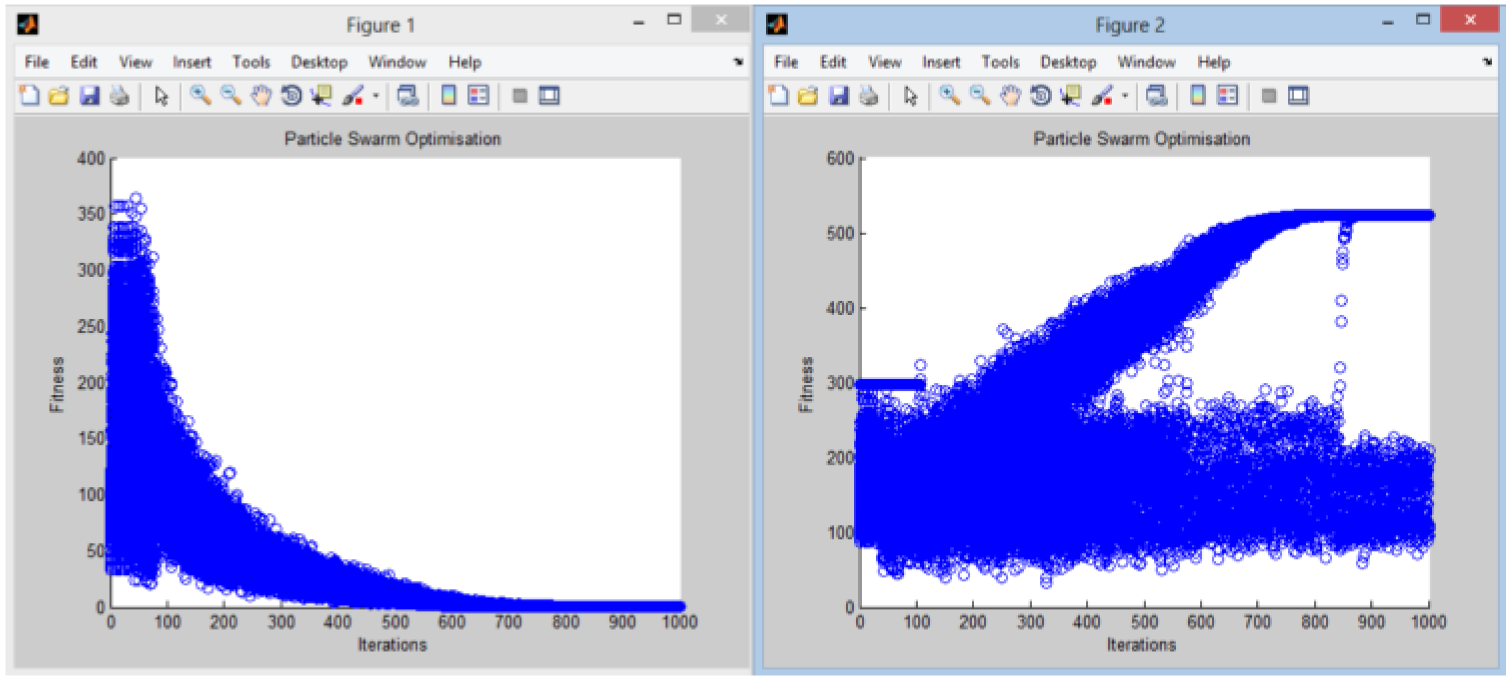


Figure B.19: Optimisation results of particle swarm optimisation implemented in Shark

References

Miguel, E et al. (2012). "Data-Driven Estimation of Cloth Simulation Models." In: *Computer Graphics Forum* 31.2, pp. 519–528.

Mongus, D. et al. (2012). "A hybrid evolutionary algorithm for tuning a cloth-simulation model." In: *Applied Soft Computing Journal*.

Ng, Hing N. and Richard L. Grimsdale (1996). "Computer Graphics Techniques for Modeling Cloth." In: *IEEE Computer Graphics & Applications* 16, pp. 28–42.

