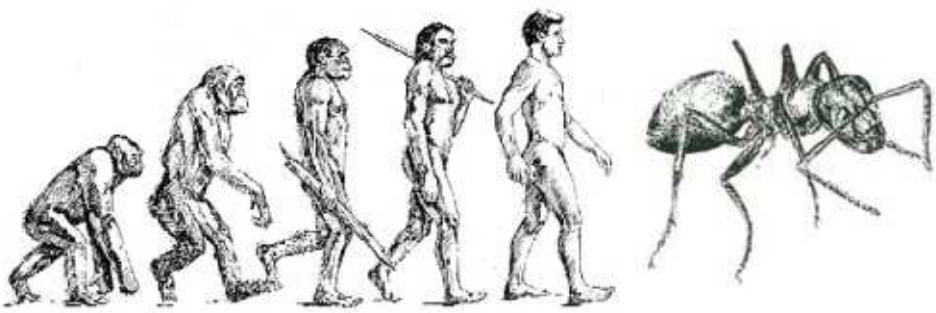
GeneAnt 2003



Aalborg University

Department of Computer Science.
Fredrik Bajers Vej 7E, 9220 Aalborg Ø.

Title:
GeneAnt 2003

Project Term:
F7S/BOS,
October 2003 - December 2003

Project Group:
E2-208

Group Members:
Ingibjörg Ásta Rúnarsdóttir
Kenneth Vittrup
Morten Zinck
Thomas Winterberg
Kasper Ørum Nielsen
Frederik Dannemare
Peter Sönder

Supervisor:
Jens Dalgaard Nielsen

Abstract

This report deals with the development of an ant for the game `MyreKrig` and a system that employes the use of a genetic algorithm to evolve the ant through ant generations.
Within the system a randomly generated ant population is used as the first generation of ants. Using various reproduction and selection methods, parents in a generation are used for producing the next generation of ants. This Darwinistic approach to (hopefully) producing better and better ants is repeated until a satisfactory result is achieved.
Upon running the genetic algorithm we process the collected data and analyze it along with the chosen strategies for evolving our ant. Upon analyzing we conclude on our findings.
Finally, we have a look at the possibilities that future work on the subject of developing an ant for `MyreKrig` could lead to through the use of other techniques than the one we decided to rely on.

Contents

Chapter 1

Introduction

1.1 Artificial Intelligence

In everyday life we are all surrounded by some kind of artificial intelligence (AI). AI is employed in many different scenarios. Just think of the autopilot in planes and the AI used to help personnel maneuver a spacecraft. These are examples of highly advanced AIs compared to the AI of more simple cruise-controls and ABS systems used in cars, or the AI present in an average spam filtering system to help us avoid unsolicited emails. So what is AI exactly? Below we have presented two loose definitions of AI:

- "It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable."[11]

- "This is a discipline with two strands: science and engineering. The scientific strand tries to understand the requirements for, and mechanisms enabling, intelligence of various kinds in humans, other animals and information processing machines and robots. The engineering strand attempts to apply such knowledge in designing useful new kinds of machines and helping us to deal more effectively with natural intelligence, e.g. in education and therapy."[22]

1.1.1 History of Artificial Intelligence

The first AI research projects began in the 1940s. The English mathematician Alan Turing may have been the first to start doing AI research. Turing believed that an intelligent machine could be created by following the blueprints of the human brain. He wrote a paper in 1950 describing what is now known as the "Turing Test". The test consisted of a test person asking questions via keyboard to both a person and an intelligent machine. He believed that if the tested person could not distinguish the machine from the person after a reasonable amount of time, the machine was somewhat intelligent. This test has become the 'holy grail' of the artificial intelligence community. Turing's paper describing the test has been used in countless journals and papers relating to machine intelligence.[10][20]

Figure 1.1: Alan Turing, English mathematician, 1912-1954

1.2 Autonomous agents

Autonomous agents are software entities or robotic entities which operate in environments constantly changing. They are programmed to make decisions about which actions to take when their surroundings change. Autonomous agents need not to be fed input from other programs because they collect it themselves from their environments via sensors and they do not send outputs to other programs either. In that sense they resemble living beings. Such agents are used in lots of computer games, where the computer-controlled actors are autonomous agents who react to the way the player affects the

environment in the game. Since our project is about `MyreKrig`, we will focus on this application area.

1.2.1 Defining autonomous agents

One of the questions usually asked regarding agents is "How are agents different from regular computer programs?". A definition of an agent would then come in handy. Many definitions of autonomous agents have been developed throughout the years, that may give rise to some confusion and/or even contradiction, so here we use a very qualified definition[6] of an autonomous agent which tries to define the largest group of autonomous agents.

Definition 1.1 (Definition of an autonomous agent). An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to affect what it senses in the future.

It can be disgussed whether argued one should not try to define what an agent is and is not, because it is a fuzzy area. The notion of an agent is really only meant to be a tool for analysing systems. Since our definition of an agent relies on the agent sensing its environment, errors may occur if an agent is moved from one environment to another. Our definition will occationally classify an agent or non-agent incorrectly and will therefore be flawed around the "edges". It is important to note that all software agents are programs, but not all programs are agents. To see this recall that agents collect their data themselves, but regular software programs have to be fed with their inputs.

It is possible to create agents that have subagents. This can be done by creating the agent system in layers, where each layer is accepted by our definition of an agent. Some agents with a layered architecture are not multiagent systems. Müller, Pischel, and Thiel (1995) classify such architectures into vertically and horizontally layered. In horizontally layered systems each layer has access to sensing and acting functionality, making a decomposition into subagents likely. In vertically layered system, only the lowest layer senses, and only the highest acts, making a multiagent decomposition unlikely.

1.2.2 History of autonomous agents

The commercial popularity of autonomous agents became really popular throughout the 1990s and have since then been used in many diverse fields of computer science. The subject of autonomous agents is one of the most interesting and important research areas at present. The word *autonomous* means that the agent exercises control over its own actions, at least to some level.

An important and interesting thing to note about the creation of agents is that while the creators of agents are mostly in control of their creations, because they can kill them at any point in time if they wish so, there are examples of free agents that now roam the internet outside of their creators control. This marks an interesting and important change in researchers' philosophy and in their approaches to agents, in that more value is sometimes placed in watching something evolve than in maintaining full control over it.

There are several categories of agents. An example of a kind of agent could be a robot that explores the surface of a moon or planet and logs info about the geometry. Another kind of agent could be one which is used by researchers to learn more about our own evolution and about evolution in general. It is also concievable that we might create an agent, whose purpose it is to perform some minor tasks that make certain tasks easier for us to complete. The concept of agents has great potential.

Although great strides have already been made in the field of autonomous agents, one should remember that their intelligence is still only an illusion. When the agents take some unexpected actions it is usually just because it has some algorithms with some randomized events in them, but this "intelligence"-effect usually satisfies our needs.

1.2.3 Practical application of autonomous agents

Autonomous agents are applicable in various scenarios. The relevance of autonomous agents in the industry of computer games is probably the most interesting to most people. When talking about agents in computer games, they are often referred to as bots (short for robots). Another example, again relevant for the computer-loving community, is IRC (Internet Relay Chat) bots that may control things such as who is allowed on an IRC channel, and whether people should be kicked or banned from a channel as a result of improper behaviour (e.g. channel flooding, SHOUTING, writing in colours, etc).

Military robots

Agents in the form of military robots have lately been put into use in both Afghanistan and Iraq. Prior to Afghanistan, the military was using robots for search-and-rescue and ordnance disposal, but mostly viewed them as long-term research. Airborne drones had proved easier to build than effective land robots.

The newest land robots used in Afghanistan and Iraq are able to navigate terrain and obstacles and, for instance, extend a "neck" that allows them to peer around corners. They can carry out tasks such as mine sweeping, placing explosive loads, laying down a smokescreen, and test for chemical weapons. The machines are also learning how to right themselves if they flip over as well as how to follow their tracks back home if they lose contact with their base.[16][17]

Figure 1.2: A "PackBot" produced by iRobot Corporation[8]

Project Alpha, a U.S. Joint Forces Command rapid idea analysis group, is in the midst of a study focusing on the concept of developing and employing robots that would be capable of replacing humans to perform many, if not most combat functions on the battlefield. The study, appropriately titled, "Unmanned Effects: Taking the Human out of the Loop," suggests that as early as 2025, the presence of autonomous robots, networked and integrated, on the battlefield might not be the exception, but, in fact, the norm.[24]

Robots will someday master many of the complex, individual tasks required in combat, experts claims. Then, something even more powerful will follow: robots that work together.

Chapter 2

The game MyreKrig

This chapter is used as an introduction to MyreKrig and its environment. It also describes the basic way of implementing an ant to some degree of detail.

2.1 Our terminology regarding MyreKrig

Starting from the next section of our report and throughout the rest of it we will be using the terms skeleton, individial, ant and ANT as defined in this section of our report. The terms are explaned below.

When the term skeleton is used we are refering to the program we have written in the programming language C, which is linked to MyreKrig.

When the term individual is used we are refering to a string of bits representing parameters for our skeleton to use (more on this later).

When the term ant is used we are refering to one of the dots you see when you run a game of MyreKrig. Those ants are exact copies of each other and are created by combining the skeleton and an individual.

We use the term ANT when we are refering to the C program and its accompanying parameters from an individual. When those two components are used with MyreKrig they are regarded as one ant design, which is simply called an ant. This is the terminology, which is used by the creators of MyreKrig. However, we will use the designation ANT to show that we are talking about the design and not just one of many ants in MyreKrig. Ants in a game of MyreKrigare displayed as simple dots.

It should be noted that some sections of the report describing parts of our ANT's design aspects where the designation ANT can safely be exchanged

with ant and vice versa. This is not an error in any way. An example could be the movement pattern of the ANT/ant on a map. Here both designations would be correct.

Sometimes we refer to an ant breed or team. Here ant refers to the ants running around in a game of MyreKrig. When talking about a breed or team we are talking about all the ants created with the same individual along with the skeleton of course.

Although there is a hint of technicality in the above explanations, it is only because it is required to properly distinguish between the terms' meanings and uses. Very specific design and implementation details about our ant will follow in later chapters.

2.2 Background for MyreKrig

The following has been taken from www.myrekrig.dk. "MyreKrig is a programming game which is a contest where each participant writes a program (edit: to control an ant breed in the game), after which each of the implemented ANTs play against each other. The background story is as follows:

A number of hostile colonies of ants are established simultaneously on top of a *torus*[1]. Piles of food are scattered throughout this torus which the ants can collect and bring home to their base, at which point new ants are born. Inevitably there will be violent confrontations between the different ant breeds in their fight for survival and conquest. The goal of the game is to construct an ANT algorithm which can efficiently administrate the collection of food, control the communication between (friendly) ants, and also protect og fight against the other hostile ant breeds." [5].

The idea of the game came to Aske Simon Christensen, the founder of MyreKrig, in the fall of 1996. Back then he programmed the game on his *Amiga*. One of the motivating factors was to learn to program in *C*, but also to expand his passion for programming with his friends.

Today the game is still on. Whether it is used for competition or just for simulation must be up to the gamer/programmer to decide, but ask Aske Simon Christensen and he will say that he still sees the game as a cosy passion for programming.[5]

[1] A torus is a rectangular area where opposite sides are adjoining to each other

2.3 About the game

Some variables are set by `MyreKrig` when loading the game. One of them is *StartAnts(10-50)*. This is the number of ants that each ant breed has available, before the game begins. In each game a new torus is created which is defined by *MapWidth* and *MapHeight*. The size of this torus is set to *MapWidth(100k-200k)* and *MapHeight(100k-200k)*, where k is the square root of the number of ant teams. This size is unknown to ANTs, but when playing on maps created by default by `MyreKrig`, the maps are constructed in such a way that their sizes are always divideable by 64, and ANTs can take this into account.

Also upon creating the torus, each ant breed is placed randomly on it. This though, is not totally random, since there must be a minimum distance between each base as defined within the game.

When starting `MyreKrig` all ants are put into a list, which is then randomized. Then each ant moves according to its position in the list starting from the top running down to the bottom. When the turn is finished, the list is then again randomized and all ant can move again.

This could have some impact on the movement of ants. For instance, one ant could move once during some round of turns, and then an opposing ant could move during the same round of turns. Then in the next round of turns, the second ant may get the chance to move once more before the first one moves again. This can be seen in figure 2.1.

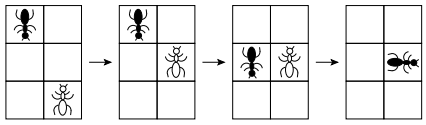


Figure 2.1: An ant moving twice.

In figure 2.1 the white ant moves once during a round of turns. Then the black ant moves as shown during the same round. In the next round of turns the black ant is allowed to move before the white. Because the black ant can see the white ant, it attacks, and kills the white ant.

After each turn new food is placed on the torus. Food is more or less placed randomly. As with placement of bases, this is not totally random, as the chance of food appearing in areas, where none has appeared for a long time, increases.

```
struct SquareData
{
  u_{char} NumAnts;
  u_{char} Base;
  u_{char} Team;
  u_{char} NumFood;
};
```

Table 2.1: Information about the surrounding squares.

When an ant is born, a variable of type `long` is initialized to a completely random number, which can be used by the ant. As it moves around on the torus, it gets information from surrounding squares and the one it is standing on. The surrounding squares, from which it gets information, are limited to only four squares, namely the neighbouring squares to the north, south, east and west as seen in table 2.1.

SquareData is an important part of the ANT, because this is used as an argument in the ant function in `MyreKrig`.

1. $NumAnts$ gives the number of ants on a square.
2. $Base$ tells whether a base is present on the square (1 if there is, else 0).
3. If ants are present and/or a base is present, then $Team$ holds information about what team it/they belong to (own team number is always 0).
4. $NumFood$ holds the number of food pieces present on a square.

2.4 Basic rules of implementation

When implementing an ANT algorithm there are a few basic rules which must be obeyed. First of all, the ANT must be implemented in *ANSI C*. Including files other than `Myre.h` is illegal if you want your ANT to join the league. `Myre.h` is the file which links the ANTs to the game and this should therefore be included in the top:

```
#include ‘‘Myre.h’’
```

```
struct AntBrain
{
  u_{long} random;
  short homex,homey;
  short foodx,foody;
};
```

Table 2.2: An example of an ANT brain.

Besides this there are a few basic requirements which must be implemented in the ANT in order to interact it with MyreKrig. These are summerized below and explained afterwards:

1. A structure which represents the brain of the ANT [2].
2. A function returning a value which defines an ANT's next move.
3. Some type definitions linking it all together [3].

2.4.1 The brain of the ANT

Each ANT is equipped with a brain in the form of a limited amount of memory. Ants on the same team can access each others memory, since they can read and write in that memory area. For this to happen, the ants must be on the same square; otherwise it is impossible. In other words, no global variables are allowed in MyreKrig. The only pieces of information available to the ants are the five squares and the constants *NewBaseAnts* (the number of ants required to create a new base), *NewBaseFood* (pieces of food required to create a new base), *BaseValue(NewBaseAnts+NewBaseFood)*, *MaxSquareAnts* (max number of ant on a square) and *MaxSquareFood* (max pieces of food on a square). These constants are all information provided by MyreKrig.

An example of the brain can be seen in table 2.2.

When moving an ANT around the torus, it would be nice to remember where the base is placed. Therefore many, who create an ANT, use a two-dimensional system of coordination. This allows the programmer to give the

[2] There is no upper limit to the size of the brain, but as described in 2.4.5 the size of the brain has an impact on the final score.

[3] Linking the ANT with the game (e.g. visit [5]).

ANT a sense of placement on the torus. This also makes it easier for one ant to tell another ant where food is placed for instance.

As mentioned, an ant generates a random value upon birth. This randomly generated number can then be used by the ant to create random moving patterns etc. Next, *homex* and *homey* are defined as `short` variables. These are used by an ant to remember where the base is [4]. When an ant encounters a resource of food, it could take a piece of this and move it towards the base. If it then encounters a friendly ant, it would then tell that ant where it had found the food. This is what *foodx* and *foody* is used for. Upon birth these `short` variables are initiated to zero.

2.4.2 Movement of the ANT

When implementing an ANT there are a few basics which need to be taken into consideration. At each turn each ant of each ant breed gets a chance to do one of the following four things:

- Stand still.
- Move one square to the north, south, east or west (possibly attacking simultaneously).
- Move to the north, south, east or west and bring along one piece of food (if food is present in the current square).
- Build a new base [5].

Killing other ants is done by moving one of your ants onto another square occupied by a hostile ant. If there is also a base on the square, this is simply destroyed.

2.4.3 Linking the ANT to `MyreKrig`

At the bottom of the file, the ant breed is identified. This is done by:

```
DefineAnt(name, title, func, mem)
```

[4] This is a simple example, as there could be more than one base. If the breed has more than one base, then *homex* and *homey* will only tell where one of the bases are placed.

[5] In order for this to happen there must be at least 25 other ants (*NewBaseAnts(25)*) and 50 pieces of food (*NewBaseFood(50)*) in the square. The 25 ants and 50 pieces of food are then converted into a base, and can therefore not be used in the game anymore.

