Master Thesis

By

Jens Dalgaard Nielsen and Jørn Holm

June 2002

close combat strategies for agents in the game of Unreal Tournament. The work presented in Holm and Nielsen (2002) is used as the framework for this thesis.
We propose four different extensions to the basic Genetic Programming algorithm, in order to gain control over the growth in average size of the solutions, and to improve the genetic operators, with respect to maintenance of diversity.
In conclusion we find that a guidance of the genetic operators to be applied within effective code, shows good performance. A direct pressure that rewards unique solutions and punishes common solutions shows good performance and an insignificant degree of bloat.

Supervisor:
Dr. Jose M. Peña
jmp@cs.auc.dk

Number Printed: 6
Number of Pages: 119
Finished: 6*th* June 2002

I denne tese bruger vi Genetisk Programmering til at udvikle nærkamps strategier for agenter i spillet Unreal Tournament. Stoffet præsenteret i Holm and Nielsen (2002) bliver brugt som basis for denne tese. Vi foreslår fire forskellige udvidelser af den normale Genetiske Programmerings algoritme, med formålet at kontrollere den gennemsnitlige vækst i størrelse af løsningerne og forbedre de genetiske operatorer med henblik på diversitets opretholdelse.
Vi konkluderer at de genetiske operatorer med fordel kan anvendes på den effektive del af koden. Et direkte pres der belønner unikke løsninger og straffer almindelige løsninger viser også gode resultater og desuden en ubetydelig grad af "bloat".

Vejleder:
Dr. Jose M. Peña
jmp@cs.auc.dk

Antal Eksemplarer: 6
Antal Sider: 119
Afsluttet: 6. Juni 2002

Acknowledgments

We acknowledge Unreal Tournament as a registered trademark of Epic Games, Inc.

We would like to thank Rimantas Benetis for providing useful modifications to the Gamebots module.

We would like to thank our supervisor Dr. Jose M. Peña for being our mentor and mahatma.

Jørn Holm

Jens Dalgaard Nielsen

2.2 The Gamebots Domain 14
2.2.1 The Features 14
2.2.2 The Gamebots System 14
2.2.3 The Gamebots Interaction Protocol 15

3 Evolutionary Algorithms **17**
3.1 Introduction 17
3.1.1 Search Strategies 17
3.2 Genetic Algorithms 19
3.2.1 The Basic Loop of Evolution 19
3.2.2 Ensuring Convergence 22
3.2.3 Maintaining diversity 23
3.2.4 The Fitness Function 28
3.3 Genetic Programming 31
3.3.1 Genetic Operators and Parse Trees 31
3.3.2 The Basic Building Blocks $\mathcal{F}$ and $\mathcal{T}$ 33

4 Summary of previous work **35**
4.1 Previous Goals 35
4.2 Language Design 35
4.2.1 The Most Basic Skills 36
4.2.2 The More Offensive Skills 37

5.2.1 Popular Bricks And Building Blocks 54
5.2.2 The Flower of The Tree 54
5.2.3 The Withered Leaves 55
5.2.4 Causes of growth of non-executed code 62
5.2.5 Effects of code growth 63
5.3 Project Goals 64

II Our Approach **67**

6 System Modifications **69**

7 Size Ranking **71**
7.1 The Bloat Phenomenon 71
7.2 Previous Efforts 72
7.2.1 Primitive Parsimony Pressure 72
7.2.2 Adaptive Parsimony Pressure 73
7.2.3 Explicitly Defining Introns 73
7.3 Our Approach 74

8 Diversity Ranking **75**
8.1 Our Approach 76

11.1 Parameters of the Evolution 95
11.2 Results 95
11.2.1 The BASIC run 97
11.2.2 Size Ranking 104
11.2.3 Diversity Ranking 104
11.2.4 Enhanced Context Free Grammar 106
11.2.5 Executed Path Guided Operators 109

12 Conclusion and Future Work **113**
12.1 Summary of Results 113
12.2 Failures 114
12.3 Successes 114
12.4 Future Work 114
12.4.1 Agent Specific Extensions 114
12.4.2 System specific Extensions 116

Bibliography **119**

A Node Frequencies **123**

B New Node Frequencies **131**

C Language Reference **133**

the mesh topology. The mesh loops, giving a toroidal structure. 26

3.6 A snapshot of a population of five subpopulations (black, red, yellow, green and blue) evolving. This is not empirical data, but this is what we expect of the island principle. A-E are the individuals that recently migrated. 27

3.7 Organization of subpopulations in the MDPGA. The neighborhood of sub-population a and c are shown as dotted circles. Subpopulation b is a common neighbor to both a and c. . . . 27

3.8 The all-against-all competitive fitness approach, and cup tournament fitness approach, (a) and (b) respectively. The approaches were previously presented by Axelrod (1987) and Angeline and Pollack (1994). 29

3.9 A co-evolutionary environment constituted by two parallel populations, as used by Hillis (1992). 30

3.10 Crossover and mutation applied to parse trees. 32

4.1 The enemy movement as perceived by the golden (upper) bot. Distinctions between leftward/rightward and forward/backward depicted in (a) and (b) respectively. Arrows of the same color within the same subfigure corresponds to directions of movement that yield the same result. 38

4.2 The use of relative destination points when strafing. The coordinatesystem relative to the red bot is rotated as the bot moves, and the point of destination (x',y') then changes dynamically (with respect to global coordinates), as depicted in the change of (a) to (b). 39

run has been pitted against 7 different enemies. The executed nodes is colored, figure (a) shows the result of 6 out of the 7 matches, figure (b) shows the result of the last test. 60

5.6 The fittest individual from the 6th island, 10th generation, 3rd run has been pitted against 7 different enemies. The executed nodes is colored, figure (a) shows the result from 6 out of the 7 matches, figure (b) shows the results of the last test. 60

5.7 The evolution of average size for the 3 test runs. 60

5.8 A global intron in a parse tree is typically caused by redundant sensor-checks. 61

5.9 A local intron in a parse tree is caused by a specific configuration of the test case, and hence is dependent on the current test case. 62

7.1 The 6 different fitness classes, produced by our tournament based fitness function, when applied to a population of 32 individuals, with fitness decreasing from left to right. 74

8.1 Two trees are compared, executed nodes are colored. A match of size 4 has been encircled. 76

8.2 Two trees are compared, executed nodes are colored. A match of size 5 has been encircled. 77

9.1 In (a) a randomly generated tree is depicted. Notice the semantical equivalence with the tree depicted in (b) which is recognized by our CFG. 82

the beginning of an accumulation of good building-blocks in a still growing tree. Two later generation individuals selected for crossover (d), and the offspring produced by choosing the black dots as cut-points (e). 100

11.4 Performance(a) and size(b) graphs for the SR run. 103

11.5 Performance(a) and size(b) graphs for the DR run. 105

11.6 Performance(a) and size(b) graphs for the E-CFG run. 107

11.7 Frequency of node usage during the E-CFG run. 108

11.8 Performance(a) and size(b) graphs for the EPGO run. 110

A.1 The frequency of node usage on island 1 (fig. a-b), 2 (fig. c-d) and 3 (fig. e-f) of the 1st run. 123

A.2 The frequency of node usage on island 4 (fig. a-b), 5 (fig. c-d) and 6 (fig. e-f) of the 1st run. 124

A.3 The frequency of node usage on island 7 (fig. a-b) of the 1st run. 125

A.4 The frequency of node usage on island 1 (fig. a-b) and 2 (fig. c-d) of the 2nd run. 125

A.5 The frequency of node usage on island 3 (fig. a-b), 4 (fig. c-d) and 5 (fig. e-f) of the 2nd run. 126

A.6 The frequency of node usage on island 6 (fig. a-b) and 7 (fig. c-d) of the 2nd run. 127

A.7 The frequency of node usage on island 1 (fig. a-b) of the 3rd run. 127

8.2 Algorithm for matching subtrees. 79
9.1 The alphabet of the E-CFG. The functions are assigned uppercase letters in the leftmost box and the terminals are assigned lowercase letters in the rightmost box. 83
9.2 The E-CFG of the constrained syntactic rules for custom tree generation, crossover and mutation. 86
11.1 The different parameters used in all runs. 96
11.2 The winner from generation 55 of the BASIC run. 101
11.3 Two common building blocks extracted from the winner of generation 55 of the E-CFG run. 101
11.4 The best individual from the 9th (the leftmost) and 10th (the rightmost) generation 102
11.5 Two common building blocks from the BASIC run. 109
C.1 Terminal reference 134
C.2 Function reference the # column contains the number of arguments required by the function. 135

tivate the further analysis. Chapter 2 introduces the specific problem domain of the game Unreal Tournament, and presents the extension provided by the Gamebots module. Chapter 3 presents the basic theory of Genetic Algorithms and Genetic Programming necessary for understanding our system. Chapter 4 presents a summary of our previous work reported in Holm and Nielsen (2002). Finally chapter 5 concludes this part by narrowing the problem space and defining the goals of this project.

of Genetic Programming, mostly credited to Koza (1992), has secured its foothold within the community of Artificial Intelligence.

From this brief historical survey we move on to motivate the general focus of this thesis.

1.1 Motivation

Some everlasting obstacles to evolutionary optimization exists, and in this thesis we focus on some of these obstacles, or more precisely on ways to bypass these obstacles. Before going into technical details about the contents of this thesis, we will include a reflection over *The Investment Principle* presented by Minsky (1988), as it very nicely presents the essence of what is the focus of this thesis. Also, we find it reasonable to mention the principle of Occam's Razor, in order to put work presented in this thesis in a broader perspective.

1.1.1 The Investment Principle

The investment principle as stated by Minsky (1988) is:

> *"Our oldest ideas have unfair advantages over those that come later. The earlier we learn a skill, the more methods we can acquire for using it. Each new idea must then compete against the larger mass of skills the old ideas have accumulated."*

Minsky (1988) argues that natural evolution is a good example of a process being enslaved by the investment principle. That is, good skills that were developed in the early stages of the evolution, is hard to change without

In more modern English, this is often translated to *"Pluralities should not be posited without necessity"*. This principle is one of the cornerstones of many scientific disciplines, especially when developing models of natural phenomenon and physical processes. Models of high complexity is typically less general than models of low complexity. The more complex the model, the more details it has captured, and hence it becomes fragile to otherwise insignificant changes. We can connect this to the investment principle described above. As evolution continually constructs solutions to fit the current environment by patching up and recombining old ideas, the evolved solutions become more and more complex, and hence more and more specialized. This phenomenon is evident for most spices in nature. That is, most spices are highly specialized to benefit from the environment in which they exist, or more specific, the environment in which their ancestors existed.

We have now argued that evolution does not inherently obey the principle of Occam's Razor.

1.2 Problem Space

The problem space of this thesis, is defined to be within the application of the Genetic Programming paradigm to a real time computer game. The experiments and results reported in this thesis builds upon work previously reported by Holm and Nielsen (2002).

In the previous sections, we argued that one inherent property of evolution is the bias toward favoring old ideas over new ones. We should realize that this property can be damaging to the evolution, if the early ideas are not optimal for solving the current problem. If these sub-optimal ideas gain strong

computer screen displays the world through the avatar's eyes as a human visually perceive a 3D world (1st person perspective). UT is a fast, complex and dynamic game domain. It offers a broad set of game types, eg. Death Match, Domination and Capture the Flag, which is further described in section 2.1.1. In addition, multiple different world maps, varied both in size and semblance, are available as indicated in J. Gerstmann (1999). Each of these game types requires a number of opponents, which can be AI or human controlled opponents. The AI controlled opponents in UT, will be described in section 2.1.2.

2.1.1 Game Types

There are several possible game types available in UT. Most of these, are mutations of the three (probably) most popular game types which are described below:

Death Match: Kill as many competitors as possible and try to avoid being killed by them. The player who reaches the frag [1] limit first (or has the most frags when the time limit is reached) is the winner.

Domination: Two teams or more, fight for possession of several control points scattered throughout the map. To take a control point, a player simply touches it, and that control point is now owned by that player's team. When a team owns a control point, their score increases steadily until the other team touches the control point.

[1] UT term for kill.

Capture the Flag: The players are divided into two teams. Each team has a base with a flag that they must defend. Points are scored for a team when a team member captures the opposing team's flag, by bringing it back to the team's base while their own flag is safely contained in the home base.

Common to all these game types is that players respawn at random locations when killed.

2.1.2 Opponents in Unreal Tournament

In UT the player can play against the built-in botsor other players connecting through LAN's or the Internet. The AI controlled bots in UT is by many people considered to be formidable opponents. They can be extremely hard to beat and to a certain level their behavior can be conceived as that of a human. The strength of the bots is only partly due to cleverly programmed scripts. Since the bots access information hidden to the human player (eg. player positions, though not visible in the line of sight), they have an advantage compared to human players. The UT environment is using built-in noise as default when aiming - meaning that even perfect aim at an opponent, will not guarantee hitting the opponent. The noise varies depending on the weapon used. The bots gain an advantage when increasing their skill level, because their aiming noise will be reduced, which is not the case for the human player.

2.1.3 Items In Unreal Tournament

To succeed in any of the different game types - let it be Domination, Death Match or Capture the Flag - it is an asset to be superior in combat against an enemy. To master this ability, one should be able to perform a number of actions at the right time. Some of these actions are listed below:

- Pick up health.
- Pick up armor.
- Pick up weapons.
- Pick up ammunition.
- Initiate offensive or defensive movement patterns.
- Choose an appropriate weapon.
- Aim with the chosen weapon.
- Execute a strategy (combine and plan possible actions).

As can be seen above, some of the mentioned actions involve the presence of health, armor, weapons and ammunition. These items will be described in the following sections. The basic elements of motion control for movement will be described in section 2.1.5

The characteristics of the various kind of weapons can be seen in the list below:

Impact Hammer: Close combat weapon which fires slowly and inflict a medium amount of damage. This weapon does not use any kind of ammunition. When fired against solid objects, the weapon can damage the instigator. The player is equipped with this weapon at the start of a game.

Chain Saw: Close combat weapon which inflicts continually damage when in contact with the enemy.

Enforcer: The basic weapon in UT. This handgun is accurate on medium to short range distances. The weapon has a slow firing rate. The weapon is inflicting instant but low damage.

Double Enforcer: When a player picks up a second enforcer he is given the possibility to utilize them both at the same time, by carrying one in each hand.

Shock Rifle: This rifle fires slowly but accurate, also on medium and long distances. It inflicts medium damage instantly.

Bio Rifle: This weapon fires clumps of sludge which glom onto solid surfaces. It then explodes after a short time or when touched by a player (also the instigator), causing a medium amount of damage. The weapon fires at a medium rate, has close to medium range and the clumps are flying slowly.

and hits instantly inflicting a high level of damage. When the projectile hits the opponent's head it kills instantaneously. A picture of the sniper rifle can be found on figure 2.2.

Ripper: The Ripper fires sharp blades which can ricochet off solid surfaces. The blades are can when ricocheting inflict damage on the instigator when careless. The blades is moving with high speed and can kill instantly if they hit a player's head, else they will inflict medium to low damage. The Ripper is firing at a medium rate.

Minigun: This weapon is firing the same projectiles as the Enforcer, but at a very rate of fire. If not used with care this weapon can run out of ammo in seconds, but can also reduce the enemy's health in seconds.

Flak Cannon: This weapon works virtually as a real life shotgun. It fires chunks of jagged metal which, like the razor blades from the Ripper, can ricochet. The closer it is fired against the enemy, the more damage it inflicts. In close encounter one shot is often enough to kill the enemy.

Rocket Launcher: This weapon launches rocket-propelled grenades that explodes on impact. The grenades are moving slowly, but inflict a medium amount of splash damage on impact with solid surfaces, and a high amount of damage when hitting the enemy. The exploding grenades can hurt the instigator if he fires against a nearby solid surface.

Table 2.1 shows the specific attributes for each weapon.

All players in a UT game starts with 100 initial health points. When the player reaches zero, he dies and respawns at some random spawning point. To avoid death in UT, a player can pick up different kinds of health packets. These are listed in the following:

Health Vial: Each health vial gives the player 5 health points, to a maximum of 199.

Health Pack: Replenishes 20 points of health, up to a maximum of 100.

Keg O' Health: Gives the player 100 health points, to a maximum of 199.

Armor

Besides the possibility to pick up health to avoid death, a player also has the option to pick up armor. Armor provides the player with armor points of which he initially has zero. The maximum of armor points a player can retain is 150. Besides providing the player with armor points, the different types of armor protect the player in different ways. The characteristics of the three armor types can be found below:

Thigh Pads: Provide the player with 50 armour points. They will absorb a percentage of all damage dealt a player, until they wear away.

Body Armor: Provide the player with 100 armour points. It absorbs a significant amount of, though not all the damage dealt to the player.

follows:

Weapons	Impact Hammer & Enforcer
Ammo	Bullets pack (50)
Health	100
Armor	0

2.1.5 Controlling a Player

A player who manouevre an UT agent has a set of commands available he can control by his keyboard and mouse. The set is given by the following description:

Strafe Right/Left: These two commands will cause the player to move sideways, either left or right, as can be seen in figure 2.3(a).

Move Forward/Backward: These two commands will cause the player to run forward or backwards as can be seen in figure 2.3(b).

Turn Left/Right: These two commands will cause the player to turn right or left and will also change the point he is facing. This is because the player is not able to turn his head. This movement can be seen in figure 2.3(c).

Shoot: This command will cause the player to fire the weapon he is holding in the present moment. He will aim in the direction he is facing.

These four kinds of basic control commands can be executed in parallel. As an example a player can strafe while turning against a fixed point and

- Provides a more friendly looking environment - The Magic Wizard theme; for an example look at figure 2.4.

- Makes it possible to do research within the field of human-AI collaboration and competition.

- Publicly available at Gamebot-Project (2001).

- Is built on a very popular game, which makes it interesting for other people than the usual researcher. This leaves a chance to gather a broad community working on similar tasks and share experiences.

2.2.2 The Gamebots System

The Gamebots system allows players in a UT game to be controlled by network sockets connected to clients that can be controlled by an application. It is thereby possible for an application running a player to send actions, which should be executed by the player in the game. The application also gain information about the game state, which makes it possible to plan the next action. In this way both remote controlled AI players, human players and the built-in UT bots can play at the same time, in the same game.

