Optimizing a Team of Quake 3 Bots for the Capture the Flag Game Type

J.G. Gerrits

July 2009

Master Thesis
Artificial Intelligence
Dept. of Artificial Intelligence,
University of Groningen, The Netherlands

First supervisor:
Dr. M.A. Wiering (Artificial Intelligence, University of Groningen)

Second supervisor:
Prof. Dr. L.R.B. Schomaker (Artificial Intelligence, University of Groningen)

Abstract

In recent years we have seen a huge increase in the popularity of video games. The market for video games has already outgrown the movie industry with annual sales over 31 billion dollars. With the decreasing costs of computational power, the graphics and physics of the games become more and more realistic. However, these are not the only features that make a game successful. The behavior of non-playing characters (NPC's) is equally important to make a game realistic, and thereby fun to play. The industry does not seem to be able to tackle this problem. Even for recently released games, the community still is not satisfied with the game AI at all. In this research, a genetic algorithm and a particle swarm optimizer will be used to train the neural networks of a team of NPC's for the 'capture the flag' game type in Quake III Arena. The teams will be trained with and without a means of communication. Results show that all methods perform better than the fuzzy logic approach of the Quake III engine.

"Is that all you've got?"

Bot Darwin to bot Sarge, in epoch 4, after playing for 12 days, 16 hours, 35 minutes and 20 seconds

Acknowledgements

First of all, I would like to thank dr. Marco Wiering for his assistance and support over the past eight months. This work would not have been possible without our discussions on the subject matter.

I would also like to thank prof. dr. Lambert Schomaker for his time to discuss the progression and sharing his fresh insights on the research.

Many thanks go to my girlfriend Sonja, for her patience, love and support over the years.

I cannot end without thanking my family, on whose constant encouragement and love I have relied throughout my time at the university.

Contents

Chapter 1

Introduction

1.1 Artificial Intelligence

Man is called *homo sapiens*, human the wise. We think of ourselves as being wise and intelligent. For millennia, we've tried to understand how we think, perceive, understand, act and anticipate. When do we say someone is intelligent? Is it that they are very good in mathematics or quantum physics? These people are very good at making abstract models of the world. But what about a musician? He can be said to be intelligent because of his creativity. Intelligence encompasses just about every form of human activity.

A precise definition of intelligence is unavailable. It is probably explained best by discussing some of the aspects. In general, intelligence has something to do with the process of knowledge and thinking, also called *cognition*. These mental processes are needed for, i.e., solving a mathematical problem or playing a game of chess. One needs to possess a certain intelligence to be able to do these tasks. Not only the deliberate thought processes are part of cognition, also the unconscious processes like perceiving and recognizing an object belong to it.

Usually, cognition is described as a sort of awareness, enabling us to interact with the environment, think about consequences of actions and being in control of these processes. It is also associated with our 'inner voice' that expresses our deliberate thoughts and our emotions. The field of Artificial Intelligence tries to gain more insight in intelligence by building models of intelligent entities. Principles from different scientific fields, like mathematics, psychology, neuroscience, biology and philosophy are incorporated to design such models.

1.2 The Goal of Game AI

Artificial Intelligence in games (game AI) is a very important subject in game design nowadays. The graphics and physics of games become more and more realistic with every release. To ensure good gameplay, the Non Playing Characters, called *bots* throughout this thesis, need to exhibit realistic behaviour as well. The player's expectations need to be met. These expectations can be categorized into three classes.

- The player expects novel situations, thereby creating a high 'replay value' of the game. If we can make sure the bots exhibit unpredictable behaviour, it will help in creating novel situations.

- The player also expects total interactivity. They want every character to react to them, and to be able to manipulate them. The bot needs to be aware of its environment, and react to it.

- Players want a significant challenge. The gaming community, especially the more advanced players who have been playing for years, need to be challenged to keep them interested in the game. The bots need to be able to perform very good without cheating.

In order for bots to meet these criteria, they should possess three properties. First, a single bot should be *intelligible*, meaning that the bot is limited in his capabilities like the player. It should see things the player can see, hear things the player can hear and do things the player can do. Probably, the inverse is even more important: the bot shouldn't be able to do things a player can't do. The bot should have transparent thought processes. The behaviour and actions of the bot should be obvious and serve a certain goal. Players want to see adversaries that look intelligent, and always perform actions that are benificial to the character. In case of a team game, used in this research, we want the bots to cooperate with each other, making it more realistic and a bigger challenge to the player.

The second property for a bot to possess is *reactivity*. The player doesn't just want to shoot at enemies. They also want to be able to impress them, fool them, ambush them and hinder them. When a player does so, he wants realistic behavioural feedback from the bot, giving the player a realistic feeling of the game.

Finally, the bot needs to be *unpredictable*. The easiest way to make a bot unpredictable is to let the bot take random actions. This goes against the previous goals discussed, because the bot won't act rationally and realistic when random actions are taken. We need to rely on a combination of factors to generate emergent behaviour of the bot. The bot needs to be able to react to the complex environment and have notion of analog quantities of time, position and aiming performance. Because the environment is constantly changing, the bot doesn't exhibit predictable behaviour.

1.3 A Short History on Game AI

Game playing has been a research area of Artificial Intelligence from its inception. In the early 50's, Christopher Strachey wrote the first working AI program ever: a checkers player. A year later the program could, as Strachey reported, "play a complete game of checkers at a reasonable speed". Around the same time, Dietrich Prinz wrote a chess-playing program. It couldn't play a complete game, but it could solve simple problems for the mate-in-two variety. It examined all possible moves until a solution was found.

The first video games developed in the late '60s and early '70s, like 'Spacewar!', 'Pong' and 'Gotcha' were games without a single trace of AI. They were merely based on the competition between two players. Later in the '70s, single player games with Non Playing Characters (bots) were developed. The movement and behaviour of the bots was based on stored patterns in the program. The incorporation of microprocessors would allow more computing power and random elements in these stored patterns. These games were mainly arcade coin-up games which could be played in gaming halls. The random patterns were a big improvement on the gameplay, since the behaviour of the bots could not be predicted anymore. The most famous examples of these games are 'Space Invaders' and 'Pac-Man'.

At the beginning of the '90s, games gradually moved away from basic and heavily scripted bot behaviour. With more powerful microprocessors being developed all the time, programmers started using in-game AI, in the form of Finite State Machines. One of the first games that incorporated this technique was the real-time strategy game 'Dune II', in which there are two competing armies on one piece of land. The enemy needs to decide where to attack first, but it also has to set priorities in order to make the attack realistic. Although the AI looked quite good in the basics, it wasn't able to pull off ambushes or flanking the players when the player didn't expect it. Subsequent real-time strategy games, like 'Command and Conquer', improved the techniques used in 'Dune II'.

Since the second half of the '90s nondeterministic AI methods were used. The game 'Goldeneye 007' (1997) was one of the first First Person Shooters (FPS's) to use bots that would react to players' movements and actions. They could also take cover and dodge fire. Although the AI was unfair, because the bots knew where the player was at all times, the game was far more realistic than other games in the same genre. In 1999, two popular FPS's were released. First, on November 30, Unreal Tournament (UT) was released by Epic Games. The game really focused on multiplayer action. The AI in UT was scripted, which means that scenario's are written out for the bots. This results in predictable behaviour and actions of the characters. Ten days later Id Software released 'Quake III Arena'. The AI for this game used fuzzy logic to make decisions, thereby making sure that the bot doesn't exhibit predictable behaviour. These two games were heavy competitors on the market. Although none of them actually 'won' this competition, some remarks could be made about the differences. The gaming community found that the graphics of Quake heavily outperformed that of Unreal Tournament, but the AI in Quake wasn't as realistic as it was in UT.

In the last decade lots of video games have been released with increasingly complex AI, starting with Lionhead's 'Black & White' in 2001. Although the game itself was one of the most disappointing titles released that year, the AI was stunning. The player could interact with a virtual being and model it to his accord. The behaviour and the appearance of the creature fully depends on how it is treated during the game. Also in 2001, the FPS 'Halo' was released. The AI could recognize threats in its surroundings and move away accordingly. The AI was found to be very good, due to heavy scripting. The game could be played over and over again, each time offering new situations. In 2004, 'Far Cry' was released. The AI used real military tactics to attack the player, and was highly reactive to the players' actions. Although the movement of the bots wasn't always realistic, the behaviour of the characters, especially among each other, was very impressive. In February 2009, the Dutch company Guerilla released 'Killzone 2', the long expected sequel of the very popular 'Killzone'. In 2007, the game won the EGM award for 'worst AI', based on a trailer the organization reviewed. After the release of Killzone 2, Game Informer Magazine stated that "the AI in Killzone 2 is aggressively good". The gaming community is also very enthousiastic about the behaviour and actions of the bots. The game has different types of enemies, all with their own AI engine. This ensures different 'personalities' among different enemies.

This short history is not an exhaustive one. It should give the reader an idea about the development of AI in video games.

1.4 Research Questions

In this research, a lot of comparisons can be made. The research question is twofold. First, we will use two optimization strategies that can be compared, so the first main research question is:

What is the effect of different optimization algorithms on the performance of a team of bots in the capture the flag game type?

Two subquestions can be derived from the main question:

- *What are the characteristics of a team of bots controlled by conventional neural network evolution?*
- *What are the characteristics of a team of bots controlled by a particle swarm optimizer?*

Next, a means of communication among the bots is added. The second main research question is:

What is the effect of evolving communication between members of a team of bots?

1.5 Thesis Outline

In this introductory chapter, a short introduction is given to the nature of intelligence and the goal of artificial intelligence in games. Also, a historical perspective of game AI has been given.

The next chapter will start by giving an introduction to the game *Quake III Arena*, and the reason why this game has been chosen for this research. Next, some AI techniques used in this research are explained. Finally, some related work is discussed.

Chapter 3 explains the methods that are used in this research. First, the optimization strategies are outlined, followed by an in-depth description of a genetic algorithm and the particle swarm optimization algorithm. Finally, a method of adding a means of communication to these algorithms is sketched.

Chapter 4 consists of the experiments that have been done with the algorithms described. The experimental conditions and parameter settings are explained.

Chapter 5 shows the results of the experiments, with detailed explanations of the outcomings.

Finally, in Chapter 6 I will draw conclusions based on the results, and will discuss them. Also some possible future work will be discussed.

Chapter 2

Background

2.1 Quake III Arena

2.1.1 Introduction

In December 1999, id Software released the long-expected Quake III Arena. The game is a First Person Shooter (FPS), which can be considered to be one of the most popular game genres nowadays. An FPS is a game in which the player sees the action as if through the eyes of the player character. The game focuses on action gameplay with firefights against enemy players. The player is situated in a 3D environment together with its opponents. These enemies can be controlled by the computer (bots) or by other human players in a network game. The player needs weapons to defend himself or to attack an enemy player. These weapons can be picked up at different locations in the level, and can be selected in the game. The choice of which weapon to use in which circumstance will have a large impact on player performance. Quake III Arena has nine distinct weapons, although not all weapons are available in every level.

Next to the weaponry, there are other items situated in the level which can be beneficial for the player character. One of the most important items is the medical kit. When the player gets shot by an enemy player, his health will decrease. When a medical kit is picked up, the health of the player will increase by an amount depending on the type of medical kit. Next to health, there is also armor to protect the player. When a player gets shot, the armor will decrease first. Once the player is out of armor, the health will decrease. Armor can be increased by picking up armor items in the level.

Finally, powerups can be picked up in a level. As with the weapons, not all types of powerups are available in every level. The powerups range from doubling the running speed of the player to being invisible. An overview of all items and weapons is given in appendix A.

The predecessors of Quake III Arena were mainly single player games, which could be played against NPC's. The new release focused more on multiplayer action, so new game types were introduced.

- **Free for All** This game type is also called "Deathmatch". The goal is to kill, or frag, as many other players as possible until the fraglimit or timelimit is reached. The winner of the match is the player with the most frags.
- **Team Deathmatch** The players are organized in two or more teams, each with its own frag count. Friendly fire can be enabled or disabled. If enabled, every time a team mate is fragged, the frag count decreases. The team with the highest frag count at the end wins.
- **Tournament** A deathmatch between two players. The player with the highest frag count after the ending condition wins.
- **Capture the Flag (CTF)** A symmetrical level where teams have to recover the enemy flag from the opponents' base while retaining their own. It ends after a certain number of captures or when a timelimit is hit. This is the game type that is used in this research, and will be discussed in more detail later.

2.1.2 Game Choice

On August 19, 2005, id Software released the complete source code for Quake III Arena [1]. This was an excellent opportunity for AI researchers to test AI algorithms in a video game. There already existed some 3D video game test-beds, but with the release of this source code, researchers could really alter the way in which bots were controlled at its foundation.

