Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Rudolf Kadlec

Evoluce chování inteligentních agentů v počítačových hrách

Evolution of intelligent agent behaviour in computer games

Department of Software Engineering

Supervisor: RNDr. Petra Vidnerová, PhD.
Institute of Computer Science,
Academy of Sciences of the Czech Republic

Study program: Computer Science, Theoretical Computer Science

2008

Acknowledgment
I would like to thank to my family, supervisor Petra, Pogamut team, all people behind Java, R, Graphviz, Inkscape and Latex. This work was supported by the grant GA UK 1053/2007/A-INF/MFF.

I declare that I have written this thesis by myself and that I have used only the cited resources. I agree with making this thesis public.

In Prague Rudolf Kadlec

On various places in this thesis there are cited web pages and blog entries. The Internet is a living place, millions of new pages are being published every day and others are disappearing. Fortunately there are services that are trying to preserve this wealth of information for future generations. Hence if some of the referenced pages will be unavailable by the time you are reading this thesis, try using services like the Internet Archive[1] or the Google Archive[2] or some similar service available.

[1]http://www.archive.org

[2]Available through the "Archive" link in the search results at www.google.com

Contents

1 Introduction **7**

1.1 Artificial Intelligence and Computer Games . . . 7

1.2 Possible use of genetic algorithms in games . . . 9

1.3 Structure of the Thesis . . . 11

2 First Person Shooters **12**

3 Related work - use of GA in FPS **15**

4 Methods Used **17**

4.1 Evolutionary Algorithms . . . 17

4.1.1 Selection . . . 19

4.1.2 Crossover . . . 20

4.1.3 Mutation . . . 21

4.1.4 Extensions . . . 21

4.2 Genetic Programming . . . 21

4.2.1 Crossover and mutation . . . 22

4.2.2 Random function generation . . . 22

4.3 Artificial Neural Networks . . . 24

4.3.1 Evolution of Neural Networks and NEAT algorithm . . . 25

5 Bot Architectures **27**

5.1 Finite State Machines . . . 29

5.2 Behaviour trees . . . 30

6 Proposed Evolutionary Bot's Architectures 32

6.1 Functional architecture - high level ASM 32

6.1.1 Example 33

6.2 Neural networks - dodging behaviour 34

7 Implementation 36

7.1 Interfacing UT with Pogamut 36

7.2 Evolutionary frameworks 37

7.3 Functional architecture 37

8 Experiments 39

8.1 Potential Pitfalls 39

8.2 Deathmatch 41

8.3 Capture The Flag 43

8.3.1 Coevolution 44

8.4 Dodging 47

8.4.1 Experiment Setup 47

8.4.2 First Model 48

8.4.3 Second Model 51

8.5 Discussion 55

9 Future Work 62

10 Conclusion 64

Bibliography 67

A List of functions 71

A.1 Behaviour Functions 71

A.2 Sensory Functions - general 72

A.3 Functions for CTF 73

A.4 Mathematical Functions 74

B CD-ROM 75

Název práce: Evoluce chování inteligentních agentů v počítačových hrách
Autor: Bc. Rudolf Kadlec
Katedra: Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. Petra Vidnerová, PhD.
E-mail vedoucího: petra@cs.cas.cz

Abstrakt: V této práci je studována evoluce vysoko i nízkoúrovňového chování agentů v prostředí komerční hry Unreal Tournament 2004. Pro optimalizaci vysokoúrovňového chování v herních módech Deathmatch a Capture the Flag byla navrhnuta a implementována nová funkcionální architektura umožňující popis hráčova chování. Metody genetického programování byly použity pro optimalizaci této architektury. Práce představuje experimenty se standartní evolucí i s koevolucí. V druhé sérii experimentů byl použit algoritms NEAT pro evoluci nízkoúrovňového chování pro vyhýbání se střelám (takzvaný "dodging").

Klíčová slova: počítačové hry, evoluční algoritmy, genetické programování, first person shooter, NEAT

Title: Evolution of intelligent agent behaviour in computer games
Author: Bc. Rudolf Kadlec
Author's e-mail address: rudolf.kadlec@gmail.com
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Petra Vidnerová, PhD.
Supervisor's e-mail address: petra@cs.cas.cz

Abstract: In the present work we study evolution of both high-level and low-level behaviour of agents in the environment of the commercial game Unreal Tournament 2004. For optimization of high-level behaviour in Deathmatch and Capture the Flag game modes a new functional architecture for description of player's behaviour was designed and implemented. Then a genetic programming technique was used to optimise it. Experiments with both standard evolution schema and with coevolution are presented. In second series of experiments the NEAT algorithm was used to evolve low-level missile avoidance behaviour (so called "dodging").

Keywords: computer games, evolutionary algorithms, genetic programming, first person shooter, NEAT

Chapter 1

Introduction

The goal of this thesis is to propose, implement and test models of bot's behaviour suitable for evolutionary optimization that would eventualy simplify the process of bot's creation. Models of both high-level and low-level behaviour were implemented and tested in the environment of commercial game Unreal Tournament 2004.

This chapter introduces current mainstream computer games and discusses the possible use of artificial intelligence methods in these games. Finally the structure of the rest of this work is outlined.

1.1 Artificial Intelligence and Computer Games

The connection between community of academic AI researchers and computer game developers has become strong in the recent years. Cooperation between these two groups is beneficial for both. The game developers are aware of raising demand for improved AI in their games. On the other hand researchers need environments where they can test their models. These environments has to provide rich set of maps with different types of items, they also should be easily customisable and extensible. Mature game engines are usually extensible by scripting languages and they are shipped with variety of content creation tools (map editors, data conversion tools etc.) thus they are a good choice for AI researchers interested in team strategies and embodied agents.

Different types of games are suitable for inspecting different problems. The following list covers most of the current main stream games and suggests which aspects of these games can be interesting for AI researchers.

- First person shooters (FPS) — Main objective is to kill as many opponents as possible. In addition, there are game modes oriented on team cooperation e.g. *Team Deathmatch* or *Capture The Flag* (CTF). In FPS games the player sees the world through the eyes of his avatar. In a single level there can be up to tens of virtual characters. The computer controlled characters are usually called bots or non player characters (NPCs). These games are suitable for simulation of highly reactive decision making of an individual and also for strategies of small teams (e.g. coordinated attack of squad of bots [5]). The models proposed in this thesis were tested in a game of this genre called Unreal Tournament 2004 [1]. Other FPS games are e.g. Unreal Tournament 3 [2], Quake 4 [3] or Doom 3 [4]

- Role playing games (RPG) — RPGs have strong narrative component thus they are ideal for experiments with Virtual Storytelling. Popular game of this genre is Neverwinter Nights 2 [5].

- Strategy games — In strategies there are up to thousands of units simulated. The decision making of individuals is not as sophisticated as in the previous types of games. The emphasis is on high-level coordination of enormous number of units, long term planning, resource management or spatial reasoning [18].

  - Real time strategies (RTS) — The simulation is running in a pseudo real-time. Since the AI must be highly reactive there is a field for use of anytime planning algorithms that can offer an approximation of the solution at any given time of the computation. Well known games of this genre come form the Command & Conquer [6] series.
  - Turn based strategies (TBS) — Between the rounds of simulation of TBSs there is a variable length delay usually of several seconds. Standard planning algorithm can be used during this time. Representative of this genre is for example Civilization 4 [7].

[1]URL: www.unrealtournament.com [11.6.2008]
[2]URL: www.unrealtournament3.com [11.6.2008]
[3]URL: http://www.quake4game.com/ [11.6.2008]
[4]URL: http://www.doom3.com/ [11.6.2008]
[5]URL: http://www.atari.com/nwn2/UK/index.php [11.6.2008]
[6]URL: http://www.commandandconquer.com/ [11.6.2008]
[7]URL: http://www.2kgames.com/civ4/ [11.6.2008]

- Sport simulators — The most popular games of this genre are team sports simulations (e.g. NHL, FIFA, NBA series [8]). The game engines can be used for gameplay analysis and sweet spot detection [30].
- Race car and Flight simulators — the common property of these simulators is complicated physic model driving the movement of vehicles. Neural networks and similar machine learning techniques can be used as the underling model for controlling these vehicles [23, 20].

There is a strong contrast between techniques used in the gaming industry and by researchers. Most popular techniques used in the industry are finite state machines, behaviour trees and in last years planning algorithms. On the other hand researchers are mostly concerned with neural networks, genetic algorithms or machine learning methods, but these methods are still marginal in commercial games [9].

1.2 Possible use of genetic algorithms in games

One can identify at least two domains, where using genetic optimization of bot's behaviour might be beneficial for game development. Firstly, the whole behaviour or just its part (e.g. obstacle avoidance) may be evolved and shipped with the game. Secondly, evolved bots may help to improve the game design during the testing.

The former scenario corresponds to the ideal genetic optimization use case: "specify the goal in a fitness function and let the evolution find the best solution for you". However, there can be found only a few examples of this approach in current commercial computer games (for example computer drivers in the IndyCar™Series[10]).

Games must be in the first place entertaining for the player. This implies that the opponents must be believable, and their abilities must be balanced compared to the human player. Players do not like to play against undefeatable opponents as well as against opponents that they always defeat. In this case the criterion of

[8]Official homepages of these series can be found on http://www.easports.com/ [11.6.2008]

[9]Rigor data are lacking but you can read this informal survey (http://aigamedev.com/discussion/little-used-tools) for list of techniques most commonly used.

[10]Codemasters press release, http://www.codemasters.co.uk/press/index.php?showarticle=3149

optimality is a “fun factor”, i.e. how much attractive the bot is for human players to play with. Capturing this criterion in a fitness function is not a trivial task, therefore this effort is not always successful.

In the latter scenario, genetically optimized bots are used in the phase of game design testing. There are many aspects of the game design that influence the way in which players will play the game. The task of a game designer is to keep the possible ways of achieving success in balance.

In FPS games the optimal behaviour of both the computer controlled bot and the human player is determined mainly by:

- game rules
  - properties of weapons and items in the map
  - mode of the game (e.g. Deathmatch, Capture the Flag as the most common)
  - physics
- map of the level — maze, open space, etc.
- opponents — their skills and abilities

All these components must be chosen carefully with respect to each other. For each combination of these properties there might be different optimal behaviour. The question is whether this is the behaviour that the game designers intended. In the current work flow this property is tested by human players. We think that at least part of their job can be automatized and genetically optimized bots can be utilized for this purpose. The advantage of solutions found by genetic algorithms is that they are constrained only by the game rules and the fitness function, not by “common sense” that shapes humans’ reasoning. Therefore genetic algorithms are able to exploit unintended features of the environment. If such characteristics of the environment would be unrevealed during the testing phase, then they can be fixed and will not appear in the final product. Here, the fitness function is optimizing the “performance” (how many opponents were killed, etc.). This category can be formalized by the fitness function much more precisely than the “fun factor” in the former case.

Time for which the game design is tested by rented testers is negligible compared to the summed time for which is the game played in first weeks after the release. Thus it is more likely that possible problems will be revealed after the release, which is not desired. Automated testing with evolved bots similar to Unit Testing from Extreme Programming [3] conduces to a proposal of new work flow, where the evolved bots are doing a part of the testers' job:

1. Designers prepare the set of game rules and properties of the environment.
2. Bots are evolved in this environment.
3. Behaviour of the best bots is inspected by human testers. If the bots are not behaving as the designers expected, it is recommended to reconsider the game setup and repeat the process.

This approach may reduce the time needed for the testing phase and so designers will be able to build more complex and better tested worlds in shorter time. However, there is a long way to achieve this goal. First, methods for bots' evolution have to be designed. Second, feasibility of these methods has to be verified by extensive tests. Work presented in this thesis focuses on the first stage of the process.

1.3 Structure of the Thesis

The thesis is divided into four main parts. First part (Chapters 1. to 3.) is an introduction. Chapter 2 describes aspects of the FPS games needed for the purpose of this thesis. Chapter 3 presents previous research concerning the use of evolutionary methods in the FPS games.

Second part is theoretical (Chapters 4. to 6.). Chapter 4 overviews the well known methods from the AI used to control the game bots. Chapter 5 shows how bots for these games are usually implemented and Chapter 6 sketches the models of bot's behaviour proposed and implemented in this thesis.

Third part is practical (Chapters 7. and 8.), Chapter 7 discusses the implementation issues, Chapter 8 describes the task bots were evolved for, specifies the architecture of bots in more depth and analyzes their performance.

Fourth part concludes the thesis. Chapter 9 writes about possible future direction of the research in this field. Chapter

Appendix A contains list of functions used in one of the models presented in Chapter 6. Appendix B is a CD-ROM medium with source code of software implemented for this thesis and electronic version of this text.

Chapter 2

First Person Shooters

In section 1.1 a short overview of popular game types has been given. This section concentrates on the domain of FPS games, it describes typical environments found in these games and possible representations of these environments for the game bots.

In FPS games a human player controls a virtual character situated in hostile environment and his main objective is to kill as much opponents as possible. Each character has these properties:

1. Health — Ranges from 0 to 100. When it decreases to 0, then the character dies.
2. Armor — Ranges from 0 to 100. It acts as extra health. When the character has armor and is damaged, then first the amount of armor is decreased, the health level is decreased only after the armor is used up.
3. Weapons — List of weapons that the character has picked up.
4. Ammunition — Weapons need ammunition (ammo) to fire. Different weapons usually need different type of ammunition.