Players must master advanced AI capabilities to achieve the aims of the game types, this include path planning, memorizing the characteristics of the 3D environment (items, paths, etc.) and strategic planning.

[2]UT and Gamebots is ported to at least Linux, Windows and Mac.

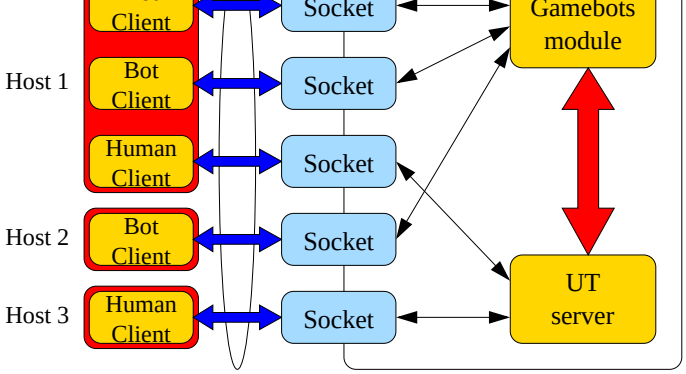


Figure 2.5: A sketch of the general organization of the Gamebots software (copied from Adobbati et al. (2001), used with permission).

2.2.3 The Gamebots Interaction Protocol

The Gamebots interaction protocol is a text based protocol of single-line messages between the server and the gamebots modification. The gamebots server sends sensory information messages to the bots containing the current state of the UT world. The bots operate in the environment by sending action messages back to the server. There are two kinds of messages; synchronous and asynchronous. The synchronous messages include things like visual updates and status of the bot itself. As the name implies they come at a regular interval. The asynchronous messages are typical events which are expected to happen less frequently, eg. if another player is visible or messages about incoming fire. The action commands are for example the movement

is performed (i.e. the search space) and F is the function to be optimized (i.e. the fitness function or objective function), and measures the goodness of every solution in the search space. Then: $F : \Omega \to \mathbb{R}$. Note that Ω can be finite, infinite, or defined according to complicate restrictions and F may be multimodal, non-differentiable, or defined according to complicate restrictions.

The objective is to find a solution (or, alternatively, solutions) $z^* \in \Omega$ such that: $F(z^*) \geq F(z)$ for all $z \in \Omega$ in the case of maximization, or $F(z^*) \leq F(z)$ for all $z \in \Omega$ in the case of minimization. A visual example of a fitness landscape can be seen in figure 3.1.

3.1.1 Search Strategies

When we want to solve a optimization problem, e.g. maximization of Ackley's [1] function depicted in figure 3.1, we have to consider some kind of search strategy. The choice of search strategy can depend on problem complexity, available computation power and various other parameters. A number of search strategies are available and a rough classification of these is sketched below:

- Brute force (e.g., depth-first)
- Heuristic:
  - Deterministic (e.g., hill-climbing)

[1] $F(z) = F(z_1, z_2) = e^{\frac{1}{2}\sum_{i=1}^{2}\cos(2\pi z_i)} + 20e^{-0.2\sqrt{\frac{1}{2}\sum_{i=1}^{2} z_i^2}} - e - 20$, where $-3 \leq z_1, z_2 \leq 3$.

EAs are:

- Genetic algorithms.
- Evolutionary programming.
- Evolution strategies.
- Genetic programming.

In short, EAs is a class of algorithms designed for searching in very big search-spaces. Although simplistic from a biologist's viewpoint, these algorithms are sufficiently complex to provide robust (good performance across a variety of problem types) and powerful adaptive search mechanisms. Basically, two steps are common to most types of EA:

1. Exploitation of known solutions.
2. Exploration of new solutions.

In the first step, the best solutions of a small set of known solutions are selected in a process inspired by natural selection, i.e. "survival-of-the-fittest". These selected solutions are exploited by construction of new solutions through recombination. Intuitively, this step might be thought of as a sense of small steps towards better solutions is exercised. This step is generally a tradeoff between random variation and structured variation. In the second step, new solutions are explored by creating new solutions from variations of the known solutions. Intuitively, this step aims to perform long

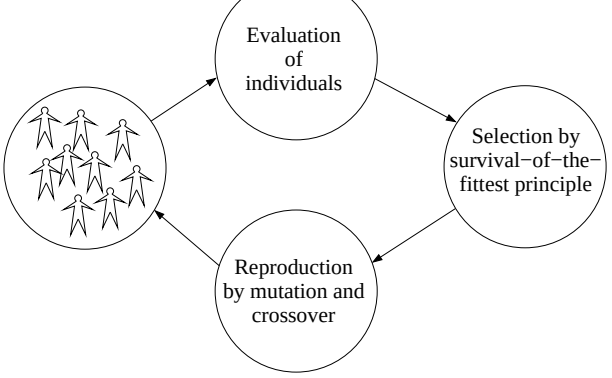


Figure 3.2: The loop of evolution.

the leftmost circle. This population is our current population, and moving to the topmost circle we evaluate each individual in our population with respect to some predefined fitness function. After doing so, we are able to perform selection according to the principle of "survival-of-the-fittest". Selection can be implemented in numerous ways, and we mention only the most commonly used:

Roulette Wheel Selection (RWS): Each individual is assigned a probability of being selected proportional to its actual fitness. Formally the probability P_i of individual i being selected is:

$$P_i = \frac{F_i}{\sum_{j \in Pop} F_j}, \tag{3.1}$$

random with the probability that a less fit individual being pulled. With a tournament-size of two we get the probability P_i of individual i being selected:

$$P_i = \frac{2}{N}\left(1 - \frac{N - R_i}{N - 1}\right), \tag{3.3}$$

where R_i is the ranked fitness of individual i and N is the size of the population.

TBS compared to RBS allows superindividuals a higher level of dominance, but still not to the extend of RWS. As an example, a superindividual in a population of size 10 has the probability of approximately 18% using RBS and 20% using TBS. Furthermore, the least fit individual can never be selected using TBS.

The last step of the evolution, is the process of generating the next population from the set of individuals selected for reproduction, and the bottom circle in figure 3.2 contains the typical operators for doing just that, namely mutation and crossover. From applying these genetic operators to the subset of the population that was selected in the previous step, a new population emerges and the loop is ready to start a new cycle. Many different types of crossover and mutation has been investigated throughout the years, the most primitive and probably most commonly used being the one-point crossover and one-point mutation. These basic operators are depicted in figure 3.3 and amongst a few other commonly used operators are described below:

[2] A superindividual is an individual with a fitness several degrees of magnitude higher than the second most fit individual in the population.

from each parent should be recombined into two offspring.

Uniform Crossover: A randomly generated mask identifies the cut-points that should be used. This mask is typically implemented as just a randomly generated array of bits with a length equal to the number of possible cut-points. Based on the value in this array cut-points are either used or not used.

Single-point Mutation: The property at a randomly selected mutation-point of the individual is mutated to create one offspring. In doing that we might destroy the good properties that made us select this individual in the first place. On the other hand we might as well improve the individual by changing some bad property to a better one. And even more important, mutation is the only way new genes can be introduced in the population, and thereby keeping a level of diversity[3]. No matter what happened, this new individual is added to the next population. In figure 3.3, the head of the individual is mutated to create the new individual. The mutation activity is part of the exploration phase.

Gaussian Mutation: Instead of mutating a property totally at random, the mutation operator chooses the new value of the property based on a Gaussian distribution around the current value.

Many other methods for crossover and mutation exists, we have just mentioned a few.

So, the population of individuals in the next generation is first of all composed of offspring produced by the genetic operators, thereby replacing indi-

[3] In sections 3.2.2 and 3.2.3 we discuss the existence of diversity in a population.

be: [001|10][100|11]→[00111][10010]. Mutation would be just to flip some randomly chosen bit, i.e.:[0111**1**]→[0111**0**]

In this section we have only included the most common and basic methods for crossover and mutation. These are good for the purpose of explanation, and others are typically modifications of or extensions to these basic ideas.

3.2.2 Ensuring Convergence

We stated earlier in this section that GAs perform a search in some large search space. Therefore it is reasonable to ask the question: can we be sure to find what we are looking for? [4]. The answer is: No, not if the search space is too large to search by brute force. But we can ensure that if we visit the best individual in the search space (the global optimum), then we will keep that individual. This is ensured by adding the concept of elitism to the loop in figure 3.2, which means that the best individual of a generation is copied (without modification) to the next generation.

Apart from the insurance of keeping the globally best individual if visited, keeping the best individual has an immediate advantage if the fitness function is simple and with few optima. If only a few optima exist, then why not try to climb one as soon as on is found?

A side-effect of adding elitism is that the algorithm in general will converge faster to some optimum, and you run the risk that this optimum is not the global optimum but some local optimum. This situation is often referred to as premature convergence. Figure 3.4 shows a population, and their location in a fitness space. This population is in danger of converging prematurely at

[4] We are looking for the best individual in search space, remember?

by adding elitism. You might even consider to copy the two or three best individuals to the next generation, to further speed things up.

It is possible to investigate the convergence properties of GA with a more formal approach than the one taken in this thesis. It has previously been done, famous examples are *Building Block Hypothesis* by Goldberg (1989) or the seminal *Schema Theorem* by Holland (1992). Schemata are genotype templates that define a subset of the search space. A schema is encoded in the same language used for encoding solutions with the addition of a special *don't care* symbol #. So, if we encode solutions using fixed length bit strings, an example of a schema could be [00#1#], and this schema is sampled by the individuals [00010],[00011],[00110] and [00111]. The *order* of a schema is the number of fixed positions (non-# symbols) and the *defining length* of a schema is the distance in positions between the first and the last fixed position of the schema. For instance the schema [0###1#] has *order* 2 and *defining length* 4. The *Schema Theorem* states that short low order schemata which is sampled by fit individuals will rapidly spread throughout the population. The *Building Block Hypothesis* states that late generation individuals samples many such short low order schemata, and hence are composed from many small and good building blocks.

3.2.3 Maintaining diversity

We need to explore new areas of the search space and not exploit a known optima for too long, in order not to get trapped in a local optimum. As a side effect the search is slowed down, that is, convergence will be delayed. This is very important for complex domains in which fitness functions have multiple local optima, as the risk of converging prematurely is generally higher for a

In equation (3.4), Pop is the population. The sharing function maps a distance measure $d(i,j)$ to a sharing factor in the interval [0..1]. As a rule of thumb, the maximum sharing factor should occur between two individuals if they have the minimum possible distance to each other, e.g. identical individuals i and j should have $s(i,j) = 1$. For a population of size N, a total of $\frac{N(N-1)}{2}$ distances must be computed to totally order the population, and hence this calculation can not be allowed to be very expensive.

The aim of this technique is to avoid the situation where all individuals in the population occupies the same peak in the fitness landscape. By investigating equation (3.4) it is obvious that the greater the peak, the more individuals can be allowed to inhabit it.

Crowding

This commonly used non-niching technique is one of the earliest of its kind, first proposed by De Jong (1975). Using this technique, generations are not clearly bounded but rather a *steady state* model is used, in which generations overlap. A proportion of the population (referred to as the *generation gap* G) is chosen to reproduce, and the new offspring replaces existing individuals according to some scheme. A common scheme is to extract a sample set of size CF (*crowding factor*) from the population. An offspring then replaces the member from the sample set that is most similar to the offspring. The effect of this approach is intuitively that, as offspring replace individuals according to similarity, subpopulations or spices could be expected to emerge. The number of spices that we would expect to emerge is of course controlled by CF, De Jong (1975) had success with CF values of 2 and 3 leaving room for a few spices to evolve.

have a lethal impact on the evolutionary process itself.

The Island Principle

The island principle is actually not just one method, but rather a collection of methods where subpopulation explicitly (sometimes even physically) are evolved in parallel. Most of the methods has been developed mainly for distributing the GA, but the methods also holds the inviting property of introducing an extended level of diversity [5] into the population. The methods (like everything else in GA) is inspired by natural evolution in which you do not see one global population evolving as a unity. Instead numerous subpopulations evolve in parallel [6] and, from time to time, individuals migrate from one population to another, thereby spreading its inherited genes to new areas. In GA we therefore can do similarly. The initial population is split up into X subpopulations, and the loop of evolution (see figure 3.2) is started on each subpopulation. From time to time some migration should take place, according to some topology. Fernández et al. (2001) suggest the two commonly used topologies depicted in figure 3.5. Figure 3.5a shows a ring topology, in which individuals migrate only in one direction contrary to the mesh topology, in which individuals have a choice of four directions when migrating.

The migration could either proceed in some random order allowing populations to grow and shrink. But we are not interested in ending up with

[5] E.g. see Fernández et al. (2001).

[6] These parallel evolving subpopulations are (in nature) often based upon different fitness functions. That is, a strategy might be successful on the south pole, but may fail in Africa.

migrations per generation, you would over time approximate an even distribution of the number of individuals on each island. Another approach is to have a fixed deterministic migration schema, and thereby keeping the population sizes static. Using the ring topology, it would be obvious to just let one individual migrate each generation.

If you implement elitism and are afraid that the evolution will suffer from premature convergence, then a possible approach could be to let PM be proportional to the fitness of the individual. This ensures that the best individual will migrate more often than other individuals and hence, not be able to dominate one specific subpopulation. It is of course true, that such a superindividual then just as well could dominate the total population by moving around spreading its genes in every subpopulation on its way. But it will take appreciable more generations for it to do so, as it never stays for more than a few generations in the same subpopulation. And it is likely to encounter a subpopulation in which some individual is more fit, and hence will no longer be subject to elitism.

The evolution that we expect by applying the island principle, could be describe by the snapshot of a population of five subpopulations shown in figure 3.6. In figure 3.6 we see that there are typically more than one subpopulation present at a specific peak. This is what will happen if the best individual of a subpopulation is forced to migrate. As a subpopulation converges to a local optimum it is generally guided by the best individual in the population, as this is the individual that will be selected for reproduction most often. But as the best individual is replaced by migration every once in a while, the whole population will generally shift direction, and ideally search all local optima on its way. Here is a scenario, based on figure 3.6: The islands are connected according to the ring topology, and the order is black,

Algorithm (MDPGA) is a kind of compromise between the single global population approach, and an island approach. As the name suggests, it was mainly developed for distributing GAs, but it is nonetheless interesting when investigating diversity and convergence properties in populations.

As in the basic island approach, several small populations are maintained, but the individuals are organized in a mesh structure as depicted in figure 3.7. In the approach described by Shumeet (1992), each subpopulation contains

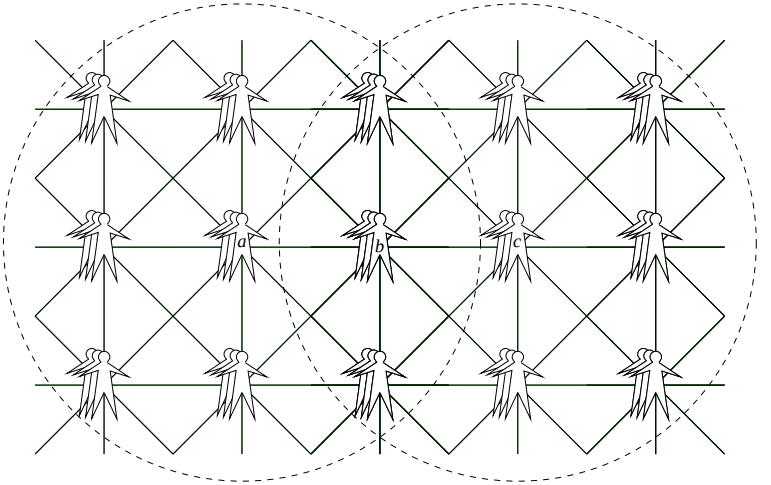


Figure 3.7: Organization of subpopulations in the MDPGA. The neighborhood of sub-population a and c are shown as dotted circles. Subpopulation b is a common neighbor to both a and c.

10 individuals. After the initial populations have been randomly generated, the algorithm proceeds in each generation as follows:

which efficiently slows down evolution, as genes are only spread in a few directions at a time. This allows for a high level of diversity.

The most improvement is of course obtained when implemented on large cluster machines as the topology of the subpopulations is inviting to these architectures. With the heavy communication taking place in the MDPGA, it is directly designed for lucrative implementations for MIMD [7] machines, which is also emphasized by Shumeet (1992). The island principle with a lower rate of migration is more suited for an heterogeneous distributed system with lower bandwidth.

3.2.4 The Fitness Function

The activity of designing the fitness function should of course be given profound attention, as this parameter will guide the evolution more than anything else. In this section we will present a number of co-evolutionary approaches. Most of the references given exemplifies these approaches in the domain of GP, but they are not specific to GP and could be used in GA as well.

The optimal solution to the task you are optimizing is not known, and in addition it might not be possible to guarantee that it will score maximum fitness. That is, if you try to evolve a strategy for a certain game, the fitness function that assesses a given individual against all valid strategies would intuitively be able to evolve good strategies. However, apart from very simple games, it is typically not possible to construct all possible strategies for this fitness function, and hence we need another approach. Koza (1992) presents

[7] **M**ultible **I**nstruction **M**ultible **D**ata.

rest of the population, that is, an absolute fitness. As argued in Angeline and Pollack (1994), this traditional fitness measure is not likely to perform well for very complex tasks. Instead a competitive approach in which individuals are evaluated relative to the rest of the population is suggested. Angeline and Pollack (1994) investigate three different competitive fitness functions, two of which were previously used by Axelrod (1987) and Hillis (1992). The principle of these two methods can be seen in figure 3.8(a) and 3.8(b).

All-Against-All Tournament Fitness

Axelrod (1987) used the all-against-all approach (see figure 3.8(a)) to evolve strategies for the iterated Prisoners Dilemma. Using this method, the fitness of an individual is based upon that individuals' performance against all other individuals in the entire population. With this strategy and a population size of N we need:

$$\sum_{i=1}^{N} N - i = \frac{N(N-1)}{2}, \tag{3.5}$$

competitions in order to determine fitness of all individuals in the population. The obvious drawback of this approach is the extensive computation due to the high amount of competitions. The advantage is that a total ordering of all individuals relative to the population is obtained.

or even correct. First of all, the ordering only ensures you that the most fit individual is found. The looser in the final round is ordered second place, but the globally second best individual might by chance have been paired with the globally best individual in the first round, and therefore ordered equally with the least fit individuals. This is the main disadvantage of tournament fitness.

