Development of a Framework for Genetic Algorithms

Håkan Wååg

Examination Degree 2009

Computer Science
This thesis is performed at Tekniska Högskolan in Jönköping in the area of Computer science. The work is part of the three-year long Bachelor of Science in Engineering. The author is himself responsible for views, conclusions and results.

Instructor: Ragnar Nohre

Scope: 15 points

Date: 2009-10-12

Archiving number:
Abstract

Genetic algorithms is a method of optimization that can be used to solve many different kinds of problems. This thesis focuses on developing a framework for genetic algorithms that is capable of solving at least the two problems explored in the work. Other problems are supported by allowing user-made extensions.

The purpose of this thesis is to explore the possibilities of genetic algorithms for optimization problems and artificial intelligence applications.

To test the framework two applications are developed that look at two distinct problems, both of which aim at demonstrating different parts. The first problem is the so called Travelling Salesman Problem. The second problem is a kind of artificial life simulator, where two groups of creatures, designated predator and prey, are trying to survive.

The application for the Travelling Salesman Problem measures the performance of the framework by solving such problems using different settings. The creature simulator on the other hand is a practical application of a different aspect of the framework, where the results are compared against predefined data. The purpose is to see whether the framework can be used to create useful data for the creatures.

The work showed how important a detailed design is. When the work began on the demonstration applications, things were noticed that needed changing inside the framework. This led to redesigning parts of the framework to support the missing details. A conclusion from this is that being more thorough in the planning, and considering the possible use cases could have helped avoid this situation.

The results from the simulations showed that the framework is capable of solving the specified problems, but the performance is not the best. The framework can be used to solve arbitrary problems by user-created extensions quite easily.

Keywords

Genetic algorithms, optimization, c++, Travelling Salesman Problem, artificial intelligence, framework, library
Sammanfattning


Syftet med detta examensarbete är att utforska möjligheterna med genetiska algoritmer för optimeringsproblem och tillämpningar på artificiell intelligens.


Applikationen för handelsresandeproblemet mäter prestandan på ramverket genom att lösa såna problem med olika inställningar. Varelsesimulatortorn är en praktisk applikation av en annan del av ramverket, där resultatet jämförs med fördefinierad data. Syftet är att se om ramverket klarar av att skapa användbar data åt varelserna.

Arbetet har visat hur viktigt en detaljerad design är. När arbetet började på demonstrationsapplikationerna, upptäcktes dock behövde förändras inuti ramverket. Detta ledde till att ramverket omdesignades för att stödja de saknade detaljerna. En slutsats från detta är att en noggrann planering och att beakta de möjliga användningsfallen kan hjälpa till att undvika sådana situationer.

Resultaten från simulationerna visade att ramverket klarar av att lösa de specificerade problemen, men att prestandan inte är den bästa. Ramverket klarar av att lösa godtyckliga problem genom användarutvecklade tillägg.

Nyckelord

Genetiska algoritmer, optimering, c++, Handelsresandeproblemet, artificiell intelligens, ramverk, bibliotek
# Table of Contents

## 1 Introduction ..................................................................................8

1.1 Background........................................................................................................9
1.2 Purpose and goals................................................................................................10
1.3 Delimitations.......................................................................................................10
1.4 Outline.............................................................................................................11

## 2 Theoretical foundation .................................................................12

2.1 Optimization.................................................................................................12
   2.1.1 Introduction....................................................................................................12
   2.1.2 Terminology..................................................................................................13
2.2 Genetic Algorithms.......................................................................................14
   2.2.1 Biological background................................................................................14
   2.2.2 Introduction to the genetic algorithm.........................................................17
   2.2.3 Problem representation............................................................................19
   2.2.4 Genetic operators.......................................................................................20
   2.2.5 Diversity......................................................................................................28
   2.2.6 Termination criteria...................................................................................30
2.3 Decision Trees...............................................................................................32
   2.3.1 Overview.....................................................................................................32
   2.3.2 Learning decision trees.............................................................................32
   2.3.3 Decision trees for behaviour....................................................................32

## 3 Realization .........................................................................................35

3.1 The framework...............................................................................................35
   3.1.1 Overview.....................................................................................................35
   3.1.2 Design..........................................................................................................36
   3.1.3 Chromosome types....................................................................................38
   3.1.4 Testing.........................................................................................................39
3.2 The Travelling Salesman Problem...............................................................41
   3.2.1 Introduction....................................................................................................41
   3.2.2 The circle tour.............................................................................................42
   3.2.3 Western Sahara – 29 cities.........................................................................43
   3.2.4 The application...........................................................................................44
3.3 The Predator and Prey Application...............................................................47
   3.3.1 Overview.....................................................................................................47
   3.3.2 Behaviour of creatures.............................................................................47
   3.3.3 Simulation....................................................................................................49
   3.3.4 Genetic algorithm.......................................................................................51

## 4 Results ...............................................................................................52

4.1 The Travelling Salesman Problem...............................................................52
   4.1.1 The circle tour.............................................................................................52
   4.1.2 Western Sahara...........................................................................................56
4.2 Predator and Prey.........................................................................................58

## 5 Discussion and conclusion ............................................................60

5.1 Discussion.......................................................................................................60
5.2 Conclusion.......................................................................................................62
5.3 The future of the framework.......................................................................62

## 6 References .........................................................................................63
Table Index
Table 1: Circle tour - GA parameters.................................................................................42
Table 2: Western Sahara tour - GA parameters.................................................................43
Table 3: Prey parameter checks..........................................................................................48
Table 4: Prey actions..............................................................................................................48
Table 5: Predator actions......................................................................................................49
Table 6: Predator parameter checks.....................................................................................49
Table 7: Parameters for the creatures..................................................................................51

Figure Index
Figure 1: Global minimum for a function..............................................................................13
Figure 2: Chromosome with a gene and it's alleles..............................................................14
Figure 3: Two parent chromosomes and their child..............................................................14
Figure 4: Natural selection in three stages with parents and children.................................16
Figure 5: Flowchart of a genetic algorithms process...........................................................17
Figure 6: Selection of mates and crossover[12]....................................................................18
Figure 7: Example bit-string.................................................................................................19
Figure 8: Roulette Wheel selection – individuals' slice of pie..............................................20
Figure 9: One point crossover.............................................................................................21
Figure 10: Two point crossover............................................................................................21
Figure 11: Uniform crossover...............................................................................................21
Figure 12: Genes as nodes in a graph....................................................................................23
Figure 13: Parents for edge recombination..........................................................................24
Figure 14: Initial adjacency matrix........................................................................................24
Figure 15: Edge recombination offspring............................................................................25
Figure 16: Tree chromosome parents...................................................................................26
Figure 17: Tree chromosome crossover offspring...............................................................26
Figure 18: Tree mutation - replacing a node........................................................................27
Figure 19: Tree mutation - growth........................................................................................27
Figure 20: Tree mutation – removal of node.........................................................................28
Figure 21: Rank scalings effect on population.....................................................................29
Figure 22: Sigma scaling formula.........................................................................................29
Figure 23: Sigma scalings effect on population....................................................................30
Figure 24: Decision tree for playing tennis[5]......................................................................32
Figure 25: Example tree of behaviour[8].............................................................................33
Figure 26: Basic UML model of gafw...................................................................................36
Figure 27: Base classes and provided implementations.......................................................37
Figure 28: Chromosomtypes and base class.........................................................................38
Figure 29: gafw classes........................................................................................................38
Figure 30: Euclidean distance.............................................................................................41
Figure 31: Optimal circle tour..............................................................................................42
Figure 32: Western Sahara optimal tour..............................................................................44
Figure 33: The TSP application............................................................................................45
Figure 34: Three stages of a circle tour, getting closer to the optimal solution......................46
Figure 35: Standard tree for predators................................................................................50
Figure 36: Standard tree for prey........................................................................................50
Figure 37: No scaling, cycle crossover - generations...........................................52
Figure 38: No scaling, cycle crossover - time......................................................53
Figure 39: No scaling, edge recombination - generations.................................53
Figure 40: No scaling, edge recombination - time..............................................53
Figure 41: Rank scaling, cycle crossover - generations..................................54
Figure 42: Rank scaling, cycle crossover - time..................................................54
Figure 43: Rank scaling, edge recombination - generations.............................54
Figure 44: Rank scaling, edge recombination - time.........................................54
Figure 45: Sigma scaling, cycle crossover - generations.................................55
Figure 46: Sigma scaling, cycle crossover - time...............................................55
Figure 47: Sigma scaling, edge recombination - generations..........................55
Figure 48: Sigma scaling, edge recombination - time......................................55
Figure 49: Western Sahara tour, cycle crossover - time..................................56
Figure 50: Western Sahara tour, cycle crossover - average tour length.............56
Figure 51: Western Sahara, edge recombination - generations.......................57
Figure 52: Western Sahara, edge recombination - time....................................57
Figure 53: Western Sahara, edge recombination - average tour length.............57
Figure 54: Most successful tree for predators after evolution.........................58
Figure 55: Most successful tree for prey after evolution.................................58
1 Introduction

Optimization is the process of finding an optimal solution to a problem. There are many different kinds of algorithms that fall under this category, and this thesis will focus on genetic algorithms.