Here *name* identifies the ANT and *title* is the `string` (max 10 characters) which is used by `MyreKrig` in the output of the game [6]. It is also in the title that the programmer can define the color of his ANT. This is denoted by #abcdef, where abcdef is a six-figure RGB-number, but this is optional. *func* is the name of the ant function and *mem* is the type of the ANTs brain. An example of this could be:

```
DefineAnt(ExampleAnt, ‘‘Example’’,
    ExampleFunction, struct ExampleBrain)
```

2.4.4 How to win the game

To win the game, you must basically gather most food in the shortest time, thereby becoming the biggest ant colony. This is a very simplified way of defining which ANT wins. A *point* system is used in the game, and the rules for giving points to an ANT is defined as follows:

`points = (ants on team) + (bases owned by team) x` *`BaseValue`*

where *BaseValue* is given by *NewBaseAnts + NewBaseFood*.

Another quantity used in the game is *totalpoints*. This is defined as follows:

```
totalpoints = (points of best team) + (points of second best
                                team)
```

There are three ways in which an ANT can win the game. If just one of these conditions are satisfied, the game is won:

1. One ant colony has reached *WinPercent(75 percent)* of *totalpoints*.
2. At *HalfTimeTurn (10000 turns)* one ANT has gained *HalfTimePercent (60 percent)* of *totalpoints*.
3. *TimeOutTurn (20000 turns)* has passed, meaning the game has ended. Whichever ANT has the most *totalpoints* wins the game.

The winner of the game is the ANT that, at the end of the game, has the most points, as defined by one of the three conditions mentioned above.

[6] The output comes in the form of statistics about the battles played.

2.4.5 Ranking in `MyreKrig`

As mentioned earlier, `MyreKrig` creates an output. This output shows *Performance*, *Prestige* and *Victory* which are calculated by the game.

$$\begin{aligned} Performance &= \frac{Victory}{Time} \\ Prestige &= \frac{Victory}{BrainSize} \\ Victory &= \frac{GamesWon}{GamesPlayed} \end{aligned}$$

Time is the average execution time in microseconds used by the ANT. This means that the more time the ANT is using, the less *Performance* it gets. *BrainSize* is the size of the brain in bytes. If *BrainSize* is 0, then *Prestige* will be 0. Therefore, it is a good idea to make the brain as small as possible.

The goal for each ANT is to get the highest *Overall Performance*:

$$OverallPerformance = \frac{Performance * Prestige}{Victory}$$

A note should be made that *Victory* can become zero. In that case the result is an *Overall Performance* of zero.

With all the rules and equations in mind, it should now be possible to construct an ANT which is both a legal ANT in `MyreKrig` and also has a slight chance of survival.

Chapter 3

Theory

3.1 Genetic algorithms

Genetic algorithms is a subsection of *evolutionary computation*. To optimize parameters in our program we use genetic algorithms. A genetic algorithm simulates natural processes such as reproduction, *crossover* and *mutation* of genes from a darwinistic point of view: from a population of individuals we select for reproduction the individuals from the "survival of the fittest" principle.

It is possible to set up different survival criteria in a genetic algorithm. It depends on which problem we wish to solve with the genetic algorithm. If we e.g. wish to minimize the length of a freight route, then the survival criteria will be a route with a short length i.e. the shorter the length of a route, the greater the probability that the algorithm will always try to reuse the shortest route.

If we instead wish to develop mobile robots that can avoid obstacles in different environments, then the survival criteria will be speed and distance to the obstacles and maybe also minimizing sway.

3.1.1 Definition

The fundamental idea with genetic algorithms is to find good solutions by combining and changing already known solutions. Inspired by *natural selection* and *natural evolution*, genetic algorithms search through very large spaces by moving a set of solutions (the individuals in a population) to new regions in the fitness function. It is expected that the algorithm makes an

improvement.

3.1.2 Biological terminology vs genetic algorithms

Biological terminology

Before we start with the theory of genetic algorithms in computer science, we introduce genetics in nature and how the terms are used in analogy with the terms used in relation to genetic algorithms. The genetic algorithm though is much simpler than the real biological ones. All organisms consist of *cells* and each cell contains *chromosomes*. If we look deeper into a chromosome, we will see long spiral chains of *deoxyribonucleic acid*, also known as *DNA*. Each spiral winding is a *gene*. Each gene contains different proteins. See figure 3.1 to see an illustration of the relationship between a cell, a chromosome, DNA, genes and proteins. All the properties that a person inherits are defined in the genes. The genes dictate, e.g. how we look. Organisms whose chromosomes are arrayed in pairs are called *diploid*. Organisms whose chromosomes are unpaired are called *haploid*. Most genetic software programs use haploid organisms.

The *fitness* of an organism in nature is typically defined as the probability that the organism will live to reproduce, or defined as a function of the number of offspring the organisme has.

Genetic algorithm terminology

The name chromosome typically refers to a possible solution to a given problem. This is often encoded as a *bit string*. We define the genes in a genetic algorithm to be either single bits or short blocks of adjacent bits that encode a particular element in the possible solution. The reason why a gene could also be represented by adjacent bits is that the bits in that case together will represent the property of one gene. Since the setting of genes is done through bit manipulation, we set them by writing either a zero or a one.

Crossover produces two new offspring from two parent bit strings by copying selected bits from each parent. Mutation means flipping a bit/bits in a gene at a randomly chosen position.

Because we have chosen to use bit sequences to represent the genes of a chromosome, we have chosen to represent a chromosome as a bit string which consists of the smaller bit sequences of the genes, so the bit string is therefore

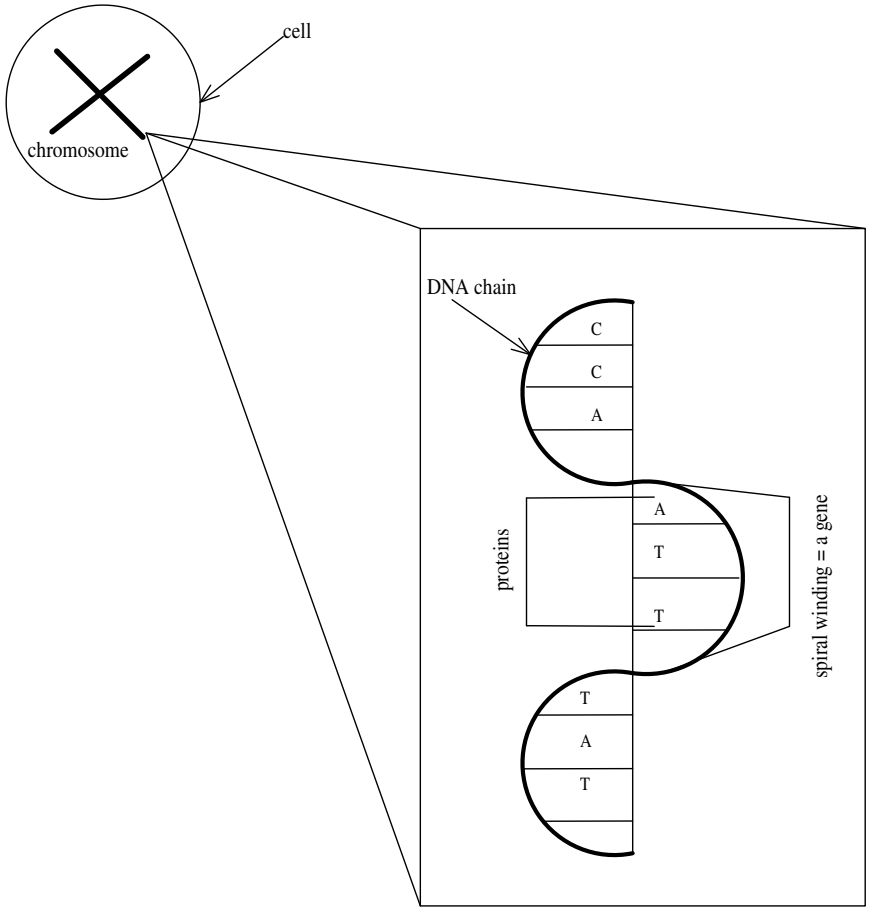


Figure 3.1: Cell, chromosome, DNA, genes and proteins

the representation of the genes in the chromosome. We have chosen to use the term individual as the representation of the chromosome. Each individual therefore consists of one or more genes, and each gene consists of one or more bits. See figure 3.2 to see an illustration of the relation between individual, genes and bits. In the figure one can see that the individual consists of three genes. The first gene consists of three bits, the second eight bits and the third one bit.

3.1.3 The structure of the algorithm

Although different implementations of genetic algorithms vary in complexity, they typically share the same structure. The input to the algorithms is:

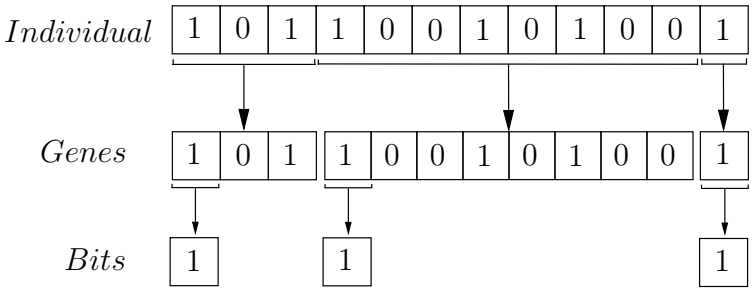


Figure 3.2: Individual

- *The population size* is defined by the number of individuals in the population.
- *The fitness function* is used to evaluate each individual in the population. This is done to find the fitness of each individual.
- *The termination criterion* defines an acceptable level of fitness for terminating the algorithm. An acceptable level of fitness is typically a combination of the number of generations developed and some number of generations that pass without improvement.
- *The value for the crossover probability* is the probability that a pair of individuals are crossed over.
- *The mutation probability* is the probability of an individual being mutated. The mutation probability is quite small in nature and is kept quite low for genetic algorithms as well, typically in the range of 0.1% and 1%[15]. Mutation helps to get genetic diversity.

It will now be described in steps how genetic algorithms work:

Step 1
The population contains N individuals.

Step 2
The fitness of each individual in the population is calculated using the fitness function.

Step 3
If the termination criterion is satisfied the algorithm is stopped. If it is not satisfied, the algorithm continues to the next step.

Step 4
Individuals are selected from the population for mating.

Step 5
The value for crossover probability is used to find out how many individuals to use crossover on. That number of individuals is even, because the crossover probability is always used on two individuals at a time. The crossover operator creates two offspring from each pair.

Step 6
The mutation probability is used to find out how many individuals are going to be mutated. The mutation operator produces an offspring from a single parent by randomly changing a bit/bits in the parent.

Step 7
A new population with N individuals exists. The current population is replaced with the new one and then the algorithm returns to step two.

These steps show that the process is iterative where one iteration is one generation.

3.1.4 Selection

After evaluating all the individuals in the population, it is found out which individuals will get a chance to reproduce i.e. the probability for each individual is calculated.

Selection methods

The selection methods use the fitness of the individuals to select individuals for the next generation (according to rule of survival of the fittest). A few different selection methods can be used, and three of them are described below:

1. *Roulette wheel selection.*
The probability of selecting an individual h_i is:

$$Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{N} Fitness(h_j)}$$

where N is the number of individuals in the population.

Example:

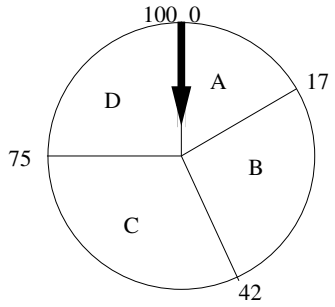


Figure 3.3: Roulette wheel

Figure 3.3 shows a roulette wheel where the number of individuals in the population is four i.e. A, B, C and D. Each individual is given a slice of the roulette wheel. The area of the slice within the wheel is equal to the individual fitness ratio, i.e. $Pr(h_i)$. The fitness ratio of individuals A, B, C and D is 17, 25, 33 and 25 respectively. The roulette wheel is now turned, and when the arrow comes to rest on one of the segments, the corresponding individual is selected. The wheel is turned four times because there are four individuals. This means that the same individual can be selected more than once and some individuals may never be selected.

2. *Rank Selection.*
The probability of selecting an individual ($Pr(h_i)$) depends on the position of the rank of the individual, where the best individual obtains a ranked fitness of N (the size of the population). The second best obtains a ranked fitness of $N - 1$ and so forth.

$$Pr(h_i) = \frac{Rank(h_i)}{\sum_{j=0}^{N} N-j}$$

where N is the size of the population.

3. *Tournament selection.*
To start with, the tournament-size has to be defined e.g. two.
One tournament: Two (the tournament-size) different individuals are drawn randomly from the population. The fittest individual, of the two individuals, is then selected to be in the next generation.

The number of tournaments that will be made depends on the size of the population. If the population size is e.g. fifty then there are made fifty tournaments.
With a tournament-size of two the probability of individual h_i being selected is:

$$Pr(h_i) = \frac{2}{N}(1 - \frac{N - Rank(h_i)}{N-1})$$

where $Rank(h_i)$ is the ranked fitness of individual h_i and N is the size of the population.

Selection pressure
Selection pressure is the ratio between the probability that the most fit individual in the population is selected as a parent, and the probability that an average individual in the population is selected as a parent. Too high selection pressure results in the population converging too early.[4]

Roulette wheel-, rank- and tournament selection
In roulette wheel selection there may be one individual (the best individual) that occupies a large portion of the wheel. This means that there is a significant probability for this individual to be selected. This can cause less diversity in the population and too high selection pressure.
Rank selection tries to overcome the disadvantages of roulette wheel selection by ranking individuals according to their fitness. Tournament selection also uses ranking but in a different way. Tournament selection ensures that the least fit individual is not selected to be in the next generation.

The selected individuals are used for creating new offspring by applying operations such as crossover and mutation.

3.1.5 Crossover

When it is known which individuals in the population will reproduce, the crossover operation can be used. Crossover creates two offspring from two parents by mixing their bit strings.

Crossover methods

A few different methods can be used to perform the crossover. Three of them are described below:

1. *Single-point crossover.* Each time the Single-point crossover is applied, the crosspoint n is chosen randomly. This crosspoint decides how the crossover mask is going to look, e.g. if bit strings of length ten are used for each individual, and the crosspoint is four, then the crossover mask will look like this: 1111000000. The ones show which bits are going to be taken from parent one and the zeros show which bits are going to be taken from the second parent to make the first offspring. When the second offspring is made the ones show the bits taken from parent number two and the zeros show the bits taken from parent number one.

Example:
parent1 = 1000110011
parent2 = 0001111000
crosspoint $n = 4$
crossover mask = 1111000000
The resulting offspring will look like this:
offspring1 = 1000111000
offspring2 = 0001110011

As shown in the example, the crosspoint n breaks each parent in two parts. The two parents therefore give four peaces and the two offspring are constructed by recombining these four peaces into two distinct offspring.
A visual illustration of Single-point crossover is shown in figure 3.4.

2. *Two-point crossover.* Each time the Two-point crossover is applied, the crosspoints, n_0 and n_1, are chosen randomly. These crosspoints decide what the crossover mask will look like.

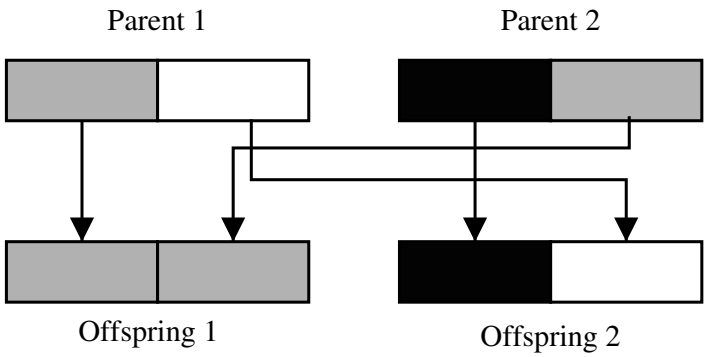


Figure 3.4: Single-point crossover

Example:
parent1:0001110000
parent2:1111000111
crosspoint $n_0 = 2$
crosspoint $n_1 = 7$
crossover mask = 0011111000
The resulting offspring will look like this:
offspring1 = 0011000000
offspring2 = 1101110111

As shown in the example, the crosspoints breaks each parent in three parts. The two parents therefore give six peaces and the two offspring are constructed by recombining these six peaces into two distinct offspring.
Figure 3.5 shows a visual illustration of Two-point crossover.

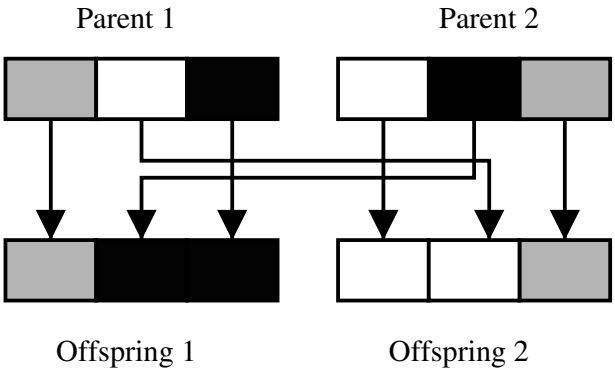


Figure 3.5: Two-point crossover

3. *Uniform crossover.* The crossover mask is generated as a random bit string, meaning each bit is chosen at random. The number of crosspoints is not specific as in the other two methods.