Amount of health, armor and ammo can be raised by picking appropriate items in the map (health, armor and ammo packs).

From programmers point of view the 3D representation of the game locations constructed from triangles is not proper for needs of bot's decision making and navigation. Game engines usually provide special data structures for this purpose.

1. Navigation graph — all paths in the level are represented by an oriented graph. The bot is guaranteed to move safely along the edges of this graph. Navigation through the level can be then transformed to searching a path in this graph, this can be done by an A* algorithm [2] or its hierarchical variant. The disadvantage of navigation graph is that it does not provide any information when the bot is not following the edges. Then a raycasting sensors has to be used to detect obstacles, dangerous cliffs, etc.

2. Navigation mesh — all walkable surfaces in the map are covered by a mesh of polygons [25]. Movement inside the polygon is safe. For higher-level path planning the navigation mesh can be transformed into navigation graph (polygons are transformed to nodes and edges connect nodes representing adjacent polygons). This representation removes the main disadvantage of simple navigation graphs — bots can move freely in the mesh of polygons instead of only following the edges. There are more variants of this approach, e.g. circles can be placed on a important junctions and joint tangents of these circles define safe paths.

Game Unreal Tournament 2004 (UT), which is used as a simulator for experiments in this thesis, uses the navigation graph. Figure 2.1 shows an example of typical environment in UT.

Figure 2.1: Unreal Tournament 2004 screenshot showing a bot firing from a rocket launcher

Chapter 3

Related work - use of GA in FPS

Vast majority of current bots with evolved behaviour fall into one of these categories according to the freedom in changing their controling program often called action selection mechanism (ASM):

Models where the whole ASM is evolved. These models are usually based on neural networks [8] or other offline machine learning algorithms, e.g. 1-NN [21]. They use low level actions (e.g. move forward, look up) and sensory primitives. For example, both models mentioned above are using raycasting for sensing the environment. The bot has only a limited notion about the structure of the environment thus he is unable to navigate over larger distances. Approach used in these models is inspired by the evolutionary robotics, where sub-symbolic sensors are the only available sensors.

However, besides this subsymbolic information, computer games also offer high level symbolic information (e.g. location of all health packs). This allows for higher level of decision making and thus better performance.

Models where evolution is used only on subproblem of ASM. In these models the majority of ASM is hardcoded and the evolution is used for optimization of selected subproblems, e.g. weapon selection [9, 27, 7, 12]. Models of this type are more human competitive because the hardcoded ASM mechanism can take advantage of symbolic information provided by the game engine, including navigation graph, annotation of navigation points (e.g. angle where to expect the enemy) or AI scripts prepared by the designers. Evolution is used as a tuning mechanism for parameterisation of limited aspects of bot's behaviour.

The first type of models has complete freedom of choice and can produce new innovative behaviours, however low level actions and navigation based on raycasting handicaps these models compared to the second type. The second type has already some preprogrammed skeleton of ASM and evolution optimizes only the subbehaviours.

This work presents models from both categories, first a genetic programming is used for whole ASM evolution, second a neural network is used to optimize bot's movement.

Chapter 4

Methods Used

This section describes the algorithms and methods used in construction of our models of bot's behaviour. Readers familiar with evolutionary algorithms, genetic programming, artificial neural networks and the NEAT algorithm could skip this chapter.

4.1 Evolutionary Algorithms

Evolutionary algorithms fall into the category of stochastic optimization algorithms. They are inspired by the seminal work by Charles Darwin "On the evolution of species" [10]. Ideas presented by Darwin were later utilized by John Holland as optimization method in computer science [11], in this context they are known as evolutionary and genetic algorithms (EAs and GAs).

Evolutionary theory supposes that each individual inherits it's traits from his parents. Inherent properties are coded in a structure called genotype. Genotype is a collection of genes, each gene corresponds to some trait of the individual (e.g. hair colour). But the sole knowledge of genotype is not sufficient to exactly determine the perceivable properties of the individual — a phenotype. The phenotype is influenced by both the genotype and the environment. Some properties (e.g. height [1]) are influenced by the extragenous factors.

[1]Since the World War 2 the average height of European population is rising due to better nutrition (the influence of environment) but the genetical predispositions remained probably the same as centuries ago.

Evolution theory describes how the phenotype, through changes to the genotype, adapts to the environment. The theory supposes existence of three mechanisms that makes this possible:

- Selection — all species (individuals with similar genotype) are competing in a race for resources. Individuals better adapted have more offsprings than the worse adapted. The measure of how well the individual is adapted to the environment is called its fitness.
- Crossover — offsprings genotype is a mixture of genes of its parents. Crossover takes place only in sexual reproduction. However some species use asexual reproduction.
- Mutation — genes can be randomly changed by the external effects (e.g. by radiation). Mutation has often destructive nature, but sometimes it can create individuals with advantage over the rest of the population.

Evolutionary algorithms follow the same scheme, only a semantics of some terms is overridden. The environment specifies the problem to be solved and an individual is one possible solution to this problem. The fitness is a measure of how well the solution solves the problem.

Algorithm 1 Evolutionary Algorithm

```
1: $population \leftarrow createInitialPopulation()$
2: **while** stop-criterion is not met **do**
3:     $population \leftarrow selection(population)$
4:     $population \leftarrow crossover(population)$
5:     $population \leftarrow mutation(population)$
6: **end while**
```

Simple skeleton of EA is shown in Algorithm 4.1. In the first step the initial population is created. The most common way to achieve this is to create a random population. Then starts the main loop where the evolution takes the place. The loop runs until the stop-criterion is met. Common stop-criteria are: required fitness value of the best individual, computational time elapsed or convergence of the fitness to assumed bound. In the 3rd step a fitness is computed for all individuals in the population and the frequency of individuals is altered according to these values. In the 4th step the individuals are cross-bred. This step takes place only when a sexual reproduction is used, which is the most common scenario.

However when asexual reproduction is used, then this step is skipped. In the last step the individual are mutated. The crossover and mutation operators are applied only with some given probability. These probabilities are referred to as mutation and crossover rates.

The genes can be encoded as either fixed length structures (e.g. n-tuples of bits, double precision numbers or literals) or as variable length structures (e.g. a graph topology, programs). Fixed length structures constrain the search space of all solutions thus making it more likely to find some local optimum, on the other hand the global optimum can be outside of this prematurely restricted search space. This is where variable length genes can take an advantage.

In the later sections possible implementations of the genetic operators will be described and discussed. Together with extensions to the basic evolution algorithm.

4.1.1 Selection

Selection determines how many offspring will the individual have in the next generation. In general, selection should favour the more fit individuals on the expense of the least fit as this is one of the basic assumptions of the evolutionary theory. Possible strategies to selection are:

- Proportional (deterministic) — number of offsprings is proportional to individual's fitness f_i. There are different strategies how to deal with the rounding.
- Roulette (stochastic) — let f_i be fitness of the i-th individual and N number of individuals in the generation then the probability p_i of the i-th individual being selected to the next generation is $f_i / \sum_{i=1}^{N} f_i$. The next generation is chosen by playing the roulette with probabilities p_i N times.
- Tournament — the individuals are drawn into random pairs (or n-tuples). After the match is played the winner advances to the next generation. This has to be repeated N times.

The fitness function f can be computed:

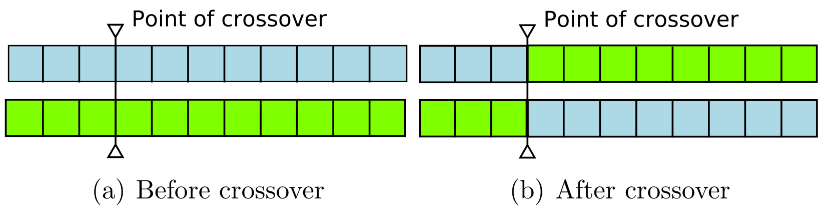
- Directly through the objective function — the performance of each individual is measured by the objective function h, hence $f = h$.

- Rank fitness — if we have total ordering on performances of the individuals and if we sort them in an ascending order, then the fitness f_i of the i-th individual whose index in the ordered sequence is k is $k/\sum_{j=1}^{N} j(j+1)$, where N is number of all individuals. The total ordering of individuals can be obtained through ordering by values of objective function for these individuals. Alternatively a play-off tournament (e.g. used in tenis) is played between the individuals. At the begining the individuals are drawn into random pairs. After the match is played the winner advances to the next round. The disadvantage of this method is that the individuals are only partially ordered, the ordering can be augmented to total ordering although it will not be as accurate as the first alternative (the defeated finalist is ranked as second but the second best individual is one of those that lost with the tournament winner and this could happen in the very first round). The advantage is that only $N * log(N)$ matches has to be played to find the winner.

4.1.2 Crossover

In sexual reproduction the genes of newborn individuals are combination of their parents' genes. Supposed that the genetic information is coded as a linear sequence of features, then the most simple (and most often used) method of crossover is one point crossover. One point crossover chooses random point in the gene and swaps the two parts determined by this point (see Figure 4.1).

One point crossover is a special case of n point crossover where n crossing points are randomly selected and the regions between these point are swapped between the individuals.



(a) Before crossover (b) After crossover

Figure 4.1: One point crossover in linear coding

4.1.3 Mutation

Mutation introduces into the population new features (that may have not been introduced only with selection and crossover). This is desired mainly when the whole population has converged to some local optimum. Without mutation it will not be possible to jump off this optimum and explore another areas of the search space.

In linear coding the most common implementation of mutation is: iterate over all features in the chromosome, change each feature with probability p_{mut}, if the feature is to be mutated then alter its value.

4.1.4 Extensions

Basic scheme of GA as presented in Algorithm 4.1 is often extended by elitism. In elitism n best individuals advance to the next generation without being crossovered and mutated. Elitism eliminates the destructive fallout of these operators on the best individuals, whose fitness would be probably only decreased.

In vast majority of use cases the most time consuming operation in GAs is the computation of fitness values. Parallel implementations of GAs can reduce time required for this part of computation and they scale linearly to number of provided computational units. Overview of different parallelization schemas is in [6].

Speciation is a technique that tries to minimize the destructive effect of the crossover operator. In some scenarios the solutions coded in the genes become so diversed after a few generations that there is no meaningful way how to crossover those genes. In such cases it is convenient to define metrics on the genomes that defines how compatible they are. Two individuals can be crossovered only if theirs compatibility measure is above certain threshold, they are said to belong to the same species.

4.2 Genetic Programming

Genetic Programming (GP) is a subclass of Evolutionary Algorithms where the individuals being optimized are computer programs. GP was popularized mainly by John Koza [16] and it has proved to be successful in many domains.

Programs can be coded in various ways. Linear coding [4] stores programs as sequences of instructions, tree coding [16] stores the program as a tree describing the functional representation of the program. Since all Touring complete programs can be expressed through functions, both approaches are equivalent with respect to their expressive power.

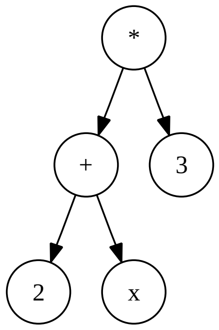


Figure 4.2: Tree for expression $(x+2)*3$

The search space of all possible solutions is determined by the language L. The choice of L is important and should be considered with great care. L must be powerful enough that a solution of desired quality could be expressed in it, on the other hand it should restrict the search space as much as possible. These two requirements are usually contradictory. Instead of common programming languages like C or Java the language of L is usually a domain specific language designed with the problem in mind. These languages are often build on top of functional languages like Lisp.

In the further text the case of tree coding of programs will be described more in depth. Tree coding assumes that all elements of L are functions. In that case all valid expressions constructed from L can be directly coded as trees. Figure 4.2 show one such example. The original concept of GP as presented by Koza assumes that all functions have the same return type (e.g. double). But the model can be extended to functions with various return types.

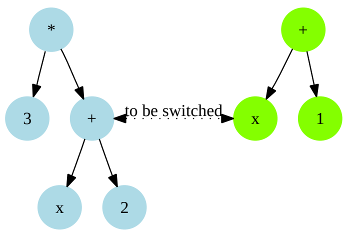
4.2.1 Crossover and mutation

Crossing over two trees is implemented as switching random subtrees as shown on figure 4.3. In typed genetic programming only subtrees of the matching type can be switched in order to produce valid offsprings.

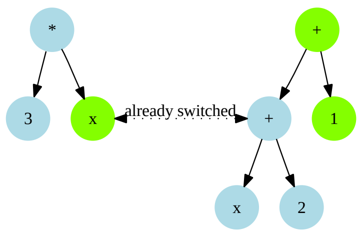
Mutation replaces random subtree with new randomly generated tree of matching type. Figure 4.4 shows one example of mutation.

4.2.2 Random function generation

Procedure for generation of random functions (listed in Algorithm 2) is used for creation of initial population and in the mutation. It generates random tree of

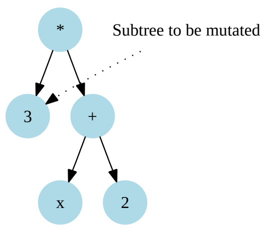


(a) Before crossover

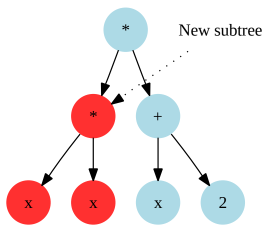


(b) After crossover

Figure 4.3: Example of crossover of expressions $3 * (x + 2)$ and $x + 1$



(a) Before mutation



(b) After mutation

Figure 4.4: Example of mutation of expressions $3 * (x + 2)$ into $(x * x) * (x + 2)$

functions with matching type and desired maximal depth or ends with $\perp$ if such tree is not constructible from the provided set of functions F.