Genetic algorithms (GA) are based on the principles of evolution, where a population of individuals evolve over time. Their goal is to maximize the “fitness” which is an arbitrary measurement of success. What makes genetic algorithms interesting, is that they are good at finding solutions to problems where general optimization algorithms are unable to work effectively.

Take a problem where an unknown amount of suboptimal solutions exists. Getting past those local optimum points requires a different approach compared to many simple optimization algorithms. This is a perfect example of a problem where genetic algorithms shine. The result from the GA would be an evolved population that consist of the global optimal solution. Likely, it would also have many of the suboptimal solutions that are very close to the optimal solution.

This thesis will present a framework implementing genetic algorithms, which will be aptly named gafw (genetic algorithms framework). It is meant to be used from other applications as a library. Two applications using the framework will also be developed. They are two very different applications, demonstrating different aspects of the framework. One solves Travelling Salesman Problems while the other simulates survival with predators and prey. The two groups of creatures have a predefined set of behaviour that is compared against a set of behaviour the framework has created.
1.1 Background

The choice of subject for this thesis comes out of my interest for artificial intelligence (AI) and problem solving. AI is a very wide area to study, and holds many interesting subjects. The development of an artificial life simulator is one topic that inspired my work. This is interesting in many ways, where one is from a pure AI experimentation point of view, but something else that drives me to explore this further are the possible adaptations to computer games.

Artificial life simulators can be developed using many different kinds of techniques, with the goal being to create lifelike creatures. The choice of algorithm depends on how the simulation is supposed to play out. If evolution of the creatures is supposed to take place a genetic algorithm is a good choice.

Genetic algorithms have been used on a wide variety of optimization problems, game theory problems like “The Prisoner's Dilemma”\cite{11} is an example of its use.

My original idea was to develop a simulator similar to this, where I could incorporate AI to create creatures that compete for survival. Also, I would look into the possibility of cooperation between creatures to take down prey. Some form of learning mechanism was supposed to work in the background so that the creatures would become better with time. My first approach to this was to use a so called neural network\cite{5} for the creatures “intelligence”, but combined with the abstract problem it would have been hard to analyze properly.

After discussing the project with my instructor, we agreed to change the focus of the thesis toward the development of a general purpose genetic algorithms framework foremost, with the simulator as an application using the framework. Genetic algorithms provide a different approach to learning, by slowly creating better solutions to a problem, which will provide a different approach to the simulator.
1.2 Purpose and goals

The goal is to develop a general purpose framework for solving optimization problems using genetic algorithms. The intention is to build it so that other applications can use it as a library providing this functionality. Two demonstration applications will be built that utilize the framework.

A wide variety of problems should be possible to solve using this framework, but for the purposes of this thesis, I will focus on two such problems. They are the Travelling Salesman Problem (TSP) and an abstract problem involving simulation of creatures struggling for survival.

The TSP is straightforward, the goal is to find the shortest path through a set of nodes while visiting every node only once. For the purposes of this thesis it is sufficient if the framework is capable of solving problems involving relatively small amounts of cities. Time is the limiting factor when the amount of nodes increase.

The creature simulator will involve two groups of creatures that take on the role of predator and prey respectively. The events that take place during simulation will be visualized graphically in a window. This application's goal is to explore the functionality of parts of the framework, and to attempt to evolve populations of predator and prey with behaviour that is similar to a predefined population.

My purpose with the thesis is to expand my knowledge in the fields of artificial intelligence and problem solving by developing a genetic algorithms framework and practically using it to solve two problems.

1.3 Delimitations

Genetic algorithms may use different types of so called chromosomes. To limit the scope of the project, only the chromosome types necessary to solve the intended applications will be built into the framework.
1.4 Outline

The report begins with a thorough theoretical section to inform the reader of the different technologies talked about. The topics explained in this section are:

- Optimization
- Genetic algorithms
- Decision trees

Next up is the realization section of the report, which describes the development process of the GA framework and what design decisions were made. The test units used to verify the functionality of the framework are described.

The two applications using the framework are presented, with some general information on their functionality. Their incorporation of gafw is explained, with the problem representation used and problem-specific data necessary for the experiments.

The results from the experiments done on the two applications are presented in the results section.

To conclude the report there is a section where my thoughts on the project and the results are discussed. I also consider what future development potential the framework has.
2 Theoretical foundation

2.1 Optimization

2.1.1 Introduction

To make something better is the fundamental principal in optimization. A set of input data is put through an optimization algorithm and returns data that is better.

There are numerous algorithms that can perform optimization. Which algorithm to use is largely dependent on what kind of problem it has to optimize, and under what conditions it has to work.

If finding the global optimum isn't important, a hill-climbing algorithm can easily find the first local optimum. Under certain conditions it may also be able to move past some local optimum. On the contrary, if the global optimum is necessary, the hill-climbing algorithm is a poor choice, because it's not designed to traverse past local optimum without modifications.

Often it is not possible to determine whether the algorithm has converged on a local or global optimum. If the algorithm converges on a local optimum this is called premature convergence. Methods of avoiding this differ between algorithms. One approach is restarting the algorithm from a random point in time[1]. Other methods to avoid the problem exist in evolutionary algorithms, which involve genetic algorithms among others.

Another factor that impacts the choice of optimization algorithm is whether the calculations can be done offline or if they have to be done online. Offline algorithms can take all the time they want to solve the problem, the only thing important is the finding of an optimal solution. An online algorithm on the other hand has time constraints imparted on it[1], a solution is often required within milliseconds or minutes depending on the problem.

Figure 1 illustrates a function with multiple locally optimal solutions, and one globally optimal solution, which in this case is the global minimum.
2.1.2 Terminology

For all optimization algorithms, there are a few common terms which are useful to know.

- Problem space
- Search space

For a problem, the set of data that can theoretically be a solution, is called a problem space. Inside the problem space, the set of data that the algorithm can operate on is called the search space. The search space can be very small in algorithms that are prone to premature convergence.
2.2 Genetic Algorithms

2.2.1 Biological background

Every living organism is made up of cells. Each cell in an organism contains the same chromosome set, more known as DNA. A chromosome is built up of a set of genes. Every gene is responsible for a certain trait of an organism, such as eye color. The different values that a gene can take are called alleles\(^\text{[11]}\).

Think of the line in Figure 2 as a chromosome. The boxes numbered 1 through 5 are genes inside this chromosome. Each gene represents some trait for the organism and gene number 5 is responsible for it's eye color. The boxes in the same column, blue and brown are gene 5's alleles.

![Figure 2: Chromosome with a gene and it's alleles](image)

An organism can have multiple chromosomes in each cell, and many organisms do. All chromosomes in a cell are together called a genome.

A species is a population of organisms, with similar genomes. To keep the species alive, most organisms use sexual reproduction to create new members of the population. This offspring will have a genome similar to it's typically two parents. The process of how to determine this genome is done in multiple steps. The most prominent effect on the offspring's genome is recombination, or crossover. Crossover takes pieces of both parent's genome and combines them to create a new genome with traits from both parents.