Example:
parent1 = 0001110000
parent2 = 1111000111
crossover mask = 1001101001
The resulting offspring will look like this:
offspring1 = 0111100110
offspring2 = 1001010001

A visual illustration of uniform crossover is shown in figure 3.6.

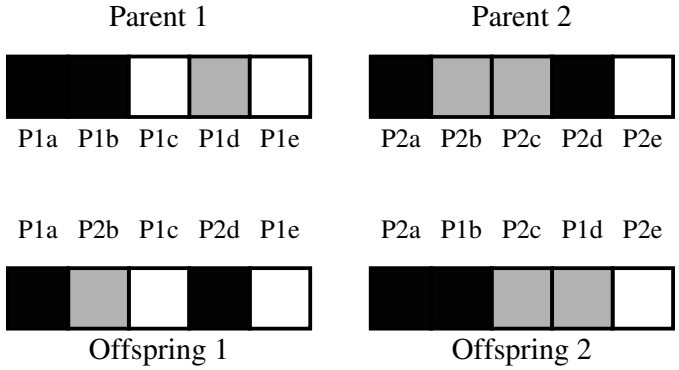


Figure 3.6: Uniform crossover where the crossover mask is 10101.

3.1.6 Mutation

Mutation creates a single offspring from a single parent by changing the value of a randomly chosen bit opposite crossover where you create two offspring from two parents using the crossover mask to determine which parent contributes which bits. Mutation is often performed after crossover has been applied. Mutation may lead to a significant improvement in fitness, but often it has a rather harmful results.
So the question of why one should use mutation at all arises. The reason is that we can use the mutation as a background operator. Its role is to provide a guarantee that the search algorithm is not trapped on a local optimum. The sequence of selection and crossover may stagnate at any homogeneous

set of solutions. Under such conditions, all individuals are identical and thus the average fitness of the population cannot be improved. However, the solution might appear to become optimal, or rather locally optimal, only because the search algorithm is not able to proceed any further.
Mutation is equivalent to a random search, and helps us in avoiding the loss of genetic variety. See figure 3.7 to see the connection between mutation and crossover in genetic algorithms. Mutation can occur at any bit, in an individual, with some probability. Genetic algorithms assure the continuous improvement of the average fitness of the population, and after a number of generations (typically several hundred) the population evolves to a almost-optimal solution.

Mutation methods

The mutation operator flips a randomly selected bit or bits in an individual to create one offspring. By doing this one might improve the individual by changing some bad property to a better one.

In the example below we have chosen to flip one bit.

Single bit mutation

Example:

In this example we have chosen that the mutation function should change the third bit in the parent bit string producing the offspring's bit string. A visual illustration of Single bit mutation is shown in figure 3.8.

3.1.7 Multiple bit mutation

In Multiple bit mutation one chose how many bits one like to mutate.

Gaussian mutation

Gaussian mutation consists of adding a random value from a Gaussian distribution to each element of an individual's vector to create a new offspring. Gaussian mutation can be applied when a real value encoding is used. In our project we will not be using Gaussian mutation because our individuals consist of bit strings, which are of course represented by ones and zeros.

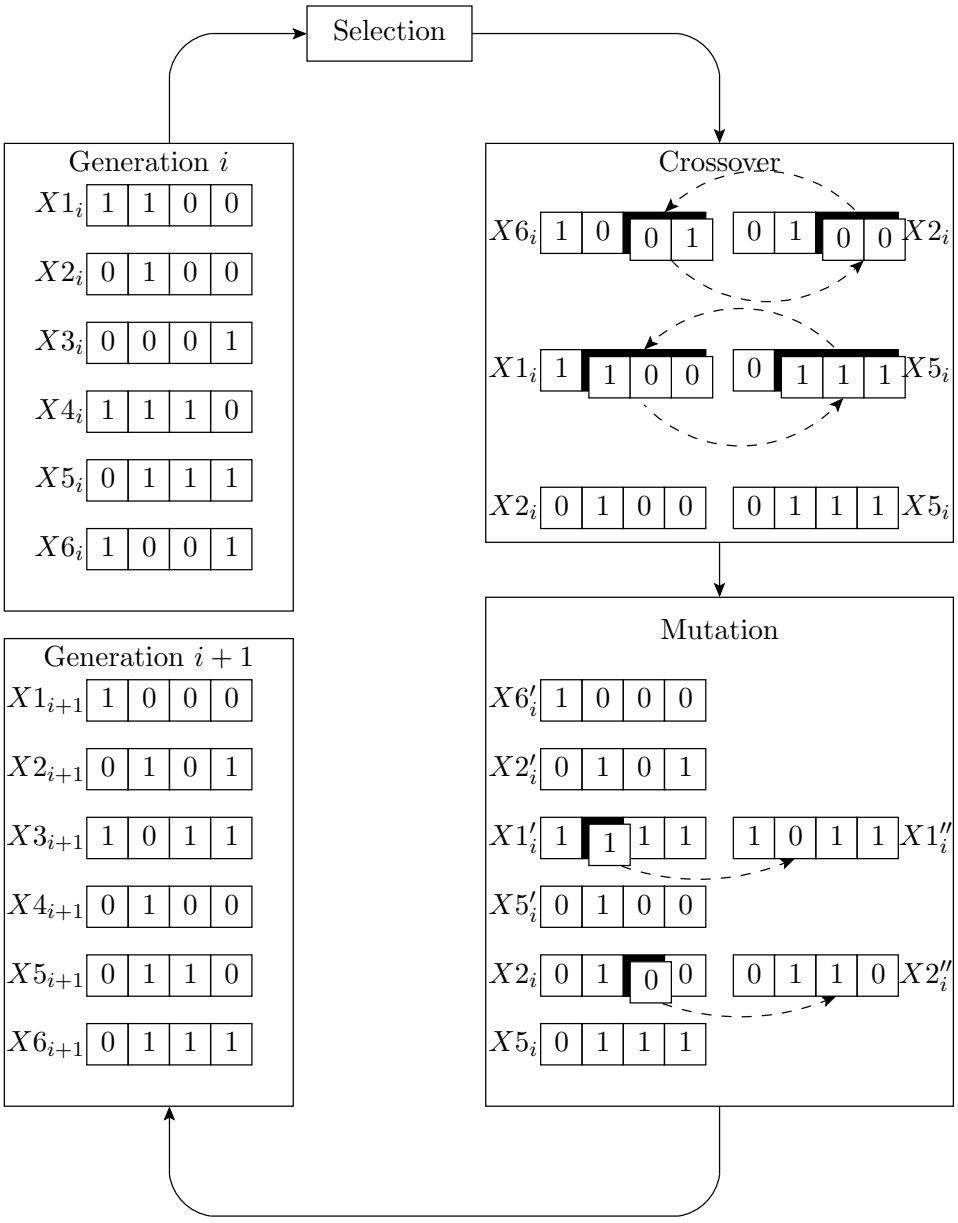


Figure 3.7: The genetic cycle

3.1.8 Local maximum and global maximum

Figure 3.9 (a) shows the initial location of the individuals on surface and contour plot of the fitness function, which is our starting point. Each individual is represented by a ball. The initial population consists of randomly

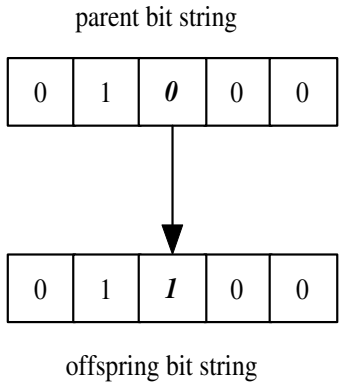


Figure 3.8: Single bit mutation where the third bit is flipped

generated individuals.

In the second generation crossover starts to recombine features of the individuals and the population starts to converge on the peak containing the maximum as shown in 3.9 (b). From then until the final generation the genetic algorithm is searching around this peak via mutation resulting in diversity. Figure 3.9 (b) shows the final generation.

However the population has converged on an individual lying on a local maximum, not a global one. But we are looking for at global maximum which has a higher peak. So can we be sure that we have found the optimal solution? One way to check if the optimal solution has been found is to compare the results obtained under different rates of mutation. To try to find the maximum of the fitness function we could increase the crossover probability and the mutation probability.
When creating a genetic algorithm you can specify how many generations you want to produce. You can also decide that the search algorithm should run for e.g. ten days to find the best solutions, or you could let the algorithm run until the best individual has not been improved in a number of generations. The population might now converge like the individuals in figure 3.9 (c).

In figure 3.9 (c) one can see that if we increase the mutation rate the fitness value will also rise. The mutation operator allows a genetic algorithm to explore a landscape in a random manner. Mutation may lead to improvements in the population fitness but more often the mutation decreases it. To ensure diversity and at the same time to reduce the harmful effect of mutation we can increase the size of the population.

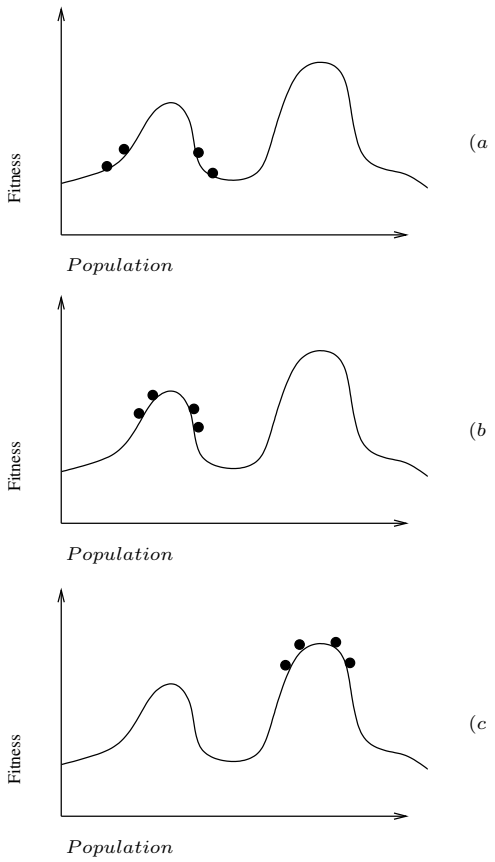


Figure 3.9: This figure shows some individuals' locations on the surface of the fitness function; (a) Initial location; (b) Individuals converging; (c) Global maximum solution

3.1.9 Elitism

Elitism is when the best individuals in the current population are being carried forward into the new population without modification.

Figure 3.10 shows a population with a size of one hundred. The individuals in the current population are sorted after fitness i.e. the fittest individual is number one and the least fit individual is number one hundred. Individuals number one and two are copied into the new population due to elitism, but selection, crossover and mutation are used to find the rest of the individuals i.e. the remaining ninety eight individuals. The individuals in the new population are not sorted by fitness.

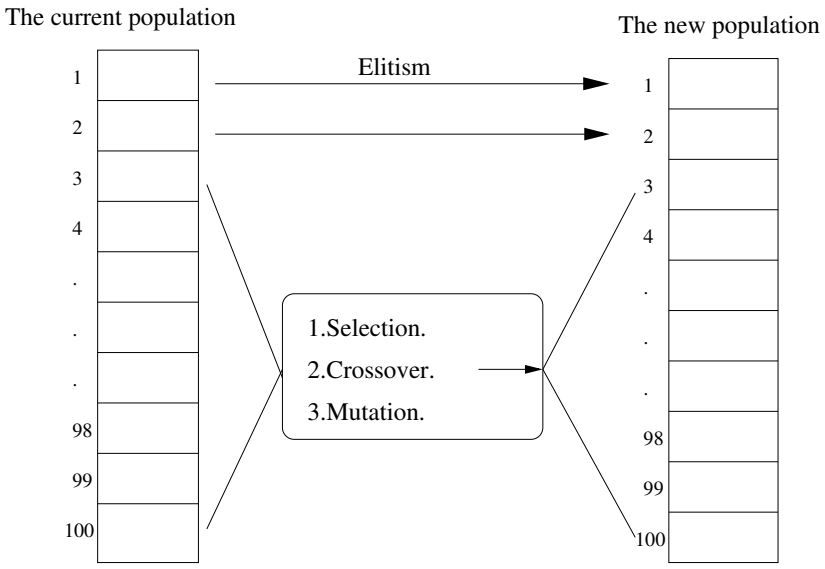


Figure 3.10: The current population and a new population

The advantages of elitism is that the most fit individual never changes when it is moved to the next generation i.e. the good genes are kept. This will result in the fact that the algorithm could find an optimum faster i.e. find a good solution (but maybe not the perfect solution which would be the global optimum).
If the algorithm is supposed to find the perfect solution then the disadvantages of elitism could be that the algorithm would find an optimum that is not a global optimum but a local optimum ("just" a good solution). This means that the evolution would halt and the global optimum would never be reached. As mentioned before, mutation could help to get more diversity in the population and the algorithm would then have the possibility of find another optimum.[7]

3.1.10 Crowding

Crowding is when some individual in the population is a superindividual, which means that it has very good fitness compared to the other individuals. This superindividual can therefore reproduce quicker than the other individuals in the population. This means that copies of this individual and very similar individuals will be a large fraction of the population. This will of course reduce the diversity of the population and slow down further progress

by the genetic algorithm[13].
Several strategies have been explored to reduce crowding:

- *Low selection-pressure:* When selecting individuals to be in the next generation, use either Rank selection or Tournament selection. By choosing one of these methods there is a higher chance for diversity in the population and low selection pressure i.e. the superindividual does not reproduce as quick as if the Roulette wheel selection was used.
- *Fitness sharing:* The measured fitness of an individual is reduced if other similar individuals are present in the population. This means that the other individuals (that are not similar) in the population have a higher chance of getting selected. This will increase the diversity in the population and the superindividual will not reproduce as quick as it else would.
- *Restricted mating:* Restrict the types of individuals allowed to recombine to form offspring. This could be done e.g. by allowing only the most similar individuals to recombine. This would cause many "subpopulations" in the population. The best fit individual in each subpopulation can be a superindividual in its own subpopulation. In a population with crowding one local (or global) optimum would be found, but when using subpopulations, many optimums would be found. One of these optimums could be the global optimum.

Chapter 4

The ant breed

When creating a winning ANT you have to invent a winning strategy. Many ANTs have been developed over a long periode of time and many of the players in the `MyreKrig` league have a lot of experience in making winning strategies. We have therefore investigated these strategies to find out what seems a good idea and what does not. The following section describes the ANT's skeleton and the strategies that lie behind it.

4.1 A winning strategy?

The first strategy was to create an ANT, which moves in a star shape pattern. This pattern can be seen in figure 4.1. This is an actual screenshot from `MyreKrig`. Here the white dots are ants, and the gray is the area visited by the ants.

At first this pattern seemed a good ide, because we could cover a large area fast. This, however, quickly proved to be a bad idea. Moving the ANT in our star shaped pattern required some random values, and these random values made it harder for us to find the fitness of each individual. How does one define and control good and bad random values? As a consequence we then tried to predefine some fixed ways of moving. This on the other hand made the ANT perform badly and it started to move in the same paths all the time.

The star pattern was fast abandoned as a movement pattern and we looked for another way of moving the ANT.

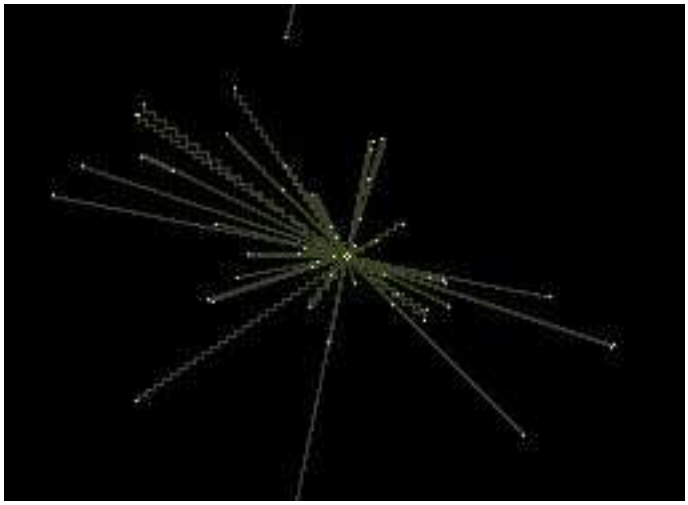


Figure 4.1: The star shaped pattern from the first strategy

4.2 An improved strategy

Because the other strategy had a lot of random values, we decided to make a new strategy with no random values. We therefore had to have a fixed movement pattern. The strategy is first of all to find food by sending out ants, who take on the role of explorers, in a fixed pattern. When an explorer ant finds a resource of food it then takes on the role of a helper, which moves the food to one of the axes north/south or east/west, depending on which is closer to the ant. Then other ants, acting as carriers, carry the food back to its home base, hereby distributing the work between the ants and thereby getting the food back to the base faster.

Although each ant only gets to move once and only moves one square each turn, it is possible to move food more than one square each turn. Some luck is required for it to happen. It is possible if an ant carrying food moves to a square with another friendly ant, which has not moved yet, because the other ant can then move the food even further during its turn (this is showed in figure 4.2). In fact, it is possible to move one piece of food from the food resource to the base in one turn.

