transnationale
UNIVERSITEIT LIMBURG

School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

Evolving Virtual Agents using Genetic Programming

Proefschrift voorgelegd tot het behalen van de graad van
Doctor in de Wetenschappen, richting Informatica,
te verdedigen door

Patrick MONSIEURS

Promotor : Prof. dr. E. Flerackers
Co-promotor : Prof. dr. F. Van Reeth

2002

DOCTORAATSPROEFSCHRIFT

Universiteit Maastricht

LIMBURGS UNIVERSITAIR CENTRUM
PARTNER IN DE UNIVERSITEIT LIMBURG

U

transnationale
UNIVERSITEIT LIMBURG

L

School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

Evolving Virtual Agents using Genetic Programming

Proefschrift voorgelegd tot het behalen van de graad van
Doctor in de Wetenschappen, richting Informatica,
te verdedigen door

Patrick MONSIEURS

Promotor : Prof. dr. E. Flerackers
Co-promotor : Prof. dr. F. Van Reeth

2002

According to the guidelines of the Limburgs Universitair Centrum,
a copy of this publication
has been filed in the Royal Library Albert I, Brussels,as publication
D/2002/2451/45

Acknowledgements

I would like to thank my promoter, Prof. Dr. Eddy Flerackers, and my co-promoter Prof. Dr. Frank Van Reeth for their guidance and support, and for giving me the freedom to investigate several areas of research. Thanks also go to Prof. Dr. Karin Coninx for helping me with the writing of several papers.

I would also like to thank my colleagues for the support and friendship they have given me over the past years. In particular I would like to thank Michael Bar, Johan Claes, Fabian Di Fiori, Guy Linsen, Kris Luyten, Chris Raymaekers, Jan Van den Bergh, and Tom Van Laerhoven.

Thanks also go to Nathalie Cossement, Kurt Driessens and Nico Jacobs for their help and advice while working on the Robocup domain. Without them, the early work on developing Robocup players would not have been possible.

I would also like to thank the friends I made over the course of my studies and afterwards, in particular those from the student organizations Filii Lamberti, Åstøriå, WINA, and Biomedica.

Last but not least, I would like to thank my parents and my sister, who supported me all these years. Without them, this would not have been possible.

Abstract

Virtual environments are used in a diverse number of applications, ranging from medical applications, military simulations, modeling and engineering, to entertainment such as games or virtual communities. In these applications, virtual agents can be used to make the environment more realistic, perform tasks that are tedious and time consuming for humans, or even simulate the presence of other users in the environment.

When constructing an agent for a virtual environment, several issues are encountered that must be resolved. First, a virtual agent must be able to explore and navigate in the virtual environment in a realistic way while avoiding collisions with obstacles. If the virtual agent does not have access to the internal representation of the environment, it will have to use its virtual sensors to observe the environment. In this thesis, an algorithm is presented to perform obstacle avoidance and map construction in a virtual environment using a synthetic vision sensor. The constructed map can then also be used to navigate in the environment.

A second issue is communication between agents and users in the environment. Agents and users must be able to locate agents that can perform certain tasks, and agents may offer their services to users or other agents. These issues are discussed briefly in this thesis, and a prototype of a multi-agent virtual environment is presented.

The most difficult issue of virtual agents is learning to solve problems in an environment, without knowing the constraints and rules of the environment in advance. This thesis will examine the use of genetic programming to train virtual agents. Two important problems are encountered when using genetic programming in this domain. First, programs constructed using genetic programming tend to grow rapidly before an acceptable solution is found. Several techniques will be presented to reduce the size of the evolved genetic programs, and a comparison will be made between these techniques. Secondly, evaluation of candidate solutions is usually very time consuming, making it impractical to maintain a large population of candidate solution. A large population is usually a requirement to evolve good solutions. Therefore, an algorithm to reduce the size of the population while maintaining the diversity of a larger population is presented. These optimizations will also be applied to the virtual multi-agent system of robotic soccer to examine the effects of these optimizations in a complex environment.

Table of contents

List of figures

List of tables

Chapter 1: Introduction

The goal of this thesis is to design agents that exist in a virtual environment. These agents must be able to interact with the environment, with each other, and with users present in these environments. Furthermore, these agents must be able to perform certain tasks, and learn to improve themselves through experience in the environment. The most important method used for learning in this thesis will be genetic programming. The key elements of the thesis will be discussed in the remainder of this chapter, followed by an overview of the thesis.

1 Virtual environments

Because of the recent technological advances in computer graphics hardware and computing power, realistic virtual 3D environments have become a reality. Computer generated environments are currently being used for various applications. Some typical examples of virtual environments are virtual networked communities, first person shooter games and virtual marketplaces.

Virtual environments can also be useful for other applications. For example, a simulation of a real environment can be a useful tool for training military personnel in dangerous situations. They are also used for the creation of animation films or for adding special effects to movies, where the realism of the environment is of great importance. Virtual environments can also be used to provide a more natural interface to existing applications.

The advantage of using a virtual environment is that the user has a feeling of being present in the environment. Because of this immersion, the user is able to interact more naturally with the environment. When the user also has a graphical representation of himself in the environment (called an avatar), it is also possible for other users to easily recognize each other, identify them, and see what they are doing.

A disadvantage of virtual environments is that some tasks are more difficult to perform graphically. For example, entering numeric data, placing an object at an exact location or finding a certain object or user in the environment. For some of these problems, agents moving around in the environment can provide a solution. For example, an agent that regularly walks around in the world may know the locations of objects (or other agents). These virtual agents will be discussed in the next section.

2 Intelligent virtual agents

Software agents are programs designed to perform a specified task for a user or another agent. In contrast to normal computer programs, however, agents have several unique characteristics:

- Autonomy: The agent is able to make some decisions without needing to contact the user that uses the agent. Specifically, the decisions that the user should not be concerned with are handled by the agent.
- Social ability: Agents can interact with users or other agents.
- Long-lived: This means that an agent is usually not terminated when a task has been completed. The agent will remain present in the system until it is needed again, possibly by another user.
- Reactivity. The agent will modify its parameters based on the current state of the environment. For example, the agent can respond to a situation change in the environment and change its activities to reflect these changes.
- Proactive: An agent can execute a task before a user has requested the agent to perform that task. The agent can monitor the environment, and may start to execute a task before the user notices that the task must be performed.

A virtual agent is an agent that is present in a virtual environment. The agent will have an avatar representation in the environment, to facilitate the interaction with users.

An intelligent agent is an agent that has the capability to learn some of the tasks it has to perform, either by experimenting in its environment and examining the results or by studying previously executed correct examples of the task. How a computer program can learn to perform a task will be discussed in the next section.

3 Genetic programming

Evolutionary algorithms are an optimization technique based on Darwin's principles of survival of the fittest. In this technique, a population of candidate solutions for a problem is maintained, and the quality of these solutions is determined and called the fitness of the solution. Based on their fitness, solutions are selected from the population (the parents) and combined with each other to form new candidate solutions in the population (the children). This way, the good parts of different solutions have a chance to be combined in a new solution. After a 'generation', all the parents in the population have been replaced by their children.

Genetic algorithms are evolutionary algorithms where a solution is sought for one specific problem. Often, a set of numbers must be found that optimize a criterion. Solutions can be combined by cutting the bit string representing these numbers and swapping the parts between individuals.

Genetic programming is an extension to genetic algorithms where the candidate solutions represent a program that attempts to solve several instances of a problem. The representation is often a tree consisting of operations on values, variables or other operations. Solutions are combined by swapping subtrees between candidate solutions. The tree representation has a variable length.

Genetic programming has several problems that must be dealt with:

- Because of the variable length encoding, the size of the solutions can grow rapidly. As a result, the solutions take more time to evaluate, require more memory to store. Also, because of the principle of Occam's Razor, these solutions tend to be less general than shorter solutions.
- For complex problems, types and syntactic constraints must be added to the nodes of the tree.
- After several generations, all the candidate solutions in the population tend to have a similar structure, and it becomes very hard to find better solutions. In this case, the diversity of the solution has become too low.

In this thesis, solutions for these issues will be presented.

4 Contributions of this thesis

This thesis has the following contributions to the domains of virtual reality and genetic programming:

- Algorithms will be presented to perform map construction, collision avoidance and navigation of an agent in a virtual environment, using synthetic vision sensor data of the agent (depth information). These problems are also encountered by mobile robots in non-virtual environments.
- A method to remove code from genetic programs that has no effect on the result of the program will be presented. The effects of this method on the size and convergence speed of the population will be examined.

- Several methods to reduce the average size of the solutions in the population of genetic programming will be presented and compared with each other. Again, the effect on average size and convergence speed will be studied.
- An algorithm will be introduced to maintain the diversity of a population in genetic programming, and its effects on convergence speed will be presented. This method can be used to drastically reduce the population size without negatively affecting performance.
- Genetic programming will be applied to agents on the Robocup domain. The effects of the optimizations on genetic programming described above are studied on a complex problem.

5 Thesis overview

This thesis is divided in two parts. Part I will discuss the topics related to virtual agents, but will not discuss the learning of the agents in the environments. Chapter 2 discusses the uses of virtual environments, and gives examples of existing applications. Chapter 3 deals with several important problems that occur in virtual environments: constructing a map of the environment, navigation and collision avoidance. Chapter 4 introduces Robocup as a research domain for virtual environments and collaborative agents.

Part II of the thesis will focus on the artificial intelligence aspect of virtual agents, using genetic programming. In chapter 5 the concepts of genetic programming will be explained and several examples of genetic programming will be given. Chapter 6 and chapter 7 present several optimization techniques that will increase the convergence speed of genetic programming and produce better solutions. In chapter 8, genetic programming will be applied to learn behaviors for agents in the Robocup domain using the optimization techniques presented in chapter 6 and chapter 7. Finally, conclusions and directions for future work will be presented in chapter 9.

Part I: Virtual agents

Chapter 2: Agents and virtual environments

1 Introduction

This chapter introduces the key concepts of this thesis: virtual environments and agents. First, definitions of these concepts will be given. Afterwards, several application areas of virtual environments and agents will be presented. Finally, some important issues that must be covered in an application using agents in virtual environments will be presented. These issues will be covered in the following chapters.

1.1 What is a virtual environment?

Several definitions of virtual environments can be found in the literature. According to Dix et al. [21], "*Virtual Reality (VR) refers to the computer-generated simulation of a world, or a subset of it, in which the user is immersed. It represents the state of the art in multimedia systems but concentrates on the visual senses*".

Bryson [8] states that "*Virtual reality is the use of various computer graphics systems in combination with various display and interface devices to provide the effect of immersion in an interactive three-dimensional computer-generated environment in which the virtual objects have spatial presence. We call this interactive three-dimensional computer-generated environment a virtual environment*".

A third definition is given by Kalawski [39]: "*Virtual environments are synthetic sensory experiences that communicate physical and abstract components to a human operator or participant. The synthetic sensory experience is generated by a computer system that one day may present an interface to the human sensory systems that is indistinguishable from the real physical world*".

All these definitions have in common that the representation of the artificially created world is represented to a user through the sensory system. Vision is the most important sensory system, and therefore most work concentrates on the graphical aspects of virtual environments. However, to achieve the highest level of immersion for the user, audio and haptic interfaces to the virtual environment must also be considered [88]. The virtual environment is used to execute a specific task in a more natural and intuitive way. The task to be executed depends on the domain in which the virtual environment is used. The work in this thesis will focus mainly on the visual aspects of virtual environments.

1.2 What is an agent?

According to Wooldridge and Jennings [112], in a weak notion of agency, the term agent is used to denote a hardware or software system that has the following properties:

- Autonomy: Agents operate without direct intervention of humans or others, and have control over their actions and internal state.
- Social ability: Agents interact with other agents or humans via some kind of agent-communication language.
- Long-lived: Because agents are autonomous, they can remain active in the environment after a task performed for a user is completed. The agent can remain present in the system until it is needed again, possibly by another user or agent.
- Reactivity: Agents perceive their environment and respond in a timely fashion to changes that occur in it.
- Pro-activeness: Agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative.

A virtual agent is an agent that exists in a virtual environment and uses the sensory inputs provided by the environment. In this case, the virtual agent uses a set of synthetic sensors to observe the environment, and interacts with the environment in an identical way as human users. The virtual agent also has a physical representation in the virtual environment, called an avatar.

2 Examples of virtual environments

To demonstrate the usefulness of virtual environments and virtual agents, several examples of virtual environments and/or virtual agents will be presented. These application areas include medical and military applications, design and engineering, modeling, virtual communities and entertainment.

2.1 Medical applications

Virtual environments have several applications in medicine. Lee et al. [54] use virtual environments to train medical students on eye examinations. Neumann et al. [75] use a virtual model of a patients skull to plan surgery to correct skull deformations, and to predict the result of this surgery. Di Girolamo et al. [20] investigated the use of virtual reality in both the assessment and rehabilitation of vestibular conditions. However, Rajani and Perry [87] noted that virtual reality techniques should only be applied in medicine if the starting point of the research is the nature of medical work and not the technology itself.

2.2 Military applications

Virtual environments are a useful tool to train personnel for situations that are too dangerous or expensive to train in real life. For example, Everett et al. [24] describe a virtual environment to train fire fighters aboard a navy ship. Another use of virtual environments is the simulation of combat situations. Hix et al. [33] and Julier et al. [38] describe how such a simulation can be visualized in order to determine the outcome and effectiveness of the simulation. Because these simulations require many participants, a lot of research focuses on networked virtual environments [96]. Some of these participants can be replaced by virtual agents. Hill et al. [32] use agents to control helicopters in a battlefield simulator.

2.3 Design and engineering

Virtual reality has the advantage that objects can be examined before they are actually created. This can be a useful tool in design and engineering applications. For example, Sastry and Boyd [95] demonstrated the usefulness of virtual environments in this area by creating applications for virtual prototyping, virtual assembly and virtual simulation. Virtual assembly can also be used for the design and evaluation process of mechanical machines, as demonstrated by Jayaram et al. [37]. Giallorenzo et al. [26] used virtual reality to model the ventilation system of a hospital using computational fluid dynamics. Virtual agents could be used in virtual simulation to model humans walking around in the environment, making the simulation more realistic.

2.4 Modeling

To design a virtual environment, a modeling tool is used. Often, these tools operate in a virtual environment. The UNC-Chapel Hill Immersive Modeling Program (CHIMP) is a modeling environment that allows a modeler to design virtual rooms using libraries of existing objects [66]. IM-Designer, developed by Coninx [15][16], is another environment that also allows the creation of new objects. IM-Designer uses a combination of 2D and 3D user interfacing. ICOME, which is an abbreviation for Immersive Collaborative 3D Object Modeling Environment, is a modeling environment where several users can work together on a modeling task [89]. ICOME is the successor of IM-Designer, and will be used in the next chapter as an environment to test navigation and map construction.

2.5 Virtual communities

Virtual communities are networked virtual environments where multiple users can join and interact with each other. Applications of virtual communities include virtual conferences, virtual marketplaces or multiplayer on-line games. The Distributed Interactive Virtual Environment (DIVE) [10] is an internet-based multi-user VR system where participants navigate in 3D space and see, meet and interact with other users and applications. DIVE applications and activities include virtual battlefields, spatial models of interaction, virtual agents, real-world robot control and multi-modal interaction. MASSIVE is another distributed virtual reality system and concentrates on the interaction between users and/or agents in the environment [28]. VLNet [9][82] is a virtual environment using realistic human-like avatars. VLNet supports the simulation of the virtual environment, communication with agents and users, object behavior, and navigation. The environment also supplies an interface for detailed face and body representations.

2.6 Other applications

Virtual environments can also be used in many other application areas. For example, the virtual environment can be used as an interface for another computer program, to make the interaction with this program more intuitive. An example is given by the UNIX process manager PSDoom, developed by Chao [11]. This is a UNIX administration tool based on the first-person shooter game "Doom". The monsters in the environment represent the processes running on the computer. The player can move around in this environment and shoot at the different processes, reducing their priority, or even terminating them when they are killed. Processes can defend themselves by attacking the user, or even other processes.

3 Issues concerning virtual agents

When developing virtual agents, several important issues arise which must be resolved.

3.1 Navigation in the virtual environment

Because virtual agents have a spatial presence in the environment and use the same sensory and motion systems as normal users, navigation of the agents in the environment is a problem. The agents are expected to move towards specified places in the environment without bumping into obstacles or getting trapped in the environment. Therefore, the agent must be able to explore the environment and construct a map of the environment. The agent can then use path planning to move around in the environment in an efficient way. In a multi-agent system, some specific agents can be assigned to this task, and can then help the other agents with path planning. Navigation and collision avoidance will be covered in detail in chapter 3.

3.2 Communication between users and/or agents

When constructing a community of virtual agents and users, several communication issues arise. First, methods must be developed on how agents and/or users can communicate with each other. Secondly, a method to locate an agent that can perform a given task must be present. Thirdly, when the avatar of an agent or user is encountered in the virtual environment, a method must be developed to discover the capabilities of this avatar. These issues will be discussed briefly in Appendix A. In [83], Pandzic et al. discuss the addition of autonomous actors in VLNet, and their interfaces with the environment and users.

3.3 Intelligent virtual agents

It is difficult to give an accurate definition of intelligence. However, intelligence of virtual agents implies that the agents are able to adapt to the environment they inhabit and learn to improve themselves, without giving the agents explicit knowledge about this environment. Using evolutionary methods, it may be possible to achieve this adaptation. Chapter 5 of this thesis deals with the evolutionary computation paradigm of genetic programming, and discusses some specific problems that must be dealt with when developing virtual agents. Possible solutions for these problems are developed in chapter 6 and chapter 7, and in chapter 8 these solutions will be tested on the more complex domain of virtual robotic soccer players.

Chapter 3: Navigation in virtual environments

The algorithm presented in this chapter allows an agent to navigate through a virtual world in real time without colliding with obstacles, and to construct an isometric topological map of the environment using only a virtual depth sensor. The maps created are compact and contain the necessary information for navigation and path planning.

A future goal of our research is to speed up map construction by having several agents explore the world simultaneously, and exchanging pieces of the map when they encounter each other. Another possible extension is to construct three-dimensional maps. This would complicate the algorithm, but the same ideas could be used in three dimensions.

1 Introduction

The world of virtual agents is similar to that of mobile robotics: in both areas, an entity must be able to navigate in an environment. In this chapter a method will be presented for an agent to navigate in a virtual world, without requiring access to the internal representation of this world. The only perception the agent has of the world is available through the depth information of an image that is rendered from the position of the agent. The advantages of using a rendered image are:

- The agent can be used with different virtual environments that use different internal representations.
- The image of the environment may be rendered on a different machine. In this case, only the rendered image is available to the agent.
- The technique more closely resembles the real world where robots use distance sensors to sense the environment.
- The technique does not depend on the complexity of the environment.

By working in virtual reality, it is possible to avoid the hardest problems that are encountered when working with mobile robots in the real world. These problems are noisy sensors and determining the exact location of the robot. The problem can be simplified even further by limiting the movement of the agent to two dimensions. Because of these simplifications, it is possible to concentrate on the problems of collision avoidance, path planning and map construction.

Depending on the view angle of the rendering system, the sensors of the agent detect only a small portion of the environment around the agent, typically an area of about 60 degrees. To extend this range to the full 360 degrees around the agent, it rotates and previous measurements are used to supplement the current measurements (see figure 3.1). When the agent makes a full turn around its center, it will have observed the entire environment around its current position.

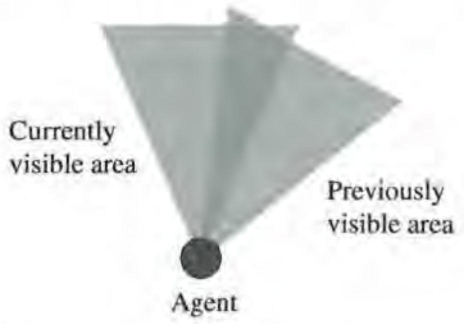


Figure 3.1: Current and previous sensor data around the agent.

The recorded depth information is used in several ways. At the lowest level of navigation, it is used to avoid collisions during the next time step by comparing the current speed of the agent with the distance of obstacles in its path. At a higher level, this information is used to move around obstacles when moving to intermediate destinations that are selected by higher levels. At the highest level, a map of the environment and an accessibility graph connecting areas is created that is used for path finding.

2 Related work

Other researchers have already used virtual perception for navigation. Reynolds [90] discusses several techniques for low-level obstacle avoidance. These include using silhouettes of obstacles, obstacle density images, and the use of a depth image of the environment. The main drawback of the technique is that without higher level planning, the agent tends to move into local openings that turn out to be dead ends. Blumberg [7] used motion energy of images for low-level navigation of virtual creatures through corridors. Rabie and Terzopoulos [107] use color histograms of stereo images to detect a target in the environment.

Kuffner and Latombe [46] render a scene with flat shading where every object has a unique color to determine the visible objects in the scene. The objects are then retrieved from the internal representation to determine their position and speed.

The problem of map construction has been studied extensively in mobile robotics, although most work in that area concentrates on location determination and resolving sensor noise. For the actual map construction, two methods can be used: topological maps and grid-based (metric) maps. When using a grid-based map, the environment is subdivided by a grid of equally spaced cells. Each cell contains a value that indicates the probability that an obstacle is present in the area of the cell. A topological map consists of a graph where nodes represent locations in the environment, and two nodes are connected by an edge if a direct path exists between the locations of their nodes.

Both approaches have their strengths and weaknesses. These are discussed in [108], but we mention here the most important differences. Grid-based maps are easy to construct but have large space and time complexity. They also have a fixed resolution. Topological maps are compact and allow fast path planning, but are difficult to construct.

Noser et al. [79] use depth information retrieved from the Z-buffer for navigation and map construction in a virtual environment. The constructed map is stored in an octree. The map is dynamically updated when new obstacles are detected, or when previously observed objects are moved. Navigation is performed by creating high level and low level goals, such as following corridors or moving to a specified location. Obstacle avoidance uses low level vision information to avoid collisions.

Thrun [108][109] combines both approaches by building a topological map on top of a grid-based map. The algorithm creates a small set of regions by decomposing the grid-based map. These regions are separated by narrow passages, called critical lines. The partitioned map is then mapped to an isomorphic graph where the regions correspond to nodes and the critical lines to edges.

Choset [13] uses an algorithm that constructs a generalized voronoi graph of an environment using only local sensor information. This graph is used as a map of the environment, where its edges can be used for path planning. The edges of the generalized voronoi graph represent locations that are equidistant to the nearest surrounding points in the environment. The graph is created incrementally by tracing the edge of the generalized voronoi graph, using the location of the *m* nearest points to the robot (where *m* is the dimension of the environment). To determine the nearest points, the robot uses omni-directional sensor information around the robot at every step of the algorithm. This feature makes this approach inappropriate for our system, because only sensor information about a limited view angle in front of the agent is available.

The agents of Kuffner and Latombe [46] remember the ID's of visible objects, and their transformation at the moment they were last observed. This information is stored to assist the agent with navigation.

Zhukov et al. [114] use agents that build maps of the environment off-line by subdividing the environment in areas that are connected by an accessibility graph.

3 Synthetic vision

Our goal is to develop a synthetic vision system that resembles real world vision, and is still efficient enough to operate in real-time. This section describes the synthetic vision system developed by us, as presented in [68] and [69]. Human vision is able to detect the distance of objects by comparing two stereo images. While this is possible to do, this approach is difficult and too slow to implement in a real-time agent. Therefore, most mobile robot systems are also equipped with proximity sensors like sonar or a laser range finder. However, in virtual reality, the depth information of the visible environment is already calculated by the graphic pipeline and is available in the depth buffer of the rendered image. This provides the agent with the distance to the objects in the environment over an angle of about 60 degrees in front of the agent, without performing time-consuming calculations.

The virtual agent can sample the depth buffer, and store the retrieved distances in a vision buffer that represents the entire 360-degree area around the agent. This vision buffer is a discrete subdivision of this environment in small segments. Each segment contains one depth value of a small angle around the agent (about 1 to 3 degrees, depending on the resolution of the vision buffer).

When the agent moves around in the world, previously invisible areas around the agent will become visible, and these readings will be added to the vision buffer. New readings are added to the previous readings, so the agent can obtain the depth information about the entire area around it when it rotates around its axis.

During every simulation step, the sensors are updated with the currently visible depth values. The following steps must be performed during every update:

- Reading and correcting the depth information from the depth buffer.
- Filling the vision buffer.
- Transforming the depth information after every move of the agent.

In the remainder of this paragraph, we will elaborate on each step.

3.1 Reading depth information

Since we restrict the movement of the agent to two dimensions, only obstacles located at the same height as the agent need to be detected. Therefore, it suffices to scan the single line of the depth buffer at the same height as the virtual camera. This simplification can cause problems with concave objects, but these problems can be solved when the system is expanded to three dimensions.

In the next step, two corrections must be performed on the retrieved depth values. In the OpenGL Z-buffer implementation, the values stored in the depth buffer are a non-linear mapping of the distance between the near and far clipping planes to the interval [0; 1]. This mapping has a higher resolution at the nearby distances. The measured values can be transformed to the real distance by equation (3.1):

$$\text{RealWorldDistance} = \frac{-2 \cdot F \cdot N}{2(Z - 0.5)(F - N) - (F + N)} \tag{3.1}$$

where *F* and *N* are the distances to the far and near clipping planes respectively, and *Z* is the depth value that was extracted from the depth buffer.

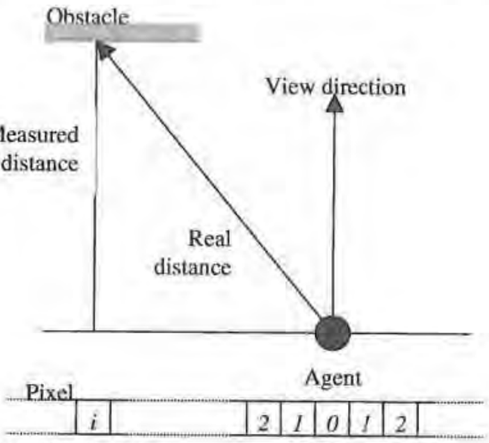


Figure 3.2: Correcting the depth value.

A second transformation is necessary because the measured depth value is not the distance from the camera to the obstacle, but the distance to the plane orthogonal to the camera position (see figure 3.2). To correct this, the distance at pixel$_i$ is multiplied by Correction$_i$, where pixel$_0$ is the pixel in the center of the view port, and pixel$_i$ is the pixel at a distance of *i* pixels from pixel$_0$. The values Correction$_i$ are calculated with equation (3.2):

$$\text{Correction}_i = \sqrt{1 + (i \cdot C)^2} \tag{3.2}$$

where the constant *C* is defined by equation (3.3):

$$C = \frac{\tan(\text{ViewAngle}/2)}{\text{ViewPortW}/2} \tag{3.3}$$

where ViewAngle is the camera's field of view in the horizontal direction, and ViewPortW is the resolution of the view port in the horizontal direction. The values Correction$_i$ need to be computed only once and are then stored in memory.

3.2 Filling the vision buffer

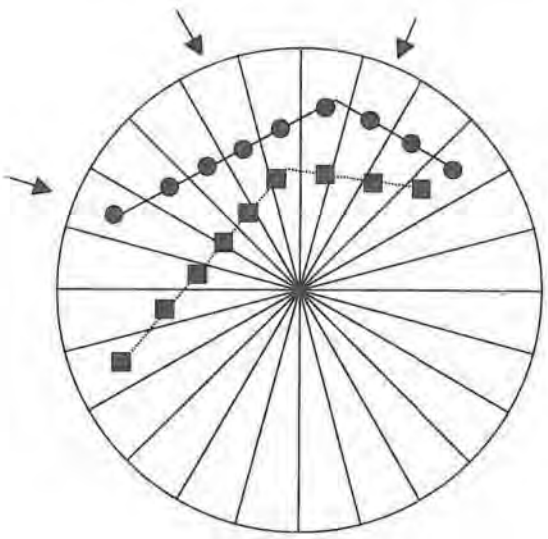
When the depth values are transformed, the value at the center of every visible segment is placed in the vision buffer, overwriting any previous estimated values. By comparing the detected value with the estimated value, it is possible to detect changes in the environment. This information could then be used to identify moving objects. However, this is currently not implemented in our system. The obstacle avoidance and map construction algorithms can now use the data in the vision buffer.

3.3 Transforming the depth information

The coordinate system of the vision buffer is relative to the position and orientation of the agent. This means that the depth information about the visible area is always stored in the same segments. As a result, after the agent moves using a transformation matrix M, the depth values in the buffer must be transformed to correspond to the new position and orientation of the agent. To calculate the new vision buffer, every depth value in the buffer is transformed to its coordinates in 2D-space. The inverse transformation M^{-1} is applied to these coordinates, and the new position is then inserted in the new buffer (as is shown in figure 3.3). This gives the agent a prediction of the environment. This prediction will be updated with the sensor readings of the new position.

The simple approach presented above has several problems. First, it is possible that two adjacent points in the original buffer will be transformed to non-adjacent positions in the new buffer. In this case, a "hole" is created in the buffer (this is demonstrated in figure 3.3). To fill these holes, a depth value has to be interpolated based on the surrounding values.

A second problem is the occurrence of rounding errors when a depth value is transformed around the agent. The rounding errors are due to the discrete subdivision around the agent. The transformed point will be placed in the segment that is nearest to its new position, resulting in a small error. This error is cumulative with the transformation errors of all the previous time steps, and as a result the point will appear to be rotating faster or slower than it should be. This effect is shown in figure 3.4. In this figure, a number of segments are represented by the columns. Each row represents the position of a point after several successive rotations. The first row contains the correct position of the points, and the second row has the positions of the points with the cumulative error that was just described. In this case, the point is rotating too slowly. The next two rows contain two possible corrections that are described below.



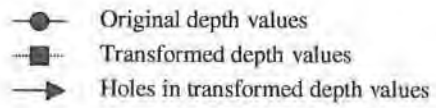


Figure 3.3: Transforming the depth values can leave holes in the vision buffer.

A first solution to this problem is to clip the rotational component of M^{-1} to the nearest multiple of the segment angle. The remainder of the rotation will be stored as a global offset and is added to the transformation in the next time step. The result of this approach is demonstrated in the third row in figure 3.4, and has only a small error in every step that is not cumulative.

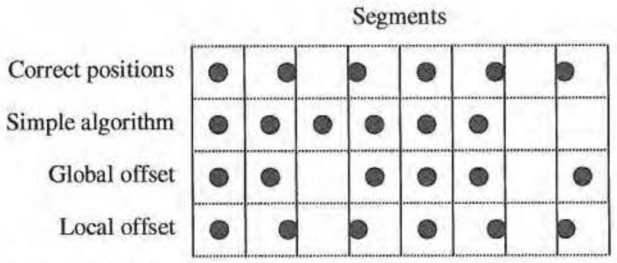


Figure 3.4: Rotating a point in the vision buffer using several methods.

The second solution is to store the remainder of the rotation as a local offset in every segment of the vision buffer. This is shown in the last row in figure 3.4. As can be seen, this gives the best approximation and this is the method that is currently used in our implementation. There is a small memory overhead because every segment must store an additional value, but this overhead is negligible.

4 Collision avoidance

When the agent knows the distance to the obstacles surrounding it, it can use this information to avoid colliding with them. Using the vision buffer, obstacle avoidance can be performed at two levels. At the lowest level this is done by ensuring that the movement commands issued at the current time step will not cause a collision. At the higher level, when moving towards a target position, a temporary short-term goal can be created to move around nearby obstacles. These two methods will now be explained.

4.1 Short range collision avoidance

At the end of a time step, the navigation system will calculate the movement of the agent during the next time step based on the current translation and rotation speed of the agent. The vision buffer is checked to see if sufficient space is available for this move. If this is not the case, an additional speed vector is added to the agent's speed that will prevent the collision. This system responds directly to the sensors, and the agent is therefore able to react immediately to unexpected obstacles.

4.2 Medium range collision avoidance

During the simulation, higher-level processes (like the path planner or map construction algorithm that will be discussed in the next paragraph) will create short-term destinations for the agent. A short-term destination is a position that is reachable from the agent's current position without complex path planning, but it does not exclude the possibility that small obstacles are present on the direct line between the agent and the destination. In this case, the agent must make a small course correction to avoid the obstacle.

The destination is represented as a depth value in the vision buffer. This way, obstacles can easily be detected by testing for a nearer value in the buffer. If an obstacle is found, the segment next to the obstacle that is the closest to the original direction is selected (see figure 3.5). A temporary destination is then created along this direction, at the same distance as the original destination. The agent will then move towards this temporary destination until the original destination becomes visible. At that time, the temporary destination is replaced by the original destination.

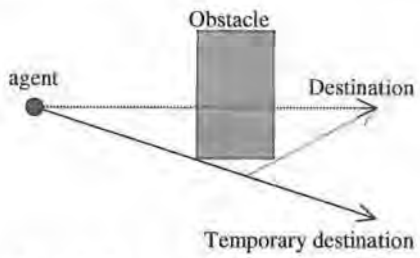


Figure 3.5: Avoiding an obstacle by creating a temporary destination.

Using the aforementioned methods of collision avoidance, in most cases the agent is able to reach the short-term destinations. It is however still possible that the agent will get stuck in the environment. This is possible because some obstacles are only detected when moving closer towards them. Also, some obstacles can be closer than detected because of inaccuracies of the depth sensor. If such an obstacle is detected while moving, the collision avoidance module signals a failure to the higher-level task. The higher-level task can then try to resolve its task in another way.

5 Map construction

As the agent wanders around in the virtual world, it constructs a map of the environment. This map can then be used for high-level path planning. The structure of the constructed map is explained below.

The environment is divided into overlapping areas. Each area has a center, where the area is defined as the environment that is visible from its center. The borders of this area will be approximated by a collection of straight lines, and a set of links to other areas that are reachable from the center of the area is also maintained. The areas and links form an isometric accessibility graph of the environment. An example of a single area is given in figure 3.6. In this figure, the thick lines represent the borders of the area. Discontinuities in the border are marked by the squares originating from the center of the area (the square in the middle). These squares, called the open points, represent passages to other hidden parts of the environment and are potential locations of the centers of adjacent areas in the map.

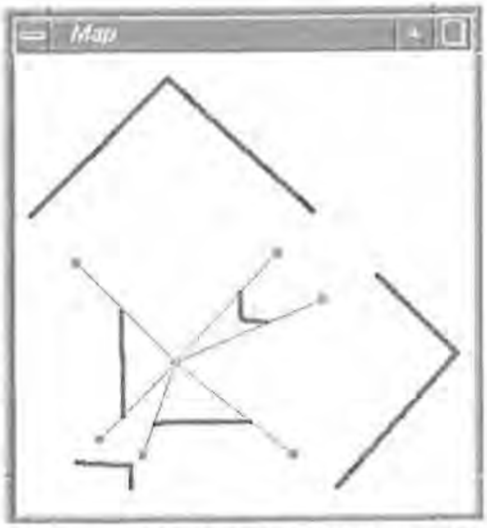


Figure 3.6: Example of an area. The thick lines represent the borders of the area, and the squares connected to the central square represent the open points of the area.