Algorithm 2 generateRandomFunction

```
Require: $d$ — max depth of the generated function
Require: $\Phi$ — type of value generated by the function
 1: if d = 0 then
 2:     return $\perp$ Fail, no such tree can be constructed
 3: else
 4:     $candidates \leftarrow$ sequence of all $f \in F$ whose return type is $\Phi$
 5:     Permutate(candidates)
 6:     for all $f \in candidates$ do
 7:         $params \leftarrow$ sequence of all parameters of $f$
 8:         $i \leftarrow 0$
 9:         for all $p \in params$ do
10:             $\Psi \leftarrow$ type of parameter $p$
11:             $g_{i++} \leftarrow generateRandomFunction(d-1, \Psi)$
12:         end for
13:         if $\neg \exists i : g_i = \perp$ then
14:             return $f(g_0, g_1, ...g_n)$ all subfunctions were constructed
15:         end if
16:     end for
17:     return $\perp$ failed
18: end if
```

4.3 Artificial Neural Networks

Artificial neural networks (ANNs, or just NNs) are biologically inspired computational model capable of interpolating arbitrary function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$. Basic computational unit of NN is one neuron. Each neuron has at least one real input and exactly one output. The output y of a neuron is computed by equation:

$$y = f(\sum_{i=1}^{N} x_i w_i); w_i, x_i \in \mathbb{R}$$

Where f is an activation function, x_i is a value of i-th input, w_i is a weight of i-th input and N is a number of neuron's inputs. Activation function is usually chosen among bounded nonlinear functions (e.g. sigmoidal functions, radial basis function, etc.) Outputs of neurons connect to inputs of other neurons and thus the network is formed.

There are many types of NNs that differ in topology and exact evaluation algorithm. Common are layered NNs, in this topology there are three distinct groups of neurons called layers — input layer, hidden layer(s) and output layer. The pattern from $\mathbb{R}^n$ is presented to the n neurons in the input layer and the response from $\mathbb{R}^m$ for this pattern is output of m neurons in the output layer. Between the input and output layer lies hidden layers.

To sum up findings from the previous paragraphs, each neural network is specified by its:

- Topology
- Weights
- Activation functions

In the process of learning values of these parameters are searched in order to compute optimal responses for the given patterns. In supervised learning the desired responses are known and the optimality measure is based on distance between these responses and responses of the network. In unsupervised learning the task is to minimize specified utility function describing expected model of the data. Similar learning paradigm is reinforcement learning. Outputs of the network are mapped onto actions that change agent's state. Each state has associated reward value and the overall goal is to maximize the summed reward.

4.3.1 Evolution of Neural Networks and NEAT algorithm

In cases where no analytical methods (e.g. backpropagation) are applicable, the evolutionary algorithms can be used as a learning method for NN.

The most basic approach to evolution of NN is to evolve weights of a network with fixed topology. The drawback of this approach is that the chosen topology can be too complex thus making the search space unnecessarily huge or the topology is too simple for given problem. Both cases result in network's poor performance.

This basic scheme can be enhanced by including the topology of network into the genotype. Then the question how to crossover two networks with different topologies arises.

The NEAT algorithm [22] specifies how to explore the space of different topologies and how to meaningfully crossover them.

Main features of the NEAT algorithm are:

- Incremental complexification — in the first generation network starts with the minimal topology (fully connected input and output layer, no hidden neurons). New neurons and inter neuron connections are incrementally added by mutation operators through the course of the evolution.
- Genome with history tracking — the network's topology is encoded in linear genome. Genes representing neurons and connections have associated so called *innovation numbers*. When a new neuron or connection gene is introduced, it receives a global innovation number by one higher than the last added gene. Genes with the same innovation number origin from the same common ancestor thus they will likely serve a similar function in both parents. NEAT's crossover operator exploits this observation, genes with matching innovation numbers are randomly chosen from both parents for the new offsprings, the rest of genes is taken from the more fit parent.
- Speciation and fitness sharing — based on the topology individuals are arranged into non-overlapping species. Individuals mate only within the same species — this raises a chance that the crossover will produce meaningful offspring. Number of offsprings is proportional to the summed fitness of the species — this protects more complex networks that have usually lower fitness at the beginning.

Chapter 5

Bot Architectures

In the context of the FPS games the computer controlled opponents are called bots. Bots are implemented as an embodied agents as defined by Wooldridge and Jennings [28]. Bots perceive the game environment through provided senses and they can influence the environment by provided set of actions. The data flow between the bot and the environment is depicted on the Figure 5.1.

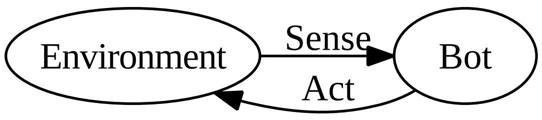


Figure 5.1: The Act-Sense loop

Bots are designed to accomplish large variety of tasks (e.g. patrolling, attacking). The objective for which the bot was designed influences structure of its action selection mechanism (ASM). However there are some common building blocks that can be found in majority of FPS bots. It is convenient to think about the architecture of bot's ASM in the terms of layered design [26]. Common layers of this design are shown on Figure 5.2.

Each layer works on different level of abstraction and each module is responsible for different aspect of bot's behaviour or reasoning:

- 1st layer is responsible for planning bot's actions. It chooses from behaviours implemented on the lower level of abstraction. This layer can be for example responsible for team tactics, long term planning, knowledge representation etc.

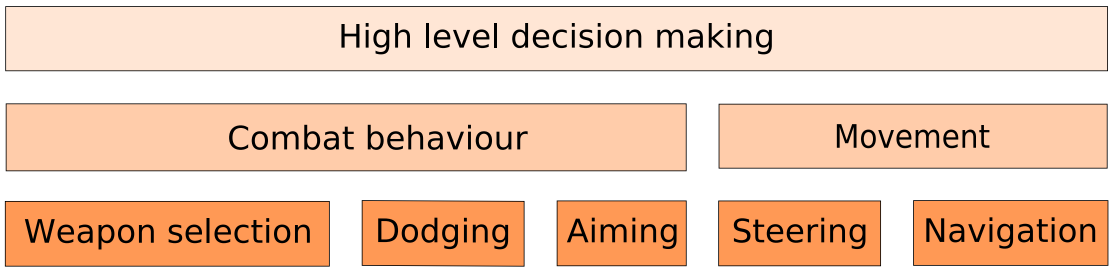


Figure 5.2: Conceptual layers of bot's behaviour

- 2nd layer implements functionally homogeneous behaviours, these are for example:
  - Combat behaviour — combat situations are integral part of the gameplay of the FPS games
  - Movement — movement is typically the most frequently used behaviour that is responsible for planning of the path and also for fluent movement along this path.
- 3rd layer implements the smallest conceptual blocks of bot's behaviour, these can be:
  - Weapon selection — which weapon is the most appropriate one given the distance to the enemy and enemy's characteristics (e.g. some enemies can be resistant to certain type of damage).
  - Dodging — how to best avoid the incoming projectile.
  - Aiming — at which location should the bot shoot in order to hit moving enemy
  - Navigation — planning of path from bot's location to any other place in the map (e.g. to the nearest health pack that will raise bot's health level).
  - Steering — how to avoid obstacles (players or any other movable objects in the level) laying in the preplanned path.

This list is not exhaustive and it enumerates only the most common modules.

There are many possibilities how to implement these individual modules. In the next sections a finite state machines and behaviour trees will be discussed.

Both these techniques are popular in game development community. They are easy to grasp and hence can be utilized even by game designers without computer science degree.

5.1 Finite State Machines

Finite state machines (FSMs) [13] are commonly used model for describing the bot's behaviour. In most cases the FSMs used in computer games are extension of standard FSMs from the automata theory. For each state of the automaton there is an associated script that is being executed as long as the automaton remains in this state. Transitions between states occurs when the associated formula is evaluated to be true.

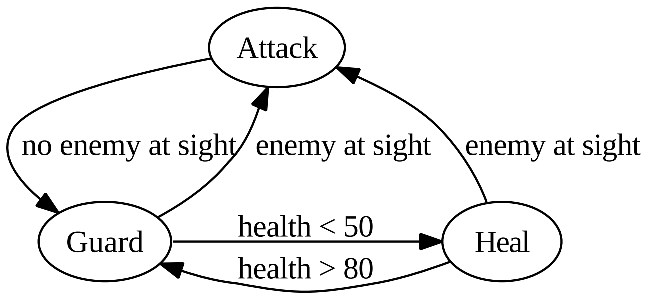


Figure 5.3: Example of FSM controlling a guard bot. States Guard, Heal and Attack have associated scripts. For example the script for the Heal state could find the nearest health pack and pick it.

One of the disadvantages [1] of FSMs is that there can be up to $n(n+1)/2$ transitions where n is number of the states. This can be overcome by Hierarchical FSMs, where states are grouped to higher level states and there are transitions between states on this level. FSM were for example used in the game Halo 2 [14].

[1]Other disadvantages of FSMs are discussed on http://aigamedev.com/questions/fsm-age-is-over

5.2 Behaviour trees

Behaviour trees are another popular model for description of bots' behaviour. Leaves of the tree represent atomic behaviors that can be executed straightaway. Internal nodes represent behaviours that can be decomposed into smaller subbehaviours. They act as arbiters that decide, which of their children will be executed. Hierarchical nature of behaviour trees reduces number of transitions compared to the FSMs.

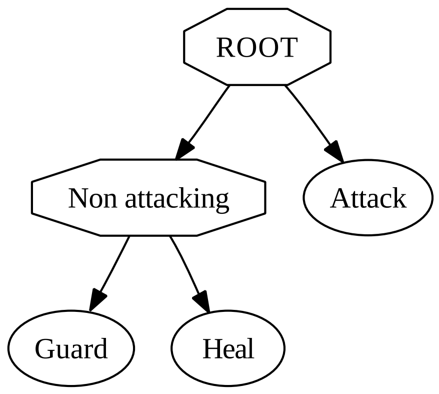


Figure 5.4: Example of behaviour tree controlling a guard bot. The same algorithm coded in FSM is shown in figure 5.3. Octagonal nodes are internal nodes — arbiters.

The tree can be evaluated:

1. Top - down — computation starts at the root node. The root selects only one of its children for execution. This repeats for each selected internal node until a leaf is selected. Then an action proposed by the leaf is executed in the environment. The advantage of this approach is well defined behaviour and computational speed, only one path from the root to a leaf is evaluated.

2. Bottom - up — computation starts at leaves. In this scenario each leave computes the proposed action and passes it to the parent, parent node chooses among all proposed actions and passes the winning action up. This repeats until the root is reached. The action passed to the root is then executed. In this approach the whole tree has to be evaluated.

Bottom-up evaluated behaviour trees can be modified to allow for compromise solutions which, at least in certain scenarios, enhances their performance [24].

Chapter 6

Proposed Evolutionary Bot's Architectures

Previous chapter defined possible bot architectures, this chapter shows two models proposed in this thesis in order to allow genetic optimization. The first model is suitable for high-level behaviour optimization through evolution of behaviour trees. The second optimizes low-level dodging behaviour with use of neural networks. Behaviour trees are better suited for evolution of high-level decision making than the neural networks. Whole behaviours can be exchanged between individuals by a mutation operator as they are represented by single subtrees. This property does not apply for NNs. On the other hand NNs are supposed to be better when it comes to approximation of real functions, which is the case of dodging behaviour.

6.1 Functional architecture - high level ASM

The behaviour trees as presented in Section 5.2 were used as the framework for the functional architecture.

Behaviour trees can be directly translated into functional representation and hence genetic programming methods can be used for their optimization. In contrast to neural networks and other "black box" models, genetic programming leads to solutions in a form of a program that is even human readable if the initial set of functions is chosen appropriately.

Architecture similar to our functional model has already been tested in the

Robocode simulator [29] where the robots were trained for simple combat tasks.

Our ASM architecture has a form of a tree containing possibly three types of functions:

- Behaviour functions — these functions compute the action to be performed in the environment. Their return type is always a tuple $\langle action, its\ suitability \rangle$, we call this type *BehResult*. There are two types of behaviour functions:
  - Primary behaviour functions — primary functions represent atomic behaviours, e.g. attackPlayer(Enemy) function returns the best action that can be issued in order to attack the given enemy (this can be changing the weapon, firing, etc.).
  - Secondary behaviour functions — these take two *BehResult*s as parameters and their return type is also *BehResult*. More complex behaviours can be constructed with use of secondary behaviour functions.
- Sensory functions — their return value is typically a floating-point number normalized to $\langle 0, 1 \rangle$ (distance to a player, bot's health, etc.) or a game specific data type (e.g. function nearestEnemy returns handle to the nearest player from the opposite team). Sensory functions are used to parameterize the actions returned by primary behaviour functions (e.g. attackPlayer(nearestEnemy())). Together with mathematical functions they can also influence the suitability of actions.
- Math functions — $+, *, 1 - x, \sin, \min, \max, \text{constant}$. These functions are used to combine floating-point encoded senses.

The functions used in our experiments are listed in Appendix A.