The carriers move in straight lines going north/south or east/west, and the explorers move in a pattern between the axes' straight lines. Figure 4.3 shows the three different types of ants: explorer (described in section 4.2.6), helper (described in section 4.2.7) and carrier (described in section 4.2.8).

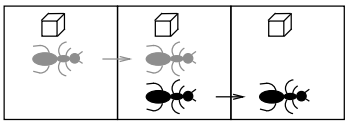


Figure 4.2: An example of moving food more than one square in one turn.

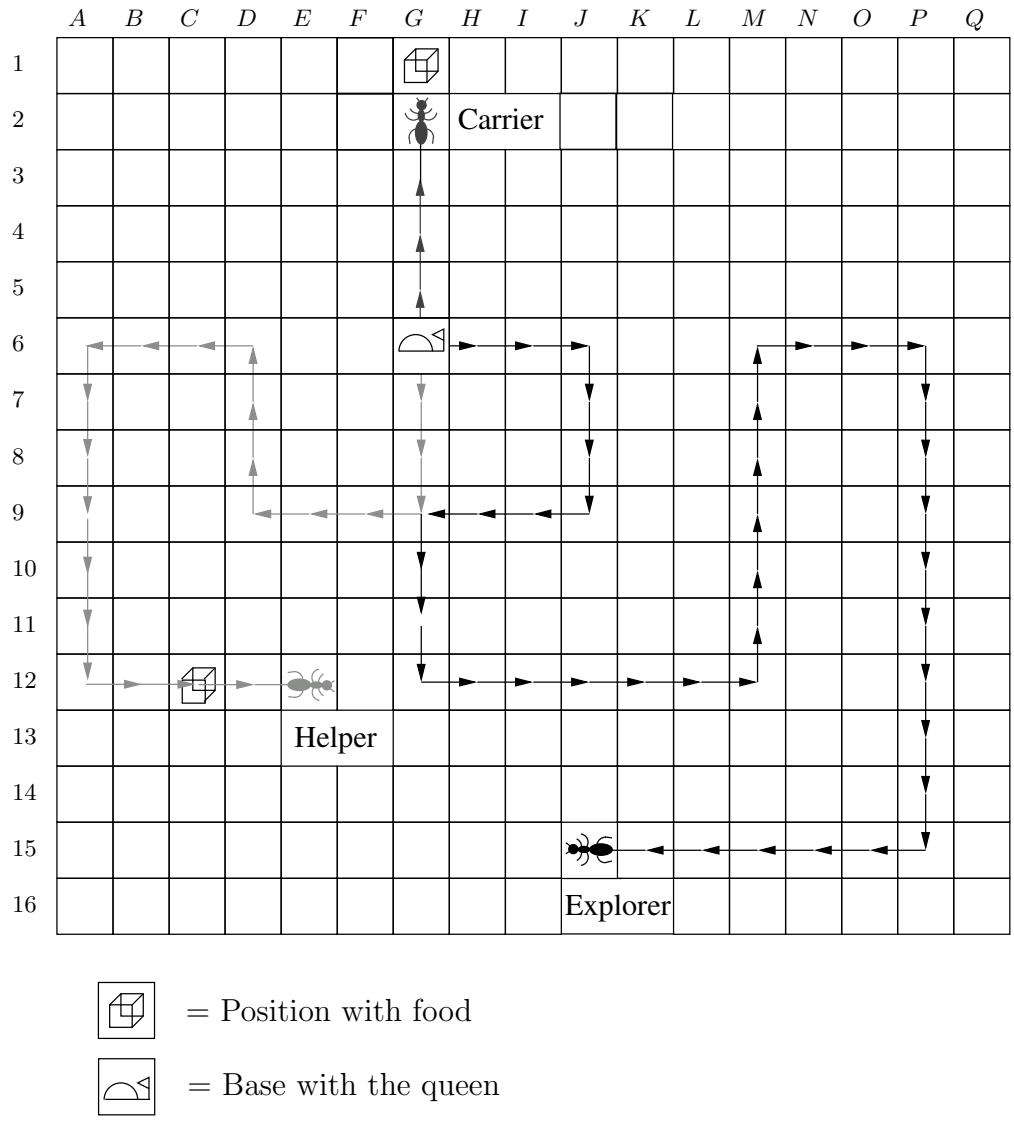


Figure 4.3: Example of strategy.

4.2.1 The ants brain

An important part of the ANT is its brain. When an ant is born, it is given a specific role and this role tells the ant what to do. The first ant to move turns itself into a queen, and the rest are then simply ants waiting for a role given by the queen.

The ant remembers its relative position on the map to its base. As it uses an `xy` coordinate system, this is fairly easy as the base is placed at `(0,0)`. When it moves around the map, it updates this coordinate. When it moves around and finds some food, it also remembers that location and how much food is present here, thereby making it possible to tell other ants about the location of the food.

4.2.2 A strategy for attacking other ants

The attack strategy is simple. Each time an ant sees an enemy ant, it kills it. Then it must wait for `x` turns, where `x` is defined in its bit string (described in section 4.2.4). The size of `x` is modelled using the genetic algorithm and is a number between `0` and `127`. It seems to be a good idea to wait for `x` turns, because the probability of another enemy ant showing up is fairly big. But as we use the genetic algorithm and as `x` can be `0`, we have no influence on whether the ant should wait or not. If the ant has waited `x` turns and it does not see another ant, it simply falls back to the routine it had before it saw the enemy. If, on the other hand, it sees another enemy while waiting, it attacks again and resets the wait counter, so that is starts to wait again.

4.2.3 Memory usage

As described in section 2.4.5, ANTs are also ranked by how much memory their brains use. We have desided to look away from this aspect of the game and simply let the ANT use as much memory as it may need. Optimizing the ANT's use of memory could be a task in potential future development of the ANT (read more about this in chapter 8).

4.2.4 Identifying the bit string

The bit string representing an individual in our program is defined by 46 bits. This structure is divided into nine different genes as follows:

Gene	Bits	Range
step1	3	0 - 7
step2	3	0 - 7
step3	3	0 - 7
initial_max_range	7	0 - 127
stepRange	5	0 - 31
stepCount	6	0 - 63
turnCountEXPLORER	6	0 - 63
turnCountCARRIER	6	0 - 63
turnCountKillWait	7	0 - 127

Table 4.1: Overview of the bit string

step1-3 describe how far an ant walks in a straight line before it changes direction. This fixed pattern is described in the explorer section 4.2.6.

initial_max_range is how many steps a carrier or explorer will move away from the base before turning around. Note that this is only the initial number of steps. This value will be modified by stepRange and stepCount while running.

stepRange is the number added to initial_max_range to make an ant expand its search pattern.

stepCount is the number of turns to wait before adding stepRange to the initial_max_range hereby letting the ants move longer away from the base.

turnCountEXPLORER is the number of turns the queen waits until she sends out new explorers (explorer counter).

turnCountCARRIER is the number of turns the queen waits until she sends out new carriers (carrier counter).

turnCountKillWait is the number of turns times 10 an ant will wait after it has killed an enemy.

4.2.5 The queen

When the first ant from the currently used breed is selected to move by MyreKrig it will make itself a queen. All other ants will be initialized as ants waiting to receive orders from the queen. The queen is the coordinator of work and is used to assign all the other ants their roles.

Each turn the queen gets the chance to assign roles to the ants waiting. The

queen keeps track of the number of turns that have passed using an internal counter. Whenever the internal counter is dividable by the explorer counter (described in table 4.1), the queen makes four new explorers (if four ants are present on the base) and assigns them a direction, so they do not go the same way. This also happens when the carrier counter (also described in table 4.1) becomes dividable by the internal counter, only here the queen sends out carriers instead of explorers. If a carrier returns home with food, it tells the queen where it found the food, so the queen can deploy new carriers to collect it.

The new explorers and carriers are simply created from waiting ants by setting their memory to either explorer or carrier and assigning them their direction. To ensure that the newly deployed ants do not move in the same direction (i.e. all explorers or carriers go east) the ants four assigned as explorers or carriers will be assigned to go east, south, west and the last going north. Then the next time explorers or carriers are assigned, the first ant will go south, the next west, then north and the last east. This is done because we cannot ensure that four waiting ants are present, but maybe only one or two.

The queen does not move. Not even when she sees an enemy. The reasonning behind this decision is that the queen holds the essential knowledge about where to send carriers for food, and it is assumed that there are always friendly carriers or explorers nearby to kill the intruder. If not, we assume that we have lost so many ants that we will lose the game anyway and resistance is therefore futile.

4.2.6 The explorer

The role of the explorer is essential for the expansion and thereby survival of the breed, since expansion is the only way to survive. The explorer moves after a predefined pattern as mentionted in section 4.2. Figure 4.4 displays this movement pattern.

In table 4.1 we identified three genes, `step1`, `step2` and `step3`. These are used by the explorers. Each explorer is assigned one of these values as its step count (see figure 4.4). In this example the explorer is assigned the value in `step1`, which is set to three. This means that the explorer moves three steps to the east, then turns south moving three step, turning west moving three steps, turning south again and moving three steps, turning east and this time moving six steps to make the desired pattern (as shown in figure 4.4). This extra movement is simply done by having an extra counter, which

is initialized to twice the value of `step1`. This means that in our example it will be six. Then when the ant has used this counter it simply adds `step1` to it (making it nine in our example).

This pattern repeats itself from the point where the ant is now, hence making the explorer move further and further away from the base.

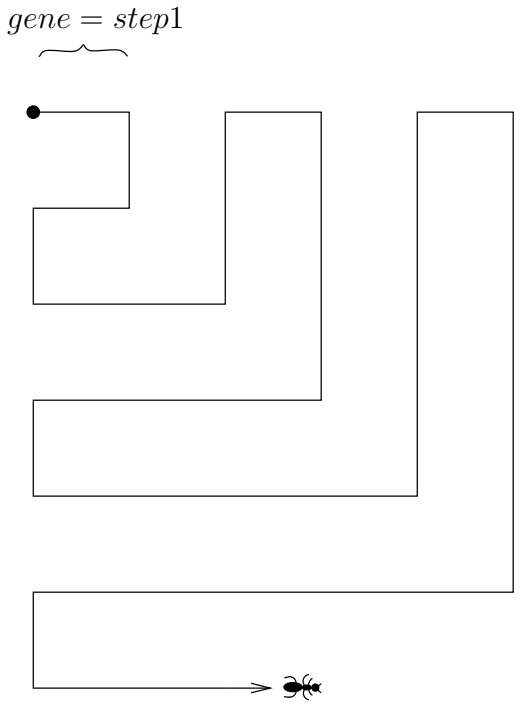


Figure 4.4: The basic movement pattern of this strategy.

4.2.7 The helper

When an explorer finds food its role is changed to the helper role. The only purpose of the helper is to transport the food found in the field to the nearest north/south or east/west axis. When it finishes transporting all the food to this axis, it again changes role. This time to a carrier, and begins moving food from the axis back to the base (this is described in section 4.2.8).

As the helper moves food to the axis it remembers how many pieces of food are still left at the location of the food resource. If this is greater than `0`, it returns to that location to retrieve more food. If it, on the other hand, comes to that location and all the food is gone, it simply becomes a carrier,

and goes back to the axis to move food on the axis back to its base.

4.2.8 The carrier

The role of the carrier is as mentioned to carry food back to the base. This is its main function, but it also guards the north/south and east/west axis. As the number of ants grows, so does the number of carriers. It therefore becomes easier to protect these axes due to a huge number of ants moving to and from the base along these axes.

As mentioned, it is the queen who initializes the carrier. Each carrier is given a direction, and this could for instance be east. The carrier will then move east all the time, unless it finds food (placed randomly by `MyreKrig` on the axis or by a helper) or if it reaches `initial_max_range` as defined in the bit string. If it reaches `initial_max_range`, then it must turn around and go home and its direction then becomes west. If it finds food, it will take a piece and go back to the base. When it returns to the base it tells the queen where it found the food and how much food was there.

Chapter 5

The system

In the system, one generation contains several ant breeds. Each ant breed consists of the skeleton presented in chapter 4 and an individual i.e. a bit string which describes the properties of a breed.
Given the facts in section 4.2.4 we notice that an individual is defined by 46 bits. This gives us 2^{46} possibilities for creating breeds. This chapter explains how we search this vast space of candidates in order to find the better breeds.

The purpose of the system is to start the game `MyreKrig`, run the genetic algorithm and link `MyreKrig` and the genetic algorithm together. The system is the back-bone of the entire evolutionary process of the ants. Every ant breed is evaluated by `MyreKrig` by testing it against various opponents under various conditions. Once this is done the system uses the rules of genetic algorithms, presented in section 3.1, to find the new individuals by modifying some or all of the individuals in the current generation, to create individuals for a new generation. This is done iteratively by using the output from the genetic algorithm as input to `MyreKrig`. The output from `MyreKrig` is written to a file and is then used as input to the genetic algorithm.

5.1 Design of the system

In order to better understand how the system operates, an *evolutionary* half (see section 5.1.1) and a *linking* half (see section 5.1.2) was developed. The purpose of the first half is to use an evaluated generation to generate a new population. The purpose of the last half is to link `MyreKrig` and genetic algorithms together and decide if a certain termination criteria has been

reached.

5.1.1 The evolutionary half

The evolutionary part applies the rules of the genetic algorithm to a generation of individuals. This part was programmed in Java[1], because with a modular build it is easy to replace components and maintain the program. This part of the system can therefore be applied to other problems requiring development by genetic algorithms.

The evolutionary part works by using an input in the form of individuals (bit strings) and statistics (which come from `MyreKrig`). The input is sent to a fitness function (*Fitness* in figure 5.1) and then some individuals are added to the new generation unmodified (*Elitism* in figure 5.1). When finished selecting individuals (*Selection* in figure 5.1), crossover is applied using parents from the current population. The two offspring are added to the new population (*Crossover* in figure 5.1). The system applies a mutation function on a small number of individuals in the new population with some probability. (*Mutation* in figure 5.1). This means that mutation might not even take place if the probability is too low. Finally a few random individuals (*Random* in figure 5.1) are added to the new population instead of the least fit individuals, to prevent premature convergence to a local maximum[19]. The individuals in the resulting generation, are written into a file. This file is read by the linking part of the system.

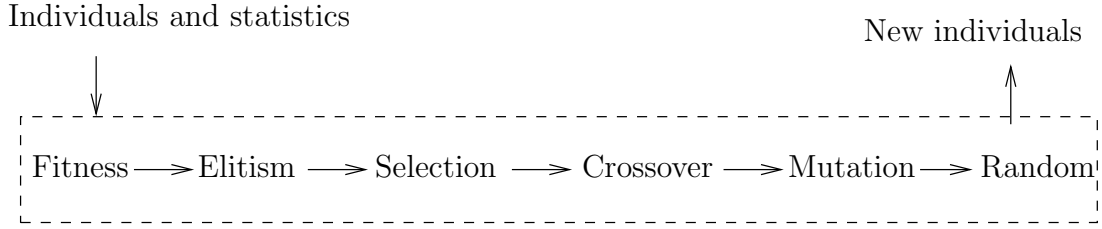


Figure 5.1: The working of the evolutionary part of the system is shown with a dotted line.

[1]For more information about Java visit *http://java.sun.com*

5.1.2 The linking half

The linking part of the system was implemented using perl[2]. Perl was chosen because its primary advantage is simple manipulation of files, and good incorporation with the operating system (Linux/SunOS). This part of the system works by first obtaining the individuals of the new generation, and then obtaining a file with the skeleton. An ant breed is created by combining the skeleton with the individuals from the new generation (*Ant* in figure 5.2). The created ant breeds are compiled into `MyreKrig` and a number of games are run (*Compile* ***MyreKrig*** and *Run* ***MyreKrig*** in figure 5.2). The results output by `MyreKrig` in the form of statistics are caught (*Catch Output* in figure 5.2). The statistics along with the individuals are then written to a file. When an entire generation has been evaluated by `MyreKrig`, the evolutionary part is activated thus allowing the next generation to be developed, using the information of the current generation.

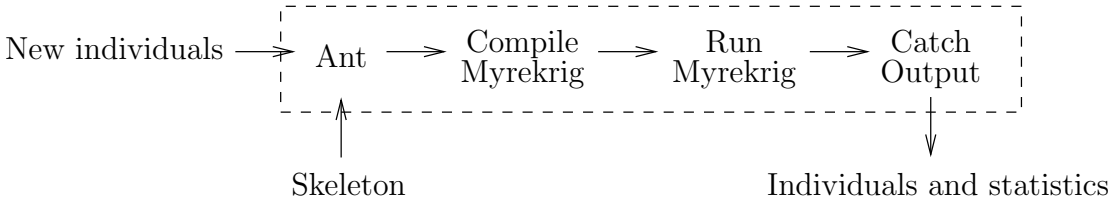


Figure 5.2: The linking part of the system is shown with a dotted line.

5.1.3 The information flow between the linking and the evolutionary halfs

The entire system starts by creating a random generation of individuals and running them through the linking part of the system. The output from the linking part is used as an input to the evolutionary part. The output from the evolutionary part is used as an input to the linking part of the system. The system thus works iteratively, as shown in figure 5.3.