Co-Evolution Fitness

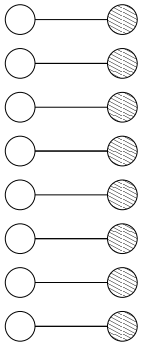


Figure 3.9: A co-evolutionary environment constituted by two parallel populations, as used by Hillis (1992).

A third approach that avoids the use of traditional fitness measures is co-evolution. Koza (1991) presents the concept of co-evolution in EA as a process in which the environment of one population is constituted by one or more population(s), evolving in parallel. The environment of the parallel population(s) is of course constituted by all other parallel populations. Individuals from the first generations will in general be highly unfit when compared to an absolute optimal solution, and fitness-values assigned relative to the current environment are used.

Hillis (1992) used the co-evolution strategy (see figure 3.9), in which individuals are assessed against an individual from a parallel population. This is best explained with the example below.

Assume that our individuals are evolved to classify textstrings, then we create two populations namely a population of classifiers and a population of text-strings. Now, these two

formulation of a solution than GA.

The field of GP is relatively new, and the presentation given in this section is based upon the work by Koza (1992), as it contains one of the most comprehensive introductions to GP.

One of the most important considerations is the definition of the language from which our individuals will be built. The language is union of the sets $\mathcal{T}$ (the set of all terminals) and $\mathcal{F}$ (the set of all functions). To create an individual elements from the two sets are combined to construct a parse tree. The leafs of such a parse tree are of course elements of $\mathcal{T}$, while the inner nodes are elements of $\mathcal{F}$. In the following sections we will describe how to apply genetic operators to individuals represented as parse trees, discuss how to construct $\mathcal{T}$ and $\mathcal{F}$ and last discuss the fitness function.

3.3.1 Genetic Operators and Parse Trees

The parse tree representation is often chosen throughout the literature, and it has numerous advantages over e.g. higher-level source code or low-level binary machine code. The problem you face with both alternatives to parse trees is the same: it is hard efficiently to ensure that only syntactically correct programs are constructed by crossover and mutation. Even when applying these operators to parse trees results in syntactically correct trees, we still have to check that all data types are correct for the different functions in our tree, unless we require $\mathcal{T}$ and $\mathcal{F}$ to satisfy the closure property, stated by Koza (1992).

Property 3.3.1 (The Closure Property) *The closure property requires*

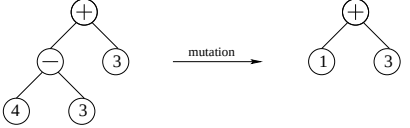


Figure 3.10: Crossover and mutation applied to parse trees.

exist. Below we have described some of these:

Subtree-swapping Crossover: After two individuals have been selected for reproduction, a random cut-point being a node in the tree is chosen, independently for them both. In contrary to GAs, this cut-point need not be common to the parents, as GP individuals generally differ in both shape and size. When cut-points have been chosen, the two subtrees with root at the cut-points are exchanged, producing the two offspring.

Context preserving Crossover: The chosen cut-point must be common to the parents. That is, the path from the root to the cut-point must be the same for both parents. Apart from this constraint on the chosen cut-point, this operator swaps subtrees like the above mentioned method.

Subtree Mutation: A random cut-point is chosen, and the subtree with root at the cut-point is exchanged with a randomly generated subtree. In figure 3.10 the randomly generated subtree is a tree consisting of only the terminal node with value 1.

Point Mutation: A single node (function or terminal) is chosen for mutation. Only the node is mutated, that is, any subtree(s) below this node

1 and -1 respectively.

Redefinition of comparative operators: Instead of returning a boolean value, the comparative operators accept two additional arguments, and execute one of them based on the result of the comparison. That is, the `equals` operator should accept four arguments, so we have:

`equals(arg1,arg2,true-branch,false-branch)` instead of

`equals(arg1,arg2)`.

And the semantic would of course be to execute `true-branch` if `arg1` is equal to `arg2`, and execute `false-branch` otherwise.

Redefinition of branching operators: The common branching operators could be redefined to check on some condition external to the program, typically some sensory information. In this way no general branching constructs should be included in $\mathcal{F}$, but only specialized branching functions. For instance the sensory information `enemy-in-reach` should not be accessed directly like:

```
if(enemy-in-reach, then-branch, else-branch)
```

rather it should be encapsulated in a function like:

```
if-enemy-in-reach(then-branch, else-branch)
```

that accepts two arguments `then-branch` and `else-branch`. Based on the value of the external sensor variable `enemy-in-reach` the function executes either `then-branch` or `else-branch`.

It should be noted that Montana (1993) proposes a method for Strongly Typed Genetic Programming (STGP) in which the closure property is totally

One way to check if the sufficiency property is satisfied, is to try to construct a known solution to the problem with the $\mathcal{F}$ and $\mathcal{T}$. If this is possible, then you know that one solution is possible. And if you did not design $\mathcal{F}$ and $\mathcal{T}$ towards expressing this specific solution, then chances are that it is also possible to express other solutions.

- To build a bot by means of evolutionary methods, and more specifically the GP paradigm. That is, a system that allows our bots to evolve through generations must be implemented.

- The system should be designed in a fashion that makes the evolutionary process, to some extent, immune to many of the known problems connected with the application of evolutionary methods.

- The bot should be benchmarked against the UT bot that comes with the environment, e.g. the UT bot or some similar bot. Also, it should be evaluated by human experienced players. The buildin bot provides a good benchmark test, as it has at least as much information at its service as does our bot. Also, it has to work under the same conditions as our bot. That is, parameters like maximum speed of movement and aiming noise will be the same, which is not always true for a human player.

In the reminder of this chapter, we will summarize our attempt to reach these goals.

4.2 Language Design

This section describes the language developed for describing strategies, and it is composed by the two sets $\mathcal{F}$ and $\mathcal{T}$ - the set of functions and terminals, respectively. The language is designed without general purpose in mind, rather it is designed for the specific domain of Gamebots as described in chapter 2.

`(armor)`.

These terminals assume the values of health, ammo for the current weapon (mapped to the range [0..200]) and armor of the bot in the game. Health and armor are already in the range [0..200] and need not be mapped.

We want the bot to be able to evolve its own perception of what conditions are good and what conditions are bad [1], therefore we include the general branching structure:

```
(if-less-than arg1 arg2 arg3 arg4)
```

The semantics of this function is, of course, that based on the boolean value of the comparison ($<$) of `arg1` and `arg2`, either the value of `arg3` or `arg4` is assumed. We could also include functions like `(if-greater-than arg1 arg2 arg3 arg4)` and `(if-equals arg1 arg2 arg3 arg4)`, but this would not add to the expressiveness of the language and is therefore not included.

We add the possibility to represent constant integer values from the range [0..200]. This is done by adding the terminal:

```
(const x)
```

where `x` is replaced by an integer value from the range [0..200]. The following branching functions are included:

```
(if-health-in-reach arg1 arg2)
```

[1] What value of health is considered low, what value is considered high, etc.

`(face-enemy)`

`(shoot)`.

The `(face-enemy)` automatically rotates the bot to face the enemy if the enemy is in sight, and otherwise does nothing. The function assumes the value 200 if the enemy was in sight, and 0 otherwise. The `(shoot)` terminal commands the bot to fire a shot with the current weapon and the terminal assumes the constant value 0.

The following terminal assumes a value equivalent to an estimate of the current damage taken by the enemy. It has max-value 200 and minimum 0, and it decreases over time to 0, as it is reasonable to assume the enemy picking up health packets over time. We include the terminal:

`(enemy-damage)`.

In order to compare weapons, the following terminals are included:

`(my-weapon)`

`(enemy-weapon)`.

These terminals assume values ranging from 0 to 200, according to the weapon currently used by either the bot or the enemy. This enables strategies to compare weapons, and take different actions accordingly.

`(if-enemy-move-left arg1 arg2)`

`(if-enemy-move-right arg1 arg2)`

of the enemy. They all execute and assume the value of `arg1` if the enemy is in sight and has the respective direction of movement. Otherwise they execute and assume the value of arg2. The distinction between different relative enemy movements are depicted in figure 4.1.

The specific movement of the enemy might not always be of interest if you are too far from your enemy. Therefore we add the terminal:

```
(enemy-distance).
```

This terminal will at any time assume an integer value in the range of [0..200] corresponding to the distance to the enemy, or 0 if the enemy is not within sight.

4.2.3 Higher Level Skills

As mentioned previously in section 2.1.5, it is common to combine different actions in a series of parallel actions, like `(strafe-left)` and `(face-enemy)` resulting in a circular movement with the enemy as center. This combination of movements is common to all players of UT, novice as well as master. It therefore seems reasonable for us to include specific terminals and one function for this movement:

```
(circle-strafe-left)

(circle-strafe-right)

(strafe-relative arg1 arg2).
```

$$y' = y_{enemy} + (\sin(\theta) * u - \cos(\theta) * v) \quad (4.2)$$

where (x_{enemy}, y_{enemy}) is the location of the enemy, u is the value assumed by `arg1`, v is the value assumed by `arg2` and θ is the yaw [2] orientation of the enemy. That is, the point (u, v) is transformed from a local coordinate-system inserted on top of the enemy (the enemy located at (100,100)), to the global coordinatesystem. This is depicted in figure 4.2(a) and figure 4.2(b).

As the last element, we add a function that enables strategies to put more functions and terminals in sequence:

`(prog-2 arg1 arg2)`.

This function evaluates first `arg1`, then `arg2` and finally assumes the value of `arg2`.

4.3 Designing the Algorithm

The common GA/GP cycle was described in section 3.2.1 in figure 3.2. In this section it will be extended, with the steps we find feasible for the development of successful genetically programmed bots. The specific operators and methods chosen, will be described including minimal justification. As with every topic covered in this chapter, extensive justification, argumentation and discussion of the decisions made is found in Holm and Nielsen (2002). The different steps of the extended GA/GP cycle in figure 4.3 are briefly described in the following enumerated list:

[2] In terms of Yaw, Pitch and Roll systems, the Yaw component describes the rotation about the Z-axis.

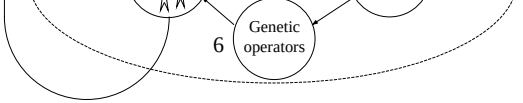


Figure 4.3: The extended loop of evolution.

1. The initial population is generated by either the custom tree generator or the random tree generator. The tree generators are described in section 4.3.1. There are no limit to the size of the initial population, since it is going to be reduced by the assessment step 2.

2. The assessment function is implemented to ensure a certain level of quality in the run of the first generation in the extended GA/GP cycle. The assessment function will remove individuals which can not satisfy a basic set of constraints, these are described in section 4.3.2. The assessment function assesses individuals much faster than the evaluation function.

3. The population used in the extended GA/GP cycle consists of 224 individuals evenly divided into 7 subpopulations. This yields a subpopulation size of 32. Every subpopulation will then run its own GA/GP cycle. This could be done in parallel if implemented on a multiprocessor system.

4. Every subpopulation is hereafter evaluated by the competitive evaluation function described in section 4.3.2.

strongly typed Genetic Programming (STGP) is a way to limit the space of possible trees. We have applied a slightly different method for a custom tree generation and it is described in section 4.3.1.

Another important factor in this step is the size of the population, and we have chosen it to be 224 divided into 7 subpopulations or islands of 32 individuals each.

Tree Generators

To generate an initial population, and to create the subtrees used in the mutation operator, we need an algorithm for tree generation. For this purpose we have adopted the traditional tree generation algorithm, GROW in Koza (1992), which is described and motivated in the following section. GROW has shown serious weaknesses in more simple experiments. As a consequence we have designed a custom tree generation algorithm based upon the theory on STGP, as described by Yu (2001).

The Random Tree Generator The common tree generation algorithm, GROW, is used to generate trees for the initial population and the subtrees used by the mutation operator. The algorithm uses a set, S, of functions and terminals to place as nodes in the tree. It chooses all the nodes randomly from S, until the chosen maximum depth is reached. The algorithm works in a recursive manner by selecting a root and then call itself to find descendents to the currently selected node until a terminal is selected or the maximum depth is reached. The algorithm is shown in table 4.1.

[3] Of course mutation can reintroduce functions and terminals in the population.

Table 4.1: The most common tree generation algorithm, GROW

The obvious advantage of GROW is that it is easy to implement and runs in linear time. Still some disadvantages remains, and as mentioned by Luke (2000), GROW has the three main weaknesses:

- It selects between all the possible functions and terminals with equal probability, which in some cases can be undesirable.
- It does not allow any control over the tree structures, except for the size.
- It does not create trees with a fixed or average tree size or depth (this weakness is mentioned as the most significant).

The Custom Tree Generator By introducing the custom tree generator we will also introduce an abstraction of typed GP. Yu (2001) states that there are two causes to prefer STGP:

- STGP removes the closure requirement and thereby increases the applicability of GP.
- STGP helps GP searching for problem solutions using type information.

box.

With the chosen function and terminal set we describe and motivate in section 4.2, we need not remove the closure requirement - this eliminates the first motive for STGP. But it might be possible to reduce the tree search space. When examining the function and terminal set, we find it sensible to reduce the set of possible arguments for certain functions. Koza (1992) made the first attempt to introduce types to GP with what he described as constrained syntactic structures. That is, the trees constituting the individuals in the population must obey some special problem specific rules of construction. When adopting this system of constrained syntactic structures he mentioned some issues which should be considered:

- The initial population must inherit the defined constrained syntactic structure.
- Genetic operators, such as mutation and crossover, that alter and create individuals, must produce trees that also preserve the constrained syntactic structure.

In our problem, a constrained syntactic structure is just another way to perceive STGP. A convenient way to describe the constrained syntactic structure is through a Context Free Grammar (CFG). We have designed such a grammar and the alphabet of the grammar, representing the functions and terminals of our language, is described in table 9.1.

consists of all the functions from our function and terminal set.

The "if" set

$$I_{set} = \{D..N\}$$

consists of all the functions being `if` constructions except `if-less-than` which differ in that it uses two of its arguments for evaluation.

The "action" set

$$a_{set} = \{h..q\}$$

consists of all terminals which causes some kind of bot action when executed.

The "sensor" set

$$s_{set} = \{a..g\} \bigcup \{r\}$$

consists of all terminals representing game information and the constant terminal.

The complete CFG is observable in table 4.3.

4.3.2 Fitness Functions

We designed two functions; the assessment function and the evaluation function. They will be described in the following two sections.

1. We are dealing with a complex problem and for such a problem it can be difficult for the population to evolve if no precautions have been taken.

2. Due to the dynamic nature of the domain it would be preferable that the evolved bots are not specialized to certain situations and can be efficient and effective against various types of opponents.

3. The aim, i.e. to evolve a bot for combat situations, should be kept in mind. That is, the environment should mirror a typical combat scenario as much as possible.

4. We want relative fast evaluation times, since we can expect to run a considerable number of generations before a satisfying solution is reached (if ever). When taking the population size of 224 individuals into account, it should be obvious that every second saved for evaluation of an individual will be important.

These issues address considerations which should be done and problems which are to be dealt with.

We will use a competitive fitness function, since this type of function deals with several of the issues mentioned. Firstly, this type of function deals with the complexity of the problem and the typical aftermath of building fitness functions to these kind of problems [4].

[4] For further reading see Luke (1998) or Nolfi and Floreano (2000).

reduce total evaluation time.

When dealing with these issues, another element to consider is the choice of environment in which the bots are going to compete. It would be preferable to use a dynamic environment which should reflect the distinctive surroundings of typical combat scenarios. One way of doing this (and maybe the best), would be to switch maps once in a while, when running the algorithm. This would avoid specialization of the bots for certain maps and instead evolve bots with more general strategies. Due to the time constraints of this project and since it has proven difficult to implement this feature, we have been forced to look at alternatives to this approach. In figure 4.4 a sketch of the map we have used is depicted. The scale and dimensions of the map depicted in figure 4.4 and the real map is not kept, but the general nature of the real map is kept. The map is circular and without any corridors or staircases, so minimal path finding and navigation is required. All four different classes of items described in section 2.1.3 are present, that is: weapons, first aid kits, ammunition and armor. The spawning points are located on a circle around the center of the map, and bots are spawned with orientation towards the center, as shown in figure 4.4.

To determine a fitness of each individual, all we need, when using the cup tournament based competitive fitness function, is a way to determine a winner when a pair of bots are competing. For doing this, it seems obvious to choose the bot with the highest number of frags as the winner, hence this will be the primary decision factor. If the score in frags should be even, the amount of damage given is compared and should the match still be a draw, the individual with the least suicides wins. As a last resort, the winner is found by coin toss. The way of determining the fitness of individuals could raise the following question: does this method not just seem too simple for

their actual fitness.

After the evolution was started, we performed a test to confirm or affirm our suspicion of the fitness noise. A randomly chosen best individual from one of the late generations was set to compete against a clone of itself, in 100 games where every game lasted 60 seconds. Figure 4.5 depict the frequency with which results of the matches was observed, yielding a histogram. As can be seen in figure 4.5 the noise of the competitions is obvious, though the results is far from random as the density is clearly higher, near to zero.

4.4 Genetic Operators

The genetic operators adopted or designed for our problem will be described in the following sections.

4.4.1 Selection

As explained in chapter 3, the choice of selection operator will influence the selection pressure in the population. We have already argued that we want to keep a relative low selection pressure. We have therefore chosen the tournament selection operator which is suited for this and can easily be adjusted by altering the tournament size, which we have chosen to be as low as possible; namely two.

4.4.2 Reproduction

The reproduction operator implements the concept of elitism, can regulate the rate at which the population converges. Furthermore it ensures that the best strategies survives and can be further evolved. A high reproduction setting will eliminate current weaker strategies at a faster rate, that is, a high reproduction setting will speed up local search but it can also have the drawback of eliminating potential strong but immature individuals. This property is one of our reasons behind the introduction of the islands principle and migration in our extended GA/GP cycle, since this allows us to maintain a high reproduction setting to refine local strategies while avoiding domination of the total population by decelerating the migration rate.

4.4.3 Crossover

The crossover operator is implemented as described by Koza (1992). The constrained syntactic structure is not enforced in the crossover operation due to time constraints and the reason that we want to let evolution decide what is good and what is bad, when the initial generation has been generated. When two individuals have been selected, a random cross point is chosen and a subtree swapping is performed as described in section 3.3.1, producing two offsprings.