5.1 Creating a single area

First, the agent finds the complete depth information around its position by rotating (without translating) around its axis until it has completed a full circle. Then, it can construct the borders that limit the area by grouping adjacent depth values in the vision buffer that are not separated by a discontinuity. Detecting these discontinuities based on the depth values is not a trivial task. The only concrete rule that we could discover was that the distance between depth values of adjacent points must be greater than the size of the agent (otherwise the agent would not be able to pass through the opening). To solve this problem, a neural network was used to detect a discontinuity. To construct a training set for the neural network, the depth values of a mapping run (using a collection of complex rules) were collected, and the incorrect values were manually corrected. To detect a discontinuity between two points, the four depth values (d_1 to d_4) surrounding the possible discontinuity were used. Also, the differences between the values ($e_i = d_{i+1} - d_i$ ($i = 1, 2, 3$), $f_i = e_{i+1} - e_i$ ($i = 1, 2$) and $g_1 = f_2 - f_1$ were used as inputs. The neural network could classify the training set very well, and generalized reasonably to other scenes as well. At the discontinuities, an open point is created halfway between the edges of the discontinuity. The result is shown in figure 3.6. The working of the used neural network is explained in Appendix B.

5.2 Creating additional areas

The open points of an area are candidates for the centers of adjacent areas. A new area will be created at an open point unless that point is already part of another area reachable from the current area, and vice versa. Another area is reachable if the agent can move there in an almost straight line (using the obstacle avoidance algorithm described in section 4). If the other area is reachable, the open point is removed and an accessibility link to the other area is created (if one was not already present). Otherwise, the agent moves to the open point, which then becomes the center of a new area. When there are no remaining open points in all the areas, a map of the entire environment has been created. The entire map can be displayed to a user by drawing the borders of all the areas simultaneously. However, due to rounding errors, obstacles that are visible in multiple areas may not be perfectly aligned in the combined map (see figure 3.7).

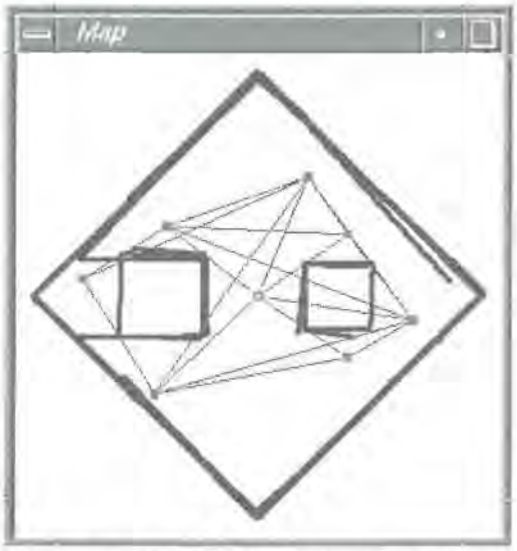


Figure 3.7: A completed map. Thick lines represent obstacles, the centers of the areas are represented by the squares, and a line indicates that two areas are reachable from each other.

In some cases, it is possible that a discontinuity is not detected (this can be seen in figure 3.7 where the left inner square appears connected to the outer square). This can happen when the agent looks at an obstacle at a very sharp angle. In such a case, because of the discrete nature of the vision buffer, it can be impossible to differentiate between a wall viewed from a sharp angle and a discontinuity. This is, however, not a serious problem because the discontinuity can often be observed from another area that has a better view of it, and will be detected in that area.

When the neural network for discontinuities is changed to detect more open points, the opposite can happen. In this case, open points that are located in an obstacle will be detected. These false open points will be identified when the agent tries to move towards the point, and discovers that the point is located inside a wall. When this happens, the agent will remove the false open point and move towards the next open point.

Some improvement for optimizing the neural network is still possible, but it will always be impossible to correctly detect every discontinuity in the environment due to the finite resolution of the sensors. It is therefore necessary that the agent can recover from errors: when the agent is moving towards an open point, but is blocked by an obstacle in the environment, the open point is discarded. This will be detected by the low-level collision avoidance.

The map construction algorithm can be used in two ways. The first way is using it to build a map of the entire environment. This is done by running the algorithm until all the open points are removed. The other way to use it is for moving the agent towards a high-level destination in an unexplored environment. To accomplish this, in every new area, the agent will first move towards the open point that is closest to the destination (instead of the nearest open point). If no open points exist in the area, the agent will move to the area closest to the destination that still has open points. When the destination becomes visible, the agent moves to the destination and the search is complete. An example of this is given in section 0 of this chapter.

5.3 Adding and removing links between areas

When a new area has just been created, all areas whose centers are visible from the new area are immediately connected with the new area. Note that the presence of a link does not represent a path between the areas: it only indicates that the areas are reachable from each other.

As can be seen in figure 3.7, the map construction algorithm creates a large number of links. This number can be reduced by removing redundant links. For example, when areas A, B and C are fully connected (see figure 3.8), and the distance AC is not much smaller than the distance AB + BC, link AC can be removed because C can still be reached from A via B without much overhead. The result of applying this reduction on all the links of the map in figure 3.7 is shown in figure 3.9.

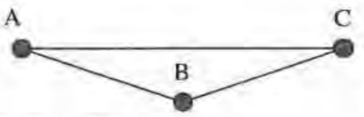


Figure 3.8: Removing redundant links.

6 Path planning

Using the map, it is simple to plan the shortest route between two points in the world. First, the areas containing the begin- and endpoints of the route are located. Then, the A*-algorithm is used to find the shortest path between the centers of these areas.

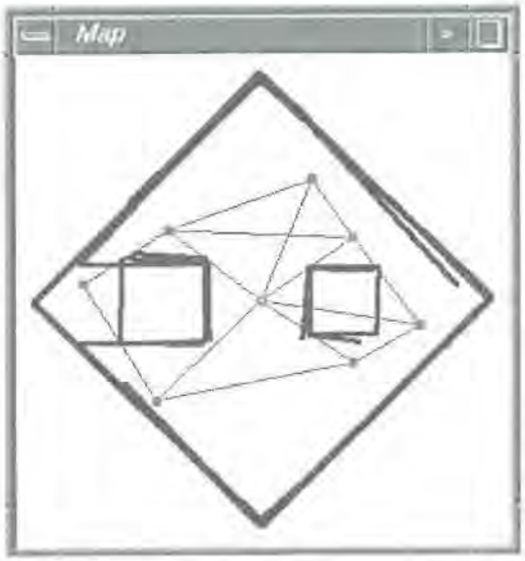


Figure 3.9: Reduced map.

When the path is constructed, the agent can follow it by creating short-term destinations that correspond to the centers of the areas on the path. It is possible to create shortcuts when following this path: as soon as the next point on the path becomes visible, that point replaces the current short-term destination. This way, the agent follows a path that is able to cut some corners.

When a link on the path is blocked, the collision avoidance module notifies the path planner. At that time, the blocked link is marked in the map as temporarily inaccessible and an alternative path is calculated.

7 Results

The algorithm described in this paper is implemented on an SGI InfiniteReality2 running IRIX 6.4, using only a single 195 Mhz MIPS processor to run both the agent and the virtual environment. The virtual environment ICOME [89] was used for the experiments. Constructing the map shown in figure 3.10 took 1702 simulation steps of the agent. The agent was able to complete 72 simulation steps per second, showing that the algorithm is fast enough to work in real time. The code was also ported to a Windows 2000 platform, and run on a Pentium III/500 PC using a NVIDIA RIVA TNT2 Model 64 video adapter, where a similar speed of 75 simulation steps per second was achieved. These frequencies (72 and 75 steps per second respectively) correspond to the refresh rates of the displays used. This means that the highest possible speed was reached, since the virtual sensor (the depth information) can only be updated at this frequency.

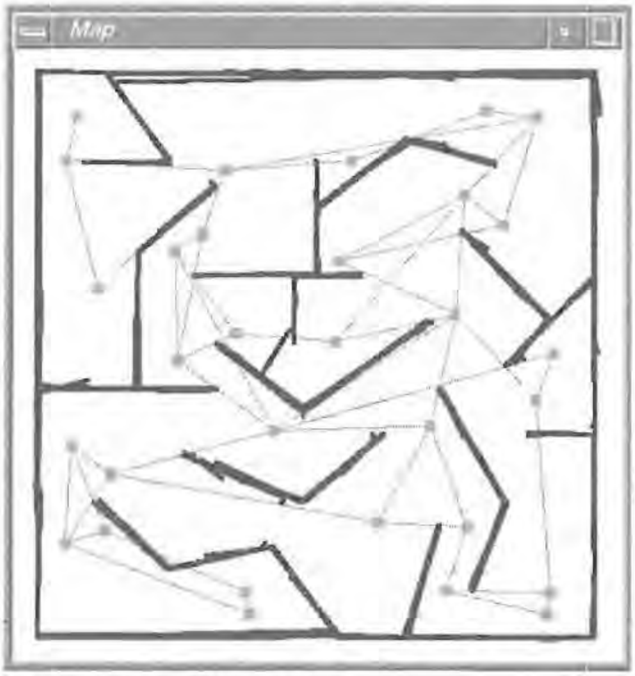


Figure 3.10: A more complex environment.

The same environment was used for moving towards a destination in an unexplored environment. The starting point is located in the lower right corner of the environment, and the destination was placed at the top left corner. The result of the search is shown in figure 3.11, and took 214 steps to complete.

The environment that was used represents a square room that contains 23 wall segments. The wall segments are placed at various angles, and a large number of intersections between corridors are present to complicate the map construction.

An interesting feature was discovered when the obstacle avoidance code was accidentally linked with the avatar of a human user in the environment. The user can move in the environment normally using the keyboard, but when a command of the user would result in a collision with an obstacle, the obstacle avoidance routines would override the user commands and either step around the obstacle or stop the users avatar. This behavior is an example how an agent can pro-actively take action to assist a user.

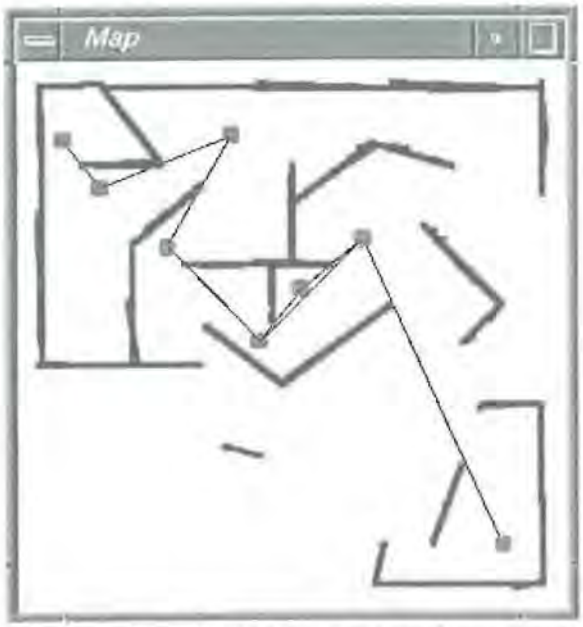


Figure 3.11: Moving towards a destination in an unknown environment from the bottom right corner towards the top left corner of the environment.

8 Conclusion

The algorithm presented in this chapter allows an agent to navigate through a virtual world in real time without colliding with obstacles, and to construct an isometric topological map of the environment using only a virtual depth sensor. The virtual depth sensor is used to scan the distance to objects around the agent. The distance to surrounding objects is then used to prevent collisions with detected objects while moving. The entire environment can be mapped by moving to different areas, and by overlapping the distance information of these areas. The maps created are compact and contain the necessary information for navigation and path planning.

In the next chapter, the collision avoidance techniques described here will be used by virtual soccer players in the Robocup domain. In this domain, the agents that control the soccer players receive distance information about objects on the field, similar to the virtual depth sensor described here.

Chapter 4: Robocup

1 Introduction

The Robot World Cup Initiative (Robocup) is an attempt to create a standard task for AI research on fast-moving multiple robots, which collaborate to solve dynamic problems [43]. The long-term goal of Robocup is to create a soccer team with real robots that play against the current human world champion team. To reach this long-term goal, a more limited version of robot soccer was developed that uses small robots playing on small soccer fields. The robots in this domain have to control their movements, process sensor information, handle communication with each other, and develop multi-agent strategies to play a soccer game. As a result, much focus is placed on the problems that are typical in robotics, and less emphasis is placed on multi-agent strategies. For researchers that prefer to work mainly on the higher-level multi-agent aspects of robot soccer, a software platform was developed in which robots are simulated. As a result, the robotics aspects, such as object recognition, communications and hardware issues are avoided. A graphical representation on the software environment is shown in figure 4.1. In the remainder of this thesis, only the Robocup software simulator will be discussed.

To compare different AI techniques used in Robocup, a yearly competition among teams build by different research groups is organized. In order to make a meaningful comparison between two teams, it is necessary that the environment in which both teams play is standardized. For this purpose, the soccer simulator that was developed handles the playing field, the physics of the world, a sensor model and a referee. Programs controlling the players must connect with the simulator through a fixed interface, and communication with other players is only allowed through the simulator. To design a team of Robocup players, a multi-agent system must be designed, consisting of 11 agents that control the players and process sensor information.

The Robocup domain will be used to test some of the ideas presented in this thesis. First, the obstacle avoidance techniques developed in chapter 3 will be applied to Robocup. Then, in chapter 8 evolutionary computation will be applied to train the behavior of Robocup players.

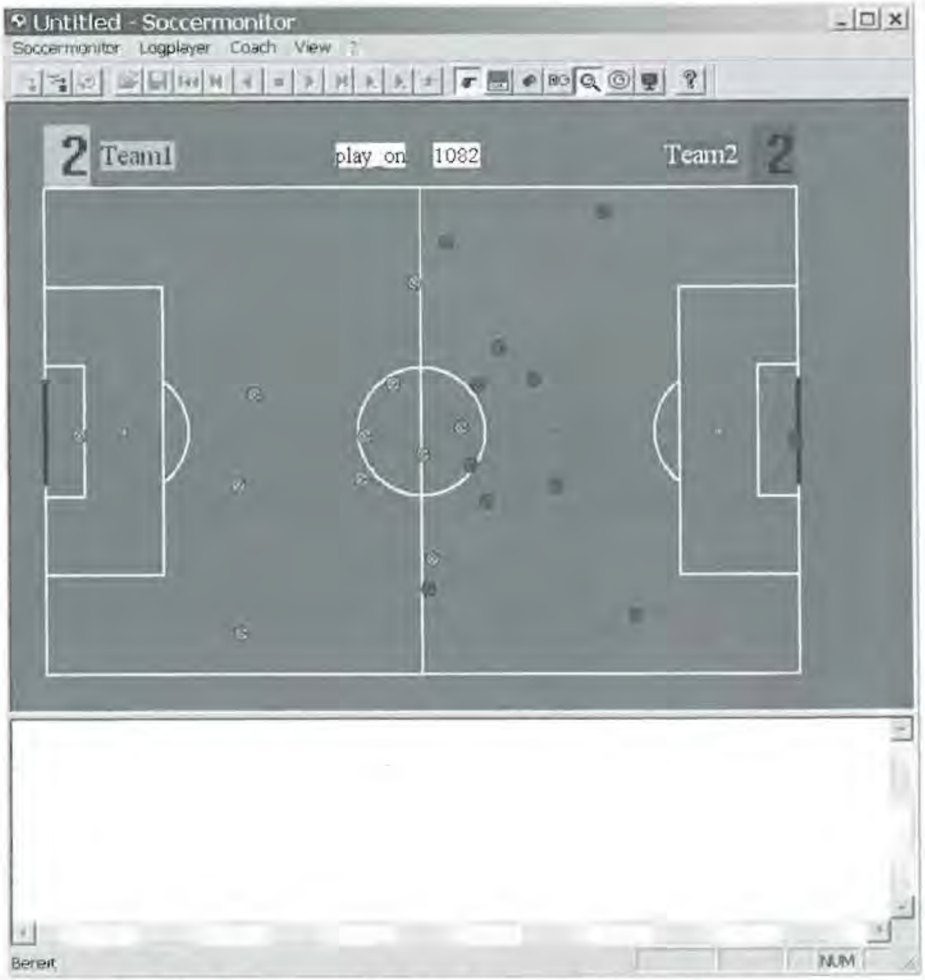


Figure 4.1: Graphical representation of the Robocup software simulator.

1.1 Challenges of Robocup

The Robocup domain provides the following challenges to multi-agent systems:

- The environment is highly dynamic. As a result, short and medium-term plans of the agents must be updated continuously because unexpected events will make previous plans obsolete.
- The perception of each agent is limited. Visual information is restricted to a 90 degrees field of view in the direction the player's head is facing, and the accuracy of the received information depends on the distance of the observed objects. As a result, the agent must be able to handle incomplete and noisy

information and must be fault-tolerant to compensate for unexpected events and errors.

- Communication between agents is limited. Therefore, a trade-off must be made between the cost of communication and their potential benefits. Consequently, communication protocols that minimize communication and promote distributed reasoning must be developed.
- Each player has limited stamina. As a result, players must decide when to perform great efforts and when to recover from these efforts.
- The simulation occurs in real time. The agents must therefore be implemented efficiently, and heuristics often have to be used instead of exact analytic solutions.
- The agents do not know the exact results of the actions they perform. The agent must observe the changes in the environment to determine the result of the executed actions. This leads to the design of verification mechanisms that detect errors in the execution of actions.

1.2 Description of the simulation environment

The soccer simulator program is the central part of the simulation environment. It manages the physical aspects of the objects in the environment: the players, the ball and the field. This includes the movement of ball and players, and the information received from the virtual sensors of the players. The simulator also provides protocols to allow programs to communicate with the simulator over a network connection. These programs include the different players of both teams, but also a monitor program that gives a graphical representation of the soccer field, as well as coach programs. Coaches can either be programs that give limited advice to players during a match, or can be a privileged program that can position objects on the field during the training phase of a team.

The simulator works with discrete time steps of a fixed duration (currently 100 ms). At regular intervals (by default 150 ms), the player receives visual sensor information about the objects that are currently within visual range of the player. Once during every time step, a player agent may send a command to the simulator to control the movement of its player.

The communication between the simulator and the other programs happens with a client/server model using network communication. This is demonstrated graphically in figure 4.2. The communication protocol between the simulator and the player agents will be described in the next section. A more detailed description is given in [12].

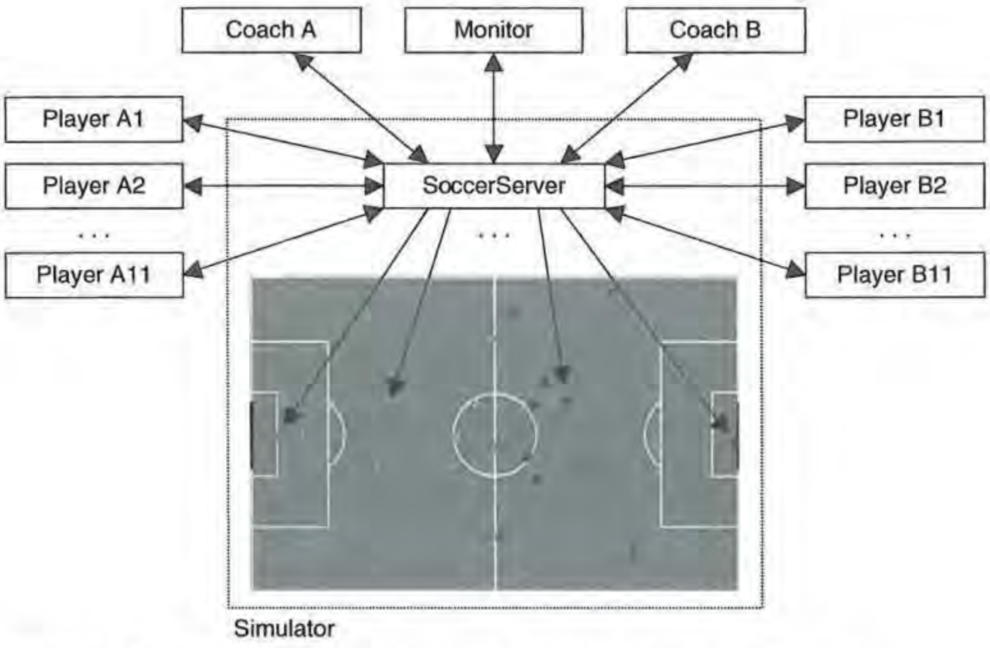


Figure 4.2: Overview of the Robocup soccer simulator client/server model.

1.2.1 Movement commands

The following commands are available to control the movement of a player:

- (**turn** *Moment*): This command rotates the player over an angle *Moment*. If the player is currently moving, *Moment* will be reduced by an amount depending on the current speed of the player. This simulates that turning is more difficult while moving rapidly.
- (**dash** *Power*): This command increases the speed of the player in its current direction by an amount *Power*. When no dash commands are sent to the server, the speed of a player will decrease every time step. If *Power* is negative, the player will slow down or move backwards.
- (**kick** *Power Direction*): If the player is within range of the ball, this command will kick the ball in the specified *Direction* with the given *Power*.
- (**move** *X Y*): This command can only be used before a kick off to immediately place the player at a given position (*X, Y*) on the field.
- (**catch** *Direction*): This command can only be used by the goalie player to catch the ball, if within range. The *Direction* towards the position of the ball must be specified.

1.2.2 Sensor commands

A player can send the following commands to the simulator to control its sensors. These commands can be sent in addition to a movement command:

- (**change_view** *AngleWidth Quality*): This command changes the viewing angle of the visual sensor and the precision of the received information. Larger viewing angles and higher quality increase the interval by which sensor information is send to the players. The default viewing angle is 90 degrees, and the default quality is 'normal'.
- (**say** *Message*): This command sends an ASCII string *Message* to all the players near the sending player.
- (**turn_neck** *Angle*): The visual sensor of a player does not have to face in the same direction as the body of a player. The angle of the visual sensor relative to the angle of the body can be modified with this command. This simulates turning the neck of a player.

The simulator sends the following sensor information to the players:

- (**see** *Time ObjectInfo**): This message contains a symbolic representation of the objects that are visible by the player at the specified *Time*. Visible objects include other players and the ball, but also fixed reference points on the field like goals, lines and 'flags' placed at various places. The visual information about the fixed reference points can be used to estimate the absolute position of the player on the field. *ObjectInfo** is a list of information about visible objects. This information contains the name of the object, the distance and the direction to the object. Mobile objects also contain the change in distance and direction since the previous observation, and the direction a player is facing. If the object is far away from the player, some of this information may be omitted.
- (**hear** *Time Sender Message*): This message contains an ASCII *Message* that was send by another player through a say message. If multiple say messages are received by a player, only one randomly selected message will be sent to the player by the simulator.
- (**sense_body** *Time PlayerInfo*): This message is transmitted every time step to the players and contains information about the current state of the player. This information contains, among other information, the current stamina level, speed and head angle of the player.

2 Architecture of an agent's brain

2.1 Layered behaviors

Because of the complexity of the robotic soccer domain, it is impossible to construct a solution directly from the available communication primitives. It is therefore necessary to decompose the task in several less complicated tasks. In [102], Stone and Veloso introduce the concept of layered learning. This approach splits the problem in several layers of complexity. The tasks in every layer can then be trained by separate learning techniques. At the lowest level, a number of skills are build from the basic commands of the communication protocol. These skills are primitive actions like intercepting the ball, dribbling or shooting the ball to a position. Using these primitive skills, higher level skills can be constructed. The different layers of learning Robocup are shown in table 4.1. The individual player skills will be described in the next section. One-to-one and one-to-many player skills will be discussed in sections 3.1 and 3.2 of this chapter. Finally, action selection will be learned using evolutionary techniques, which will be discussed in chapter 8.

Layered level	Examples
Individual player skills	Intercept, MoveTo
One-to-one player skills	PassTo
One-to-many player skills	Avoiding other players
Action selection	Pass or dribble or shoot?
Team collaboration	1-2 combination, positioning

Table 4.1: Different layers of behaviors.

2.2 Individual player skills

In [17], Cossement implemented the following individual player skills:

- Estimating the position of objects based on previous observations: Given the speed and position of an object at a previous time, an estimate of the current position can be made. This is useful because sensor information is not received every time step, and objects behind the player are not visible.

- Moving to a position on the field: This skill will execute a number of turn and dash commands to move to the specified position.
- Moving to the ball: This skill is similar to the previous skill, moving to the current position of the ball.
- Marking a player: This skill will move the player near an opponent, hindering the opponent's ability to receive or pass the ball.
- Turning the player: This skill turns the player over a specified angle, taking into account the current speed of the player. At higher speeds, it is more difficult to turn.
- Shooting the ball to the goal: This skill kicks the ball to the goal at the highest possible speed.
- Passing the ball to another player: The ball will be kicked to the position where the other player is estimated to be after several time steps.
- Running with the ball: This skill will combine kick, dash and turn commands to move with the ball to a specified position.
- Dribbling: This skill is similar to running with the ball, but will also try to avoid opponents.
- Intercepting the ball: This skill moves to a position where the ball is estimated to be at the time the player can be there as well.

3 Obstacle avoidance in Robocup

Because the simulated soccer field does not contain any static obstacles, there is no need to use the map construction techniques discussed in chapter 3. However, the short and medium range collision avoidance techniques discussed there are still useful in the Robocup domain. The sensor information received from the simulator is similar to the synthetic vision used in chapter 3, since only the objects in a 90 degrees angle in front of a player are visible and their depth information is available. This information can then be stored in a vision buffer, as shown in figure 4.3. The positions of objects that are not in front of the player are estimated based on their last known position and speed. When these objects were observed in the recent past, these predictions are relatively accurate and the objects can be added to the vision buffer. The vision buffer can then be used to avoid collisions with other players, and also to keep some distance from opposing players who will try to steal the ball. The vision buffer can also be used to determine the best direction for passing a ball, selecting a path that keeps away from opposing players. These two forms of obstacle avoidance will be discussed next.

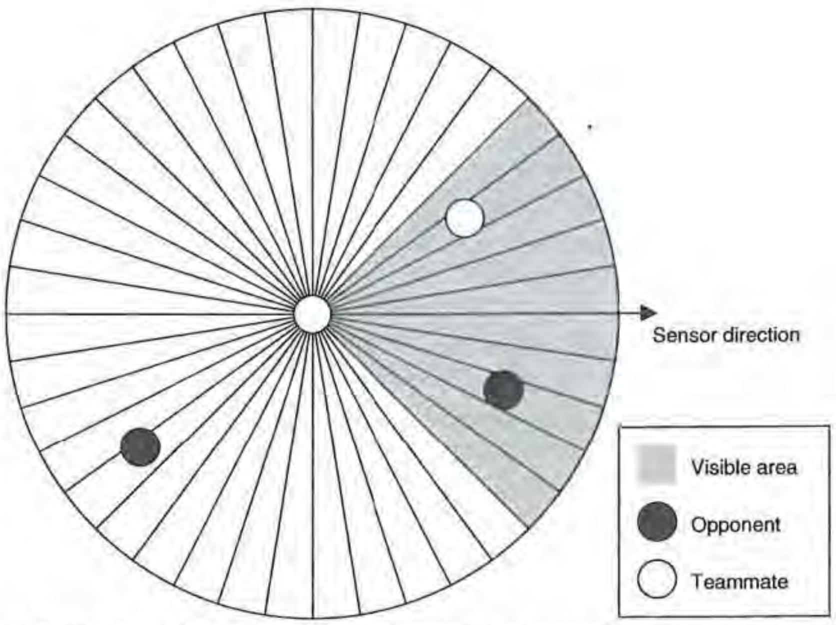


Figure 4.3: The positions of visible and recently observed players are stored in the vision buffer of a player.

3.1 Determining safe directions for movement

When a player is observed and placed in one or more segments of the vision buffer, those segments are considered blocked from the position of the player onward. However, because players are mobile objects whose movements are difficult to predict, adjacent segments are marked as blocked as well. To account for the uncertainty of the future positions of players, a danger value is associated with blocked segments depending on the chance that the player will be at that segment after several time steps. The segment that a player currently occupies is given the highest danger value, and adjacent segments are assigned increasingly lower danger values. If a segment is given a danger value because of multiple players, the danger values will be added together. Since teammates are less likely to disrupt the movement of the player, less segments are marked as dangerous around teammates, and the danger values are lower as well. This is demonstrated in figure 4.4.

When the segments surrounding all the players are marked, a safe movement direction can be selected. The segment closest to the target direction that has no danger value is selected as the safest direction. If the player is surrounded by other players, it is possible that all the segments in the vision buffer are marked as dangerous. In this case, the segment with the lowest danger value can be selected as the movement direction, but the action selection module of the player may decide that movement is not a good option and select an alternative to moving instead.

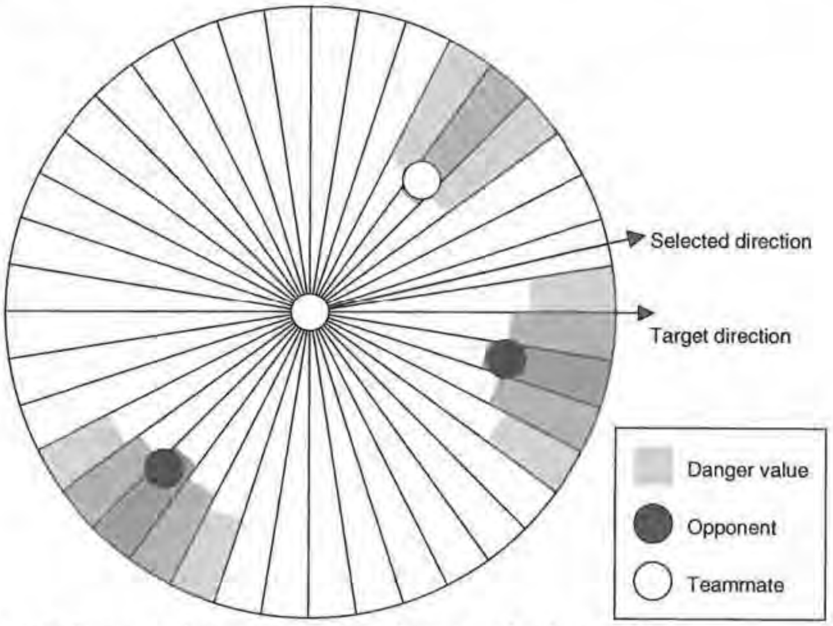


Figure 4.4: Danger values surrounding players and selected direction.

3.2 Determining safe directions for passing the ball

The objective of a pass is to kick the ball towards a teammate, who then controls the ball, without losing control of the ball to an opponent. Finding an optimal direction for passing can be accomplished in a similar way as finding a safe movement direction. The segments containing opponents are given a danger value, as are their surrounding segments. On the other hand, segments containing teammates are given safety values, as are their surrounding segments. If a segment contains both a danger value and a safety value, the two values are subtracted from each other, considering the safety value as a negative danger value. The segment containing the highest safety value is then selected as the direction for the pass. This is demonstrated in figure 4.5, where a pass is not given directly to a teammate but to the direction of the segment next to him, because an opponent is nearby.

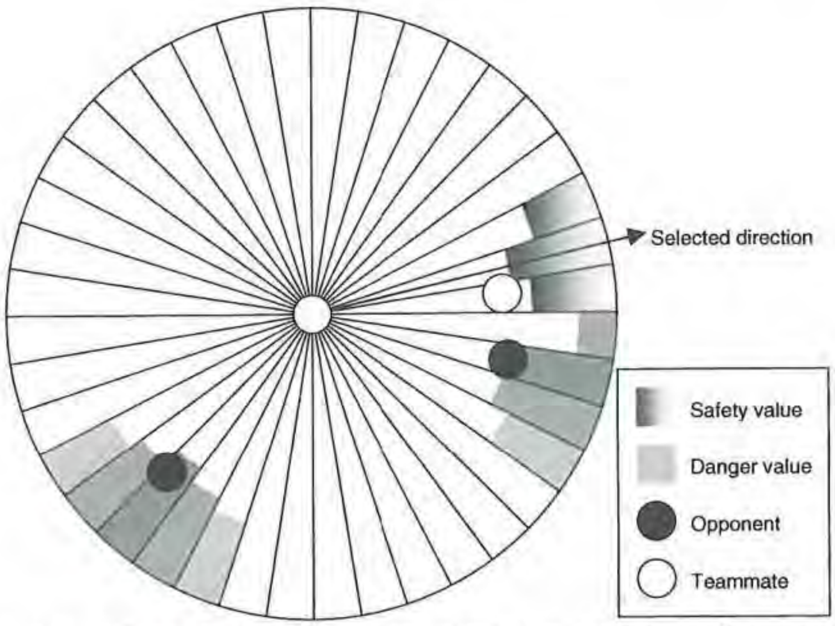


Figure 4.5: Danger and safety values around opponents and teammates to determine the safest passing direction.

4 Teamwork in Robocup

The highest layer of behavior presented in table 4.1 is team collaboration. These behaviors ensure that the individual players work together as a team with the same team goal. However, it is not sufficient to supply every member of the team with a fixed pre-planned strategy, since unforeseen events can disrupt this strategy. Therefore, it must be possible to modify the team plan during its execution.

	Joint intentions framework	Locker-room agreements
Communication cost	High	Low
Applicability	High	Medium
Complexity	High	Medium

Table 4.2: Comparison of teamwork strategies.

In this section, the following teamwork strategies will be discussed briefly:

- The joint intentions framework.

- Locker-room agreements.

These methods have different advantages and disadvantages concerning the amount of communication needed, the domains where the technique can be applied, and the complexity of implementing the technique. These advantages and disadvantages are summarized in table 4.2.

4.1 The joint intentions framework

The joint intentions framework was developed by Cohen en Levesque in [14][55] and was also used by Tambe in [104]. In this framework, a team Θ jointly intends a team action when all members are jointly committed to completing this team action, while mutually believing that they are doing it. A joint commitment in turn is defined as a joint persistent goal (JPG). A JPG to achieve *p*, where *p* stands for completion of a team action, is denoted as JPG(Θ, *p*).

JPG(Θ, *p*) holds if three conditions are satisfied:

- All team members mutually believe that *p* is currently false.
- All team members mutually know that they want *p* to be eventually true.
- All team members mutually believe that until *p* is mutually known to be achieved, unachievable or irrelevant, each holds *p* as a weak goal (WG). WG(μ, *p*, Θ), where μ is a team member in Θ, implies that μ either:
    - Believes *p* is currently false and wants it to be eventually true.
    - Having privately discovered *p* to be achieved, unachievable or irrelevant, μ has committed to having this private belief become Θ's mutual belief.

This model of teamwork requires that the agents communicate with each other to share their beliefs about the current team actions, to prevent other agents from performing actions that are no longer relevant or even interfere with the current team goal. However, communication in Robocup is expensive and is not guaranteed to be successful. Therefore, the unconditional commitment to communicate a change in a team's activities is modified to be conditional on communication benefits to the team outweighing costs to the team.

It is not always necessary to communicate a change in a player's belief. First, it is checked if the other team members require the new information to complete the team goal. If this is not the case, no communication is necessary. Secondly, a check is made to test if other team members are already aware of the new information. If this is true, there is no need for communication.

4.2 Locker-room agreements

This technique was used by Stone and Veloso for their CMUnited Robocup teams [102]. Since the Robocup players have only a limited amount of bandwidth available for communication, it is useful to agree to a team strategy when the team is able to synchronize privately. This can happen for example before play begins and during half time of a game. These team strategies are called locker-room agreements. During the game, the players can keep track of the state of world using its sensors, the player's internal state, and the locker-room agreements made.

By giving every player a predetermined set of team actions, it is possible that the team will not be flexible or robust to failure. This can happen when the locker-room agreement divides the team goal into several rigid roles, and assigns one player to every role. The team is then inflexible, both to short-term changes like the unavailability of a player, and to long-term changes such as an ejected player. Reassigning a task to a different player will then be difficult because of the limited communication that is available.

To increase the flexibility of a team, players can switch to a different set of behaviors when certain sensory triggers are received. For example, the team can switch from a defensive setup to a more offensive setup when the team is losing. This switch can be triggered when the opposing team leads by two goals. Also, predefined multi-agent plans can be specified for some frequently occurring situations such as free kicks. The positions of players are also specified in the locker-room agreement. These include the area on the field that is the 'home'-position of the player, and the role of the player such as attacker or defender. These positions can change because of sensor triggers.