[2]For more information about perl visit *http://www.perl.org*

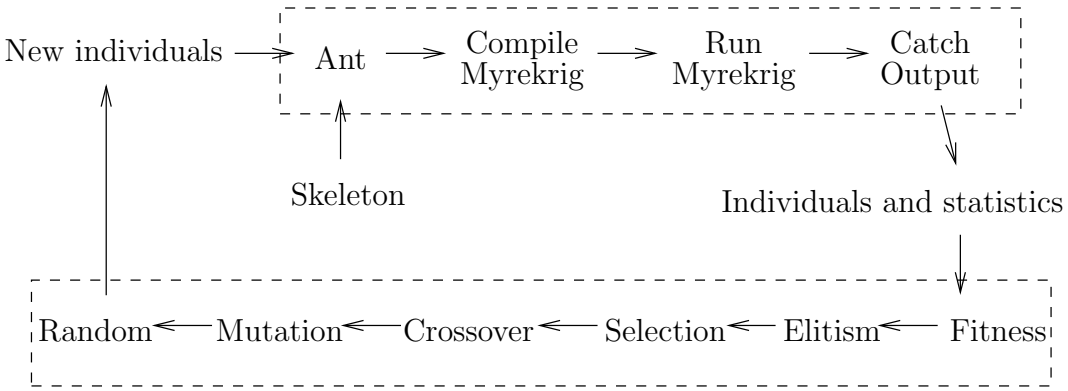


Figure 5.3: The linking part and the evolutionary part of the system are joined iteratively.

5.2 Parameters for the system

The system is structured such that various parameters can be given, and are thereafter used for tuning the system. Common elements for both parts of the system are:

- The number of bits defining an individual.
- The number of individuals in each generation.

The genetic algorithm runs until a termination criterion is satisfied. An example of a termination criterion is when a maximum number of generations has been developed. Other examples could be a given fitness threshold or convergence of the population.

The linking half of the system contains the parameters for the game `MyreKrig`. The two most important parameters are:

- The number of games in which to evaluate each ant breed in `MyreKrig`.
- The opposing teams to test the ant breed against.

The linking half also contains the termination criterion (e.g. the number of generations to develop).

The evolutionary half of the system contains all the parameters for the evolution:

- The function to use for evaluating individuals (fitness function).
- The number of individuals selected due to elitism.
- The method used for selection (Roulette wheel-, Rank- or Tournament selection).
- Crossover probability and method (One-point-, Two-point- or Uniform crossover).
- Mutation probability and method (Single-bit mutation).
- The number of random individuals to insert.

5.3 Implementation of the evolutionary part

The linking part of the system writes a statistical output from `MyreKrig` and the individuals to a file. This file is read by the evolutionary part. The evolutionary part finds the fitness of each individual, uses elitism, selects indviduals to mate, and uses the crossover and mutation operators.

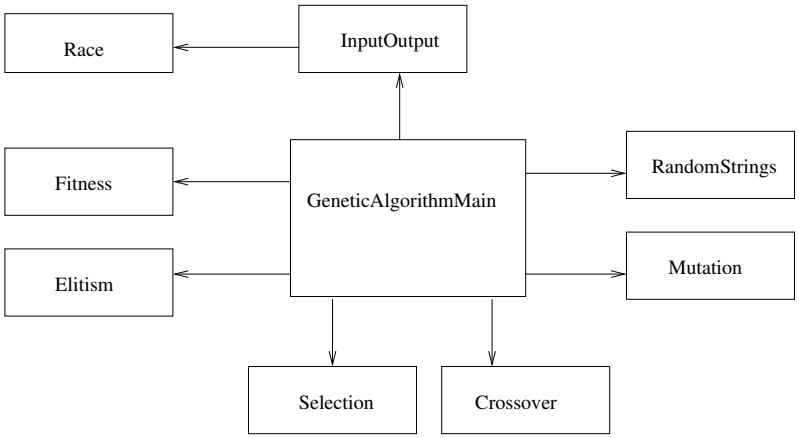


Figure 5.4: A class diagram.

Figure 5.4 shows the classes that were implemented in the evolutionary part.

5.3.1 GeneticAlgorithmMain

GeneticAlgorithmMain is the main class and runs the genetic algorithm.

5.3.2 Race

Race is a class which encapsulates the data: *bit string (individual), victory value, performance value and overall performance value.* Everytime a bit string is inputed into the the system an object of the class Race is created. All the objects of the type *Race* are then saved in a vector[3].

5.3.3 Fitness

The fitness function used to find the fitness of the individuals is: Overall Performance = $\frac{Performance*Prestige}{victory}$ (from chapter 2). Overall performance was used as fitness function because it describes really well how the individuals are performing in `MyreKrig`.
When the fitness of all the individuals have been calculated, the individuals are sorted according to fitness.
The information (bit string, performance, prestige, victory and fitness) about all the individuals are written to a statistic file. When finished writing to the file, a number (the number is a constant e.g. 4) of individuals, which have the lowest fitness, are removed from the vector. The rest of the individuals continue to *Elitism*.

5.3.4 Elitism

By using the rate of elitism (e.g. 20%) it is found out how many individuals to copy unmodified to the next generation. Because the individuals are sorted according to fitness in the vector, elitism is applied by simply copying a number of individuals from the top of the vector to another vector. This new vector is then used later in the classes *Selection*, *Crossover* and *Mutation*.

5.3.5 Selection

All three selection methods (roulette wheel-, rank- and tournament selection) have been implemented in the *Selection* class.
The fitness of an individual is used to calculate the probability for the individual to get selected. Figure 5.5 shows how the probabilities for all the

[3]The name of a data structure in Java

individuals are put in an array.

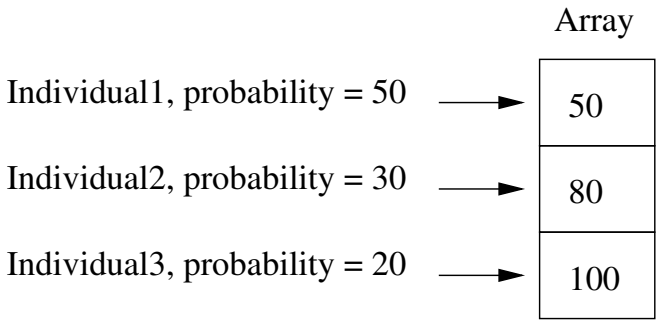


Figure 5.5: The individuals and their probabilities. If the random number is e.g. 67, individual2 is selected ($50 < 67 <= 80$).

A random number, between 0 and 100, is generated and then the individual, that this number belongs to, is selected and put in a vector. The vector will eventually contain all the selected individuals, and it is then used in the class *Crossover*.

5.3.6 Crossover

Two crossover methods single-point crossover and two-point crossover are implemented in the *Crossover* class. Crossover is not used on the individuals from the class *Elitism*, but it is used on some of the other indviduals depending on some crossover probability each individual is given. Two individuals are used at a time. When crossover is applied to two individuals the two offspring that are created with those two parent individuals are put into a vector. If crossover is not applied then the two individuals are put into the vector unmodified. Which parts of a pair of parent individuals are used to create the offspring is dependant on one or more crosspoints, defining where the bit strings are seperated.
The crosspoint in details: A random number is generated and this number is then used as a crosspoint. The crosspoint can cut the gene of an individual, thereby destroying the information in the gene, or it can cut between the genes (see figure 5.6), thereby preserving the genes' information. We use the technique which allows cutting the genes because it decreases the probability that the same genes will always continue to the next generations. Mutation probability is very low, so one should not count on mutation alone to create

diversity in the bit strings. In other words, it ensures more diversity in the population.

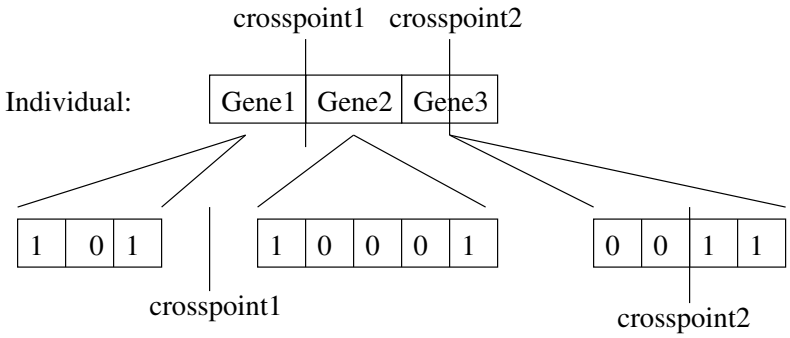


Figure 5.6: Two-point crossover. Crosspoint1(random number = 3) cuts between genes and crosspoint2 (random number = 10) cuts a gene.

5.3.7 Mutation

In our project we have chosen to use single bit mutation. The single bit mutation happens in the *Mutation* class. The individuals chosen in *Elitism* are not mutated. It is only the rest of the individuals that are mutated with a probability. One individual is mutated at a time. If mutation is used then one bit in the parent bit string is flipped to form a new bit string - the offspring bit string. This new bit string is then replaced with the parent bit string in the vector.
Single bit mutation in details: A random number is generated. This number is between one and the length of the bit string. It defines which position should be flipped in the bit string. The offspring bit string is, as mentioned above replaced with the parent bit string in the vector.

5.3.8 RandomStrings

A method in the class *RandomStrings* gets the number of removed strings (mentioned in the section 5.3.3) and a vector, containing the individuals from *Mutation*. The method creates some random individuals which replace the removed individuals. The method then returns the final vector which contains the new generation.

The new generation is written to a file, by the *InputOutput* class, which is read by the linking part of the system.

5.3.9 InputOutput

The *InputOutput* class is a class that reads and writes data to and from the linking part. The class also writes statistical data to a file, and that data is used to produce graphs which show how the fitness envolves.

5.3.10 The flow

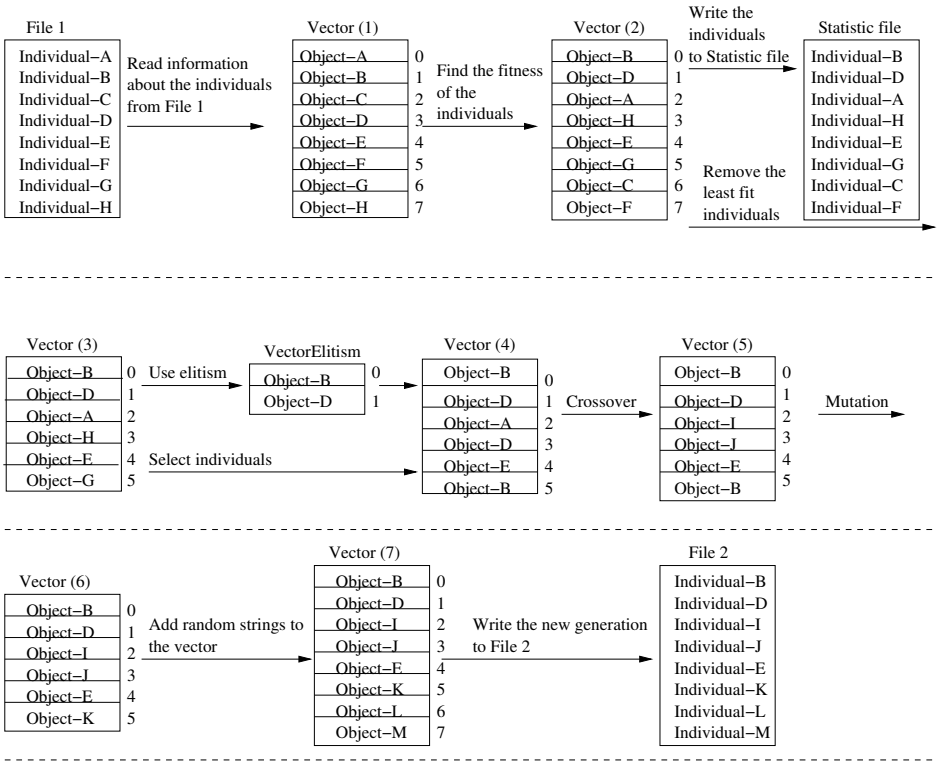


Figure 5.7: The flow of the genetic algorithm in the program (an example with eigth individuals).

Figure 5.7 shows how the flow of the genetic algorithm *could* be in the program, when the generation consists of eight individuals. This figure is based on to figure 3.7 in section 3.1.

The example in details:
The vectors vector (1) - vector (7) describe the same vector (with different values).

- *File 1* is a file that contains the individuals and their statistics from `MyreKrig`.
- *Vector (1)* contains objects from the *Race* class. The objects encapsulate the information about all the individuals.
- *Vector (2)* contains the same objects as vector (1). The only difference is that the fitness of each individual have been added to the information of the individuals (the fitness was zero before it was changed). The elements in the vector have also been sorted according to fitness.
- *Statistics file* contains the statistical information about the individuals.
- *Vector (3)* shows what the vector looks like when the two least fit individuals, Object-C and Object-F, have been removed from the vector.
- *VectorElitism* is a new vector which contains the two most fit individuals i.e. Object-B and Object-D.
- *Vector (4)* contains the objects from VectorElitism and the selected objects.
- *Vector (5)* shows the vector when crossover has been applied. Object-A and Object-D (position two and three in vector (4)) have been removed and the vector instead contains their offspring, object-I and Object-J.
- *Vector (6)* does not contain Object-B in position five anymore since the individual in this object was mutated. Object-B was replaced with Object-K.
- *Vector (7)* now contains eight objects again. The new objects, Object-L and Object-M, were created randomly. This vector contains a new generation.
- *File 2* contains the individuals making up the new generation.

Chapter 6

Data processing

In order to evaluate the individuals a series of evolutions were run by the system in chapter 5.
The parameters for each evolution were: the rate of elitism, selection method, crossover method, crossover probability, mutation method, mutation probability and a number of individuals to remove. A base evolution was made (section 6.2) to make it possible to compare the evolutions, i.e. for each evolution, one of the parameters was changed compared to the base, in order to estimate how the different parameters affect the outcome.
To be able to compare the different parameters above, it also had to be defined how many generations to develop, population size, how many games each individual should play, the number of opponents, how strong the opponents should be, and the maximum number of turns.

6.1 Noise

Most of these parameters are here because of the great amount of noise `MyreKrig` produced on the output, mostly because of the many random factors, we would have to extend our search area. If our ANT plays on many small playing field it would win more than on larger fields, so if one individual is lucky it will win a lot more than the exact same individual from a later generation.

6.2 Parameters

The parameters of the system run are shown in table 6.1. A genetic algorithm is typically iterated for anywhere from fifty to five hundred or more generations [12]. We only use twenty generations. We have chosen to use a population size of fifty, which is in the lower part of what is considered typical values [12]. The number of battles for each bit string has been set to thirty and the number of turns run by `MyreKrig` is set to ten thousand. These parameters are a result of searching for a fast running ANT, but with parameters that are usable for our fitness function.

Mutation

Inside the genetic algorithm, mutation is defined using a single bit mutation with a 2% chance of mutation happening. Here a typical value is in the range of 0.01% and 1% [15]. The reason why we used 2% chance of mutating is that this value statistically will mutate one bit string for every generation ($2 * 50/100 = 1$). If we instead used a probability of e.g. 0,2 % a bit string statistically will mutate every 10th generation ($(0.2*50/100)*10 = 1$), which we consider to be too seldom. First single bit mutation seemed to be a good idea, but we discovered early that it did not have any greater impact on the bit string, due to our genes' ranges being chosen with care. The only gene which could have an impact is `initial_max_range` (see section 4.1). But this value will be between 0 and 127, meaning that the ANT will start running 0 squares away from the base or at most 127 squares away from the base.
As we discovered, a mutation value of 2% was not enough, as we only have fifty individuals in each generation. Therefore we introduced four randomly generated individuals into each generation. They replace the four least fit individuals. The reason for this is to avoid drawbacks of our small population by applying a 100% mutation rate on these four individuals [19].

Crossover

Crossover has been implemented using 70% as our base value. To examine the effects of various degrees of crossover we have run some evolutions with 50%, 60%, 70%, 80% and 90% probability of crossover. We have only implemented single point crossover, and hereby we converge faster than if we have used two point crossover. Using two point crossover means, that the new individual would be more different than its parents hence making the search area larger.

Parameters	Values
Number of generations	20
Population size	50
Number of battles	30
Turns	10000
Opponents	Cascade, InformAnt, Equalizer and Turbo
Mutation operator	Single bit mutation
Random operator	4
Crossover operator	Single point crossover
Selection operator	Rank selection
Individual length	46 bits

Table 6.1: Overview of different parameters used during the evolution

Selection

Selection has been implemeted using rank selection. We could also have chosen to use roulette selection. This on the other hand could result in crowding and this means that the best individual would be too dominating. We have also considered to use tournament selection. The reason why we have not used this selection method is because it converges faster than rank selection. We believe that rank selection will create more variation in the population thus increasing the search space to reach the global maximum.

Elitism

We chose to set our base probability of elitism to 20%, which is the most commonly used probability with elitism [19]. As with crossover we have also run evolutions with other values to be able to interpret on the results of these degrees. This might lead us to find the optimal degree for our program. Beyond using 20% we also tried using 10% and 30%.

The length of the individual

The length of the individual is 46 bits. In order to change this value, we must reimplement the ant, as this value is defined within the ant.

Tabel 6.2 shows the different parameters we have chosen to use when running different series of evolution.