4.4.4 Mutation

The mutation operator can take the custom tree generator in use when generating a subtree. The operator chooses a random cut point and inserts the

4.5 Tests

Three different runs were performed with different parameters, listed in table 4.4. The different parameters were chosen with comparison of results in mind. The most important task though, was the investigation of the evolutionary process itself and how the bots evolved. As the project served as preliminary studies preceding this thesis, it was more important to identify problems and opportunities in the applied domain, than to analyze all of the data collected through the test runs. It was also necessary to postpone a thorough examination of the collected data, since the amount of data was simply to extensive. In the end the evolved bots were compared to the UT bots and tested against a human opponent.

4.5.1 Parameters of the Evolution

In table 4.4, Assessment pool is the amount of custom generated individuals from which our initial population is created. Initial population is the size of the initial population, and Islands is the number of islands used. P_{cross}, P_{muta} and R_{migr} are the probability of crossover, the probability of mutation and the migration rate [5], respectively. R_{elite} is the amount of individuals that are transferred to the next generation unchanged. Initial time is the time that one match in the cup tournament based fitness function [6] will last

[5] For instance, $\frac{1}{2}$ means that the fittest individual migrates from an island every second generation.

[6] Refer to section 3.2.4, for a description of the cup-based fitness function.

during mutation.

Halting conditions

Neither of the three runs have had explicitly stated halting conditions, we have just evolved for as long as possible. Still no more than 100 generations seemed reasonably, since we spend about 2 hours evolving one full population of size 224. In effect that meant a few weeks of evolution on two standard PCs (700 Mhz and 1333 Mhz). As the problem domain (the game of UT) is running in realtime, an increase in cpu-cycles would have no effect on the evaluation time.

4.5.2 Performance Tests

To derive information about the abilities of the evolved bots, we have performed two kind of experimental tests. Firstly, we have tested some of the evolved bots against the UT bot. Secondly, we carried out a test between a human and an evolved bot. A description of the tests and a debriefing about the results will conclude this chapter.

Unreal Tournament Bot vs. Evolved Bots

The UT bot are widely known and esteemed in the gaming community for their strength of play, especially when they are compared to other bots in similar games [7]. For this reason we thought it would be interesting to see

[7] Quake, Half-Life, etc.

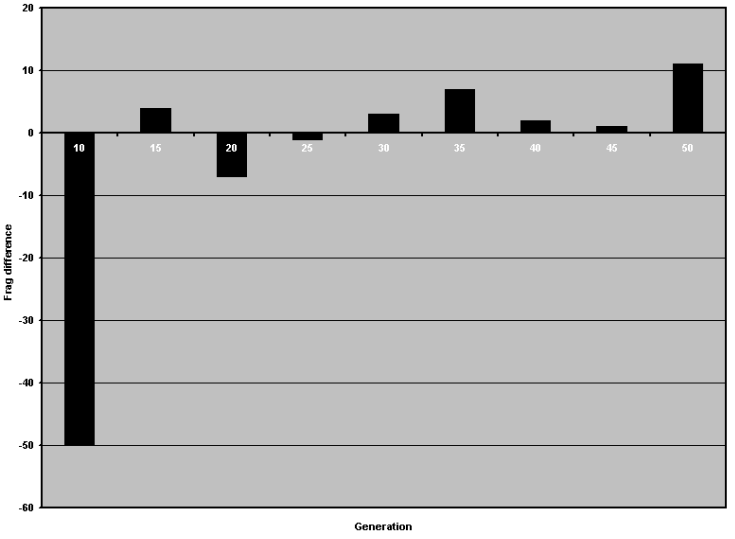


Figure 4.6: The difference between the UT bot and the different generations of the 3rd island 1st run. Notice that the evolved bots gradually improve the performance against the UT bot.

The UT bot has played bots from the 6th island of the 1st run, chosen with 5 generations interval from the 10th generation to the 50th generation. The fights were terminated when one of the contestants reached a frag limit, which was set to 50. The UT bot gets defeated the first time against the 15th generation bot. The last five evolved bots are all victorious and the 50th generation bot is distinctly superior to the UT bot.

evolved bot. In this test the best bot from the 32nd generation of island 3 in the 3rd run was pitted against an UT bot and a human. In addition the human played the UT bot. At last the human played a bot from the 82nd generation of island 3. Again, the match was ended when one of the contestants reached 50 frags. The result of the test can be found in table 4.6.

Firstly it can be seen that the evolved bot from the 32nd generation was superior to the UT bot. The human was even more superior to the UT bot, which was to be expected because of humans adaptive capabilities. It was also clear when observing the actual match that, as the match progressed, the human adapted more and more and was much better in the last half of the match. The match between the evolved bot and the human was more equal. In the start of the game the evolved bot was dominant and the human showed greater difficulties in adapting to this bot. As an observer it was hard to see if this was because the evolved bot used a more general difficult strategy to deal with for humans or it was because the strategy in general was evolved to be more resilient to a broad range of opposing strategies. To look further into this matter the human was pitted against the bot from the 82nd generation. In this match the human faced even greater difficulties and lost it. It would require a more exhaustive test to make definite conclusions about the general causes of the test results, since humans introduce a lot of difficulties when evaluating due to inconstant performance and the ability to adapt. Nevertheless we can conclude that the evolved bots tend to show resiliency against different strategies applied by a human and they are superior to the UT bot.

Agent specific: Extensions that affect the general architecture of the agents, and to some extend these such extensions are inherently isolated from the GP system. The only direct connection between agent architecture and the GP system is the language for describing strategies.

System specific: Extensions that affect the GP system. For example, extending the genetic operators used during reproduction with more features, or extending the population type to include a generation gap parameter and thereby using a steady state model. These extensions are not tightly connected to the agent architecture.

In Holm and Nielsen (2002) we proposed extensions of both classes as possible future work. In this project, however, we have chosen to concentrate on system specific extensions. More specifically, we will evaluate different extensions that might improve the search process.

In the following sections, we will perform an analysis of the result and experiences gained in Holm and Nielsen (2002) that will conclude in a more specific definition of the project goals.

5.2 Analysis of Previous Results

We concluded chapter 4 with a presentation of some performance tests on the bots previously evolved. These tests were not of exhaustive quantitative nature, but more meant as a random test to measure the quality of the evolved solutions. The results of these tests left us with the impression that the most efficient strategies evolved were relatively simple strategies and not very complex. That is, when playing against the evolved bots or viewing the

used. It should not come as a big surprise that simple strategies can be highly efficient. The arena is circular with no obstacles, so a circular movement is natural. A circular movement is easy to produce by on of the three constructs `relative-movement`, `circle-strafe-left` or `circle-strafe-right`. All these functions moves the bot in a circular motion relative to the enemy if he/she is in sight. However, this is always the case initially, and as there is no where for your enemy to hide, you can easily track him/her down with a good circular search path. So, combining any of these functions with a few `shoot` constructs you have a pretty good strategy. The reason `relative-movement` gains dominance unlike any of the `circle-strafe` functions must be that with `relative-movement` it is possible to move the bot out of the field of vision of the enemy, whereas the `circle-strafe` functions uses no knowledge of the current orientation of the enemy. So in effect, the `relative-movement` construct has good defensive properties in addition to the obvious offensive properties of dynamic movement.

The dominance of `relative-movement` and `shoot` is less noticeable in the 2nd and 3rd run (figures 5.2(a-b) and 5.3(a-b)), but still recognizable. In these two runs, the `shoot` construct is accompanied by the `const` construct.

5.2.2 The Flower of The Tree

So now that we have realized that the population seems to be dominated by a few constructs in the later generations, we ask ourself if everything in the tree should be considered flowers in bloom, or if some withered leaves can be identified. Following this question further, we will investigate the

[1] Refer to appendix A for the corresponding data divided among islands.

5.2.3 The Withered Leaves

We have now identified that large parts of the parse trees are not affecting the performance of the solution. We will postpone a discussion of the macroscopic effect of this phenomenon to a following section, and first absorb ourself in an analysis of different kinds of withered leafs, often referred to as introns.

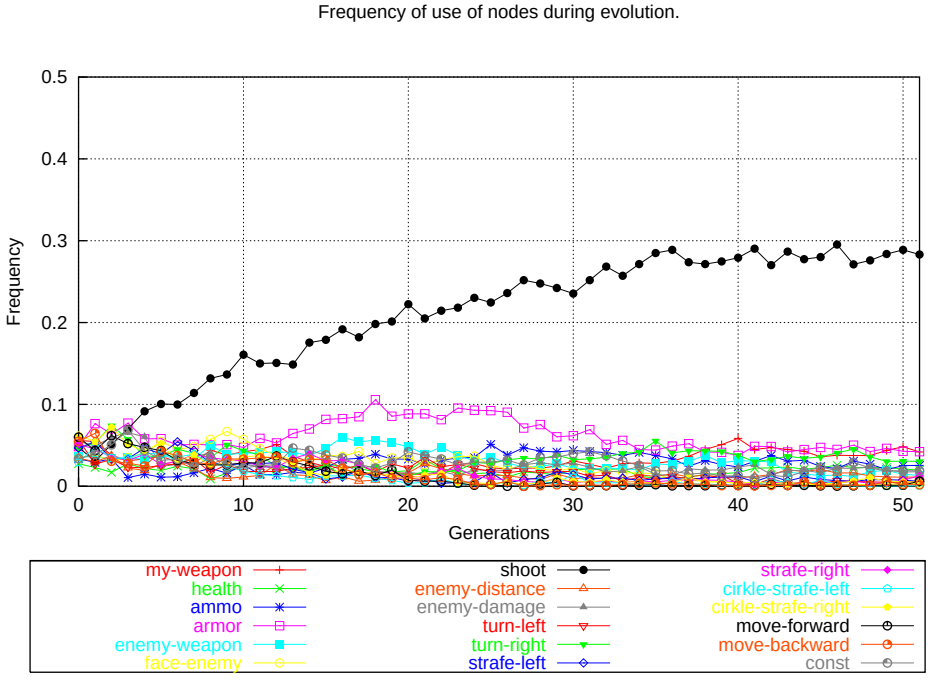
If individual X is produced by applying the subtree-swapping crossover operator to a cut-point within a large unreachable block of individual Y, then X and Y will perform equally and hence, no good qualities of Y has been successfully exploited. X and Y only differ within an unreachable part of the program, so in all reachable parts of the program X and Y are identical. This specific property is part of the definition of introns given by Nordin and Banzhaf (1995). They define introns to be blocks of code with the following properties:

1. The block has no effect on the performance of the program.

2. Offspring produced by applying the crossover operator inside the intron of the parent, will display performance and behavior equivalent to its parent.

[2]This observation is thoroughly described throughout the literature, Blickle and Thiele (1994) and Nordin et al. (1995) just to mention a few.

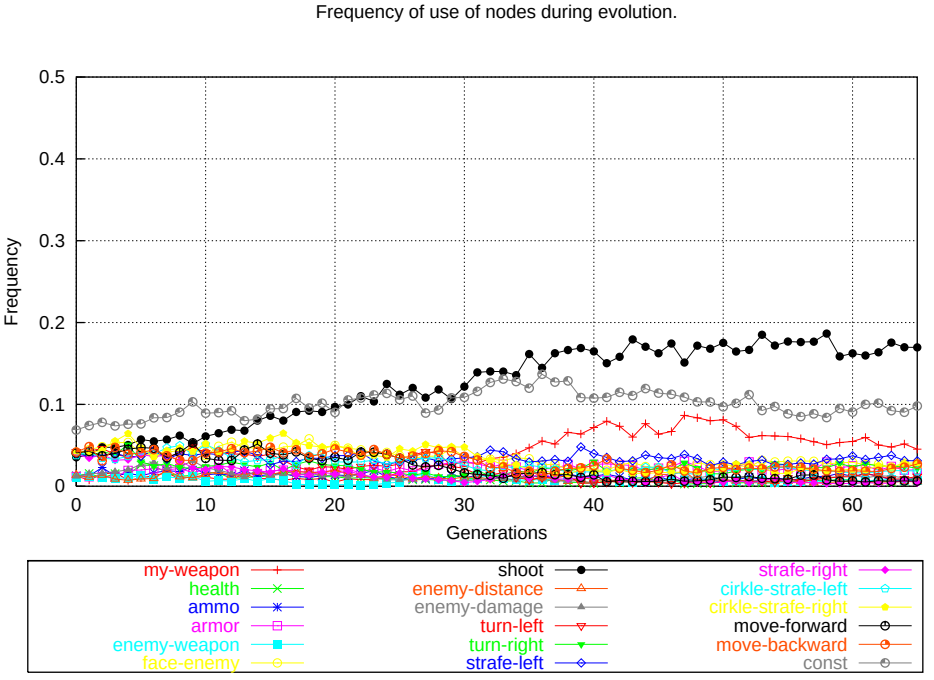
(a) Functions



(b) Terminals

Figure 5.1: Distribution of language constructs for the population of the 1st run.

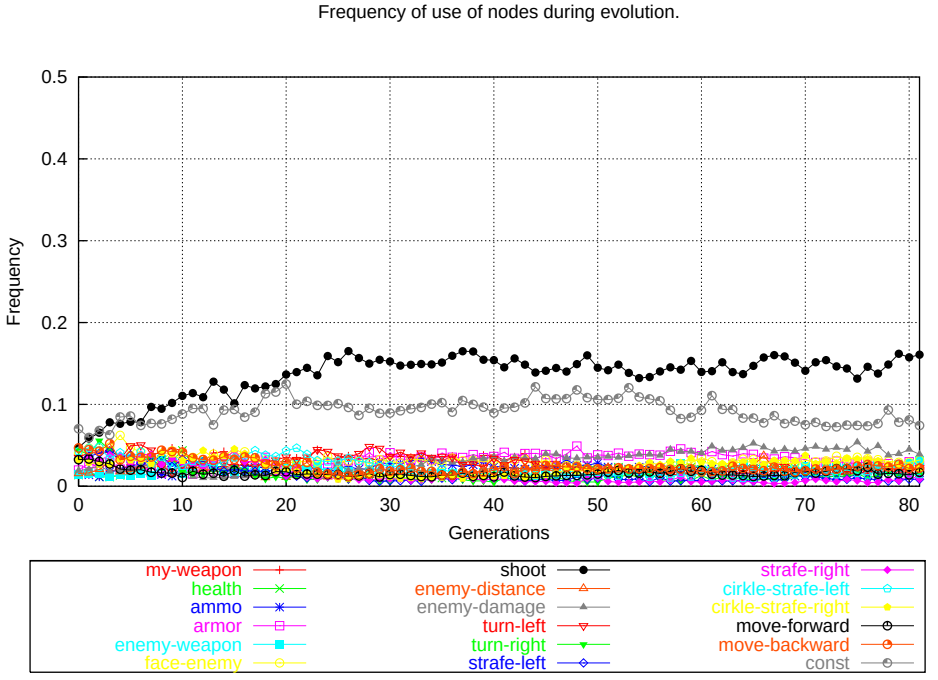
(a) Functions



(b) Terminals

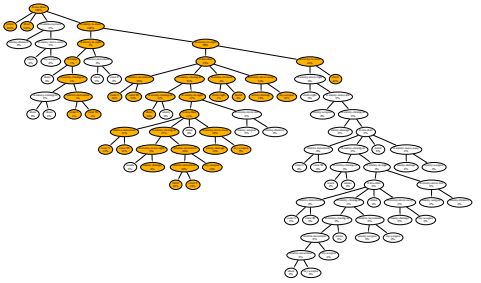
Figure 5.2: Distribution of language constructs for the population of the 2nd run.

(a) Functions

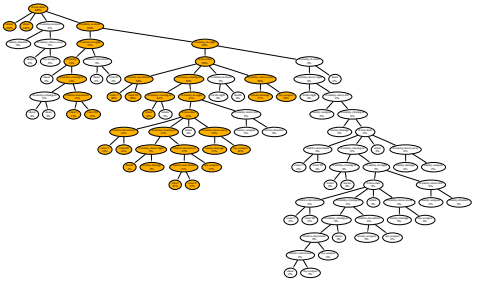


(b) Terminals

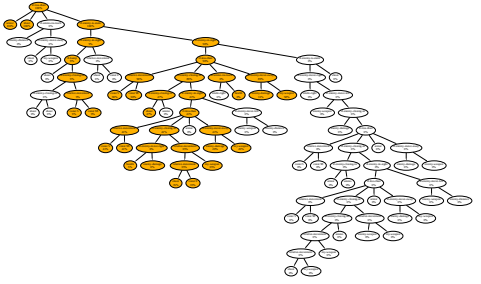
Figure 5.3: Distribution of language constructs for the population of the 3rd run.



(e) Match 5



(f) Match 6



(g) Match 7

Figure 5.4: The fittest individual from the 4th island, 60th generation, 3rd run has been pitted against 7 different enemies. Executed nodes is colored, and all nodes has been tagged with the proportion of executions.

(a) (b)

Figure 5.6: The fittest individual from the 6th island, 10th generation, 3rd run has been pitted against 7 different enemies. The executed nodes is colored, figure (a) shows the result from 6 out of the 7 matches, figure (b) shows the results of the last test.

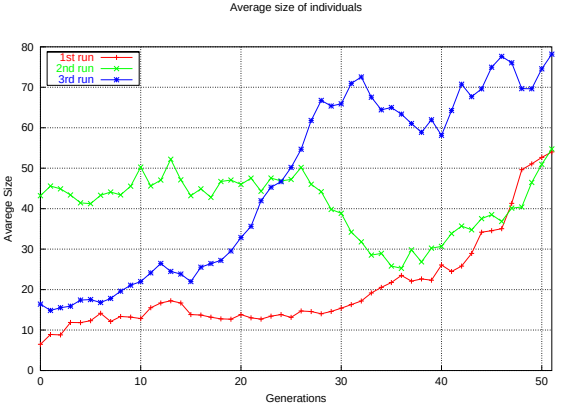


Figure 5.7: The evolution of average size for the 3 test runs.

input.

Local Introns: An intron is local if it is an intron for the current test case, but not necessarily for any other valid program input.