5 Conclusion

This chapter has introduced Robocup as a virtual multi-agent system that simulates robotic soccer. In this systems, virtual agents control individual soccer players. To construct behavior for these agents, several layers of behaviors are used. The lowest layer implements individual player skills, which are mostly manually coded. The next layers implement obstacle avoidance and passing the ball, using the vision buffer techniques described in chapter 3. The next layer implements action selection, and will be developed using evolutionary computation. This technique, and its application to Robocup, will be described in Chapter 4 of this thesis.

Part II: Genetic programming

Chapter 5: Genetic programming

1 Introduction

This chapter introduces genetic programming as a general problem solving technique. Genetic programming is very different from traditional AI problem solving methods. Traditional AI uses computational models and requires a large amount of task specific knowledge about the problem to be able to solve it. These are called strong AI methods, and while they may be able to rapidly find an exact solution for a problem, collecting sufficient task specific knowledge may not be easy. Also, strong AI methods often have problems with real-world problems where unpredicted events cause the problem solving method to fail completely. A typical example of traditional strong AI methods is a knowledge base containing thousands of task specific rules. Weak AI methods on the other hand rely less on built-in task specific knowledge. Instead, knowledge about the environment is added when it is identified through experimentation. Therefore, these methods are more robust and will work on a wider variety of problems, but finding a solution may take longer compared to a strong AI method.

Genetic programming is an evolutionary weak AI method inspired by natural evolution. Task specific knowledge is gathered by testing the quality of a set of candidate solutions. New candidate solutions are generated by combining the features of the best solutions. Initially, no knowledge about the environment is available and the candidate solutions are generated randomly.

This chapter will tackle genetic programming in section 4, preceded by introducing the related techniques of evolutionary computation in section 2 and genetic algorithms in section 3. Several typical genetic programming problems will be discussed in section 5, and finally some often used extensions to genetic programming are given in section 6.

2 Evolutionary computation

Evolutionary computation is an optimization technique inspired by the process of natural evolution, as described by Charles Darwin [18] in the 19th century. This process is often called "survival of the fittest". In natural evolution, a species consists of a population of individuals. In this population, individuals that are better adapted to their environment (i.e. fitter individuals) have a larger chance to survive long enough to produce offspring. These fitter individuals pass their genetic material on to their offspring. On the other hand, the less fit individuals either die before they can reproduce, or reproduce less often. As a result, offspring will have a higher chance to have their fitter parents' properties that caused them to be better adapted to the environment. Thus, natural selection causes the average fitness of the individuals in the population to increase, as new individuals are more likely to have an above average fitness.

The concept of evolution can also be applied to optimization problems in computer science. The basic idea is to maintain a population of candidate solutions for a problem that compete among each other for the chance to reproduce. The quality of these candidate solutions is measured and a fitness value is awarded accordingly. This fitness value will then determine the probability by which an individual will be selected to survive in the population and/or produce offspring. This process will be described in section 2.1 and 2.2. Initially, the algorithm uses a population of randomly generated candidate solutions. In every subsequent step of the algorithm, called a generation, a new population will be created from the existing population. When an acceptable solution is encountered, or when a predefined number of generations are completed, the algorithm ends.

2.1 Selection

The fitness of a solution determines the probability that it will be selected to survive and produce offspring. Several selection methods are possible, and have different advantages and disadvantages [86]. The most frequently used techniques are presented below.

2.1.1 Fitness proportionate selection

This method is also referred to as roulette wheel selection, and is the most frequently used selection technique. With this method, the selection probability of an individual is directly related to its fitness value. If f_i is the fitness value of individual *i*, and the population consists of *N* individuals, the average fitness of the individuals in the population is defined by equation (5.1):

$$\bar{f} = \tfrac{1}{N} \sum_j f_j \tag{5.1}$$

The selection probability p_i of individual *i* is then given by equation (5.2):

$$p_i = \frac{f_i}{\sum_j f_j} = \frac{f_i}{\bar{f} \cdot N} \tag{5.2}$$

Advantage:

- The method is biologically plausible.

Disadvantages:

- Premature convergence: If an individual in an early generation has a very high fitness compared to the average fitness of the population, this individual will be overselected and copies of this individual will occupy the entire population after several generations. At this point, new individuals can only be introduced through mutation.
- Stagnation: After several generations, when all individuals have fitness values that are close to each other, there is very small selection pressure, and it will be difficult to discriminate good solutions from slightly worse ones. However, this problem can be solved by scaling the fitness values of the population to the interval between the lowest and highest fitness values.

2.1.2 Rank selection

Rank selection is used to overcome the disadvantages of fitness proportionate selection. This selection method assigns a selection probability to an individual based on that individual's rank in the current population. To determine the rank of an individual, the *N* members of a population are sorted according to their fitness value. The best individual will have a rank of 0, the worst will have a rank of $N - 1$. The selection probability of an individual with rank *i* can then be given by a linear function ($p_i = a*i + b$) or by a negative exponential function ($p_i = a*\exp(b*i + c)$). The constants *a*, *b* and *c* must be chosen so the sum of all probabilities will be 1.

Advantages:

- No premature convergence.
- No stagnation
- The explicit fitness value of the individuals is not needed. Consequently, it is sufficient to compare the result of individuals.

Disadvantages:

- Overhead of sorting the members of the population.
- Theoretical analysis of convergence is difficult.
- Biologically not very plausible.

2.1.3 Tournament selection

When using tournament selection to select an individual, a small group of N ($N > 1$) individuals is selected uniformly from the population. The individual in the group with the highest fitness value is selected, while the rest is ignored. This method behaves like a noisy version of rank selection.

Advantages:

- No premature convergence.
- No stagnation.
- The explicit fitness value of the individuals is not needed.
- No overhead needed to sort the population.
- Naturally inspired.

Disadvantage:

- Noise.

2.1.4 Elitist selection

When elitist selection is used, one or more of the best solutions of a generation are automatically transferred to the next population. This selection method is used in combination with one of the selection methods described before.

Advantage:

- Convergence: if the global maximum is discovered, the search will converge towards it.

Disadvantage:

- Risk of getting trapped in a local optimum.

2.2 Reproduction

Reproduction is an operator that is used to generate new candidate solution(s), called offspring, from existing solutions, called the parent(s). During reproduction, the genetic material from the parent(s) is somehow used to create the genetic material of the offspring in an attempt to transfer the good qualities of the parent(s) to the offspring.

Two types of reproduction are possible: sexual and asexual reproduction. With asexual reproduction, a single parent is used to create offspring. Asexual reproduction is also observed in single cell life forms like bacteria, where the genetic material is copied during a cell division. Small errors can occur during this duplication process, which may lead to improvements. Sexual reproduction requires two parents, and the genetic material from both parents is combined to form the genetic material of the offspring. This method attempts to combine the favorable properties of both parents to create a better individual. In nature, sexual reproduction is typically encountered in higher life forms.

2.3 Mutation

A fundamental property of evolutionary algorithms is that a diverse population of different candidate solutions is maintained. Because of this, the search space is examined in several places simultaneously. New places in the search space are examined by combining solutions at different locations in the search space. As a result, when all the individuals in the population resemble each other, the potential to sample the search space is reduced and it will be difficult to find new solutions. Mutation can be used to introduce diversity in the population by randomly making small modifications to the genetic structure of some individuals. As a result, mutation allows the evolutionary algorithm to escape from local optima in the search space by re-introducing lost genetic material in the population.

2.4 Basic evolutionary algorithm

The basic evolutionary algorithm is given below:

```
// Initialization:
Gen = 0
Create a random initial population P(Gen)
Evaluate the population P(Gen) and assign fitness
While the termination criterium is not satisfied:
   // Select the parents for the next generation:
   P2 = SelectParents(P(Gen))
   // Create the next population:
   Gen = Gen + 1
   P(Gen) = Reproduce(P2)
   Apply mutation to P(Gen)

   Evaluate the population P(Gen) and assign fitness
End while
```

3 Genetic algorithms

Genetic algorithms are a type of evolutionary computation that was originally devised by John Holland [34]. The candidate solutions of genetic algorithms are typically represented by fixed length character strings. The characters of these strings are often the binary numbers 0 and 1, but other characters, such as a range of integer numbers or a set of symbols are also possible. The character string, the genotype of the solution, can be decoded to the actual solution, called the phenotype. The basic genetic algorithm is identical to the basic evolutionary algorithm described in section 2.4, where the reproduction operator is usually crossover. This operator will be explained in section 3.3.1.

3.1 Example

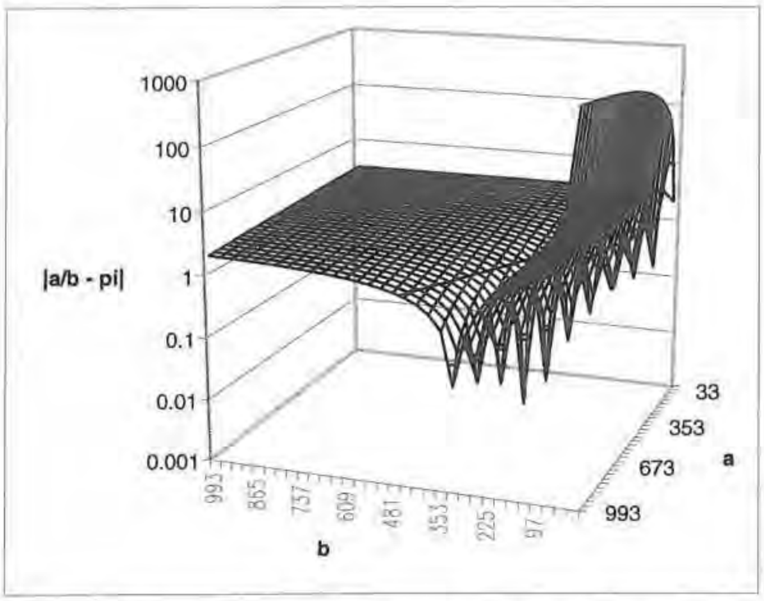


Figure 5.1: Fitness landscape of a rational approximation of the number pi.

As an example for genetic algorithms, consider the problem of finding the best rational approximation of the number π [86]. A candidate solution will consist of two integer numbers *a* and *b*. The objective is to find a pair of numbers that minimizes the equation $|a/b - \pi|$, which gives a fitness value for every candidate solution (a_i, b_i). When the fitness value is calculated for all values of the search space, the fitness landscape is obtained. A graph representing the fitness landscape of this problem is shown in figure 5.1.

3.2 Representation

When genetic algorithms are used to find a solution for a problem, it is necessary to construct a representation that encodes candidate solutions as a fixed length string. A representation of the rational approximation of pi needs to encode two integer numbers. If the range of these two integers is restricted to the range {0, 1, 2, ..., 1023}, a candidate solution can be encoded using a 20 bit string, where the first 10 bits represent the value *a*, and the last 10 bits represent the value *b*. These numbers can be represented using the standard binary encoding of integers, or a more complex encoding such as Gray codes can be used. The use of Gray codes has the advantage that small mutations in the genotype will cause only small changes in the phenotype. An example of standard binary encoding is shown in figure 5.2, where the number *a*, with a value of 311, is represented by the first ten bits and the value *b*, with a value of 157, is represented by the last ten bits, shown in italic font. This genotype represents the value $v = 1.98089...$ (a/b) and has a fitness value of $1.16070...$ $(|v - \pi|)$.

0	1	0	0	1	1	0	1	1	1	*0*	*0*	*1*	*0*	*0*	*1*	*1*	*1*	*0*	*1*

Figure 5.2: Binary representation of the two 10-bit integers 311 and 157.

3.3 Genetic operators

The genetic operators are responsible for the creation of new individuals from the individuals in the current population. In genetic algorithms, the crossover operator is typically used for sexual reproduction, and the cloning and mutation operators are used for asexual reproduction. The cloning operator simply copies an individual to the next generation. The other operators are discussed below.

3.3.1 Crossover

This form of sexual reproduction selects two individuals from the current population and exchanges genetic material between them to form two new individuals. In the case of one-point crossover, one crossover point is selected in both character strings, and the genetic material at one side of this crossover point is exchanged between the individuals. This is demonstrated in figure 5.3.

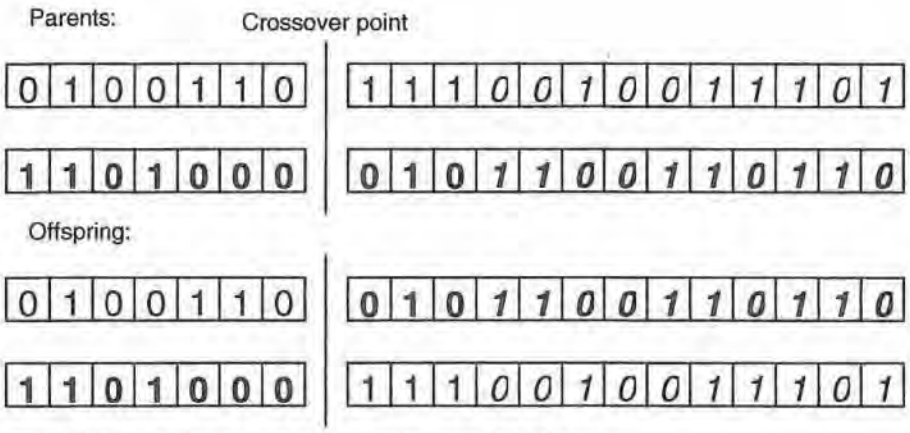


Figure 5.3: One-point crossover between two individuals.

The disadvantage of single-point crossover is that the two pieces of genetic material at both ends of the character string can not both be transferred to the new individual, because they will always be separated by the crossover point. This can be solved by selecting two or more crossover points. Two-point crossover is demonstrated in figure 5.4.

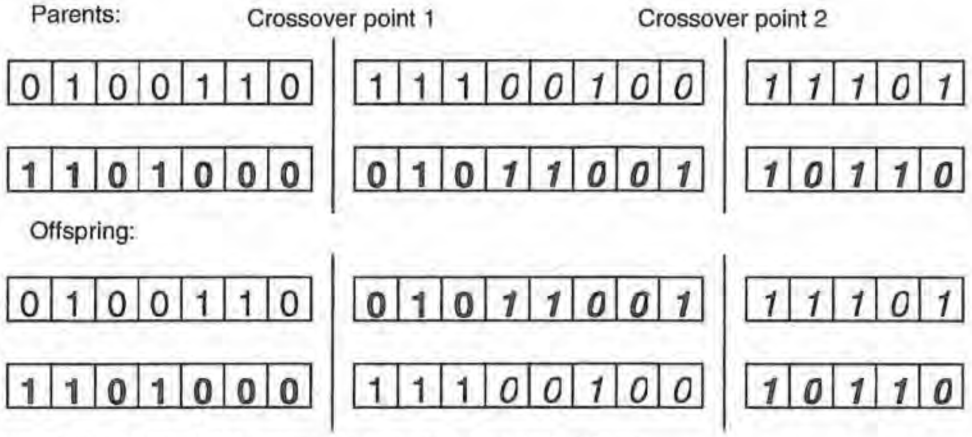


Figure 5.4: Two-point crossover between two individuals.

3.3.2 Mutation

Mutation is a form of asexual reproduction that randomly modifies one or more characters in the character string. This operator is necessary to maintain diversity in a population. Diversity can be lost when not all possible characters exist at a given position in the representation over all individuals in the population.

Because crossover can only exchange the characters that are present at that position, in this case it will be impossible to retrieve these lost characters. Mutation is then the only way to reintroduce lost characters. Mutation is demonstrated in figure 5.5.

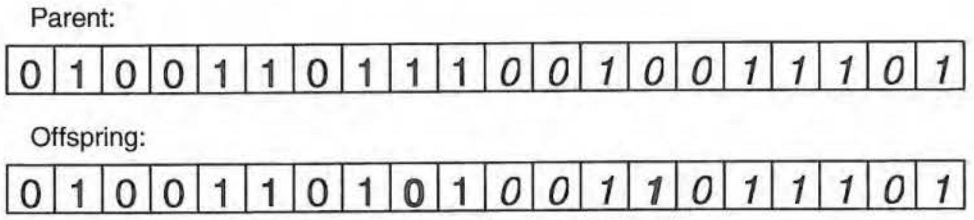


Figure 5.5: Example of two random mutations on a bit string.

4 Genetic programming

Genetic algorithms are an evolutionary search technique whose candidate solutions represent the solution for a single problem. Genetic programming is an extension to genetic algorithms, where the candidate solutions are programs that, when executed, solve a class of problems. While the underlying evolutionary algorithm is identical to the one described in section 2.4, the representation of candidate solutions and genetic operators differ from those used in genetic algorithms. The selection methods, however, are still identical to those described in section 2.1.

4.1 Representation

Computer programs typically have a variable length. Consequently, programs in genetic programming are typically represented by trees. For example, the expression ((3+x)*5) can be represented by the tree shown in figure 5.6.

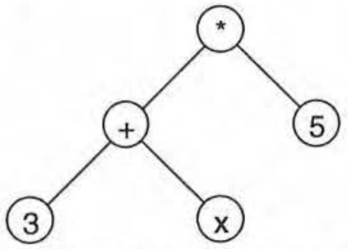


*Figure 5.6: Tree representation of the expression (3+x)*5.*

When a representation is chosen to solve a problem using genetic programming, it is necessary to choose appropriate sets of terminal and non-terminal nodes. Some examples of terminal and non-terminal sets are presented below.

4.1.1 Non-terminal sets

The set of non-terminal nodes contains operators that have one or more other nodes as children. Some typical non-terminal nodes are given below, with the arity of the operators specified in subscript.

- Arithmetic: $+_2$, $-_2$, $*_2$, $/_2$, $-_1$, ...
- Mathematical: sin_1, cos_1, exp_1, ...
- Boolean: and_2, or_2, not_1, xor_2, ...
- Conditional: if-then-else_3
- Looping: for_4, while_2, ...

4.1.2 Terminal sets

The set of terminal nodes contains all nodes that don't have any child nodes. Typical terminal nodes are given below.

- Variables: x, y, ...
- Constants: 1, -3, 3.1415, ...
- Functions without arguments: rand(), move-forward(), ...

4.2 Genetic operators

Because genetic programming uses a tree representation to represent the candidate solutions of a problem, the genetic operators have to be modified to work with trees. The operators that are most frequently used are tree-based versions of single-point crossover and mutation. A new operator called combination will also be presented in this section.

4.2.1 Crossover

As in genetic algorithms, the tree-based crossover operator is a sexual combination operator that exchanges genetic material between two parent individuals. Single-point tree-based crossover selects a link between two nodes in both individuals as the crossover points. The subtrees under the crossover points are then exchanged between the two individuals to produce two new individuals.

This operator is demonstrated in figure 5.7. The size of offspring is usually different from the size of their parents.

Other forms of crossover are also possible. It is easy to implement multi-point crossover, where several crossover points are selected in both individuals. Yet another variation is homologous crossover, where the crossover point is selected at an identical position in both individuals. This causes the genetic code to be moved to a similar position in the offspring.

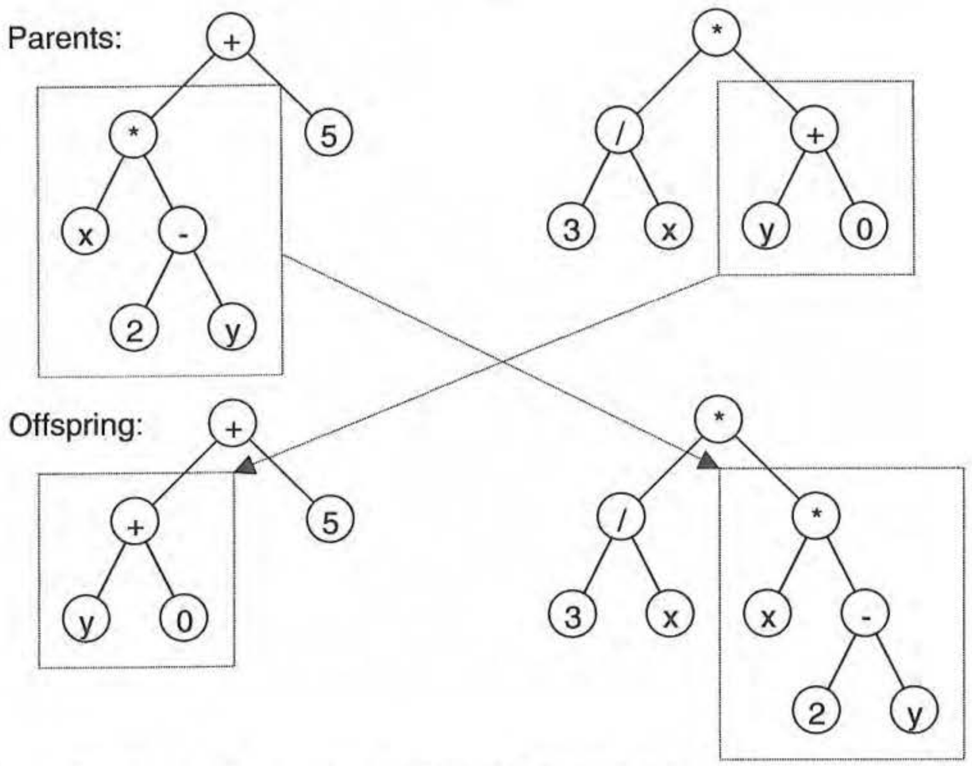
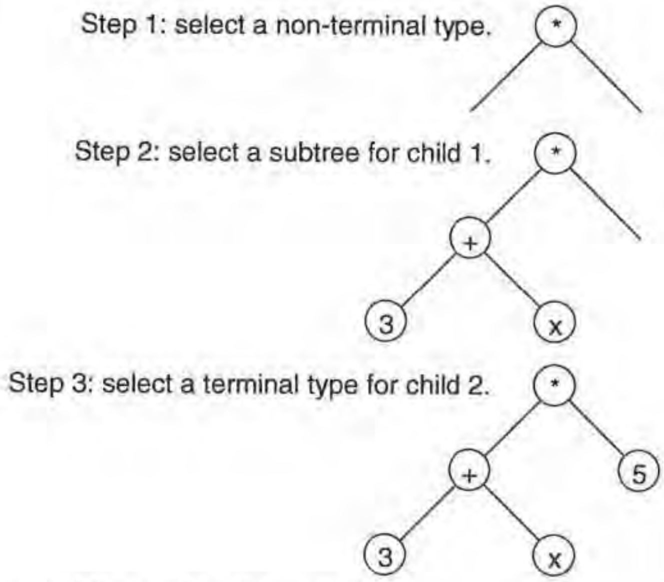


Figure 5.7: Single-point tree-based crossover between the expressions ((x(2-y))+5) and ((3/x)*(y+0)).*

4.2.2 Combination

Combination is an asexual genetic combination operator that selects a root node for a new individual by randomly selecting a node from the non-terminal set. The children of the root node are then added using selection, crossover, or by recursively applying combination. When selection is used to select a child node, it is possible to either select an entire individual from the population or only select a subtree of an existing individual. In the recursive application of the combination operator, nodes can also be selected from the terminal set. This operator can be useful in problem domains where concatenating partial solutions has a good chance to produce a better individual. This operator is demonstrated in figure 5.8.


Step 1: select a non-terminal type.
Step 2: select a subtree for child 1.
Step 3: select a terminal type for child 2.
*
+
3
x
5
Figure 5.8: Example of the combination operator.

4.2.3 Mutation

Mutation is an asexual combination operator that modifies a single individual. Several types of tree-based mutation exist, and result in varying degrees of change in the new individual. The least destructive form of mutation selects a single node in an individual, and replaces it by a different node that is compatible with the replaced node. A more destructive form of mutation will select a node from an individual and replace it with a randomly generated subtree. The most destructive form of mutation, called headless chicken crossover, generates a new random individual and applies crossover between the selected and the random individual.

4.3 Closure

Closure in genetic programming means that every non-terminal node must be able to work with the result provided by any possible child node. For example, the non-terminal node '+' requires two values as its child nodes to calculate its result. If one of the child nodes is the subtree (2/0), the result will be undefined. The terminal and non-terminal set must satisfy the closure property to ensure that all programs generated by the genetic operators are syntactically correct. It may therefore be necessary to modify the terminal and/or non-terminal set slightly to ensure that only valid results are generated. For the example given above, it will be necessary to modify the division operator to handle division by zero and instead return a default value such as 0.

If all the operators in the terminal and non-terminal sets are replaced by protected versions that satisfy the closure property, syntactical correctness is ensured if the return values of every operator have an identical type. Unfortunately, this is only possible for simple problems. When nodes of different types are present in the terminal and/or non-terminal set, the genetic operators will have to be modified to ensure syntactically correct individuals are generated. In section 6.1, strongly typed genetic programming will be presented as a way to handle multiple types.

5 Example problems

This section will introduce several of the toy problems that are typically used to test or demonstrate the usefulness of genetic programming. Most of these problems are described by Koza in [44].

5.1 Symbolic regression

Symbolic regression attempts to find a symbolic representation of a function, given a set of data points of this function. There are four requirements to solve a symbolic regression problem with genetic programming:

- A set of data points must be available.
- The dependent variable of the data points must be selected.
- A fitness function must be defined that calculates the quality of a candidate solution. Usually this means that a candidate solution is evaluated over all the points in the data set, and the result is compared with the dependant variable of the data set. The sum of all the errors over all data points can then be used as a fitness value.
- The terminal and non-terminal set must be selected. The terminal set must at least contain the independent variables of the data set. The non-terminal set must contain sufficient operators to solve the problem. For example, it will be impossible to find a correct solution for data points from the function log(1 + x) when the terminal set only contains the functions {+, -}.

Simple functions that are often used to test the performance of genetic programming systems are $x^4 + x^3 + x^2 + x$, $x^5 - 3x^3 + x$, and $x^6 - 3x^4 + x^2$.

5.2 Boolean functions

Boolean functions are programs that use a number of Boolean variables and operators, and calculate a Boolean value. Similar to symbolic regression, a set of data points is used to evaluate programs, and the fitness of a program is equal to the number of correct results returned by the program. The Boolean problems most frequently used are multiplexer and parity functions.

5.2.1 *N*-Multiplexer

The inputs used in the *N*-multiplexer problem are *k* Boolean address variables a_0, ..., a_{k-1} and 2^k Boolean data variables d_0, ..., d_{2^k-1}, where $N = k + 2^k$. The desired return value of the *N*-multiplexer is the value of the data variable d_i, where *i* is the address formed by the address variables: $i = \sum_{j=0}^{k-1} a_j \cdot 2^j$. Typical values of *k* are 6 and 11.

The terminal set of the multiplexer problems consists of the *N* address and data variables of the multiplexer. The non-terminal set contains the four Boolean functions {and, or, not, if-then-else}. The fitness function returns the number of correctly classified inputs.

5.2.2 *N*-Parity functions

N-parity functions have *N* Boolean input variables d_0, ..., d_{N-1}. An even *N*-parity function returns the value true if the inputs d_0, ..., d_{N-1} contain an even number of true values and false otherwise, while an odd *N*-parity function tests for an uneven number of true values.

The terminal set used to solve *N*-parity functions consists of the *N* input variables, and the non-terminal set contains the Boolean functions {and, or, nand, nor}. Note that the Boolean xor-function, which would be very useful to solve this problem, is not present in the non-terminal set. Even though the function set is sufficient to construct any possible Boolean function, the complexity of constructing a solution for an *N*-parity function increases quadratically. If xor is added to the function set, the complexity is only linear.

5.3 Artificial ant

The task of the artificial ant problem is to construct a program that navigates an artificial ant in an environment, picking up all the food that lies on an irregular trail in the environment. The environment is a toroidal grid of square cells. The following commands can be used to control the ant:

- Move-forward: This moves the ant one space forward in the current direction it is facing. If the ant moves to a square containing food, the food is eaten and removed from the grid.
- Turn-left, turn-right: These commands turns the ant 90° left or right.
- If-food-ahead: This function tests if the square directly in front of the ant contains a piece of food. If true, the first argument of this function is executed, otherwise the second argument is executed.
- Prog2, prog3: These functions are used to sequentially execute 2 or 3 arguments.

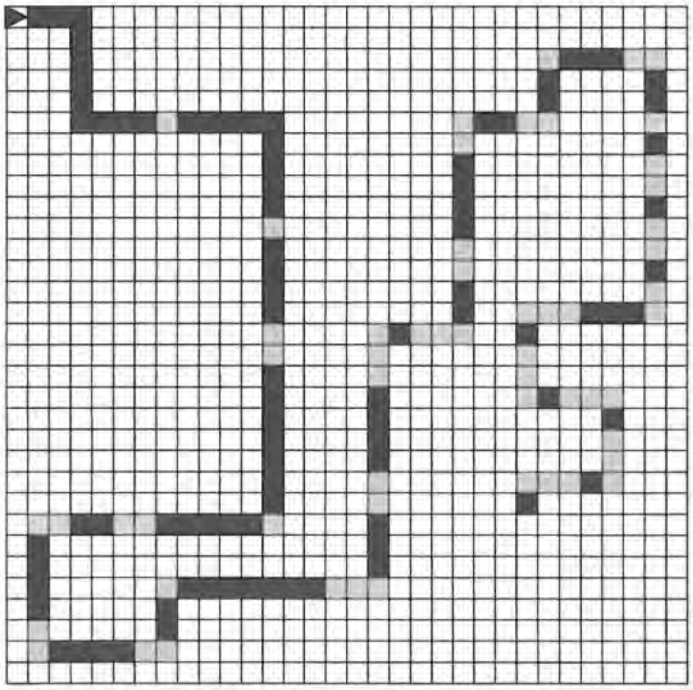


Figure 5.9: The Santa Fe trail used in the artificial ant problem.

The commands move-forward, turn-left and turn-right each require one time unit to be executed. A program is run for a limited amount of time. The programs that control the ant will be repeated, until either the time limit has been reached or all the food in the environment has been eaten.

The terminal set contains the instructions {move-forward, turn-left, turn-right} and the non-terminal set contains the instructions {if-food-ahead, prog2, prog3}. The fitness function returns the number of food pieces remaining in the environment when the program ends.

The environment that is typically used to train the artificial ant is called the Santa Fe trail. This is a 32x32 grid, containing a trail of 144 squares with 21 turns and 89 pieces of food. The Santa Fe trail is shown in figure 5.9. The black squares represent the pieces of food, and the gray squares denote empty squares on the trail. Initially, the ant is placed in the upper left corner facing the trail, indicated by the arrow. The time limit in the Santa Fe trail is set to 600 time units.

5.4 AI planning

AI planning is a separate field of artificial intelligence, where the objective is to find a sequence of actions that lead from an initial state to a goal state. The initial and goal states are described using a list of predicates that are true initially and in the goal state respectively. The actions that can be applied in the environment are also specified. These actions contain a number of preconditions that must be satisfied before the action can be executed, and a list of results that modify the current state of the environment.

A typical example of AI planning is the briefcase problem. The environment contains a number of locations, briefcases and pencils. Briefcases and pencils are at one location. Pencils can be put in and taken out of briefcases, and briefcases can be moved from one location to another location. The task is, given an initial distribution of the briefcases and pencils, to move all the pencils to a specified target location. This environment can be described as follows:

```
# Briefcase domain

(define (domain briefcase-world)
  (:types location physobject briefcase)
  (:predicates (at-briefcase ?b - briefcase ?l - location)
               (at-obj ?o - physobject ?l - location)
               (in ?o - physobject ?b - briefcase))

  (:action mov-b
     :parameters (?briefcase - briefcase
                  ?from ?to - location)
     :precondition (and (at-briefcase ?briefcase ?from)
                        (not (= ?from ?to)))
```

```
      :effect (and (not (at-briefcase ?briefcase ?from))
                   (at-briefcase ?briefcase ?to)
                   (forall (?obj)
                       (when (in ?obj ?briefcase)
                             (and (not (at-obj ?obj ?from))
                                  (at-obj ?obj ?to))))))

   (:action put-in
      :parameters (?object - physobject
                   ?briefcase - briefcase
                   ?location - location)
      :precondition (and (at-briefcase ?briefcase ?loc)
                         (at-obj ?object ?loc)
                         (forall (?b)
                             (not (in ?object ?b))))
      :effect (in ?object ?briefcase))

   (:action take-out
      :parameters (?object - physobject
                   ?briefcase - briefcase
                   ?loc - location)
      :precondition (and (in ?object ?briefcase)
                         (at-briefcase ?briefcase ?loc))
      :effect (not (in ?object ?briefcase)))

)
```

The above code defines the types and predicates used in the environment, and describes the actions that can be used (mov-b, put-in, take-out). To define a planning task, an initial state and goal state can be described in the following way:

```
# Briefcase problem

(define (problem get-paid)
   (:domain briefcase-world)
   (:init (location Home) (location Office)
          (briefcase Briefcase1)
          (physobject P) (physobject D)
          (at Briefcase1 Home) (at P Home) (at D Home)
          (in P Briefcase1))
   (:goal (and (at Briefcase1 Office) (at D Office)
               (at P Home)))
)
```

Muslea [73][74], and later Westerberg [110], developed an AI planning system based on the genetic programming paradigm. To represent a plan using genetic programming, the following terminal and non-terminal sets are defined:

- Terminal set: This set contains the objects specified in the domain and the problem. In the briefcase problem defined above, this set is {Home, Office, Briefcase1, P, D}.
- Non-terminal set: This set contains the actions described in the domain. Additionally, the functions prog2 and prog3 are added to make it possible to construct a sequence of actions. In the briefcase problem, this set is {prog2, prog3, mov-b, put-in, take-out}. mov-b , put-in and take-out have a number of child nodes equal to the number of parameters defined by the domain.

The fitness of a plan is calculated by executing the plan, starting from the initial state. If an action is encountered whose preconditions are not satisfied, that action will be ignored. When the plan is executed, the number of goal conditions that are not satisfied is counted and is used as the fitness of that plan.

6 Extensions to genetic programming

In the previous sections, the basic principles of genetic programming have been introduced. However, for practical reasons it is often necessary to extend this basic algorithm. This section will discuss strongly typed genetic programming, automatically defined functions and the use of an acyclic directed graph to represent a population.

6.1 Strongly typed genetic programming

The closure property requires that every object can be used as a child node of any non-terminal object. Unfortunately, not all problems can be represented easily with only a single type. For example, the briefcase problem described in section 5.4 uses terminal nodes that have different types (location, briefcase, physobject). The actions in this problem require child nodes that have a specific type. There are several possibilities to enforce these type requirements:

- Greatly reduce the fitness value of individuals that violate the type requirements. The problem of this approach is that almost every generated individual is likely to contain invalid types, and all individuals will have identical fitness. Evolution will be severely hindered in this case.
- Transform object of invalid type to the correct type. This technique was used by Muslea [74] to convert terminal nodes in the AI planning domain to the correct type. In this domain, non-terminal nodes all have the same type, so no conversion was needed there.

- Modify the genetic operators to only create or modify syntactically correct individuals. In the case of the crossover operator, this means that the cut-off points selected in both individuals must be of the same type. This method will be described here.

To overcome the closure constraint, Montana [72] introduces strongly typed genetic programming. In this system, a type must be associated with the arguments and return types of all terminal and non-terminal nodes. The genetic operators are modified to generate only syntactically correct types. A restriction of the method used by Montana is that only two levels of typing are possible: all types are derived from a common parent class, but no types can be derived from a non-parent type. Haynes [30] removes this restriction by allowing a type hierarchy. His system allows subtyping, meaning that any object that requires a child of a type T will also accept any type that is a subtype of T. This allows a form of polymorphism in the definition of type constraints of child nodes.