Westra [2] describes in his thesis how he used evolutionary neural networks to optimize an NPC in Quake III Arena. He already did much work deep into the source code to enable the separation of decision making of the original bots and the implemented ones. Since he was kind enough to hand over his source code, this research builds forward on his results.

2.2 Bot Programming

2.2.1 Hard-coding

There are different ways in which a bot can be programmed. One of the most basic ways to generate bot behaviour is to program every single behaviour separately. By using *if-then* constructions, the programmer has full control over any situation he can come up with. The program code becomes very large when *hard-coding* all the behaviours. It takes a lot of time to implement a single behaviour, like expressing the aggression level of the bot. This type of behaviour depends on a lot of different factors, like the amount of health, available weapons and ammunition levels.

2.2.2 Scripting

Another way to implement the behaviour of the bot is by *scripting* it. The scripting language is a more specialized language that makes it easier for programmers to generate the desired behaviour. The person that programs the bots doesn't need to have thorough knowledge of the game source code, but just needs to understand the scripting language. The scripts can represent complete scenarios of the game. Downfall to this method is that the bots could exhibit predictive behaviour, which is something that has to be avoided.

2.2.3 Finite State Machines

Finite State Machines (FSMs) can also be used to program the bot behaviour. An FSM consists of states and state transitions. The bot is in one of the states, and when certain conditions are met, the bot will go to another state. This is the technique used in Quake III, in which the FSM consisted of nine different states: standing, walking, running, dodging, attacking, melee, seeing the enemy, searching and idle. The bot can only be in one state. The behaviour and actions for each state has to be defined by the programmer. A good way of visualizing an FSM is to think of it as a directed graph of states, as can be seen in Figure 2.1.

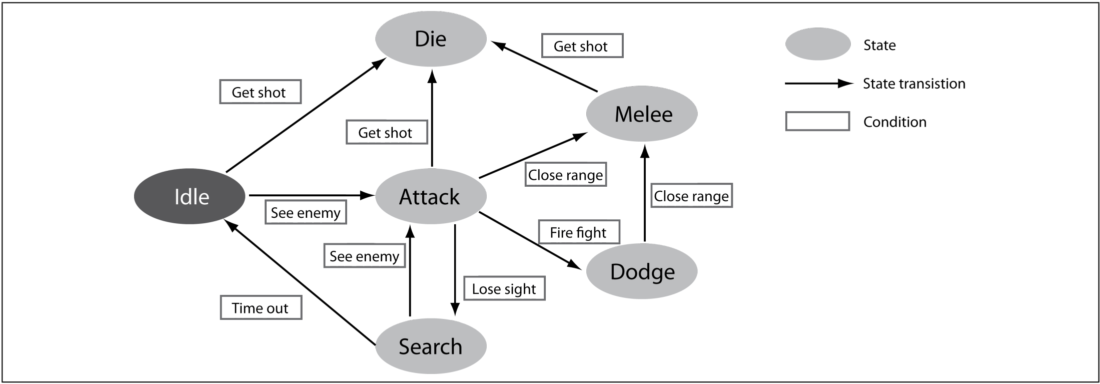

Figure 2.1: Finite State Machine. An example of state transitions.

The bot starts in the idle state. In every game loop, the system checks if the enemy is in sight. If this is the case, a state transition is made to the attack state, and a fire fight will commence. When the bot gets shot, it will change into the die state.

2.2.4 Fuzzy Logic

Quake III Arena uses *fuzzy logic* to let the bot make decisions about certain situations. Where normal (boolean) logic says something about the truth of a situation, fuzzy logic says something about how good a certain situation is. This information is used by the bots to express how much they want to have, or do certain things. For instance, based on the weapon the bot is holding, and the ammunition level for that weapon, the bot can attach a fuzzy value to how much it wants to retrieve more ammunition for the weapon.

```
weight "Lightning Gun"
{
  switch(INVENTORY_LIGHTNING)
  {
    case 0: return 0;
    default:
    {
      switch(ENEMY_HORIZONTAL_DIST)
      {
        case 768:
        {
          switch(INVENTORY_LIGHTNINGAMMO)
          {
            case 1: return 0;
            case 50: return 70;
            case 100: return 77;
            case 200: return 80;
            default: return 80;
          }
        }
        case 800: return 0;
        default: return 0;
      }
    }
  }
}
```

The code above shows the preference the bot has for using the lightning gun in a combat situation. The `INVENTORY_LIGHTNING` variable is set to 1 if the bot carries the weapon. The `ENEMY_HORIZONTAL_DIST` represents the distance towards the enemy, and `INVENTORY_LIGHTNINGAMMO` is the ammunition level of the lightning gun. This gun is only used when the bot is in possession of the weapon and when the enemy is in range. The range of the weapon is 768 distance units. A preference level is set based on the amount of ammunition the bot has for the lightning gun. Every weapon has a fuzzy relation like this. Downfall to this system is that all preference levels have to be coded by hand.

In this research, an attempt is made to let the bot learn its own decision making behaviour using different techniques inspired by biology. First, some basic AI techniques on which these algorithms are based are addressed. The learning algorithms themselves are explained in Chapter 3.

2.3 Neural Networks

2.3.1 Introduction

An artificial neural network (ANN) is an information processing system that is inspired by the way biological nervous systems, such as the human brain, process information. The human nervous system consists of approximately 100 billion [3] interconnected units, called neurons. The brain's ability to process information is thought to emerge from networks of these neurons. An ANN tries to mimic the fault-tolerance and capacity to learn of the brain by modelling the low-level structure of the nervous system. First, an elaborate explanation will be given of the artificial neuron, since it is one of the key elements of this research. Next, the ANN will be introduced.

2.3.2 The McCulloch-Pitts Neuron

In 1943, McCulloch and Pitts published an article [4] called "A logical calculus of the ideas immanent in nervous activity". In this paper, the authors tried to explain how the brain could produce highly complex patterns by using many basic interconnected neurons. They made a model of a neuron, shown in Figure 2.2, which we will call the MCP neuron. An MCP neuron is characterized as being 'on' or 'off'. It is switched 'on' when it is stimulated by a sufficient number of neighboring neurons. This mathematical model of the neuron has become very important in computer science.

Suppose we want to represent the logical AND function with an MCP neuron. For this example, we use two inputs and one output. The truth table for this function is given in Table 2.1. The table consists of all possible input / output patterns. The inputs correspond to the two inputs of the MCP neuron in Figure 2.2. Notice that the signals that are sent to the MCP neuron and the signal that it sends out are all boolean values. This "all or nothing" principle is one of the assumptions McCulloch and Pitts made in their article.

Another assumption the authors made, was that somehow, a real neuron "adds" all the inputs, and decides whether to send a true or false signal based

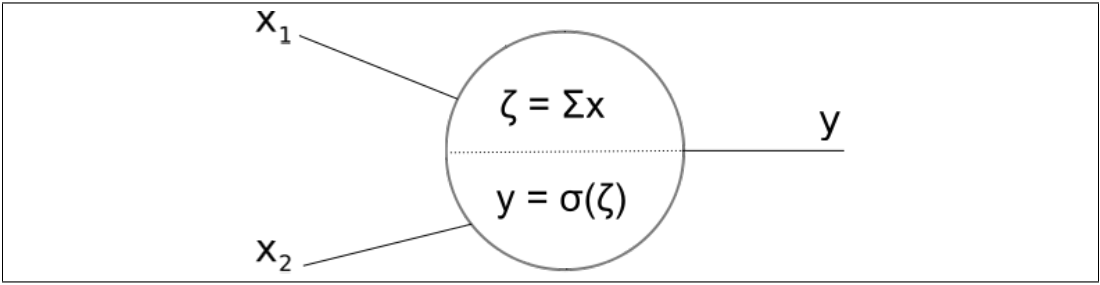

Figure 2.2: The McCulloch-Pitts neuron. A simple mathematical model of a biological neuron.

input 1	**input 2**	**output**
false	false	false
false	true	false
true	false	false
true	true	true

Table 2.1: Truth table for the logical AND function

on the total of signals it received. To be able to do arithmetic with the signals, true will be encoded as '1', and false as '0'. If the summed input is high enough, the MCP neuron will send out a positive signal. Otherwise, a negative signal will be sent out. The summed input is high enough if it exceeds a certain threshold. According to Table 2.1, only if both inputs are positive, the output should be positive as well. In all other cases, the output is negative. When we add up both positive inputs, we have a summed input of 2. This summed input is called the activation of the MCP neuron. The threshold is the Heaviside step function σ, giving output 1 if the activation value exceeds the threshold, and 0 otherwise. The step function is shown in Figure 2.3. If we set the threshold value to 1.5, the MCP neuron will always give the correct response for the logical AND problem.

In 1949, Donald Hebb would help to the way that artificial neurons were perceived. In his book "The Organization of Behavior" [5], he proposed a learning rule for real neurons. He stated: *"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased".* So when two neurons fire together, the connection between the neurons is strengthened. He also stated that this activity is one of the fundamental operations necessary for learning and memory.

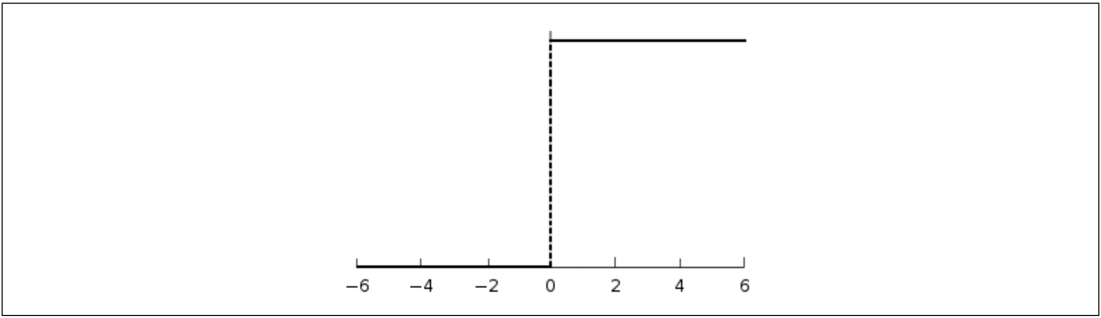

Figure 2.3: The Heaviside step function with threshold 0.

2.3.3 Training

In the previous example of the logical AND function, the value of the threshold had to be calculated by hand in order to get the correct functionality. We can also make an MCP 'learn' the type of behaviour that is required. We need to take into account that one input can be of more influence than another, so we assign a strength, or weight value to each input. This can be represented as

$$a = \sum_{i=1}^{n} w_i x_i \tag{2.1}$$

where a is the activation of the neuron, n is the number of inputs, w is the weight value of the input and x is the input value.

The weight value represents the amount of influence the input has on the MCP neuron. A weight is a decimal number, and can be positive or negative. A positive weight means an excitatory signal (it excites the neuron towards the threshold), a negative weight represents an inhibitory signal (it inhibits the signal away from the threshold). We will call such a neuron a Threshold Logic Unit (TLU).

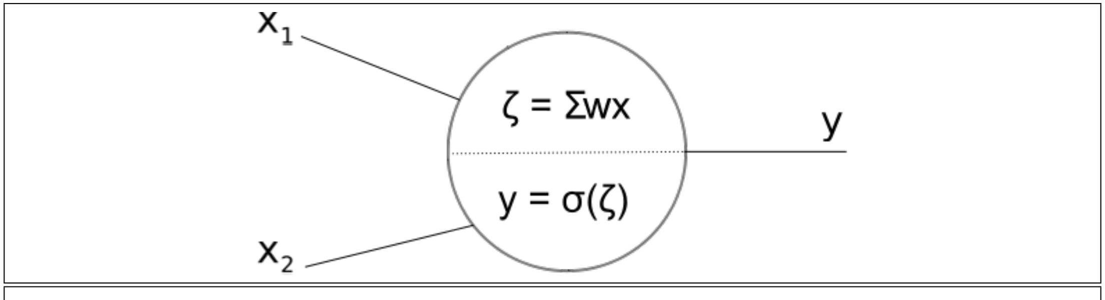

Figure 2.4: The Threshold Logic Unit. An MCP neuron with a weight value associated with an input.

One of the major difficulties with the MCP neuron was that it used the Heaviside step function on activation. Both inputs and outputs are binary. We could change the signal representation to a continuous range, however, the use of the step function at the output limits the signals to be boolean. This can be overcome by using a sigmoid function instead of a step function. As can be seen in figure 2.5, this is a smoothed version of the step function.

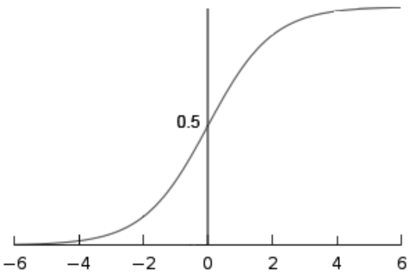

Figure 2.5: An example of a sigmoid function.