Crossover probability	Mutation probability	Elitism	Random
50%	2%	20%	4
60%	2%	20%	4
70%	**2%**	**20%**	**4**
80%	2%	20%	4
90%	2%	20%	4
70%	2%	10%	4
70%	2%	30%	4

Table 6.2: Overview of different sets of parameters used during the evolution. The emphasized parameters are from the base evolution

6.3 Elitism

The graph in figure 6.1 shows the evolutions of three generations, one with a 10% elitism, one with 20% elitism and one with 30% elitism.

There is a little difference between the three evolutions within the first three generations because all three generations started with random generated individuals. The result is that the individuals in these generations are *relatively* less fit than the individuals in the later generations. When the generations evolve the average fitness gets better because the best individuals get the biggest chance to reproduce, i.e. the number of good individuals increases.

The evolution of the generation with 30% elitism, has the best average fitness of the three evolutions. The reason for this is probably that more individuals (than in the generations with 10% and 20% elitism), with good fitness, are moved to next generation unmodified, i.e. more good genes are kept. The evolution for the generation with 10% elitism has the worst average fitness (compared to the evolutions of the generations with 20% and 30% elitism) because it does not keep as many good genes as the other two. If only viewing the graph with 30% elitism it looks like this is about to converge.

Finally the evolutions for the three generations show that the breeds become more fit as time goes by.

6.4 Crossover

In figures 6.2 and 6.3 five evolutions are displayed with different degrees of crossover. These evolutions range from 50% in crossover to 90%, hereby

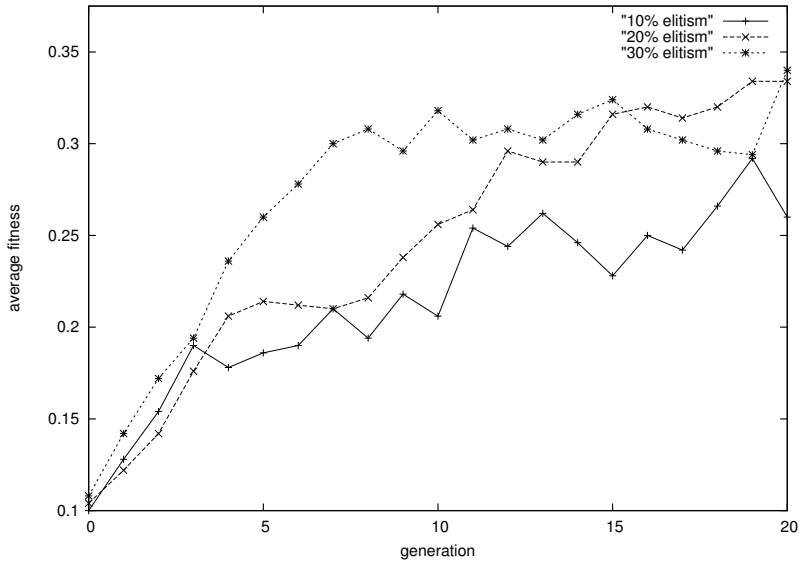


Figure 6.1: Average for various values of elitism

covering a vast range of crossover.

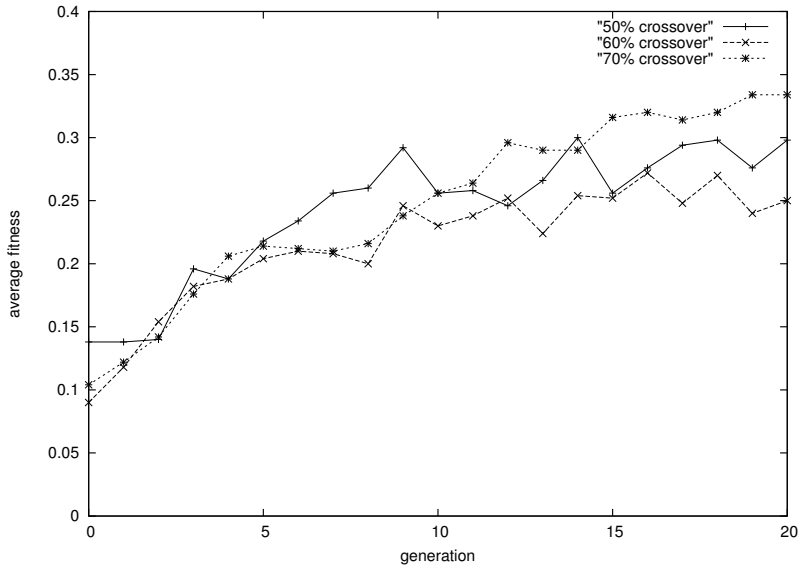


Figure 6.2: Average for various values of crossover (part 1/2)

We can say from this that the low crossover values will have a broader search range because the graph fluctuates more, and is trying many more differ-

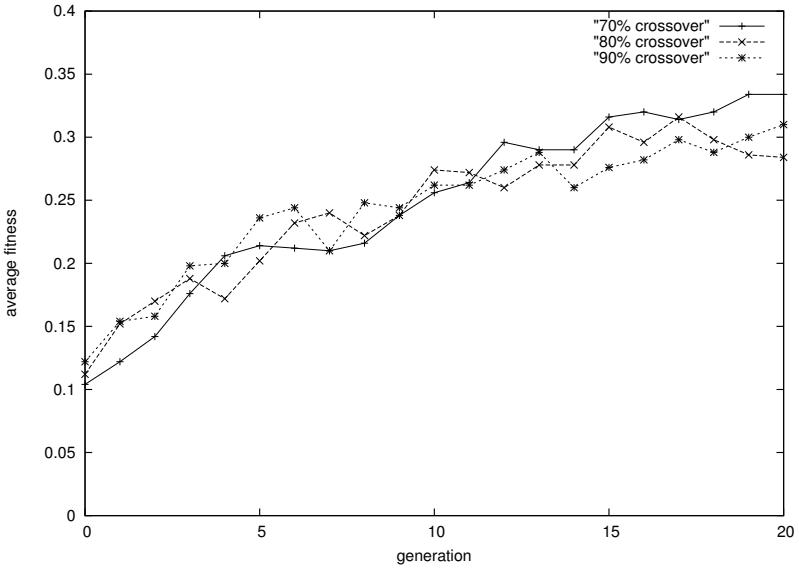


Figure 6.3: Average for various values of crossover (part 2/2)

ent bit strings. The high values will mostly use the "good" bit strings for crossover and will therefore converge faster.

6.5 Converging the bit strings

To sum up, we have to compare the bit strings returned after the different runs, and then conclude if our strings have converged. We have defined that if half the strings are alike, then we can see a convergence between the individuals. In graph 6.4 the number of similar individuals in various generations is shown. The graph illustrates that it is not possible to exactly conclude anything, unless several more generations are run.

Like graph 6.4 a graph is made with elitism, and this is shown as graph 6.5. From this we can see that a high value for elitism results in a very fast convergence. On the opposite side a low value will result in a very slow convergence, however, this searches a bigger search space.

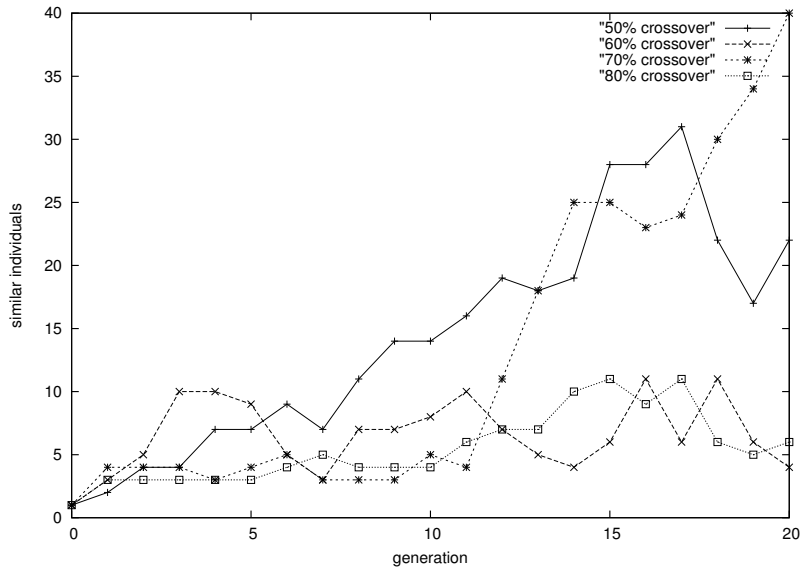


Figure 6.4: Similar individuals in various generations(crossover)

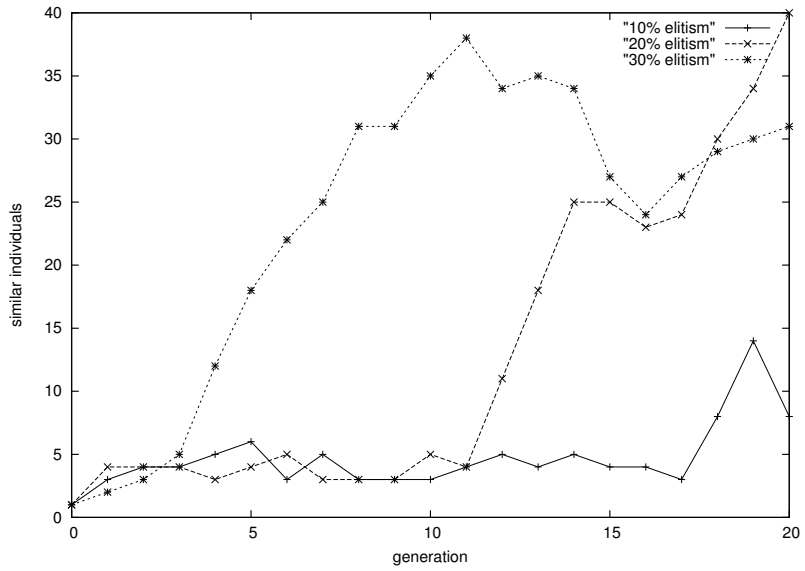


Figure 6.5: Similar individuals in various generations(elitism)

Chapter 7

Conclusion

The idea behind this project was to create an ANT for `MyreKrig` and show that genetic algorithms can be used to optimize this autonomous ANT agent.

7.1 Breed and the chosen strategy

The chosen strategy behind the ANT was not as good as we could have hoped. At first the pattern in which the explorer (section 4.2.6) moved showed promising features. This on the other hand proved to be a good idea only in the begining of the game, but as the turn count made its way past one thousand, it proved to be a bad idea. This is due to new explorers moving in the same path as the first explorer sent out and hereby searching for food where an explorer recently had been.

On the other hand the idea with the helper (section 4.2.7) was good. Its function worked very well. Also the carrier (section 4.2.8) was a good idea. It quickly found the food placed by the helper, and the breed therefore grew rapidly. This was still early in a game of course. The breed was made without the use of guards, which also was an idea that proved to be fairly good. Only fairly though, because when one ant (explorer, helper or carrier) saw an enemy ant it attacked it and waited for a while, thereby acting as a guard. Although this can sometimes be good, it also mean that the food it was carrying will arive later (maybe much later) at the base, and the creation of new ants therefore slows down some.

Even though the evolution only lasted for twenty generations we could see a clear improvement in the ANT. But even if we had one hundred generations it would not be able to beat the best ANTs in `MyreKrig`. There are a lot

of aspects of ANT strategies that need to be taken into account, and we are sure that elements from genetic programming (see section 8.4) could help us find these aspects by modifying the pattern instead of having a fixed pattern, as we have now.

7.2 Noise

Due to what is commonly known as *noise* in the data, which we have collected from running our genetic algorithm on different randomly generated ant generations, we find it difficult to conclude much from our data set. We have, for instance, processed our data set with a script that identifies the number of identical bit strings for each generation of a run, plus the total number of identical bit strings in a total run accumulated from generation to generation. This examination did not show us any patterns, that enabled us to conclude much, if anything, about the different degrees of elitism, crossover and mutation.

7.3 Results from data processing

As can be seen from the graphs in chapter 6 it is hard to distinguish the seperate effect that tuning the various parameters has on the graphs. One thing that is possible to conclude from the resulting graphs is that an evolution is present. It would have been nice if we had a final resulting graph where it was clearly marked what/where the local maximum(s) and the global maximum was. It is however unlikely that we could have such a graph as `MyreKrig` will not nessesary make the same output given two similar breeds. We assume that this is because we need to increase the amount of data on which to base the graphs. Another possible reason that this did not happen was because of the flaws in the ANT described above. Finally a more radical change of the parameters might have given more detailed graphs. Perhaps it would have made a better representation of the effects of *crossover* if the range was from 10% to 100% (rather than from 50% to 90%). Similarly this could have been done with the other parameters as well.

It would have been nice to be able to conclude on the *exact* values that is best for evolving our ant. This might have been possible if a larger amount of data, based on a larger number of individuals, was explored. This must be regarded, though, as highly unlikely, as we have infinitely many possibilities for setting parameters. We might be able, though, to find a set of nearly

optimal settings for the parameters.

7.4 Design decisions for the system

The idea of using two programming languages for the system worked fine. We utilized the best of both worlds by making Java do all the calulations and making Perl do all the processing of the data going directly to and coming directly from `MyreKrig`. One consequence of this idea is that it is easily adaptable to other problems requiring development by genetic algorithms. This is achieved with a minimum of changes to the system. On the downside, in order to change parameters specific to the algorithm, one has to edit the source code for Java, and in order to change the parameters for the Perl based part of the system, one has to edit the scripts. It is possible to take the part developed with Java and use in a similar context. All that needs changing is the interface.

Chapter 8

Future work

In this chapter we will look at what future work could contain and what we would do differently for instance by using other methods such as genetic programming or neural networks. We will also take a look at the exsisting ANT and examine what could be optimized here.

8.1 Neural network

A technology we could have employed in order to develop our ANT is *neural networks*. This could have been employed as the solely technology or in addition to other technology.

A neural network is a simplified way of modelling the human neural network. This method excels over the other theories mentioned in this chapter, when the domain of the problem is very specific. One of the requirements needed in order to take advantage of neural networks is the ability to inform the system about whether the result from a given input is correct or incorrect. This is done by making it more likely, in the future, to follow the path to the correct answer than to an incorrect answer. This allows the system to learn how to continuously make a better estimate on the correctness of a given input. Often this method is used in tasks of recognition (i.e. symbols and sounds). [13]

We have not employed neural networks in our project because we are not able to tell the system whether the responce to a given input is correct or incorrect. For more information on this topic see appendix B.

8.2 Bayesian networks

Instead of relying solely on a genetic algorithm for the evolution of our ANT in `MyreKrig`, we could have looked into the area of using *Bayesian Networks* and whether it would have proved useful. We could have applied this technology along with the use of our genetic algorithm.

Bayesian networks are often put to use when dealing with *reasoning under uncertainty*. Using bayesian networks in systems has helped in predicting future prices on the oil market, weather forecasts and medical diagnosis[2][1]. Although bayesian networks have proven useful in many scenarios, a major barrier when using bayesian networks lies in the difficulty of applying them in complex domains. This may also be one of the reasons why the gaming industry has yet to put bayesian networks to use in agent behavior in computer games[3]. The modeling of a Bayesian Network with hundreds, or maybe thousands, of variables is no easy task to master. Also, it takes a long, long time to process large bayesian networks even on high-end computers. Thus, using Bayesian Networks in real-time games may not be feasible.

We have not employed Bayesian Networks in this project mostly due to a lack of time. Early in the project we felt that it would be easier to solely make use of a genetic algorithm and simple bit strings for ANT behavior, rather than also including the use of bayesian networks.

However, using a bayesian network with our ANT in `MyreKrig` could perhaps have helped our ANT evolve further, since a bayesian network would have provided us with a useful tool in establishing probabilities which could be used for better reasoning under uncertainty during a battle in `MyreKrig`. Deciding when to have the ANT act in a certain way due to various established probabilities may have brought our ANT higher up on the ranking ladder.

For instance, a bayesian network could help our ANT decide what its next move should be from knowing *some* things about its surroundings and *not* knowing *other* things about its surroundings. This leads to having our ANT make decisions from probabilities established with the help of a bayesian network. These probabilities could for instance be: "What is the probability of food being two steps from my current location?", or "What is the probability of an enemy ant being two steps from my current location, if I just killed an enemy ant at my current location?".

For more information on the theory of bayesian networks, see appendix A.

8.3 Optimizing our exsisting ANT

In chapter 4 we mentioned that the ANT was implemented without regard to memory usage. An obvious improvement could therefore be optimizing the memory usage. As the generic algorithm (described in 5.3) bases its fitness somewhat on the ANTs brainsize, this would be a good ide to minimize. This could be done by choosing smaller data types. A variable of type `short` could for instance be used instead of an `int` variable. Of course this is different from operating system to operating system, which must be taken into account. An optimization like that is easy to implement. Similar optimizations with other variable types might also be possible in our ANT.

Also reimplementing the ANT using `switch` instead of the exsisting implementation, which uses `if` statements, could speed up the ANT's performance. This will reduce execution time, which would give the individuals more prestige (2.4.5).

Another way of optimizing the ANT could be to implement more movement patterns. As mentioned in 4.2.6, the exsisting ANT has a fixed pattern. For instance it would be possible to implement a star pattern on top of the exsisting explorer. That way the ANT would able to cover a larger area to find food faster and thereby causing the number of ants to grow faster.