Provot, Xavier (1995). "Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behaviour." In: *Graphics Interface'95*, pp. 141–155.

Xinrong, Hu et al. (2009). "Review of cloth modeling." In: *2009 Second ISECS International Colloquium on Computing, Communication, Control, and Management, CCCM 2009*.

Yalçın, M Adil and Cansın Yıldız. *Techniques for Animating Cloth*. URL: `http://www.cs.bilkent.edu.tr/{~}cansin/projects/cs567-animation/cloth/cloth-paper.pdf` (visited on 05/25/2016).

Zhang, Dongliang and Matthew M F Yuen (2001). "Cloth simulation using multilevel meshes." In: *Computers and Graphics (Pergamon)*.

Bibliography

Akiya, Naoko, Scott Bury, and John M. Wassick (2011). "A General Model for Soft Body Simulation in Motion." In: *Winter Simulation Conference*. 2010, pp. 2194–2205.

Baraff, David and Andrew Witkin (1998). "Large Steps in Cloth Simulation." In: *SIGGRAPH*, pp. 43–54.

Bartels, Pieterjan (2014). *Simulation of Tearable Cloth*. URL: `https://nccastaff.bournemouth.ac.uk/jmacey/CGITech/reports2015/PBartels{\_}PieterjanBartels{\_}CGITECH{\_}final.pdf` (visited on 03/10/2016).

Bender, Jan, Daniel Weber, and Raphael Diziol (2013). "Fast and stable cloth simulation based on multi-resolution shape matching." In: *Computers and Graphics (Pergamon)*.

Bourg, David M. and Bryan Bywalec (2013). *Physics for Game Developers*. Second. Sebastopol: O'Reilly. ISBN: ISBN 9781449361051. URL: `https://www.dawsonera.com/abstract/9781449361051`.

Boxerman, Eddy (2003). "Speeding Up Cloth Simulation." Master's Thesis. The University of British Columbia.

Bridson, Robert, Ronald Fedkiw, and John Anderson (2002). "Robust Treatment of Collisions, Contact and Friction for Cloth Animation." In: *SIGGRAPH*, pp. 594–603.

Choi, Kwang-Jin and Hyeong-Seok Ko (2002). "Stable but Responsive Cloth." In: *SIGGRAPH*, pp. 604–611.

— (2005). "Research problems in clothing simulation." In: *Computer Aided Design* 37, pp. 585–592.

Conger, David (2004). *Physics Modeling for Game Programmers*. First. Boston: Course Technology / Cengage Learning. URL: `http://site.ebrary.com/lib/staffordshire/reader.action?docID=10065752`.

De Farias, Thiago S M C et al. (2008). "A high performance massively parallel approach for real time deformable body physics simulation." In: *Proceedings - Symposium on Computer Architecture and High Performance Computing*.

Desbrun, Mathieu, Peter Schröder, and Alan Barr (1999). "Interactive Animation of Structured Deformable Objects." In: *Graphics Interface*. San Francisco: Morgan Kaufmann Publishers Inc., pp. 1–8.

Enqvist, Henrik (2010). *The Secrets Of Cloth Simulation In Alan Wake*. URL: `http://www.gamasutra.com/view/feature/132771/the{\_}secrets{\_}of{\_}cloth{\_}simulation{\_}in{\_}.php?page=1` (visited on 03/03/2016).

Ertekin, Burak (2014). *Cloth Simulation*. URL: `https://nccastaff.bournemouth.ac.uk/jmacey/CGITech/reports2015/BErtekin{\_}burakertekin-cgitech.pdf` (visited on 03/10/2016).

Garre, Carlos and Alvaro Pérez (2008). *A simple Mass-Spring system for Character Animation*. URL: `http://www.gmrv.es/{~}cgarre/APO{\_}MassSpring{\_}Jun08.pdf` (visited on 05/25/2016).