The activation of the perceptron is still given by Equation 2.1, but the output function is given by

$$y = \sigma(a) \equiv \frac{1}{1 + e^{-(a-\theta)}} \tag{2.2}$$

where θ is the threshold value and e is a mathematical constant.

Hebb stated that the connection between two neurons is strengthened when two neurons fire together. To incorporate this idea in a TLU, we need to alter the threshold during the process of feeding inputs to the TLU. Let's take the example of the logical AND function again. First, we need to set default values for the weights and the threshold. The first weight will be set to 0.5, the second weight to 0.3 and the threshold to 1. Suppose the input pattern is (1,1). The output of the TLU will be

$$a = \sum_{i=1}^{n} w_i x_i = (1 * 0.5) + (1 * 0.3) = 0.8 \tag{2.3}$$

Since 0.8 is smaller than the threshold value of 1, the TLU will not fire. According to Table 2.1, it should have. We should alter the weights and threshold in order to gain the correct functionality. So let's add 0.1 to the weights, and subtract 0.1 from the threshold. If we try the same pattern (1,1) again on the new TLU, the outcome is

$$a = \sum_{i=1}^{n} w_i x_i = (1 * 0.6) + (1 * 0.4) = 1.0 \tag{2.4}$$

This value is higher than the new threshold of 0.9, so the TLU will fire. If we try the other patterns, the TLU will not fire.

We can also automate this process, by using the perceptron learning rule. The TLU needs to find the decision line by itself by learning the characteristics of the input patterns. Suppose we present the perceptron with an input pattern with the desired response '1', and with the current weight vector, it produces output '0'. The perceptron has misclassified and some adjustments have to be made to the weights. The activation must have been negative, where it should have been positive. We will adapt the weights and threshold by increasing them with a small proportion of the input. If we would change them drastically, previous learning would be 'overwritten'. The weights are updated according to

$$w'_i = w_i + \alpha(t - y)x_i \tag{2.5}$$

where w'_i is the new weight value, w_i is the current weight, $0 < \alpha < 1$ is the learning rate, which defines how much the weights are altered, t is the target output, y is the real output and x_i is the input.

The learning rule can be incorporated into an overall scheme of iterative training as follows.

```
repeat
  for each training vector pair ($i$,$t$) do
    evaluate output $y$ on input pattern $i$
    if $y \neq t$ then
      form new weight vector according to equation 2.5
    else
      do nothing
    end if
  end for
until $y = t$ for all input patterns
```

This procedure is called the perceptron learning algorithm. If the problem at hand is linearly separable, like the logical AND problem, application of this algorithm will eventually result in a weight vector that defines a decision line that separates both possible outputs.

2.3.4 Linear Separability

The TLU can only solve linearly separable problems, like the logical AND function. The output of the TLU can be separated in two classes, 1 and 0. For the two-input case, we can visualize this in a diagram by putting the output value in the field based on the input values on the x- and y-axis. Figure 2.6 shows this for the logical AND function. A decision line can be found to separate the 0's from the 1's.

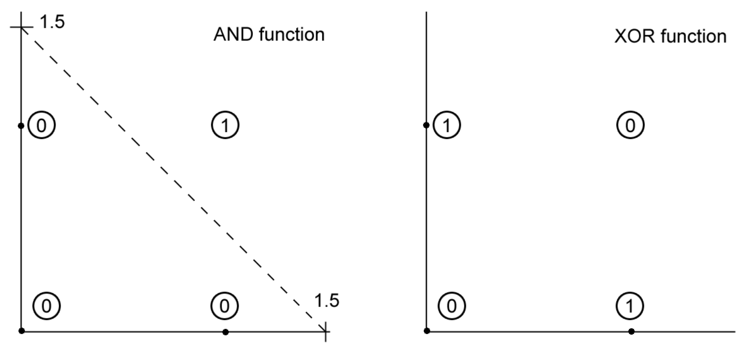

Figure 2.6: On the left: the decision line of the logical AND function. On the right: no decision line is possible for this function.

There are functions that are not linearly separable, like the logical XOR function. The function only returns true if only one of the inputs is true. Otherwise, it returns false. Figure 2.6 shows the diagram of the XOR function. It is not possible to draw a straight line to separate the output values according to their class. This function can not be represented by a single TLU.

input 1	**input 2**	**output**
false	false	false
false	true	true
true	false	true
true	true	false

Table 2.2: Truth table for the logical XOR function

2.3.5 Beyond Linear Separability

If we wish to solve nonlinearly separable problems, we will have to connect units together. Suppose we want to classify data into four different classes *A, B, C, D* and that they are separable by two planes in hyperspace. We can not separate four patterns by a single decision line. We could try to solve the problem by dividing the data into linearly separable regions.

We could separate the data into two classes. Class AB will represent the patterns that are in A or B and class CD will do the same for patterns in C or D. We could train two TLU's with outputs y_1 and y_2 to the values given in Table 2.3.

Class	y_1
AB	1
CD	0

Class	y_2
AD	1
BC	0

Table 2.3: The outputs of the two units

Suppose we have a pattern of class A as input to both of the units. From the tables we can conclude that $y_1 = y_2 = 1$. Doing this for all four distinct patterns, we obtain a table with unique codes in terms of y_1 and y_2, as can be seen in Table 2.4.

y_1	y_2	Class
0	0	C
0	1	D
1	0	B
1	1	A

Table 2.4: The unique codes for classification

These codes can be represented by four two-input TLU's, each one connected to the two units described above. To classify a pattern in A, we need a TLU that fires when it receives input (1,1), and output 0 for all other combinations. The two-layered network is shown in Figure 2.7.

This example is based on two assumptions. The four classes were separable by two hyperplanes, AB was separable from CD and AD was separable from BC. Without these factors it wouldn't be possible to solve this problem with the network in Figure 2.7. This network is called a two-layer net, since there are two layers of artificial neurons, and two layers of weights.

In this example, we had to come up with the code to divide the data into four separable parts. It would be much better to let the network do this for us. The less information we have to supply, the more useful the network will be.

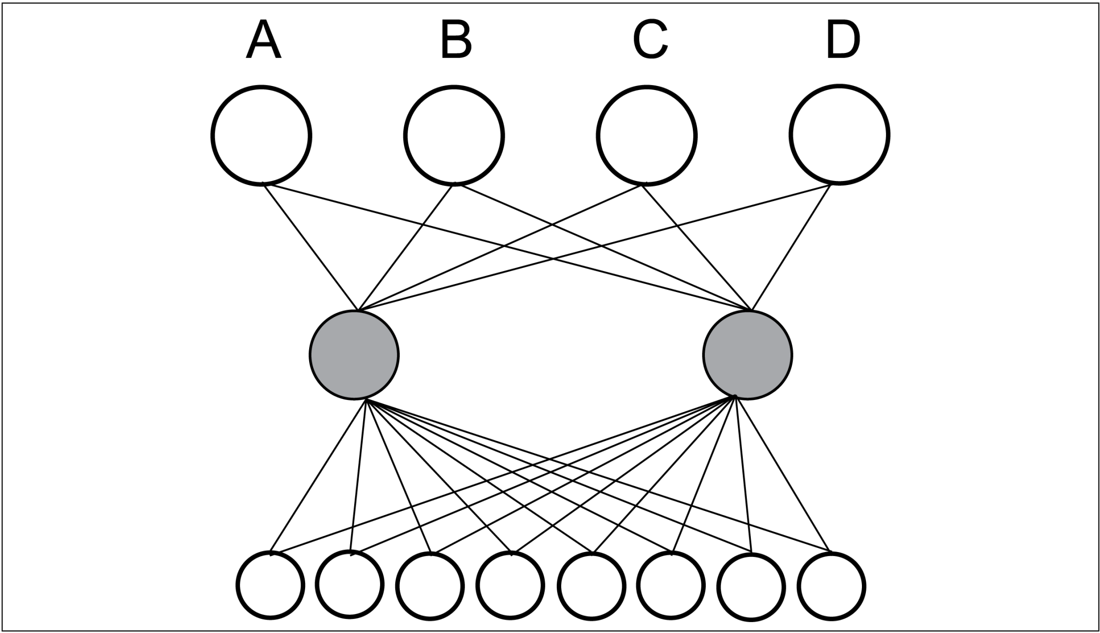

Figure 2.7: Two-layer network for classifying four patterns.

For the perceptron to perform classification of patterns, it must have a decision line (or decision surface with more inputs). Since the decision line is determined by the weights and the threshold, these values need to be adjusted to bring around the required functionality. This can be done by iteratively presenting examples to the perceptron, and comparing the results with the ones desired. Based on that difference, the weights and threshold can be adjusted to bring them more in line with the desired values. This process is called supervised learning.

2.3.6 Artificial Neural Network

To be able to solve nonlinear separable problems, we need a network with multiple layers. The network should consist of multiple input nodes, and at least two layers of semilinear nodes. The inputs are first processed by a layer of *hidden nodes*. These are called hidden because we don't have direct access to their outputs. They form their own internal representation of the inputs. The outputs from the hidden layer are propagated to the output layer, which will give the network's response to the inputs.

There are several ways in which a neural network can be trained. One of these is the *backpropagation* algorithm, which is able to express a rich variety of nonlinear decision surfaces. It uses gradient descent to attempt to minimize the squared error between the network output and the target values for these outputs.

When learning starts, all network weights are initialized to small random values. Next, the main step is to train on an input pattern, in the following order:

- Present the input pattern to the input nodes
- Calculate the output of the hidden nodes
- Calculate the resulting output from the output nodes
- Apply the target pattern to the output layer
- Calculate the errors in the output layer
- Calculate the errors for each hidden node
- Train the output vector weights
- Train the hidden vector weights

The first three steps are known as the forward pass, because the flow of information is moving forwards through the network. Steps 4-8 are known as the backward pass, since the error is propagated back in the network.

2.4 Communication and Language

2.4.1 Introduction

In chapter 1, some insights in the nature of intelligence were given. It is hard to imagine that our intelligence would be at the current level without the usage of some means of communication. This section first describes the nature of communication and language. Next, a previously used method for evolving communication is outlined.

2.4.2 Communication

Communication is a method of conveying information between two or more individuals. There is a sender and one or more recipients. In 1949, Weaver and Shannon [6] created a framework regarding communication, in which they stated that communication is a means to reduce the uncertainty about a particular event. They formalized this as follows:

$$H(X) = -\sum_{i=1}^{k} p_i log_2 p_i \tag{2.6}$$

where H stands for the uncertainty of whether a series of events x_1, x_2, ..., x_i, denoted as X, will occur. The probability for a certain event to occur is denoted by p_i. If the uncertainty about a series of events is not reduced after having sent the message, no communication occured.

Steels [7] describes another definition of successful communication. He emphasizes the inferential nature of human communication. From his viewpoint, communication is successful when the receiver has correctly identified that part of the world the sender is communicating about, and executes behaviour accordingly. This means that communicating successfully is not about the message itself, but about the meaning of the message. To determine if communication was successful, it is not enough to check if the message was transferred correctly, but also if it was understood correctly.

To be able to give meaning to a message, a common language that is understood by both sender and receiver is needed. The characteristics of this language is given in the next section.

2.4.3 Language

Language is many things. It is a system of communication, a medium for thought and a means for literary expression. Everybody speaks at least one language, and it would be hard to imagine not being able to speak a language. The scope and diversity of our thoughts and experience place certain demands on language. It must be more than provide a package of ready made messages. It must enable us to generate and understand new words and sentences. A language must be creative.

2.4.4 Evolving Communication in Robots

De Greeff [8] used evolving communication in his research. He placed two simulated robots in a 110x110cm arena. Two target areas were placed in random positions somewhere in the field. The goal of the robots was to switch places on the target locations. The robots need to communicate with each other to coordinate their behaviour. The idea is to let the robots generate their own communication language. The robot controller consisted of a neural network. This network is shown in Figure 2.8.