![Figure 3: Two parent chromosomes and their child](image)
Figure 3 illustrates the concept of crossover. The line with boxes represent that parents' genome. The child receives half the chromosomes from parent 1 and the other half from parent 2.

Besides crossover, offspring are also subject to mutation. This is a random occurrence where certain genes may become something completely different from the parents' genes. Mutation may occur as a result of a copying error.[11]

In nature, an organism's fitness is typically a measure of the probability that the organism lives to reproduce, or alternatively based on how many offspring the organism has.

Evolution is the term given to the process in nature where populations of organisms change over time. A new generation of the organisms is made up of offspring from the previous generation, and subject to crossover and mutation which may alter the organism's DNA. The organisms' that are better suited for survival, and thus have a larger chance of reproducing, are a result of what is called natural selection. A population may become more successful with every generation as a result of this. Fitness is the decisive measurement of natural selection – survival of the fittest.

Figure 4 shows in three stages how a population of organisms will become fitter as a result of natural selection. The first chart illustrates a starting population, which is considered the original population for the example. The organisms in this population become parents, where some of them eventually have children. The middle chart shows this situation, with overlapping parents and children in the population.

Some of the parents, as well as some of the children may die, because of predators or accidents. Generally, the fitter organisms will be the one that manage to overcome these obstacles, and live on as the population in the last chart.

Diversity can refer to the number and variety of members of a population as well as between species. The diversity in a population can be seen in the charts by looking at how many different fitness levels are represented. A small diversity generally means that the population or species is highly specialized for some environment or task, whereas high diversity means they are adaptable.
Charles Darwin is the man who came up with the idea of evolution\cite{10}, presenting evidence for the theory. That evolution occurs was accepted early on, but it was only after his death that natural selection got widespread acceptance.
2.2.2 Introduction to the genetic algorithm

Genetic algorithms work by the same principles as evolution. For a problem, potential solutions to it are stored within chromosome-like data structures\(^{[12]}\). Organisms, or as they will be called henceforth, individuals, have genomes that may consist of multiple chromosomes, depending on the problem.

A search space of potential solutions is modelled as a population of individuals. The size of the population is typically predefined. Reproduction is simulated by taking one or more individuals from the population and applying crossover and mutation.

Genetic algorithms work by traversing the search space for a given problem, maximizing or minimizing the fitness values. The Schema theorem is an attempt to prove that this process leads to viable solutions\(^{[12]}\).

*Figure 5* shows the process that a genetic algorithm typically goes through. There is no set definition of what a genetic algorithm should include, but these steps are the minimum for any genetic algorithm\(^{[11]}\).

*Figure 5: Flowchart of a genetic algorithms process*
The initial population is normally randomly generated based on the types of chromosomes used. If a so called indirect representation is used for chromosomes, the chromosome is decoded for the next step.

Fitness calculation is a problem specific operation that evaluates the genes in every chromosome and assigns a value that determines this individual's fitness. Based on these fitness values, a selection algorithm will be used to choose parents for the crossover operation.

Crossover and mutation are used to change the search space by combining and modifying the individuals in the current population. Selection along with crossover and mutation are known as genetic operators. At this point a new generation of individuals will exist, created by the genetic operators. The procedure of selection and crossover that results in offspring is illustrated in Figure 6. The parentheses on the offspring show which individuals were parents for this offspring. The method of selecting individuals for mating varies with different implementations of genetic algorithms.

![Figure 6: Selection of mates and crossover](image)

The final step is to evaluate the population based on a termination criteria, to see if it is time to stop the algorithm. If the criteria is not met, the algorithm starts over by decoding the new generations' chromosomes. Three criteria used are termination by number of generations executed, termination by detecting fitness convergence and termination by reaching a specific fitness value.

A concept that is frequently used is called elitism. This is is a means of automatically transferring the fittest individuals to the next generation. How many that are transferred is specified by a fixed number or a percentage of the population. This usually improves the performance of genetic algorithms significantly[11].
A genetic algorithm typically takes a few parameters that define its behaviour. Crossover rate and mutation rate are the most common parameters. Crossover rate is the probability that crossover happens between two individuals. What happens instead of crossover is typically user-defined. One possibility includes transferring the individual to the new generation, in effect becoming a form of selected elitism. Another way of dealing with the situation is to simply exclude the individual from crossover, letting it have no children.

The effect mutation rate has can vary between the different types of problem representations. In general it defines the probability that mutation will occur in a chromosome or gene.

### 2.2.3 Problem representation

To use a genetic algorithm to solve a problem, it has to be represented somehow as a genome. Traditionally, genetic algorithms use what is called bit-strings. This is a string consisting of the alleles 0 and 1. Figure 7 shows how a bit-string may look.

```
101100110001010011101
```

*Figure 7: Example bit-string*

Bit-strings are the most common chromosome types in genetic algorithms applications. Using this chromosome requires encoding the problem in binary, either directly or for example via gray coding. This chromosome is an indirect representation.

Different problems have different requirements, thus other, direct representations have been devised. Real-valued representations use chromosomes that directly contain integers, floating point numbers or whatever the specific problem may need. Tree structured chromosomes are also used, but they are more common in genetic programming. Trees have the advantage that if the tree grows, so does the search space. This could lead to unexpected solutions, but if care is not taken, the trees may grow too big.

An interesting effect of real-valued representations is that they may provide better performance than their binary counterpart.

Permutation problems are an example where real-valued representation is the most suitable. What kind of real-valued representation is used doesn't matter, but the order of the genes in the chromosome does. This is what makes up the solution in a permutation chromosome.
2.2.4 Genetic operators

Selection is one of the most important genetic operators in a genetic algorithm. It is responsible for choosing which individuals will be paired for mating, and therefore has the option of making intelligent match ups to create better offspring.

The easiest method would be to simply iterate through the population and pair them as they come, ignoring individuals with fitness below a threshold value. This method is called truncation selection. Less fit individuals may still have certain genes that are beneficial in combination with others, but this combination may never take place using this method of selection. As such, using this method of selection for anything but the most trivial problems may result in quick convergence on suboptimal solutions.

A frequently used selection operator is the Roulette Wheel. This method assigns each individual a slice sized according to their fitness. Parents are chosen randomly, with preference toward fitter individuals, but even the most unfit individual has a chance of being selected. Figure 8 shows a population of 10 individuals, with the tenth individual being the fittest, and thus getting the largest slice.

![Figure 8: Roulette Wheel selection – individuals' slice of pie](image)

The genetic operators crossover and mutation perform differently depending on the type of representation used for the solutions.

2.2.4.1 Bit-strings

There are many variations to this chromosome. Three types of crossover operators will be described here, one point, two point and uniform crossover. These are the three most common operators.
One point crossover is the simplest possible operator. Choose a random point in the bit-string to use as a divider, and use the first half from parent one and combine with the second half of parent two. Figure 9 illustrates the procedure.

![Figure 9: One point crossover](image)

Two point crossover works similar to one point crossover, using two random points on the parent's strings. The bits between the two chosen points are swapped, resulting in two offspring. Figure 10 is an illustration of two point crossover.

![Figure 10: Two point crossover](image)

Uniform crossover is different, in that it iterates over every bit in the string, and swaps the bit based on a predefined probability. With a chromosome of length of 15 and uniform crossover probability of 0.5, Figure 11 is an example of how it could look.

![Figure 11: Uniform crossover](image)

Mutation of bit-strings means flipping the bits in the string. Based on a probability p, every bit in the string is tested and randomly flipped.
2.2.4.2 Permutation

For permutation chromosomes, every value in the chromosome can only exist once, and the order is what makes up the solution. This requires special care during crossover so that repetition of genes are not introduced. The two algorithms used in this thesis are cycle crossover and edge recombination.

Cycle crossover is not unlike the bit-string crossover operators, a method of randomly traversing the search space while avoiding repetition. The procedure begins with swapping one gene from both chromosomes with each other. Which gene to begin with can be randomly chosen, or a simpler version would be to always use the first gene. This will cause a duplicate to exist in both parents unless an equal gene was positioned as the first in both parents. Locate the duplicate and use this genes' position to perform a swap between both parents again. This process continues until no more duplicates are found[2].