A heavier feature to implement would be to make the ANT build more bases. The exsisting ANT only uses one base, and this due to the complexity and lack of time to implementing the ANT and the fact that many ANTs in the `MyreKrig` league only use one base each and still get good ratings, so it is not nessecary to have more than one base. Implementing support for building more bases is a possibility though and that would improve the ANT's ability to expand on a map.

8.4 Genetic programming

To continue with the idea of using evolutionary computation with `MyreKrig`, we could use *genetic programming*[1].

The general idea with genetic programming (GP) [13] is to represent the movement pattern of each individual in our population as a computational tree instead of as a bit string.

Using this method we would first have to create a population of random

[1]For more information go to http://www.genetic-programming.com

computational trees, then we could use the standard methods from genetic algorithms such as crossover, elitism and mutation and thereby evolve the search pattern of the ANT.

If we were to fit our current pattern in a tree, we would first have to create a computational language for the ANT's movement pattern.

A simple language could be:

- `Move X` : Move forward X turns.
- `Turn L/R` : Turn ANT so that it is facing left or right.
- `Halt X` : Stand still for X turns.
- `Add X Y` : Add the number Y to the variable X.

The movement patterns of our explorer ants, created as a computational tree using our simple language, could be described as in figure 8.1. Of course this simple language could have other commands sutch as `if` and some kind of *loop* command, but this should be enough to reprecent our pattern.

Note that a is a variable and `step` represents our step size, taken from the ANT's memory.

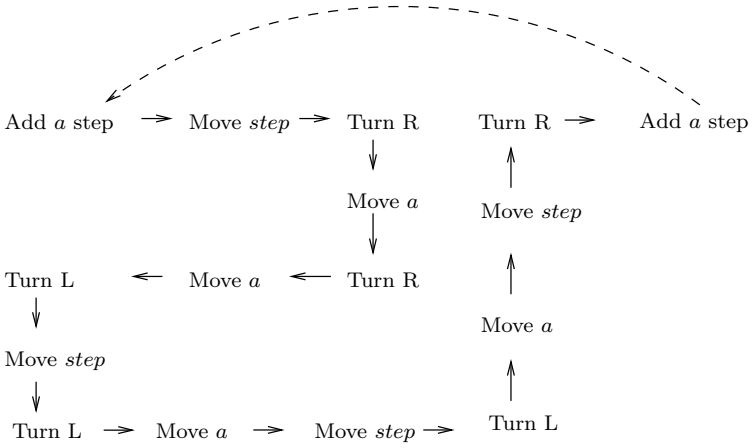


Figure 8.1: Explorer movement pattern using the simple language

We assume all variables are initialized with the value zero and that `step` is a fixed value. The computational tree could look like that shown in figure 8.2.

The way to traverse the tree in figure 8.2 is using a breadth first search algorithm.

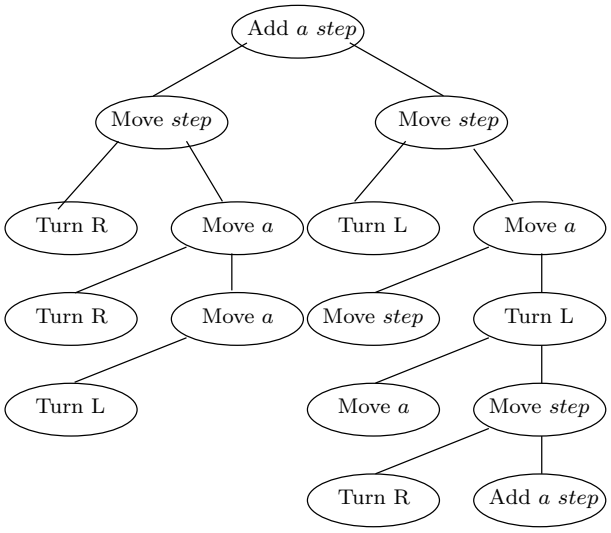


Figure 8.2: The explorer's movement pattern as a computational tree

This pattern is a fixed pattern in our algorithm and we only use genetic algorithms to optimize the variables connected to the pattern. That includes step size, initial max range, etc., but, if you use GP, you will be able to change the code directly and hereby change the way the tree looks using the standard crossover and mutation algorithms.

This will generate a very stupid ANT in the beginning, but eventually it will, even with this simple language, become capable of creating a very good search pattern.

Even though it sounds like a good idea, implementing a skeleton for GP is a very large task and we weighted that it would have taken too many of our available resources on this project, to complete such a task.

Appendix A

Bayesian networks

Definition A.1 (Definition of a Bayesian Network). A Bayesian Network consists of a set of variables and a set of arrows between the variables. Each variable has a finite* set of states which must be mutually exclusive. The variables together with the arrows represent a directed acyclic graph (DAG).

*Variables may in fact have an infinite set of states, but in this report we will limit ourselves to variables with a finite set of states to keep a certain level of simplicity to the subject of Bayesian Networks.

Let's elaborate on definition A.1. A Bayesian Network[9] represents the probability distribution of a domain of variables $(A_1, ..., A_n)$. These variables represent *events* and each variable has a finite set of states.

States in variables must be mutually exclusive. Consider an event "pregnancy test" with the two states "positive" and "negative". These states are mutually exclusive, since the outcome of the test cannot be both positive and negative. If the test is positive, it is not negative, and vice versa.

The graphical representation of a Bayesian Network is a directed acyclic graph (DAG) (see figure A.1) consisting of nodes (variables) and directed edges (arrows) between the nodes. If there is an arrow from A to B_1, we say that A is a *parent* of B_1, and B_1 is a *child* of A. The direction of an arrow indicates the casual direction between two variables.

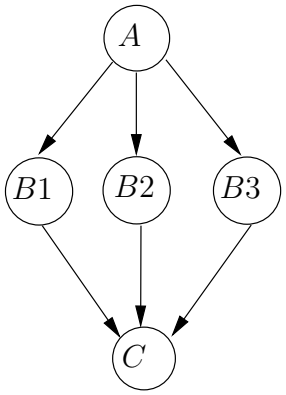


Figure A.1: Bayesian Network Example.

D-separation

The connection between variables in a Bayesian Network can be either *serial*, *diverging*, or *converging*. If we know the state of a variable, the variable becomes *instantiated*. When the state of a variable is known, we also say that we have *evidence* on the variable.

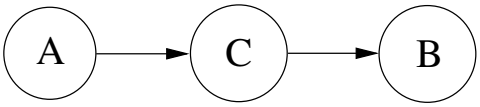


Figure A.2: Serial connection

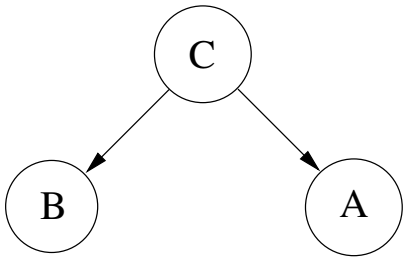


Figure A.3: Diverging connection

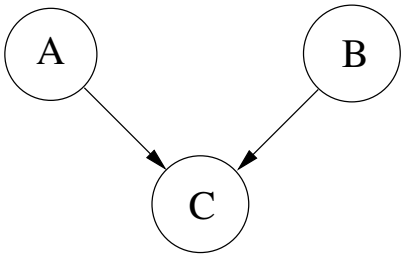


Figure A.4: Converging connection

Definition A.2 (Definition of d-seperation). Two distinct variables A and B in a Bayesian Network are *d-seperated* if, for all paths between A and B, there is an intermediate variable C (distinct from A and B) such that either

- the connection is serial or diverging, and C is instantiated

  or

- the connection is converging, and neither C nor any of C's descendants are instantiated.

When two variables are d-separated, communication between them is blocked. In figure A.2 and figure A.3 this means that if C is instantiated, information about the state of A does not influence the state of B; and vice versa. On the other hand, in figure A.4 instantiation of C is required in order for communication to flow between A and B.

Probabilities

If A is a variable with the states $a_1, ..., a_n$, then $P(A)$ denotes the probability distribution over these states. This can be expressed as

$$P(A) = (x_1, ..., x_n); \qquad x_i \geq 0; \qquad \sum_{i=1}^{n} x_i = 1,$$

where x_i is the probability of A being in state a_i.

Probabilities lie in the range of 0-1. The probability of an event a (variable A being in state a) is expressed $P(a)$. If $P(a)$ is 1, there is evidence on a.

Conditional probabilities

The direction of an arrow between two variables indicates the relationship between the two variables. Hence, we say that the arrows between variables represent *conditional probabilities*.

The probability of an event a given the event b (meaning we know the state of b), can be expressed like this:

$$P(a|b)$$

Important rules for probability calculus

We have two very important rules for probability calculus. These are the *Fundamental Rule* and *Bayes' Rule*.

With the Fundamental Rule we can find the probability of the joint event $a \wedge b$:

$$P(a|b)P(b) = P(a,b) \tag{A.1}$$

$P(a|b)P(b)$ equals $P(b|a)P(a)$ which leads us to Bayes' Rule. Using Bayes' Rule we can find the probability of one event given another event:

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)} \tag{A.2}$$

Example using probability calculus

Say we want to estimate whether the insemination of a cow leads to a pregnant cow. Or perhaps we want to know the joint probability of two different tests (blood and urine) having a positive outcome.

For this purpose we can start out by establishing the following model (see figure A.5) to indicate the casual relationship between the variables Pr, Ho, BT, UT (Pregnancy, Hormonal state, Blood Test, and Urine Test, respectively). Here Pr is our *hypothesis variable*, and BT and UT are our *information variables*.

The variable Ho acts as a *mediating variable* in the model, because the variables BT and UT should not be conditionally independant given Pr. I.e. having evidence on Pr and thereby having a d-seperation of BT and UT should not be allowed, since knowing whether the Blood Test is positive or negative should allow us to change our expectations for the outcome of the Urine Test.

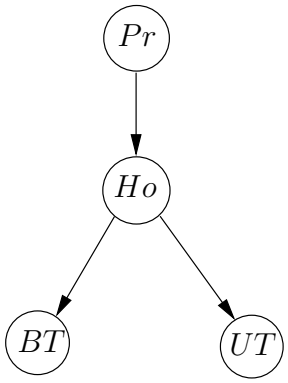


Figure A.5: Bayesian Network Model.

Furthermore, we receive the probability tables A.1, A.2, and A.3 from "experts". Finally, it is established that $P(Pr) = (0.87, 0.13)$.

	$Pr = y$	$Pr = n$
$Ho = y$	0.9	0.01
$Ho = n$	0.1	0.99

Table A.1: Conditional probability table for $P(Ho|Pr)$.

	$Ho = y$	$Ho = n$
$BT = y$	0.7	0.1
$BT = n$	0.3	0.9

Table A.2: Conditional probability table for $P(BT|Ho)$.

Given the probabilities of tables A.1, A.2, A.3, and $P(Pr) = (0.87, 0.13)$, we can now calculate the joint probability table of $P(BT, UT)$:

	$Ho = y$	$Ho = n$
$UT = y$	0.8	0.1
$UT = n$	0.2	0.9

Table A.3: Conditional probability table for $P(UT|Ho)$.

First step is to calculate $P(BT)$ and $P(UT)$.

$$P(BT) = (0.57, 0.43)$$
$$P(UT) = (0.65, 0.35)$$

Second step is to calculate the conditional probability table of $P(BT|UT)$ as shown in table A.4.

	$UT = y$	$UT = n$
$BT = y$	0.68	0.37
$BT = n$	0.32	0.63

Table A.4: Conditional probability table for $P(BT|UT)$.

Third step is to calculate the actual joint probability table of $P(BT, UT)$.

The Fundamental Rule is used in order to calculate $P(BT, UT)$ for each of the four joint probabilities.

$$P(BT = y|UT = y)P(UT = y) = 0.68 \cdot 0.65 = 0.4420$$
$$P(BT = n|UT = y)P(UT = y) = 0.32 \cdot 0.65 = 0.2080$$
$$P(BT = y|UT = n)P(UT = n) = 0.37 \cdot 0.35 = 0.1295$$
$$P(BT = n|UT = n)P(UT = n) = 0.63 \cdot 0.35 = 0.2205$$

Using these values we can create the table for $P(BT, UT)$ as shown in table A.5. The table tells us something about our initial goal; namely that the joint probability of the two tests having a positive outcome is 44.2%.

Learning

There are many ways of learning a Bayesian Network. One of the most common ways is done by collecting data in a large database and then modeling the network from these data. There are several methods for doing this. We will go through a few here.

	$UT = y$	$UT = n$
$BT = y$	0.4420	0.1295
$BT = n$	0.2080	0.2205

Table A.5: Joint probability table for $P(BT, UT)$.

Batch learning

Batch learning is done by taking data from known examples and by looking at these data, thereby calculating the probability. Then we set up the structure of the network by examining the data and finding the causal direction between nodes.

- Calculating probabilities.

This is done by taking all information in the database and calculating the frequencies of each entry in the database and use them as probabilities.

- Finding structure.

When probabilities are calculated, we try to find the causal directions mostly by using logic, or we can use different software tools (e.g. Hugin).

When a structure is found, it must be checked against known information. An example of a structure is shown in figure A.6

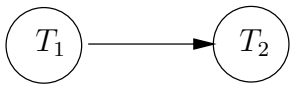


Figure A.6: Simpel Bayesian Network

Then a frequency table is calculated using the chain rule $P^*(T_1, T_2) = P(T_1)P(T_2|T_1)$ shown in table A

	Eggs	Beans
Juice	0.28	0.24
Milk	0.22	0.26

Table A.6: P^*: Like eggs and beans for breakfast.

The "distance" between these two tables, P and P^*, are calculated using the formula:

$$dist_k(x, y) = \sum_i y_i(log_2 y_i - log_2 x_i)$$

The final structure is only one of many accepting structures, so we have to consider the size of the structure:

$$SizeM = \sum_{A \in U} Sp(A)$$

Then the acceptance measure will look like this:

$$Acc(P, M^*) = Size(M^*) + k * dist_q(P, P^*)$$

If the model is not within the acceptance measure, the model will need to be changed. In principle we would have to check all possible combinations of the models nodes. It would be easy in the example above, but if you have a lot of nodes it would take a very long time to calculate all combinations.

Adaptation

When a system is at work you repeatedly get new cases, and you would like to learn from these cases. E.g. you have a system where the structure of the network is fixed, but the probabilities are dependent on a context that varies from place to place. Then you want a network that adapts to that place.

The easiest way of doing this are by introducing a Type variable T between cases in the Network

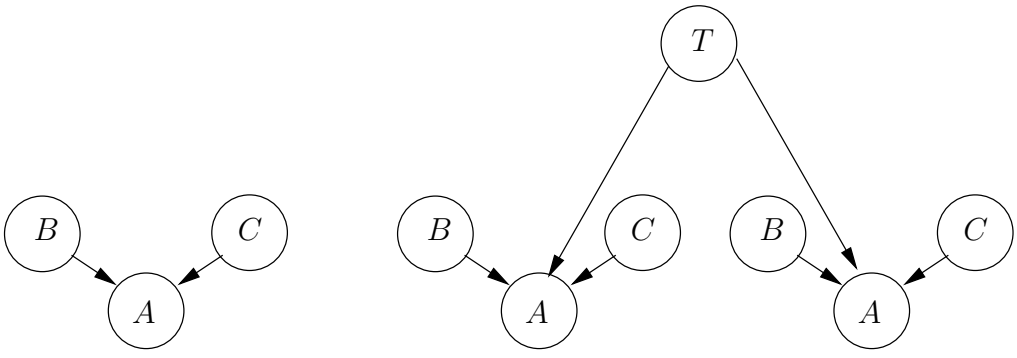


Figure A.7: adaptation

In the model A is modeled as $P(A|B,C)$ and the uncertainty of this can be modeled as a parent to A, which we call T. The Type variable T is initialized $P(T)$ and will be set as parent of all nodes whose tables are dependent on the context of T.

When a new case is introduced into the system, the propagation will yield a new $P^*(T)$. The change of T is the adaptation of the new case into the network and hereby the network will learn from the new case.

- Fractal updating.

In some cases it will not be possible to model the uncertainty as a type variable. In these cases we can use fractal updating.

Fractal updating can be done by considering a new case as new data in the database. So when a new case is found, we must find what nodes it affects and modify the probability distribution accordingly.

Consider $P(A|b_i,c_j)$ from figure A where b_i and c_j are states for which a large number of past cases have occurred and we want to modify the distribution of probabilities accordingly. We must consider a sample size s which is a number representing the size of the past cases of b_i and c_j and consider a set $(n_1,n_2,n3)$ such that $s = n_1 + n_2 + n_3$ and

$$p(A|b_j,c_j) = (\tfrac{n_1}{s}, \tfrac{n_2}{s}, \tfrac{n_3}{s})$$

Now if the probabilities of $P(A|B,C) = (x_1,x_2,x_3)$ and we get a new case e the update to the size s will be $P(b_i,c_j|e) = z$ where z is update to the size. If we use the distribution $P(A|b_i,c_j,e) = (y_1,y_2,y3)$ then the general fractal updating can be done this way:

$$x_k := \tfrac{n_k + z y_k}{s+z}$$

There are known drawbacks to using fractal updating, namely that it tends to overestimate the count of s and thereby overestimating our certainty of the distribution.