The first eight inputs are the values of the infrared sensors. Numbers 9 and 10 are the ground sensors, encoding binary if the robot is on one of the target locations. The next three inputs are values retrieved from camera sensors. The

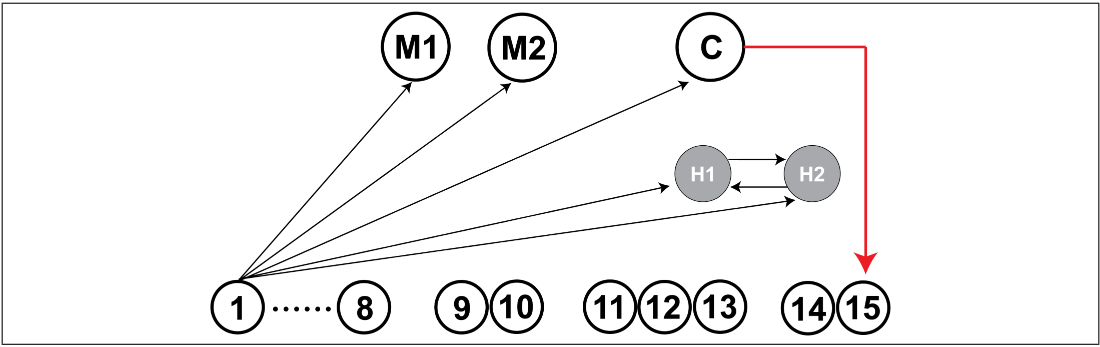

Figure 2.8: Neural network for evolving communication.

first one is the relative angle of the robot in the right visual field, the second one is the angle in the left visual field. The third input is 1 if the robot is not perceived in either visual field. The last two inputs are the communication inputs. The robot has the ability to perceive a sound, and encode this sound as a value between 0 and 1. This is input number 14. The last input is the robot's own communication signal it sent to the other robot.

The purpose of the communication inputs and output is to find a pattern in the robot's own signal combined with the response from the other robot. There are two interconnected hidden units and three outputs. The first two are the motor outputs for the left and right wheel, the third output is the communication signal it sends to the other robot.

Results

De Greeff used a genetic algorithm to let the robots learn the task and evolve the language. The genetic algorithm is explained in chapter 3. The robots have exactly the same characteristics, so they can be seen as one individual. This means they both have the same objective.

A lot of experiments were done, and the behaviour of the robot was tested on various conditions. In general, the results were very good. Results showed that a team of autonomous robots was able to integrate different signals (like sound, visual and infrared) in order to achieve an effective communication system. Eventually, four different sound signals could be distinguished. The robots were able to combine sensory inputs of different channels in a functional manner. How this technique is used in this research is explained in 3.

2.5 Related Work

After the release of the source code of Quake 3 Arena [1] and the test bed GameBots [9] for the FPS Unreal Tournament, a lot of research has been done on AI in video games. Laird used the cognitive architecture SOAR to implement anticipation in a Quake bot [10]. The SOAR bot simulated the perception of human players, like visual and auditory information. The bot was able to predict the behavior of the enemy. A disadvantage of this technique is that the bot shows predictable behavior, if a similar situation occurs again.

Zanetti et al. [11] implemented a bot for Quake 3 that used three neural networks: one for movement of the bot in a combat situation, one for the aiming and shooting skills, and one for path planning. These networks were trained using recorded actions from expert players. The bot turned out to be far from competitive.

In [12], Stanley et al. use evolving neural network agents in the NERO video game. In this game a team of agents can be trained for combat by human players. The realtime version of the NEAT algorithm (rtNEAT [13]) was used. They show that this technique is flexible and robust enough to support real-time interactive learning of decision tasks.

In [14], Bakkes et al. introduce an adaptability mechanism based on genetic algorithms for a team in Quake 3. The key idea is that NPC's don't evolve by themselves, but as a team with a centralized agent control mechanism. The team-oriented behavior is learned by cooperation between multiple instances of an evolutionary algorithm. Each instance learns a relatively uncomplicated behavior. Although they were able to generate adaptive behavior to non-static opponent behavior, the time needed to learn this behavior was extremely long.

Doherty et al. explore the effects of communication on the evolution of team behaviors for teams of bots in FPS's [15]. They show that using communication in difficult environments increases the effectiveness of the team.

Westra uses evolutionary neural networks to train bots in Quake III [2]. He uses weapon and item selection as learning tasks. The bot that evolved performed better than the original Quake 3 bot. The current research builds forward on the research done by Westra.

Chapter 3

Methods

3.1 Introduction

In the previous chapter, the Artificial Neural Network (ANN) was described. There are many ways in which such a network can be trained. This chapter starts with a thorough explanation of the topology of the ANN's of the different bots that are used in this research. In the following sections, two biologically inspired learning algorithms for the neural networks are described. First, a genetic algorithm with evolution of neural networks is addressed. In the following section, the Particle Swarm Optimization algorithm is described. Finally, a method for the emergence of communication among the bots is outlined.

3.2 Optimization Strategies

3.2.1 Introduction

There are many aspects of bots that can be optimized. To keep the amount of work within the scope of this project, choices about what exactly to optimize have to be made. In this research, two decision processes are optimized. First there is weapon selection: which weapon should the bot use under which circumstances. Next, there is item selection: which item in the environment is most valuable for the bot at a certain time.

3.2.2 Fitness Function

To be able to optimize the bots, some sort of performance measurement is needed. In the game this measurement, called the fitness function, is the personal score of each bot individually. This personal score is used by the game to rank the bots' performance. The personal score consists of many different factors. The points given below are additive.

- 5 points for capturing the flag (every bot scores 5 points when a capture is made, the flag carrier gets no bonus)

  2 points for fragging someone who has recently damaged the flag carrier
- 1 point for fragging an enemy player within sight or near the team's flag carrier
- 1 point for fragging an enemy player in the home base
- 1 point for fragging an enemy player
- 2 points for fragging the enemy flag carrier
- 1 point for returning the flag
- 1 point for assisting a capture by returning the flag and the team capturing in four seconds
- 2 points for assisting a capture by fragging the enemy flag carrier and the team captures in four seconds

3.2.3 Weapon Selection

The neural network for weapon selection is shown in Figure 3.1. The network has 13 inputs. The first eight inputs correspond to the amount of ammunition the bot has for the eight different weapons. The other inputs correspond to health, armor, red flag status, blue flag status and damage, respectively.

The network is evaluated very often during the game, because ammunition can run out while in play. All weapons are evaluated in just one pass. In order to make the right choice of weapon, the network must have knowledge of the amount of ammunition the bot has for each weapon. Since all the weapons' ammunition is evaluated at the same time, the choice of the weapon doesn't only depend on the ammunition of the weapon itself, but also on that of other weapons. The weapons that are not available or weapons that lack ammunition can not be chosen by the bot.

The next inputs are the amount of health and armor. These values are important for weapon selection in the sense that the bot shouldn't choose a risky weapon like a rocket launcher when it is running out of health, because he might get hurt or die shooting it. The flag status inputs tell the bot something about the overall game state. If the red (enemy) flag input is 1, the bot's team has stolen the enemy flag, so it might be necessary to play more aggressive to defend the flag carrier. Same goes for the situation where the blue flag status is 1. The bot's own flag has been stolen, so he needs to play more aggressive to kill the

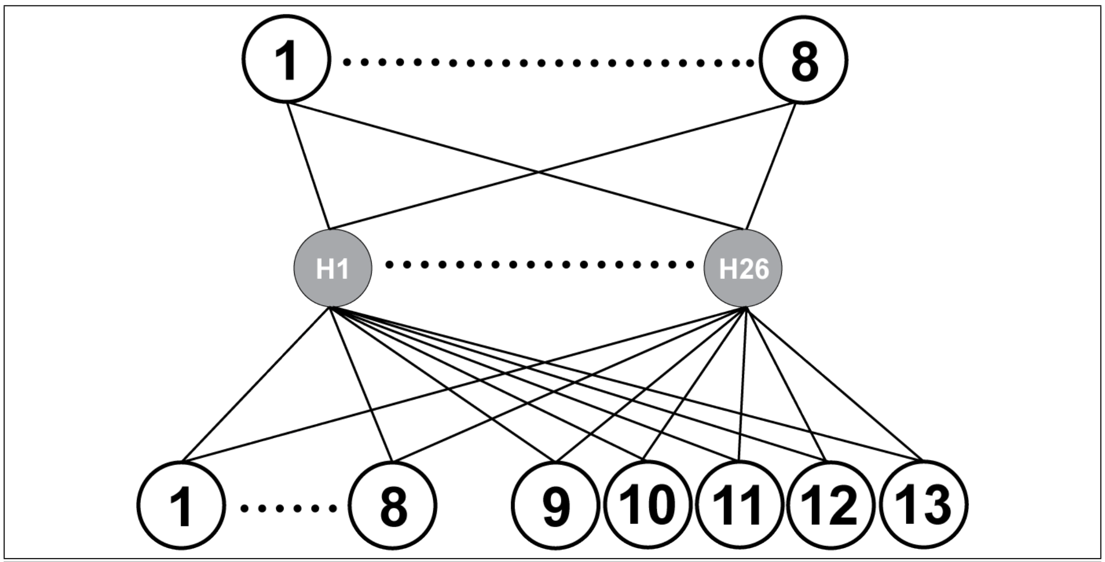

Figure 3.1: The neural network for weapon selection

enemy flag carrier. The last input is the damage the bot has taken since the last evaluation. The amount of damage the bot took might say something about the kind of weapon he was shot with. This might be of influence on the choice of weapon the bot makes.

The inputs of the neural network are scaled to keep the influence of all inputs at the same level. The scaling factor for the weapons differ from one to another. Realistic values of the ammunition level are used, instead of the maximum values. The eventual values are clamped to a value of 1. The health, armor and damage inputs are divided by the maximum amounts (200). The flag status inputs are 0 or 1, so these are not scaled. The values are summed up in table 3.1.

3.2.4 Item Selection

Item selection is part of the 'goal selection' module of Quake III Arena. Other goals, for example, are fighting and retreating. When a bot is not in a fighting situation, the bot picks up items in the level that are beneficial to the bot at that moment. The choice of which item to pick up in which circumstance is optimized. When an item is picked up, it will reappear (respawn) after a certain time.

The neural network for item selection is shown in figure 3.2. The network has 17 inputs, the first ten for each available item type. After that, respectively health, armor, dropped item, travel time, avoidance time, red flag status and blue flag status. The inputs are explained below.

This neural network differs from the weapon selection network in the sense

number	input	scaling factor
1	Machine gun	200
2	Shotgun	25
3	Grenade launcher	25
4	Rocket launcher	25
5	Lightning gun	200
6	Rail gun	20
7	Plasma gun	100
8	BFG	40
9	Health	200
10	Armor	200
11	Red flag	1
12	Blue flag	1
13	Damage	200

Table 3.1: Scaling factors for the weapon selection neural network

that it has just one output. This is because it is not possible to evaluate all items in just one pass, due to the fact that item specific inputs are used. For every item, a new pass is made through the network to evaluate a preference level for that item. After all items have passed the network, the item with the highest preference level is chosen.

As said, the first ten inputs correspond to the item types. The input of the item type is set to 1 if that item is passed. The other inputs are set to 0. The health and armor inputs are normalized with a scaling factor of 200 again.

When a player gets shot and dies, he leaves a weapon behind to pick up for other players. In general, this weapon has a preference over any other weapon, because usually, it is a good weapon. If a weapon was dropped, the weapon gets a very high value (1000) attached to it. The thirteenth input is scaled by a factor 1000, so the input becomes 1 if the item was dropped, otherwise, it is 0. The next input is the travel time. This factor is very important in making a choice of which item to pick up. A nearby item is easier to pick up than an item far away. The fifteenth input is the avoidance time. This represents the time the bot did not visit the item in question. When the bot picks up an item, it should not go back to the item within a short period of time, because it hasn't respawned yet. This factor is represented by the avoidance time. The last two inputs are the blue and red flag status, being a 0 or a 1 if the flag is home or away, respectively. The scaling factors for the inputs are shown in Table 3.2.

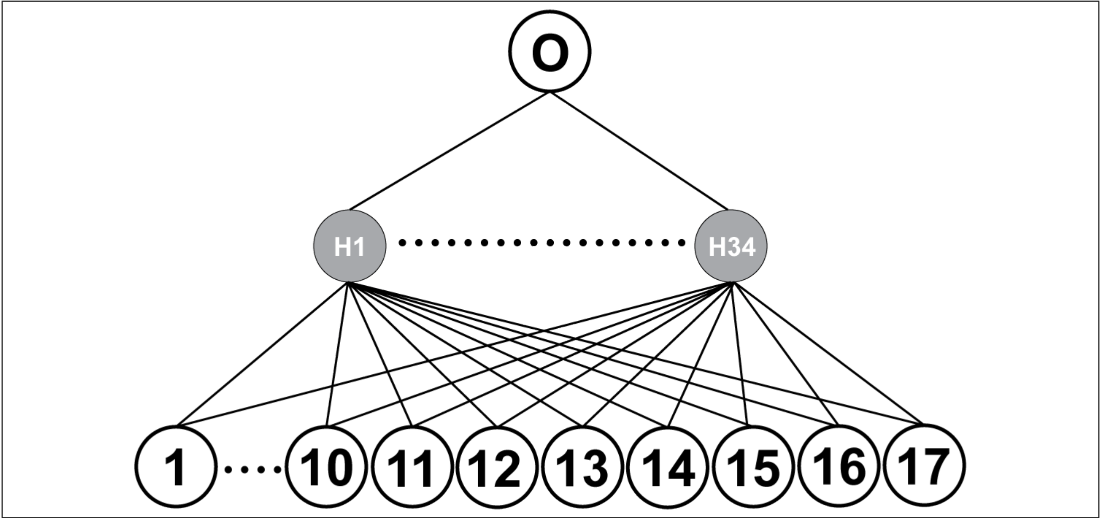

Figure 3.2: The neural network for item selection

3.3 Genetic Algorithm

The original source code, written by Westra and described in his thesis [2], used evolutionary algorithms to optimize the neural networks. A lot of different combinations of techniques were tested in his thesis. This research uses the technique that performed best on the task, and is described below.