To demonstrate the procedure, consider these two permutation chromosomes containing 6 integers numbered 1 through 6:

First both parents item on position 1 is swapped:

- **Parent 1** [3 4 6 2 1 5]
- **Parent 2** [4 1 5 3 2 6]

**Step 1**

- Offspring 1 [4 4 6 2 1 5]
- Offspring 2 [3 1 5 3 2 6]

This leads to a duplicate on position 2, so proceed to swap this position:

**Step 2**

- Offspring 1 [4 1 6 2 1 5]
- Offspring 2 [3 4 5 3 2 6]

Another duplicate, now on position 5, will be swapped:

**Step 3**

- Offspring 1 [4 1 6 2 2 5]
- Offspring 2 [3 4 5 3 1 6]

Position 4 is next to be swapped:

**Step 4**

- Offspring 1 [4 1 6 3 2 5]
- Offspring 2 [3 4 5 2 1 6]
With step 4, there are no more repeated integers, so the process is finished.

Edge recombination is a specific operator that works by looking at the genes as nodes in a graph. This concept is illustrated in Figure 12. Two adjacent nodes make an edge between them. This operator was designed specifically for use in solving problems such as the Travelling Salesman Problem\textsuperscript{[13]}. It only changes a few edges during crossover, in an attempt to avoid losing good edges.

![Figure 12: Genes as nodes in a graph](image)

This operator uses two parent chromosomes, and produces one offspring. Before the actual recombination takes place an edge list is constructed for each node. An edge list for a specific node is a list of adjacent nodes. All nodes' edge lists together form what is generally called an adjacency matrix.

Building the offspring chromosome is done iteratively in 5 steps\textsuperscript{[13]}:

1. Choose the first node from one of the parents, and mark this as the “current node”. This is either done randomly or according to step 4.

2. Remove all occurrences of “current node” from the adjacency matrix.

3. If “current node” has entries in it's edge list, proceed to step 4, otherwise go to step 5.

4. Check which node in the edge list of “current node” has the fewest entries in it's own edge list. This will become the new “current node”. Any tie is resolved randomly. Proceed to step 2.

5. If there are still node not yet introduced to the offspring, randomly choose one of them and go to step 2. Otherwise, the offspring is done and the algorithm can stop.

As an example, take these two parent chromosomes:

```
Parent 1   [A B C D E F]
Parent 2   [B D C A E F]
```
Figure 13 shows the parent chromosomes as graphs.

![Parent 1](image1.png)  ![Parent 2](image2.png)

**Figure 13: Parents for edge recombination**

First an adjacency matrix is built from the two parents. Each row is an edge list for the node specified in column 1.

1. The start of the new chromosome is randomly chosen between A and B. B is randomly chosen.

2. B is now removed from the edge lists.

3. The next node is chosen from B's edge list, which is either A, C, D or F. Three of these nodes have two entries in their own edge list: C, D and F. Assume that C is randomly chosen.

4. C is removed from the edge lists.

5. The edge list for C shows that A and D are the candidates for the next node. D is randomly picked. Note that B is completely gone from the adjacency matrix since it is not used anymore.
6. [B C D] has been chosen so far. D must be removed from all edge lists now. And since we are done with C that row is also removed.

7. D's only edge is E so E becomes the next node.

8. E has two adjacent nodes, A and F. Both A and F have one edge left, so a random choice is made for A.

9. A only has one adjacent node left at this point, so F is the final node.

The resulting offspring chromosome is:

[B C D E A F]

*Figure 15 shows this offspring chromosome as a graph. The algorithm combined edge AF from parent 1 and edge AE from parent 2.*

Mutation is easy to accomplish on permutation chromosomes. To ensure that no duplicate items are introduced, a swap between two random positions in the chromosome takes place.
2.2.4.3 Trees

The tree representation of chromosomes is most common in genetic programming. This is an area where evolution is used to produce computer programs, typically using what is known as syntax trees [4]. Trees have also been used for solving classification problems, using so called decision trees.

The trees used in this thesis will all be binary trees. One crossover operator will be used for tree structures, subtree swapping. Two parents will produce two offspring using this method. A random node is chosen on each parent. The first parents' node along with every child-node below it will be swapped with the second parents' node.

Two parent tree chromosomes are shown in Figure 16. From parent 1 node C is randomly picked, and from parent two node B is randomly picked. To perform the crossover operation, the trees are cut at the selected nodes. Node C with children of parent 1 switches place with node B of parent 2.

Figure 17 shows how the offspring will look after crossover. Introducing duplicate nodes is not seen as a problem using this method.

![Figure 16: Tree chromosome parents](image1)

![Figure 17: Tree chromosome crossover offspring](image2)
Using this method of crossover, the chromosomes will have varying lengths, but the nodes that get switched between trees are always the same. Mutation takes care of introducing new nodes, either by adding new nodes, or by replacing an existing node with a random value from a list of possible values. Mutation can also cause a node to be removed entirely.

*Figure 18* shows how node replacement works. The second node containing B is randomly picked for mutation, and B is replaced, introducing the value C in that position.

![Figure 18: Tree mutation - replacing a node](image)

*Figure 19* shows how mutation can lead to the tree growing. The first node with value B is randomly picked by the mutation algorithm. A new node will be created as a child to this node, containing a randomly picked value from the predefined list of possible values. In this case a second A was added leading to the tree on the right.

![Figure 19: Tree mutation - growth](image)

The last possibility of mutation is the removal of a node. Since mutation generally shouldn't have as big an impact on chromosomes as crossover does, only the leaves of the tree are considered for removal. The leaves are the outermost nodes that have no children of their own.

*Figure 20* shows this process. The third leaf, with value E, is randomly picked. The right tree shows the same tree after mutation.
2.2.5 Diversity

The diversity of a genetic algorithm can be said to be the capability of making the search space expand into the problem space. Low diversity means that the search space likely will be unable to traverse the entire problem space, and will at the same time be faster to converge on a solution. The opposite, high diversity, will lead to a larger search space, and thus, more time will be spent before a solution is found. This is because those local optimum that may have been regarded as solutions in a population of low diversity, are now being passed in order to come closer to the global optimum. There are benefits to both methods, and the time that can be spent searching for a solution is an important point.

Since the initial population is often randomly generated, there may be some individuals that have a much larger fitness value than the rest of the population. This may lead to premature convergence since the few very fit individuals will be used the most for mating, and thus limiting the search space.

The diversity can be changed using what is known as a scaling function\[11\]. A scaling function modifies the populations' fitness values, which changes the diversity without regard for the parameters of the genetic algorithm.

Two different scaling functions will be looked at in this thesis. These are rank scaling and sigma scaling.

2.2.5.1 Rank scaling

Rank scaling is a method of avoiding premature convergence on suboptimal solutions\[11\]. This is accomplished by modifying the populations' fitness by their rank. An individuals' rank is it's position in the population when it has been sorted by fitness, either ascending or descending.
In *Figure 21*, the left diagram shows a population of 10 individuals with their fitness. The tenth individual is twice as fit as the ninth individual, which means the genetic algorithm will let it reproduce more than any other individual. Rank scaling can modify this behaviour, by changing the populations' fitness to give other individuals more chance to reproduce. The diagram to the right in *Figure 21* shows the same population after being scaled according to rank. This means that each individuals' fitness will be the same as their numeric position in the population. In the diagrams this is the number below each bar.

Each individual keeps their relative position in the population. This promotes diversity while not discarding the fittest individuals.

### 2.2.5.2 Sigma scaling

Sigma scaling is a method of keeping the selection of mates relatively constant throughout the execution of the genetic algorithm\(^{[11]}\).

This scaling function depends on the standard deviation and mean fitness of the population. *Figure 22* shows the formula used to calculate each individuals' fitness.

\[ \text{scaled fitness} = \frac{f(i) - \mu}{2\sigma} \]

*Figure 22: Sigma scaling formula*

\(f(i)\) is individual \(i\)'s fitness, \(\mu\) is the population mean fitness and \(\sigma\) is the standard deviation of the population.
The effect of Sigma scaling on a population is seen in Figure 23. The left diagram is the original population of 10 individuals again, with the tenth individual twice as fit as the ninth. After applying Sigma scaling to the population, the difference between all individuals have lessened. This relative difference will be kept throughout the course of the genetic algorithm.