When strongly typed genetic programming is used, the search space of the problem is reduced. As a result the search time needed to solve a problem decreases. In [65], McPhee uses strongly typed genetic programming on essentially typeless problems, and notes an improvement in performance in some cases. However, this increase may also be due to the fact that the strongly typed representation makes it easier for internal nodes in the typeless representation to be modified, which can be an advantage in some problem domains.

6.2 Automatically defined functions

In any programming task, the use of subroutines can make that task significantly easier. Consider for example the odd N-parity problem discussed in section 5.2.2. A simple solution for this problem would be the program (d_0 xor d_1 xor ... xor d_{n-1}). However, the function xor is not part of the non-terminal set, even though an xor(a, b) function can be constructed with the expression ((a nand b) and (a or b)). This program, containing 7 nodes, is a solution for the odd 2-parity problem. A solution for the odd N-parity problem, where $N = 2^{n+1}$ for some $n \in \mathbb{N}_0$ can be constructed by joining two solutions of the odd 2^n-parity problem using the xor function. However, this solution of the odd N-parity problem will contain $2N^2$-1 nodes (see Appendix D). If the xor-primitive can be used, the solution contains only $2N$-1 nodes. This demonstrates that the introduction of new primitives, even if those primitives can be constructed from the existing primitives, can reduce the complexity of the solution. Therefore, if the genetic programming system is able to construct new primitives from existing ones, finding a solution may become easier.

In [44] and [45], Koza introduces the concept of an automatically defined function (ADF). When ADFs are used in genetic programming, the non-terminal set must be extended with the symbols $\mathrm{ADF}_0, \ldots, \mathrm{ADF}_{n-1}$, each with arity $a_0, \ldots, a_{n-1}$. The number n and the values $a_0, \ldots, a_{n-1}$ are part of the representation of the problem. The ADF-symbols represent calls to subroutines that will be evolved along with a solution. Additionally, every individual in the population will contain n additional subtrees containing the definitions of the n ADFs. These ADFs will be represented using a tree similar to a normal genetic program, but with slightly modified terminal and non-terminal sets. The terminal set usable by ADF_i is the terminal set of the original representation, extended with the a_i formal parameters of the subroutine. The available non-terminal set is the terminal set of the original representation, extended with the symbols $\mathrm{ADF}_0, \ldots, \mathrm{ADF}_{i-1}$. As a result, every ADF can only use previously defined ADFs, and infinite recursion between subroutines is not possible.

When ADFs are used by the genetic programming system, the crossover operator must be modified to handle the ADF definitions in individuals correctly. This crossover operator, called structure-preserving crossover, will only swap subtrees between individuals that are part of the same ADF, or that are both part of the main program. This ensures that the terminal and non-terminal sets of the main tree and ADF trees remains constant in all individuals.

6.3 Representing the population with a minimal directed acyclic graph

In genetic programming, the population contains many individuals represented by a tree. In this population, there will be many individuals that contain identical subtrees. Normally, every instance of the identical subtree will contain a different copy of that subtree, causing a large amount of duplication. If all these identical instances are represented by a single object that is shared between all individuals, the memory required to store the population will decrease considerably. This is demonstrated in figure 5.10, where the expressions $(a-(a+b))$ and $((a+b)*c)$ share their identical subtree $(a+b)$. Also, the first expression uses the terminal a two times, and re-uses the object representing a. When the entire population is represented in this way, a directed acyclic graph (DAG) is formed. The DAG is minimal if every distinct subtree is only represented once in the DAG. This representation for a population in genetic programming was first used by Handley [29], and was later also used by Keijzer [40].

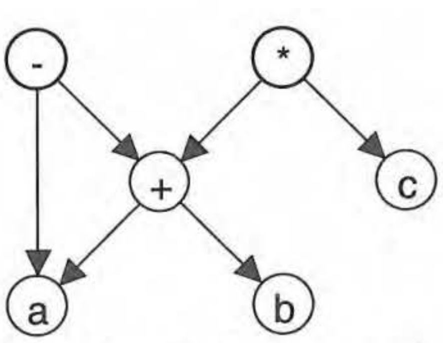


Figure 5.10: Individuals share identical subtrees in a directed acyclic graph.

6.3.1 Representation of objects

Because identical subtrees must be shared between individuals, it is necessary that identical subtrees can be detected easily. This can be accomplished by using an appropriate representation:

- All objects, both terminals and non-terminals, are given a unique index number.
- All types of non-terminal object (such as the types '+' or 'and') are also given a unique index number.
- An appropriate data structure is used to store all terminal symbols. For example, integers can be stored in a hash table and real numbers can be stored in a sorted tree. When a new terminal object is about to be created, these structures can easily test if the new terminal object is already present in the population. If the object already exists, that object will be re-used. Otherwise, a new object will be created, added to the data structure and will be given a new unique index.
- All non-terminal objects of arity a and type index t can be represented by a vector containing $(a + 1)$ indices. This vector is $(t, c_1, \ldots, c_a)$, where c_i is the index of child node i of the object. These vectors can be stored in an appropriate data structure, for example a sorted tree using lexicographical ordering. When a new non-terminal object is about to be created, this data structure can be tested to determine if the new object is already present in the population. If the object is found, it will be re-used. Otherwise, a new object will be created, added to the data structure and will be given a new unique index.
- A population of N individuals is represented by a set of N indices, where every index represents the root node of every individual.

6.3.2 Advantages of the DAG representation

The DAG representation has the following advantages:

- Less memory is needed to store the entire population. In a population of size 500, Handley [29] reported a 15 to 28-fold reduction in the number of nodes.
- When the evaluation of subtrees have no side effects, the results of previous evaluations of a subtree can be cached in a node and can be re-used when the subtree is evaluated again. Handley [29] reported a 11 to 30-fold reduction in the number of evaluated nodes, again using a population size of 500.
- The DAG representation can be used to easily find structurally similar elements between different individuals. Keijzer used the DAG representation to define a distance measure between individuals [40]. In the next chapter, the DAG representation will be used to increase the diversity of a population by removing similar individuals from the population.

Unfortunately, the DAG representation also has some disadvantages:

- Implementation of the representation is more complex.
- Adding an individual containing *n* nodes to a population containing *p* nodes takes O(*n*log*p*) time. With the traditional representation, this takes O(*n*) time.
- Removal of nodes from the DAG representation is difficult. This typically requires some form of garbage collection, which is computationally expensive.

7 Conclusion

This chapter introduced genetic programming as a general machine learning technique. This technique creates variable length programs, represented as trees, that are used as candidate solutions for a problem. A number of programs is maintained in a population. The quality of the candidate solutions is measured, and a fitness value is assigned based on this quality. Better solutions have a higher chance to be selected by genetic operators such as crossover or mutation. These operators are used to create new programs based on existing programs, in an attempt to combine good properties of the existing programs.

The crossover operator generates two new programs based on two existing programs. Although the combined size of both new programs is equal to the combined size of the existing programs, the individual lengths of both new programs are usually different from their parents. Interestingly, the larger program will tend to have a better fitness value then the smaller program. Consequently, the size of programs can grow rapidly. This phenomenon will be examined in detail in the next chapter, and new and existing techniques will be presented to reduce this growth.

Chapter 6: Reducing code growth in genetic programming

When genetic programming is used to evolve a solution for a problem, it quickly becomes apparent that the average size of the individuals in the population increases rapidly. This phenomenon, often called „bloat", has been reported in many genetic programming publications [3][4][44][49][50][51][52][61][63][98]. This phenomenon is also present in other evolutionary algorithms that use variable length representations. According to Langdon in [53], at early generations the size of individuals grows sub-quadratic with respect to the number of generations. Later, this appears to converge towards quadratic growth.

After several generations, bloat can become a problem because of the increased need of memory to store the solutions, and a longer evaluation time when assigning a fitness value to these individuals. Other problems of large solutions are that the generated solutions are less general, and bloat can also have a negative impact on the speed at which genetic programming can find solutions.

In [44], Koza introduces two methods to reduce code growth. First, a fixed maximum tree depth of 17 is imposed on individuals. Second, parsimony pressure is used where the size of an individual influences its fitness value. In this case, smaller individuals have a better fitness value. However, more advanced methods are needed.

In this section, first the causes of bloat and their effect on the evolution of the population will be discussed. Then, the effect of detecting and removing inactive code from individuals will be examined. Finally, several methods to reduce bloat will be compared with each other.

1 The causes of bloat

When observing the large individuals in a population after several generations, it becomes apparent that a large part of these individuals are nodes that have no effect on the result of that individual. These can be nodes that are never visited during the execution of the individual (for example a conditional expression whose condition is always false). They can also be nodes that have no effect on the result of the individual when executed (for example adding 0 to another number). Nodes that have no effect on the result, and therefore on the fitness of an individual are often called introns. This is a term used in biology that describes genes in a genotype that are not expressed in the phenotype.

The cause of bloat has been the subject of extensive research. In [61], Luke describes four major theories of bloat:

- **Hitchhiking:** This theory, introduced by Tackett in [103], says that introns are spread from parents to offspring because of the crossover operator. Introns are transferred along with the essential nodes of the parents.
- **Defense against crossover:** Blicke and Thiele in [6], and Nordin and Banzhaf in [77] argue that adding introns to partial solutions protects these partial solutions from being split up and destroyed because of crossover. As a result, the offspring has a higher chance to have a fitness similar to its parents fitness, which is typically above average.
- **Removal bias:** Soule and Foster [100] focus on a special case of introns, namely unviable code. This is code which can never have any effect on the result of an individual, even when modified. Typically, unviable code is located near the leaves of an individual. Consequently, removing a small subtree near the leaves is more likely to remove only unviable code and leave the individual intact. The size of the subtree that is added on the other hand has little or no effect on the fitness of the individual. This favors the removal of small subtrees during crossover, while no bias exists for the size of the added subtrees. As a net result, the average size of individuals will grow.
- **Diffusion:** Langdon and Poli [49][50] argue that because a solution can be represented in different ways, and because there exist more solutions with a greater size than simple solutions, it is only natural that more of the larger solutions are found. As a result, bloat is caused by fitness based selection.

While defense against crossover and removal bias are reasonable explanations of bloat in a population where improvement becomes difficult, it does not explain the early growth in a population [64]. Therefore, it is likely that bloat is caused by a combination of these factors, and possibly others as well. This is confirmed in [4], where Banzhaf en Langdon use a simple model to simulate the size, fitness, and the amount of active and inactive code of the individuals in a population. The model was used to test the theories of removal bias and diffusion. Their results indicate that both these theories partially explain bloat. On the other hand, Soule and Heckendorn [101] perform experiments to test the effects of defense against crossover, removal bias, and diffusion. These experiments show that defense against crossover and removal bias are responsible for code growth, but diffusion has little or no effect.

An exact theory of code growth was developed by Poli and McPhee. This theory was originally limited to linear structures in [64][84], but was later expanded to include variable-length structures when using homologous crossover [85]. Their work shows that standard crossover has a bias towards (over-)selecting smaller structures on a flat fitness landscape when using a linear representation. Also, when there is an infinite population and only two possible fitness values, the average size of the population converges towards the average size of the fitter individuals. However, the average size of the individuals does not change because of crossover when no individuals are removed from the population. This implies that when duplicate individuals are removed from a population, the average size of the individuals will increase. This is because it is more likely that duplicate elements occur among small individuals because the search space contains less individuals of a small size. As a result, small individuals have a larger change to be removed. Also, when less fit individuals are removed, the average size tends to increase because larger individuals with more inactive code have a higher chance to keep the above average fitness of their parents after crossover.

When observing the code of a bloated individual, large parts of this code constitute of inactive or non-executed code, also called first-order introns. However, these first-order introns are not the only ways to increase the size of individuals. Another way to add unnecessary nodes to a solution is by adding code to a solution, as well as other code elsewhere that nullifies the effects of the first section of code. This type of introns are called higher order introns. These pieces of code can be located at great distances from each other in the representation, and are in general impossible to detect in less than exponential time. For example, in the expression $(\underline{x+}((x*x)\underline{-x})$, the underlined pieces of code cancels each other out.

Another cause of bloat is called incremental fitness introns described by Smith and Harries in [97]. These are large pieces of code that have only a small effect on the result of the entire individual. This can happen when the results of these nodes are dominated by the results of other nodes. For example in the individual $((x^5 + 4x^4) + C)$, the term $(x^5 + 4x^4)$ will dominate the result if C consists of a large sum of constants and instances of the lower orders of the variable x. Adding terms to C will cause only small changes on the fitness of the individual, either positive or negative. If the code $(x^5 + 4x^4)$ has a high fitness, adding the term C will still result in a high fitness. This provides the evolutionary process with an easy way to produce individuals with a high fitness value, and the size of individuals will begin to grow rapidly.

2 The advantages and disadvantages of bloat

The most obvious disadvantages of bloat are the higher memory requirement and the longer execution time. However, memory has become very cheap to upgrade nowadays and is therefore not a significant problem. The longer execution times can be reduced somewhat by smart evaluation, such as only needing to execute one branch in an AND-node if the first node returns false. Execution time can further be reduced by caching partial results of subtrees when using the directed acyclic graph representation described by Handley [29]. Unfortunately, this is only an option when the unexecuted nodes have no side-effects.

On the other hand, according to the removal bias and protection against crossover theories, bloat appears to be helpful for the evolution. Introns can reduce the chance of destructive crossover, thereby increasing the average fitness of the offspring. However, the chance of a successful crossover is also reduced and crossover will usually have no effect on the fitness and behavior of offspring [64]. In this case, introns help insure that the successful individuals are transferred to the next generation, embedded within introns. However, this can also be accomplished by an elitist approach.

In [111], Wineberg and Oppacher claim that adding introns dynamically reduces the search space of a problem. They use a fixed-length representation for a solution, and simulate a variable-length representation by using a non-terminal node that ignores all but one of its child nodes. The ignored child nodes are thus by definition introns. Because of the fixed-length representation, introns cause a reduction of the phenotype size, and therefore limit the search space. However, because of the fixed representation length, these introns are not useful to study the effects of code growth in genetic programming in an unlimited variable length representation.

Another feature of introns is that they can store potentially useful code, that may become more useful after changes in the environment [3][50][111]. This can be especially important when looking for a solution in a dynamic environment, where features that were useful in the past can be saved to be later reintroduced as active code in the population.

In [1], Andre and Teller report that the occurrence of introns is damaging in experiments on the 5-parity, lawnmower and symbolic regression problems.

There are several disadvantageous of bloat:

- The memory requirement and execution time are higher because of introns.
- Crossover has no effect when introns are selected in both parents [1]. While children are not worse than their parent, neither will they be better and

evolution slows down and eventually the population consists of instances of similar solutions [97].

- When the depth of individuals is restricted, the solution space becomes cramped [1][25]. This means that the result of crossover can be rejected because the depth of the individual is too large, regardless of its fitness value. If bloat is prevented, the individuals would not easily become large enough to be rejected. Thus, introns limit the solution space.
- Occam's razor: small solutions to a problem tend to be more general then larger solutions [42][92][93][98]. Zhang and Mühlenbein measured a decrease in generalization of larger individuals when evolving neural networks [113].
- Changes to large programs tend to be local [50][51]. This is because the effect of a node at a deeper level tends to be smaller than the effect of higher nodes. In a large individual, more nodes are situated at deeper levels and therefore the changes are smaller. As a result, the search may become trapped in a local maximum in the search space.
- Larger solutions are more difficult to understand by human observers [42].

In [78], Nordin, Banzhaf and Francone introduce explicitly defined introns (EDI) for linear structures of machine code. An EDI is a special type of node that has no effect on the result of an individual, and has an associated weight. This weight influences the probability of a crossover happening immediately before or after the EDI. On the problems discussed in [78], EDI's have the following advantages:

- Average fitness, generalization and CPU time needed to evaluate individuals is frequently improved.
- Implicit introns and EDI's work together.
- Sometimes, EDI's replace implicit introns (those that are naturally present).
- EDI's protect against the destructive effect of crossover.
- When combining parsimony pressure and EDI's, bloat is reduced while maintaining the advantages of introns.

In [97], Smith and Harries implement EDI's for tree structures. They conclude that the effects of EDI's on linear structures are very different from their effects on tree structures. Because introns protect against disruptive crossover, introns cause the evolutionary process to maintain its current fitness instead of improving the best fitness of the population, and caused evolution to stop. As a result, average fitness of a population is not necessarily a good measure of the quality of evolution.

O'Neill [80] uses explicitly defined introns in grammatical evolution to improve the performance of the system in some cases. These explicit introns introduced a bias in the selection of certain terminal or non-terminal nodes. In the cases where these terminals are more useful for finding a solution, the use of introns improved the convergence speed. Because these side-effects have such a strong impact on the results, it was not possible to measure the effects of the increased size and inactive code on the results.

3 Detecting and removing inactive nodes

Since inactive code appears to make up most of the code of a bloated individual, it makes sense that bloat can be reduced if the inactive nodes can be removed from the individuals. In this paragraph, this idea will be explored for several problem domains where it is easy to detect and remove inactive nodes. After discussing some related work on removing inactive code, a method will be described to measure the influence every node has on the result of an individual. Nodes that have no influence are by definition introns, and a method to remove them from the individual will be presented. The effect of removing inactive code will be tested experimentally, and finally some conclusions will be given.

3.1 Related work

In [77], Nordin and Banzhaf identify different types of introns:

- **Global and local introns**: Global introns are nodes that have no effect on the behavior of an individual for every possible input in the program domain. Local introns have no effect for every input in the training set used to evaluate the individual. This code may become active when used in a different context.
- **Absolute introns**: These are nodes that have no effect on the behavior of an individual, and applying crossover on these nodes will not modify this behavior.
- **Continuously defined introns behavior**: In this case, nodes are given a numerical value of their sensitivity to crossover. An example is the underlined code in (/ $\underline{\text{(- Y 3)}}$ (EXP (EXP 10))), whose result is dominated by the value of the second part. Changes to the underlined code will have a negligible effect on the result of the individual.

Koza [44] describes a way to add syntactic rules to simplify individuals. This includes reducing trees with only constant nodes to a constant value, and reducing expressions like (Not (Not X)) to X. The purpose of these syntactic rules was mainly to make individuals easier to read when displayed, but they were not used to influence the behavior of the genetic programming system.

Soule, Foster and Dickinson [98] use a set of syntactic rules to remove inactive code from individuals. They report that when non-functional code is removed, bloat still occurs but the non-functional code is replaced by non-executed code instead. The use of parsimony pressure is more effective to limit the growth of individuals. However, the fitness of the best individual does not appear to be affected by size limiting approaches.

Nordin and Banzhaf [77] detect inactive code in strings of machine code by replacing every single instruction with a no-operation instruction and comparing the results of both programs. If the results are identical, the replaced instruction is inactive and is removed. Unfortunately, this technique can only detect single instructions that have no effect and is computationally expensive.

Blickle and Thiele [6] detect inactive code in the 6-multiplexer domain by inserting a NOT-node between the nodes of every edge of an individual. If the result of this new individual is identical to the result of the original, the subtree under the inserted NOT-node is by definition inactive. However, this process is very time-consuming. They also propose a new genetic operator called marking crossover. To use this crossover, nodes in an individual that are traversed during execution of the individual are marked. When two individuals are selected for crossover, only nodes that are marked can be selected as a crossover point. This prevents the crossover operator from exchanging useless code. Unfortunately, this method can only detect non-executed code, not non-active code. For example in the expression And(*Inactive*, False), the subtree *Inactive* will be executed but has no effect on the result. Marking crossover was successful in some domains, but was ineffective in other domains.

Rosca [92] measures the size of executed code in individuals by examining which nodes of a tree are executed during the evaluation of an individual. This method is able to detect non-executed code, but code that is executed but has no effect on the result of an individual is not detected in this way. Also, only the provided fitness cases are used to test if code is executed, meaning local introns instead of global introns are detected. This may be an advantage, because otherwise code for which no fitness is known can survive in the population. This can lead to a mutation-like behavior when the non-executed code is transferred to an active location after a crossover operation. Consequentially, the system may perform a higher percentage of mutation-like operations than specified by the mutation probability.

In [105] and [106], Teller calculates the influence of nodes to assign explicit credit to the nodes of an individual. The correlation between the desired output of an individual and the output value of individual nodes is calculated. This value is an approximation of the contribution of that node to the result of the individual.

3.2 Measuring the influence child nodes have on their parents

In genetic programming, a program is typically represented by a tree of nodes. The result of the program is produced by the individual nodes of this program tree. However, some nodes have a larger impact on the final result than other nodes. For example, nodes near the top of the tree will usually have a larger effect on the result of the program than nodes located near the base of the tree. But even two children of the same non-terminal node can have a different influence on the result of their parent node, and thus on the final result of the genetic program. For example, in the program (2 + 5), the child node 5 will have a larger effect on the result than the child node 2. It is also possible that a child node has no effect on the result of the parent node. For example, in the program (0 + 4), child node 0 has no effect, and in the program (0 * 4), child node 4 has no effect on the result of the calculation. In these cases, the inactive nodes can be removed from the program without altering its result, resulting in the programs (4) and (0) respectively.

In general, it is impossible to measure the influence each child node has on its parent. However, for some typical non-terminal nodes, it is possible to discover specific formulas for different node types that can calculate the influence of their child nodes during the evaluation of a program. In this section, several formula are designed by us to calculate the influence of child nodes for some specific non-terminal nodes. These formulas will then be used to detect and remove inactive code, and to test its effect on code growth. These results will indicate whether it is worthwhile to calculate the influences of child nodes. Otherwise, other methods need to be used to reduce code growth.

When calculating the fitness of an individual, the code of the individual is typically evaluated several times using different inputs. For example, a mathematical function can be evaluated using different values for its variables. While the influence of a subtree can be very low during some evaluations, the subtree may produce valuable results during other evaluations. Therefore, a subtree can only be considered inactive when it consistently has a low influence over all evaluations. As a result, the influences of all evaluations must be combined to produce the total influence value.

3.3 Removing inactive code

Immediately after the evaluation of an individual, when the influence values of the children are measured, inactive children can be removed from the individual.

When the influence of a subtree is determined to be 0, the subtree is inactive and can be removed without changing the result of the individual. Cutting away a subtree however will produce a syntactically invalid individual, because the parent node's arity is usually constant. Therefore, a syntactic rule must be provided that removes a subtree from an individual. For example, the subtree (X - 0) can be reduced to (X), removing both the inactive subtree (0) and the parent node '-'.

Other methods that only detect completely inactive code have difficulty in continuous domains where subtrees have a negligible (but non-zero) effect. This was reported in [5] and [97], and is the cause of incremental fitness introns. In [97], these nodes where prevented by using a hill climbing approach where the result of the individual must be greater than a small threshold value. By measuring the influence of nodes and using a threshold value, these nodes can be detected and removed. If the influence of a child node is very low, the node is an incremental fitness intron [97] and can be removed.

Sometimes a parent node with an inactive child can be reduced by replacing it with a different node with lower arity, keeping only active children. For example, the subtree (0 - X) can be reduced to (- X), where '-' is a unary minus operation. Similarly, a Prog3 node with one inactive child can be reduced to a Prog2 node.

3.4 Examples

In this section, several examples are provided to calculate the influence of child nodes and remove inactive subtrees. While the influence calculated in these examples is only an approximation of the real influence of a child node, it is a good heuristic when the influence of a node is low. As a result, this method suffices to detect inactive code and incremental fitness introns.

3.4.1 If-Then-Else nodes

An If-Then-Else node has an arity of 3, with child nodes Condition, True-case, and False-case. The Condition is a Boolean node, and the other children can be of any type (but must both be of the same type). The return type has the same type as True-case and False-case. The result of If-Then-Else is equal to the result of the True-case if Condition is true, and is equal to the result of the False-case otherwise.

During the fitness evaluation of the individual, an If-Then-Else node will be evaluated a number N times. The number of times the condition will be true is represented by T, and the number of times it will be false is then $(N - T)$. The influences of the child nodes are then, after N evaluations (when $N > 0$):

- Influence of Condition: 1/3.
- Influence of True-case: $(2/3)*(T/N)$.
- Influence of False-case: $(2/3)*((N - T)/N)$.

The sum of these influences is 1. If Condition was false in every evaluation, the influence of True-case will be 0. In this case, the entire If-Then-Else node can be replaced by False-case. Similarly, if Condition is always false the If-Then-Else node can be replaced by True-case.

3.4.2 Add/subtract nodes

Addition and subtraction are nodes of arity 2, whose child nodes and return value have a numerical type.

During a single evaluation, the child nodes have numerical values N_1 and N_2. The total effect of this node is defined as $| N_1| + | N_2|$. For example, the node (2 + 5) will have a total effect of 7, and the node (4 + -4) will have a total effect of 8, even though the final result of that node is 0. If N_1 and N_2 are both 0, this evaluation will not be used to calculate the total influence of the child node over all evaluations.

The influence of child node i during this evaluation is set to $| N_i|/(| N_1| + | N_2|)$. Consequently, when the result of one child is small compared to the result of the other child, that node will have a low influence value.

To obtain the influence of a child node over all evaluations, the average is taken of all the influence values of the single evaluations (where either N_1 or N_2 was not 0). As a result, the final influence of a child node will be low if the value of that child node is consistently smaller than the value of the other child node. If the influence of a child node is below a threshold value (for example 0.01), the child node can be removed without modifying the result of the parent node much.

To remove a child node from an add node, the inactive node and the add node are removed, and only the active node is retained. An inactive second child node of a subtract node can be removed in an identical way. However, if the first child node of a subtract node is inactive, the parent node must be replaced by a unary minus node, with the active child node as the child node of minus. For example, (0 – 5) can be replaced by (- 5). If a unary minus operator is not part of the non-terminal set of the genetic programming system, the inactive child node can not be removed.

3.4.3 Multiplication/division nodes

Multiplication and division nodes are nodes of arity two with numerical children and return type. In this case, the child nodes have two ways to affect the result of the parent node, and an influence value for both is calculated:

- The magnitude of the result of the child nodes will affect the magnitude of the result of the parent node.
- The signs of the result of the child nodes will affect the sign of the result of the parent node.

The magnitude influence of the child nodes is calculated after every evaluation, and is averaged after all evaluations to obtain the total magnitude influence. If the child nodes have numerical values N_1 and N_2, the magnitude influence is calculated as follows:

- If both N_1 and N_2 are 0, 1 or infinity, this evaluation is not used to calculate the final magnitude influence.
- If N_i is 0 or infinity, the magnitude influence of child i is 1 and the magnitude influence of the other child is 0.
- Otherwise, calculate the intermediate values M_1 and M_2: if $|N_i| < 1$, $M_i = 1/|N_i|$, otherwise $M_i = |N_i|$. This operation ensures that M_1 and M_2 are both ≥ 1, and $M_1M_2 > 1$. The effect of a child node increases as its value moves further away from 1. A value X has a similar effect on the magnitude of the multiplication as the value $1/X$, both modifying the other value of the multiplication by a factor X. Therefore, using the inverse of the value when it is less than 1 does not affect the magnitude influence and allows us to compare the effects of both children. The magnitude influence of child i is then given by equation (6.1):

$$MagnitudeInfluence_i = \frac{\log M_i}{\log(M_1 \cdot M_2)} \qquad (6.1)$$

The total effect of the node is represented by $\log(M_1 \cdot M_2)$, and the effect of node i is represented by $\log(M_i)$. The sum of the magnitude influences of both children is 1, and the influence of a child becomes 0 when M_i (and thus $|N_i|$) approaches 1. The magnitude influence of all evaluations is the average of these influences.

The sign influences of the children can be calculated by counting the number of times a value was positive or negative (ignoring evaluations where the value is 0). If after N evaluations child i was negative N_i times, the sign influence of child i will be N_i/N.

To calculate the total influence of a child, the magnitude influence and sign influence must be combined. If a child has a negative value, the result of the multiplication will change from a value A to a value $-A$, causing an equally large change as the total magnitude of the multiplication. After a single evaluation, the influence of child node *i* is given by equation (6.2):

$$Influence_i = \frac{\log(M_i) + Neg_i \cdot \log(M_1 \cdot M_2)}{(1 + Neg_1 + Neg_2) \cdot \log(M_1 \cdot M_2)} \qquad (6.2)$$

where Neg_i is 1 if the value of child *i* is negative, and 0 otherwise. For example, the influences of the children of (-1 * 100) are both 0.5. The sum of the influences is again 1, and the total influence after all evaluations is again the average of the single influences.

A child node can be removed if two conditions are satisfied:

- The magnitude influence of the child node is very small.
- The sign influence of the child node is either 0 or 1. If the sign influence is 0, the parent node is replaced by the other child node. Otherwise, the parent node can be replaced by a unary minus operator with the active child node as a child, if this operator is present in the non-terminal set. Otherwise, the node can not be removed.

If the first child node of a division is removed, the division node can be replaced by a unary Inverse operator if this operator is present in the non-terminal set, with as child the active child node. If this operator is not available, the node can not be removed.

3.4.4 And/Or nodes

And nodes are nodes of arity 2 that have Boolean arguments and return value and perform a logical And operator on its arguments. The child influences of a single evaluation are calculated as follows:

- If both children have the same logical value, this evaluation is not used to calculate the total influence after all evaluations.
- Otherwise, the child node with the True value will have an influence of 0, and the child node with the False value will have an influence of 1. This is because False dominates the result of the And node, setting the result to False regardless of the value of the other node.

If a child node always has a value of False, the influence of this child node will be 1 and the And node can be replaced by this node. If a child node is always True, the And node can be replaced by the other child node. If both children always have the same value, the And node can be replaced by either child node, preferably the smallest one.

A similar approach can be used to calculate the influence of logical Or nodes, changing the influence of True child nodes to 1 and False child nodes to 0.

3.4.5 Prog*N* nodes

Prog*N* nodes (where *N* is an integer value larger than 1) are used to sequentially execute *N* nodes that perform actions with side effects in an environment. To calculate the influence of the child nodes of a Prog*N* node, three functions must be provided by this environment. It should be noted that it may not be possible to implement all three functions in complex real world problems. In these complex environments, it will not be possible to calculate the influences of the child nodes.

- GetState(): This function returns information about the current state of the environment, such as the position of a robot, the current time, the number of food items eaten, the number of completed goals, etc.
- Effect(State1, State2): This function returns a positive value that represents the amount of change between two states. Changes are for example changes in position, and the difference between eaten food items or goals completed.
- Effort(State1, State2): This function returns a positive value that represents the effort that was needed to move from State1 to State2. This can be for example the difference of time between the two states, or the number of actions taken.

During the evaluation of the Prog*N* node, the following algorithm is executed:

```
State$_0$ = GetState()
for i = 1 to N do
   Evaluate child node i
   State$_i$ = GetState()
```

The influence of every child *i* can then be calculated with equation (6.3):

$$Influence_i = \frac{Effect(State_{i-1}, State_i)}{C \cdot (Effort(State_{i-1}, State_i) + 1)} \tag{6.3}$$

where *C* is a normalization factor that is equal to the total effect of the children of the Prog*N* node, to ensure that the sum of the influences is 1, shown in equation (6.4):

$$C = \sum_{i=1}^{N} \frac{Effect(State_{i-1}, State_i)}{Effort(State_{i-1}, State_i) + 1} \tag{6.4}$$

Effort is increased by one in equation 3 to avoid a possible division by zero. When the effect between the evaluation of two child nodes is 0, the influence of that child node will be 0. If the Prog*N* node contains *K* inactive child nodes ($0 > K > N - 1$), the Prog*N* node can be replaced by a Prog($N - K$) node that contains the active child nodes. If $K = N - 1$, the Prog*N* node can be replaced by the only active child node. If $K = N$, the Prog*N* node itself if inactive and can be removed by its parent node, or can be replaced by a No-Operation node if this node is part of the terminal set.

3.5 Experimental results

In this section, the effects of removing inactive subtrees will be examined on the 6-multiplexer and symbolic regression problems. Both the effect on the average size of the members of the population, and the number of generations needed to find a perfect solution will be tested. The GP system uses a steady state algorithm, where identical individuals are not allowed in the population. Individuals with identical fitness value are ordered by size. The steady state algorithm starts with a random population of a size *N*, and in every following generation *N* new individuals are created, evaluated and added to the population. Identical copies of existing individuals are rejected and in this case a new individual is generated. At the end of the generation, the best *N* individuals are retained in the population.

For all experiments, the following settings were used: population size 200, 70% of the individuals are created by crossover, 30% by combination, and a 2% chance that an individual is mutated when it is created. The mutation operator replaces a random node with a compatible node. The combination operator selects a random non-terminal primitive and creates children for this primitive using selection, crossover or combination. A run is terminated when no solution is found within 65 generations. The averages shown are taken over 100 runs, but only the runs that have not yet found a perfect solution are included in the average.

3.5.1 6-Multiplexer

This is the standard problem described in paragraph 5.2 of chapter 5. The non-terminal set is {And, Or, Not, If-Then-Else}, which means that all inactive nodes can be removed. The syntactic replacement rule that (Not (Not X)) = X is also used in the experiment, and the influence threshold used is 0.0.

When inactive nodes are not removed, a solution was found in 72% of the runs. The maximum average size of the population climbs to 677 at generation 50, and then lowers again to 538 at generation 65. When inactive nodes are removed, a solution was found in 94% of the runs, and the average size of the population rises to 194 at generation 41, and then drops to 129 at generation 65. The average size drops at a given moment because at some point all individuals of the population have the same fitness (only 32 different fitness values are possible for the 6-multiplexer problem). At this moment, smaller individuals are considered better then larger individuals, and therefore replace the larger individuals. This happens in the case when introns are removed around generation 40, and in the case when introns are not removed around generation 50. The delay in the second case is due to the fact that when introns are removed the population tends to converge faster to an optimal solution, and therefore reaches a point of identical high fitness values earlier.

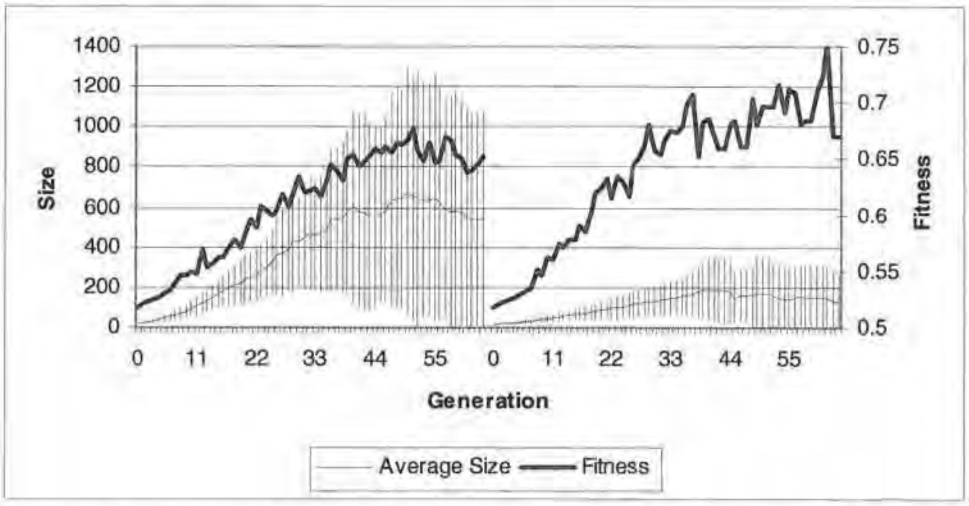


Figure 6.1: Average best fitness, average size and standard deviation of the 6-multiplexer problem without removing inactive nodes (left) and removing inactive nodes (right).

A significant improvement on both size and convergence speed can be detected in this example by removing the inactive subtrees, although the average size of the individuals is still much larger than necessary to find a solution.