3.3.1 Basic Concept

There are many different variants of Genetic Algorithms, but in the basics, these algorithms share the same philosophy. In a population of individuals, the strong

number	**input**	**scaling factor**
1-10	Item type	1
11	Health	200
12	Armor	200
13	Dropped	1000
14	Travel time	2000
15	Avoidance time	50
16	Red flag	1
17	Blue flag	1

Table 3.2: Scaling factors for the item selection neural network

ones survive longer than the weak ones: *survival of the fittest*. This causes a rise in the overall fitness of the population. Based on the fitness value of the individuals, the weak ones are terminated. The strong ones are chosen to reproduce themselves by using recombination and/or mutation on them.

Recombination is an action that is applied to two or more of the selected candidates (called parents). It will result in one or more new candidates (children). Mutation is applied to one candidate and results in one new candidate. Execution of these two operators leads to a set of new candidates that compete with the old ones for a place in the next generation. When this process is repeated, the average fitness of the population will increase until a maximum has been reached. There are two elements that are important in an evolutionary algorithm.

- The variation operators (recombination and mutation) that create diversity. Different techniques can be used for both of them.

- The selection process of which individuals will be terminated, and which will be parents for the next generation.

Evolution is a process of adaptation. The fitness function that is used for evaluating the individuals tells us something about the requirements that are needed to survive in the environment. To obtain a higher fitness value, the population needs to adapt increasingly more to the environment. Before different methods and operators are explained, the basic algorithm is discussed.

```
for each candidate in the population do
  initialize candidate with random solution
  evaluate each individual
end for
repeat
  EVALUATE population;
  RECOMBINE pairs of parents;
  MUTATE the resulting children;
  REINSERT new candidates;
  TERMINATE the non-parents;
until maximum fitness
```

Algorithm 3.1: The evolutionary process

First, the population needs to be initialized with random candidates. Next, the loop is entered in which the best candidates are selected as parents. After that, the variation operators are applied. First, pairs of parents are recombined. This is a stochastic process. The pieces of the candidates that are recombined are determined randomly. Next, mutation is applied. The parts that are mutated are chosen randomly. The next step in the loop is reinserting the new candidates (the children). Finally, the individuals that were not parents are terminated. This process is visualized in Figure 3.3.

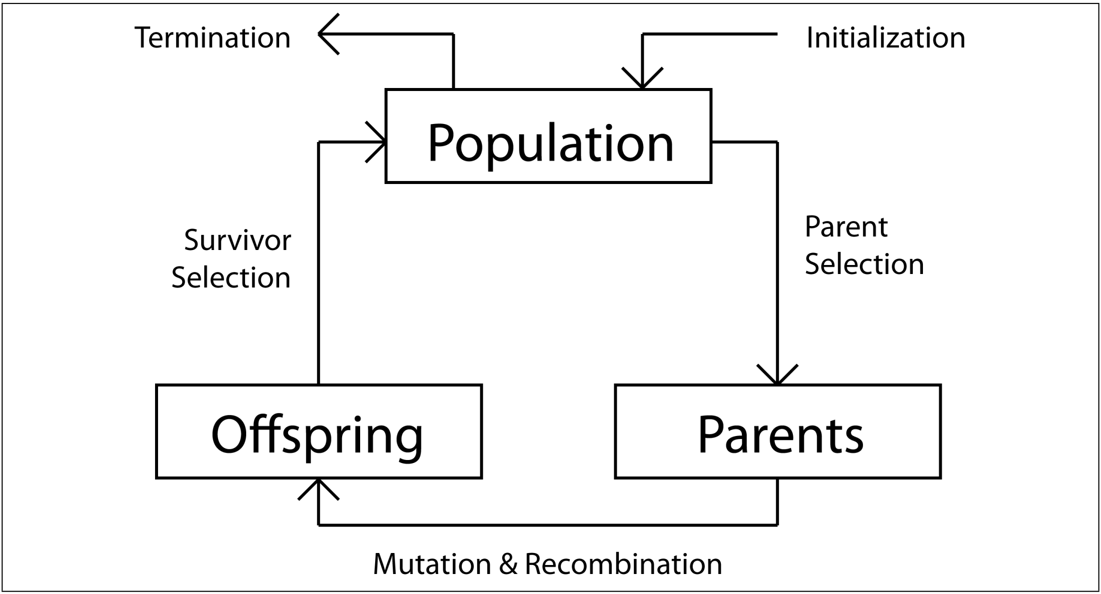

Figure 3.3: The general scheme of an Evolutionary Algorithm

3.3.2 Optimizing the Neural Networks

As already mentioned, there are many different variations of genetic algorithms. In this section, the algorithm that has been used to optimize the neural networks of the bots is explained. It has a variety of components, which are described below.

Representation
Each individual of the population is a bot, which consists of two neural networks, one for weapon selection and one for item selection. So the weights of the neural networks represent a candidate solution.

Fitness Function
The role of the fitness function is to represent the requirements to adapt to. It forms the basis for selecting the best candidate solutions. The fitness function of

the EA is the personal score of each individual, and is the same for both neural networks. The way the personal score is derived is explained in section 3.2.

Population
All candidate solutions together are called the population. The population size in this research is six. Sometimes individuals are removed from the population, and new offspring is added to the population.

Selection
A selection of the fittest individuals needs to be made. The three bots (half of the population) with the highest fitness value are selected as parents. The other three bots are terminated and removed from the population. The selection process is called truncation selection on 50% of the population.

Recombination
Recombination is the technique to generate offspring from parents. Offspring is born by combining the weights of two selected individuals. The weights are averaged, with a chance of 10% of that happening. With three parents, the first child is generated by combining the weights of parents 1 and 2, the second child with weights of parent 2 and 3, and the third of parents 3 and 1.

Mutation
All offspring is slightly mutated with Gaussian values with mean 0 and standard deviation 0.05. By mutating the children, more diversity is guaranteed in the population.

By constantly applying this algorithm, the overall fitness of the population increases. The algorithm is applied approximately once every hour of play in real time, so the fitness values of the bots have time to get to a reliable value.

3.4 Particle Swarm Optimization

3.4.1 Basic Concept

In 1995, James Kennedy and Russell Eberhart published an article called "Particle Swarm Optimization" [16], in which they proposed a new method to optimize non-linear functions. This method was discovered through simulation of a simple social model: the flocking of birds. Previous research showed that the flocking of birds, or the schooling of fish was a function of the animals' effort to maintain an optimum distance between themselves and the neighboring animals.

The authors explain this phenomenon by walking through the conceptual development of the particle swarm optimizer. The first computer simulation they made was based on two properties: nearest neighbour velocity matching and something they called "craziness". A population of agents (birds) was randomly initialized with x and y velocities. In each loop of the program, each agents' nearest neighbour was determined, and assigned that agent's velocities to the agent in focus. Using this simple rule created movements similar to bird flocking, but after a short period of time, the flock headed one direction flying at constant speed. Therefore, a stochastic variable "craziness" was introduced. At each iteration of the program, some variation was added to the x and y velocities. This made the simulation look realistic.

Of course, the flock needs to have a goal, like finding food. Suppose a group of birds are randomly searching for food in an area. There is only one piece of food in the area being searched. No bird knows where the food is, but they do know how far away the food is. The best strategy is to follow the bird that is closest to the food. Kennedy and Eberhart made a model in which each agent remembered the best coordinates in that space, best meaning the closest to the position of the food in that space. That best value is called the personal best, or *pbest*. The second important part of the optimizer is the best position any agent has found so far. This position is called the global best, or *gbest*. The positions of the particle are updated in a way to direct the particle towards its *pbest* and to the *gbest* position.

The authors came to the conclusion that the nearest neighbour velocity matching and the "craziness" factor were not necessary anymore for the swarm to find the food, so they were removed.

3.4.2 The PSO Algorithm

Particle Swarm Optimization (PSO) can be used to solve a wide variety of different optimization problems, especially neural network training. In this research, PSO is used to train the neural networks of the bots. Every bot is represented as a particle. There are a number of characteristics that define a particle:

x_i**:** The *current position* of the particle;

v_i**:** The *current velocity* of the particle;

y_i**:** The *personal best position* of the particle.

Position in this overview doesn't mean an actual position of the bot in the environment, but merely the state of the weights of the neural network. The personal best position, is the position the particle was (a previous value of x_i) where its fitness value was the highest. The fitness value of the particle (bot) is given by its personal score, explained in 3.2. The update equation is defined as

$$y_i(t+1) = \begin{cases} y_i(t) & \text{if } f(x_i(t+1)) \leq f(y_i(t)) \\ x_i(t+1) & \text{if } f(x_i(t+1)) > f(y_i(t)) \end{cases} \tag{3.1}$$

in which f represents the fitness value of the particle. The *gbest* position is called $\hat{y}$. This variable will only be updated when a position is found by a particle that is better than the previous global best solution.

The algorithm makes use of two independent random numbers between 0 and 1, to ensure the stochastic nature of the algorithm. Next, two constants called the acceleration coefficients are used to influence the maximum step size a particle can take in a single iteration of the algorithm. In every iteration of the algorithm, each particle is updated by following the two best values, *pbest* and *gbest*, using the following formulas:

$$v_i(t+1) = v_i + c_1 * r_1 * (y_i - x_i) + c_2 * r_2 * (\hat{y} - x_i) \tag{3.2}$$

$$x_i(t+1) = x_i + v_i(t+1) \tag{3.3}$$

where v_i is the particle velocity, x_i is the current particle solution, y_i is the personal best solution of the particle, $\hat{y}$ is the global best solution of all particles, r_1 and r_2 are random numbers between 0 and 1, and the constants ($c1$ and $c2$) are the acceleration coefficients, both set to 2. The value of v_i is clamped to the range $[-v_{max}, v_{max}]$ to make sure the particle doesn't fly out of the search space. The algorithm consists of repeated application of the update equations above, and is given by

```
for each particle do
  initialize particle
end for
repeat
  for each particle do
    calculate fitness value
    if fitness > pbest then
      set current as new pbest
    end if
  end for
  choose particle with best fitness as gbest
  for each particle do
    calculate particle velocity using Equation 3.2
    update particle position using Equation 3.3
  end for
until maximum iterations
```

Algorithm 3.2: The Particle Swarm Optimization algorithm

First, the particles need to be initialized. This is done by assigning random values to the weights of the neural network of the particle, and setting the velocities to 0. Then the algorithm main loop starts. Initially, one of the particles is assigned to be the best particle based on its fitness. All particles are accelerated in the direction of this particle, but also in the direction of its own personal best solution. Sometimes, a particle will overshoot the best solution, enabling them to explore the search space beyond the best solution. This may lead to a better global solution.

3.4.3 PSO in Neural Networks

In this research, PSO is used to optimize neural networks. Since every bot consists of two neural networks, for weapon and item selection, a bot consist of two particles. The particles for weapon selection of all the bots are the first swarm, and those for item selection are the second swarm. These swarms are optimized separately. The only thing they have in common is the fitness value of the bot. The positions of the particle are represented by the weight values of the neural network, for example: the network for weapon selection has (19 * 25) + (25 * 9) = 700 weights. These weights together represent a position of the particle. The positions are updated in the way described above, with the addition that the values of the velocity vector of the particle are clamped to the values $[-1.5, 1.5]$, to make sure the particle does not fly out of the search space.

3.4.4 Social Behaviour

As already said in the introduction, PSO is a biologically inspired algorithm. Consider the velocity update equation (3.2). The term $c_1 * r_1 * (y_i - x_i)$ can be considered to be the *cognition* part of the equation, since it only takes its own personal experience into account. The last term $c_2 * r_2 * (\hat{y} - x_i)$ is viewed as the social part of the optimizer, since this represents interaction between the particles.

Kennedy performed an experiment using only the cognition part of the equation. The resulting model performed worse than the original, due to the fact that there was no interaction between the particles.

3.5 Evolving Communication

The capture the flag game type is played by a team of bots. Adding a means of communication between these bots could help the team perform better. There already is some communication available, but this is restricted to the division of tasks within the team. There is one team leader, who tells the other bots what task they should perform. These tasks consist of attacking the enemy base, getting the flag, defending the home base and guarding the flag.