In the beginning there may be big differences in fitness. This could potentially lead to premature convergence if one or two individuals are much fitter than the rest of the population. With Sigma scaling, this problem is alleviated by making sure that no individual is far fitter than the rest of the population. As seen in the right diagram of Figure 23, even though the tenth individuals' scaled fitness is more than twice as high as the ninth, the numeric fitness value is not nearly as high. This impacts the selection process and ensures that the population diversity is maintained.

2.2.6 Termination criteria

When to end the execution of the genetic algorithm depends on the problem. There are four simple methods of termination that are commonly used. These are:

- termination after elapsed time
- termination after number of generations the GA has run
- termination on fitness converging on expected value
- termination on population mean fitness converging

The first method terminates the GA after a predefined amount of time has elapsed. The second alternative is to run for a set amount of generations, after which the GA will terminate.
The third criteria relies on a fitness value that is specified beforehand. A threshold value is also specified to determine how close to the expected fitness an individual should be to be considered as done.

For an expected fitness value of 100 and a threshold of 5%, any individual with a fitness value within 95 – 105 would be considered converged by the GA. The reason for specifying a threshold is because it’s not certain any individual is capable of attaining the exact value, or maybe it is possible, but it would take much longer time to reach that precision.

Another criteria to terminate on is the mean fitness of the population. This requires a method of detecting if the populations' fitness can be considered converged. Generally this is done by comparing the mean fitness of the population with the fittest individual, and if the mean fitness is within a fixed percentage the GA should terminate.
2.3 Decision Trees

2.3.1 Overview

A decision tree is a graph in the shape of a tree, modelling events or decisions that need to be made and the consequences of these. The tree structure can be both binary or ternary, depending on what is deemed the most appropriate for the problem.

Decision trees are easy to understand, and a fast way to come to a decision. The nodes that make up a tree are decision points, connected to form a path to a final decision.

An example decision tree is in Figure 24 where the decision to be made is whether to play tennis today or not. This is a ternary tree, with one node having three possible outcomes and the decision is a simple yes or no.

![Decision Tree for Playing Tennis](image)

To traverse the tree and come to a decision, one begins at the root node and evaluate it's expression. The path that best fits the node's expression, is the path taken. This continues for every node in the tree until a leaf node is encountered. Leaf nodes are different, as is evident in Figure 1, where they have the value of either "Yes" or "No". This is the decision we will make on the original question on whether to play tennis or not.

The decisions do not have to be limited to yes or no however. They are used in many different areas, all with their own requirements. These areas include decision analysis, data mining, machine learning and artificial intelligence for games.
2.3.2 Learning decision trees

Learning decision trees is a method for adapting or creating trees. They have been used for a broad range of tasks, including diagnosing medical cases and assessing the credit risk of loan applicants[5].

Decision tree learning has traditionally been done using training data under supervision using an algorithm called ID3[9]. This algorithm uses a so called top-down, greedy search through all possible decision trees for a set of data. This algorithm has largely been replaced by its optimized successors C4, C4.5 and C5 today[8].

Another choice for learning algorithm would be genetic algorithms. They have been used for evolving tree structures in both genetic programming and classification problems[7]. In the aforementioned case ID3 and genetic algorithms are used in combination to evolve decision trees. This will not be explored further in this thesis.

2.3.3 Decision trees for behaviour

This is a special case of decision trees which deal with the behaviour of an object or character. This is not the same as behaviour trees however, which most often aims at systems and software engineering.

Applying decision trees for character behaviour is often used in computer games[8], to allow characters to quickly make an informed decision on how to react. The nodes typically pose a question to an attribute that is present in the world, and the attributes value determine which child node to proceed to. Leaves present the actions that can be taken.

![Example tree of behaviour](image)

*Figure 25: Example tree of behaviour*[8]*
Figure 25 illustrates how a binary decision tree may look. It considers the case of a character that has to decide whether to attack, move or creep, depending on the enemy's position.
3 Realization

This section looks at the realization process of the framework and its two applications. The framework's internals are described, while looking at how it was designed using principles of biology and current knowledge of genetic algorithms as the base.

The two applications are also described, beginning with the TSP application. Their function and approach to the problem they are designed to solve is discussed, showing how the framework will be used to that end.

3.1 The framework

The framework is written in C++, and it is also meant to be used from this language. This language was chosen because the author has the most experience using it, and it has been examined quite thoroughly during the studies. Some effort has been put into simplifying the use of the framework, to create a logical structure which can be understood by the user. Making it flexible to support problems outside of the scope of this thesis has also been done to some degree.

Genetic algorithms have their roots in biology and evolution as mentioned in the theoretical section. This naturally influences the design with the logical separation into components and how they communicate.

3.1.1 Overview

The basic components of a genetic algorithm include:

- Population
- Genome
- Selection
- Crossover
- Mutation

Most implementations use these components. Figure 26 illustrates the basic layout of the framework. This is the core of the framework, which everything else depend on. This is a simplified model, which does not show the auxiliary classes that are necessary for complete operation.

Population is the head class, storing the current generation of individuals. An individual has a genome, consisting of one or many chromosomes. Using this layout gives the possibility to simulate
problems where multiple different problem representations are used.

The entire population will use the same types of chromosomes. For example, an individual with both a tree chromosome and a bit-string chromosome could be used. This way, when calculating the individuals' fitness, one of the chromosomes could improve the individual's fitness while the other doesn't.

Crossover and mutation are genetic operators that work on chromosomes, as such they are member functions of the base chromosome class.

Evolution is handled by the population class, which delegates the work of selecting mates upon a chosen selection operator. During evolution, every individual's fitness is calculated. The actual calculation is done in a user-provided function since it's a problem-specific task. If a scaling function is chosen, it will be applied to the population before the selection operator is called.

3.1.2 Design

The framework is designed to be easily extended. This is achieved by using a modular design with base classes and inheritance.

In the population class, which selection operator, scaling function, and which types of chromosomes to use for the individuals are set using member functions. This design provides an advantage during runtime over using templates, which is the ability to change how the genetic algorithm operates easily during a simulation.

Each step in the genetic algorithm can be customized by defining a new class inheriting from the appropriate base class. Figure 27
shows how selection, scaling and termination criteria inherit from base classes. These classes are provided with the framework and work on a wide variety of problems.

The types of chromosomes that can be used are also inherited from a base class, as seen in Figure 28. This allows the genetic algorithm to work on any kind of problem representation without having any knowledge of how it is actually structured.

The class names for the termination criteria in Figure 27 are taken directly from the framework. The termination criteria are from top to bottom:

- termination after a specified amount of generations
- termination on elapsed real time
- termination on convergence on expected fitness value

Any combination of these termination criteria can be used.
Figure 28: Chromosometypes and base class

Figure 28 shows the chromosome base class and the implemented types of chromosomes in the framework. These support the bit string, tree and permutation type problem representations respectively.

Looking at the complete framework, Figure 29 shows how all the classes work together. Population communicates with SelectionBase, ScalingBase and TerminationCriteriaBase. For an application, this will be one of the existing implementations from the framework or a custom defined one. The same goes for ChromosomeBase, which will be a specific type of chromosome. As mentioned before, since an Individual object can support multiple ChromosomeBase objects, it is possible to use different types of chromosomes.

3.1.3 Chromosome types

A challenge in designing the framework is how to support different chromosome types. Some points that were taken into consideration while designing this part of the framework are:

- the framework should function regardless of the underlying representation
• easy to implement new types of chromosomes
• possible to use any number of different chromosome types on one individual
• possibility to dynamically change chromosome type

The first two points can be done using two different approaches. The first is to use templates, which would provide the necessary functionality. The second approach is to use pointers to base classes, and through inheritance provide the different chromosomes. Considering the third and fourth point, templates become less of an option. Attempting to use templates for this means the code for that implementation becomes big and harder to manage.

The choice fell upon using inheritance instead, with pointers to the base class. Using this design supports all four of the points.