3.5.2 Symbolic regression without unary operators

This problem attempts to discover a symbolic representation of the polynomial $x^5 - 3x^3 + x$. The non-terminal set consists of the functions {+, - (binary), *, /}, and the terminal set is {*x*}. Note that the unary minus and inverse operators are not present, so not all nodes with inactive code can be removed. The search space is more restricted though, which can lead to a faster convergence. The experiment was run with influence threshold values of 0.01 and 0.05.

When inactive child nodes are not removed, in 83% of all runs a solution is found in less than 65 generations. The average size of the elements in the populations that have not found a solution grows to 870. When inactive code is removed with an influence threshold of 0.01, a solution is found in only 72% of the runs, and surprisingly the average size increases to 1144 after 65 generations. If the influence threshold is increased to 0.05, the success rate drops further to 66%, but the size also decreases to 851 after 65 generations. In this example, removing inactive code has a negative effect on the convergence speed and no effect or a negative effect on the size.

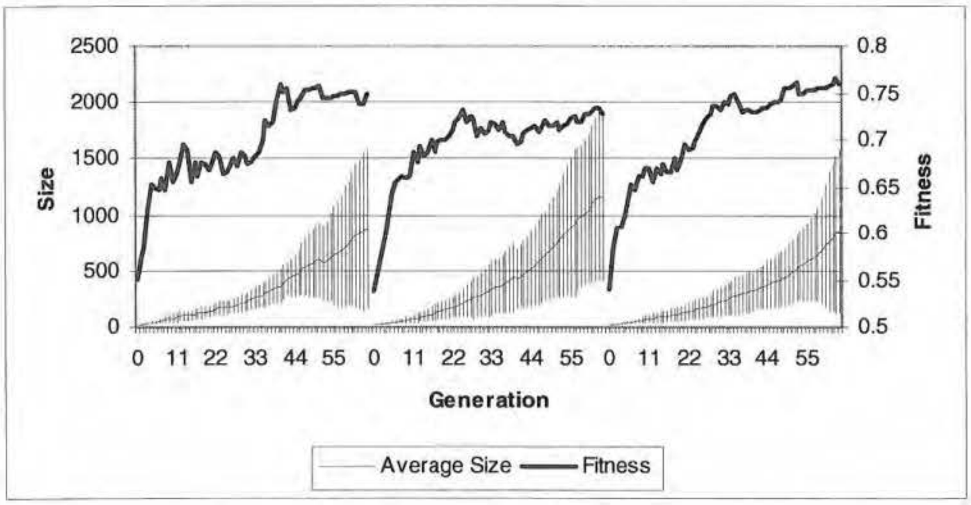


Figure 6.2: Average best fitness, average size and standard deviation of symbolic regression problem without unary minus and inverse operators, without removing inactive nodes (left), and removing inactive nodes with influence threshold of 0.01 (middle) and influence threshold of 0.05 (right).

3.5.3 Symbolic regression with unary operators

This problem tries to solve the same problem as in the previous paragraph, but the non-terminal set is extended with the unary minus and inverse operators. As a result, all inactive child nodes can be removed. However, by adding these two oparators, the search space has been increased and finding a solution becomes more difficult. The syntactic rules that (-(-X)) = X and (Inv(Inv(X))) = X are also used. All other settings are identical to the settings used in paragraph 3.5.2.

When inactive nodes are not removed, a solution is found in only 38% of the runs, and the average size of the population of the unfinished runs at generation 65 is 916. If the inactive code is removed with a threshold of 0.01, the average size drops to 674, but a solution is found in only 35% of the runs. Using a threshold of 0.05, the average size drops further to 564, but the success rate also decreases to 31%. In this example, removing inactive code has a positive effect on code size but a negative effect on convergence speed. However, the average size still grows rapidly, well beyond the minimum size needed to solve the problem.

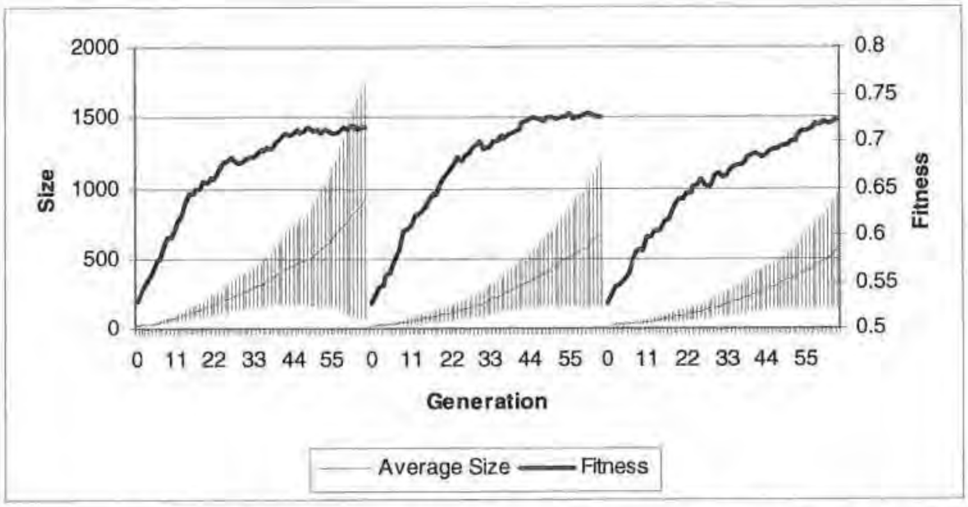


Figure 6.3: Average best fitness, average size and standard deviation of symbolic regression problem with unary minus and inverse operators, without removing inactive nodes (left), and removing inactive nodes with influence threshold of 0.01 (middle) and influence threshold of 0.05 (right).

3.6 Conclusion

In a first attempt to reduce code growth, a technique was developed to detect and remove inactive code from programs. Inactive code was detected by calculating the influence child nodes have on their parents. Child nodes with a low influence were removed from the individual. This technique can detect inactive and unexecuted code, as well as continuously defined intron behavior. To calculate this influence, formulas were designed for several specific types of nodes. The effect of removing inactive code on code growth was then tested experimentally on the problems of 6-multiplexer and symbolic regression.

In the case of the 6-multiplexer problem, a significant increase in convergence speed and decrease in average size was demonstrated by removing inactive nodes. However, in the symbolic regression problem, a smaller decrease in size was observed in some cases, but convergence speed decreased when inactive subtrees were removed.

Even in the 6-multiplexer problem where the average size of the population was significantly decreased, the average size would still grow beyond the size needed to solve the problem. This is consistent with the results of Soule, Foster and Dickinson in [98], where inactive code was replaced by non-executed code. The approach described in this section is also able to detect and remove non-executed code, which demonstrates that both inactive and non-executed code, as well as continuously defined intron behavior, are not the only causes for the rapid growth of the individuals' size. A similar result was also reported by Luke in [57].

Also, when working with other problem domains, it may be more difficult of even impossible to calculate the influence of some child nodes. To combat bloat effectively, it is therefore necessary to use other approaches, possibly combined with this technique. A comparison of combining this technique with several other techniques is described in [70] and in the next section.

4 Limiting code growth

4.1 Introduction

The previous sections of this chapter introduced the problem of bloat and investigated the causes and problems of bloat. Also, an attempt was made to reduce bloat by removing what appeared to be the most important contributor to bloat: inactive and unexecuted code. While this approach was somewhat successful in some problem domains, it had almost no effect, or even a negative effect in other domains. Even in successful domains, bloat was still considerable. Therefore, it is necessary to consider other means to combat bloat.

In paragraph 4.2, related work on different techniques to reduce bloat will be discussed. Paragraph 4.3 will present several methods, and these will be tested and compared with each other in paragraph 4.4 by running several experiments. Finally, conclusions will be presented in paragraph 4.5.

4.2 Related work

Because the problem of bloat is known since the beginning of genetic programming, a lot of work on different techniques to reduce code growth exist. Koza [44] uses the following techniques to fight bloat:

- The individuals that are created for the initial populations are restricted to depths between 2 and 6.

- Subtrees that are generated for subtree mutation are limited to a depth of 4, and non-terminal nodes are selected as mutation points with a probability of 90%.
- The creation of new individuals is limited to a depth of 17. Since this puts a hard limit on code growth, this may slow down the discovery of useful features. Also, this approach tends to generate fuller trees, which may or may not be advantageous for a given problem domain.

Another often used technique to slow bloat is the use of parsimony pressure [98][99][113]. In this case, an individual's fitness function is a (linear) combination of its performance and its size, where a larger size decreases the fitness value. While this approach reduces the average size of the population, it can also reduce the speed at which better solutions are discovered. Furthermore, it is possible that the population will get trapped in a local minimum where the combined fitness function is dominated by the size component and only individuals of a minimal size exist [77][99]. An additional difficulty is that an appropriate value must be found to combine raw fitness value and size. Zhang and Mühlenbein [113] solve this by calculating a parsimony value (also called Occam Factor) that depends on the complexity and raw fitness value of the best individual of the current population. Iba et al. [36] use a minimum description length principle to define the fitness of classifiers. This principle uses the size of an individual, combined with the size of the incorrectly classified test cases. The objective is to minimize this value.

In [94], Ryan describes a method where two populations of individuals are maintained. The first population contains individuals that are selected based only on their performance, while the second population contains individuals that have a small size. Crossover is then applied between individuals from both populations to create new individuals that are small and have a high performance.

Code editing is another way to reduce the size of individuals. In this approach, inactive code is removed from the individual. However, inactive code is not the only way to introduce bloat, as was demonstrated in section 3. By removing the inactive code, other forms of bloat will begin to dominate the population.

Another way to reduce the size of individuals is to use a multi-objective method, as described in [19] and [22]. When using this approach, the population of individuals is restricted to those members that either have good fitness or a small size, but where no member exist that has both better fitness and smaller size. This set of individuals is called the set of Pareto-optimal solutions.

In [51], Langdon describes a simulated annealing technique where the maximum size of offspring depends on its parents' size. In the beginning of a run, the resulting size is allowed to be considerably larger than its parents' size, but this allowance decreases over time as the system "cools down".

It is also possible to introduce a form of hill climbing in the crossover process [51][81], as described by Langdon and O'Reilly. Using this approach, individuals created by the crossover operator are rejected if the result is not fitter or smaller than its parents. This approach was reported to be vastly superior over the simulated annealing approach, and restricts bloat considerably. If a strict hill climbing approach was used, bloat disappears even completely, but more time is needed to find better solutions. This will be confirmed in paragraph 4.4. However, strict hill climbing tends to stifle evolution when a local optimum is reached.

Another method to modify crossover was discussed in [97] by Smith and Harries. They introduce a same-depth crossover operator that selects subtrees of equal depth when performing crossover. As a result, the depth of individuals can not be changed because of crossover. While same-depth crossover eliminated bloat, the success rate of the evolution was reduced too drastically to be useful. When same-depth crossover was used in 50% of the time and standard crossover was used otherwise, bloat was reduced, but the success rate was lower as well.

In [5], Blickle compares the use of simple parsimony pressure, marking crossover [6], explicitly defined introns [97], and adaptive parsimony pressure [113]. He concludes that the use of explicitly defined introns and the marking crossover offer no advantage over the use of a simple parsimony pressure in continuous problems. On the other hand, the marking crossover was superior in discrete problems. These results are comparable to the results obtained in paragraph 3.5 of this chapter. However, all methods were successful in reducing the size of individuals in the population.

In [64][84][85], Poli and McPhee develop an exact theory of bloat. When the exact causes of bloat are understood, it would be possible to construct genetic operators that are unbiased towards creating bloat. However, more research on this subject is needed.

In [50][52], Langdon and Poli examine bloat in dynamic environments. The "defense against crossover" theory of bloat indicates that children having an identical behavior as their parents is an evolutionary advantage. As a result, in a dynamic environment where the requirements of survival can change every generation, this effect should disappear. Their experiments indicated that in this case bloat was indeed reduced, but the generalization capability of the resulting individuals decreased as well. This confirms that defense against crossover is part of the cause of bloat.

4.3 Methods to reduce code growth

In this section, three methods to reduce bloat are presented: removing inactive code, hill climbing and dynamic size limiting. The effect of these methods on the growth of the average size of the population and the convergence speed will be examined in paragraph 4.4.

Removing inactive code: This is the method described in section 3 of this chapter, where subtrees that do not contribute significantly to the result of the individual are removed.

Hill climbing: This is the approach used in [51] and [99]. The method was described in paragraph 4.2.

Dynamic size limiting: In this new approach developed by us, new individuals that are larger than a value MaxNewWeight are rejected. MaxNewWeight is set at the beginning of every generation, and is equal to C*CurrentBestWeight, where CurrentBestWeight is the weight of the current best individual of the population, and C is a constant. The best individual of a population is the individual with the highest fitness value. When several individuals have an identical fitness value, the smallest individual is the best one. C must be larger than 1, and typical values are 1.33 and 1.5.

4.4 Experiments

To compare the different optimization techniques, the domain of symbolic regression was used. In all experiments, the objective was to discover the function x^5-2x^3+x. The terminal set only contained the variable x, and the set of non-terminal nodes was {+, -, *, /}, where '/' is protected division. The population size in every experiment was 200, and a maximum of 100 generations were calculated. The three different optimization techniques (and combinations of those techniques) described above were tested on this domain. Child nodes are considered inactive when their influence value is below 0.05. For every experiment, the algorithm was run 50 times, and the averages of those runs are displayed. A run is terminated when a correct solution is found, or after 100 generations. In the graphs, the individual's average size at every generation is shown, as well as the number of runs at every generation that have not yet discovered an optimal solution.

4.4.1 No optimizations

Figure 6.4 shows the result when no optimizations are used. As expected, the average size of the individuals rises rapidly, up to 650 nodes at generation 100. In 74% of the runs, a solution is found within 100 generations.

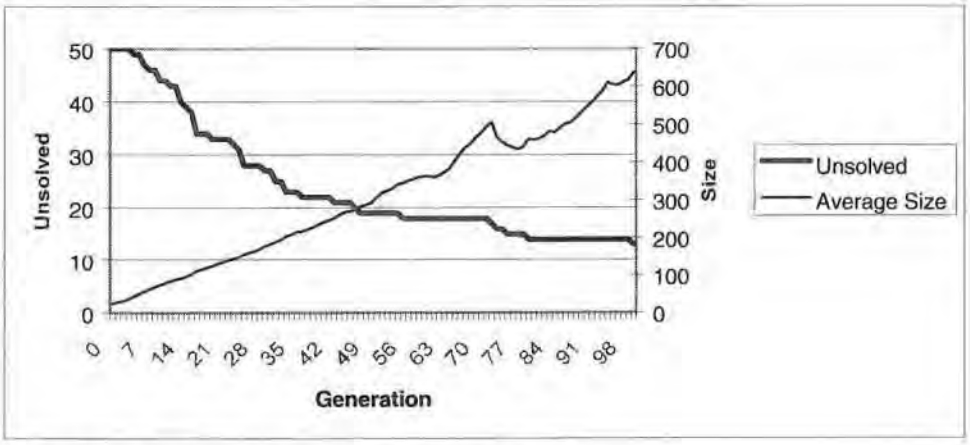


Figure 6.4: Result when no optimizations are used.

4.4.2 Removing inactive code

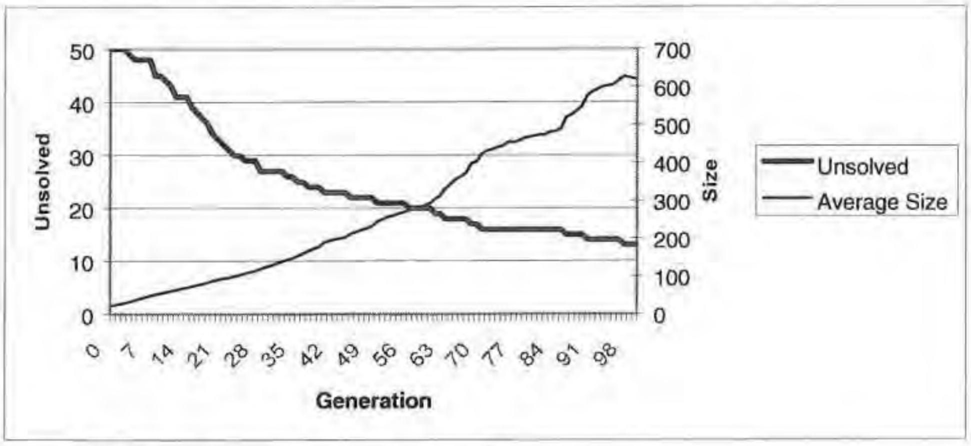


Figure 6.5: Result of removing inactive code.

In figure 6.5, the effect of removing inactive code is demonstrated. The results are only slightly better than the non-optimized case. Initially, the average size rises more slowly, but still rises to 625, and 74% of the runs find a solution. This demonstrates that inactive code is only a small part of the cause of code growth in the domain of symbolic regression.

4.4.3 Hill climbing

Figure 6.6 shows the result of using the hill climbing approach. The average size of the individuals is reduced significantly to 90 nodes after 100 generations, but in only 70% of the runs a solution is discovered. This is consistent with results reported by Langdon in [51], indicating that hill climbing slows down evolution when a local maximum is reached.

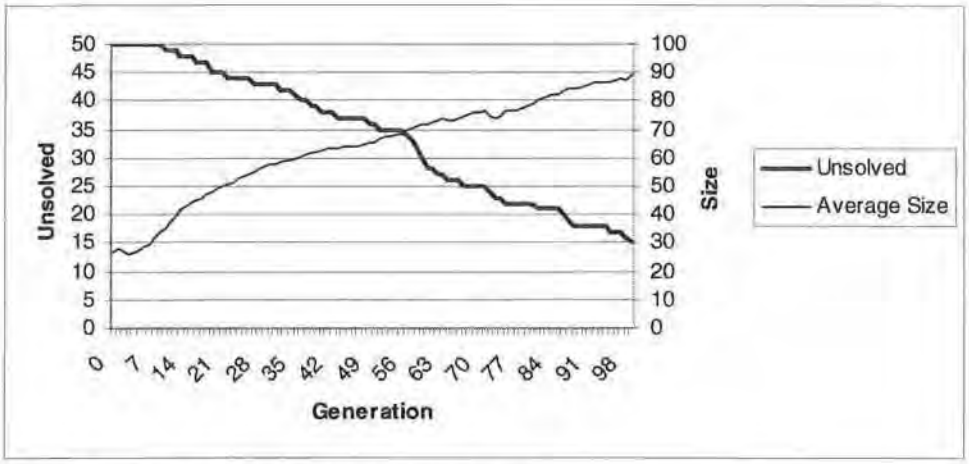


Figure 6.6: Result of hill climbing optimisation.

4.4.4 Hill climbing and removing inactive code

When code editing is combined with hill climbing, the results shown in figure 6.7 are achieved. The average size is reduced significantly to 60, but the chance to find a solution decreases to 54%.

4.4.5 Dynamic size limiting

In this case, the size of new individuals is restricted to 1.333 times the size of the best individual of the population (or to a size of 20 for the initial population). For small sizes of the best individual, the new maximum weight is set to at least 2 more than the size of the best individual, to ensure that evolution is not stopped (because the minimum increase in size of a binary tree is 2). Figure 6.8 illustrates the result of this approach. The average size of the individuals is reduced even more compared to the hill climbing approach, to a maximum size of 32. Additionally, all of the 50 runs were completed successfully at generation 74.

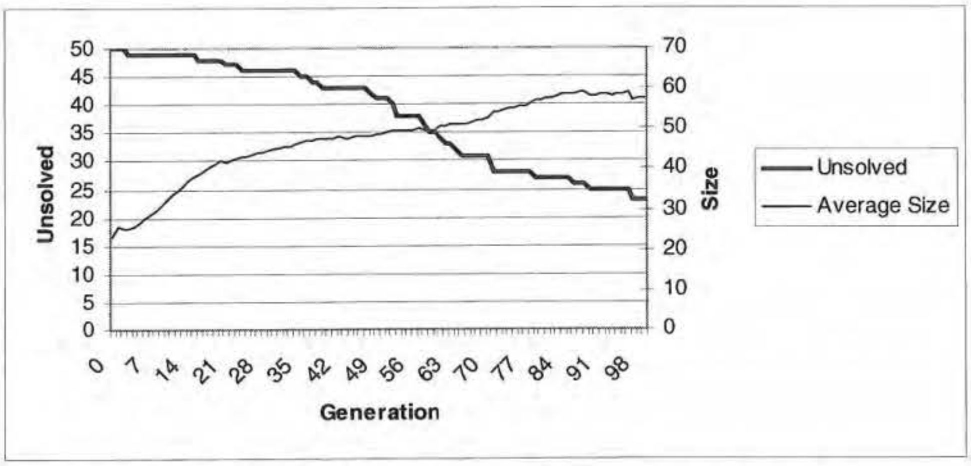


Figure 6.7: Result of combining hill climbing and removing inactive code.

4.4.6 Dynamic size limiting and hill climbing

By combining these two approaches, the average size of the individuals is reduced even more (to a size of 17), but the success rate decreases to 70%. These results seem to confirm that hill climbing is a successful way to reduce the size, at the cost of finding solutions slower. These results are shown in figure 6.9.

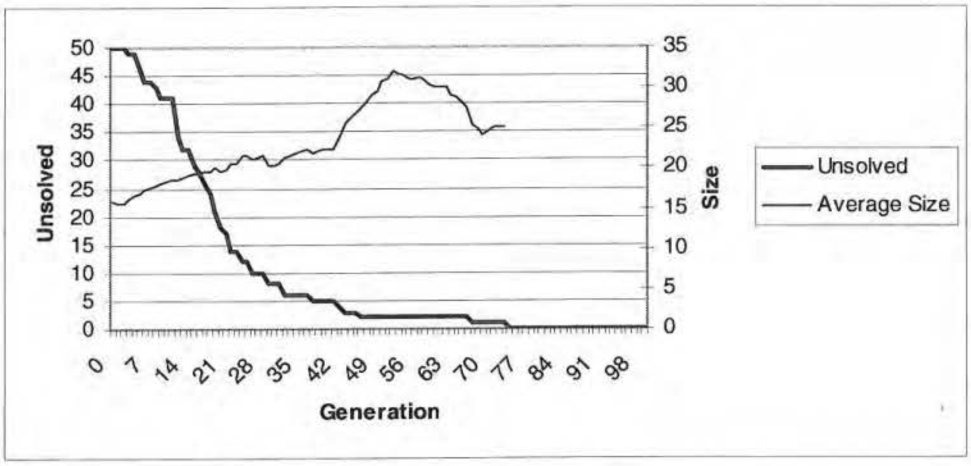


Figure 6.8: Result of dynamic size limiting.

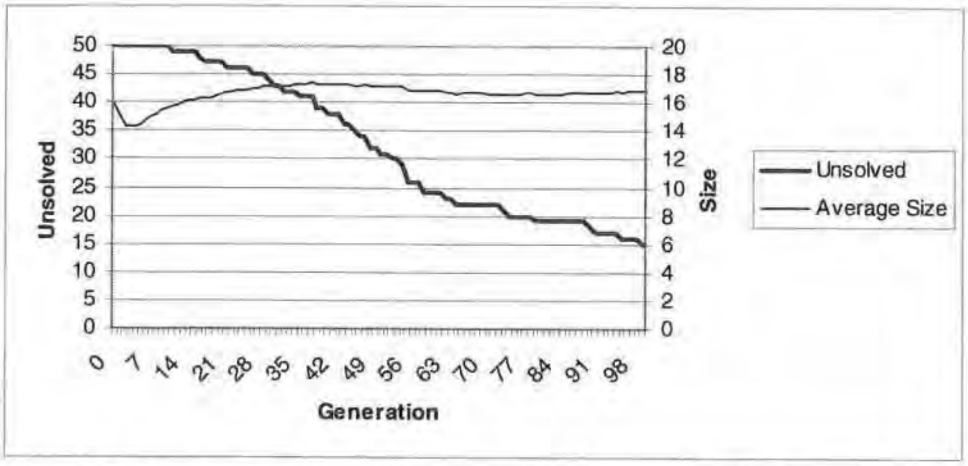


Figure 6.9: Result of combining dynamic size limiting and hill climbing.

4.4.7 Dynamic size limiting and removing inactive code

Combining these two optimizations gives similar results to dynamic size limiting, as shown in figure 6.10: the average size is reduced to 28, and 98% of the runs are successful after 61 generations. Only a single run failed to find a solution in less than 100 generations.

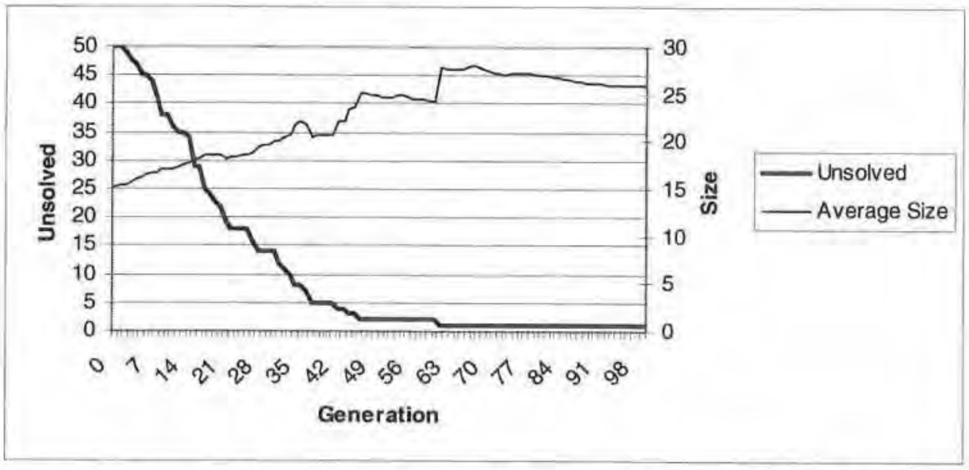


Figure 6.10: Result of combining dynamic size limiting and removing inactive code.

4.4.8 Combining all techniques

When all three optimization techniques are combined, the results of figure 6.11 are obtained. The average size of individuals remains small, but the success rate is only slightly better than the results of only combining dynamic size limiting and hill climbing or combining hill climbing and removing inactive code. However, it is significantly worse than the result of combining dynamic size limiting and removing inactive code. The combination of all techniques seems to be too restrictive to discover improvements.

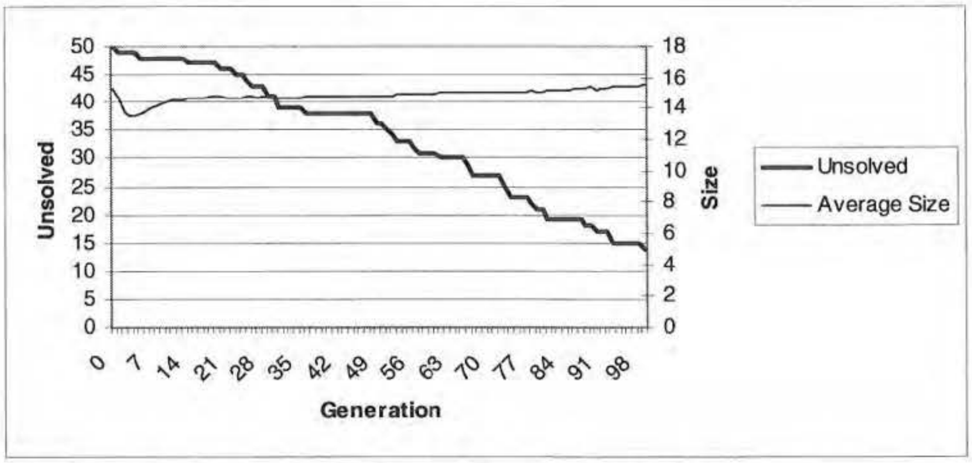


Figure 6.11: Result of combining all three optimization techniques.

4.5 Conclusions

In this section, a new approach to reduce bloat in genetic programming was introduced, and compared with other approaches. This technique imposes a dynamically changing maximum size on newly created individuals, based on the size of the best individual of the previous generation. This approach has benefits over other approaches because it is very simple and even improves the speed at which a solution is found in the symbolic regression test case. The technique will also be used later in this thesis on the AI planning test case and the Robocup domain. In these cases, the technique also maintains a small average population size.

A possible concern about the dynamic size limiting technique is that it may stop evolution when a small best individual is found, and better individuals are only possible for a size that is larger than the allowed limit. Therefore, when no better individual is discovered for several generations, the size limit is increased gradually until an improvement is found. Surprisingly, when evolution failed to find better individuals for several generations (around 20) and the maximum size limit was so large that it was essentially no longer present, bloat was still not observed, even after a large number of generations.

When excessive code growth is eliminated, it becomes possible to evolve a population over a large number of generations without becoming unmanageably large. However, evolution over a large number of generations often introduces the problem of premature convergence. In this case, a highly successful individual manages to create a large number of offspring that resemble their parent. As a result, a large part of the population consists of individuals that are very similar to each other. When this happens, the diversity of the population has been reduced too much and creating original new individuals becomes very difficult. Therefore, it is necessary that a method is used to maintain the diversity of a population. This issue will be discussed in the next chapter.

Chapter 7: Measuring and maintaining the diversity of a population

1 Introduction

An important problem encountered in evolutionary algorithms is that after several generations, the individuals of the population begin to resemble each other, or put in other words: the diversity of the population has decreased. This happens when a highly fit individual is used as a parent for most of the new members of the population, which also have above average fitness. As a result, it is possible that genetic material, needed for a complete solution, is removed from the population. If this happens, the population has converged prematurely.

To counter this problem, the size of the population is often set to a large value. In this case, it takes longer before the diversity of the population is lost. However, evaluation of a generation will also take longer in this case. Another solution is to use a method to increase the diversity of a population. Such a method must perform two tasks. First, it must be able to determine the similarity of the individuals in the population. This is usually done either with a distance metric between two individuals, or with a distance metric between an individual and the rest of the population. Secondly, the method must remove those individuals that are too similar to others, and/or add individuals that are different from the existing individuals of the population. In this chapter, we develop a method that measures the similarity of individuals with the rest of the population, and removes individuals that are too similar [71].

2 Related work

A typical method used in genetic algorithms to increase the diversity of a population is called fitness sharing [27]. When using this approach, similar individuals have to „share" their fitness value. This means that the fitness f_i of an individual is reduced by a value m_i, called a niche count. This value is defined by equation (7.1):

$$m_i = \sum_{j=1}^{p} S(d(i,j)) \tag{7.1}$$

In this equation, p is the population size, d is a distance metric between the individuals i and j, and S is a decreasing function, called the sharing function. A typical sharing function is given in equation (7.2):

$$S(d) = \begin{cases} 1 - \left(d/\sigma\right)^{\alpha} & \text{if } d \leq \sigma \\ 0 & \text{if } d > \sigma \end{cases} \tag{7.2}$$

Here, σ is the niche radius that is user-specified, and α is typically set to 1. Fitness sharing causes a decrease in the fitness of large groups of similar individuals. As a result, some of these individuals will be removed from the population and the fitness of the other members of the group will increase.

The disadvantages of this approach are that calculating the distance between all individuals is time consuming ($O(p^2)$), and it may be difficult to find an appropriate value for σ.

2.1 Distance measures for genetic programs

In genetic algorithms, the distance between two individuals can be measured by calculating the hamming distance between their bit string representations. In genetic programming however, where a variable length representation is used, this is not possible and more complex methods have to be employed.

In the time complexity formulas in this paragraph, n represents the average size of individuals and p represents the population size.

Keijzer [40] defines the distance between two individuals using the directed acyclic graph representation discussed in [29] and in section 6.3 of chapter 5. The distance is defined as the difference between the number of different nodes of the union of the two individuals and the number of subtrees the individuals have in common, shown in equation (7.3).

$$\delta_{dag}(X,Y) = |D(X) \cup D(Y)| - |D(X) \cap D(Y)| \tag{7.3}$$

In this equation, $D(X)$ represents the number of different subtrees present in X. This measure can easily be calculated when using the directed acyclic graph representation in $O(n)$.

Mawhinney uses the Unix diff program to calculate the difference between individuals [62]. This gives a rough measure of the syntactic similarity between two individuals. Individuals that are similar to other individuals are replaced by new random individuals. To calculate this similarity measure, every combination of two individuals is compared using the diff function. Thus, comparing two individuals has a time complexity of $O(n)$, and processing the entire population has a time complexity of $O(p^2n)$.

In [41], Keller and Banzhaf use the "edit distance" to measure the difference between two individuals. This distance measures the number of primitive edit operations needed to transform one individual into another. The primitive actions used are divided in two categories. The first category involves the adding or removing of a child node, and the second category involves modifying the type of a node. A two-dimensional vector can then be associated with every individual, representing the number of operations needed to transform an individual with only a single node to the given individual. The first value of the vector represents the number of operations of the first category, and the second value the number of operations of the second category. Consequently, every individual of a population can be represented by a point in $\mathbb{R}^2_{+0}$ within the bounding rectangle containing all these points. The largest area rectangle containing none of these points is calculated. The diversity of the population is then represented as 1 – (area of largest area rectangle)/(area of the largest possible bounding rectangle). When the diversity of a population drops below a threshold, individuals that have the same position vector as another individual are replaced by a new individual that approximates the position of the largest area rectangle. Processing the entire population has a time complexity of $O(p^*(n + \log p))$.

In [91], Rosca uses the fitness and "expanded structural complexity" of individuals to determine similarity between individuals. Because these features are computed during the evaluation of the individual without significant extra cost, there is no added time complexity for using this method. However, because structurally different individuals can still have identical fitness and/or expanded structural complexity, this method is not very accurate in detecting similar individuals.

Nienhuys-Cheng [76] defines a metric between two nodes p and q with arity of n and m respectively in equation (7.4):

$$d(p(s_1,s_2,\ldots,s_n),q(t_1,t_2,\ldots,t_m))=\begin{cases}1 & \text{if } p\neq q \text{ or } n\neq m\\ \frac{1}{2n}\sum_{i=1}^{n}d(s_i,t_i) & \text{if } p=q\end{cases} \tag{7.4}$$

This distance between the individuals T_1 and T_2 can be computed in O(min($|T_1|$,$|T_2|$)), where $|T_i|$ is the number of nodes of T_i.

Ekárt uses a method based on the structure of individuals to measure the difference between them [23]. The method works in three steps:

- The two individuals are placed on an identical tree structure, adding empty child nodes where necessary.
- The distances between the types of the nodes at identical position in the identical tree structure is calculated.
- These distances are combined in a weighted sum to result in the distance between the two individuals.

Only ordered binary trees were considered, because the time complexity of aligning unordered trees is exponential. In this case, comparing two individuals has a time complexity of O(n). The diversity of a population is defined as the mean distance of two individuals of that population, and has a time complexity of O(p^2n). The fitness of a population is maintained through fitness sharing: individuals close to each other share the same fitness value, meaning their fitness decreases as more individuals are in their neighborhood. This causes other (and therefore more diverse) individuals to be selected by genetic operators, which tend to create new individuals in the less populated environments and remove individuals in crowded areas.

Most of the distance measures between two individuals described in this paragraph are based on the syntactic similarity between two individuals. This similarity is determined by comparing the tree representations of the individuals from root node to leave nodes. As a result, two individuals that have a similar upper part are closer to each other than two individuals with different types of root node. However, two syntactically identical upper parts of a tree can have completely different behaviors, depending on the results of the lower parts of the trees. After a crossover operation between two individuals, the upper part of one of the individuals will be transferred to the new individual, together with some of the lower part of the other individual. If the transferred subtree is not an intron or identical to the replaced subtree, the behavior of the new individual will probably be different from their parent. It is therefore important that the diversity of the lower subtrees in the population is also maintained. Otherwise, crossover may tend to produce similar structures because the replaced subtrees will often be similar. A method that determines the diversity of these lower subtrees will be introduced in section 3.