In this research, an attempt is made to add a communication channel to the neural networks of the bots. The signal that goes into the network is the output of another neural network, which combines communication signals of all other team members. There is also a communication output node in the network, that is an input to the communication neural network.

This means that there are two extra neural networks to be optimized. There is a communication neural network for weapon selection and one for item selection. The networks for weapon selection are shown in Figure 3.4. The items in red are added with respect to the original weapon selection network.

There is an extra input and output added to the original weapon selection network. The communication network consists of six inputs. One for each bot in the team. Again, twice the number of input nodes is chosen as the number of hidden units in the network.

Figure 3.5 shows the networks for the item selection mechanism. Since the items are evaluated in subsequent network passes, only the communication signal of the eventual chosen item is used as input to the communication network. The communication network works exactly the same as in the weapon selection module.

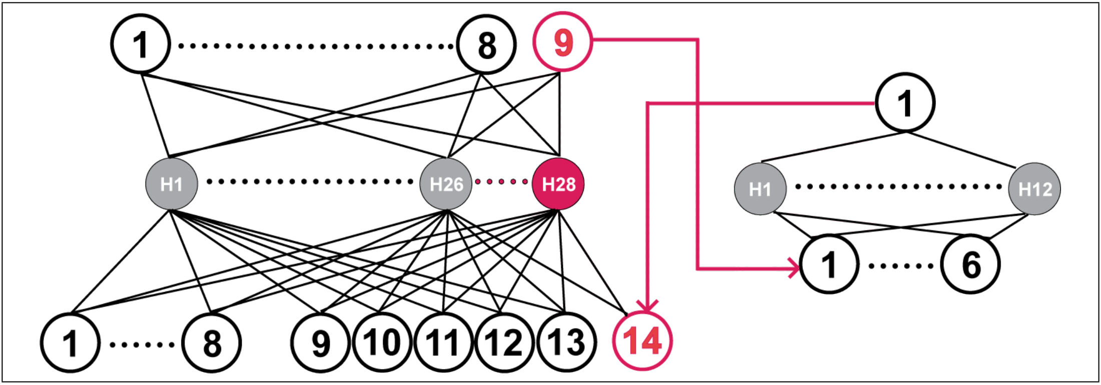

Figure 3.4: The weapon selection network with the communication network

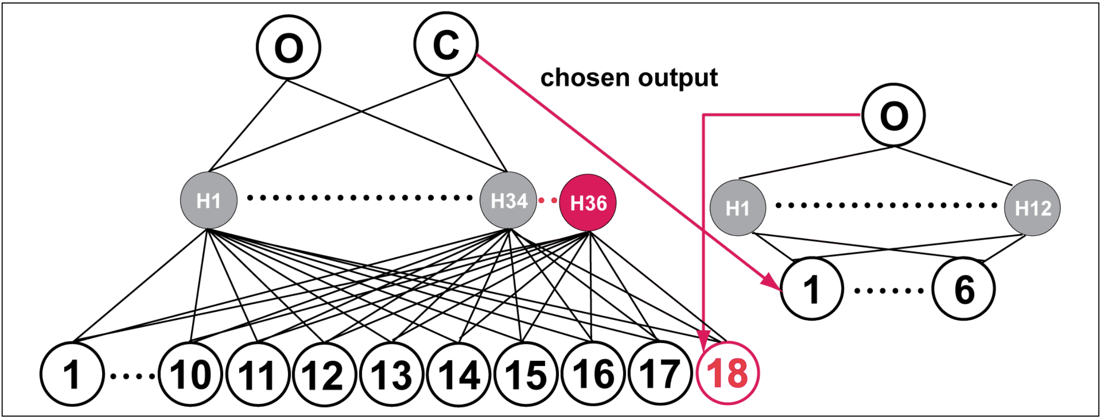

Figure 3.5: The weapon selection network with the communication network

Chapter 4

Experiments

4.1 Introduction

Most of the experiments done in this research all share the same experimental setup. There are several game settings that are of importance. These are given in Table 4.1. The first setting sets the number of players in the game. The twelve bots are automatically loaded into the game. The thirteenth player is needed to get into the game as a spectator, to keep track of the progress of the game. The gametype is set to 4, which corresponds to the capture the flag game type. The map, or level, is equal for all experiments. The third capture the flag map is used. By setting the timelimit and fraglimit to 0, there is no limit. The capturelimit is set to 4,000 captures by one of the teams. The last setting is the timescale factor. This value is set to 40, which means the game is speeded up by a factor 40. This way, we can run one experiment in approximately 10 hours, while in reality, it would last for more than two weeks.

setting	**value**
sv_maxclients	13
g_gametype	4
map	q3ctf3
timelimit	0
fraglimit	0
capturelimit	4000
timescale	40

Table 4.1: Game settings for all experiments

During the game, every 10 seconds the scores are written to log files. Later, this data can be used to generate graphs about the performance of the bots during the game. Algorithm specific settings are given in the next sections.

4.2 Genetic Algorithm

The experiment with the genetic algorithm was run ten times. In the end, the results were averaged. The experimental conditions are summed up in Table 4.2. The population size is six, since there is a team of six bots that is being optimized. The selection procedure is truncation selection on 50% of the population. This means half the bots will generate offspring, and half wil be terminated. Mutation is done by adding Gaussian terms to the weights. The mean and standard deviation of the Gaussian function are 0 and 0.05, respectively. Crossover is done by averaging the weight values of both parents. The chance of this happening per weight is 10%. The number of hidden neurons is approximately twice the number of inputs for that network.

parameter	**setting**
Population size	6
Selection	truncation selection 50%
Mutation type	Gaussian
Mutation mean	0
Mutation SD	0.05
Crossover	average
Crossover chance	10%
Hidden neurons weapon selection	25
Hidden neurons item selection	30

Table 4.2: Parameter settings for the genetic algorithm

4.3 Particle Swarm Optimization

The experiments with the particle swarm optimizer were also repeated ten times, after which the results were averaged. With this algorithm, there are two swarms of particles. Both swarms, one for weapon selection and one for item selection, consist of six particles. There are only three parameters that need to be set. Two acceleration constants, both set to 2. This value is often used in literature. Third parameter is the maximum velocity the particle may have in any direction. This value is set to 1.5.

4.4 Evolving Communication

The experiments with the genetic algorithm with evolving communication has exactly the same experimental setup as the original genetic algorithm. The only addition is the communication network and the extra input and output to the original networks. Also, the hidden units are increased by 2.

4.5 User Study

To check if there is any difference in performance of the optimized bots in playing against other bots or against a human player, a small user study is performed. There are four subjects, who all have experience with playing first person shooter games. The subjects all got 30 minutes of training time, to get used to the level, controls and weapons.

In the first run, the test subject was teamed up with 5 Sarge bots. They played against a team of 6 Sarge bots. The capturelimit was set to 10.

In the next run, the team of the test subject was not changed, but the opponents were replaced by already optimized Darwin (genetic algorithm) bots.

After the experiments with the four test subjects, the two runs are compared.

Chapter 5

Results

5.1 The Genetic Algorithm

The first experiments were performed with the genetic algorithm. The team of six bots optimized by the genetic algorithm (called Darwin), played capture the flag games against a team of six 'Sarge' bots. The experiment was repeated ten times, with a capture limit of 4000. Figure 5.1 shows the captures taken over time.

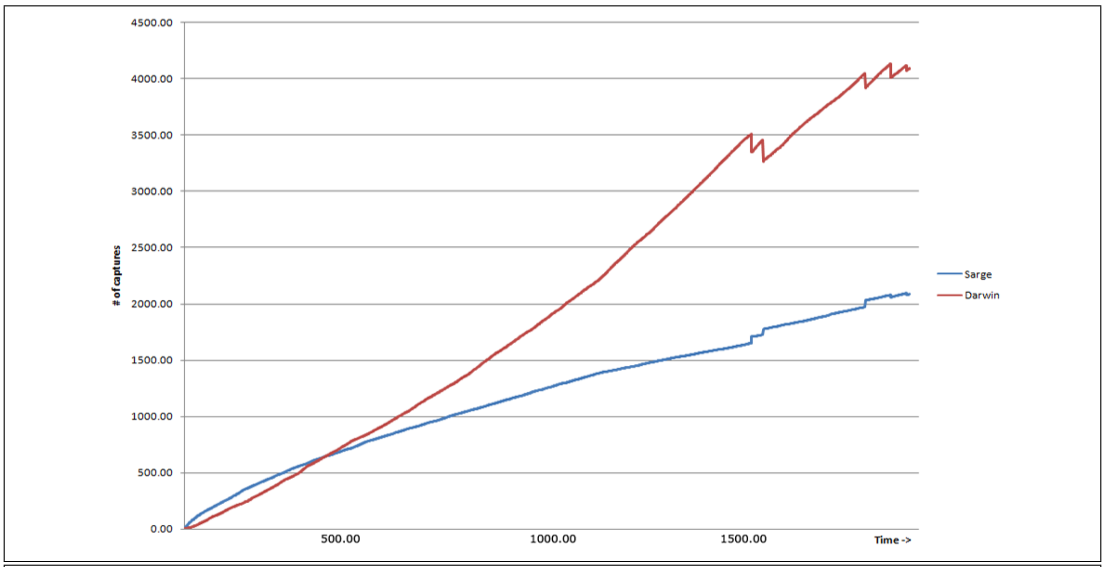

Figure 5.1: The captures taken over time by the two teams.

As we can see in the graph, Sarge is winning from the start. After approximately 650 captures, Darwin takes the lead, and eventually wins the match with 4,000 to a little over 2,000. To be able to see the speed of capturing the flags, Figure 5.2 shows the number of captures per time unit.

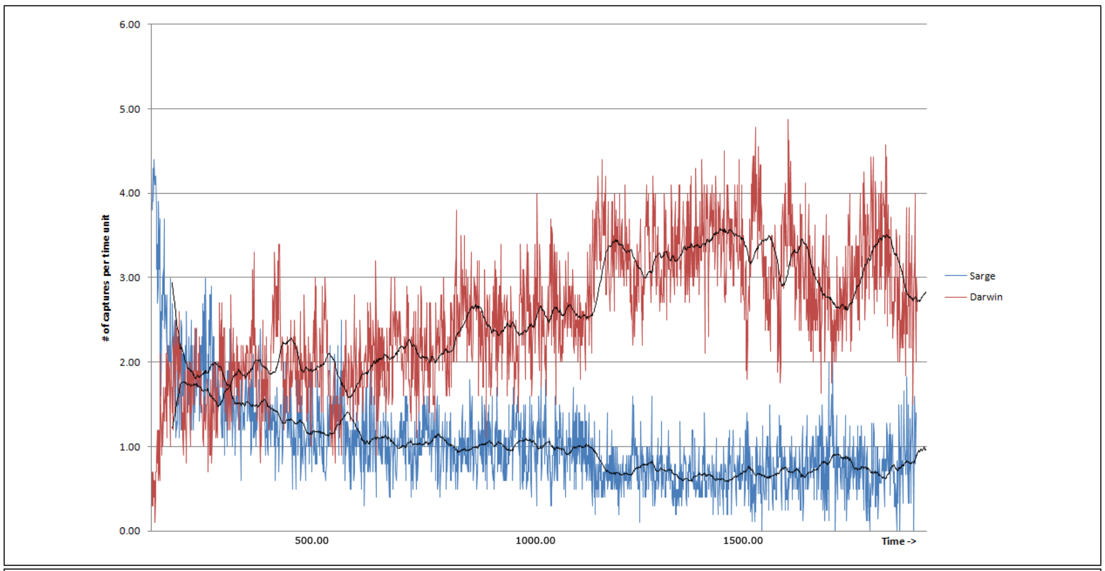

Figure 5.2: The captures taken per time unit by the two teams.

Again we see, that in the beginning, the number of captures per time unit is bigger with Sarge. From the start, this number decreases, while the captures of Darwin are increased. They seem to stabilize around three captures per time unit, while Sarge stabilizes around 0.8 captures. The black trend line is a floating average with a period of 50 time units.

Both graphs show that the genetic algorithm is capable of learning.

5.2 The Particle Swarm Optimizer

The second experiment was done with Particle Swarm Optimization (PSO). The experimental conditions are the same as with the genetic algorithm. The bot using the optimizer is called Kennedy. The number of captures over time is shown in Figure 5.3.

We see that in the beginning, Sarge has the lead over Kennedy. Later, after approximately 1,700 captures Kennedy takes the lead and eventually wins with 4,000 over 2,500. Figure 5.4 shows the number of captures per time unit.

Again we see that in the beginning, Kennedy starts out slow, but the captures per time unit increase, eventually stabilizing around three captures. The captures per time unit of Sarge decrease, stabilizing around 1 capture per time unit.