When investigating the parse trees in figures 5.4 and 5.5 further all white nodes are introns, either global or local. In figure 5.8, the parse tree from figure 5.4(a) is reprinted, and this time the root of a global intron is magnified, corresponding to the block of code:

```
(if-enemy-is-static
   (true-branch)
   ( ... false-branch ...
      (if-enemy-is-static
         (..global-intron..)
         ( ... false-branch ...))))
```

As can be seen from this construction, the redundant use of the boolean sensor check (`if-enemy-is-static`) gives raise to a global intron. Likewise we can identify local introns, this time we inspect the parse tree from figure 5.5. In figure 5.9, the parse tree from figure 5.5(a) is reprinted, this time with the root of a local intron magnified (by coincidence, this is also the root of the tree). The magnification corresponds to the code:

```
(if-enemy-in-sight
   (..local-intron..)
   (if-weapon-in-reach ..))
```

introns, so if all local introns are isolated you can be sure that all global introns has been isolated as well.

5.2.4 Causes of growth of non-executed code

Many theories as to why the proportion of introns seems to grow as the evolution progresses exist, three slightly different theories will be described in the following.

1. Amongst others, Blickle and Thiele (1994) and Nordin et al. (1995) describe the growth of proportion of introns as a kind of protection against the destructive effect of crossover and mutation operators. That is, crossover and mutation cannot alter (neither increase nor decrease) the performance of the program, if it is applied to an intron part of the program. Therefore, it could be expected that the evolution to a certain degree promotes solutions that are somewhat immune to the destructiveness of crossover and mutation. Another way of putting it is, that the number of introns in the population serves as an adjustment of the parameter controlling the crossover and mutation frequencies.

2. Soule and Foster (1998) has a slightly different explanation, also originating in the destructiveness of crossover and mutation, namely the removal bias. When offspring is produced by replacing a subtree from a parent, the good qualities of the parent are in danger of being destroyed. The larger the subtree being remove the more likely it is that some good qualities are lost. Hence, we would expect the evolution to bias towards those individuals that has been produced from a replacement procedure that removed as little as possible from the parent.

that when lowering the frequency of mutation, you narrow your search and will not explore new solutions as often. And when lowering the frequency of crossover, the good solutions in the populations is not as aggressively exploited. In effect, the evolutionary search is slowed down which is also recognized by Blickle and Thiele (1994) and De Jong et al. (2001) amongst others.

Another argument for controlling the size of individuals is presented by Rosca (1996), where strategies for a simple Pac-Man game are evolved. Here, the smallest evolved programs were also the most general ones. With generality we understand the performance of a program when presented a new set of test cases that have not been included in the training data used in the evolution.

Gathercole and Ross (1996) describes the interaction between the crossover operator and the absolute tree size. They argue that in populations were trees have reached a considerable size, the discovery of good subtrees and the distribution of these subtrees, mainly occurs in the lower levels of the trees, that is, near the leafs. When the size of the trees grow, the probability of selecting a crossover point near the root decreases dramatically, and hence the upper part of the trees converge, while diversity is maintained in the lower parts of the trees. So if a suboptimal solution gains foothold in the early generations, this solution can be dragged on in the upper levels of the trees fairly immune to further improvement. It is obvious that in a situation like this, there exists an upper bound on the quality of solutions that can be expected to evolved from the current population.

Lastly, in some systems solutions with a high proportion of introns require more evaluation time. For instance, Brameier and Banzhaf (2001) use a Linear Genetic Programming system to evolve a classification program to

local optima. We believe that it is essential to the quality of the evolved solutions of a GP system applied to real time domains, to avoid convergence for as long as possible. So, when considering the quality of the system as a whole, it is highly dependent on the systems ability to avoid premature convergence.

We therefor state the goals of this project to be:

1. Address the problem of bloat or code growth. Extensions to the system for gaining better control of the size of individuals should be designed and implemented. The effect on the quality of the evolved solutions should be investigated.

2. Address the problem of maintaining diversity. New initiatives different from the island principle should be designed and implemented. The effect on the quality of the evolved solutions should be investigated.

The performance of the search should always stay in focus, and we should not give up performance in favor of one of the two above mentioned goals.

We propose the following extensions:

1. Executed Path Guided Operators. New crossover and mutation operators that only allows cut-points/mutation-points within the executed path are proposed. The crossover operator was previously proposed by Blickle and Thiele (1994), and it is inspired by the cause of growth listed as item 1 in section 5.2.4 by not allowing neutral operators. That is, individuals can not be immune to the destructiveness of the genetic operators by growing in size.

ferent extensions previously proposed. Along with each extention, related work is presented. Chapter 6 describes some modifications to the system described in chapter 4. Chapter 7 present the Size Ranking method. Chapter 8 presents the Diversity Ranking method. Chapter 9 presents our E-CFG method, and chapter 10 presents our EPGO method.

Secondly, we skipped the assessment step of our old algorithm, refer to section 4.3.2 for a description of this function. In the assessment step you basically decide where you want your search to begin, and this might be either considered good or bad. In our case we assessed the strategies according to some prior knowledge about the domain. In other words, we impose an artificial distribution on the initial population. In this project however, we want to have a system as basic as possible, and we therefore wants to start with a random initial population. This approach seems to be standard throughout the literature.

Thirdly, we have modified the language described in section 4.2, also as a result of the observations made in Holm and Nielsen (2002). It is a well known issue (e.g. see Koza (1992)), that atomic constructs with very complex semantics [2] can cause convergence by dominating good solutions and whole populations. And indeed we saw this effect with constructs like `relative-movement` (see section 5.2.1). Therefore, this construct has been removed from the set of functions. In general, we dislike the idea to mix high-level and low-level constructs in the language, and by removing `relative-movement` we feared that constructs like `circle-strafe-left` and `circle-strafe-right` could be just as dominant in the population as what we experienced with `relative-movement`, and therefore the everlasting risk of converging prematurely is strengthened. Hence, we have removed `circle-strafe-left` and `circle-strafe-right` from the language. This

[1] Individuals that can not be paired with another individual in the current round is typically given a so called "walk-over", meaning that they proceed to the next round without competing.

[2] Often referred to as high-level functions and terminals.

which is a perfect condition for successful use of the `relative-movement` function. Also, a single `shoot` node is a highly effective solution if you are facing your enemy most of the time. So in an attempt to constrain the scope of some of the very primitive solutions, the spawning-points are now scattered over the arena. Also, we wanted to make the map more dynamic, and the symmetric spawning points in the old map makes it somewhat more static, and might not reward good strategies for finding your enemy, as he/she is usually right in front of you.

2. In the old map, items were evenly distributed throughout the whole arena, making it very easy for the bots to pick up items. In fact, the bots would typically run into several items just by crossing the arena. So, in an attempt to avoid this situation, and making the environment more challenging to the bots, we enlarged the arena and dispersed the placement of items.

Before going into a detailed discussion about the effect of evolution on the size of individuals, it is useful to distinguish between effective size and absolute size of a program. Nordin and Banzhaf (1995) define the effective size (S_{eff}) of a program to be the size of the non-intron part, and the absolute size (S_{abs}) is of course the total size of the program, including all introns. In accordance with the definitions of introns given in section 5.2, we can see the effective part of a program as being the executed part of that program. In the following we will illustrate the fact that the absolute size of an individual can have a hand in the survival rate of that individual.

As described in chapter 5, neither standard crossover nor mutation can (by definition) change the raw fitness of a program when applied within a global intron block. On the other hand, crossover or mutation within any non-intron block of code will almost certainly change the performance or behavior of that program at the risk of decreasing raw fitness.

It is possible to estimate the evolution of size by calculating the expected number of copies of a given effective part of a program in future generations. In generations later than the initial, this estimate is composed of two parts. Firstly, code blocks can survive across generations through selection and genetic operations. Secondly, entirely new instances of equivalent blocks can emerge from genetic operations. We will focus on the survival of existing blocks and for now we will disregard new instances of code blocks emerging from recombination and mutation.

We have a probability of crossover and mutation at the individual level of p_c and p_m respectively. $S_{eff}(i)$ and $S_{abs}(i)$ describes the effective size and absolute size of individual i. We can formulate an upper bound of the probability of potential destruction of the effective code block of individual i,

least a low value for $\frac{S_{eff}}{S_{abs}}$. Amongst others, these facts were recognized by Nordin and Banzhaf (1995) and Rosca (1996).

7.2 Previous Efforts

A lot of different approaches to gain more control over the size and complexity of the evolved solutions have been investigated. One crude or primitive approach is simply to constrain solutions to be smaller than some explicitly defined maximum size. This technique is most often rejected as too greedy and static, especially if this size constraint is fixed by plain guessing. However, if you are already in possession of an acceptable solution, and just want to explore equally good (possibly better) but less complex (i.e. shorter) solutions, this approach is intuitively optimal. It could easily be combined with one or more of the methods described in the following sections.

7.2.1 Primitive Parsimony Pressure

A common (and a bit more sophisticated than the above mentioned) approach is to explicitly introduce parsimony pressure to the evaluation function. In effect, this means to make the effective fitness value dependent on the absolute size of the individual. We redefine P_{sel} and modify equation (7.2) as:

$$P'_{sel}(i) = P_{sel} - \sigma S_{abs}(i) \quad (7.4)$$

$$P_{sur}(i) = P'_{sel}(i) - P_{des}(i). \quad (7.5)$$

[1] For instance, two identical subtrees can be exchanged between the parents, resulting in offspring identical to the parents.

In equation (7.6), $\Delta S(g)$ is recursively defined by equation (7.7).

$$\Delta S(g) = \begin{cases} 0 & \text{if } g = 0, \\ \frac{1}{2} S_{abs}(best, g) - S_{abs}(best, g-1) + \Delta S(g-1) & \text{otherwise.} \end{cases} \tag{7.7}$$

Now the Occam factor is updated between generations according to the scheme described by equation (7.8).

$$\sigma(g) = \begin{cases} \frac{1}{N^2} \frac{E(best, g-1)}{\hat{S}_{abs}(best, g)} & \text{if } E(best, g-1) > \epsilon \\ \frac{1}{N^2} \frac{1}{E(best, g-1) \hat{S}_{abs}(best, g)} & \text{otherwise} \end{cases} \tag{7.8}$$

In equation (7.8) N is the size of the training set, and ϵ describes the maximum training error allowed for the final solution. Finally, before engaging in the selection process, σ in equation (7.4) is replaced by the new definition $\sigma(g)$ from equation (7.8). Amongst others Blickle (1996) shows good results in solving two symbolic regression problems, one continuous and one discrete.

7.2.3 Explicitly Defining Introns

Nordin et al. (1995) propose a system in which a special language constructs (an Explicitly Defined Intron (EDI)) is given the characteristic properties of introns. An EDI can be attached to any edge between two 'normal' neighbor nodes in the program, and it have no effect on the execution of the program.

[2] Named after William of Occam and his principle of simplicity (Occam's Razor): "Given a choice between two explanations, choose the simplest – the explanation which requires the fewest assumptions."

individuals within each fitness class according to their absolute size. In this way, we never run the risk of moving a individual from one fitness class to another as a result of applying the pressure. The fact that the smallest change in fitness is balanced against the biggest change in size, is in accordance with the guidelines that put up by Nordin and Banzhaf (1995).

Our approach described in this chapter, will be referred to in the following as SR.

solutions. In such domains, it is of great importance that diversity is kept under control and not permitted to drop below some threshold, which would drive the exploration of new solutions to a halt.

Especially, in our domain (described in section 2) we expect a lot of sub-optimal solutions to exist. As a result, we are not interested in damping down exploration prematurely, but rather we would prefer to explore new solutions. However, when considering our use of a competitive tournament based fitness function (see sections 4.3.2), the fitness function itself displays a very dynamic behavior, as it is dependent on the population of the current generation. As mentioned by Angeline and Pollack (1994), this kind of tournament fitness will naturally discourage convergence in most situations. The reason is in the way fitness is assigned to individuals. As illustrated in figure 7.1, individuals are assigned a fitness value according to the level of the tournament reached by that individual, in effect meaning that a lot of individuals will be assigned the same fitness. So even in the fitness distribution, we have inherently a bias towards low pressure. And as we use a selection method with low pressure [1], we have a very low selection pressure indeed. A low selection pressure inherently promotes diversity, as no superindividual will be allowed to reproduce aggressively. This being said, it might seem strange that we still want to consider methods for maintaining diversity in the population. The answer is, that a competitive fitness function only leaves room for diversity to exist, but does not directly disperse the population and thereby forcing individuals to explore new areas of the search space, and this is what we would like to do.

[1] Tournament selection with tournament size 2, see section 4.3.2.

Jong et al. (2001), where the distance between two trees is calculated by summing the number of identical nodes with corresponding positions when the two trees are overlaid. De Jong et al. (2001) normalizes the distance between two trees by division of the size of the smaller of the two. Instead of performing this kind of full tree comparison, we only consider the biggest garbage free subtrees when performing comparison. By garbage free we mean that all nodes in the full subtree have been executed. As an example figure 8.1(a), 8.1(b), 8.2(a) and 8.2(b) depict a comparison of two trees using our scheme, and two subtree matches are encircled by the dashed line. So, for all pairs of individuals in the population, we calculate the maximum common subtree, and for each individual we calculate an average of these values. Like in the approach described in section 7.3, we use this average measure to sort individuals within specific fitness classes, so that individuals with a low average common size are promoted and individuals with a high value are punished.

The motivation for only considering the biggest garbage free subtree is first of all the fact that we do not want to punish trees with identical garbage, and promote trees with different garbage. If, at some point the garbage is put into use, the garbage will not continue to be garbage, and trees with large common sizes will now be punished.

8.1.1 Measuring Common Size

The algorithm for determining the size of the biggest matching subtree of any two trees, is described by the algorithms `compare` and `subTreeMatch` depicted in tables 8.1 and 8.2 respectively. Basically, `compare` takes two trees as arguments, performs a breadth first scan through both trees and contin-

minimize this complexity, by only calling `subTreeMatch` when it is possible to find a bigger match, that is both arguments of `subTreeMatch` must have a size greater than the value of variable `max_match`. This improvement does not affect the worst case scenario, when two completely different trees are given as arguments to `compare`. The complexity of `subTreeMatch` is equal to the size of the smaller of the two trees. This can be realized by considering the worst case scenario, when two identical trees are given as arguments for `subTreeMatch`. In this case, the amount of comparisons needed is equal to the number of nodes in the tree. The same is the case, when trees only differ in leaf nodes. If we assume that most often the trees compared are of approximately equal size, we get a complexity of `compare` of $O(n^2)$, n being the average size of the two trees given as arguments. As earlier mentioned, the worst case scenario for `compare` is two completely different trees, which is the best case scenario for `subTreeMatch`, yielding a constant complexity as the test in line 10 of table 8.2 fails in the first iteration.

It is obviously of great concern that our algorithm for `compare` has a complexity of $O(n^2)$, but as we use only the biggest garbage free subtrees when comparing two trees, we expect the size of these garbage free subtrees to stay approximately constant on a relative low value. As an illustration, some preliminary experiments have shown that garbage free subtrees on average do not grow to sizes of more than 10 to 20 nodes, which yields a total of max $20^2 = 400$ comparisons. Compared to the evaluation of a generation of 255 individuals in our real-time domain ($\approx 15 \frac{\text{seconds}}{\text{individual}}$) 400 comparisons is negligible. Of course, the 15 seconds primary evaluation is done in a cup-tournaments based fashion (described in section 3.2.4), while all individuals need to be compared with all others once, yielding a total of 32385 comparisons on the individual level. However, experiments still show that this is

```
23:         }
24:     }
25:     return max_match;
26: }
```

Table 8.1: Algorithm for finding the largest subtree match within to subtrees.

not a performance bottleneck.

In the following chapters, the approach described in this chapter will be referred to as DR

```
10:       if(node1 == node2){
11:          match = match + 1;
12:          for all children x of node1{
13:             stack1.push(x);
14:          }
15:          for all children x of node2{
16:             stack2.push(x);
17:          }
18:       }
19:    }
20:    return match;
21: }
```

Table 8.2: Algorithm for matching subtrees.

three runs from Holm and Nielsen (2002) we found various useless syntactic structures which could be eliminated by altering the CFG, and hopefully not the solution space. Another reason for enhancing the CFG was the removal of the high level functions. The high level functions were part of all the best individuals from the three runs in Holm and Nielsen (2002). By removing these we expect that it will become more difficult to reach solutions of the same quality. Yet by adding rules to the CFG we can push the evolution to evolve sensible building blocks of the low level functions. In the next section we will look closer into the new rules added to the CFG.

9.2 Our Approach

The E-CFG is constructed by altering some of the old rules from the initial CFG and adding a number of new rules. The E-CFG should narrow the search space further, still it should not do it by removing satisfying solutions. It has been our goal to design the E-CFG with the removal of unsound syntactic structures in mind, e.g. structures which are obsolete because they only differ compared to other structures as genotype but not as phenotype. In figures 9.1(a) and 9.1(b) we can see an example of two trees having identical phenotype but differing genotype.

[1] Notice that when the mutation operator was applied the resulting tree was not necessarily recognized by the CFG, only the subtree inserted.

9.2.1 Context Free Grammars

A context free grammar is a four-tuple

$$(\mathcal{N}, \sum, \mathcal{P}, \mathcal{S}),$$

where $\mathcal{N}$ is the non-terminal alphabet, $\sum$ is the terminal alphabet, $\mathcal{P}$ is the set of productions and $\mathcal{S}$ is the start symbol. The productions are of the form

$$A \rightarrow b,$$

where $A \in \mathcal{N}, b \in \sum \bigcup \mathcal{N}^*$. Productions of the form

$$A \rightarrow b,$$

$$A \rightarrow c,$$

can be expressed as

$$A \rightarrow b \mid c.$$

9.2.2 The Enhancements

The alphabet of the E-CFG, representing the functions and terminals of our language, is described in table 9.1.

We will now elucidate a number of sets, which will be used in the grammar.

$$F_{set} = \{A..N\} \tag{9.1}$$
$$I_{set} = \{D..N\} \tag{9.2}$$
$$a_{set} = \{h..o\} \tag{9.3}$$
$$s_{set} = \{a..g\} \tag{9.4}$$

The "function" set (F_{set}) consists of all the functions from our function and terminal set. The "if" set (I_{set}) consists of all the functions being `if` constructions except `if-less-than` which differ in that it uses two of its arguments for evaluation. The "action" set (a_{set}) consists of all terminals which causes some kind of bot action when executed. The "sensor" set (s_{set}) consists of all terminals representing game information and the constant terminal.