6.1.1 Example

Figure 6.1 shows example behaviour tree that can be constructed from the presented set of functions. Bot controlled by this behaviour tree will be picking health or ammo if the enemy is not present. If the bot sees the enemy, then he will attack him and when his health decreases under 19%, he will try to escape from the combat.

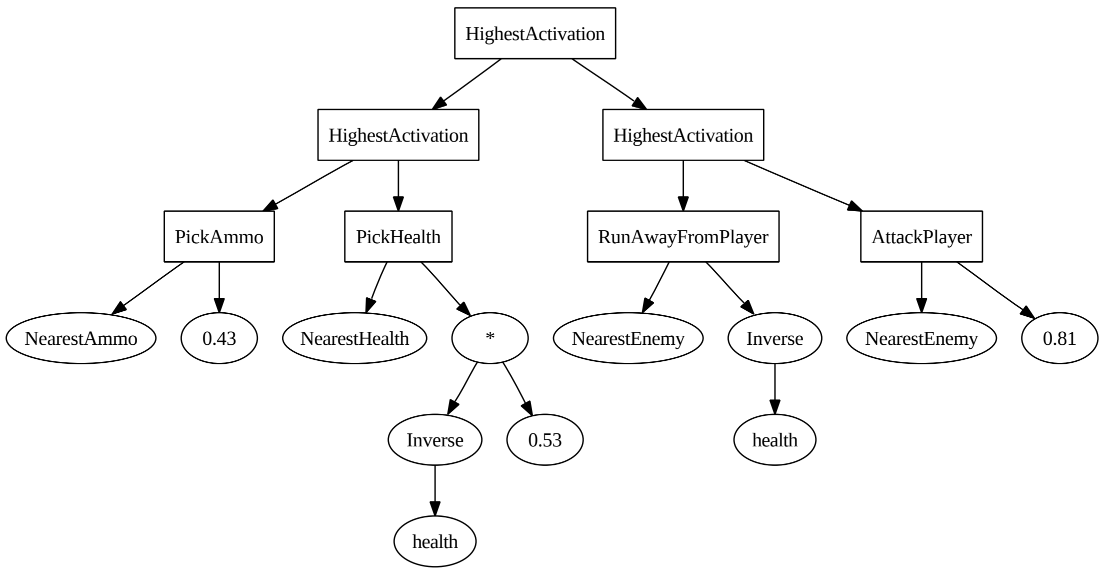


Figure 6.1: Example behaviour tree constructed from the provided set of functions

6.2 Neural networks - dodging behaviour

As mentioned in Chapter 5, dodging is one of bot's basic skills. Most of the weapons in UT have infinite projectile speed but there are also some weapons with high damage whose projectiles have finite speed (e.g. Rocket Launcher). When a player is under fire from such weapon, he has a chance to avoid the projectile even if it is initially headed in his direction. This behaviour is called dodging.

When a fired projectile is likely to hit a bot, the UT notifies him about this situation by special event. The event carries this information:

1. Estimated time till the projectile's impact
2. Angle relative to the bot's direction under which the projectile is coming
3. Radius where the bot will be damaged after the projectile explodes
4. Location from where the projectile was fired
5. Vector specifying velocity of the projectile

For optimization of dodging behaviour a feed-forward NN was used. Although different sets of inputs were chosen for neural networks from the two presented experiments the network's output $x \in \langle 0, 1 \rangle$ was always transformed to $\alpha = (2x - 1)\pi$ which was used as an angle of bot's movement in the next time step.

Inputs of the NN were: information about bot's location (raycasting, distance to the shortest path) and information about the incoming projectile. The details about the chosen inputs are in Section 8.4 which describes the experiments performed.

Chapter 7

Implementation

7.1 Interfacing UT with Pogamut

The experiments presented in Chapter 8 were conducted in the environment of the commercial game Unreal Tournament 2004 (UT). Even though UT provides its own scripting language UnrealScript, the infrastructure was coded in Java and connected to the UT through the Pogamut platform [15]. The Pogamut platform was developed at the Charles University in recent years in order to simplify connection of new bots to the game engine. The Pogamut platform features library of sensoric and motoric primitives, log management and a plugin for the Netbeans^{TM}IDE. This features simplify the bot development and reduce the time needed for debugging bot's behaviour. The platform is based on the well known GameBots [1] interface and adds to it a Java library build on top of the GameBots protocol. The Pogamut platform is free for non-commercial and non-military use and can be downloaded from the Pogamut Homepage [1].

UT is a realtime environment hence it is not suitable for genetic algorithms as it stands. Flow of the time can be adjusted but there is no option "run as fast as possible".

To bypass this disadvantage the so called "Pogamut GRID" was implemented as a part of this thesis. Pogamut GRID enables experimenter to run more experiments in parallel. Experiment is a small program that defines which bots should connect to the game, which features of the gameplay will be observed (e.g. health of bots, bot's distance to defined target etc.) and conditions terminating the ex-

[1]Pogamut Homepage, URL: http://artemis.ms.mff.cuni.cz/pogamut [2.7.2008]

periment (e.g. elapsed time). Definitions of the experiment are send from a client computer to a driver computer that is a gateway to the grid. Driver resends the definitions to the connected nodes where the experiments are executed. Results of the experiment (the observed features of the gameplay) are then send back to the client computer. Pogamut GRID is build on top of the Java Parallel Processing Framework [2]. JPPF is a general framework for building GRID applications in Java, it takes care of the network communication, fail recovery, node management and other features common to all GRID applications.

7.2 Evolutionary frameworks

For experiments with the NEAT algorithm an already existing implementation called Another Neat Java Implementation [3] was used. ANJI is build on top of a Java Genetic Algorithms Package [17] a general framework for genetic computations in Java, hence ANJI sources and architecture are more readable for programmer already familiar with JGAP, than sources of other Java NEAT implementations like JNEAT [4] or NEAT4J [5] which implement their own evolutionary framework. This was a main reason why the ANJI implementation has been chosen.

For genetic programming experiments a custom framework exploiting advanced features of the Java programming language like generics, introspection and annotations was implemented. JGAP has also support for genetic programming but the code is written in Java 1.4 which lacks the features mentioned above.

7.3 Functional architecture

The functional architecture was implemented in the Java programming language. Java is not a functional programming language [6], this means that functions are not first class objects, thus they can not be passed by reference. In object oriented approach this can be overcome by common ancestor of all functions called for

[2]URL: http://www.jppf.org [12.6.2008]

[3]ANJI, URL: http://anji.sourceforge.net [2.7.2008]

[4]JNEAT, URL: http://nn.cs.utexas.edu/soft-view.php?SoftID=5 [2.7.2008]

[5]NEAT4J, URL http://neat4j.sourceforge.net/ [2.7.2008]

[6]However there is a functional languages Scala that runs on top of the Java Platform, see http://www.scala-lang.org/ [24.7.2008]

example $Function < T >$. The $Function$ class has a method *T call()* where T is the type parameter of the function class (i.e. $BehResult$ in the case behaviour functions). All parameters of the function must also be of the type $Function$, parameters of the function are distinguished from the ordinary class fields by a custom annotation. The annotation also contains information about the desired return type of the function since it can not be inferred from the compiled class files [7]. The implementation of Algorithm 2 uses this annotations to infer the return types of function classes at the runtime.

[7]Up to Java 1.6 (the current release of Java) the language uses reification of generic types, this means that all type variables are erased after the compilation and they are not available to the Java Reflection API. However this may change in the 1.7 release of Java, see http://tech.puredanger.com/java7/ [23.7.2008]

Chapter 8

Experiments

This chapter describes the evolutionary experiments with Deathmatch and Capture the Flag game types and also with the evolution of the dodging behaviour. The models being optimized are also described in greater depth.

First section speaks about fragility of results obtained in the experiments, second and third section presents a set of genetic programming experiments optimizing a high level bot's behaviour and the last section shows an application of neural networks on optimization of bot's movement.

8.1 Potential Pitfalls

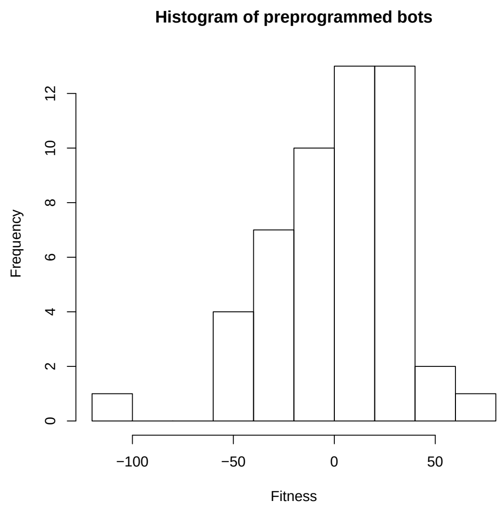
Up to this time the game engines were only single threaded (including the Unreal Engine 2.5 which is a base of UT2004). As processor architecture shifts towards multicore designs the need for proper software architectures arises. To my best knowledge the first multithreaded commercial engine is Unreal Engine 3 [1] released in late 2007. With the dawn of multithreaded programming in the computer games the game engines will no longer be discrete simulations. This implies that outcome of the simulation will be dependent on the external factors like system load and synchronization. Since the Pogamut platform uses multiple threads for logic of each bot the same applies for the experiments presented later in this thesis.

It has also implications towards flow of the time. Game engines are running in "pseudo real-time", thus different game speed may result in slightly different

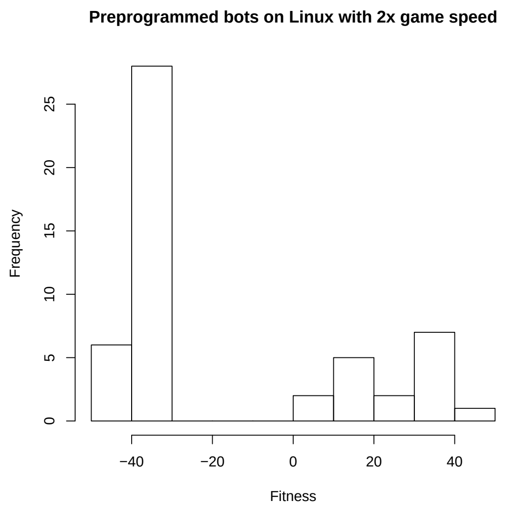
[1]See interview at URL: http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2377&p=3 [12.6.2008]

outcome of the simulation. To measure this influence the same experiment was conducted with two different game speeds. The experiment setup was purposely chosen to highlight this issue. Pogamut based bot was playing a CTF game against a native UT bot. Each bot received one point for killing the enemy and 5 points for bringing the flag back home. The outcomes were significantly different (see Figure 8.1). The main difference perceived during game play in double and single speed mode was that the Pogamut bot was more often killed when he was only a few meters away from the base where he was bringing the flag compared to the single speed mode, where the bot usually managed to bring the flag and died shortly after this.

The result from this observation is that the evaluation of individual's performance for fitness computation must be executed at a single speed. Otherwise the results will not be applicable to single speed mode.



(a) Single speed



(b) Double speed

Figure 8.1: Influence of the game speed on the results of experiment. Fitness of the individual is measured as $myScore - enemysScore$. In the single speed mode the fitness fits a normal distribution but in double speed there is a peek of bad performing individuals with fitness around -40.

8.2 Deathmatch

In the Deathmatch game mode several bots are fighting against each other and their goal is to get as much frags as possible. Bot receives one frag when he kills an opponent.

In this experiment, the evolved bot was playing a Deathmatch game against bot preprogrammed by a human. Details of the experiment setup are in Table 8.1.

Map	dm-TrainingDay
Generations	80
Individuals in each generation	150
Initial generation	random functions with maximal depth of 5
Elite	16
Crossover probability	0.2
Mutation probability	0.2
Selection method	weighted proportional deterministic distribution
Fitness	$dc + 50pk - (5d + \frac{ds}{10})$
Aiming accuracy	0.5
Fitness details	Each bot fought against the Hunter bot (standard example bot included in the Pogamut installation) for 90 seconds of the game time

Table 8.1: Setup of the experiment. In the fitness equation dc is damage caused by the evaluated bot, pk is number of players killed by the bot, d is number of bot's deaths and ds is damage suffered by the bot.

The dm-Trainingday map is the smallest one from the maps officially distributed with the UT. The fitness function favours attacking behaviour over fleeing from the battle because this is the deserved behaviour in Deathmatch type of ame. The reward for damaging the enemy is ten times higher than penalization for being damaged. Aiming accuracy is internal property of the game engine expressing how good will the bots be in using the weapons. Accuracy value 1 means that the bot is absolutely accurate in the shooting no matter what the distance is, accuracy value 0.5 gives the enemy a chance to escape. The accuracy was the same for both the Hunter and an evaluated bot.

In the first 20 generations the evolution was trying random strategies, then between generations 20 and 40 the population began to improve its performance and a local optimum was reached. In the following generations no significant improvement was achieved. Figure 8.2 shows how the fitness changed during the course of the evolution.