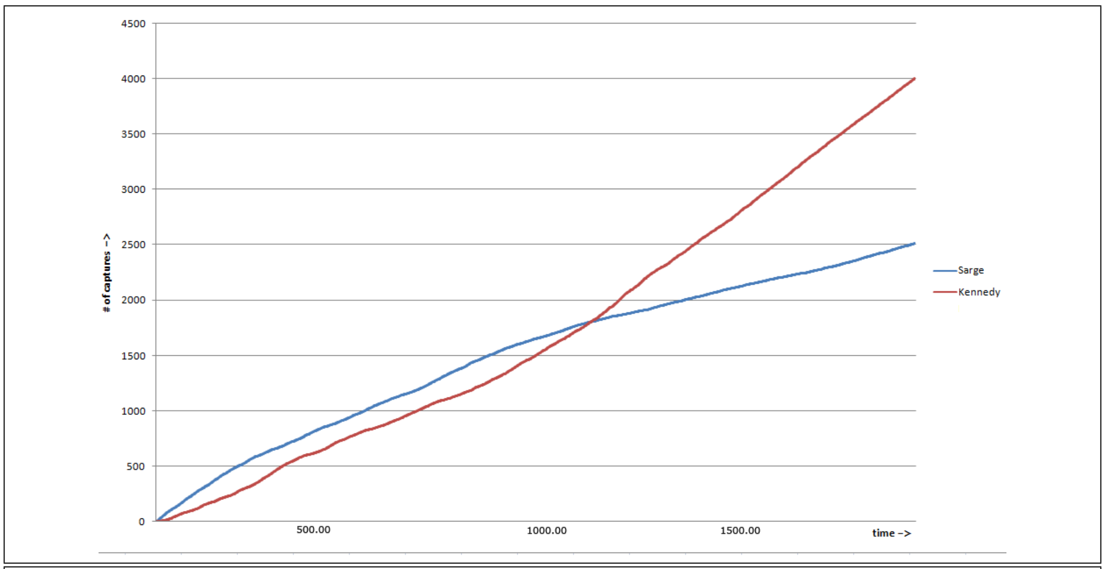

Figure 5.3: The captures taken over time by the two teams.

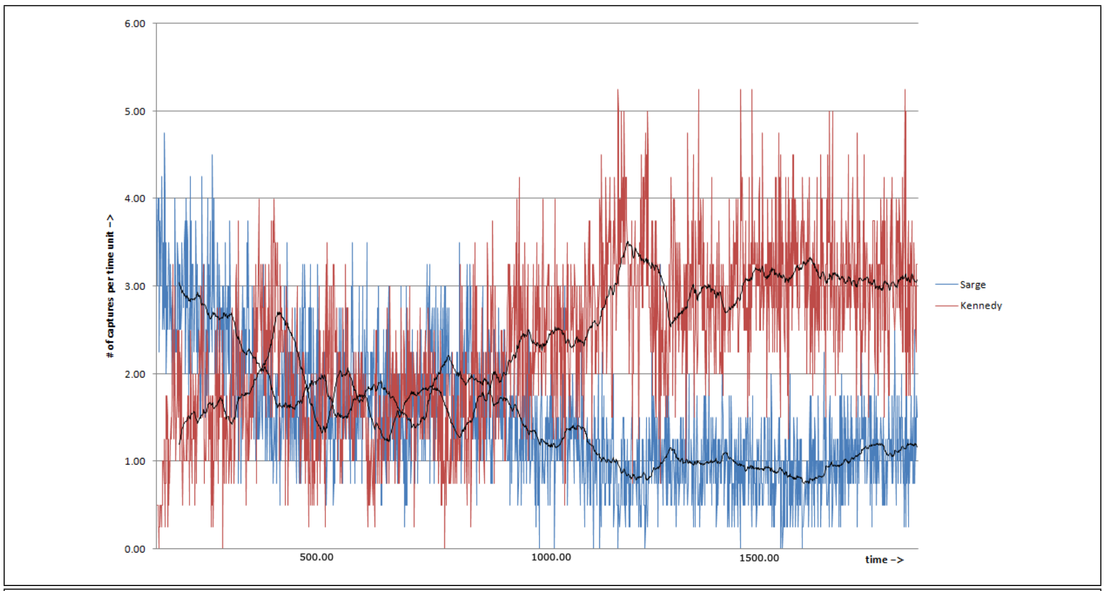

Figure 5.4: The captures taken per time unit by the two teams.

5.3 GA's and PSO Compared

If we compare the results of the genetic algorithm and the particle swarm optimizer, some observations can be made. Both graphs are shown in Figure 5.5. The time scale on the x-axis is exactly the same for both graphs. Two things can be noted. The breakpoint of Darwin is located at approximately 650 captures, and that of Kennedy around 1,700 captures. Also, the difference in captures in the end is 500 captures higher than with Kennedy.

One possible explanation for this difference is the low number of particles. Usually, at least 15 particles are used with PSO. If we had used more particles, the teams of bots in the game would increase in size as well. Since the levels of the capture the flag game type in the game are quite small, increasing the number of players would be unrealistic. Players in the game need a certain freedom of movement to perform well.

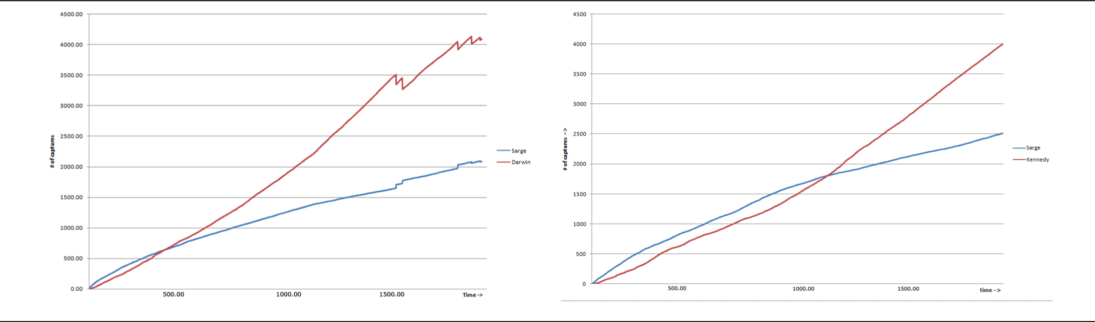

Figure 5.5: Left: the results of Darwin, right: the results of Kennedy

5.4 Evolving Communication

The genetic algorithm was chosen to test the module of evolving communication, since this method performed best in the experiments. The bot with the communication module is called DarwinCOM. Figure 5.6 shows the graph of the captures taken over time.

We see a similar picture as with the other experiments. In the beginning, Sarge has the lead over DarwinCOM. After approximately 700 captures, DarwinCOM takes over the lead and eventually wins with 4,000 against 2,000 captures. The graph with the captures per time unit (Figure 5.7) shows a corresponding picture.

In the beginning, Sarge takes more captures per time unit, but this number decreases over time to a value of 0.8 captures. The number of captures of DarwinCOM increases over time, converging to approximately 3 captures per time unit.

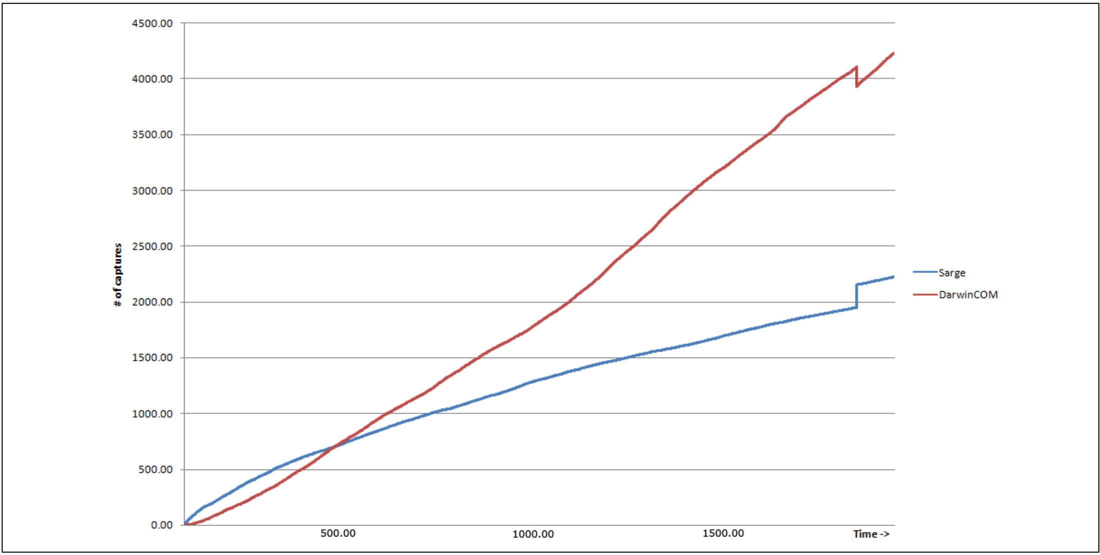

Figure 5.6: The captures taken over time by the two teams.

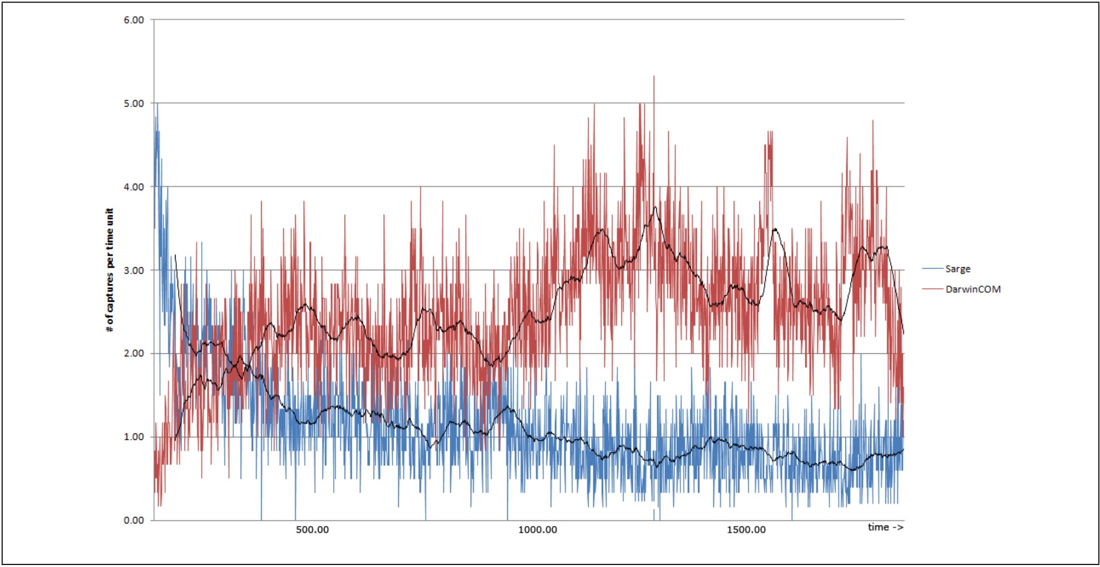

Figure 5.7: The captures taken over time by the two teams.

5.5 The Influence of Communication

The graphs of the genetic algorithm from the first experiment with the current algorithm with the communication unit is shown in figure 5.8.

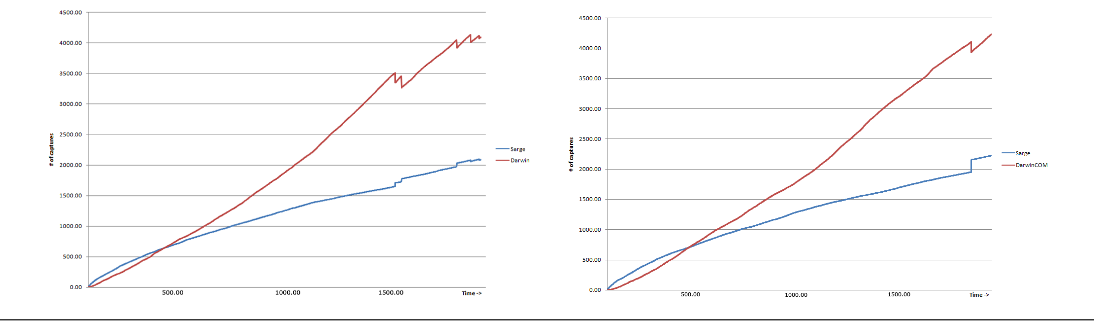

Figure 5.8: Left: the results of Darwin, right: the results of DarwinCOM

There is no noticeable difference between the graphs. The breakpoint is approximately the same, and the difference in eventual score is almost equal as well.

Possible explanation for the absence of an effect is the complexity of the environment. The experiments done by de Greeff [8] were performed in a highly static and basic environment. The levels of this game are also static, except for the events happening in the level. Twelve bots are located on a relatively small platform. This means the environment is constantly changing.