Gibson, Sarah F. F. and Brian Mirtich (1997). *A Survey of Deformable Modeling in Computer Graphics*. Tech. rep. Cambridge: MERL.

Harada, Takahiro (2006). "Real-time Cloth Simulation Interacting with Deforming High-Resolution Models." In: *SIGGRAPH*.

— (2008). "Real-Time Rigid Body Simulation on GPUs." In: *GPU Gems 3*. Pearson Education Inc. Chap. 29. URL: `http://http.developer.nvidia.com/GPUGems3/gpugems3{\_}ch29.html`.

Jakobsen, Thomas (2005). *Advanced Character Physics*. URL: `http://www.gotoandplay.it/{\_}articles/2005/08/advCharPhysics.php` (visited on 03/03/2016).

Kang, Young-Min, Jeong-Hyeon Choi, and Hwan-Gue Cho (2000). "Fast and Stable Animation of Cloth with an Approximated Implicit Method." In: *Computer Graphics International*. Geneva.

Kim, Tae-Yong (2011). *Character Clothing in PhysX-3*.

Lander, Jeff (2000a). *Collision Response: Bouncy, Trouncy, Fun*. URL: `http://www.gamasutra.com/view/feature/3427/collision{\_}response{\_}bouncy{\_}.php` (visited on 03/24/2016).

— (2000b). *Devil in the Blue Faceted Dress: Real Time Cloth Animation*. URL: `http://www.gamasutra.com/view/feature/131851/devil{\_}in{\_}the{\_}blue{\_}faceted{\_}dress{\_}.php` (visited on 03/03/2016).

Magnenat-Thalmann, Nadia and D. Thalmann (2004). *Handbook of Virtual Humans*. 1st. Chichester: Wiley.

Mesit, Jaruwan, Ratan Guha, and Shafaq Chaudhry (2007). "3D soft body simulation using mass-spring system with internal pressure force and simplified implicit integration." In: *Journal of Computers*.

Mesit, Jaruwan and Ratan K. Guha (2008). "Soft Body Simulation with Leaking Effect." In: *2008 Second Asia International Conference on Modelling & Simulation (AMS)*. IEEE, pp. 390–395.

Miguel, E et al. (2012). “Data-Driven Estimation of Cloth Simulation Models.” In: *Computer Graphics Forum* 31.2, pp. 519–528.

Mongus, D. et al. (2012). “A hybrid evolutionary algorithm for tuning a cloth-simulation model.” In: *Applied Soft Computing Journal*.

Müller, Matthias et al. (2006). “Position Based Dynamics.” In: *VRIPHYS*, pp. 71–80.

Ng, Hing N. and Richard L. Grimsdale (1996). “Computer Graphics Techniques for Modeling Cloth.” In: *IEEE Computer Graphics & Applications* 16, pp. 28–42.

NVidia (2007). *Cloth Simulation*. URL: `http://developer.download.nvidia.com/SDK/10/direct3d/Source/Cloth/doc/Cloth.pdf` (visited on 05/25/2016).

O ’connor, Corey and Keith Stevens (2003). *Modeling Cloth Using Mass Spring Systems*. (Visited on 03/10/2016).

Ozgen, Oktar et al. (2010). “Underwater Cloth Simulation with Fractional Derivatives.” In: *ACM Trans. Graph* 29.23.

Parent, Rick (2012). *Computer Animation Algorithms and Techniques*. Third. Waltham: Elsevier. URL: `https://www.dawsonera.com/readonline/9780124159730/startPage/20`.

Plath, Jan (2000). “Realistic modelling of textiles using interacting particle systems.” In: *Computers & Graphics* 24, pp. 897–905.

Provot, Xavier (1995). “Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behaviour.” In: *Graphics Interface’95*, pp. 141–155.

Santos Souza, Marco, Aldo Von Wangenheim, and Eros Comunello (2014). “Fast Simulation of Cloth Tearing.” In: *SBC Journal on Interactive Systems* 5.1, pp. 44–48.