3.1.4 Testing

To verify that all components of the framework do their job correctly, a testing module has been developed. The framework is tested by a series of test cases, where each test case works in as small an area as possible while being able to verify the outcome of each test case against expected outcome.

By testing really small portions of the framework in each test case, it is possible to catch and localize errors easily. The test cases are done sequentially, starting with construction of objects, which has to pass for anything else to be possible. After that a series of functionality tests are performed. The test cases are isolated, so that they cannot interfere with each other.

These are the test cases:

1. Chromosome construction
2. Individual construction
3. Population construction
4. Chromosome crossover
5. Chromosome mutation
6. Individual crossover
7. Individual mutation
8. Fitness scaling functions
9. Selection algorithms
10. Evolution

11. Termination criteria

Each test case only creates the objects necessary for their particular tests. By passing the construction tests 1 through 3 the other test cases know that construction of the objects won't be the problem.

Inside each test case there are checkpoints, these are hidden during normal execution. If a problem were to appear though, the latest checkpoint passed will be displayed, along with any relevant information stored within it as well as what file and line number the error occurred at.

For every test case, a certain pattern is followed. In pseudo-code this is roughly how it works:

```
Construct objects
for 1 to 1000
    Set Checkpoint
    Do Action
    assert Actions
```

For any action that involves random numbers, a loop is used for an arbitrary number of iterations to ensure that the algorithm doesn't depend on a certain input value.

There are two ways in which problems with the framework will appear. The first is if, during the testing modules execution, one of the called functions from the framework cause an error. This could be anything, like for example reading outside of the bounds of an array. This will abort the test, and no other test is performed. The failing test case, the latest checkpoint as well as a diagnostic error message is displayed.

The second way that problems will appear is through assertions. They are used to verify return values and integrity of data after function calls. If an assertion fails to pass, the data that was being monitored by the assertion is displayed, along with the condition to pass. These are generally not a cause for the tests to be aborted however, and instead the number of such errors will be counted.

These tests are not capable of catching every error that could possibly occur, such testing requires more than assertion-like tests. Memory leaks won't be caught for example. Still, the testing module does the job of catching logical errors that may have been introduced during changes to the internal framework.
3.2 The Travelling Salesman Problem

3.2.1 Introduction

The TSP is an optimization problem where the shortest tour through a set of cities is to be found where each city can only be visited once. The starting city is visited again as the final city to complete the tour.

It is classified as an NP-complete problem. This implies that there is no efficient algorithm that can solve TSP's. There are algorithms that can solve TSP's however, but they are likely to increase exponentially in worst case execution time with the number of cities involved\(^3\).

There are different variations to the TSP, all with different prerequisites. The variation explored in this thesis is the simpler symmetric Euclidean TSP, where the distance between any two cities is the same back and forth. The distance for a tour through all cities is calculated as the sum of the Euclidean distance for each pair of cities as shown in Figure 30.

\[
d = \sum \sqrt{(x_n - x_{n+1})^2 + (y_n - y_{n+1})^2}
\]

Figure 30: Euclidean distance

This equation does not take into account the path back to the starting city, but that is easily adjusted by adding the distance between the first and the last city.

Two different layouts of cities will be used. The first is a basic circle with points along it's edge as cities. The second is a 29 city tour of Western Sahara taken from ”The Travelling Salesman Problem” website\(^{14}\).

The GA will be configured to use Roulette Wheel selection, with lower fitness values considered better. Since the order of the cities make up the solution, permutation chromosomes will be used. Each simulation will be run 6 times and averaged to get a better sample of the results. The termination criteria used are fitness convergence on the known shortest path for each problem, as well as terminating on 1000 generations. The latter termination criteria is used so that there's a limit on how long the GA will run. Most likely the population has already converged if it takes this many generations, or it could take a huge amount of generations to actually get any further improvement.
3.2.2 The circle tour

A circle presents an advantage when conducting tests. The shortest possible tour for any number of cities along the circle is obvious at a glance, which is the tour following the perimeter of the circle. *Figure 31* shows this tour.

![Optimal circle tour](image)

*Figure 31: Optimal circle tour*

The circle tour is used to measure how well the GA scales with increasing amounts of cities used for the problem, with the rest of the parameters fixed. The precise length of the circle depends on the amount of cities used.

The parameters used for the circle tour are listed in *Table 1*.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>400</td>
</tr>
<tr>
<td>Crossover rate</td>
<td>70%</td>
</tr>
<tr>
<td>Mutation rate</td>
<td>10%</td>
</tr>
<tr>
<td>Elitism</td>
<td>20%</td>
</tr>
</tbody>
</table>

*Table 1: Circle tour - GA parameters*

All parameters are based on the results of a study done by De Jong\(^2\).

The number of generations it takes to find the optimal solution will be measured, along with the time it spends doing these calculations.
These measurements are recorded for all numbers of cities from 10 to 20.

The simulation is run using these scaling functions:

- No scaling
- Rank scaling
- Sigma scaling

No scaling is considered an inactive scaling function in the framework. For each scaling function, the simulation will be repeated using both cycle crossover and edge recombination.

### 3.2.3 Western Sahara – 29 cities

This tour is based on a map of Western Sahara with 29 cities shown. It is interesting since it has a higher number of cities, and there is a known optimal solution with logged computation time\(^{14}\). The results from gafw will be compared to their solution to see how well gafw compares to them.

For this tour, population size is varied, while keeping the rest of the parameters fixed, in order to see what effect the size of the population has on the results. Rank scaling will be used for the simulation, since it provides fast convergence. Using Rank scaling does mean the GA is more susceptible to premature convergence however. The choice of scaling function is not important as long as it remains the same throughout all simulations.

*Table 2* lists the parameters that are used for this tour.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crossover rate</td>
<td>70%</td>
</tr>
<tr>
<td>Mutation rate</td>
<td>10%</td>
</tr>
<tr>
<td>Elitism</td>
<td>20%</td>
</tr>
<tr>
<td>Scaling function</td>
<td>Rank scaling</td>
</tr>
</tbody>
</table>

*Table 2: Western Sahara tour - GA parameters*

The simulation is run for the following values on the population size:

<table>
<thead>
<tr>
<th>Population Size</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>800</th>
<th>1600</th>
</tr>
</thead>
</table>
Figure 32 shows the optimal tour for the 29 city problem of Western Sahara. Matching this shape is the goal of the simulation. The optimal tour is of length 27603.

3.2.4 The application

The TSP application is a graphical application developed around gafw. Every generation of the genetic algorithm, the best solution is drawn in the window. Figure 33 shows how the application looks, with a circle tour where the optimal solution has been found.

The panel to the right allows setting parameters, crossover operator and scaling function for the genetic algorithm. The circle tour is built in, and thus the number of cities along the circle can be configured from within the application. Other tours are possible by loading files from the hard drive using the so called TSPLIB format. This file format is very simple, storing the amount of nodes in the tour, and coordinates for them.
While running the simulation, the best solution during each generation is drawn on the window to show what it looks like. Figure 34 shows how this can look in 3 stages. For each stage, a chart over the populations' fitness is also shown right next to the circle. For these samples, a population size of 200 is used along with elitism of 20%.

As can be seen on the fitness charts, the entire population is showing improvement for each stage. The individuals to the left of the fitness charts represent those that were in the elite 20%, the best individuals. It is the leftmost individual among these that is drawn.
Figure 34: Three stages of a circle tour, getting closer to the optimal solution
3.3 The Predator and Prey application

3.3.1 Overview

This application simulates an enclosed world of creatures. There are two different types of creatures, predator and prey. The creatures' goal is to survive, but they need to go through evolution to gain the skills necessary.

The genetic algorithms framework is used for the evolutionary process. Although this is a simulation, no attempt is made to be physically correct. Movement speeds, energy expenditure and hunger are measured using abstract numerical values that would make little sense in reality. For the purposes of demonstrating the framework however, it is sufficient.

There are two sets of data for each creature that determines how they act. First they have a set of attributes:

- max speed
- range of sight
- energy level

These attributes are fixed during runtime. The first two are self-explanatory. The numeric values they can take are abstract and based in the virtual world. Energy level is an abstract measure of how much energy a creature has, anything the creature does expends energy. This is used as a limiter to avoid infinite actions. At a certain point of energy, the creature realizes it has lost energy, which can influence the decisions it makes.