5.6 User Study

To find out if the optimized bots also perform better against a human opponent, a small user study was performed. In the first run, the subjects played capture the flag in a team with 5 Sarge bots against a team of 6 Sarge bots. The capturelimit was set to 10. The results from the four subjects are shown in Figure 5.9.

As can be seen, all players won their matches against Sarge. The mean end score for the team of Sarge bots was 3.25 captures. The mean playing time for all matches was 29 minutes and 45 seconds.

Now we will compare the scores of this first run with a second run in which the team of six Sarge bots (the opponents of the test subjects) is replaced by a team of Darwin bots. Darwin (the optimized bots with the genetic algorithm) was chosen because it performed best in the previous experiments. The Darwin bot is already optimized by playing a game with capturelimit 4,000. These bots are loaded into the game. The results of this run are shown in Figure 5.10.

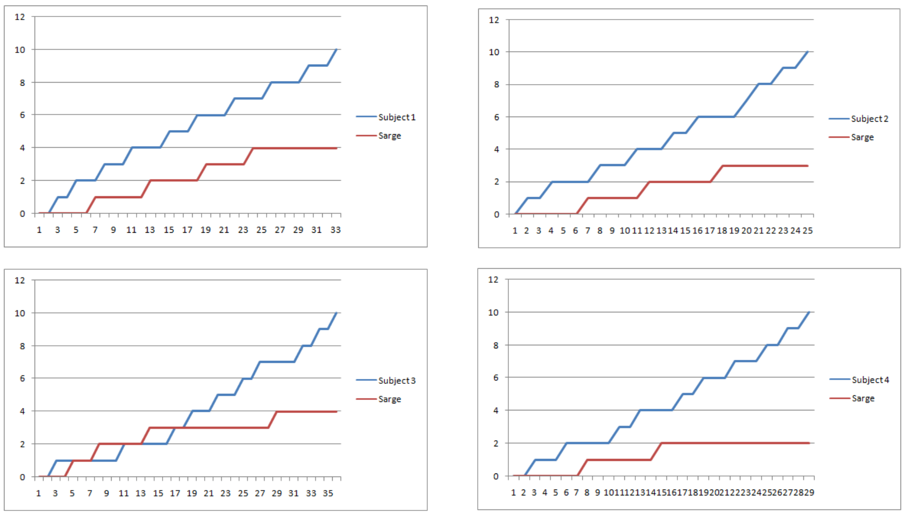

Figure 5.9: The results of the four subjects against a team of Sarge bots.

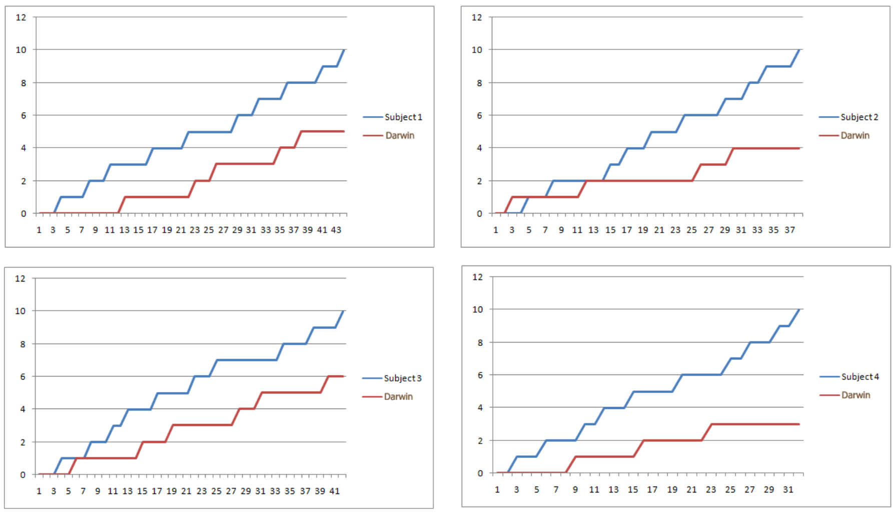

Figure 5.10: The results of the four subjects against a team of optimized Darwin bots.

Again, all test subjects won their match. Some things can be noted though. The mean of captures of the team of Darwin bots is increased by 1.25 to 4.5 captures. More interesting is the mean playing time of the test subjects. This is increased by a little over ten minutes of play. This is an increase of 34%.

Chapter 6

Conclusion

6.1 Conclusion

The goal of this research was described by the research questions in chapter 1. First, the effect of different optimization techniques on bot performance was questioned. We can conclude, that for both the genetic algorithm (GA) and the particle swarm optimizer (PSO), the effect was positive. The bots performed better than the original bots, on which they were based.

There was a difference in performance between the GA and the PSO. The GA got to a good solution far sooner than the PSO. Perhaps there were not enough particles in the PSO to be able to optimize quickly. As already mentioned in the results section, adding more particles would create an unrealistic game. Too many bots would have to play in a too small environment. An attempt was made to let the bots play a tournament game, starting with 32 bots, however, the game loop was not able to cope with the amount of calculations.

The user study also showed that the optimized Darwin bot outperformed the original Sarge bot. The capture count of Darwin was higher than that of Sarge and the playing time was increased with 34%. This means the test subjects had more trouble in defeating Darwin than with Sarge.

The second research question dealt with the influence of evolving communication between members of a team of bots. There was no noticeable difference in performance of bots with and without communication. This is probably due to the fact that the environment is too complex, and the bots did not have enough time to learn it.

6.2 Discussion

The main difficulty with this research is that the source code of Quake III Arena is quite large. A lot of searching in the code is needed to find the exact place where the action needs to take place. Also, debugging is hardly possible, since we have to run the game, and wait for results for hours.

Also, all calculations need to take place in real time during the game. When a new generation needs to be made, lots of calculations on the neural networks need to be done. I've tried to set up a tournament mode with the genetic algorithm, with a population size of 32. It took too long to do the calculations, and the program crashed.

6.3 Future Work

Perhaps, more environmental inputs could be used. Every aspect of the environment could be of importance in making a choice of weapon or item. The most important inputs are already used, but the status and location of other players could also help in making decisions.

If we could increase the actual size of the level, we could also increase the number of members in a team. The particle swarm optimizer might need more particles to be able to optimize faster. A level editor is available for Quake III Arena, but due to the amount of time this would take, this was out of the scope of this research.

In the current research, the only things that are optimized are weapon selection and item selection. There are also other aspects that could be optimized, like combat movement (dodging bullets, retreating).

More could be done with teamwork. In the current game, the tasks are given by a team leader. If two bots get the task to get the enemy flag, they could cooperate in trying to achieve this. This behaviour is currently dealt with by the original source code.

Appendix A

Quake III Weapons

Gauntlet

The gauntlet is the basic weapon every player always carries. It cannot be lost and doesn't require ammunition. The enemy needs te be touched with this weapon. Upon contact, 50 points of damage is inflicted. Usually, this weapon is used to humiliate the enemy player.

Machine gun

A player who respawns into the game already owns a machine gun. The weapon instantly fires when used, and inflicts between 5 and 7 damage points. Maximum ammunition is 200 bullets.

Shotgun

Very useful weapon on close range. A shot consists of 10 pellets. There is a delay between two shots, and inflicts 10 points per pellet. Maximum ammunition for the shotgun is 200 rounds.

Plasma gun

Quite effective weapon on middle and long range. The firing rate is high, and each plasma blob inflicts 20 points of damage. It could also inflict splash damage on impact. The amount of damage decreases with the distance to the splash. Maximum amount of plasma blobs that can be carried is 200.

Grenade launcher

This weapon fires grenades, which are timed explosives which detonate with splash damage. It also detonates when it hits another player. A maximum of 200 grenades can be carried.

Rocket launcher
One of the most popular weapons for advanced players. Rockets inflict severe damage on impact, and also inflicts considerable spash damage. On a direct hit, 100 points of damage is inflicted. The maximal ammunition for the rocket launcher is 200.

Lightning gun
This effective, instantaneous weapon fires a constant burst of lightning in a straight line, over a long distance. It inflicts 80 points of damage for every second an enemy player is hit by it. A player can carry 200 rounds of ammunition.

Railgun
Extremely powerful weapon. It inflicts 100 points of damage on impact. The blast can punch through multiple players. It requires very good accuracy, and is lethal in the hands of a skilled opponent. A player can carry 200 rounds of ammunition.

BFG10K
This weapon is not found in many levels, and if it can be found, it will be hard to acquire. It is extremely powerful, and inflicts 100 points of damage to enemy players, and considerable splash damage to anyone nearby. Maximum ammunition is again 200 rounds.

Quake III Items

HEALTH AND ARMOR

Green health
The small green health item increases the health of the player by 5 points, even beyond the normal maximum of 100 points.

Yellow health
A health increase by 25 points, with a maximum of 100 points.

Gold health
A health increase by 50 points, with a maximum of 100 points.

Mega health
Increases the health with 100 points, to a maximum of 200 points. Health above 100 points will decrease in time, with one point a second.

Armor shard
Increases the armor with five points, even beyond the normal maximum of 100 points.

Combat armor
An armor increase by 50 points.

Heavy armor
An armor increase by 100 points.

TIMED POWERUPS

Battle suit
When picking up this item, the player will be nearly invulnerable to anything. It blocks any splash damage, and the player is no longer vulnerable to lava, slime, drowning and falling damage. Direct hits will inflict half damage.

Haste
The haste item makes the player move and shoot faster than its opponents.

Invisibility
This item makes the player nearly invisible to other players. Only a subtle distortion at the position of the player is noticable.

Quad damage
When in use, every hit will cause four times the damage it would normally do. A disadvantage is that the sound the player makes (by shooting for instance) is alse louder, which means enemies can hear the player coming.

Regeneration
As with the quad damage item, enemies will hear the player coming because a distinctive sound this item makes when in use. Advantage of this item is that it will heal the player automatically to a maximum of 200 points.

CARRYABLE POWERUPS

Medkit

The medical kit is a carryable item, which can be used when the player activates it. It will increase the player's health to 100 points, regardless of the current state.

Personal teleporter

When this item is used, it will teleport the player to a random location somewhere in the level. Extremely handy when the player is in a difficult combat situation.

Bibliography

[1] Id Software Technology Downloads. Quake 3 arena source code, October 2008. http://www.idsoftware.com/business/techdownloads.

[2] Joost Westra. Evolutionary neural networks applied in first person shooters, MSc Thesis 2007. http://people.cs.uu.nl/westra/articles/scriptie.pdf.

[3] R.S.B. Williams and K. Herrup. The control of neuron number. *Annual Review of Neuroscience*, pages 423–453, 1988.

[4] W. S. Mcculloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysic*, 5:115–133, 1943.

[5] Donald O. Hebb. *The Organization of Behavior.* New York: Wiley, 1949.

[6] W Weaver and C.E. Shannon. *The Mathematical Theory of Communication.* University of Illinois Press, 1949.

[7] L Steels. The emergence and evolution of linguistic structure: from lexical to grammatical communication systems. *Connection Science*, 17:213–230, 2005.

[8] Joachim de Greeff. Evolving communication in evolutionary robotics, MSc Thesis 2007. http://www.phil.uu.nl/preprints/ckiscripties/SCRIPTIES /053/greeff.pdf.

[9] Rogelio Adobbati, Andrew N. Marshall, Andrew Scholer, and Sheila Tejada. Gamebots: A 3d virtual world test-bed for multi-agent research. In *Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, 2001.

[10] John E. Laird. It knows what you're going to do: adding anticipation to a quakebot. In Jörg P. Müller, Elisabeth Andre, Sandip Sen, and Claude Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 385–392, Montreal, Canada, 2001. ACM Press.

[11] Stephano Zanetti and Abdennour El Rhalibi. Machine learning techniques for fps in q3. In *ACE '04: Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 239–244, New York, NY, USA, 2004. ACM.

[12] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Evolving neural network agents in the nero video game. In *(CIG '05) Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games.* Piscataway, NJ: IEEE, 2005.

[13] Kenneth O. Stanley, Ryan Cornelius, Risto Miikkulainen, Thomas DaSilva, and Aliza Gold. Real-time learning in the nero video game. In *(AIIDE '05) Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference*. Demo Papers, 2005.

[14] Bakkes S, Spronck P, and Postma EO. Team: The team-oriented evolutionary adaptability mechanism. In *ICEC '04: Proceedings of the Third International Conference on Entertainment Computing*, pages 273–282. Springer, 2004.

[15] Darren Doherty and Colm O'Riordan. Effects of communication on the evolution of squad behaviours. In *(AIIDE '08) Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference.* AAAI, 2008.

[16] James Kennedy and Russell Eberhart. Particle swarm optimization. In *(ICNN '95) Proceedings of ICNN 95 - International Conference on Neural Networks*, pages 1942–1948. IEEE Press, 1995.