We will now take a look on the reasoning behind the different rules in the E-CFG.

The start rule

$$S \rightarrow F_{set} \tag{9.5}$$

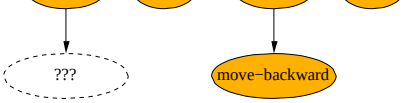
makes sure that trees at least consist of three nodes, as all constructs in F_{set} require at least 2 children with the exception of the `wait` construct, that only requires 1 child. However, the rule for `wait` (see rule (9.10)) preserves the minimum tree size of trees with a `wait` node as root. Since trees with less than three nodes can not represent satisfying solutions and we will avoid that an individual which only consists of the `shoot` node can get a good

only allows a `prog-2` function to have another `prog-2` function or one of the action terminals as its arguments. This rule is implemented with the original purpose of introducing the `prog-2` function in mind, namely to be able to execute a sequence of actions. The syntax of this rule makes it possible to execute from two to an infinite sequence of actions. We should also note that this rule does not reduces the set of possible solutions, but only reduces semantic duplicates. An example of two trees which are semantically equivalent can be seen in figure 9.2(a) and figure 9.2(b). The tree in figure 9.2(b) was generated using the E-CFG and the tree in figure 9.2(a) was not.

The following `if-less-than` rules

$$B \rightarrow B'\ B''\ B'' \tag{9.7}$$
$$B' \rightarrow s_{set}\ p \mid p\ s_{set} \mid s_{set}\ s_{set} \tag{9.8}$$
$$B'' \rightarrow a_{set} \mid F_{set} \tag{9.9}$$

helps to generate conditionals which should have a purpose, that is, evaluate sensor information and execute the appropriate branch based on this information. The analysis of the trees from the three runs in Holm and Nielsen (2002) showed that the `if-less-than` function in some cases served as a `prog-3` function or always executed the same branch, or both possible executable branches had sensor nodes as children. Illustrations of these three types of undesirable tree structures can be seen in figures 9.3(a), 9.3(b) and 9.3(c). When using the new E-CFG's `if-less-than` rules, we make sure that the two children being evaluated never are two constants, in this way we avoids that the same branch always is executed (with one exception; when `const` takes a minimum or maximum value).



(a) (b)

Figure 9.4: During tree generation a `wait` *function decides its child by using its parent's grammar rule. In (b) the* `wait` *node has decided to be a action terminal according to the* `prog-2` *grammar rule.*

The `wait` rule

$$C \rightarrow \textit{parent rule} \tag{9.10}$$

necessitates a little explanation. To limit the number of listed rules for this function we have made a notational shortcut. What we mean by *parent rule* is that the allowed child to wait is determined by looking at the children allowed for `wait`'s parent. An example of this rule in use can be found in figures 9.4(a) and 9.4(b). In this case the rule for `prog-2` will be used since this type of node is parent to `wait`, so this means that `waits` child can be either a `prog-2` node or a terminal from the A_{set} as chosen in figure 9.4(b). The I_{set} rule, defined in terms of the B'' variable previously defined by rule (9.9),

$$I_{set} \rightarrow B''\ B'', \tag{9.11}$$

allows all functions and terminals except for sensors as arguments.

CFG to generate our initial population and the subtrees used in mutation. In addition we will use the E-CFG to control the crossover operator, so that the offspring generated will recognize the rules imposed by the grammar.

upon our observations in Holm and Nilsen (2002) and the analysis in chapter 5. As previously mentioned and documented in Soule et al. (1996), without any constraint mechanism the programs generated by GP will grow indefinitely regardless of whether or not the growth acts to improve the programs' solutions.

The fact that the amount of non-functional code in our programs grows as evolution progresses raises some concerns. Contrary to most papers touching the problem with tree growth, the domain in focus is highly complex, dynamic and noisy. With this in mind, one have to consider that results and methods in these papers have to be carefully evaluated before applying any of the theory to accomplish our task.

In a domain like UT it can be expected to evolve a considerable number generations in a GP run, before an adequate solution is found. When looking at the time consuming fitness function used in our domain it should be apparent that it is relevant to optimize the exploration and exploitation properties provided by the genetic operators. When the amount of non-functional code in the population grows exponentially and we can expect to evaluate a high number of individuals it seems logical to aim for individuals only differing in size but not semantically are only evaluated once. Before going into details with the approach taken to deal with this issue let us look at similar work done within the area.

10.2 Similar work

The principle of EPGO, namely to let execution paths guide the choice of GP crossover and mutation location is yet rather unexplored. As far as is known,

in the trees. Specifically Blickle and Thiele (1994) give a more formal description of the crossover phase and redundancy phenomena in GP. In the following section some of the definitions will be summarized.

10.2.1 Crossover and Redundancy

Definition 10.2.1 *The edge A in tree T is called redundant if for all values of the leaves (terminals) the function represented by the tree T is independent of the subtree located at edge A.*

Note:

- If the edge A is redundant if follows immediately that all edges in the subtree located at edge A are redundant, too.
- The redundancy of an edge A in general depends on the context.
- All nodes located at redundant edges are *redundant nodes*.
- The non-redundant nodes are also called "atomic" by Tackett (1994).

Definition 10.2.2 *The proportion of redundant edges in a tree T is given by*

$$p_r(T) = \frac{number\ of\ redundant\ edges\ in\ T}{number\ of\ all\ edges\ in\ T}$$

Definition 10.2.3 *The redundancy class T^* is the set of all trees T that only differ from subtrees at redundant edges, i.e. for any two trees $T_1, T_2 \in T^*$,*

potential local optima decreases with time.

In a highly multimodal and complex problem domain the possibility of exploring several local maxima is evident. Therefore we find the just mentioned property highly undesirable in our domain. Blickle and Thiele (1994) suggest a method to control the redundancy and present results which demonstrate its applicability.

10.2.2 The Marking Method

The idea of the "marking" operator described in Blickle and Thiele (1994) is to mark all nodes that are traversed (or executed) during evaluation of the fitness function in the following way:

- First before evaluation the marking flags of all nodes are reset.
- Then if a node is executed during the fitness calculation the corresponding flag is set.
- At last after the calculating the fitness function, only at redundant nodes the flags are still cleared. The crossover is then restricted to edges with the flag set.

The method is applied on three problems taken from Koza (1992). For the 6-multiplexer problem the performance was almost doubled, for the *truck backer upper* problem an improvement in convergence of 20 % was measured and for the ant problem almost no improvement was measured.

part of the tree has been executed.

could be executed in a subsequent evaluation.

Figure 10.1: The colored subtree has still not been executed. This latent subtree could be executed in a subsequent evaluation.

10.3 Our Approach

We have chosen to adopt the method described in Blickle and Thiele (1994) and use it for mutation as well as crossover. The method seems to fit nicely for our problem domain. Alternate methods which try to remove redundant code could cause a problem in our domain. Let us take a look at an example. In figure 10.1(a) we see an initial generated tree. We could apply heuristics to remove possible redundant nodes, but we risk to remove sound code and we also risk to alter the initial diverse distribution of functions and terminals.

In figure 10.1(b) we see the executed nodes in a tree after an evaluation. These nodes will be allowed as cutpoints in the following crossover phase using EPGC.

In figure 10.1(a) we see another executed path in the same tree for a succeeding evaluation, which could be caused by another opponent using a different strategy. This time only these nodes will be allowed as cutpoints. Now it is natural to ask: Why don't we also allow the previously executed nodes as cutpoints? Imagine an evolution running for several generations, as we observed in Holm and Nielsen (2002) the solutions of the population gradually adapt to different strategies (as a result we see different executed paths), some of the strategies encountered early in the evolution are primitive and

Programming algorithm, extended with the different methods designed in the previous part. The results are presented in chapter 11, and finally we conclude upon these in chapter 12.

11.1 Parameters of the Evolution

We have used the same set of parameters for all runs, these are listed in table 11.1. As previously mentioned, we have discontinued the use of the island principle and now only maintains a single population without subpopulations and with a total size of 256. The crossover frequency P_{cross} is set to 0.9, the default value used by Koza (1992). Unlike default Koza-parameters [1], we use mutation at a frequency of 0.1. As we have a very low selection pressure presented by our combination of the cup-tournament-based fitness function and tournament selection with tournament size 2, and as we do not use reproduction as defined by Koza (1992), we use elitism that copies the 10 most fit individuals unchanged to the next generation. We have reconfigured the UT-server to run at double speed, so the evaluation time of 30 seconds is in real-time only 15 seconds. The minimum and maximum initial individual depth constraints the individuals in the initial population (either random or custom generated) on depth. The minimum and maximum mutation depth, constraints the subtrees (either random or custom generated) inserted by our mutation operator on depth.

11.2 Results

In this section we will first of all take a look at how the five runs have evolved with regard to tree sizes. The figures will depict average absolute size, average effective size and absolute size of winner. A short description of these three measures can be found below (see section 7.1 for a detailed

[1] Default mutation frequency used by Koza (1992) is 0.0.

Average effective size: The effective size of a tree is the number of nodes executed in that tree. The effective size of a tree is based on the first match every individual play in the tournament based competitive fitness function (note: not the accumulated nodes executed during a tournament). The average is calculated from the entire population of a generation.

Secondly, we describe the results of a benchmark test performed on individuals from all generations of the five different runs. An All-Star team was summoned to act as the benchmark test environment. The best individual from the 25th, 50th and 75th generation of all five runs were drafted to play on the All-Star team, as we hoped this would compose a diverse and broad spectrum of different strategies. The tests were performed by pitting the winners from each generation against all 15 members of the All-Star team, one at a time, in a 30 seconds match in the well known arena. For all winners of all generations of the different runs, the following values were logged:

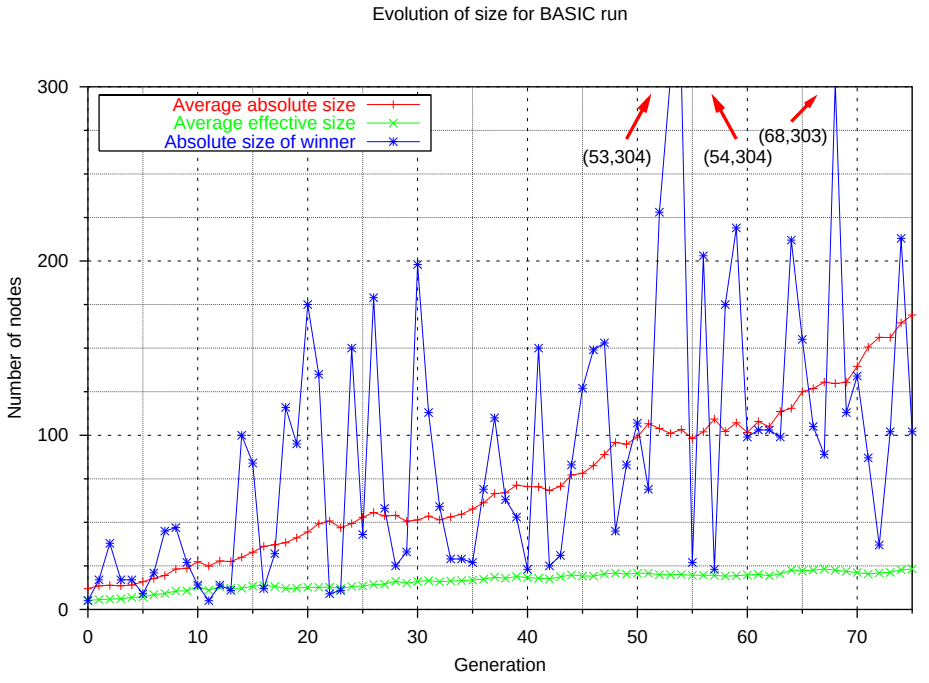
Points An accumulation of points received in the 15 matches. 1 point is given to the individual with the most frags, -1 to the other, or 0 to both individuals if there is a tie on amount of frags. We have departed from the interpretation of results used in the evaluation function for the cup-based tournament fitness function [2]. The behavior we wanted to evolve in the bots, is the ability to collect more frags than the enemy. Therefore, it does not make sense to be concerned with amount of shots fired or maximum period of no movement, when benchmark testing the solutions.

[2] See section 4.3.2 for further information.

constant to be caused by the noise. At first, it seems difficult to come up with a straightforward explanation about this phenomenon, so we will have to study the evolution of the winning trees in detail. It seems reasonable to seek for a relation between the performance results and the size results, especially because we also observed fluctuations in the performance results. In figure 11.1(a) we see that after generation 40 the performance stabilizes and the small fluctuations in the later generations are probably just a result of the inherent noise in the domain. This observation rule out an intermediate connection between the size fluctuations and the evolution of new solutions in the population. To search for another explanation we have extracted detailed information on the sizes of individuals from the entire population in consecutive generations. In figures 11.2(a-e) the data from generation 52, 53, 54, 55 and 56 can be seen.

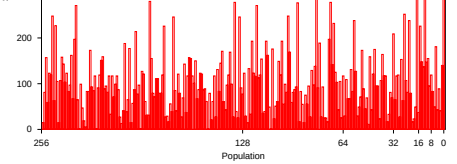
It is apparent from these graphs that not only the winners of the different generations fluctuates in size but the entire generation of individuals do. This supports us with the necessary information to form a hypothesis. Assuming that the building block hypothesis suggested by Goldberg (1989) is true, then when we start an GP evolution small building blocks will start to form as evolution progresses. Then these building blocks get combined to constitute more complex solutions, as in figure 11.3(a) illustrating an abstraction of a good solution. Some of these solutions will be better than others and hence gets selected more often, resulting in multiple offspring based on this building block. In figure 11.3(b) such two blocks are depicted where the large triangle is the building block and the small triangles illustrates two subtrees being introns. When evolution progresses and these trees are chosen for crossover as depicted in figure 11.3(b) and 11.3(c), some of the resulting offspring will be a composition of the same building blocks on top of each other. If

(a) Points

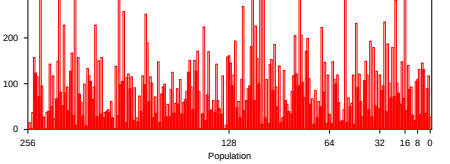


(b) Size

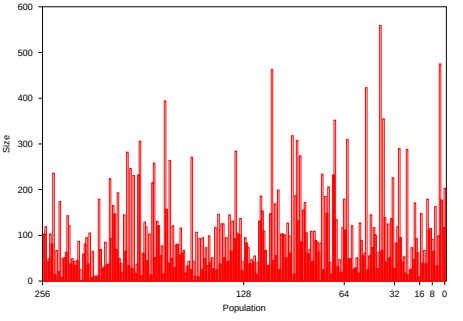
Figure 11.1: Performance(a) and size(b) graphs for the BASIC run.



(c) Generation 54, with a winner of size 304.



(d) Generation 55, with a winner of size 27.



(e) Generation 56, with a winner of size 203.

Figure 11.2: Evolution of size of individuals from generation 52 through 56.

later generation individuals selected for crossover (d), and the offspring produced by choosing the black dots as cut-points (e).

this happens through consecutive generations we can imagine individuals like the two depicted in 11.3(d). When such two are chosen for crossover the resulting offspring can look like its depicted in figure 11.3(e). As can be seen the smaller tree (figure 11.3(e)) will then still consist of the good building block, hence perform as well as the larger. Therefore we will see the huge fluctuations. When studying some of the parse trees in detail our hypothesis is supported. In figure 11.2 the winning tree from generation 55 can be observed. This tree only has a size of 27 nodes and as seen in figure 11.1 follows generation 54 which had a winning tree consisting of 304 nodes. We have identified at least two common building blocks in the winning tree from generation 55, these can be found in table 11.3. These building blocks seems to be the basic foundation for a good solution in the last half of the evolution and are found multiple times in the large trees. Taking this evidence in consideration we feel the hypothesis is further backed up.

Benchmark Result

We have now argued for the heavy fluctuation in size, and when investigating the graph in figures 11.1(a), we notice first of all that heavy fluctuation in performance is also present. For instance we notice the sudden peak at the 9th generation, and the equally sudden drop in performance of the 10th generation. In table 11.4 the 9th and 10th winner is depicted. The 10th generation is clearly the more primitive of the two. As the `shoot` node always returns 0, the `if-less-than` will always evaluate to `false`, and hence the

```
(if-enemy-in-sight
    (face-enemy)
    (turn-left))
```

```
(if-enemy-move-away
    (move-forward)
    (shoot))
```

Table 11.3: Two common building blocks extracted from the winner of generation 55 of the E-CFG run.

behavior is only composed of shooting and running backwards. The 9th generation winner however, will both try to face the enemy and shoot no matter the current states of sensors. The reason such a primitive strategy as that of the 10th generation can make it to the top, while more sophisticated strategies (like the winner from the 9th generation) exist in population is worth investigating. When looking at the 10th generation in more detail, we found that the winner from the 10th generation actually was pitted against an individual equivalent to the 9th generation winner in the semi-finals, and of course won. This confirms, that the low selection pressure that combined with a relatively noisy environment does not allow single good solutions to spread rapidly throughout the population. Apart from the fluctuation, the general trend of the graph is interesting. The trend in points collected is already above 0 around the 25th generation, which is very good indeed. In the following sections we will compare the performance of the other runs with this result.