The second is the decision tree. Every creature has a personal binary decision tree that determines how they react to events. This data is altered by the GA.

3.3.2 Behaviour of creatures

Decision trees are used for the creatures decision making, using a predefined set of behaviours. The values a node in the tree can take are of these two categories:

- parameter check (reaction)
- action

Predator and prey naturally have different possible actions and reactions. Table 1 and Table 2 list the possible nodes for prey.
Energy level is a check against the creatures' current level of energy. Walk and run set the speed for the creature and performs the movement. The rest of the checks and actions are described below.

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predator near</td>
</tr>
<tr>
<td>Friend running</td>
</tr>
<tr>
<td>Energy level</td>
</tr>
</tbody>
</table>

*Table 3: Prey parameter checks*

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walk</td>
</tr>
<tr>
<td>Run</td>
</tr>
<tr>
<td>Eat</td>
</tr>
<tr>
<td>Flee</td>
</tr>
<tr>
<td>Wander</td>
</tr>
</tbody>
</table>

*Table 4: Prey actions*

- **Predator near**

  This algorithm relies on the creatures' sight range, how far it can see, to determine if there is any predator in it's range. For simplicity, creatures can see 360 degrees.

- **Friend running**

  If another prey within range of sight is running, this creature will also start running in the same direction. This could be a playful behaviour, or a reaction to the friend running from a predator. Which of these that actually is the case is not important for the simulation.

- **Eat**

  The creature stops at this position to eat whatever is on the ground. The only restriction on how much can be eaten lies with the creature.

- **Flee**

  Fleeing from any nearby predator. The direction to flee in is calculated as a vector away from all nearby predators.

- **Wander**

  This is a default behaviour used when the creature is full, and has nothing better to do. A random direction is chosen, and the creature walks there.
Predators, relying on eating other creatures for sustenance, have different actions.

There is no eat action for predators, instead, if their hunt action is successful, in that they catch up to a prey, that prey will be eaten.

- **Prey visible**
  
  Check if any prey is within range of sight.

- **Hunt**
  
  Start hunting the closest prey, running straight toward it.

- **Search prey**
  
  To simulate searching for prey, any living prey is randomly picked, and a vector toward this creature is calculated. To make it appear more natural, the direction the predator actually moves in is changed by 60 degrees away from the original position of the prey.

### 3.3.3 Simulation

The simulation works by performing a set amount of simulation turns. In this case it will run for 1000 turns. Each turn the decision tree is evaluated, starting at the root. The chosen leaf from the tree decides what the creature will do.

After the simulation has run the predetermined amount of turns, the amount of energy left is added to each individual's fitness value. The purpose is to favour creatures that have managed to eat relatively well. In order to avoid having creatures that only eat to get huge fitness values, there is a limit set on how much they are capable of eating. The tree also affects the fitness. A higher fitness value is awarded to creatures whose tree has correct placement of actions, an action followed by walk or run. For example, the action of fleeing implies that you run. For this to happen with the decision tree, flee needs to have run as a child node.
Once all creatures fitness is determined, the current population undergoes evolution, producing offspring.

A basic tree that gives both populations a decent behaviour exists for comparison. *Figure 35* shows the predators tree and *Figure 36* shows the tree for prey. The simulation will then be run with every creature starting with a tree of only 1 node. The GA is then supposed to construct a useful decision tree for the creatures through evolution and compared with the basic trees from *Figures 35 and 36*.

![Predator Tree](Diagram1.png)

*Figure 35: Standard tree for predators*

![Prey Tree](Diagram2.png)

*Figure 36: Standard tree for prey*

Each node lists the name of the check or action that will happen when it is evaluated. For the energy level check, the left path is
chosen if the creature has enough energy that it does not need to worry about eating. On the other hand, if energy is lower, the right path is taken. For all checks that have a yes/no answer, the left path is considered no, whereas the right path is taken for yes.

### 3.3.4 Genetic algorithm

The framework is setup to use roulette wheel selection favouring creatures with high fitness. The decision tree is naturally defined using the tree chromosome.

The parameters for the GA are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>100</td>
</tr>
<tr>
<td>Crossover rate</td>
<td>80%</td>
</tr>
<tr>
<td>Mutation rate</td>
<td>20%</td>
</tr>
<tr>
<td>Elitism</td>
<td>0%</td>
</tr>
</tbody>
</table>

*Table 7: Parameters for the creatures*

Two separate populations will be used in this application. One for the predators and one for the prey. However, they will affect each others evolution to some degree. Predators for example, need prey around to hunt for food. Without food, they would eventually die.

Both populations will contain 100 individuals, with an 80% crossover rate. A 20% value of mutation rate is used to get some variation introduced to the decision trees. Too much mutation would change the trees every generation, which likely means that certain good trees will be altered by mutation and thus lost. Too little mutation would mean the GA has to work for a very long time before anything happens.

No termination criteria is used for this simulation. Instead it is necessary to manually step in and stop the evolution after inspecting and evaluating the progress.
4 Results

4.1 The Travelling Salesman Problem

The results from the simulations are presented as charts showing the number of generations and the time it took for the optimal solution to be found. All results are averaged from the 6 samples taken.

The number of generations spent is an indicator of how well the GA performs, in that it is capable of finding the optimal solution using the specified parameters. For very high numbers, it is likely that the optimal solution will not always be obtained.

The charts show the total running time of the GA for finding the optimal solution during that test run. Per generation time measures are possible, but not as interesting to see.

When measurements were made on the tour length in the Western Sahara tour, all results are averaged from the shortest tour found from each of the 6 samples.

4.1.1 The circle tour

For the circle tour, the number of cities used for the simulation are listed on the x axis for all charts, while the y axis lists either the number of generations or the time spent. For all the simulations run, the optimal solution was found at least once for each sample taken.

4.1.1.1 No scaling – Cycle crossover

Figure 37 shows that using no scaling and cycle crossover works decently for low numbers of cities, while the performance deteriorates for increasing numbers of cities. Figure 38 verifies this by following a similar increase in time spent for finding the optimal solution.

![Figure 37: No scaling, cycle crossover - generations](image)
4.1.1.2 No scaling – Edge recombination

Edge recombination shows slightly better performance in both the number of generations it takes, as well as the time spent. The number of generations it takes is more than halved throughout, which is reflected in the time spent, being slightly shorter.

4.1.1.3 Rank scaling – Cycle crossover

Rank scaling shows as expected much faster convergence. Figure 41 and Figure 42 also show an example of the randomness of genetic algorithms here, with a spike at 19 cities, taking almost twice as much time as for 20 cities.
4.1.1.4 Rank scaling – Edge recombination

Edge recombination provides a more stable result, and converges even faster than cycle crossover. The curves in Figure 43 and Figure 44 show that using Edge recombination the time it takes to find a solution grows linearly with the number of cities.

4.1.1.5 Sigma scaling – Cycle crossover

Using Sigma scaling the results are quite similar to using no scaling function whatsoever. The curve doesn't to grow as linearly however, as seen in Figures 45 and 46. The fact that 17 cities appear to converge faster is for the same reason as Figures 41 and 42 earlier,
the randomness of genetic algorithms and cycle crossover. Using more samples would make these spikes less evident.

**Figure 45**: Sigma scaling, cycle crossover - generations

**Figure 46**: Sigma scaling, cycle crossover - time

4.1.1.6 **Sigma scaling – Edge recombination**

Sigma scaling and Edge recombination together take longer to converge, closer to using no scaling. Both Figure 47 and Figure 48 follow the same growth, being mostly linear but showing some tendency to grow faster toward the higher city numbers.

**Figure 47**: Sigma scaling, edge recombination - generations

**Figure 48**: Sigma scaling, edge recombination - time
4.1.2 Western Sahara

This tour of 29 cities measures the frameworks' performance based on population size. Thus, population size is listed on the x axis, while the y axis can list one of three things:

- number of generations
- time spent
- average tour length

Because the GA was unable to find the optimal solution during some of the solutions, the average tour length is also listed to show how close it was. This is as mentioned the average of the shortest tour that was found.