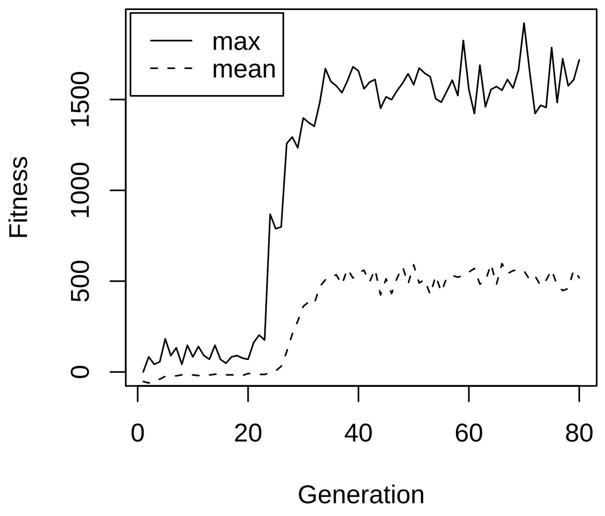


Figure 8.2: Mean and maximum fitness in each generation. Even though the elite was used the nondeterministic fitness computation sometimes causes decrease of the fitness of the fittest individual compared to the previous generation

Figure 8.3 shows the behaviour tree of the best individual from the last generation. The bot was collecting health packs when he has not seen an enemy and he attacked the nearest enemy when he encountered some.

The following table shows outcome of this behaviour tree depending on the external stimulus *SeeAnyEnemy*.

The resulting behaviour is simple but it is sufficient for the deatmatch type of game. Even human players are using the same high level strategy on small maps

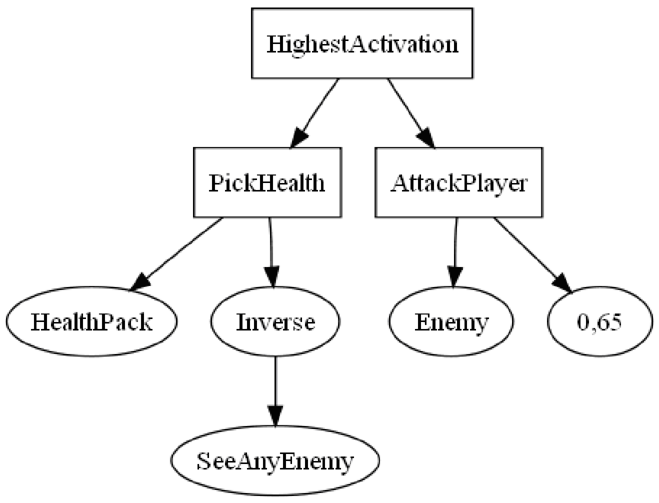


Figure 8.3: Behaviour tree of the best individual from generation 80

Condition	SeeAnyEnemy value	PickHealth activation	Attack activation	Winning behaviour
Bot sees an enemy	1	0	0.65	Attack
Bot does not see an enemy	0	1	0	PickHealth

Table 8.2: How external senses influence bot's decision

like dm-Trainingday.

8.3 Capture The Flag

Compared to Deathmatch the Capture the Flag (CTF) is a more complex game mode with greater space for strategical decisions making. First rules of the CTF game mode will be explained then the experiment will be presented.

In CTF the bots are divided into two teams. In UT2004 these are a Red and a Blue team. Each team has its team base, where is the team's flag. Flag can be in three distinct states: HOME - flag is in team's base, HELD - flag is being held by an enemy, DROPPED - flag is not in team base and it is not held by an enemy. Flags can be carried (flag in HELD state) only by players from the opposing team. When the player touches his team's flag that is DROPPED then it is immediately returned to the team base. All possible transitions between flag states are depicted on a Figure 8.4. When the player brings opponent's flag to his team's base, he scores 15 points. Killing an enemy is rewarded only by one point.

As the game rules suggest there are more ways how a bot can be successful. The bot can specialize himself on stealing opponent's flag or he can just defend his own flag and gain points by killing an enemy or he can combine both these strategies.

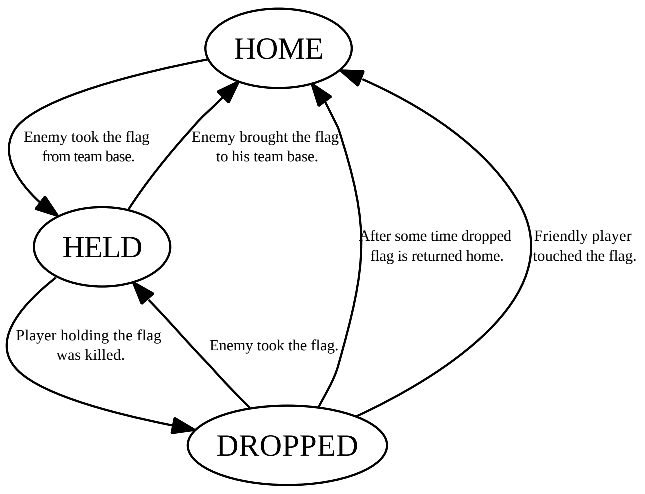

HOME
HELD
DROPPED
Enemy took the flag from team base.
Enemy brought the flag to his team base.
After some time dropped flag is returned home.
Friendly player touched the flag.
Player holding the flag was killed.
Enemy took the flag.

Figure 8.4: All possible states of the flag together with the transitions between them

To provide a GP algorithm with complete set of functions needed for emergence of behaviours capable of playing a CTF game a simple hand-coded bot was implemented. Algorithm 3 shows a decision making system of a bot that can steal enemy's and secure his own flag. The set of functions used in a Deathmatch experiment was augmented by new functions needed to code this algorithm in a behaviour tree notation. The list of added functions is in Appendix A.3.

8.3.1 Coevolution

The previous experiment used a preprogrammed bot as an opponent. This supposes that we already have some bot capable of playing the game or at least that we can programme such bot. If we do not have preprogrammed bot, then coevolution [19] might be useful.

Algorithm 3 Hardcoded bot for CTF

```
1: if see an enemy then
2:     shoot at enemy
3:     return
4: end if
5: if see mine flag and mine flag is dropped then
6:     go to mine flag //touching the flag will return it to the base
7:     return
8: end if
9: if have enemy's flag then
10:    go to the team base // try to score points by bringing the flag home
11:    return
12: end if
13: if see enemy's flag then
14:    go to enemy's flag // bot will pick up the flag when he touches it
15:    return
16: end if
17: go to enemy's base // bot is looking for a flag and enemy's base is the most
    probable place where he can find it
```

In coevolution the bots are evaluated in matches against each other instead of comparison with preprogrammed bot. As soon as one bot finds partial solution to the problem he gains advantage over the rest of the population. This increased evolutionary pressure may cause improvement of the rest of the population in the following generations.

The experiment setup is in Table 8.3. In coevolution the fitness value is only a relative measure of performance compared to the rest of population. To get the absolute measure the best bots from each generation have been evaluated in 40 minutes long match against the native UT bot (graph of fitness is on Figure 8.6) and against hand coded bot controlled by the Algorithm 3 (see Figure 8.7). Both graphs show rapid progress in first 30 generations. At the end the bots were playing balanced games with preprogrammed bot but they were worse than the native UT bot (note that fitness above 0 means that the bot was more successful than his opponent). In game inspection of bot's behaviour revealed that it lost most of the games with native bot because it was lacking the low level behaviour that helped the native bot to avoid missiles while moving in the level. This was not the case when playing against the hand coded bot because it is using the same low level routines for movement, thus it has not any advantage over the evolved bot in this skill.

In short the strategy utilized by the best evolved bot was: wander around, shoot the enemy if he is holding my flag. Whole behaviour tree of the best bot is on Figure 8.8. Most of the time only two nodes of the tree get executed, the first node $attackPlayer(enemy(), spike(...))$ and the last but one node $wanderAround(hasLoadedWeapon())$. The first node is executed when the enemy is holding the flag. The bot is close to scoring 15 point when he is holding a flag, thus it is more dangerous than when it does not have a flag. The second node is executed in most of the other cases since the $hasLoadedWeapon()$ returns 1 most of the time (bot starts with loaded weapon). The behaviour tree does not use the $teamFlagBase()$ function, this means that the bot is not purposely going to enemy's or his team base, however it is able to steal the opponents flag and bring it home. How is this possible? The key is implementation of the $wanderAround()$ function. When called for the first time, this functions plans a path around all sort of items (weapons, healths, etc.) in the map. Bot then follows this path by repeated calls to the $wanderAround()$ function. Since the 1on1-Joust map is fairly small and the items are placed near the team bases (see the map scheme on Figure 8.5) this single behaviour server for both picking items and stealing the flag.

Map	ctf-1on1-Joust (see Figure 8.5)
Generations	200
Individuals in each generation	20
Initial generation	random functions with maximal depth of 6
Elite	2
Crossover probability	0.7
Mutation probability	0.3
Selection method	weighted proportional deterministic distribution
Fitness	$15(fs - fl) + f - d$
Aiming accuracy	0.5
Fitness details	Bots fought against each other for two minutes. The overall fitness was sum of 19 independent matches.

Table 8.3: Setup of the CTF coevolution experiment. In the fitness equation fs is number of flags stolen, fl is number of flags lost (flags stolen by the enemy), f is number of frags and d is number of deaths.

8.4 Dodging

The next section specifies the dodging experiment setup in more detail, later sections present experiments with a neural network evolved by the NEAT [22] algorithm used for controlling the bot's dodging movement. First experiment shows a bot with incomplete information about the missile's position. Bots from the second series of experiments were provided with whole information about the position of the missile.

8.4.1 Experiment Setup

At the start the evaluated bot is standing on one end of the passage and its destination is on the second end. Next to its destination is a sniper bot with rocket launcher that is firing a rocket on the evaluated bot every two seconds. The sniper bot does not move from the initial location and it has infinite ammunition. Evaluated bot does not shoot, his goal is to sneak through the fire and get as

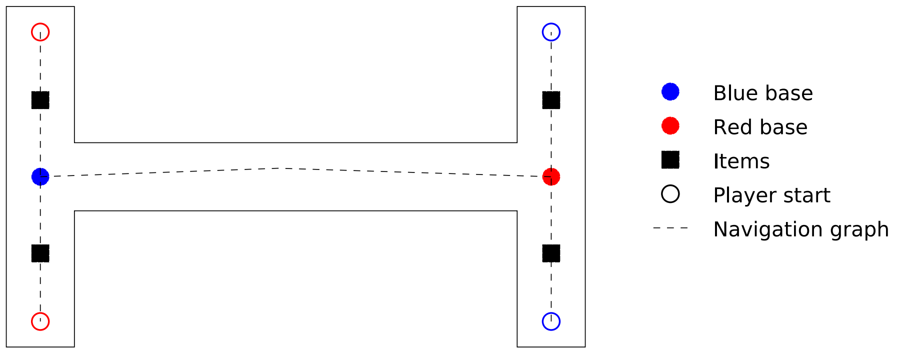


Figure 8.5: Schema of the 1on1-Joust map.

close as possible to its destination. The evaluation run ends when: 1. bot reached 95% of the distance to the target or 2. bot was killed or 3. timeout of 20 seconds was reached. The schema of the level used for dodging experiments is shown in figure 8.9.

8.4.2 First Model

Neural networks inputs used in the first experiment are listed in Table 8.4. To start with the most simplistic model the only information the bot has about the missile was estimated time till impact and relative angle of missile's trajectory at the time when it was fired. This models the situation when the bot can hear the shot, he can estimate how far the missile is but he can not see it.

The fitness function used in this experiment was:

$$f = \frac{3s + (1 - \frac{2arctan(h/5)}{\Pi}) + d}{5}$$

Where $s \in \langle 0, 1 \rangle$ is percentage of distance covered on the way to the destination, $d \in \langle 0, 1 \rangle$ is damage suffered, h is number of hits to the walls, the $\frac{2arctan}{\Pi}$ function is used to map possibly infinite number of hits to interval $\langle 0, 1 \rangle$.

Figure 8.11 shows how the evolution progressed. It can be seen that the average individuals have quite low fitness, most of the bots from all generations were unable to move any further from their starting position. Second interesting trait is that immediately in the first random generation a quite good solution was found, which is true also for the other experiments. Fitness of the best individuals

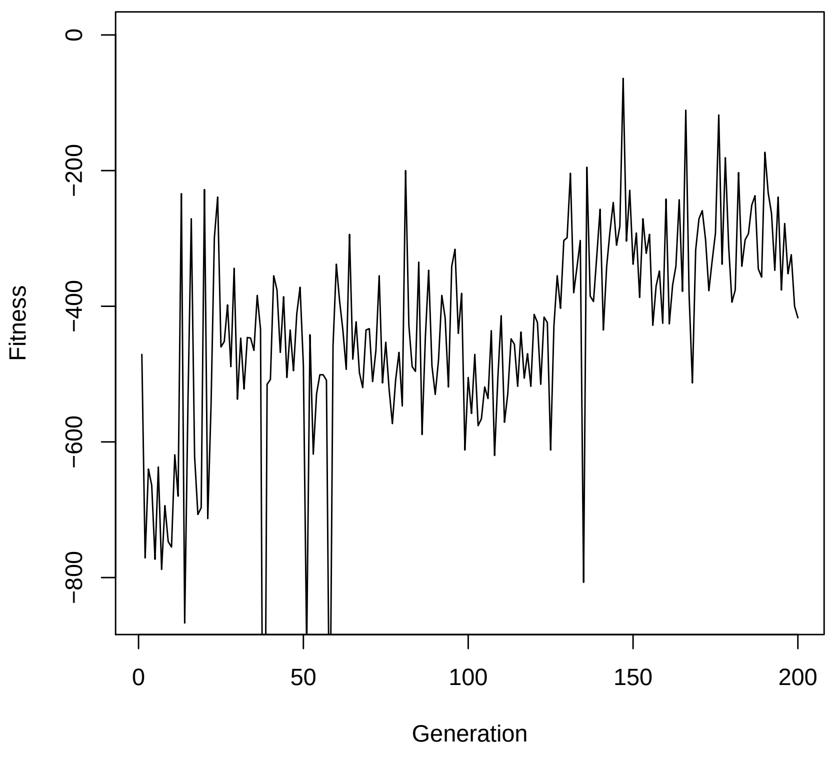


Figure 8.6: Evaluation against the native UT bot

was rather random in the first 80 generations and it began to improve in the last 20 generations. The evolution started with fully connected input and output layers and the best solution, found in the 92nd generation, added one hidden neuron with two connections.