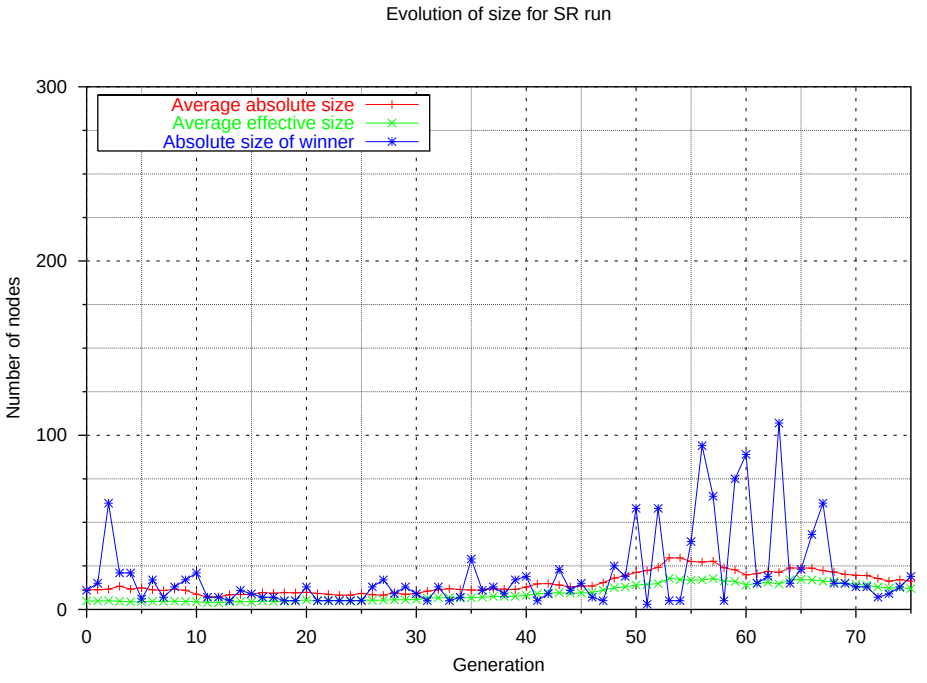
Table 11.4: The best individual from the 9th (the leftmost) and 10th (the rightmost) generation

The Undesirable Flora of Size

The average size of the population in the BASIC run was expected to grow rapidly and as can be seen in figure 11.1(b) indeed it did. The average effective size though is kept under a size of 25 during the entire run and the growth of the effective size seems to happen with an insignificant linear rate. The thing to note here is that growth of absolute size happens with an approximately linear rate through the entire run. This raises the probability of premature convergence and as can be seen in figure 11.1(a) the performance begin to converge after generation 30. Now, this it not necessarily a bad thing since the convergence happens close to the maximum score, but it surely brake further evolution of better solutions. Hence if we had increased the number of generations per run it is unlikely that further improvement would happen.

It would be nice if we could limit the growth of the trees, while still keeping the performance of the BASIC run. In the next three runs (SR, DR and E-CFG) described, this growth is limited and the DR method succeeds in matching the performance of the BASIC run. The last method (EPGO) succeeds in limiting the effective part of the trees used for crossover and mutation while still matching the performance of the BASIC run.

(a) Points



(b) Size

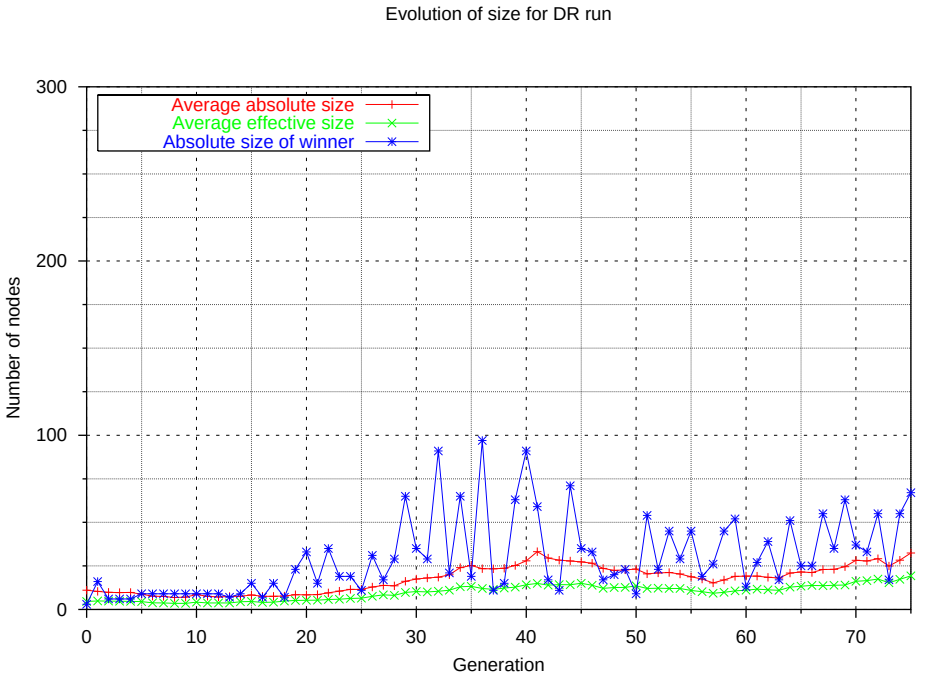
Figure 11.4: Performance(a) and size(b) graphs for the SR run.

pressure) when constructing the building blocks. To put it another way, good building blocks are not allowed enough freedom to evolve and accumulated in larger and larger trees, as we assume is the case for the BASIC run. The next method succeeds in delivering the latitude necessary for steady evolution of better solutions while still maintaining a population with a trimmed size.

11.2.3 Diversity Ranking

Although the performance graph of the DR run depicted in figure 11.5(a) is not as steep as that of the BASIC run depicted in figure 11.1(a), the tendency is clearly ascending. The fluctuation in points is more pronounced than that of the BASIC run. If we take a look at the evolution of size, it is interesting to see that many of the peaks in performance (at generations 35, 38, 42, 47, 48, 49 and 52) is matched by off-peaks in size. This is in keeping with results previously reported by Rosca (1996), were short and compact solutions are found to be more general, even though other explanations (including pure coincidence) to this phenomenon could be just as valid. One obvious question is then why the SR run performed so poorly compared to the DR run? The answer must be that in SR we just promote solutions with a small absolute size, and this makes the population converge against the same short solution. The DR method however, accomplishes the damping in bloat more indirectly, which will be elaborated in the following section.

(a) Points



(b) Size

Figure 11.5: Performance(a) and size(b) graphs for the DR run.

11.2.4 Enhanced Context Free Grammar

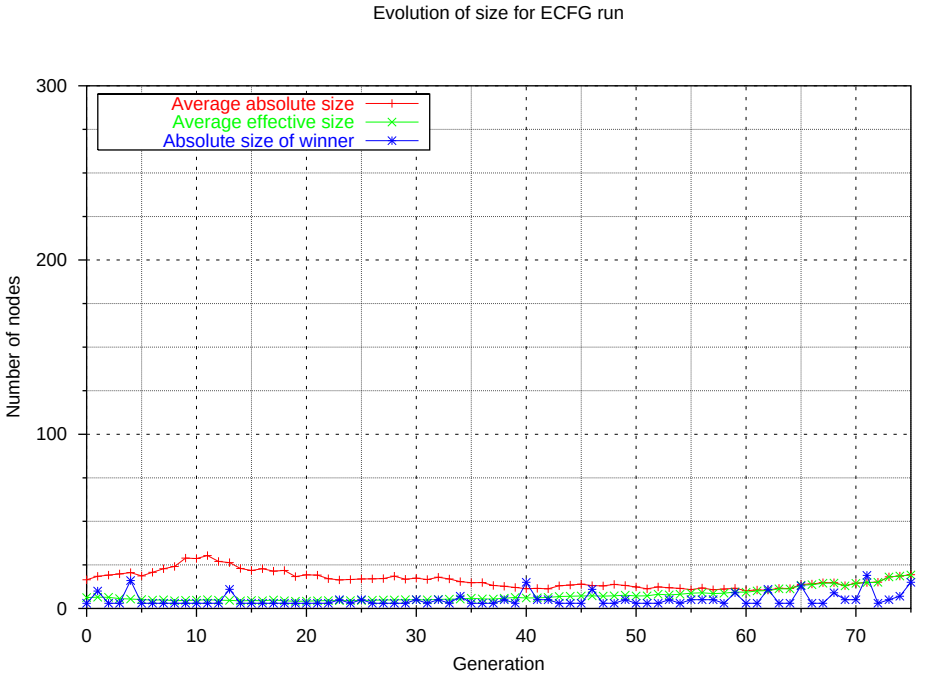
The performance test of the E-CFG as depicted in figure 11.6(a) is not good when compared to the other tests. One reason for this result could be that the rules that make up our E-CFG in fact narrows the space of possible solutions to tight. Another reason could be, that the semantical meaning of our language constructs is not as well defined as initially assumed. [3] If this is the case, we have built our E-CFG on a incorrect basis, and hence our E-CFG will act as a poor guide for the evolutionary process. Without enforcing the E-CFG upon the evolution, the true semantical meaning of the functions and terminals would emerge from the compositions of the fit individuals.

What Went Wrong

We have done extensive testing on every single function and terminal and believe that the missing performance of the E-CFG should be located elsewhere. When looking at figure 11.7(a-b) showing the distribution of functions and terminals, it seems odd that the frequency of `prog-2` and `shoot` increases so rapidly. Now, when investigating the different trees from the run, a pattern is forming. It appears that a suboptimal solution has emerged in an early generation, the core of this solution can be found leftmost in table 11.5 and

[3] An analogy is, in a real world robot, a `move-forward` command may not be well defined for all possible environments, and may very well be dependent on the friction of the surface.

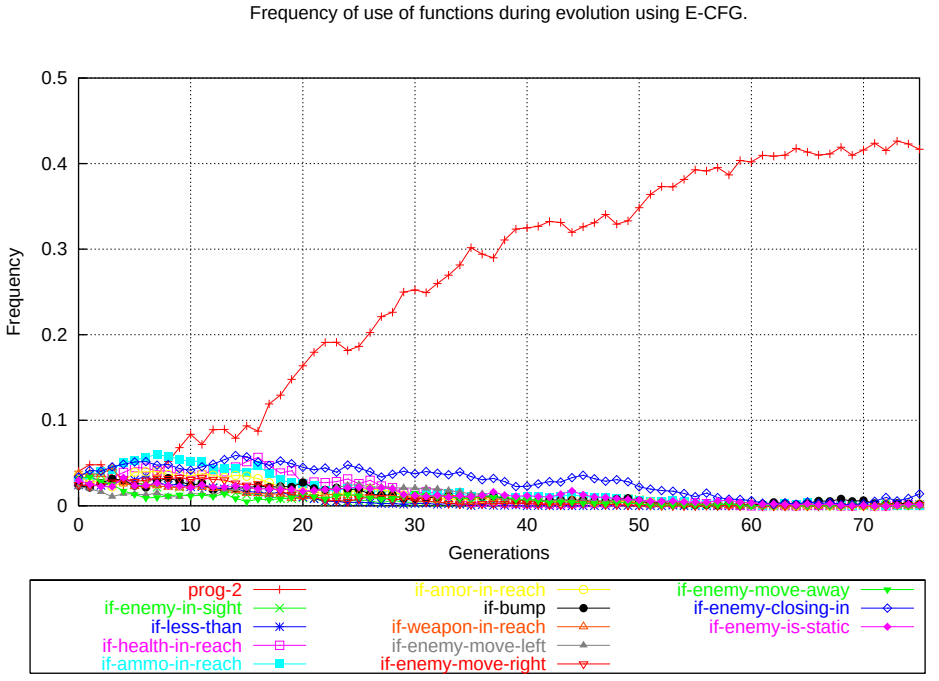
(a) Points



(b) Size

Figure 11.6: Performance(a) and size(b) graphs for the E-CFG run.

(a) Terminals



(b) Functions

Figure 11.7: Frequency of node usage during the E-CFG run.

in table 11.5. The impact of this general new solution can also be observed in figure 11.7(a), showing increased frequencies of the terminals `face-enemy` and `move-backward`.

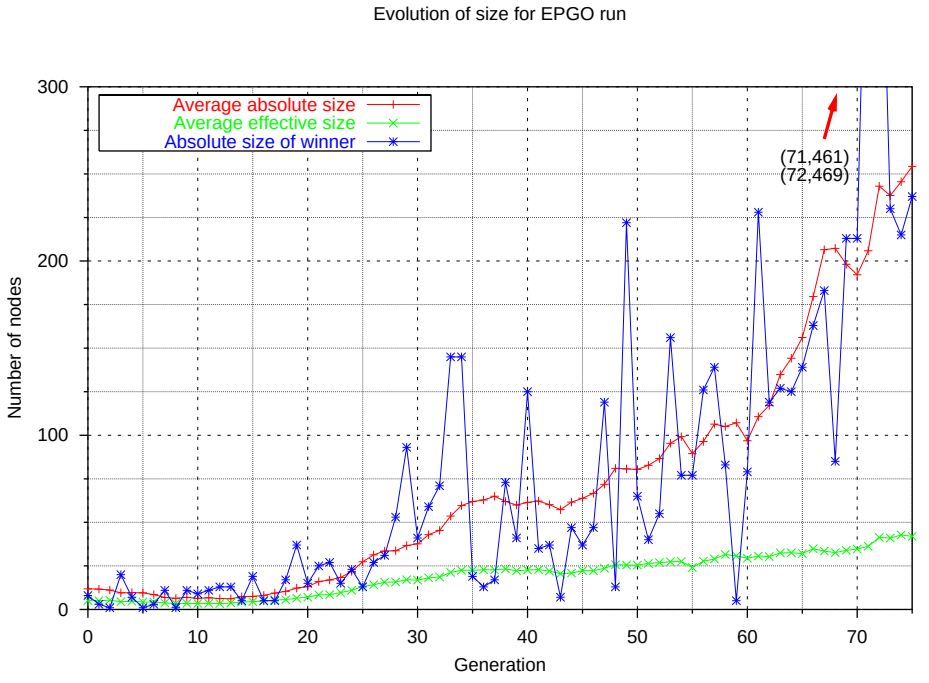
Should the CFG be Discarded

As a concluding remark to the E-CFG results, we must say that it has proven very difficult to define a CFG that generates good solutions. One should acknowledge, that imposing a CFG on the construction of solutions is equivalent to imposing a new distribution of the usage of nodes. That is, when the language is changed from being type-less (in which a totally random composition of individuals is valid) to a language constrained by a grammar, a dependency between the nodes is imposed. And hence, if some nodes gain dominance in the population, some other nodes might be nearly impossible to introduce into this population, due to the inter-node-dependency inferred by the grammar. Therefor, the genetic operators are changed in a way that might not be completely clear, when designing the CFG. When this is said, we still believe a CFG can be powerful in a complex domain, but it should be guided in some way e.g. by a heuristic. In chapter 12.4 we have proposed this as future work.

11.2.5 Executed Path Guided Operators

The performance graph for the EPGO run depicted in figure 11.8(a) is very nice, and has a trend very much alike that of the DR run (depicted in figure 11.5(a)). But unlike the DR run, the EPGO explodes in size, see figure 11.8(b). The EPGO is the run with the largest average size and also the

(a) Points



(b) Size

Figure 11.8: Performance(a) and size(b) graphs for the EPGO run.

The simplest approach, the BASIC run, seems at first to outperform the other methods, at least when it comes to plain performance. We feel, however, that it is a valid to also recognize that the performance of the BASIC run show signs of convergence. And as stated in the problem definition (see section 5.3) we believe the avoidance of premature convergence to be essential to the evolution. We can not say for sure that the BASIC run in fact has converged prematurely, but we can say for sure that neither EPGO nor DR shows sign of convergence. On the contrary, the heavy fluctuation in performance indicates that convergence has still not occurred, and yet the fluctuation of both runs contain numerous peaks close to the maximum of 15, demonstrating that good solutions are found. When considering the degree of bloat, the DR run is clearly the best amongst the three. Furthermore, the DR run is the first to evolve a solution capable of scoring the maximum of 15 points.

Regarding the SR run that only in flashes raises above zero points in performance we must conclude that the parsimony pressure did not provide the necessary latitude for good solutions to emerge. The E-CFG run performs just as poor, and with the least growth observed in the five different runs. The reason for the failure of the E-CFG is credited to the fact that imposing a CFG on genetic operators distorts the distribution with which genetic material can spread throughout the population, and also which types of new genetic material can be introduced into the population. All together the run is in high risk of premature convergence.

12.4 Future Work

In this section different extensions and modifications are suggested to the methods applied in the solution of our defined problem. We will continue to use the two classifying classes of extensions introduced in chapter 5, namely agent specific extensions and system specific extensions.

12.4.1 Agent Specific Extensions

The concepts of agent specific extensions are as follows:

1. Extend the individual to contain multiple different specialized parse trees, instead of just one general.
2. Introduce the concept of memory for the parse trees to use.
3. Increase the dynamics of the environment.

Introduction of Specialized Parse Trees

The individuals described in this thesis is represented by only one general purpose strategy, described by the parse tree. Instead, several parse trees could be evolved for each individual. One parse tree for all different subtasks like weapon selection, aiming, offensive and defensive movement and overall behavior selection. That is, one parse tree could be evolved to be responsible for selecting amongst the other parse tree, i.e. when to execute offensive

Since we want to evolve flexible bots, capable of performing well in different environments we believe increasing the dynamics of the system will improve the evolved solutions. The environment mainly consists of two components; the players and the map (including the items). We have tried to approach the challenge of player dynamics by introducing the competitive fitness function and this step showed to provide dynamics to the environment. Another interesting experiment would be to evaluate the bots against human opponents by setting up a server on the internet.

We believe that another challenge to be dealt with is the dynamics of the map. When the bots are spawned on the map for evaluation, it is always at one of the multiple spawning points on the same map. It is therefore feasible to believe that the evolved solutions to a certain degree will be adapted to the specific map. One solution could be to change map after each ended generation but bots just spawned would still have the same health, armor and ammunition as always when they start an evaluation.

Inspired by Nordin and Banzhaf (1997) we propose a bots spawning condition to be a result of the previously evaluated bots end condition, metaphorically speaking it would be like considering the UT agent body as a vessel and the GP tree as the driver, a new driver would then receive the vessel in the state the previous controller left it. In Nordin and Banzhaf (1997) this method is also used for practical reasons since it is thereby avoided to bring a mobile robot back to a start location but it is also mentioned that using the same initial starting condition could result in over-specialization and failure to evolve a behaviour that can generalize to unseen environments and tasks.

for success is a well designed function and terminal set. The trees we have been analyzing indicate that the function and terminal set is only partly exploited, for instance the `if-less-than` function does not seem to be used at all except as a `prog-x`, hence all the terminals representing sensor values are obsolete. Two approaches could be taken to deal with this undesirable circumstance. We can try to help the evolution to use the `if-less-than` function in a proper way (maybe using the same principles as in the E-CFG) or we can just remove all the unused functions and terminals.

Another thing which we think might improve the function and terminal set would be to change the action set consisting of terminals to functions taking arguments. Recall the terminals `turn-right` and `turn-left` that turns the bot a steady amount of degrees every time they are executed. If they were functions they could take an argument for deciding the value of degrees to turn. The same principle could be applied to the rest of the action set e.g. `move-forward` or `strafe-right`, where an argument could decide the distance the bot should move.