2.2 Other diversity measures for genetic programming

While using a distance measure between individuals to measure the diversity of a population allows the re-use of methods designed for genetic algorithms, other methods are possible as well. This includes our new algorithm, which will be described in the next paragraph. In the remainder of this section, we will describe the work related to these diversity measures.

In [52], Langdon and Poli detect similar individuals by comparing their fitness values on identical test cases. A fitness penalty was added to offspring that was similar to their parents. Using this technique, the diversity of the population was increased and bloat was reduced by 50%, while performance decreased only slightly.

In [111], Wineberg and Oppacher measure the randomness of nodes at given positions (called a locus) in an individual. Randomness of a given locus is calculated using the entropy measure in equation (7.5):

$$H(L) = \sum_{i \geq 1} f_i \log \frac{1}{f_i} \tag{7.5}$$

This measure is called the genic diversity of a locus. In this equation, L is a random variable over the range $R = \{g_1, g_2, \ldots\}$, where R is the alphabet of the possible genes. $f_i = \text{prob}(L = g_i)$ is the frequency of a gene g_i occurring at the specified node, looking across the entire population. The diversity of the individual is then calculated as the average of the genic diversity across all loci of the individual.

Rosca [92] also uses entropy as a measure of diversity of a population. In this case, the population is divided in a set of partitions that have similar behavior, for example an equal number of hits in a parity problem. The values f_i of equation (7.5) are the fractions of individuals that are part of the partition i.

In [19], De Jong uses a multi-objective function to reduce bloat and increase diversity simultaneously. The distance measure between two individuals that was used is the sum of different nodes at overlapping locations divided by the size of the tree, as in equation (7.4).

3 Sharing identical subtrees

The distance measure between an individual and the rest of the population developed by us relies on the directed acyclic graph representation of a population [29]. In this representation, when two individuals share an identical subtree, this identical subtree will be represented by the same object in memory. For example, in figure 7.1 the individuals (3*(2+X)) and ((2+X)-X) share the identical subtree (2+X). Also, the node (X) occurs two times in the second individual and is also represented by the same object.

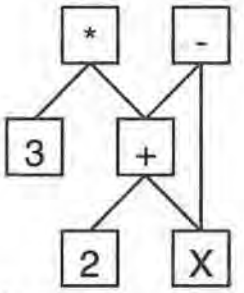


Figure 7.1: Sharing identical subtrees between individuals with the directed acyclic graph representation.

Using this representation has several advantages. Obviously, by re-using identical structures, the memory required to store the entire population decreases. Also, when the effects of the nodes have no side effects when executed, the results of calculations of shared subtrees can be re-used, potentially saving significant processing time. However, the most important advantage related to diversity is that it is easy to calculate the similarity between individuals. This is demonstrated in the next section.

4 Calculating the added diversity of an individual

Most of the techniques discussed in section 2 use a distance measure between two individuals, and determine the diversity of the population by calculating the distance between every pair of individuals in the population. The time complexity of this operation is $O(p^2)$, where p is the population size. This has to be multiplied with the time it takes to calculate the distance measure, which is usually $O(n)$, giving a total time complexity of $O(np^2)$. In this case, n is the size of the individual.

The algorithm presented here compares an individual to the entire set of previously processed individuals, and has a time complexity of $O(np)$. The only extra cost needed for the algorithm to work comes from the need to store the entire population as an acyclic directed graph. This increases the cost to construct an individual from O(*n*) to O(*n* log *n*). As a result, the time needed to construct, evaluate and remove individuals with low diversity is O((*n* log *n*)*p*).

After a generation has been completed, the algorithm will process the entire population and remove individuals that do not add sufficient diversity to it. The algorithm is presented below:

```
* Sort the individuals of the population by fitness. If
two individuals have equal fitness, order them by size
(smaller is better).
* Unmark all nodes of all individuals.
* Iterate over all individuals, starting with the fittest:
 * The added diversity of this individual =
   (number of unmarked nodes)/(number of nodes)
 * If the individual has sufficient diversity, mark all
   nodes of the individual. Otherwise, remove it from the
   population.
```

Because the identical nodes of individuals are shared between all the population's individuals, marking one individual's nodes will mark nodes of several individuals in the rest of the population. It is therefore possible to efficiently compare an individual with all previously tested individuals.

The algorithm tests how much new genetic material would be added to the population if the individual would be added to the population. Because no nodes are marked at the beginning of the algorithm, the fittest individual will always have a diversity of 1 (the highest value), and will always be accepted. If an entire individual is a part of a fitter individual, its fitness will be 0 (the lowest value) and it will always be rejected.

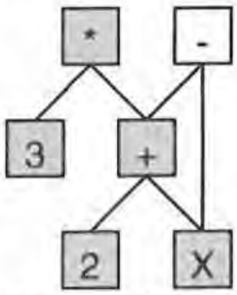


Figure 7.2: Nodes are marked to determine the similarity between individuals.

As an example, the similarity between the two individuals described in figure 7.1 will be determined. First, because the first individual's diversity is 1, all its nodes will be marked. Because the similar nodes of the two individuals are shared, this means that these nodes in the other individual are marked as well (see figure 7.2). To calculate how different the second individual is from the first, the marked nodes in the second individual are counted (4 in this example, because the X node occurs twice). The second individual contains 5 nodes, so the similarity of the second node can be defined as the number of marked nodes divided by the total number of nodes (in this example 0.8).

Note that this similarity measure is not a distance measure, since the similarity between *a* and *b* is not equal to the similarity between *b* and *a*. In the above example, if the second individual's nodes are marked first, the similarity of the first individual would be 0.6 instead of 0.8 (see figure 7.3).

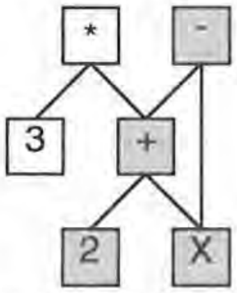


Figure 7.3: Similarity measure is not symmetric.

5 Removing individuals with low diversity

Determining when an individual will be allowed to remain in the population depends on the added diversity of the individual, and on its fitness. Very fit individual do not require much added diversity. Also, very unfit individuals that have much genetic code that does not appear anywhere else in the population can also be allowed in the population, because they introduce variety.

In our current implementation, an individual is accepted in the population when equation (7.6) is satisfied:

$$Diversity \geq \left(\frac{i}{PopulationSize} \right)^{D} \qquad (7.6)$$

In this equation, Diversity is the diversity calculated in section 4, *D* is a constant used to balance the effect of diversity versus fitness, and *i* is the rank of the individual in the population, where the fittest individual has a rank of 0. Using this criterion, the fittest individuals do not require much added diversity, while the lowest fitness individuals are only accepted when their diversity is close to 1. In the experiments performed, the constant *D* was set to 1. Varying this constant between 0.5 and 2 did not seem to affect the results significantly.

6 Experimental results

The effect of the diversity measure was tested on the problems of symbolic regression (described in section 5.1 of chapter 5) and AI planning (described in section 5.4 in chapter 5). The experiments also used the removal of inactive code and dynamic size limiting optimizations described in chapter 6.

6.1 Symbolic regression

The target function used was $x^5 - 2x^3 + x$, using the function set {+, -, *, /, x}. The following parameters were used: 70% crossover, 30% combination, 1% mutation, population size 200. A total of 100 runs were performed for a maximum of 100 generations or until a perfect solution was found. Figure 7.4 lists the number of unfinished runs after a given generation with and without using the diversity measure.

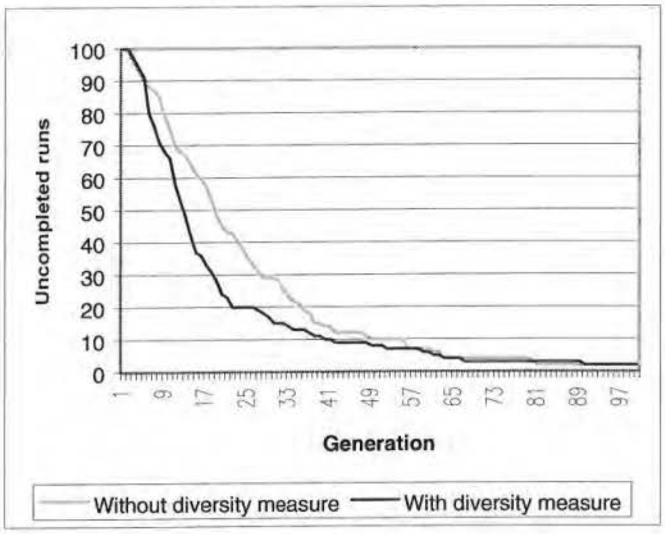


Figure 7.4: Effect of diversity on symbolic regression problem.

When the individuals with low diversity are removed from the population, a solution is discovered faster (at generation 21, 77% of the runs have found a solution versus 56% when increasing diversity). However, the overall success rate is similar (97%). It should also be noted that in early generations, the diversity measure removes most of the individuals from the population and keep only between 5 and 10 individuals. Because these individuals still contain almost all the genetic material of the entire population, new combinations of genetic material will be discovered faster. This leads to the faster convergence observed in the experiment. These results also suggest that this diversity measure allows the use of a much smaller population size, which may be very useful in domains where evaluation of individuals is extremely time consuming.

6.2 AI planning

The test case used was the briefcase problem, with 5 objects, 5 briefcases, and 5 locations. The non-terminal set was {put_in, take_out, move, prog2, prog3}, and the terminal set was {0_1, ..., 0_5, L_1, ..., L_5, B_1, ..., B_5}. The settings used were 30% crossover, 30% mutation, 40% combination, and a population size of 200. A total of 100 runs were performed, to a maximum of 200 generations or until a perfect solution was found. Figure 7.5 lists the results of using the diversity measure on this problem. When the diversity measure was used, all the runs were able to find a solution after 181 generations, while otherwise only 72% of the runs were successful after 200 generations. The convergence speed was also significantly higher in this case. It is not known whether the runs that did not use the diversity measure would eventually all find a solution.

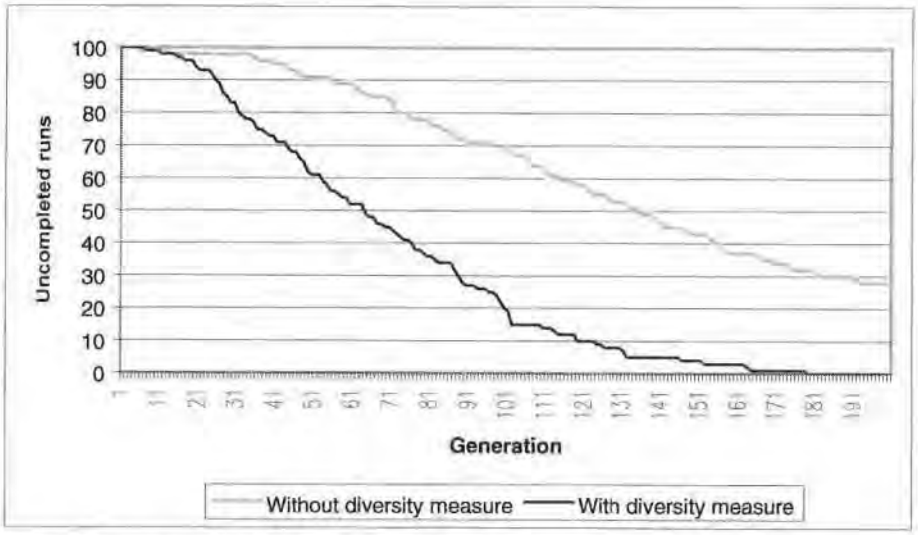


Figure 7.5: Result of using diversity on the AI planning problem.

7 Conclusion

This chapter introduced a method to measure the diversity an individual adds to the rest of the population. This measure concentrates on detecting similar subtrees in the lower parts of the tree representation of an individual. The method is based on the directed acyclic graph representation of a population to efficiently determine the similarities between two individuals or between an individual and an entire population. When using the diversity measure to remove individuals that do not contribute enough new genetic material to the population, an increase in convergence speed was observed on the problems of symbolic regression and AI planning. This appears to indicate that the evolutionary process is more efficient at discovering new structures. Moreover, the use of the diversity measure to remove individuals from the population also leads to much smaller populations. These smaller populations were still able to find equally good solutions as fast or even faster as a large population. This suggests that the use of the diversity measure is a good way to reduce the population size, which is very helpful in domains where the evaluation of individuals is very time consuming.

Chapter 8: Applying evolutionary computing to Robocup

In this chapter, we will use genetic algorithms and genetic programming to train a team of agents in the Robocup domain, described in chapter 4. All players of the team will be controlled by an identical agent program, but their behavior may depend on their player number. For example, the position of a player is determined by this number, as shown in figure 8.1. The use of identical programs has the advantage that any player can perform the actions of any other player when necessary, and all the agents can be trained simultaneously.

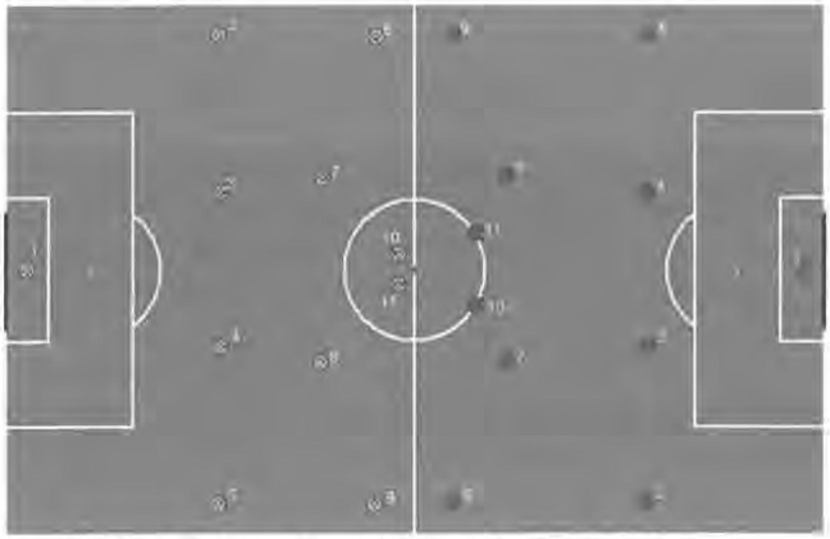


Figure 8.1: Positions of players depending on their player numbers.

When training a team, every player of the team must learn when to execute which action, depending on the detected and predicted state of the environment. This process is called action selection, and is shown in figure 8.2. Various techniques can be used to learn action selection. Two techniques will be described in this chapter. In section 1, we will use a reactive action selection network, linking the results of sensor inputs and predictions directly to action selection. The weights of the links in this network can then be trained using a genetic algorithm. In section 2, we use genetic programming to create a program that calculates a movement or kicking vector to control a player.

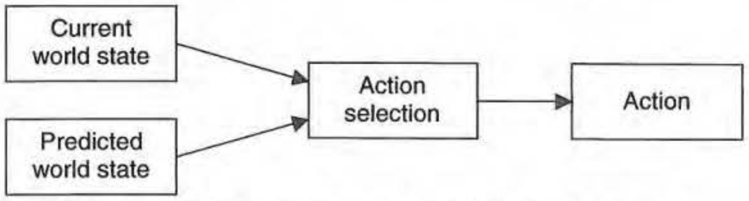


Figure 8.2: Action selection uses current and predicted world states.

1 Using genetic algorithms to train a reactive action selection network

In a first attempt to implement action selection, an action selection network was developed. This network consists, somewhat like a neural network, of nodes that accept and transmit values. The terminal nodes of this network transmit a value that depends on the received sensor information of the player. Other nodes process the values from other nodes and combine this to a new value. Some nodes can execute an action when their value exceeds a threshold.

This structure has the following advantages:

- Because the terminal nodes are directly linked to the sensors of a player, the action selection network responds immediately to changes in the environment.
- It is possible to construct several behavior groups of related actions. Different behavior groups can be divided by a node that activates a group based on the result of a sensor or other node.

1.1 Description of the action selection network

The action selection network is shown in figure 8.3. Every link in the network contains a weight value, by which the output of a node is multiplied. These values can be trained by a genetic algorithm. The nodes in the second column represent all the possible actions of a player. Terminal nodes are directly linked to sensors, or predictions of the sensor values. Mutex nodes will activate the child node that returns the highest value, and deactivate the rest. Switch nodes will activate one of two child nodes, depending on the value of a third child node. The other nodes will be discussed briefly in the next sections, while a more detailed description is available in [67].

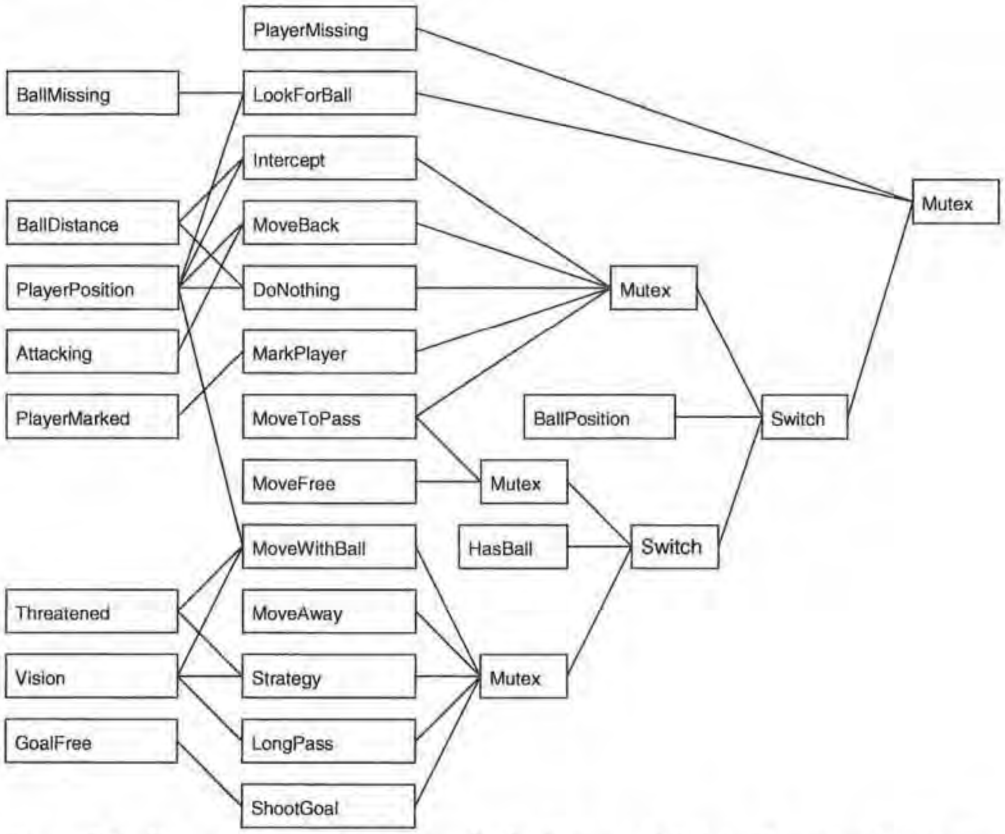


Figure 8.3: The action selection network. Actions are listed in the second column. Terminal nodes are directly linked to sensors.

1.1.1 Sensor nodes

The terminal nodes of the network indicate the current state of the environment of a player. These nodes are:

- **BallMissing:** This node returns a positive value when the ball has not been observed for several time steps. As the current position of the ball plays an important role in the action selection, it is necessary that accurate information about its position is known. The player will start looking for the ball if this node is active.

- **PlayerMissing:** In some rare cases, it is possible that insufficient reference points are observed to determine the position of the player on the field. In this case, the player will turn around to look for reference points.
- **BallDistance:** This node returns the distance from the player to the ball.
- **PlayerPosition:** Each player is assigned an area on the field in which that player is supposed to play. This area depends on the player number assigned to the player when connecting to the soccer simulator. When the player is inside this area, the node will return the distance of the player to the edge of the area. When the player is outside this area, a negative value of minus the distance to the edge of the are will be returned. This node encourages a player to stay inside its assigned position.
- **Attacking:** This node indicates whether the player is currently involved in an attack. If the player is attacking, this node will reduce the effect of the PlayerPosition node to ensure that the player will not suddenly turn around when moving with the ball to the goal. This node introduces persistence to the action selection.
- **PlayerMarked:** This node actually consists of 11 nodes, each one observing an opponent. If a teammate is close to the targeted opponent, and the opponent is in a position to receive a pass, that opponent is considered to be 'marked'. Otherwise, this node will return a positive value and this player may decide to mark the targeted opponent.
- **GoalFree:** This node returns a value that indicates the number of players standing between this player and the goal. If this value is low enough, the player may decide to shoot at the goal.
- **Threatened:** This node returns the number of opponents that are close to this player. The player may decide to pass the ball to a safer teammate when it is threatened by several opponents.
- **Vision:** This node uses the vision buffer, discussed in section 3 of chapter 4, to determine how safe it is to move or pass, based on the calculated danger values.
- **MoveToPass:** When a player gives a pass to a teammate, a message is sent to this player to notify him that a pass is given. If a player receives such a message, this node will return a positive value, activating the behavior that will cause the player to move to intercept the ball.
- **MoveAway:** This node is activated when the player is standing in the way of a teammate that controls the ball. This causes the player to move away from the ball to allow the teammate to move along.
- **MoveFree:** When the player is marked by an opponent, this node will be activated. The player will then try to move away from the player so it may receive a pass if necessary.

- **HasBall:** This node is activated when the player controls the ball and enables the behaviors that are only possible when the player controls the ball, such as passing the ball.
- **BallPosition:** This node is activated when a teammate controls the ball. This node splits the offensive behaviors from the defensive behaviors of the player.

1.1.2 Behavior nodes

Some of the nodes in the action selection network have a behavior attached to them, that is executed when the node is active. These nodes are:

- **LookForBall:** This behavior will cause the player to turn around, looking for the ball. However, when the MoveBack behavior is also active, these behaviors will interfere with each other. A negative connection to this behavior will prevent it from interfering with the MoveBack behavior.
- **Intercept:** This behavior will try to intercept the ball.
- **MoveBack:** This behavior will cause the player to return to his position on the field when it is not currently involved in an attack.
- **DoNothing:** When the player is at its position and the ball is far away, it does not have to do anything.
- **MarkPlayer:** This behavior will cause the player to mark an opponent by moving between the opponent and the ball. This will make it difficult for the opposing team to pass to that opponent.
- **MoveWithBall:** This behavior will cause the player to dribble with the ball in the general direction of the opponent's goal. This behavior can be stimulated when the player is in its position area, the player is not threatened and a safe direction is available in the vision buffer.
- **Strategy:** Sometimes, a group of players will execute a predefined sequence of actions such as a one-two combination. These sequences of actions are part of the highest layer of learning discussed in section 2.1 of chapter 4. This behavior determines when such a sequence will be started. While executing such a sequence, the action selection network is not used until the action is completed or has failed.
- **LongPass:** This behavior determines when a pass is given to another player. This depends on the safety values calculated in the vision buffer.
- **ShootToGoal:** This behavior decides when a shot at the goal is attempted. Two criteria must be met: the goal must be close enough, and the number of players between the goal and the ball must be small.

1.2 Training the action selection network

When all the behaviors are implemented and the action selection network is constructed, the action selection itself must be trained by assigning weights to the links in the network. In the action selection network described in the previous sections, it is not hard to determine whether nodes have positive or negative effects on other nodes. Positive effects are represented by positive weight values, while negative effects use negative values. However, the actual magnitude of these weights is much more difficult to estimate. If the weights of the links are stored as a string of numbers, genetic algorithms can be used to train these weights.

1.2.1 Representation and fitness function of the genetic algorithm

The genome representing the weight values of the action selection network consists of a fixed-length string of floating point values. The primitive elements of the string are floating point numbers, so crossover points will never be selected inside a value. Also, because the signs of the weights are fixed in the network, the genetic operators will never change the sign of the weights.

The weights of a team are evaluated by playing a match against another team. The other team can either be a fixed reference team, or another team from the population. After the match is played, a fitness value is calculated using equation (8.1):

$$\begin{cases} -7.5T & \text{if } S_{me} = 0 \text{ and } S_{opponent} = 0 \\ (S_{me} - S_{opponent})(1 + 0.1(S_{me} + S_{opponent}))(2T - E) & \text{otherwise} \end{cases} \tag{8.1}$$

In this equation, S_{me} and $S_{opponent}$ represent the scores of the evaluated team and the opposing team, T is the maximum duration of a match, and E is the time the game actually lasted. The initial population will contain a lot of teams that simply stand still on the field. When two of these teams play against each other, the final score will be 0-0. To remove these teams as quickly as possible from the population, both are given the lowest possible fitness value. This is handled by the first part of equation (8.1).

Normally, one or both of the teams will score several goals and the second part of equation (8.1) will calculate the fitness value. The first part of the equation will ensure that the winner of the match will receive a positive value, and the loser will receive a negative value. A draw gives a fitness value of 0. The second part of the equation encourages the scoring of goals. To decrease the evaluation time, when a combined total of 5 goals are scored by both sides before the end of the official play time, a match will end. The third part of the equation rewards matches that quickly score 5 times.

The genetic algorithm uses the following settings:

- Population size of 64.
- Elitism: the best 20% are immediately transferred to the next generation and the lowest 20% are immediately removed from the population.
- The remaining 80% of the next generation are constructed with single-point crossover, and 5% of these new individual will also be mutated. Mutation will cause one floating point value to be multiplied or divided by a value chosen randomly from the interval [1, 2].

1.2.2 Training against a fixed reference team of 6 players

In a first experiment (using version 3.x of the simulator), a team of only 6 players was trained against a fixed reference team of 6 players. The advantage of using smaller teams is that the computational needs to run only two teams is much lower, and strategies for teams are easier to learn. The team was trained over five generations against the reference team. The average fitness of the population is shown in figure 8.4. The number of wins, losses and ties of the teams in the population is displayed in figure 8.5. As the number of wins and ties increases over time, the average fitness of the teams improves after several generations.

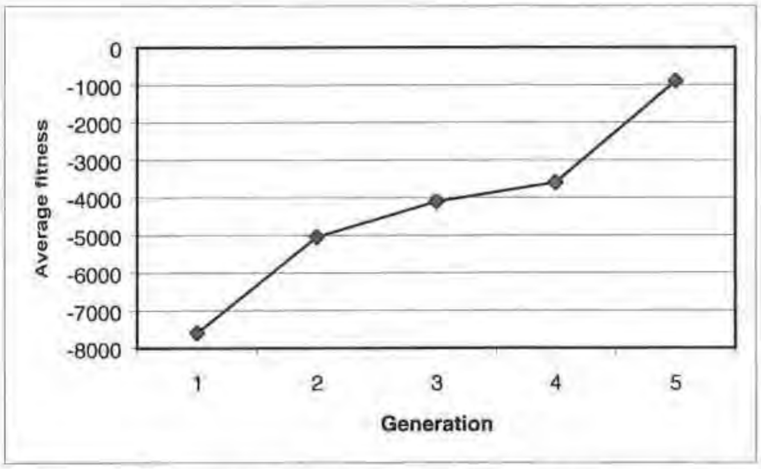


Figure 8.4: Average fitness of the population when training a team of 6 players against a fixed reference team.

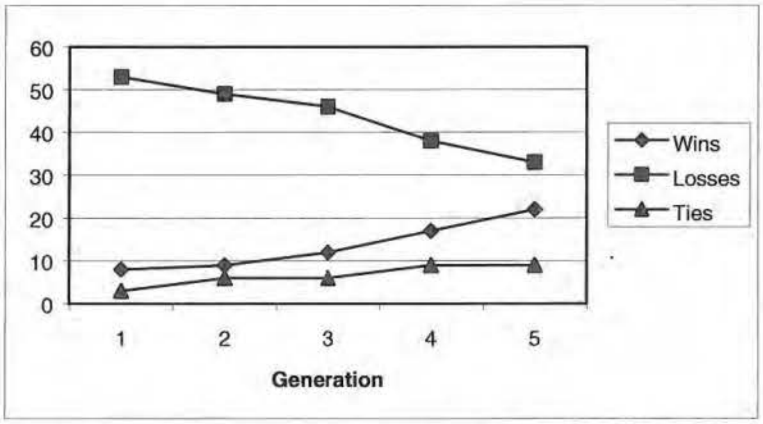


Figure 8.5: Number of wins, losses and ties of the teams in a population when training a team of 6 players against a fixed reference team.

When the experiment was repeated using complete teams of 11 players, none of the teams of the initial population was able to defeat the reference team after 4 generations, and as a result the fitness values of all teams were very low. After several generations, no improvement was noticeable. A possible explanation is that as all individuals have a similar fitness value, evolution was halted. The experiment had to be aborted after the installation of a new version (4.18) of the soccer simulator, needed to be able to participate to the Robocup world cup. Because of this, the players of both teams had to be modified considerably and were incompatible with previous versions.

1.2.3 Training using co-evolution with teams of 11 players

A second experiment was performed where co-evolution was used instead of using a fixed reference team, using version 4.18 of the soccer simulator. In this experiment, two separate populations of teams were maintained, and teams from both populations play against each other. One population evolves the left teams, while the other population evolves the right teams. After a generation, the best result of a population was compared with the best result of the initial population. Because of time constraints, the experiment could only be run for 5 generations. The fitness of the best individuals is shown in figure 8.6. The results of this limited experiment are somewhat disappointing, as the best individual does not seem to improve significantly. This may be caused by the inaccuracy of the fitness function that uses the results of a single match to calculate a result, which uses the results of a single match to calculate a result. These results can change significantly when two matches are played between the same teams.

Because training against a fixed reference team appeared to be more promising, another more extensive experiment with a fixed reference team was performed.

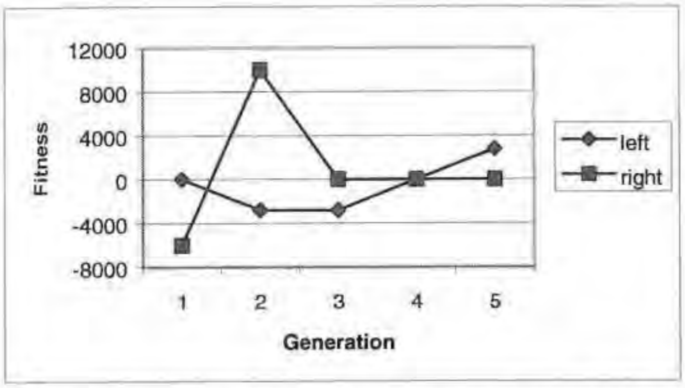


Figure 8.6: Fitness values of the best individual in the left and right population after co-evolution.

1.2.4 Training against a fixed reference team of 11 players

The experiment of training against a fixed reference team of 11 players was repeated using modified versions of the players for version 4.18 of the soccer simulator. This experiment was run for 12 generations, and the results are displayed in figure 8.7 and figure 8.8. There are no results for generation 3 because the file containing the results of this generation was corrupted after the experiment. In this case, a number of teams are able to defeat the reference team in the initial population and this number increases slowly.

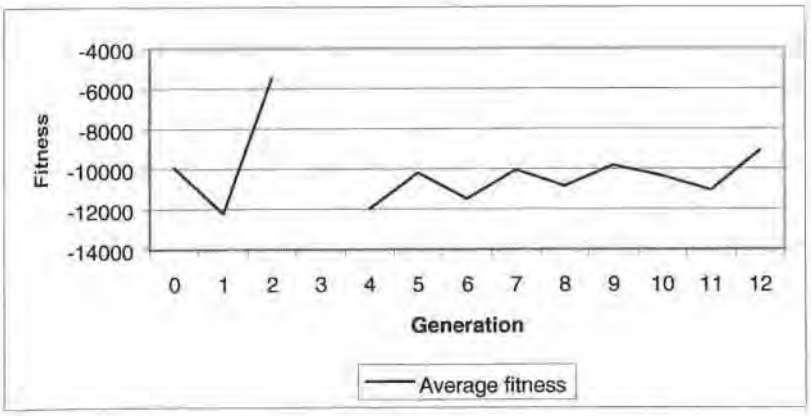


Figure 8.7: Average fitness of the population when training a team of 11 players against a fixed reference team.

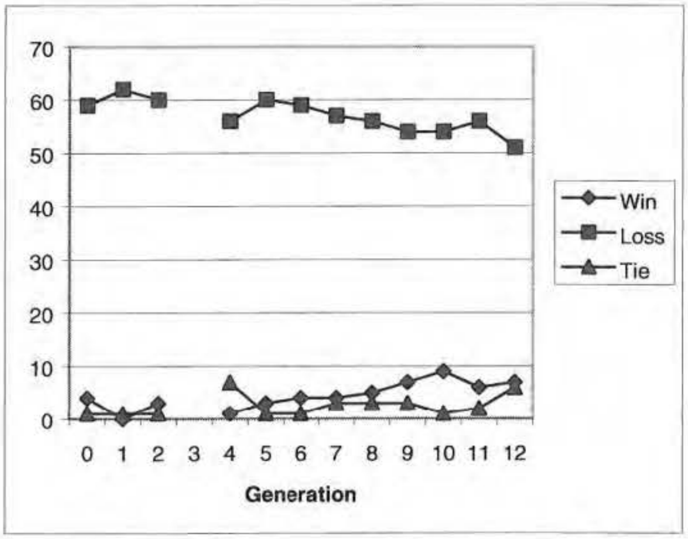


Figure 8.8: Number of wins, losses and ties of the teams in a population when training a team of 11 players against a fixed reference team.

1.3 Conclusion

In the experiments described above, a team of Robocup players was trained by optimizing a set of weights of an action selection network using genetic algorithms. After several generations, the performance of the population of teams grows slowly. Due to the fixed structure of the action selection network, however, the players of the evolved teams will tend to play using similar strategies. This may be changed by giving the evolutionary process more flexibility to implement action selection. In the remainder of this chapter, genetic programming will be used to construct a program to control the player.

2 Using genetic programming to learn action selection

Because the Robocup simulator is becoming more realistic over time and new functionality is added, Robocup players and teams have to be modified after every modification. As a result, hand coded behaviors have to be adapted to these changes and the use of new commands and sensor readings must be incorporated in the action selection of the players. Also, the behaviors and strategies of hand coded players will be inspired by how humans think soccer should be played. An unbiased learning technique on the other hand might discover completely different strategies and would automatically adapt to any changes in the simulator environment. In this section, the use of genetic programming to evolve the behavior of a team of soccer team will be studied.

2.1 Related work

The first team developed by genetic programming to enter the Robocup competition was developed by Luke et al. in 1997 [56][60][61]. Because of the complexity of the Robocup domain, several specific problems and challenges had to be solved. A first problem was the long time needed to evaluate a population of teams. The fitness value of a team is calculated using co-evolution by letting the teams of a population play against each other. The advantage of co-evolution is a better generalization compared to evaluation against a fixed reference team. Typically, a co-evolution problem needs about 100.000 evaluations to evolve to a good result [59]. As a match on the Robocup simulator takes 10 minutes, this would lead to an impractical evaluation time. As a result, several methods were used to reduce this evaluation time:

- Matches were executed in parallel on a supercomputer.
- The duration of a match was reduced from 10 minutes to between 20 seconds and 1 minute.
- The population size and number of generations were reduced. Between 100 and 400 individuals were evolved over 52 generations. (Note that "individual" refers to an entire team.)
- Higher level primitives were used that are biased towards playing soccer.
- Simultaneous runs with different genome structures were evolved in parallel, making it easier to choose an appropriate structure later.
- After 40 generations, the population was seeded with the best individuals discovered so far. The run was then continued for 12 generations.