The strategy found by the best bot was following: keep next to the right wall, when the missile approaches dodge left, then return to the right wall. By keeping next to the right wall bot forces the enemy to shoot in that direction but when the missile is coming he avoids the missile by shifting left. This strategy is successful as long as the enemy's shooting and bot's movement are synchronized.

Figure 8.12 shows a little closer the movement decisions of the best bot. The figure was created by connecting the neural network controlling this bot to a

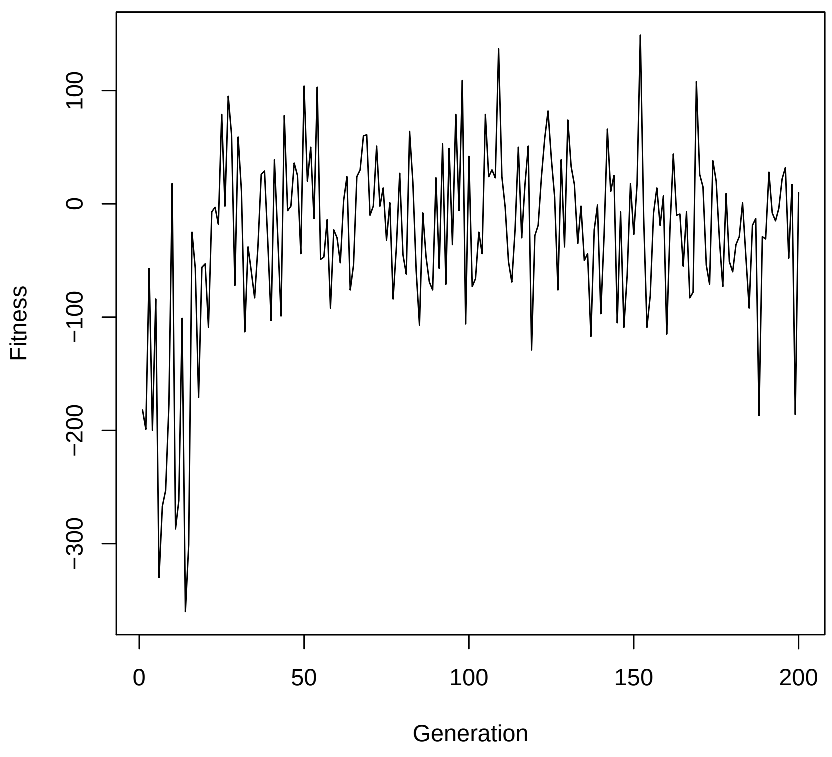


Figure 8.7: Evaluation against hand coded bot controlled by the algorithm 3.

generator producing inputs (raycasting data, time till impact, bot's distance to the shortest path etc.) that the bot would have received from the game engine if he were in the same situation. The interesting features are:

1. All arrows near the right and nearly all arrows near the left wall are directed inwards the passage, this prevent the bot from hitting the walls.
2. Arrows in the rectangular area near the lower right corner are heading slightly left, this it the case when the bot is avoiding incoming missile.
3. There are a few arrows heading forward in the upper right corner, these arrows determine the path the bot is following when a missile is far away.

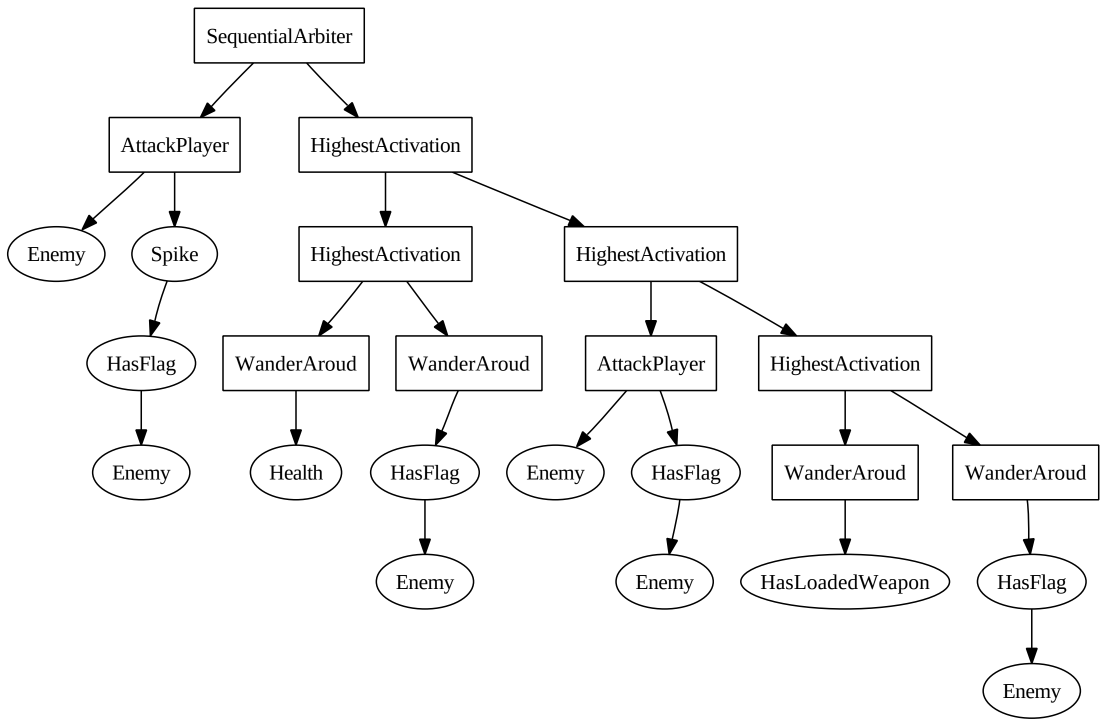


Figure 8.8: Behaviour tree of the most successful bot in play against native UT bot.

4. All other arrows are directed to the right, they lead the bot to the only forward arrows.

The bot's strategy seems reasonable supposing the bot does not have exact information about the position of the missile. Second series of dodging experiments gives the bot ability to perceive the missile's position.

8.4.3 Second Model

In the second series of experiments the "Projectile angle" input was replaced by the shortest distance d between the bot and missile's trajectory (see Figure 8.13). This enables the bot to "see" the incoming missile. The fitness function was the same as in the previous experiment. Graph 8.14 shows progress of the fitness in two differently parameterized runs, see Table 8.6 for parameters of both runs.

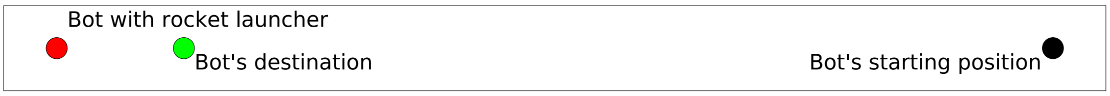


Figure 8.9: Schema of the level used for the dodging experiments. The lengths has modified ratio. Real length of the level was 6000 UT units (UTU), width 256 UTU and the bot's collision volume has diameter of 50 UTU.

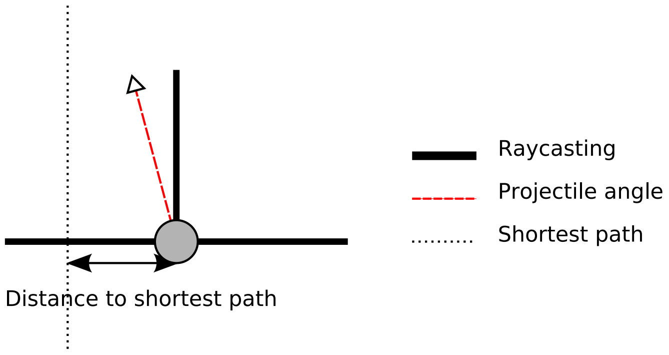


Figure 8.10: Inputs of the dodging bot. Black arrows are rays for detection of obstacles, red arrow is direction of the incoming projectile.

Both runs found solutions that were able to avoid the missile, but their movement was often a bit bumpy. As a result the experiment was rerun with new fitness function designed to smoothen the bot's movement.

New Fitness Function

The fitness function was changed to the form:

$$f = \frac{s + \frac{v}{v_{max}} + (1 - \frac{2arctan(h/5s)}{\Pi}) + d}{4}$$

Where v is average speed of the bot's movement to the target (it is a fraction of the distance the bot covered projected onto the shortest path to the target and the time elapsed) , v_{max} is a maximal speed of the bot, $d \in \langle 0, 1\rangle$ is damage suffered, h is number of hits to the walls, the $\frac{2arctan}{\Pi}$ function is used to map possibly infinite number of hits to interval $\langle 0, 1\rangle$.

Input name	Range	Semantics
Estimated time till impact	$< 0, 1 >$	Time in seconds until the projectile hits the bot.
Projectile angle	$< 0, 1 >$	Angle under which the projectile was coming in the time it was fired, continuous scale from 0 representing left most and 1 representing the right most point of the bot's view frustum.
Distance to the target	$< 0, 1 >$	Distance mapped by a *arctan* to unit interval.
Distance to the shortest path	$< -1, 1 >$	Measure of perpendicular distance to the line between start and destination mapped by a *arctan* to unit interval. Negative when the bot is to the left of this line, positive otherwise.
Trace lines	$< 0, 1 >$	3 rays for detecting walls.

Table 8.4: Inputs for the dodging bot

Two changes compared to the previous function are that the bonus for average movement speed was added and the number of hits is normalized to the covered distance.

First run was started with topology with zero hidden neurons. The fittest individuals usually used two or more hidden neurons (the fittest one used four), thus in the initial number of hidden neurons in the second run was set to 2. Both hidden neurons were fully connected to the output and input layer.

The progression of fitness is shown in the Figure 8.15, evolution parameters are in Table 8.7. In the first run the average and maximal fitness was gradually improving. The second run performed significantly better in an average case, this is probably because of the two initial hidden neurons. Also the maximal performance was higher up to 80th generation, then the progression stopped and the fitness even slightly decreased. The experiment continued for next 100 generations (not shown in the graph) and this decreasing tendency remained. One possible explanation of this decrease could be that the complexity of networks in the second run was raising faster than in the first run. The default mutation

Generations	100
Individuals in each generation	100
Elitism	1 individual from each specie
Add connection mutation rate	0.05
Add neuron mutation rate	0.01
Weight mutation rate	0.4
Std. dev. of weight mutation	1.5
Weight range	$\langle -20, 20 \rangle$
Hidden neurons at start	0
Rocket bot skill	7 - the highest skill level in UT
Fitness details	Fitness is averaged over 10 trials in order to decrease the variance.

Table 8.5: Parameters of the evolution

behaviour in ANJI implementation is that the connection (or neuron) is added in every possible place — between two neurons without connection (or in the middle of an existing connection) — with the specified probability [2]. This means that more complex network will receive more topology mutations which was the case of the second run. Combined with nondeterministic fitness evaluation the smaller networks and on average more fit networks could extinct and the more complex networks gain the whole population. But the complex networks require more time for optimization and the fitness can temporally decrease.

Despite this unfavorable behaviour the network of best individual lead to satisfactory results. Figure 8.16 shows the field of vectors driving bots movement in three configurations differing in the trajectory of the missile. The picture was obtained in the same way as Figure 8.12. The vector fields 8.16(a) and 8.16(c) are not symmetrical as could be supposed in an optimal solution, but we can see that the arrows orient the bot in an opposite direction to the missile thus keeping the bot as far from potential danger as possible. In case 8.16(b) the bot avoids the missile by dodging left but still remains moving forward.

The smoothness of bot's movement was improved and the bot's performance in this task was comparable to a human player.

[2]The original NEAT algorithm has different semantics of mutation, the mutation specifies probability that the single individual is mutated (thus only one connection or neuron is added)

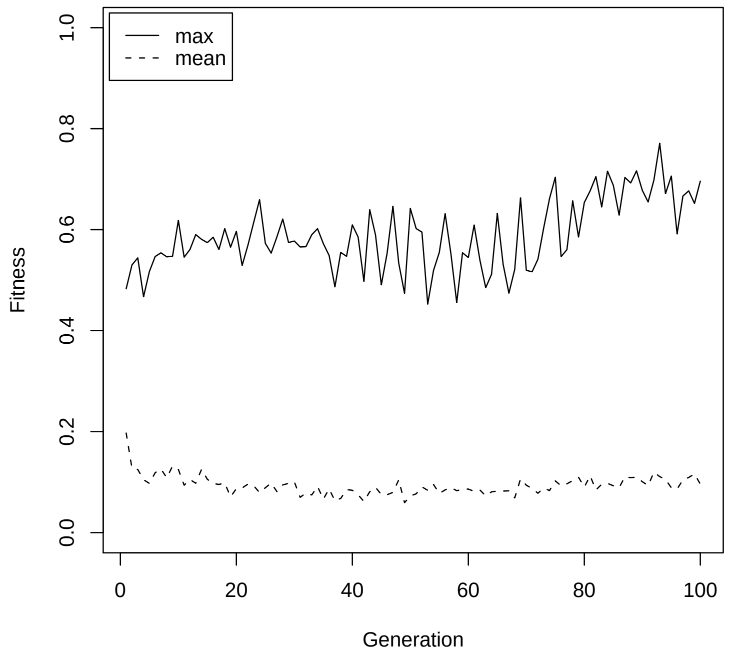


Figure 8.11: Maximal and average fitness in the first dodging experiment.