4.1.2.1 Cycle crossover

The GA was unable to find the optimal solution for any size of the population using cycle crossover. As such, the simulation ran for the full 1000 generations for each sample.

The size of the population did improve on the average tour length as seen in Figure 50. The time spent calculating increases rapidly with bigger populations however, as can be seen in Figure 49.

![Figure 49: Western Sahara tour, cycle crossover - time](image)

![Figure 50: Western Sahara tour, cycle crossover - average tour length](image)

4.1.2.2 Edge recombination

Using Edge recombination, the optimal solution was found for all tested population sizes except 50 as seen in Figure 51. The average number of generations it takes to find the optimal solution is more
than halved at a population size of 800 compared to 50. At 1600 however, the improvement was negligible. The time it takes to find the optimal solution also needs to be considered, which increased by over 800% at a population of 800 compared to 50. This is an increase of over a minute compared to the less than 10 seconds required at a population of 50. However, the optimal solution was only found once during the tests for this population size.

Looking at Figure 52 over the time spent, it increases fairly linearly for all sizes of the population.

![Figure 51: Western Sahara, edge recombination - generations](image1)

![Figure 52: Western Sahara, edge recombination - time](image2)

![Figure 53: Western Sahara, edge recombination - average tour length](image3)

The average tour length that was measured during the simulations can be seen in Figure 53. At a population size of 400 and upwards, the average tour length is less than 28000 which is very close to the optimal solution of 27603.
4.2 **Predator and prey**

The simulation ran for 52 generations, at which point a decent population emerged. Figure 54 and Figure 55 show the most successful decision tree for predator and prey respectively. 39% of the predator population had the tree from Figure 54, which gives them the ability to hunt and search for food. There is no conservation of energy however.

In the prey population, 31% had the tree from Figure 55. This tree allows the prey to eat to sustain themselves, as well as recognizing predators that are nearby. However, their ability to get away from predators is limited, with the flee node having no child node with run. To flee then, they rely on the run node to determine a proper random direction to run in.

---

![Figure 54: Most successful tree for predators after evolution](image)

![Figure 55: Most successful tree for prey after evolution](image)
Some of the evolved prey were able to flee properly, but their decision tree didn't spread enough during the simulation. 19% of the prey population remained alive at the end of the simulation and all predators were alive.
5 Discussion and conclusion

5.1 Discussion

Getting started on the framework was the toughest point. I had many different ideas that I went back and forth between, and started building prototypes of them only to come up with something better.

Even after coming up with a design that I was pleased with, the framework took more work than what was expected to finish. This happened because while building the demonstration applications, I found things that would be better off changed inside the framework instead of altering the way I used it.

A more thorough planning phase, where the possible use cases are considered, could perhaps help to make the development smoother. Some points are difficult to understand without testing them practically however. Either way, it is worth considering while still planning the project.

The result from The TSP application show that the framework is capable of solving such problems in a reasonable time for a lower amount of cities. Edge recombination repeatedly proves to be a better choice of crossover operator, since it tries to improve on the parents genes using an algorithm, rather than just using a shifted combination of their genes as cycle crossover does.

Tests done during development as well as preliminary testing for the experiments presented in this thesis, showed that very small populations were extremely prone to premature convergence. With this in mind, none of the tests were done for population sizes smaller than 50.

Looking at the 29 city tour of Western Sahara, the “Traveling Salesman Problem” website specifies that their TSP solver called Concorde was capable of finding the optimal solution on Western Sahara in 0.09 seconds. For a population size of 100, the optimal solution was found on average in 13 seconds using gafw. Compared with less than a second this is very slow. Concorde uses a different method of solving TSP's however, and obviously genetic algorithms isn't the best choice for these kind of problems.

The scaling functions proved to work as expected. Rank scaling makes the population converge much faster, while being prone to premature convergence. This is most obvious on cycle crossover, where 19 cities takes twice as much time to converge as 20 cities. Sigma scaling converges slower than Rank scaling, but faster than using no scaling function at all. This is most likely because Sigma scaling is supposed to help the population overcome premature convergence by promoting diversity. At the same time, this very
The effect of Sigma scaling also makes it converge faster since there will be less local optimum that the population attempts to converge on.

The Predator and prey application tried to evolve populations of predator and prey that would be able to behave similar to the behaviour from the predefined decision trees. This was partially successful, with both populations being able to replicate that behaviour to a certain degree.

It is of course possible to modify the fitness calculation for the populations, so that higher resemblance to the standard trees get higher fitness. But taking that too far would only lead to a much slower way of developing an equal tree, which would be pointless.

By leaving the restrictions quite loose, the intention was to see if the GA would be able to come up with an interesting tree that I had not thought of. This didn't happen, but nonetheless, the evolved trees do show that the populations learnt to eat and therefore at least do a decent job of surviving.

The fact that only 19% of the prey were alive at the end is unfortunate, but natural, since most of the population had poor fleeing behaviour. Many of them were unable to flee at all, having had bad luck during evolution where they may have lost that behaviour, or they never developed it at all. Adjusting the fitness calculations to promote fleeing more would have helped this.

The simulation was also tested while running for 100 generations. The results from that showed that the trees grow larger, but mostly with the same nodes, which meant that no new behaviour was introduced compared to the simulation of 52 generations. This points at the fact that the fitness calculations may need adjustment for the simulation to run any longer duration. Also, elitism wasn't used during this simulation, which meant that there's a larger risk of losing good candidates.
5.2 Conclusion

The implemented features of the framework that have been tested using the demonstration applications are working as intended by the defined goals. It can successfully be used to solve problems of the kind explored in this thesis.

The purpose of the thesis was for me to learn more about artificial intelligence and problem solving by developing a framework for genetic algorithms. This has been accomplished, as I have learnt about designing libraries, and different techniques that can be used to provide the features that have been built into gafw. Moreover, I have more practical experience of the development process of a larger project which will be very useful for the future.

5.3 The future of the framework

The framework currently supports only a subset of the known capabilities of genetic algorithms. The supported chromosome types are limited to what was necessary to make the demonstration applications work. Integer, real value based chromosomes would be interesting to implement because they supposedly provide better performance, as well as being easier to interpret during development.

There are many more genetic operators that are currently not implemented in the framework. These should be included in a feature complete version of the framework.

It would be interesting to try to optimize the framework for better performance and redo the tests and see how far genetic algorithms are capable of going. One way of doing this, which is very interesting, is to make the framework multi-threaded so that it can benefit from multiple processors.

Genetic algorithms are used in a variety of fields, and their nature can lead to interesting results. There are other GA frameworks out there that are currently more feature-complete, and would be a better choice for solving an arbitrary problem. I intentionally chose not to compare my work with theirs, to keep the focus on my own framework only. With further work and some external testing, I believe gafw could become a useful alternative to the existing GA frameworks available.
6 References

http://www.it-weise.de/ (acc. 2009-05-11)


ISBN: 978-1-4092-0073-4
http://www.gp-field-guide.org.uk (acc. 2009-08-12)


[6] Decision Trees


[9] ID3 Algorithm


www.cs.colostate.edu/~genitor/Pubs.html (acc. 2009-05-17)

http://www.cs.colostate.edu/~genitor/Pubs.html (acc. 2009-05-17)

[14] The Traveling Salesman Problem
7 Index

A
adjacency matrix.................................................................23, 24
allele.........................................................................................14, 19

C
crossover rate...........................................................................19, 51
Cycle crossover........................................................................22

D
decision tree............................................................................32, 33, 34, 47, 49

E
Edge recombination...............................................................23
elitism.........................................................................................18

G
genetic operator.....................................................................18, 20

I
indirect representation...........................................................19

M
mutation rate............................................................................19, 51

P
premature convergence...........................................................12, 13
problem space...........................................................................13

R
Rank scaling.............................................................................28
Roulette Wheel.........................................................................20

S
search space............................................................................13, 17, 19, 22
Sigma scaling............................................................................29

T
termination criteria...............................................................18, 30, 37
truncation selection....................................................................20