Another problem of the Robocup domain are border conditions. For example, there is no purpose in executing a kick command when a player is not within reach of the ball. This problem was solved by splitting the genome in two parts. The first part would calculate a kick vector, and this part is executed when the ball is within reach of the player. This vector causes the ball to be kicked to the specified position. The second part calculates a move vector and is executed when the ball can be observed but is not within reach. This causes the player to move and turn towards the specified position. If the ball can not be observed, the player would turn around in an attempt to locate the ball, and no part of the genome is executed.

Base	Boolean	squad1()
		opp-closer()
		mate-closer()
		ofme(Integer)
		ofhome(Integer)
		ofgoal(Integer)
		opponent-close(Integer)

	Integer	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

	MoveVector	home()
		ball()
		findball()
		block-goal()
		away-mates()
		away-opps()
		home-of(Integer)
		block-near-opp (MoveVector)
		mate(Integer, MoveVector)
		weight-+(Integer, MoveVector, MoveVector)
		if-v(Boolean, MoveVector, MoveVector)
		sight(MoveVector)

	KickVector	far-mate(Integer, KickVector)
		mate-m(Integer, Integer, KickVector)
		kick-goal(Integer, KickVector)
		dribble(Integer, KickVector)
		kick-goal!()
		far-mate!()
		kick-clear()
		kick-if(Boolean, KickVector, KickVector)

Table 8.1: Hierarchy of STGP primitives used by Luke et al.

A third problem is the credit assignment problem: which player of the team is responsible for the success or failure of the team, and which part of the genome was responsible for it? This problem can be avoided by representing the entire team by an individual. There are three ways to represent an entire team in a single genome. The first is to use a homogeneous team where every player uses the same program. A second method is the use of heterogeneous teams where every player is represented by a different program. The advantage of homogeneous teams is that a solution may be found more easily as less code must be evolved, but the team will not have specialized players. An intermediary solution is to divide the team in several groups of homogeneous players such as defenders and attackers. This allows specialization at a reduced number of programs that must be evolved. When heterogeneous teams are used, [59] demonstrated that restricted breeding was useful for promoting specialization. When using restricted breeding, genetic materiel is exchanged only between similar parts of individuals.

The function set used by the genetic programming system contains several soccer-related primitives. Some of these primitives were themselves evolved using genetic programming. The set of primitives uses strongly typed genetic programming, and the hierarchy of these primitives is shown in table 8.1.

A description of these primitives is given below. The value *max* is the maximum distance of kicking and is set to 3.5.

- squad1(): Test whether this player is the first player in its group.
- opp-closer(): Test whether an opponent is closer to the ball than this player.
- mate-closer(): Test whether a teammate is closer to the ball than this player.
- ofme(Integer *i*): Test whether the ball is closer than *i/max* from this player.
- ofhome(Integer *i*): Test whether the ball is closer than *i/max* of the player's home position.
- ofgoal(Integer *i*): Test whether the ball is closer than *i/max* from the goal.
- opponent-close(Integer *i*): Test whether an opponent is closer than *max/1.5i* to this player.
- home(): Returns a vector to the home position of this player.
- ball(): Returns a vector to the position of the ball.
- findball(): Returns a zero-length vector to the ball.
- block-goal(): Returns a vector towards the closest point on the line segment between the ball and the player's goal.
- away-mates(): Returns a vector away from the observed teammates.
- away-opps(): Returns a vector away from the observed opponents.
- home-of(Integer *i*): Returns a vector to the home position of teammate *i*.
- block-near-opp (MoveVector *m*): Returns a vector to the closest point on the line between the ball and the nearest observed opponent. If no opponents are observed, *m* is returned instead.
- mate(Integer *i*, MoveVector *m*): Returns a vector to teammate *i*. If the position of this teammate is unknown, *m* is returned instead.

- weight-+(Integer *i*, MoveVector m_1, MoveVector m_2): Returns the vector $(i*m_1 + (9 - i)*m_2)/9$.
- if-v(Boolean *b*, MoveVector m_1, MoveVector m_2): Returns m_1 if *b* is true, or m_2 otherwise.
- sight(MoveVector *m*): Returns the vector *m*, but rotated just enough to keep the ball in sight.
- far-mate(Integer *i*, KickVector *k*): Returns a vector to the most offensive-positioned teammate who can receive the ball with at least $(i + 1)/10$ probability, or returns *k* instead if no such teammate can be observed.
- mate-m(Integer i_1, Integer i_2, KickVector *k*): Returns a vector to teammate i_1 if his position is known and can receive a pass with at least $(i_2 + 1)/10$ probability, or returns *k* instead otherwise.
- kick-goal(Integer *i*, KickVector *k*): Returns a vector to the goal if the shot will be successful with at least $(i + 1)/10$ probability, or returns the vector *k* instead otherwise.
- dribble(Integer *i*, KickVector *k*): Returns the vector $k*(max*i/20)$.
- kick-goal!(): Returns a vector to the opponent's goal.
- far-mate!(): Returns a vector to the most offensively positioned teammate.
- kick-clear(): Returns a vector similar to the one calculated by away-opps, but ensures that the direction is at least 135 degrees from the player's goal.
- kick-if(Boolean *b*, KickVector k_1, KickVector k_2): Returns the vector k_1 if *b* is true, or the vector k_2 otherwise.

Initially, most of the teams did not move around or kick the ball. However, some randomly generated individuals that contained the (ball) and (kick-goal!) primitives were able to score easily and soon spread over the population. This resulted in "kiddie-soccer", where all players of both teams move towards the ball and kick towards the goal. However, this behavior disappeared after several generations when some players assumed more defensive positions. Eventually, players would spread out over the field and give passes to each other.

A second attempt to create a Robocup team using genetic programming was made one year later by Andre and Teller [2]. In contrast to the primitives used by Luke, Andre and Teller developed programs using the primitives specified in the protocol of the soccer simulator. To compensate for the higher level of complexity of using this primitive set, the fitness function was also made more complex by assigning a score for various actions of increasing complexity. These actions are, in increasing order of complexity:

- Getting near the ball.
- Kicking the ball.
- Regularly being on the same side of the field where the ball is.
- Being alive (executing at least one turn and one move action).

- Scoring a goal.
- Winning a game.

Inputs:	Player.X.pos()	
	Player.Y.Pos()	
	Dist.To.Ball()	
	Dir.To.Ball()	
	Dist.To.Goal()	
	Dir.To.Goal()	
	Ball.Dist.Delta()	
	Ball.Dir.Delta()	
	T-Dist(Real n)	(Distance to n'th closest teammate)
	T-Dir(Real n)	(Direction to n'th closest teammate)
	O-Dist(Real n)	(Distance to n'th closest opponent)
	O-Dir(Real n)	(Direction to n'th closest opponent)
Constants:	Real valued constants	
Memory:	Read(Real)	
	Write(Real, Real)	
Calculations:	Add(Real, Real)	
	Sub(Real, Real)	
	Mult(Real, Real)	
	Div(Real, Real)	
	Sin(Real)	
	Cos(Real)	
	IFLTE(Real a, Real b, Real c, Real d)	(if a < b return c else d)
Actions:	Kick(Real, Real)	
	Turn(Real)	
	Dash(Real)	
	Grab()	(only usable by goalie)
Team-shared:	ADF1, ADF2, ADF3, ADF4, ADF5, ADF6, ADF7, ADF8	

Table 8.2: List of primitives used by Andre.

Additionally, teams had to pass three tests before they were allowed to enter a competition:

- The team must be able to score on an empty field within 30 seconds.
- The team must be able to win against a fixed team of stationary opponents that kick the ball when it is within reach.
- The team must be able to win against the winning team of the previous year's competition.

The function set of the programs also contained a fixed number of automatically defined functions (ADFs). Initially, these ADFs contained code for simple hard-coded soccer actions. The teams were also non-homogeneous, but all players used the same set of ADFs within the same team. Additionally, the function set contained instructions to use an indexed memory of 10 memory cells. All the primitive functions return a real value, and are shown in table 8.2. The action that is executed by the player is the last action (kick, turn, dash, grab) executed during evaluation of the program. The grab command can only be executed by the goalie of the team. Unfortunately, no results were shown for the quality of the evolved teams.

Defender()	Vector to opponent
Mate1()	Vector to first teammate
Mate2()	Vector to second teammate
Ball()	Vector to the ball
Rotate90(Vector)	Rotate vector 90 degrees counter-clockwise
Random(Vector)	Random vector with magnitude between 0 and current value
Negate(Vector)	Reverse vector direction
Div2(Vector)	Divide vector magnitude by 2
Mult2(Vector)	Multiply vector magnitude by 2
VAdd(Vector, Vector)	Add two vectors
VSub(Vector, Vector)	Subtract two vectors
IFLTE(Vector a, Vector b, Vector c, Vector d)	If ||a|| < ||b|| return c else return d

Table 8.3: List of primitives used by Hsu and Gustafson.

Hsu and Gustafson [35] use genetic programming to learn a subtask of the Robocup domain called keep-away soccer. In keep-away soccer, the objective is that a team of three players remain in possession of the ball while one opponent attempts to steal the ball. Layered learning GP is used to simplify the learning task. Initially, a simpler task with a more primitive fitness criterion is used to create a population of solutions. These solutions are then used as the initial population for the more complex task. For the task of keep-away soccer, initially the players must remain in possession of the ball while giving passes without an opponent. This first layer is intended to train a passing behavior. In the second layer of the learning task the agents must also learn to keep away from the opponent.

The team of players used in this experiment are homogeneous players that do not use communication. The set of primitives used for the task is similar to those used by Luke [56][60] and Andre [2] and is shown in table 8.3. All the primitives return a vector. The players contain a move-tree and a kick-tree, similar to the teams developed by Luke.

2.2 Description of problems and primitive sets

In [58], Luke discusses some of the problems encountered while training his Robocup team:

- Because the evaluation time of a population takes a long time, the population size was kept very small.
- The results of a match between two teams contains a lot of randomness. Consequently, the result of a single match may give an inaccurate fitness value to the evaluated teams.
- Teams are evolved instead of individual players. The use of heterogeneous groups of players can result in the development of positional players, but the time needed to evolve heterogeneous teams was too long to be successful.
- The function set was biased with primitive functions that implement human soccer behaviors. Also, players do not maintain an internal state.

The Robocup players that are implemented by us are based on the implementation by Luke et al. To deal with the problems described above, the following modifications were made:

- The technique described in chapter 7 to remove individuals that do not contribute to the diversity of the population is used. As a result, it is possible to use a much smaller population size without losing diversity in the population.
- The teams in a population are evaluated using a Swiss tournament format [115]. In this format, all teams play an equal number of rounds. In each round, a team is paired against another team with a similar performance in the previous rounds. In contrast to a single elimination tournament format,

this gives a more detailed measure of the quality of all the teams in a population instead of just the top teams.

- Primitive functions are added that allow a player to investigate the position of that player such as goalie or attacker. This allows the evolution of heterogeneous players that still use identical programs.
- Some actions require a number of primitive actions that are executed sequentially. In this case, the primitive actions for subsequent time steps can be stored in a queue and will be executed automatically in the next time step, without the need to re-evaluate the program tree.

The set of primitive functions used is mostly identical to the one presented in table 8.1, with some small changes. More recent versions of the soccer simulator support commands to turn the neck of a player. As a result, the player can look in a direction not directly in front of him. This makes it possible to keep looking at the ball while moving in another direction. The commands to follow the ball can be hard-coded in the players, making the evolutionary primitives to track the ball redundant. Consequently, the primitives *findball* and *sight* can be removed from the primitive set. On the other hand, primitives to investigate the position of a player have been added to the primitive set. Also, primitive functions to reference a player are added. Primitives that used to reference a player with an integer number are modified. The added and modified primitives are shown in table 8.4:

Base	Boolean	MyType(PlayerType)
	PlayerType	{goalie, defender, midfield, forward}

	MoveVector	home-of(PlayerBase)
		mate(PlayerBase, MoveVector)

	KickVector	mate-m(PlayerBase, Integer, KickVector)

	PlayerBase	NearestMate()
		NearestMateType(PlayerType, PlayerBase)
		NearestMateDist(Integer, PlayerBase)

	SoccerPlayer(MoveVector, KickVector)	

Table 8.4: Modified and added primitives of the hierarchy of STGP primitives.

The primitives home-of, mate and mate-m are modified to take an argument of type PlayerType. When a player is started, a PlayerType is assigned to the player based on its number. Player 1 is a goalie, players 2, 3, 4 and 5 are defenders, players 6, 7, 8 and 9 are midfields, and players 10 and 11 are forwards. The primitive MyType can be used to test if a player has a given PlayerType.

NearestMate returns a reference to the nearest observed teammate. NearestMateType returns a reference to the nearest observed teammate of a given type, or returns the second argument if no teammate can be observed instead. NearestMateDist returns a reference to the teammate closest to the specified distance, or the second argument if no teammate can be observed instead. The Move-tree and Kick-tree of a soccer player are stored in the primitive SoccerPlayer.

The population size in all the experiments was 32: every generation, 32 new individuals are added to the population. If the population size exceeds 64 at the end of the evaluation process, only the best 64 individuals are retained in the population and the rest is removed. This only occurs if the method to remove individuals from the population based on the diversity is not used. Otherwise, the population size is already significantly smaller than 64.

2.3 Experimental results

Several experiments using different settings were performed. Due to the amount of time necessary to evaluate a population (several hours), it was impossible to perform sufficient different runs to calculate meaningful averages. However, it was still possible to observe the effects on code growth and specialization in these individual runs. The following experiments were performed:

- One run of 131 generations using dynamic size limiting and the diversity measure.
- One run of 121 generations using the diversity measure but without dynamic size limiting.
- One run of 119 generations using dynamic size limiting and the diversity measure. This run uses the same setup as the first run.
- One run of 71 generations using dynamic size limiting, but without the diversity measure.

Because the population size was not reduced based on diversity in the last experiment, more evaluations were needed to evaluate the entire population. This resulted in a significant increase of the evaluation time, and as a result the experiment was stopped after 71 generations. This indicates that the diversity measure is able to significantly reduce evaluation time of a generation.

2.3.1 Evolution of fitness

The evolution of the fitness of the best individual of every generation can be observed by evaluating a population that consists of these best individuals. The evaluation used a Swiss tournament consisting of 8 rounds. If an individual has the highest fitness for several generations, it participated only once in the evaluation tournament. The results are represented in a graph where the horizontal axis represents the best individual of the corresponding generation, and the vertical axis represents the rank of this individual after the evaluation.

The results of experiment 1 are shown in figure 8.9. Because of the randomness of the evaluation, these results contain a lot of noise. However, the teams of the first 30 generations are mostly unable to win from the teams from the last 70 generations. After generation 60, the performance of the teams does not appear to improve much, as indicated by the average over 5 generations.

The results of the second experiment are shown in figure 8.10. Again, the results of the experiments contain a lot of noise. Only during the first 10 generations is the performance of the teams considerably lower, but the average score over 5 generations rises steadily over several generations.

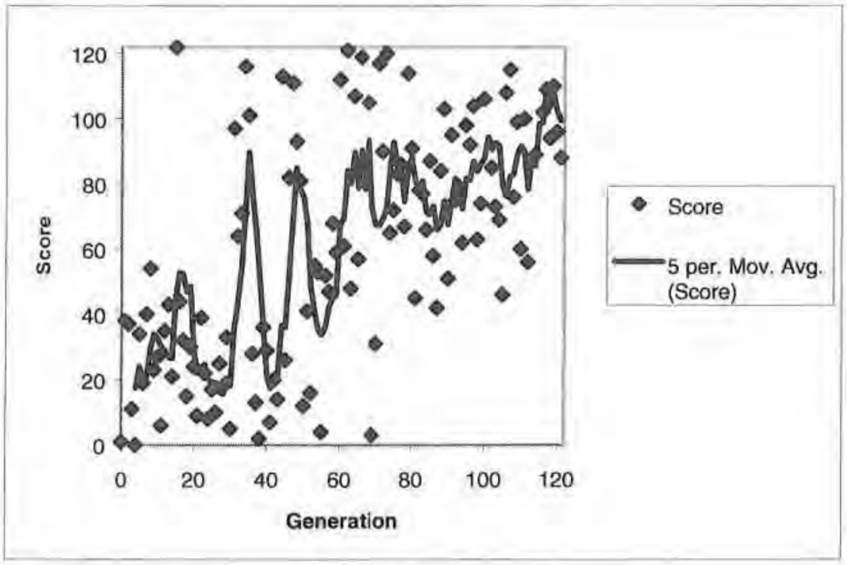


Figure 8.9: Evolution of fitness in experiment 1.

Figure 8.11 shows the results of experiment 3. During the first 15 generations, the average fitness of the evolved teams increases rapidly, but stabilizes afterwards.

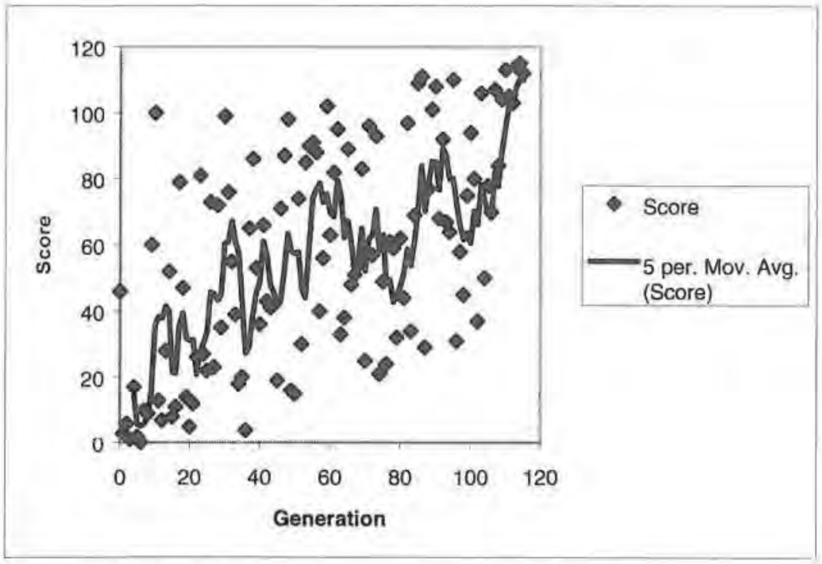


Figure 8.10: Evolution of fitness in experiment 2.

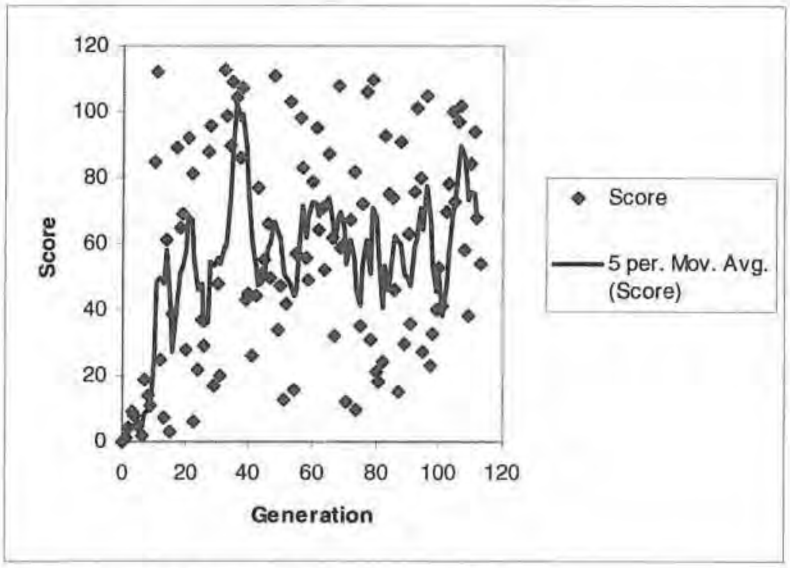


Figure 8.11: Evolution of fitness in experiment 3.

Figure 8.12 shows the evolution of fitness in the fourth experiment. The evaluation time of a single generation is much longer in this experiment, because the population size was not reduced. As a result, only 71 generations were evolved. In the first 30 generations, the fitness gradually increases. At that point, there is a small decrease in performance for about 15 generations. Finally, the average fitness increases again until the end of the run. It is however not sure if the decrease in performance is caused by the randomness of the evaluation or if the generated programs are effectively performing less good.

The results of these experiments are consistent with the results reported by Luke [60][61]. The initial populations consistently perform very bad because most teams remain stationary, while only a few individuals move towards the ball. After several generations, most teams are playing "kiddie-soccer", where all players run towards the ball and kick towards the opposing team's goal. Finally, the behavior of teams changes towards a passing game where several players remain at several positions on the field or interfere with the movement of opposing players.

The huge amount of noise in the results can be explained partially by the randomness of the environment, but the less than optimal implementation of the primitive skills may also be responsible.

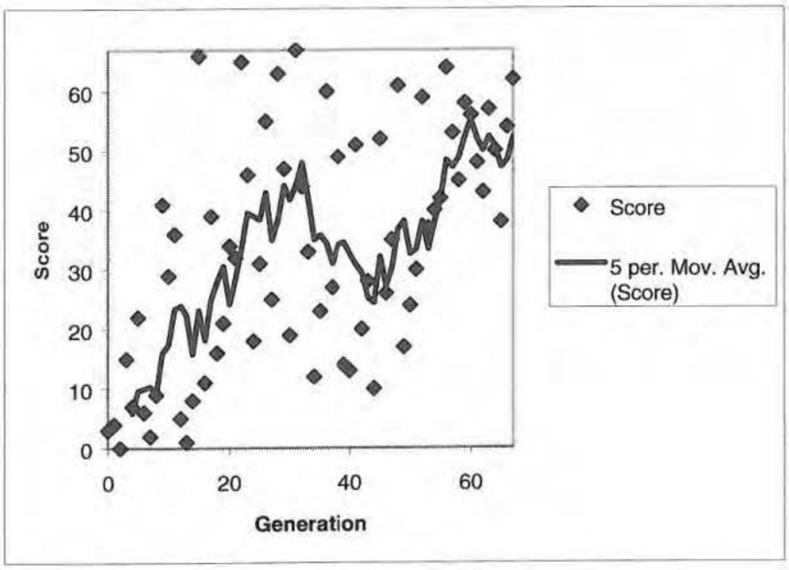


Figure 8.12: Evolution of fitness in experiment 4.

2.3.2 Evolution of code growth

In [61], Luke reported that the size of the individuals grows rapidly and becomes almost unmanageable after about 40 generations. As a result, reducing the effects of bloat is an important factor in the experiments since they lasted for more than 100 generations. This section shows the average and maximum sizes of the new individuals created during every generation over the course of all experiments.

Figure 8.13, figure 8.15 and figure 8.16 show the result of applying dynamic size limiting and the results are very similar. The average size of the individuals stabilizes around 17 in the first experiment and 20 in the third experiment. In the fourth experiment, the average size increases slowly to 20. During most of the runs, the maximum size of new individuals is set to 32, a fixed minimum size of all individuals. However, only very few individuals were created (and rejected) that exceeded this limit. Removing these few individuals is sufficient to keep the size of individuals small after a large number of generations.

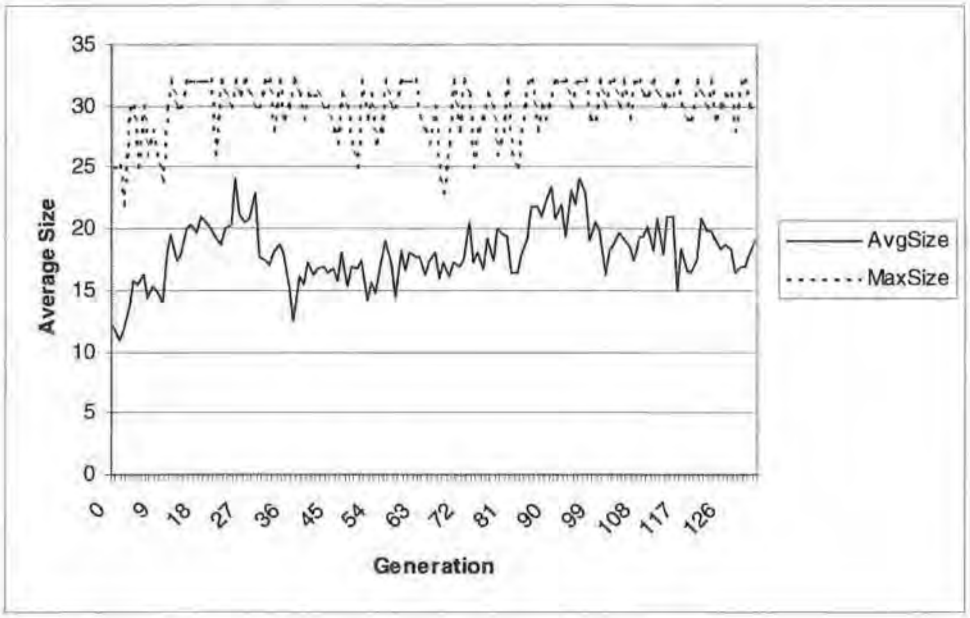


Figure 8.13: Evolution of average size and maximum size of the individuals of a generation in experiment 1.

In experiment 2, when the size of new individuals was not restricted, the results of figure 8.14 were obtained. In this run, the average size grows towards 60 after 100 generations, but decreases again later towards 35. However, many very large individuals are present. Inspection of the individuals of this experiment shows that the increase in size is mostly caused by long chains of similar instructions that have no or little effect when chained together. An example of this is given in the following individual, created at generation 103. This individual is an example of defense against crossover. Exchanging code between two chains of similar code does not change the behavior of the individuals.

```
(SoccerPlayer (WeightI (Int 4) (WeightI (Int 4) (Mate
(NearestMate) (MoveVectorTerminal Ball))
(MoveVectorTerminal Ball)) (MoveVectorTerminal Ball))
(PassFarMate (Int 4) (PassFarMate (Int 2) (PassFarMate
(Int 4) (PassFarMate (Int 2) (PassFarMate (Int 5)
(PassFarMate (Int 5) (PassFarMate (Int 5) (PassFarMate
(Int 4) (PassFarMate (Int 7) (PassFarMate (Int 5)
(PassFarMate (Int 4) (PassFarMate (Int 2) (PassFarMate
(Int 7) (PassFarMate (Int 5) (PassFarMate (Int 2)
(PassFarMate (Int 5) (PassFarMate (Int 5) (PassFarMate
(Int 5) (PassFarMate (Int 2) (PassFarMate (Int 5)
(PassFarMate (Int 5) (PassFarMate (Int 4) (PassFarMate
(Int 7) (PassFarMate (Int 5) (PassFarMate (Int 5)
(PassFarMate (Int 5) (PassFarMate (Int 4) (PassFarMate
(Int 4) (PassFarMate (Int 4) (PassMate (NearestMateType
(PlayerType defender) (NearestMate)) (Int 7) (PassFarMate
(Int 9) (KickVectorTerminal
KickGoal)))))))))))))))))))))))))))))))
```

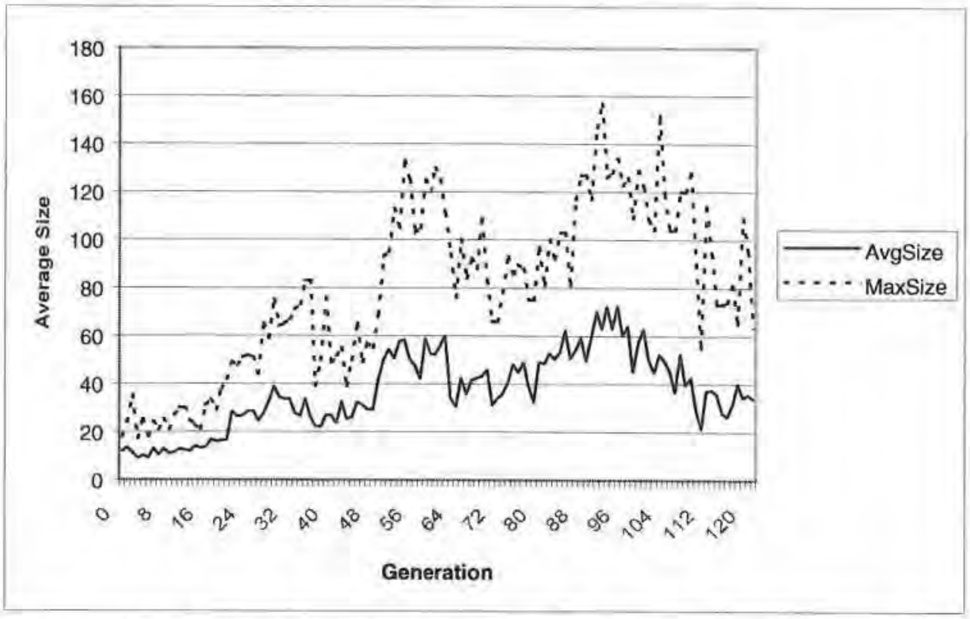


Figure 8.14: Evolution of average size and maximum size of the individuals of a generation in experiment 2.

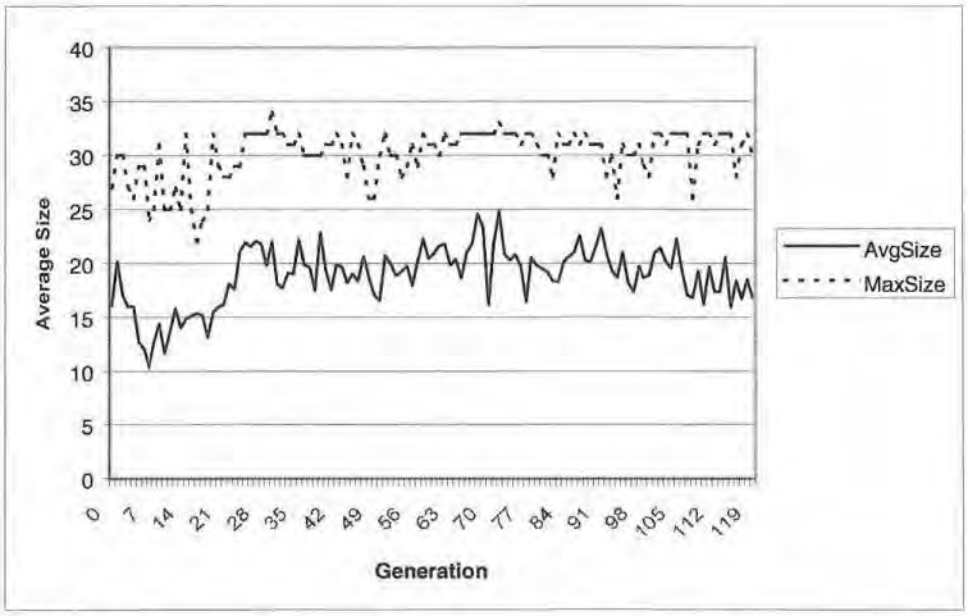


Figure 8.15: Evolution of average size and maximum size of the individuals of a generation in experiment 3.

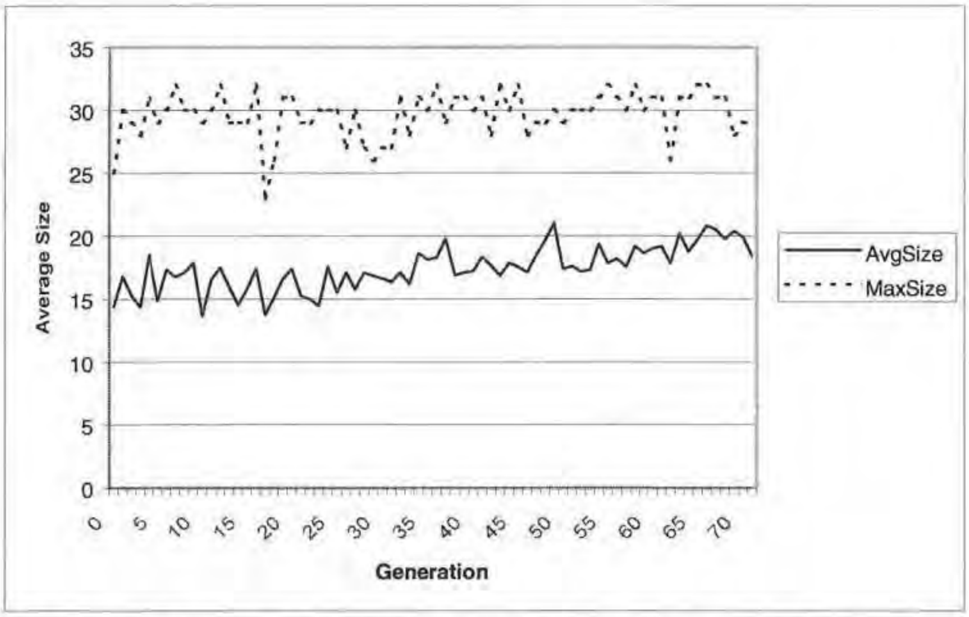


Figure 8.16: Evolution of average size and maximum size of the individuals of a generation in experiment 4.

2.3.3 Evolution of specialization

In [60], Luke discovered that teams of homogeneous agents learn to play an acceptable level of soccer before their heterogeneous counterparts. However, the players of a homogeneous team are unable to evolve to specialized position players

The teams evolved using the set of primitive functions listed in table 8.4 are essentially homogeneous. However, to introduce the possibility of evolving heterogeneous players, the primitive (MyType <Type>) was added to the primitive set. When used in combination with an if-statement, the same program can demonstrate different behaviors depending on the position of the player. This section will examine how many programs made use of the MyType primitive, and how long this primitive survives in the population. This can give a measure of the specialization of teams and their success in the evolution.

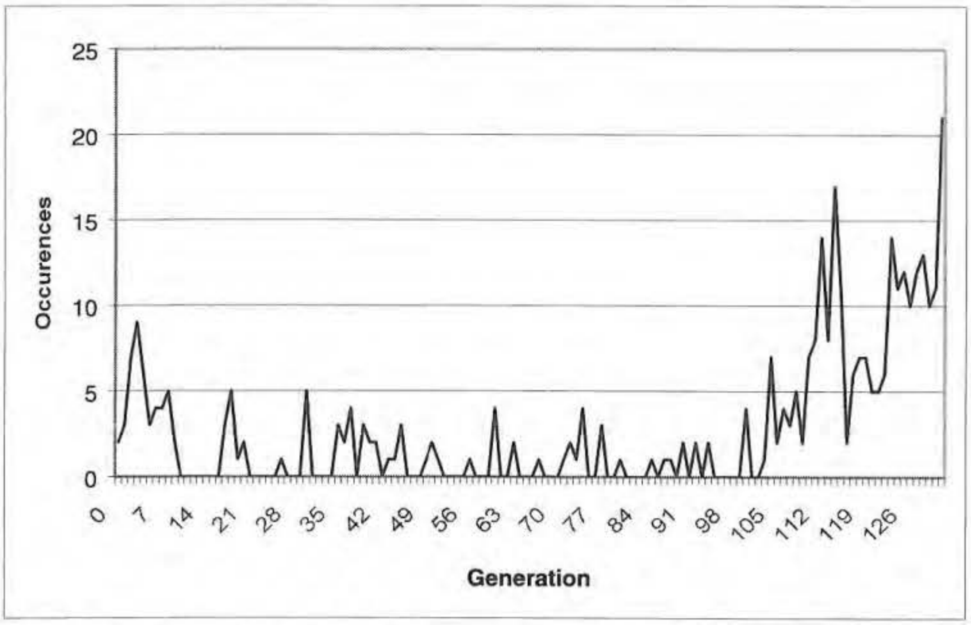


Figure 8.17: Evolution of the number of individuals containing the MyType primitive created at every generation in experiment 1.