8.5 Discussion

The GP experiments showed that this technique can be used for acquisition of simple gameplay rules. Both the Deathmatch and CTF experiments succeeded in finding at least suboptimal high-level behaviour. The coevolution experiment in the CTF game mode also showed that the evolution can work even without preprogrammed reference bot used for fitness computation. On the other hand the behaviour trees found were not as complex as algorithm of human competitive hand coded bots. The bots used the sensory inputs only in limited way, for example they did not use the internal health level for making decision whether to attack or not (even though the presented set of functions makes this possible, see Section 6.1.1).

The dodging experiments has shown that the NEAT algorithm can be successfully used for optimization of bot's movement. However it has also revealed

Run	1st	2nd
Generations	100	
Individuals in each generation	100	
Elitism	1 individual from each specie	
Add connection mutation rate	0.01	0.005
Add neuron mutation rate	0.005	0.0025
Weight mutation rate	0.8	0.6
Std. dev. of weight mutation	1.5	
Weight range	$\langle -20, 20 \rangle$	
Hidden neurons at start	0	
Rocket bot skill	7 - the highest skill level in UT	
Fitness details	Fitness is average of 10 trials	

Table 8.6: Parameters of the second series of dodging experiments

problems in the complexification phase of the algorithm that were probably caused by nondeterministic fitness evaluation. We think that this approach can be easily used for creation of different parameterizations of bot's movement. The game designer can specify bot's personality in terms of the fitness function (if the weight of damage component of the fitness function is raised, then the bot will be more cautious, on the other hand when the speed component is accented then the bot will be more aggressive).

The experiments were run over night on cluster of 25 computers (Pentium Dual Core 2.4GHz, 1GB RAM) that are used for education during the day. 100 generation of dodging evolution took on average 8 hours, the CTF coevolution experiment was run for approximately 64 hours. On each computer there were from 6 to 8 simultaneous games. With this setup the average system load was below 40% but the peeks sometimes reached 100%. If there were more simultaneous games on one computer, the peeks would be more often and the results of the simulation could be affected.

Due to high time demands of the experiments the runs were not repeated with different random seeds, which would be necessary in production use.

Run	1st	2nd
Generations	120	
Individuals in each generation	100	
Elitism	1 individual from each specie	
Add connection mutation rate	0.005	
Add neuron mutation rate	0.0025	
Weight mutation rate	0.4	0.6
Std. dev. of weight mutation	1.5	1
Weight range	$\langle -20, 20 \rangle$	$\langle -15, 15 \rangle$
Hidden neurons at start	0	2
Rocket bot skill	7 - the highest skill level in UT	
Fitness details	Fitness is average of 10 trials	

Table 8.7: Parameters of the third series of dodging experiments

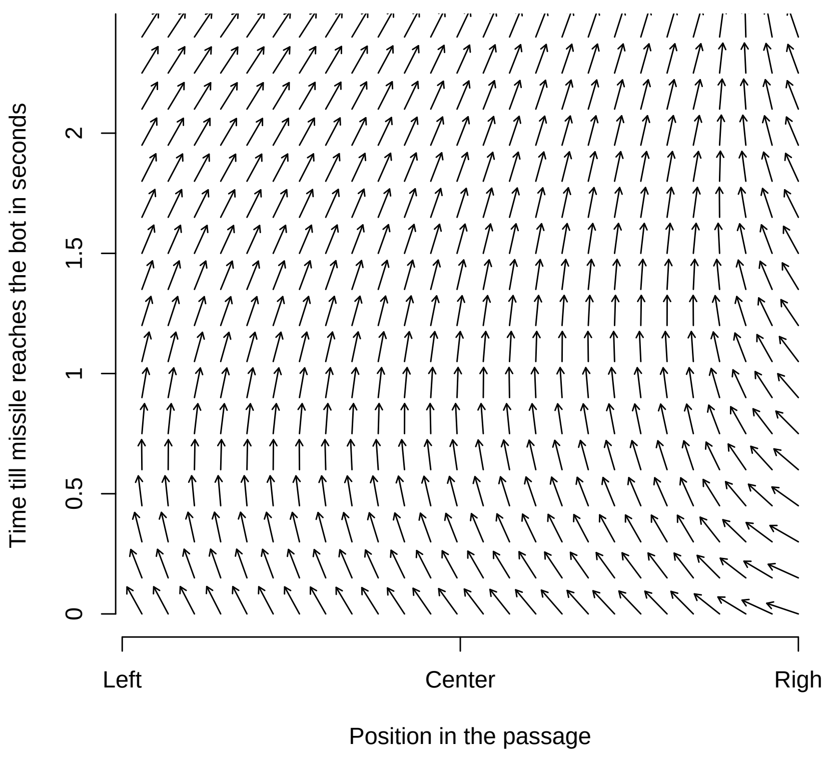


Figure 8.12: Vector field of the best individual from the second run. On the Y axis is time till the missile reaches bot's level. The X axis shows bot's position in the passage. Each arrow represents movement direction computed by the neural network at that particular time.

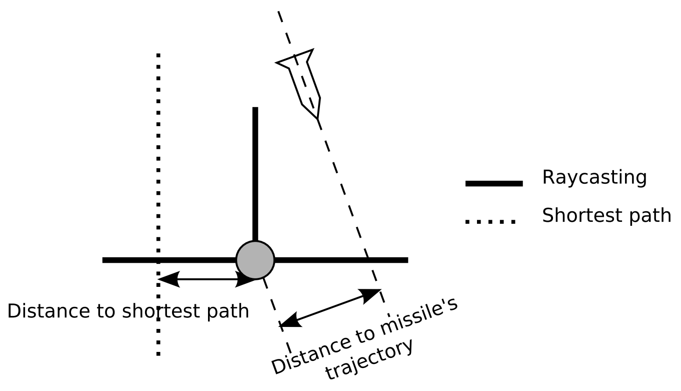


Figure 8.13: Nearest distance to the missile's trajectory.

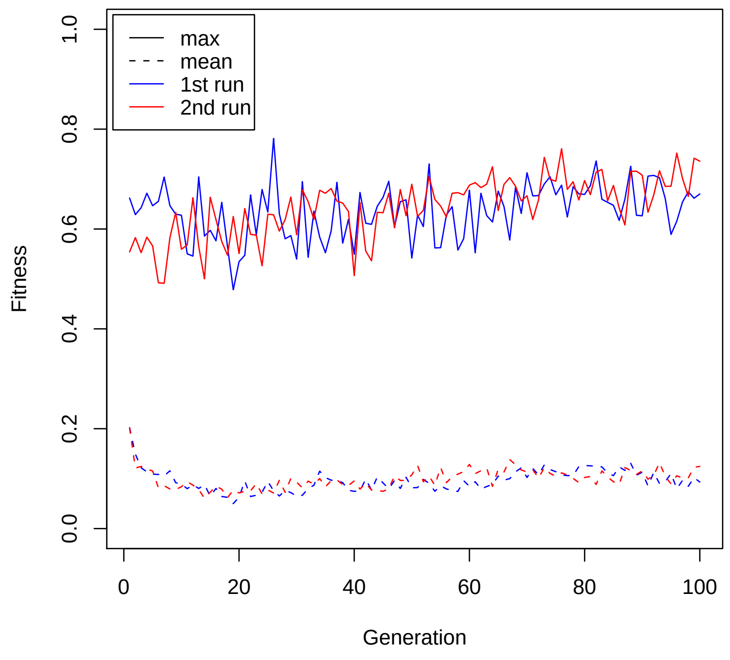


Figure 8.14: Maximal and average fitness in the second dodging experiment.

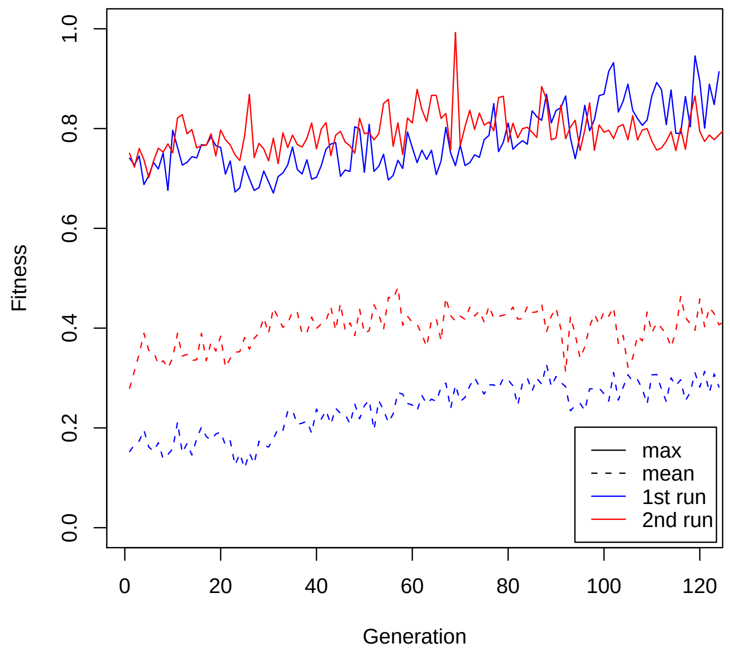
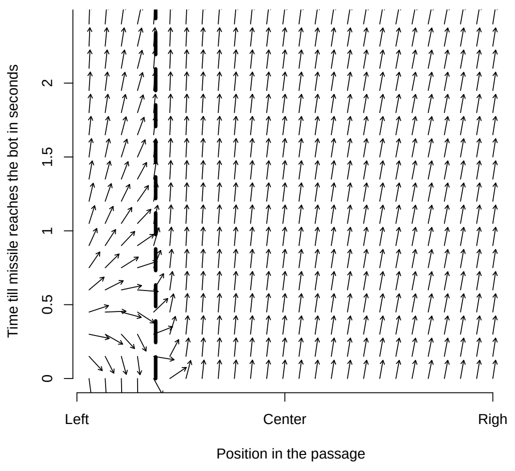
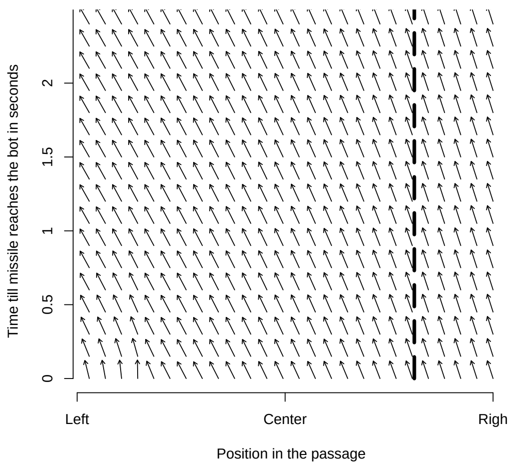


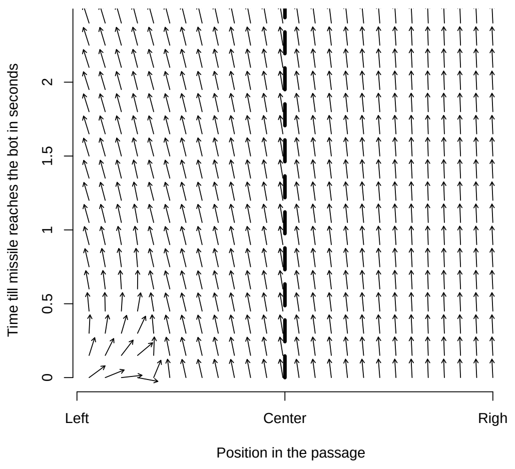
Figure 8.15: Maximal and average fitness in the third dodging experiment.



(a) Missile on the left



(b) Missile on the right



(c) Missile in the center

Figure 8.16: Force field of the best bot from the first run. The dashed line represents trajectory of the missile.

Chapter 9

Future Work

This chapter discusses possible future directions of research in the field of genetic optimization of bot's behaviour.

There are still some limitations that has to be overcome before the genetic algorithms can become aid in designing the whole ASM for bots in FPS games:

- Testing in bigger levels — current experiments were performed in relatively simple levels that are suitable for two or three players. The bigger levels provide significantly more possible strategies of winning thus the time needed for exploring this search space also increases.

- Better notion of the map — the functional language could be augmented by set of constants denoting unique instances of game objects. In current setting the bot can pick the nearest health pack by calling function $pickHealth(nearestHealth())$ but it is impossible to pick the second nearest health pack that is on a safer place, constants like $health_14()$ would make this possible.

- Problem aware genetic operators — the genetic operators used in GP experiments were general purpose implementation without any domain specific information. E.g. the mutation can replace subtrees by new functionally similar subtrees (i.e. pick ammo instead of pick weapon), or the crossover operator can exchange the subtrees computing activation of behaviour only between the same primary behaviours.

- Enhancements to the GP algorithm — speciation technique used in the NEAT algorithm can be implemented in the GP algorithm.