Combinatory Powers

We would like to test the performance of a combination of the methods. For instance the E-CFG could easily be combined with EPGO and DR or SR. In addition it seems obvious to believe that some of the GP parameters could be fine-tuned and further experiments would have to be done to decide this. As described in chapter 11, SR and DR ensure a limited growth of the trees in the population, but the pressure applied on the individuals by using these methods could influence the evolution of more advanced solutions. Additional test runs would give a better answer to this question.

more complex strategies from small, robust and basic strategies. In addition to removing all high level functions and terminals, we expect this extension to be fertile in producing new creative strategies.

Introduction of Non-Destructive Crossover

As showed in several papers (e.g. Nordin and Banzhaf (1997)) the crossover operator in GP has a tendency to produce offspring less fit than the parents. This undesirable effect also destroys a lot of potential sound building blocks in that the parents code containing the building blocks is exterminated. In an attempt to test the destructive hypothesis, Soule (1998) suggests an experiment using non-destructive crossover to eliminate the destructive effect of crossover (Soule (1998) was inspired by similar methods proposed by O'Reilly and Oppacher (1995) and Hooper et al. (1997)). In Soule (1998)'s version of non-destructive crossover, after each crossover operation, the fitness of the offspring is compared to the fitness of the parent program. An offspring is incorporated into the new population only if its fitness equals or exceeds that of its parent, otherwise the parent is kept. In O'Reilly and Oppacher (1995) multiple attempts were made to produce more successful offspring and if all the attempts failed, the parents were replaced by randomly created individuals.

The method described by Soule (1998) seems sensible to be used in general in that it maintains and improve building blocks assuming that the building block hypothesis is correct. In addition we would propose to combine non-destructive crossover with EPGO. Considering our experiments and hypothesis about EPGO we think that this combination could show powerfull. It

M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE-EC*, pages 17–26, 2001.

E. D. De Jong, R. A. Watson, and J. B. Pollack. Reducing bloat and promoting diversity using multi-objective methods. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 11–18. Morgan Kaufmann Publishers, 2001.

K. A. De Jong. *An analysis of the behaviour of a class of genetic adaptive systems.* PhD thesis, University of Michigan, 1975.

Epic-Games, Infogrames, and Digital-Extremes, 2001. URL `http://www.unrealtournament.com/`.

F. Fernández, M. Tomassini, and L. Vanneschi. Studying the influence of communication topology and migration ondistributed genetic programming. In *EuroGP2001, 4th European Conference on Genetic Programming*, pages 51–73. Springer Verlag, 2001.

The Gamebot-Project. Gamebots: Official site, 2001. URL `http://www.planetunreal.org/gamebots/`.

C. Gathercole and P. Ross. An adverse interaction between the crossover operator and a restriction on tree depth. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 291–296. MIT Press, 1996.

D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing Company, Inc., 1989.

J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

W. B. Langdon. Directed crossover within genetic programming. Technical Report RN/95/71, University College London, UK, 1995.

W. B. Langdon and R. Poli. Fitness causes bloat. In *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 1997.

S. Luke. Genetic programming produced competitive soccer softbot teams for robocup97. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 214–222. Morgan Kaufmann Publishers, 1998.

S. Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation 4(3)*, pages 274–283, 2000.

M. L. Minsky. *The Society of Mind*. Simon & Schuster Inc., 1988.

D. J. Montana. Strongly typed genetic programming. BBN Technical Report #7866, Cambridge, 1993.

S. Nolfi and D. Floreano. Evolutionary robotics through artificial evolution. *ERCIM News*, (42):12–13, 2000.

P. Nordin and W. Banzhaf. Complexity compression and evolution. In *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317. Morgan Kaufmann, 1995.

P. Nordin and W. Banzhaf. An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming., 1997.

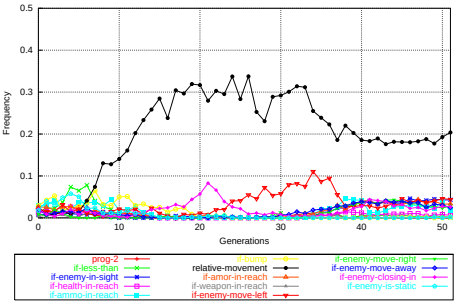
on Evolutionary Computation, pages 781–186. IEEE Press, 1998.

T. Soule, J. A. Foster, and J. Dickinson. Code growth in genetic programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223. MIT Press, 1996.

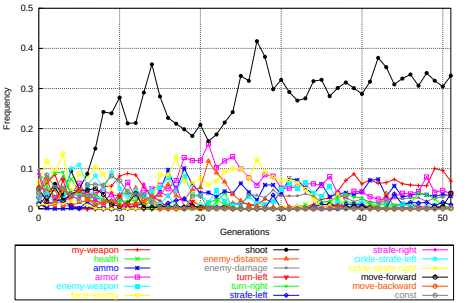
W. A. Tackett. *Recombination, Selection and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, 1994.

T. Yu. Polymorphism and genetic programming. In *EuroGP2001, 4th European Conference on Genetic Programming*, pages 437–444. Springer Verlag, 2001.

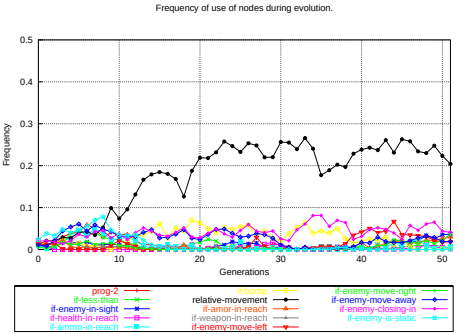
B. T. Zhang and H. Mühlenbein. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, pages 17–38, 1995.



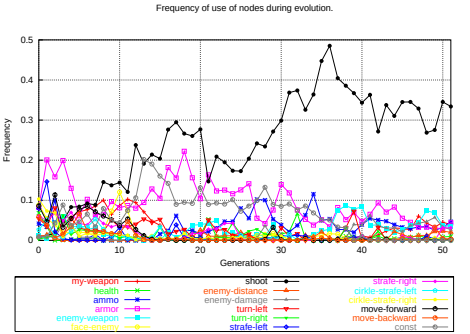
(c) Functions



(d) Terminals



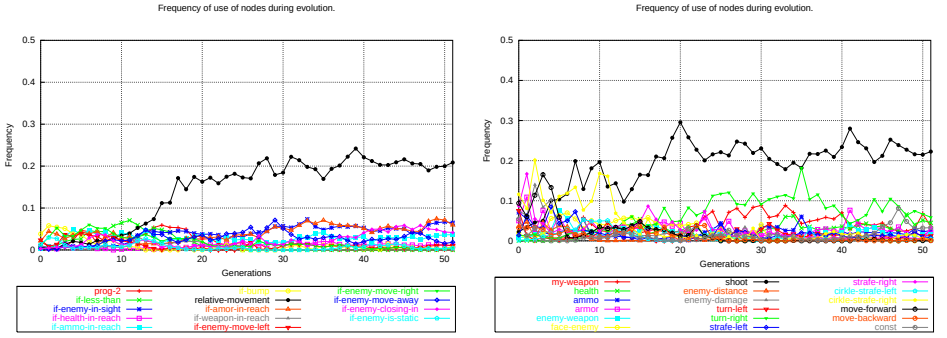
(e) Functions



(f) Terminals

Figure A.1: The frequency of node usage on island 1 (fig. a-b), 2 (fig. c-d) and 3 (fig. e-f) of the 1st run.

(c) Functions (d) Terminals

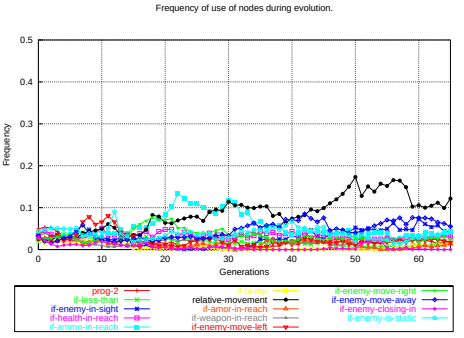


(e) Functions (f) Terminals

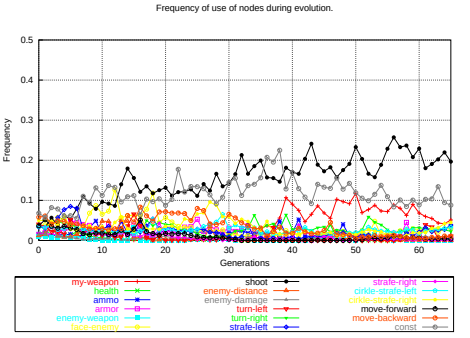
Figure A.2: The frequency of node usage on island 4 (fig. a-b), 5 (fig. c-d) and 6 (fig. e-f) of the 1st run.

(a) Functions

(b) Terminals



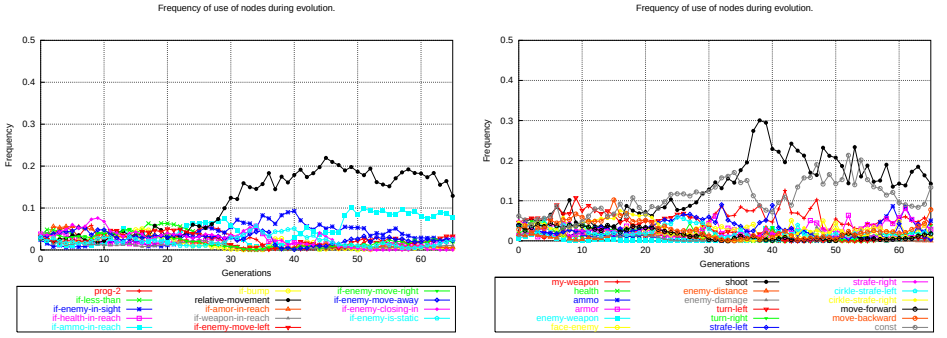
(c) Functions



(d) Terminals

Figure A.4: The frequency of node usage on island 1 (fig. a-b) and 2 (fig. c-d) of the 2nd run.

(c) Functions (d) Terminals

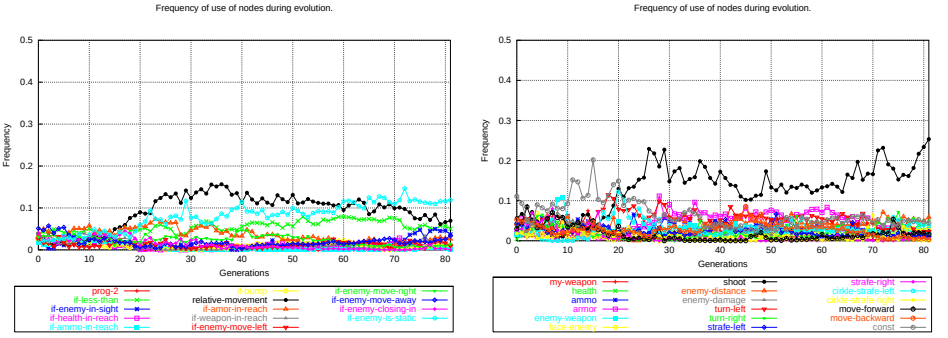


(e) Functions (f) Terminals

Figure A.5: The frequency of node usage on island 3 (fig. a-b), 4 (fig. c-d) and 5 (fig. e-f) of the 2nd run.

(c) Functions (d) Terminals

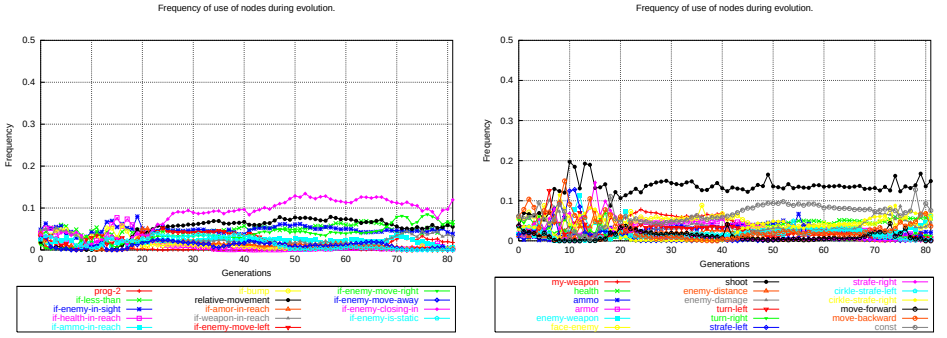
Figure A.6: The frequency of node usage on island 6 (fig. a-b) and 7 (fig. c-d) of the 2nd run.



(a) Functions (b) Terminals

Figure A.7: The frequency of node usage on island 1 (fig. a-b) of the 3rd run.

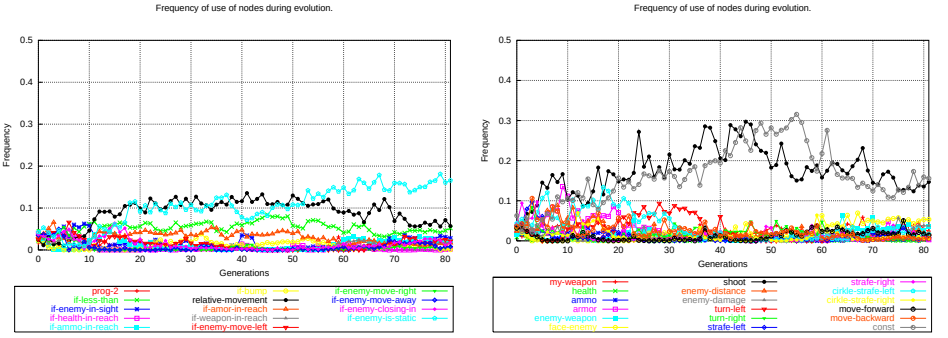
(c) Functions (d) Terminals



(e) Functions (f) Terminals

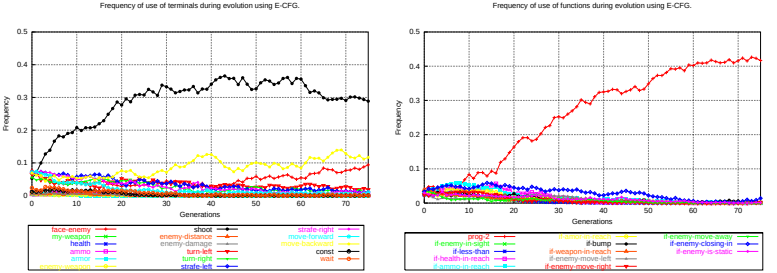
Figure A.8: The frequency of node usage on island 2 (fig. a-b), 3 (fig c-d) and 4(e-f) of the 3rd run.

(c) Functions (d) Terminals



(e) Functions (f) Terminals

Figure A.9: The frequency of node usage on island 5 (fig. a-b), 6 (fig. c-d) and 7(fig. e-f) of the 3rd run.

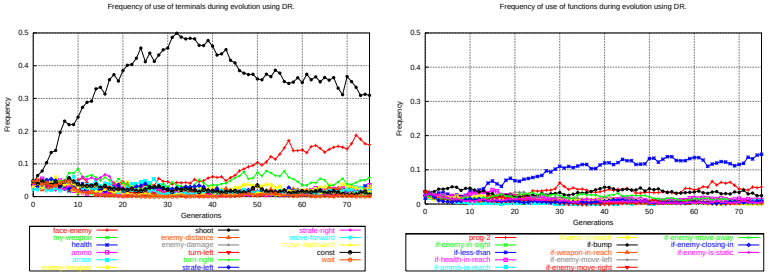


(a) Terminals (b) Functions

Figure B.2: Functions and terminals used during the evolution using the E-CFG.

(a) Terminals (b) Functions

Figure B.4: Functions and terminals used during the evolution using SR.



(a) Terminals (b) Functions

Figure B.5: Functions and terminals used during the evolution using DR.

	The mapping function used is $\frac{Ammo_{current}}{Ammo_{max}} * 200$, so the value will be in the range of [0..200].	
`(armor)`	Assumes the value of the current armor level of the bot.	No effect.
`(face-enemy)`	Assumes the value 200.	If the enemy is in sight then the bot is rotated to face the enemy, otherwise no effect.
`(shoot)`	Assumes the value 0.	Fires on shot in the facing direction. If the enemy is in sight, the bot aims at the enemy.
`(enemy-damage)`	Assumes an estimate of the current damage-level of the enemy. Every time the enemy is hit, we expect him to have taken a certain amount of damage. But this is a temporal quantity, as we expect the enemy to regenerate by picking up healthpackets. Therefore, we reduce the amount according to the formula $d_t = d_{t-1} - \exp(\Delta Hit)$, where d_t is the estimated damage at time t and ΔHit is the time in seconds since damage was last inflicted upon the enemy.	No effect.

Continued on next page

(if-health-in-reach)	2	If health is in reach, then the value assumed by arg1 is assumed, otherwise the value assumed by arg2 is assumed.	If health is in reach, it is picked up, otherwise no effect.
(if-armor-in-reach)	2	If armor is in reach, then the value assumed by arg1 is assumed, otherwise the value assumed by arg2 is assumed.	If armor is in reach, it is picked up, otherwise no effect.
(if-ammo-in-reach)	2	If ammo is in reach, then the value assumed by arg1 is assumed, otherwise the value assumed by arg2 is assumed.	If ammo for the weapon currently used by the bot is in reach, it is picked up, otherwise no effect.
(if-weapon-in-reach)	2	If a weapon that is better than the weapon currently used by the bot is in reach, then the value assumed by arg1 is assumed, otherwise the value assumed by arg2 is assumed.	If a weapon that is better than the weapon currently used by the bot is in reach, it is picked up, otherwise no effect.
(if-enemy-in-sight)	2	If the enemy is in sight, then the value assumed by arg1 is assumed, otherwise the value assumed by arg2 is assumed.	No effect.
(if-enemy-move-left)	2	If the enemy is in sight and moving left, then the value assumed by arg1 is assumed, otherwise the value assumed by arg2 is assumed.	No effect.
(if-enemy-move-right)	2	If the enemy is in sight and moving right, then the value assumed by arg1 is assumed, otherwise the value assumed by arg2 is assumed.	No effect.

Continued on next page