Schmitt, N et al. (2013). “Multilevel Cloth Simulation using GPU Surface Sampling.” In: *Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS*.

Selle, Andrew et al. (2009). “Robust High-Resolution Cloth Using Parallelism, History-Based Collisions and Accurate Friction.” In: *IEEE Transactions on Visualization and Computer Graphics* 15, pp. 339–350.

Tang, Min et al. (2013). “A GPU-based streaming algorithm for high-resolution cloth simulation.” In: *Computer Graphics Forum*.

Vassilev, T, B Spanlang, and Y Chrysanthou (2001). “Fast Cloth Animation on Walking Avatars.” In: *Eurographics*.

Vassilev, Tzvetomir and Bernhard Spanlang (2002). “A Mass-Spring Model For Real Time Deformable Solids.” In: *East-West Vision*.

Vassilev, Tzvetomir Ivanov (2010). “Comparison of Several Parallel API for Cloth Modelling On Modern GPUs.” In: *CompSysTech*, pp. 131–136.

— (2011). “Comparison of Parallel Algorithms for Modelling Mass-springs Systems with Several APIs on Modern GPUs.” In: *CompSysTech*, pp. 204–209.

Volino, Pascal, Frederic Cordier, and Nadia Magnenat-Thalmann (2005). “From early virtual garment simulation to interactive fashion design.” In: *Computer-Aided Design* 37.6, pp. 593–608.

Volino, Pascal, Martin Courchesne, and Nadia Magnenat-Thalmann (1995). “Versatile and Efficient Techniques for Simulating Cloth and Other Deformable Objects.” In: *SIGGRAPH*, pp. 137–144.

Volino, Pascal and Nadia Magnenat-Thalmann (1997a). “Developing Simulation Techniques for an Interactive Clothing System.” In: *International Conference on Virtual Systems and MultiMedia*. IEEE, pp. 109–118.

— (1997b). “Interactive Cloth Simulation: Problems and Solutions.” In: *JWS97-B*. Geneva.

— (2001). “Comparing Efficiency of Integration Methods.” In: *Conference on Computer Graphics International*.

Wacker, Markus, Bernhard Thomaszewski, and Michael Keckeisen (2005). “Physical Models, Numerical Solvers for Cloth Animation and Virtual Cloth Design.” In: *Eurographics*.

Wang, Jianchun, Xinrong Hu, and Yi Zhuang (2009). “The dynamic cloth simulation performance analysis based on the improved spring-mass model.” In: *International Conference on Wireless Networks and Information Systems, WNIS 2009*, pp. 282 –285.

Wang, Xiuzhong and Venkat Devarajan (2004). “2D Structured Mass-spring System Parameter Optimization based on Axisymmetric Bending for Rigid Cloth Simulation.” In: *SIGGRAPH*, pp. 317–323.

Xinrong, Hu et al. (2009). “Review of cloth modeling.” In: *2009 Second ISECS International Colloquium on Computing, Communication, Control, and Management, CCCM 2009*.

Yalçın, M Adil and Cansın Yıldız. *Techniques for Animating Cloth.* URL: `http://www.cs.bilkent.edu.tr/{~}cansin/projects/cs567-animation/cloth/cloth-paper.pdf` (visited on 05/25/2016).

Zeller, Cyril (2005). “Cloth Simulation On The GPU.” In: *SIGGRAPH*.

Zhang, Dongliang and Matthew M F Yuen (2001). “Cloth simulation using multilevel meshes.” In: *Computers and Graphics (Pergamon)*.

Zhenfang, Cao and He Bing (2012). “Research of Fast Cloth Simulation Based on Mass- Spring Model.” In: *National Conference on Information Technology and Computer Science*, pp. 323–327.

Zink, Nico and Alexandre Hardy (2007). “Cloth Simulation and Collision Detection using Geometry Images.” In: *AFRIGRAPH*. New York: ACM, pp. 187–195.