As for the dodging model the next natural step would be simultaneous evolution of the high-level behaviour together with the low-level subbehaviours (like dodging). The mutual influence of high and low-level behaviour layers could bring new surprising solutions to the whole problem of ASM.

Chapter 10

Conclusion

This work has proposed, implemented and tested two possible models of genetic optimization of bot's behaviour thus the main objectives of this thesis were fulfilled. The first model uses custom functional language and genetic programming to evolve the high-level ASM from scratch. The second model uses neural networks for optimization of dodging movement (avoiding the missiles while still moving forward).

In a future the first method could be used for automatic creation of bot's logic. One day the game designers could test their levels by evolving population of bots and then observing their gameplay strategies. This work has made one step towards this high level goal, however there has to be made many more steps before this technique becomes designer's aid instead of researcher's toy.

The second model presented in this work is more matured and it is in production ready state. Neural networks has proved to be successful in the dodging task and a similar approach could be probably applied to other low-level, hard to parameterize tasks the bot has to deal with.

List of Figures

2.1 Unreal Tournament 2004 screenshot showing a bot firing from a rocket launcher . . . 14

4.1 One point crossover in linear coding . . . 20

4.2 Tree for expression $(x + 2) * 3$ . . . 22

4.3 Example of crossover of expressions $3 * (x + 2)$ and $x + 1$ . . . 23

4.4 Example of mutation of expressions $3 * (x + 2)$ into $(x * x) * (x + 2)$ 23

5.1 The Act-Sense loop . . . 27

5.2 Conceptual layers of bot's behaviour . . . 28

5.3 Example of FSM controlling a guard bot. States Guard, Heal and Attack have associated scripts. For example the script for the Heal state could find the nearest health pack and pick it. . . . 29

5.4 Example of behaviour tree controlling a guard bot. The same algorithm coded in FSM is shown in figure 5.3. Octagonal nodes are internal nodes — arbiters. . . . 30

6.1 Example behaviour tree constructed from the provided set of functions . . . 34

8.1 Influence of the game speed on the results of experiment. Fitness of the individual is measured as $myScore - enemysScore$. In the single speed mode the fitness fits a normal distribution but in double speed there is a peek of bad performing individuals with fitness around -40. . . . 40

8.2 Mean and maximum fitness in each generation. Even though the elite was used the nondeterministic fitness computation sometimes causes decrease of the fitness of the fittest individual compared to the previous generation 42
8.3 Behaviour tree of the best individual from generation 80 43
8.4 All possible states of the flag together with the transitions between them 44
8.5 Schema of the 1on1-Joust map. 48
8.6 Evaluation against the native UT bot 49
8.7 Evaluation against hand coded bot controlled by the algorithm 3. 50
8.8 Behaviour tree of the most successful bot in play against native UT bot. 51
8.9 Schema of the level used for the dodging experiments. The lengths has modified ratio. Real length of the level was 6000 UT units (UTU), width 256 UTU and the bot's collision volume has diameter of 50 UTU. 52
8.10 Inputs of the dodging bot. Black arrows are rays for detection of obstacles, red arrow is direction of the incoming projectile. 52
8.11 Maximal and average fitness in the first dodging experiment. . . . 55
8.12 Vector field of the best individual from the second run. On the Y axis is time till the missile reaches bot's level. The X axis shows bot's position in the passage. Each arrow represents movement direction computed by the neural network at that particular time. 58
8.13 Nearest distance to the missile's trajectory. 59
8.14 Maximal and average fitness in the second dodging experiment. . 59
8.15 Maximal and average fitness in the third dodging experiment. . . 60
8.16 Force field of the best bot from the first run. The dashed line represents trajectory of the missile. 61

List of Tables

8.1 Setup of the experiment. In the fitness equation dc is damage caused by the evaluated bot, pk is number of players killed by the bot, d is number of bot's deaths and ds is damage suffered by the bot. 41

8.2 How external senses influence bot's decision 43

8.3 Setup of the CTF coevolution experiment. In the fitness equation fs is number of flags stolen, fl is number of flags lost (flags stolen by the enemy), f is number of frags and d is number of deaths. . . 47

8.4 Inputs for the dodging bot 53

8.5 Parameters of the evolution 54

8.6 Parameters of the second series of dodging experiments 56

8.7 Parameters of the third series of dodging experiments 57

Bibliography

[1] R. Adobbati, A. N. Marshall, A. Scholer, and S. Tejada. Gamebots: A 3d virtual world test-bed for multi-agent research. In *Proceedings of the 2nd Int. Workshop on Infrastructure for Agents, MAS, and Scalable MAS. Montreal, Canada*, 2001.

[2] A. Barr and E. A. Feigenbaum, editors. *The Handbook of AI, Vol. 1.* Heuris-Tech Press, Stanford, California, 1981.

[3] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, October 1999.

[4] M. Brameier. *On linear genetic programming.* PhD thesis, Universität Dortmund, 2004.

[5] 0. Burkert. Unreal tournament twins. Bachelor's thesis, Charles University in Prague, 2006. (in Czech).

[6] E. Cantú-Paz. A survey of parallel genetic algorithms. URL: tracer.lcc.uma.es/tws/cEA/documents/cant98.pdf [2.7.2008].

[7] A. J. Champandard. *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors.* New Riders, Indianapolis, IN, USA, 2003.

[8] N. Chapman. Neuralbot, 1999. URL: http://homepages.paradise.net.nz/nickamy/neuralbot/nb_about.htm.

[9] N. Cole, S.J. Louis, and C. Miles. Using a genetic algorithm to tune first-person shooter bots. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 139–145, 2004. URL: www.cse.unr.edu/ sushil/pubs/newpapers/2004/cec/cole/
paper.ps.

[10] Ch. Darwin. *On the Origin of Species by Means of Natural Selection.* John Murray, UK, 1859.

[11] J. H. Holland. *Adaptation in natural and artificial systems.* MIT Press, Cambridge, MA, USA, 1992.

[12] J. Holm and J. D. Nielsen. Genetic programming - applied to a real time game domain. Master's thesis, Aalborg University, 2002.

[13] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition).* Addison Wesley, November 2000.

[14] D. Isla. Handling complexity in the Halo 2 AI, 2005. URL: http://www.gamasutra.com/gdc2005/features/20050311/isla_pfv.htm [2.7.2008].

[15] R. Kadlec, J. Gemrot, O. Burkert, M. Bída, J. Havlíček, and C. Brom. Pogamut 2 - A platform for fast development of virtual agents' behaviour. In *Proceedings of CGAMES 07, La Rochelle, France*, 2007.

[16] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[17] K. Meffert. JGAP - Java Genetic Algorithms and Genetic Programming Package.

[18] Ch. Miles and J. L. Sushil. Co-evolving real-time strategy game playing influence map trees with genetic algorithms. In *Proceedings of the Congress on Evolutionary Computation*, pages 171–185. IEEE, Vancouver, Canada, 2006.

[19] J. Paredis. Coevolutionary computation. volume 2, pages 355–375, Cambridge, MA, USA, 1995. MIT Press.

[20] M. Parker and G. B. Parker. The evolution of multi-layer neural networks for the control of xpilot agents. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.

[21] S. Priesterjahn, O. Kramer, A. Weimer, and A. Goebels. Evolution of human-competitive agents in modern computer games. In *Proceedings of the IEEE World Congress on Computational Intelligence (WCCI'06) Vancouver, Canada*, pages 777–784, 2006.

[22] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. volume 10, pages 99–127, Cambridge, MA, USA, 2002. MIT Press.

[23] J. Togelius, S. M. Lucas, and R. De Nardi. Computational intelligence in racing games. In *Advanced Intelligent Paradigms in Computer Games*, 2007.

[24] T. Tyrrell. *Computational Mechanisms for Action Selection.* PhD thesis, Centre for Cognitive Science, University of Edinburgh, 1993.

[25] P. Tzour. Building a near-optimal navigation mesh. In *AI Game Programming Wisdom*, pages 171–185. Charles River Media, 2002.

[26] J. P. van Waveren. The Quake III Arena Bot. Master's thesis, Delft University of Technology, 2001.

[27] J. Westra. Evolutionary neural networks applied in first person shooters. Master's thesis, University Utrecht, 2007.

[28] N. R. Wooldridge, M. Jennings. Intelligent agents - theories, architectures and languages. In *Volume 890 of Lecture Notes in Artificial Intelligence. Springer-Verlag*, 1995.

[29] B. G. Woolley and G. L. Peterson. Genetic evolution of hierarchical behavior structures. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation. London, England*, pages 1731–1738, 2007.

[30] G. Xiao, F. Southey, R. C. Holte, and D. F. Wilkinson. Software testing by active learning for commercial games. In *AAAI*, pages 898–903, 2005.

Appendix A

List of functions

A.1 Behaviour Functions

Primary behaviour functions. Primary behaviour functions represent atomic behaviours. Atomic behaviours are the source of actions that can be performed in the environment.

- ***stay****(double)* — No action is performed, bot stays still.
- ***wanderAround****(double)* — Randomly walk around the items in the map.
- ***goTo****(Location, double)* — Go to the given place.
- ***pickHealth****(Health, double),* ***pickAmmo****(Ammo, double),* ***pickWeapon****(Weapon, double),* ***pickArmor****(Armor, double)* — Pick the given type of item.
- ***runAwayFromPlayer****(Player, double)* — Bot runs in the opposite direction to the specified player.
- ***turnLeft****(double)* — Turn left.
- ***attackPlayer****(Player, double)* — Shoots at given player. Stop shooting if is already shooting and the player is null. This behaviour is also responsible for weapon selection. The exact algorithm is:
  - Rearm if better weapon is available.
  - Rearm if out of ammo.

  – Shoot at player.

All primary behaviour functions have at least one *double* parameter, this parameter represents suitability of the action computed by the behaviour function. Some functions do not pass the suitability value directly to the *BehResult* tuple but they can change the suitability depending on the context. For example, if the bot is out of ammo, then the *attackPlayer* function always returns zero suitability Additional parameters represent the entities the behaviours are operating with.

Secondary behaviour functions. Raise of composite behaviours is possible due to secondary behaviour functions.

- ***highestActivation**(BehResult, BehResult)* — Compares suitability of two behaviour results, returns the one with higher suitability.
- ***suitabilityWeighter**(BehResult, double)* — Multiplies suitability of given behaviour result by the double value.

Since the suitability of the behaviour may change in time it is possible that the *highestActivation* function switches from the first behaviour to the second and vice versa.

A.2 Sensory Functions - general

Sensory functions are used as parameters for the behaviour functions. They correspond to internal and external senses, internal senses are originating from the agent's body, external from the environment.

Internal senses. Internal senses provide information about bot's inner state.

- ***health**()* — health normalized to $\langle 0, 1\rangle$
- ***ammo**()* — ammo of active weapon normalized to $\langle 0, 1\rangle$
- ***ammoForWeapon**(WeaponType)* — amount of ammo the bot has for given weapon type normalized to $\langle 0, 1\rangle$
- ***hasWeapon**(WeaponType)* — 1 if bot has weapon of given type, 0 otherwise.

- ***pain()*** — change of the health since last iteration / 30

External senses. External senses provide information from the environment.

- ***time()*** — game time in seconds
- ***nearestAmmo(), nearestArmor(), nearestHealth(), nearestWeapon()*** — nearest item of corresponding type.
- ***nearestEnemy()*** — nearest player from the opposing team.
- ***see****(Viewable)* — 1 if bot sees given viewable object (viewable objects are players, places etc.), 0 otherwise.
- ***seeAnyEnemy()*** — shortcut for **see**(**nearestEnemy**()).
- ***distanceTo****(Location)* — computes distance to given location, the distance is mapped to the $\langle 0, 1\rangle$ interval by the arctan function. Returns -1 if no location is provided.

A.3 Functions for CTF

These sensory functions were added for the CTF experiment (see Section 8.3).

- ***teamType()*** — function that is randomized by mutation and then returns constant value MINE or ENEMY.
- ***flagState()*** — function that is randomized by mutation and then returns constantly one of these possible values: HELD, DROPPED, HOME.
- ***flag****(TeamType)* — returns object representing flag of given team.
- ***doIhaveAflag()*** — returns 1 if the bot is holding enemy's flag, 0 otherwise.
- ***hasFlag****(Player)* — returns 1 if the given player is holding a flag, 0 otherwise.
- ***teamFlagBase****(TeamType)* — location of the base, can be used as an input to goTo function.

One new arbiter function was also added:

- ***sequentialArbiter****(BehResult, BehResult)* — returns the first behaviour result if its activation is higher than 0.1 or when the activation of second behaviour is lower than 0.1, otherwise return the second behaviour result. It is useful for creation of sequential behaviours.

A.4 Mathematical Functions

Most of the mathematical functions are self explanatory. *max*, *min* and *1 - x* functions can be viewed in some cases as logic functions *and*, *or* and *not* since boolean sensors are coded by numbers $\{0, 1\}$. Sine function together with the *time()* sense function can be source of periodical activation.

- ***spike****(double)* — if the input is greater than 0.5, then the spike function returns value 1. When the input value decreases under the 0.5, then the return value also decreases linearly with time, after 5 seconds the return value decreases to 0. It can serve as short term memory, e.g. spike(pain()) function can inform the bot about the damage suffered in a few seconds after if actually happened.

Appendix B

CD-ROM

The enclosed CD contains source codes of the Pogamut Platform, Pogamut GRID, both models presented in this thesis and PDF version of this text. Structure of the CD is described in the readme.txt file in the root directory.