Appendix B

Neural networks

Introduction to neural networks

One of the ways of improving an agent is by the use of a neural network. As we decided to work with the programming game MyreKrig in our project, we also had to take a closer look at neural networks. In this section of the report we therefore investigate what neural networks are, and the highlights of the theori are presented here.

Artificial Neural Networks (ANNs) are inspired by the human neural network. An essential part of understanding an ANN is to have an understanding of the human Biological Neural Network (BNN). Figure B.1 shows a conceptual model of a BNN.

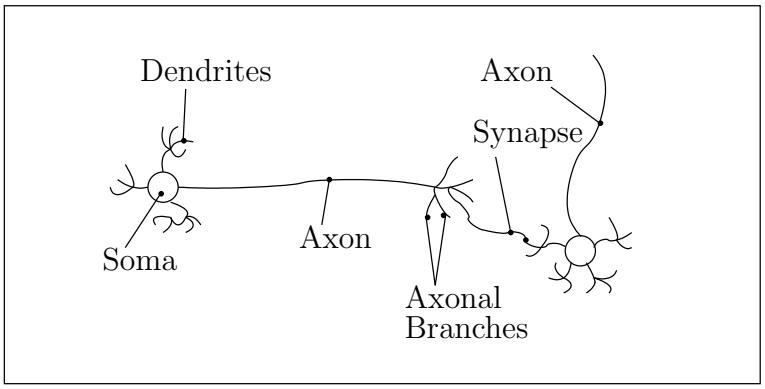


Figure B.1: Conceptual model of a Biological Neural Network

The human BNN is made up of *neurons*. A neuron is a basic information

processing unit consisting of three parts. The first and main part is the *soma* which is the cell-body. The second part is the *axon* which is a single output-connector that splits up into many different *axonal branches*. Each tip of the axonal branches connects to the endpoint of a *dendrite* from a different neuron. Dendrites are the third and final part of the neuron. Dendrites are connectors that attach to the cell-body at the opposite end from the axon. They are the neuron's input-connectors, so the neuron uses them to receive signals from other neurons. The point of connection between a dendrite and an axonal branch is called a *synapse*. This way a single neuron can receive many different inputs from many different neurons, and the neurons which receive the output from that neuron will, of course, receive the same signal or "value". In order to understand the scale of an BNN, the human brain consists of nearly 10^{11} neurons [13].

Information is passed from one neuron to the next when a soma sends a signal through its axon and out to the synaptic junctions, where the axonal branches connect to other neurons' dendrites. The axon generates the signal using a chemical substance called a *neural transmitter* which crosses the given junction and exites the connected dendrite, causing it to send an electrical charge into the soma of the neuron it belongs to. This causes a change in the eletrical potential of the soma. When the eletrical potential reaches a given treshold, a signal is then sent along that neuron's axon and so on. This way a range of signals is generated using multiple neurons.

The task in computer science is to model this biological system to the best of our ability. There are, however, some characteristical differences between the two systems, because of the inherent differences in the way in which the material of computer chips work and how the biological material of our neurons and their chemicals work. More on that later.

There are also other characteristics and abilities which should be noted about neurons. One of them is that the treshold of a neuron changes over time. It is also possible for a neuron to make new connections to other neurons, and it is possible for neurons to migrate from one place to another. All of this participates in creating the brains ability to learn. In ANNs gradient descent is often used to adjust parameters in functions. The adjustments are based on input-output values taken from pairs of values in training examples.

BNN		ANN
Soma	=	Activation function
Dendrite	=	Input
Axon	=	Output
Synapse	=	Weight

Table B.1: Analogy between BNN and ANN

The mapping from BNN to ANN

Here are the analogies between biological and artificial neural networks (given as figure B.1):

There are two major differences between the neurons in BNNs and the ones in ANNs that are interesting to analyze:

1. A biological neurons output will decline if the net input is large enough to cause an output near the maximum, and if this input persists for more than a few seconds. This is called the *fatigue effect* and is not present in our model.

2. The response of a biological neuron to changes in its net input is delayed by a few milliseconds because the electrochemical processes inside it travel with a speed of 10^{-3} seconds. Our articifial neuron reacts with a speed of approximately 10^{-10} seconds[13].

Dataprocessing in ANNs

Like a biological neuron each artificial neuron is an elementary information processing unit, known as a *perceptron*. It uses several inputs and defined values along with an activation function to calculate its activation level. Figure B.2 illustrates the process that takes place in an artificial neuron when given some input values.

Input values to a perceptron can be either raw data from an interface to the environment or outputs from other perceptrons. Output values from a perceptron can be either a final solution or an input value for other perceptrons. The multiplications shown in figure B.2 are the input values multiplied by a predefined weight value. These results are summed and the resulting value is used as input to an activation function. This function returns a boolean

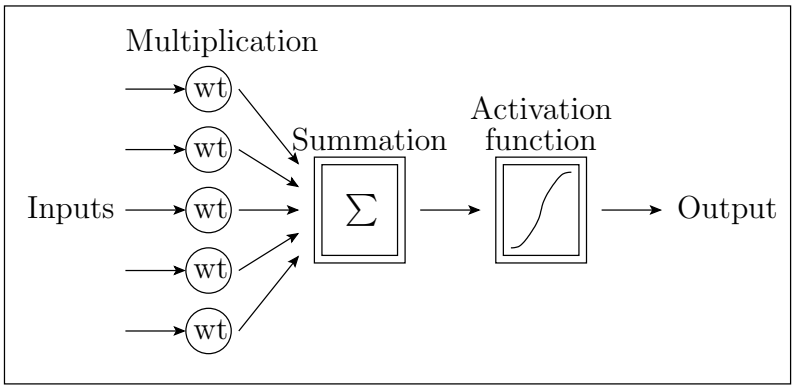


Figure B.2: Conceptual model of an artificial neuron

value which is the final output of the perceptron. In case this value is *true*, a signal is sent to the next perceptron in the ANN. Nothing happens if it is false.

Layers in ANNs (feed-forward networks shown here)

Multiple ANNs can be connected. A common way of doing this is using layers of ANNs. Working similarly to the input and output characteristics described for a single neuron, the output from a single ANN-layer is used as the input to the next ANN-layer. Neurons that are a part of the same layer do not connect to each other. They only send signals to the neurons in the next layer. The output of the last ANN-layer is the final result of all the layers in the ANN, that is the entire neural net. All this is summed up in figure B.3

Feed-forwarding

A feed-forward ANN does no recalculating and modifying of the individual perceptrons and ANN-layers, thus this inhibits the ability to learn. Feed forward ANNs get their name from the fact that there is no feed-back in the network. All signals travel forward. There is therefore no communication between nodes that reside on the same layer. There is also no communication

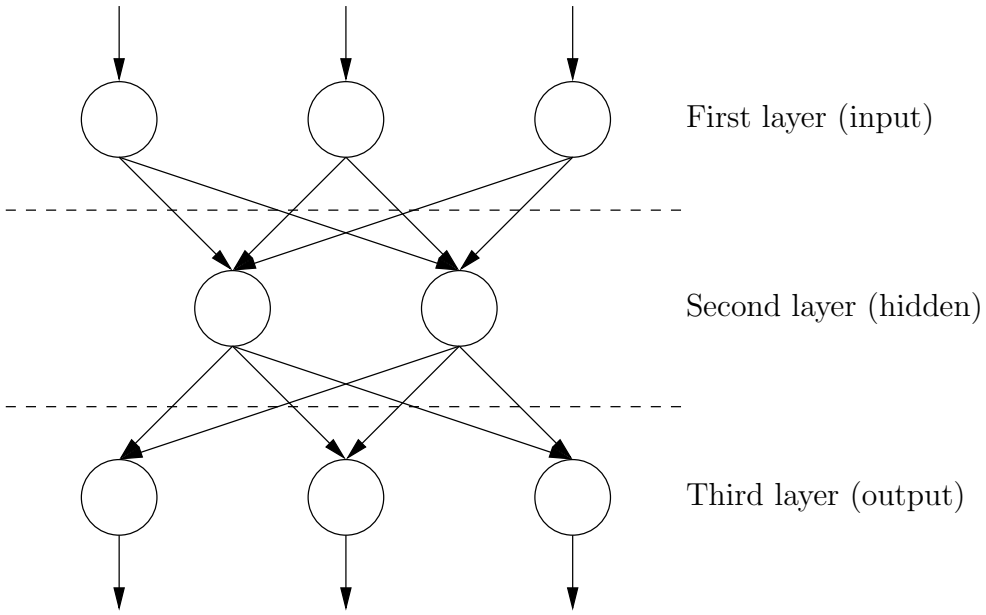


Figure B.3: Organisation of layers in a ANN

backwards through the layers unless the back propagation technique is also used. It is often ambiguous as to whether a layer is a layer of nodes, or a layer of the weights between the nodes, as both uses are common. The ANN described above is also known as a Multi-Layer Perceptron (MLP).

General characteristics of an ANN

This section lists some of the advantages and disadvantages of ANNs.

Advantages of using an ANN:

- ANNs are a robust method of approximating real-valued, discrete-valued og vector-valued functions.
- A Neural Network can extract very complex mathematical functions.
- Neural Networks have an excellent ability to derive meaning from non-linear, imprecise data, or errors in the training data.
- Almost any finite-dimensional vector function in a compact set can be approximated.

- Training takes just a fraction of the CPU time that trial and error methods take.
- ANNs have been succesful in image analysis, speech recognition and in learning control strategies for robots.
- ANNs are to date among the most succesful known learning methods when it comes to interpreting sensor data from the real world.

Disadvantages of using a ANN:

- Neural networks are not magical. Garbage in = garbage out.
- A neural network is a mathematical "black box".
- All input fields must be numeric.
- Neural networks can be difficult to implement.
- Lots of training data and CPU time may be required for training.

Interference and old data

Neural networks can suffer from a problem called *interference*. A Neural Network is trained by repeatedly presenting data to it. The network then adjusts its parameters slightly to better represent information gained using the new data. The problem araises when the data given as a training set is very similar. This will cause the ANN not to be versitile enough to be used in all the input space, because it can forget the mapping it learned in other regions of the input space.

However, the forgetting behavior can in some situations be useful if the function being modelled is changing over time. The new data will then eventually erase the effects of the out-dated information.

Memory based learning does not suffer from interference. Since there is no training phase, there is no "loss of importance" of older data. It's all in memory. If the underlying function is changing through time, the confidence intervals provided with memory based learning can be used to explicitly determine when data should be discarded.

A confidence interval gives an estimated range of values which is likely to include an unknown population parameter. The estimated range is calculated from a given set of sample data.

The width of the confidence interval gives us some idea about how uncertain we are about the unknown parameter (see precision). A very wide interval may indicate that more data should be collected before anything very definite can be said about the parameter.

The highlights of the theori of neural networks have now been described above. We have chosen to avoid using an ANN to develop our agent, because when compared to genetic algorithms, ANNs seem more complicated to work with, and the strengths of ANNs have not won us over, because ANNs do not seem significantly superior to genetic algorithms. Since our work is based on showing that we can in fact significantly improve the succes rate of our agent according to our succes criteria, we do not need a method of doing that which is more complicated, but only marginally better than one of our alternatives.

Appendix C

AI/AA in computer games

The FPS game genre

History of game bots

If we restrict ourselves to the First Person Shooter (FPS) genre, AI has been used in these games since 1992 when Id Software released Wolfenstein 3D. Only 18 months later the genre-defining DOOM emerged from the computer labs of Id Software.[25] AI has evolved from year to year ever since up to today's Unreal Tournament 2003, Half-Life, Battlefield 1942, etc.

Unreal Tournament 2003

The latest installments in the FPS genre make use of highly-evolved intelligence schemes for the bots in the game. A game such as Unreal Tournament 2003 (UT2003) comes to mind.

The bots in UT2003 (and earlier installments in the Unreal series) benefit from smart and advanced AI programming. The AI in UT2003 is one of the game's most important and highly praised features among its huge user base world-wide. The AI was developed over many years, and we have witnessed a considerable evolution of the first Unreal bots which were included in the original Unreal. The gaming community pretty much agrees that the bot AI in UT2003 is the most advanced human-like AI ever used in a FPS game to this date.

According to Epic Games (the game company behind the Unreal series), "[the bots in UT2003] will constantly fool you into believing they are actual

Figure C.1: Unreal Tournament 2003 in-game screenshot

human opponents, something which could not be said for some other recent releases in the FPS genre".[18]

Current use of theories and practices

The current use of theories and practices surrounding AI in bots today primarily make use of what is called a *schedule-driven state machine* which is similar to a *deterministic finite automata*. The schedule-driven state machine, however, has several different schedules (defining bot behavior) defined for each bot to choose from.[14]

Most computer game programmers, of cource, tend to build their bots to be the most fun and challenging to play against as a human player. Afterall, games are all about fun. A very skillful bot developer, Jan "MrElusive" Paul[1], creates his bots in the same manner and hasn't read many AI books to help him create skillful bots.[21][23]

Researchers, on the other hand, usually do not aim to build a bot which they believe will be the best bot, but rather they are interested in the AI research

[1]MrElusive is today a bot developer with Id Software.

itself and whether the research pays off and proves useful in other research projects.[14]

NeuralBot

Only very few bots to this date actually make use of AI techniques such as Genetic Algorithms, Bayesian Networks, and Neural Networks to accomplish human-like reasoning and general behavior. Also, the bots using these AI techniques are not easily accepted by the gaming community. For instance, some years ago a Quake II bot was created which relies on a Neural Network and a Genetic Algorithm to evolve its skills. While this bot, called the NeuralBot, was not esteemed amongst gamers, it gained the attention of researchers.[14]

So while being a great research avenue, the NeuralBot has been looked down upon by the gaming community, because of the way it acts. MrElusive once said about the NeuralBot: "Yes, very cool project... but NeuralBot will never really understand the 3D world or the specific map."[14]

Another quote[14] which to most gamers hits the nail right on the head:

> "In a shoot-em-up like Quake, the "smart" thing for a creature to do would be to run like hell when they saw you coming. After all, they've just seen you waste three of their pals. Trouble is, that's boring. They'd just go and hide in a corner and shoot you in the back the first chance they get... the game would be reviewed as unfair and bad gameplay... but hold on; that's real AI!"

Bibliography

[1] B. Abramson. Arco1: an application of belief networks to the oil market. http://citeseer.nj.nec.com/context/79788/0.

[2] B. Abramson and J. Brown et al. Hailfinder: A bayesian system for forecasting severe weather. http://citeseer.nj.nec.com/context/1229802/0.

[3] Henrik Jarlskov Andreas S. Værge. Using bayesian networks for modeling computer game agents. http://www.cs.auc.dk/research/DSS/Teaching/Projects/S03-bnet-games.pdf.

[4] ccdc.cam.ac.uk. Selection pressure. http://www.ccdc.cam.ac.uk/support/prods_doc/gold21/gold_21/GOLDdocn97.h%tml.

[5] Aske Simon Christensen. Myrekrig. http://www.myrekrig.dk.

[6] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. http://www.msci.memphis.edu/~franklin/AgentProg.html.

[7] Jørn Holm. *GiAnt*. 2003.

[8] iRobot Corporation. irobot corporation. http://www.irobot.com.

[9] Finn V. Jensen. *Bayesian Networks and Decision Graphs*. Springer, 1st edition, 2001. ISBN 0-387-95259-4.

[10] John M. Kowalik. Alan turing. http://ei.cs.vt.edu/~history/Turing.html.

[11] John McCarthy. What is artificial intelligence? http://www-formal.stanford.edu/jmc/whatisai/whatisai.html.

[12] Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, 1996.

[13] Tom M. Mitchell. *Machine Learning*. McGeaw-Hill, 1st edition, 1997. ISBN 0-07-115467-1.

[14] Chris Moyer. How intelligent is a game bot, anyway? `http://www.tcnj.edu/~games/papers/Moyer.html`.

[15] Michael Negnevitsky. *Artificial Intelligence*. Addison-Wesley, 1st edition, 2001. ISBN 0201711591.

[16] MSNBC News. New robots well trained for war. `http://www.msnbc.com/news/857368.asp?cp1=1`.

[17] The Detroit News. Looking to iraq, military robots focus on lessons of afghanistan. `http://www.detnews.com/2003/technology/0301/12/technology-57614.htm`.

[18] PlanetUnreal. Ut game guide - the bots. `http://www.planetunreal.com/utguide/bots.shtml`.

[19] Steve Rabin. *AI Game Programming Wisdom*. Charles River Media, 2002. ISBN 1-58450-077-8.

[20] Ayse Pinar Saygin. the turing test page. `http://cogsci.ucsd.edu/~asaygin/tt/ttest.html`.

[21] ShackNews. Devine on q3ctf & mrelusive. `http://www.shacknews.com/onearticle.x/1869`.

[22] Aaron Sloman. Artificial intelligence, an illustrative overview. `http://www.cs.bham.ac.uk/%7Eaxs/courses.ai.html`.

[23] Telefragged. Interview with mrelusive. `http://www.telefragged.com/metro/interview_mre.html`.

[24] usmilitary.about.com. Military robots of the future. `http://usmilitary.about.com/cs/weapons/a/robots_2.htm`.

[25] The Free Encyclopedia Wikipedia. First-person shooter. `http://en.wikipedia.org/wiki/First-person_shooter`.