Figure 8.17 shows the number of individuals created at every generation that contain at least one instance of the MyType primitive in experiment 1. Of the 4222 individuals created in the entire run, 368 use MyType. In the first generations of the run, when a lot of individuals are mostly random, several instances of MyType can be found. When the behavior of the individuals evolves to kiddy-soccer, specialization of players becomes an evolutionary disadvantage, because the number of teammates surrounding the ball increases the probability that a teammate can kick the ball. If some specialized players do not play kiddy-soccer, the team will perform worse and specialization disappears from the population. At this stage, new instances of MyType can only be introduced through random mutations in individuals, and often don't improve the behavior of teams, even when teams learn to pass and distribute themselves over the field. Around generation 105, a more successful application of MyType is discovered, and this application is transmitted to new individuals through crossover. An example of an individual created during generation 130 is shown below. The successful part of code that is transmitted during several generations is highlighted. This demonstrates that in this run, a position for a defender was eventually evolved.

```
(SoccerPlayer
  (If<MoveVector> (MyType (PlayerType defender))
    (MoveVectorTerminal Ball)
    (WeightI
      (Int 7)
      (HomeOf
        (NearestMateDist
          (Int 3)
          (NearestMate)))
      (MoveVectorTerminal Ball)))
  (KickVectorTerminal KickGoal))
```

Figure 8.18 displays the number of individuals containing MyType created in the second experiment. From the 4095 individuals created during the run, 907 contained at least one instance of MyType. In this run, there appear to be several periods where individuals containing MyType are reasonably successful for several generations and are able to reproduce, but die out later. Inspection of the code of these individuals reveals that the MyType primitive is not responsible for the success of these individuals. Instead, MyType appears mostly as inactive code or in parts of code that have only limited influence on the behavior of players. For example, the individual below, created at generation 25, contains specialized code shown in boldface. However, because of the two WeightI primitives preceding this code, less then 10% of the calculated vector is created by the specialized code.

```
(SoccerPlayer
  (WeightI
    (Int 4)
    (WeightI
      (Int 2)
      (BlockNearOpp
        (If<MoveVector> (MyType (PlayerType forward))
          (WeightI
                (Int 8)
                (BlockNearOpp
                  (HomeOf (NearestMate)))
                (If<MoveVector> (BoolTerminal OpponentCloser)
                  (Mate
                    (NearestMate)
                    (MoveVectorTerminal BlockGoal))
                  (Mate
                    (NearestMate)
                    (MoveVectorTerminal AwayOpps))))
              (MoveVectorTerminal Ball)))
          (MoveVectorTerminal Ball))
        (MoveVectorTerminal Ball))
      (KickVectorTerminal KickGoal))
```

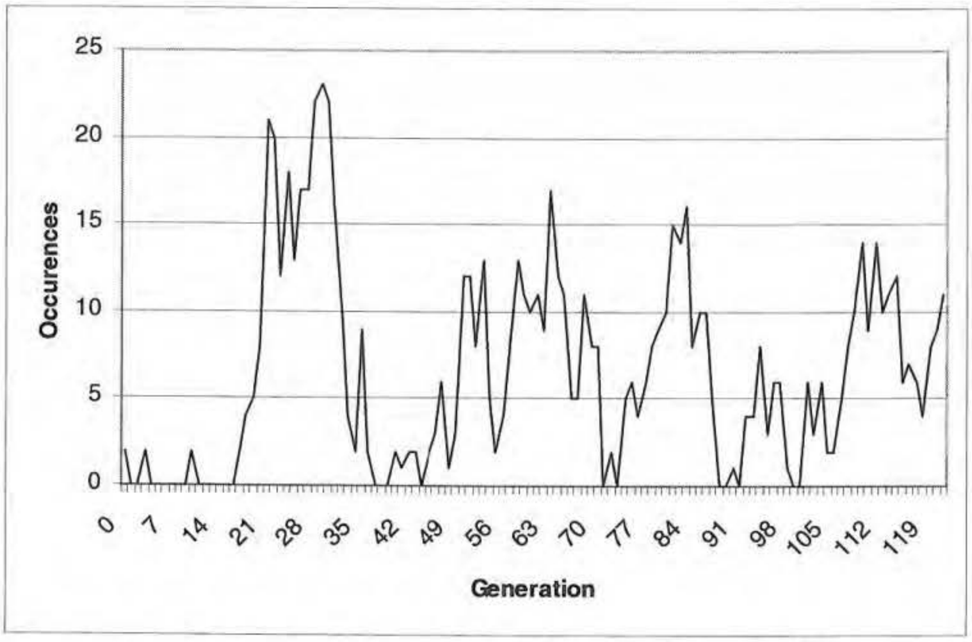


Figure 8.18: Evolution of the number of individuals containing the MyType primitive created at every generation in experiment 2.

The results of specialization in the third experiment are shown in figure 8.19. Only 78 of the 4013 created individuals in this experiment contained a MyType primitive. None of these individuals was able to provide an evolutionary advantage, resulting in the extinction of the MyType primitive. As a result, no specialization was observed in this run.

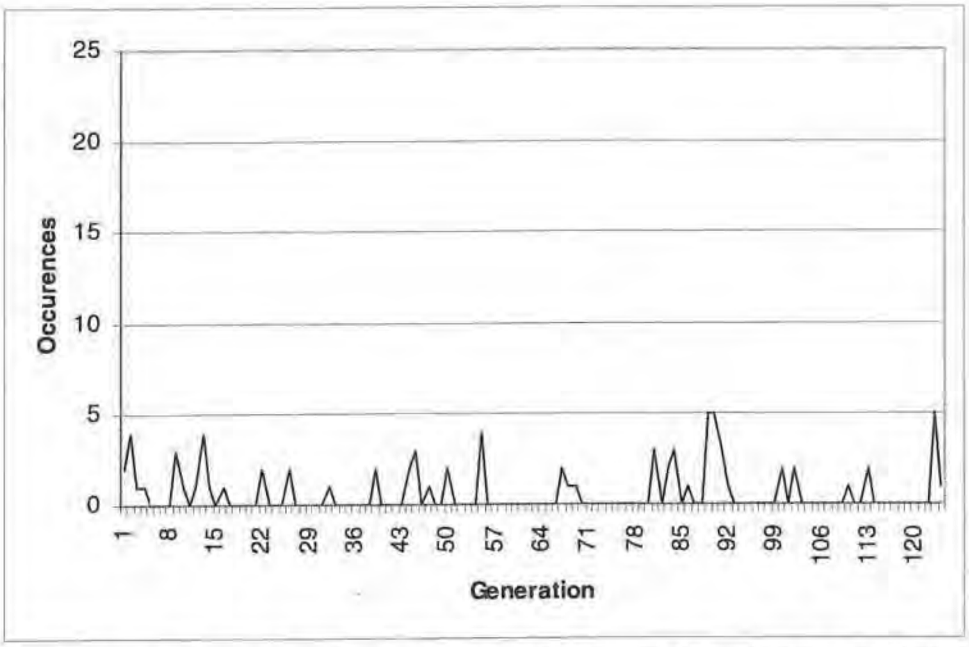


Figure 8.19: Evolution of the number of individuals containing the MyType primitive created at every generation in experiment 3.

In the fourth experiment, the following individual appeared in the initial random population:

```
(SoccerPlayer
  (BlockNearOpp
    (WeightI (Int 1)
      (WeightI (Int 0)
        (BlockNearOpp (MoveVectorTerminal AwayMates))
        (MoveVectorTerminal Home))
      (BlockNearOpp
        (Mate
          (NearestMateType
            (PlayerType goalie)
            (NearestMate))
          (MoveVectorTerminal Ball)))))
  (PassFarMate
    (Int 0)
    (If<KickVector> (MyType (PlayerType forward))
      (KickVectorTerminal KickGoal)
      (PassFarMate
        (Int 0)
        (KickVectorTerminal KickClear)))))
```

The section in boldface indicates code that specializes for an attacker. This individual proved to be very successful, and was only removed from the population after generation 13. The highlighted code above was replicated several times during this time, sometimes with small variations. In the first 20 generations, the MyType primitive also occurs frequently in the Move-branch of the code and is responsible for most of the MyType occurrences in figure 8.20. However, the attacker specialization becomes more frequent later, and most of the occurrences in the last generations are copies of this code. The only modification is that the KickClear primitive is replaced by the KickGoal primitive, and sometimes (PlayerType forward) is replaced by other player types. Strangely enough, the change to (PlayerType goalie) occurs more frequently then the other possible changes.

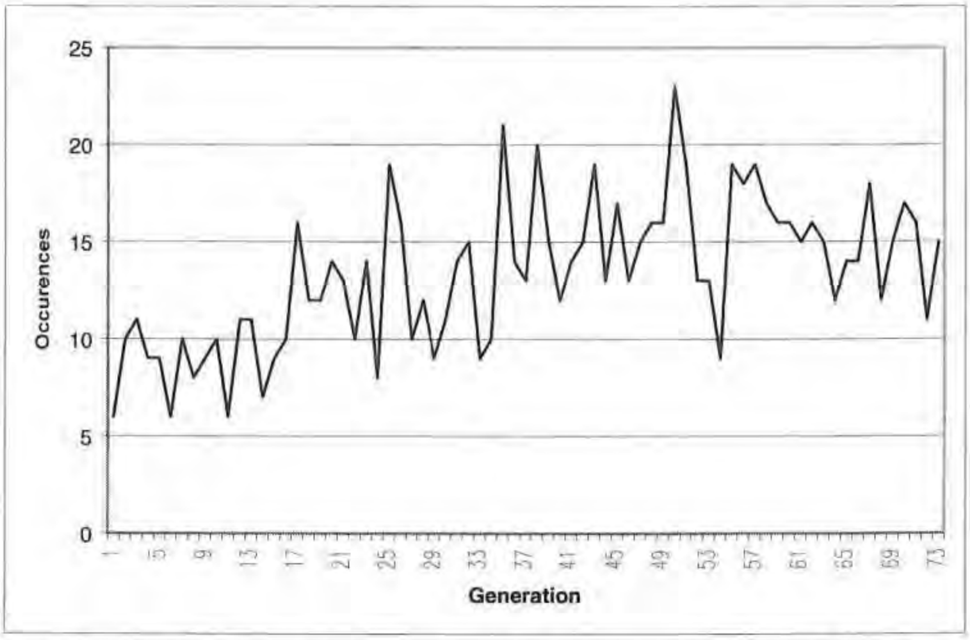


Figure 8.20: Evolution of the number of individuals containing the MyType primitive created at every generation in experiment 4.

Other types of specialization are rare. In generation 50, the following specialization for a midfield was created:

```
(SoccerPlayer
  (MoveVectorTerminal AwayMates)
  (If<KickVector> (MyType (PlayerType midfield))
    (Dribble (Int 0)
      (If<KickVector> (BoolTerminal Squad1)
          (KickVectorTerminal KickClear)
          (KickVectorTerminal KickMate)))
      (PassFarMate
        (Int 7)
        (KickVectorTerminal KickGoal))))
```

Unfortunately, this individual was unable to survive in the next generation. At that time, 46 of the 64 surviving individuals contained the specialized attacker code. This demonstrates that this piece of code dominates the population, and new original code has little chance to survive the selection process. As a result, the diversity of the population has decreased, and finding better solutions becomes very difficult.

2.4 Comparison of the experiments

To be able to compare the results of the different runs of the previous section, the best teams of every generation of all these runs are compared in a single tournament. In this tournament, 420 teams participated and played 10 rounds. The results are shown in figure 8.21 and give a relative comparison of the different runs. Again, the results contain a lot of randomness, so an average over 8 samples was added to observe any trends in the data.

The first run appears to be the most successful in this comparison. During the first 40 generations, the rank of individuals is somewhat lower, but towards the end of the run the fitness of the individuals improves. The third run also has a high rank overall. The rank of the initial generations of this run are relatively good, but hardly any improvements are made after generation 40. This explains the slow growth of figure 8.11. In the first half, the results of run 2 are comparable with run 1 and 3. At generation 60 however, a sudden drop in performance occurs and the rank slowly increases again. The results of run 4 are somewhat below the average of the other runs. After generation 30, the fitness of the population shows little improvement. Given that the diversity of the population has decreased significantly at generation 50, the chance for large improvements decreases significantly in further generations.

Of the four runs compared, the first run that uses both the size reduction and diversity measure, has the best performance. However, because of the limited number of runs, it is impossible to generalize this result. On the other hand, it is still possible to make some more general conclusions. First, the use of the diversity measure is able to significantly reduce the size of a population. As a result, the evaluation time of a population is significantly reduced. This was observed in the difference in evaluation time between the first 3 runs and the fourth run. In the last run, the evaluation time was almost twice as long, and as a result only 71 generations were evolved. Also, the average size of individuals did not grow significantly when the dynamic maximum size was imposed on new individuals in runs 1, 3 and 4. Therefore, these optimizations appear to have a positive effect on evolution in the Robocup domain.

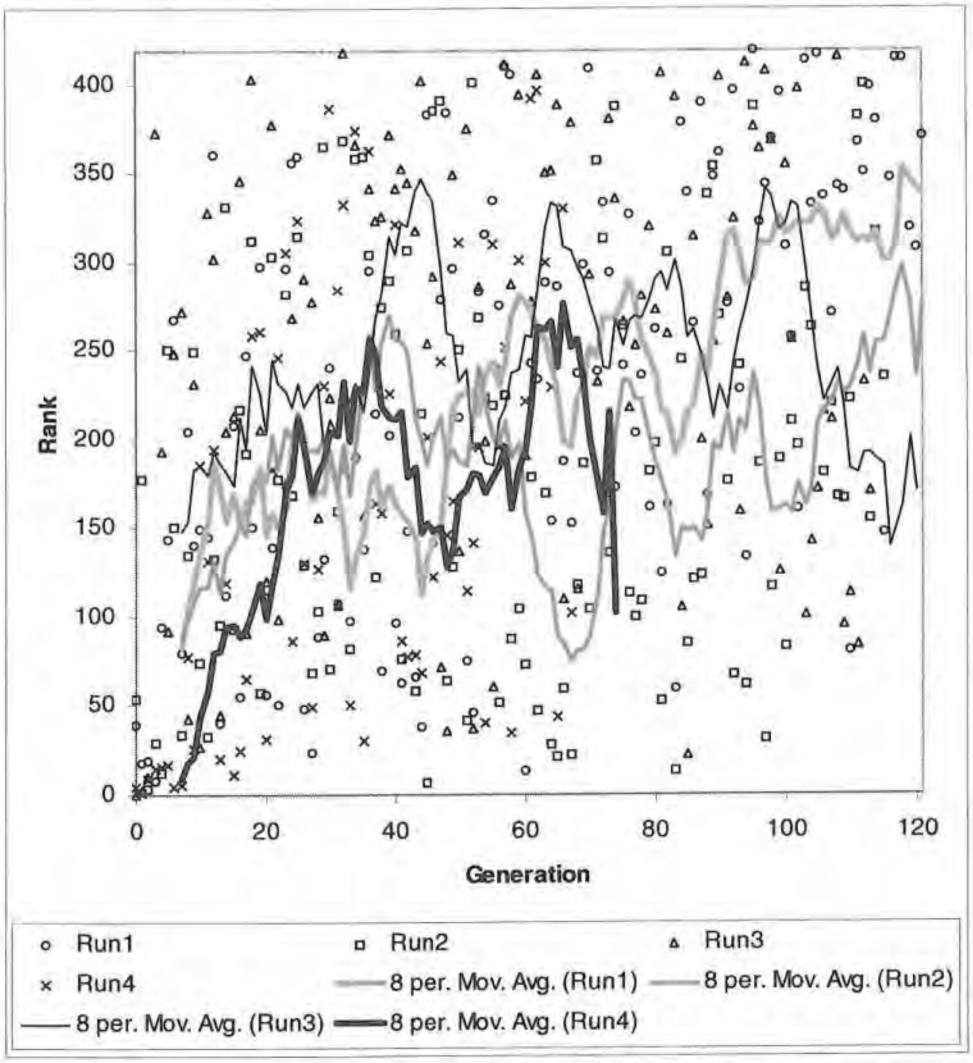


Figure 8.21: Comparison of the evolution of fitness of the different runs.

The most important problem encountered was the huge randomness of the results when comparing a population of evolved teams. Other than the randomness present in the Robocup simulator, this may be explained by the less than optimal implementation of the Robocup primitive set. Significant improvements can be made to this implementation, which may lead to a more consistent performance of evolved teams.

2.5 Conclusion

In this section, genetic programming was used to create programs to control players for the Robocup simulation based on the terminal set used by Luke [60]. Solutions were introduced for most of the problems encountered by Luke [58]:

- It was possible to reduce the population size when using the diversity measure described in chapter 7. Due to the reduced population size, the evaluation time of a generation is reduced significantly, while still avoiding premature convergence. This demonstrates the effectiveness of this technique for complex problems.
- The fitness measure was made more accurate by using a Swiss style tournament for the fitness evaluation. Unfortunately, the results of the fitness evaluation were still very noisy. However, these results are most likely still more consistent compared to the single round tournament used by Luke.
- The average size of individuals in the population was reduced significantly by using the dynamic size limiting technique described in chapter 6.
- The homogeneous team of players evolved by genetic programming can exhibit heterogeneous behavior. This is accomplished by assigning the players a role based on their number, and the addition of primitives to query this role. In some experiments, specialized players were developed.

Chapter 9: Conclusions and future work

1 Conclusions

The work performed in this thesis concerning virtual agents can be divided in two parts. In the first part, the problem of navigation and obstacle avoidance was considered. To solve this problem, the virtual agent uses a virtual visual sensor that measures the depth value of the observed objects in front of the agent. The agent then uses this depth information to construct a map of the area surrounding the agent by rotating. To construct a map of the entire environment, the agent moves to openings in the area and repeats the process, until the entire environment has been visited. The constructed map can then be used for path planning in the environment, and can be shared with other agents or users in the environment. The depth information about obstacles around the agent is also used to perform collision avoidance while moving in the environment. If movement in a direction would result in a collision with the detected object, the movement command is modified to avoid a collision. This can be done either by moving around the obstacle or by stopping the agent.

There are several advantages to the use of a virtual sensor to detect the environment instead of using the internal representation of the environment. First, using a virtual sensor is a more realistic simulation of the real world, as mobile robots often also use depth sensors for navigation. A second advantage is that it is not necessary for the virtual agent to have access to the internal representation of the environment. When only a rendered image of the environment is required, the virtual agent can work in every environment that supports the rendering of this image.

The second part of the thesis deals with training virtual agents to perform a task. The use of genetic programming in this context was examined. Two main problems were identified when evolving virtual agents using genetic programming. First, the size of the evolved genetic programs tends to grow rapidly very soon. Secondly, because evaluation of evolved virtual agents usually takes a long time, the number of evaluations must be reduced. Unfortunately, genetic programming usually requires a large population of candidate solutions to discover good results.

Several solutions were developed to remove these problems. First, two new methods to reduce the size of evolved programs were developed, and compared with existing methods. The first method detects and removes inactive code from the genetic programs. The second method imposes a dynamically changing maximum size on newly created individuals, depending on the size of the current best individual of the population. The second of these new methods was found to be superior than the existing methods, and both methods can even be used together.

The second problem was solved by constructing an algorithm that reduces the number of individuals in the population, without reducing the genetic diversity of the population. The algorithm works by detecting identical subtrees in individuals of the population, and removing those individuals mainly consisting of subtrees that are also present elsewhere in the population. Removing these individuals from the population was shown to improve convergence speed of the evolution.

Finally, these improvements to the genetic programming algorithm were applied to the virtual multi-agent system of robotic soccer. The improvements appear to have a positive effect on the evolution, but due to the still long evolution time in this domain, it was not possible to perform a sufficient number of different runs to draw general conclusions.

2 Future work

2.1 Improvements to map construction

A future goal of our research is to speed up map construction by having several agents explore the world simultaneously, and exchanging pieces of the map with each other when they encounter each other. Another possible extension is to construct three-dimensional maps, allowing the agent to move in three dimensions. This would complicate the algorithm, but the same principles could be used in three dimensions. Also, the agent currently assumes the world is entirely static. It would be interesting to add non-permanent obstacles (like doors, or moving object). The agent can then detect changes between the environment and its internal representation of the environment, and update the representation dynamically.

2.2 Improved individual player skills for Robocup

The current implementation of the individual player skills used for genetic programming is far from perfect. These individual skills can be trained using genetic programming. In this case, the primitive set of instructions includes the primitive actions of the Robocup server protocol. Training can be done by setting up a number of trials, setting up initial positions of the ball and players. After each experiment, a fitness value can be calculated, for example by calculating the distance from the ball to a specified target position. It would then be interesting to investigate if improved individual player skills can reduce the noise encountered in the evolution of action selection in Robocup.

2.3 Explicit credit assignment and directed crossover

Genetic programming assigns a fitness value to an individual after it has been evaluated, representing the quality of that individual. However, this fitness value provides no information about which parts of the individual are responsible for the quality (or lack thereof) of the result. In [47] and [48], Langdon describes a technique where a subtree with a low performance has a higher chance to be removed by the crossover operator. This technique is called directed crossover.

Using the influence values of child nodes after a single evaluation, it is possible to distribute the error value of the evaluation over the different nodes of the individual. This gives an explicit credit score to the nodes of an individual for that evaluation. Starting at the root node, the error value can be assigned to the root node and then divided over its child nodes proportional to their influence value. This process can then be applied recursively to the child nodes. The error values assigned to the nodes are accumulated over all the single evaluations.

When all evaluations have been completed, the total error value of all evaluations is again distributed over the nodes of the individual, using the total influence values of these nodes. These total error values can then be compared with the accumulated error values. If the accumulated error value is larger than the total error value, this signifies that the node lowers the overall performance of the individual. These nodes can then be given a higher chance to be removed by the crossover operator, increasing the chance that a better individual will be created.

2.4 Determining a maximum size limit

The dynamic size limiting technique discussed uses the size of the current best individual in the population to determine the maximum size of new individuals.

It may be better if more individuals of the population determine this maximum size. For example, the average size of the best 10% of the population can be used, or a weighted average of all individuals of the population, where the weight depends on the rank of the individual. It would be interesting to study the effects of these changes on code growth and convergence speed.

2.5 Improving the diversity measure

Equation (7.6), used to determine whether an individual is sufficiently different from the rest of the population, is a very simple diversity measure. Even though it appears to be a good and simple heuristic, it would be interesting to study whether a more complex measure produces even better results.

Also, when a large number of terminal elements exists (such as the set of all real numbers), it is possible that most of a population's subtrees are different from each other, and the diversity measure will not remove any elements. If the directed acyclic graph representation, used to detect similar subtrees, was expanded to be able to detect similar internal parts of individual, more similarity between individuals can be detected. The detection of similar structures can also be interesting in the automatic creation of subroutines. It would be interesting to test whether the added complexity of this approach would be offset by any added performance of the evolutionary search.

Appendix A: Communication between virtual agents

1 Introduction

Agents are constructed to help users and other agents with their work. Virtual agents are no exception to this. For example, when a user needs a map of the environment, he may contact a mapping agent (as described in chapter 3) and receive a map of the environment. If the agent has already explored the environment before the user contacted it, the map is readily available. The only problems for the user are how he can locate an agent that has a map of the environment, and how to communicate with this agent.

As virtual agents have a spatial representation in the environment (an avatar), users can encounter such an agent while exploring. The purpose of an agent can be indicated by the appearance of the agents avatar. The user can then contact the agent, and make use of the agents services.

2 Modes of communication

In networked environments, communication is performed by a network protocol such as TCP or UDP to a network port on the IP address of the receiver. This type of communication requires that the sender knows the IP address and port number of the receiver. Two forms of communication can be distinguished:

2.1 Long range communication

Long range communication takes place between any two agents and/or users in the environment, regardless of their current position in the environment. This form of communication typically takes place using the IP address and port number of both participants of the communication.

2.2 Short range communication

Short range communication simulates two participants that are close to each other and talk to each other. As a result, this form of communication can use or simulate the audio sensors of the environment. This mode of communication is useful for two participants to initiate communication when long range communication is not possible because the IP address/port number of the other participant is unknown. This IP address and port number can then be transmitted using short range communication, and the remainder of the interaction can then take place using long range communication.

Several ways are possible to implement short range communication in the virtual environment:

- If available, the audio channel present in the environment can be used to communicate with the targeted participant. A message is broadcasted to all nearby agents and users. The targeted participant can then respond with his IP address and port number and communication can be resumed using long range communication. A problem that must be solved when using this approach is that the broadcasted message must contain a way to indicate the targeted participant.
- The IP address and port number of the participant can be a part of the avatar of the participant. When the avatar is then observed, the IP address and port number are known and long-range communication can be initiated. This step is an abstraction of using an audio channel to communicate with a nearby participant and request the IP address and port number. This implementation was chosen for our prototype application.

3 Directory agents and protocols

Using short-range communication, an agent can begin communication with any other agent that has been encountered in the environment. In a second step, the agent investigates which tasks the other agent can perform. This is achieved by requesting a list of communication protocols supported by the agent. A basic standard communication protocol to retrieve this information is used, which is supported by the avatar of every agent and user in the environment. This basic protocol supports queries like requesting the names and versions of supported protocols, and the responses to these queries. This basic protocol is the only protocol that is strictly required to be supported by all avatars in the environment.

As finding an agent that supports a specific protocol may take a long time, a virtual agent can be used that maintains a directory of all agents that it encountered, and the protocols they support. Communication with this agent is then performed using a directory protocol. This protocol can request the long-range communication address of another agent with a specific protocol, or the registration of protocols with the directory agent. If an agent is known that supports a requested protocol, the address of this agent will be returned. Otherwise, the directory agent will store the request, and when an agent with the requested protocol is discovered later, the directory agent will notify the agent.

All agents that support the directory protocol are able to make use of its services. Additionally, all these agents can also perform some of the tasks of the directory agent, at least for the protocols they are interested in. These agents can also provide the long-range communication address of the directory agent they use. As a result, when a new virtual agent enters the environment and encounters another virtual agent that supports the directory protocol, the new agent is immediately able to directly contact the directory agent and locate any other agent in the environment.

4 Application prototype

To test the principles described above, a prototype application was constructed that contained virtual agents and users. When the virtual environment is loaded, a directory agent will be started that connects to the environment. The task of this agent is to wander around in the environment and contact every agent or user it encounters to store its supported protocols. However, the agent will need a map of the environment to be able to wander around the entire environment, without missing areas. Because the directory agent doesn't have the mapping algorithm to construct a map, it will create a specialized virtual mapping agent (and add it to its directory). The directory agent then contacts the mapping agent and requests a map of the environment. The mapping agent uses the algorithms described in chapter 3 to construct a map. Whenever a new area in the map is created, the directory agent gets notified that the map is updated, and the directory agent will update its map. When the mapping agent completes its task, it will also wander around in the environment looking for anyone that needs its services.

When a user enters the environment, he will soon encounter an agent in the environment, and be able to communicate with a directory agent. As soon as the directory agent is contacted, the directory agent will query the user's avatar about the protocols that it supports.

When the user needs a map of the environment, he can either click on the avatar of a mapping agent when it is visible, or press the 'm' key to request a mapping agent by the directory agent. This will display a map window, where the user can also request a path to a location by clicking on the location in the map window. The mapping agent will then calculate the shortest path to that location and return the result.

The avatars of users of the virtual environment also support a chatting protocol. When a user clicks on the avatar of another user, the supported protocols of the other user will be queried. If the chat protocol is supported by both users, it will be activated. A text window will appear where a message can be entered and sent to the other user (see Figure A.1). In this figure, the avatar of the other user is represented by a purple cylinder. When the other user receives the message, he can enter a reply message and send the reply.

To detect inactive users or agents in the environment, the basic communication protocol also supports a 'ping' message. Agents can send a 'ping' message to another agent or user, and when a reply is received, the agent or user is still active. If a directory agent detects that an agent or user in its directory is no longer active, it will be removed from the directory.

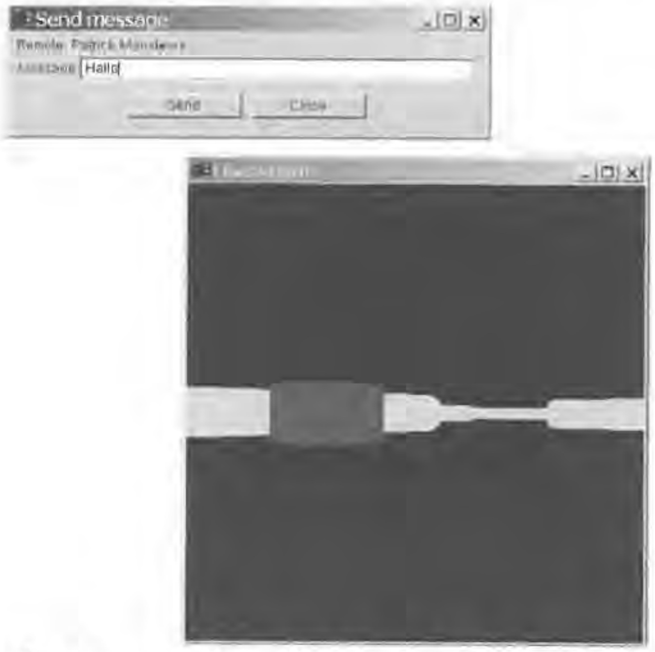


Figure A.1: Sending a message to another user in a virtual environment by clicking on its avatar.

Appendix B: Description of the neural network

1 Problem description

The objective of the neural network is to detect discontinuities in the depth values surrounding the virtual agent, as described in section 5.1 of chapter 3. Four adjacent depth values d_0, d_1, d_2 and d_3 are available as inputs, and a discontinuity between the second and third value must be detected (see Figure B.1).

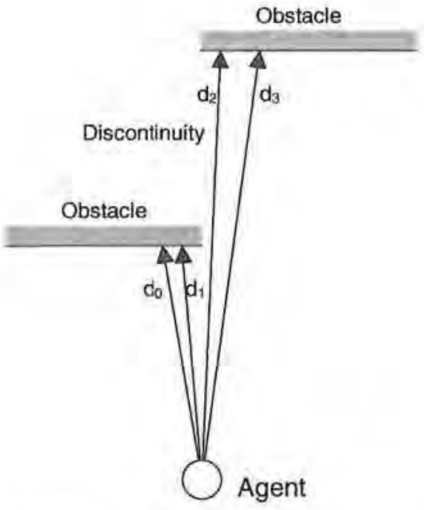


Figure B.1: Adjacent depth values used as input for the neural network.

Initially, a hand-coded algorithm was used to detect these discontinuities. Since these discontinuities are used to detect an opening in the environment where the agent can move through, the size of the discontinuity must be at least as large as the size of the agents avatar. This allows the rejection of most of the examined depth values, since the difference between the second and third depth value is usually very small. However, the algorithm still misclassified several instances. As a result, a feedforward neural network was considered to create a better classifier.

To construct a training set, all the depth values encountered during a mapping task where the difference between the second and third value exceeded the avatar size were collected. The instances that were incorrectly classified by the hand-coded algorithm were corrected manually. This produced 622 training examples from the entire run of the mapping algorithm, using the environment shown in figure 3.10 (of size 25x25) and an avatar size of 0.1. It is possible to create more training examples using the same environment by changing the size of the avatar, and changing the depth values provided as inputs correspondingly.

2 Implementation

The neural network that was used to solve this classification problem consists of 3 layers of 8, 6 and 1 nodes respectively (see Figure B.2). The learning algorithm used was the standard backpropagation algorithm, with some variations as described in [31].

In the standard algorithm, the values of the hidden nodes is calculated using equation (**B**.1):

$$h_i^\mu = \sum_j w_{ij}^\mu V_j^\mu = \sum_j w_{ij}^\mu g(h_j^\mu) \tag{B.1}$$

In this equation, w_{ij}^m is the weight of the connection from node V_j^{m-1} to V_i^m between layers m-1 and m, and g is the activation function used. At the end of every epoch, the weights are updated using equations (**B**.2), (**B**.3) and (**B**.4) :

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = \eta \sum_\mu \delta_i^\mu V_j^\mu \tag{B.2}$$

$$\delta_i^\mu = g'(h_i^\mu)[\zeta_i^\mu - O_i^\mu] \tag{B.3}$$

$$E = \tfrac{1}{2} \sum_{i\mu} [\zeta_i^\mu - O_i^\mu]^2 \tag{B.4}$$

In these equations, ζ_i^μ and O_i^μ are the target output and calculated output respectively of output i for training example μ, and η is a learning rate.

The following standard variations are used to improve the convergence speed of the neural network:

- Momentum: A momentum factor is added to equation (**B**.2) to prevent oscillation of the weights using the change of the weight of the previous time step, shown in equation (**B**.5):

$$\Delta w_{ij}(t+1) = -\eta \frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(t) \qquad (\mathbf{B}.5)$$

- Alternative cost function: A small value is added to cost function (**B**.3) to speed up convergence when the cost surface is relatively flat, as shown in equation (**B**.6):

$$\delta_i^\mu = \left(g'(h_i^\mu) + 0.1\right)\left(\zeta_i^\mu - O_i^\mu\right) \qquad (\mathbf{B}.6)$$

- Variable learning rate: The learning rate η is modified dynamically to speed up learning when the training error decreases and to learn more slowly when an increase in training error is detected, as shown in equation (**B**.7):

$$\Delta\eta = \begin{cases} +a & \text{if } \Delta E < 0 \text{ consistently} \\ -b\eta & \text{if } \Delta E > 0 \\ 0 & \text{otherwise} \end{cases} \qquad (\mathbf{B}.7)$$

- Noise was added to the training data to improve generalization.

3 Training results

3.1 Training run 1

In the first training experiment, the set of 622 training examples for an avatar size of 0.1 was used. The inputs of the neural network were the four distance values d_i of each sample in the training set, and the three differences between these distance values $dd_i = d_{i+1} - d_i$. Every node in the network is also connected with a weight to a constant value of 1, to create a bias value. After training, the neural net was able to correctly classify all members of the training set. However, when the mapping task was performed using an avatar size of 0.3, a lot of cases were classified incorrectly.

3.2 Training run 2

To improve the generalization of the neural network, the training set was expanded with the measurements obtained while mapping four regions in the environment using an avatar size of 0.3. The results of these measurements were manually corrected where necessary. This added 44 more training instances to the training set. However, the neural network was unable to converge to a solution that correctly classified all test cases, and 5 instances were classified incorrectly.

3.3 Training run 3

To improve the results of the neural network, two modifications were made to the neural network. First, instead of processing the entire training set at once, the neural net was trained on a small subset of the training set (consisting of 20 examples). When the neural net correctly classifies 95% of the reduced training set, new examples are added to the set until a new example is added that is incorrectly classified. When this happens, training is resumed until again 95% of the training set is classified correctly.

The second modification updates the weights of the neural network more frequently. Instead of calculating the errors of the entire training set, the weights are updated after 20 training examples are processed. This can result in a faster learning when the effect of the error values over the entire training set cancel each other out, which causes a smaller update of the weight values. If the weight values are updated more frequently, the chance that the errors cancel out each other is reduced and the updates have more effect.

The set of inputs was expanded with the values $ddd_i = dd_{i+1} - dd_i$ and $dddd_i = ddd_{i+1} - ddd_i$. The structure of the final neural network is displayed in Figure B.2. Nodes $h_{i,j}$ represents node i of hidden layer j, and node O_0 is the single output node.

The training examples of the second run are used. After the network has converged, all examples in the training set are classified correctly. The results of the network in the environment using an avatar size of 0.3 were good, but still a few errors occurred when the depth values were very large. These errors were removed by adding the misclassified examples to the training set. This resulted in good performance of the mapping algorithm for avatar sizes of 0.1, 0.3 and 0.5.

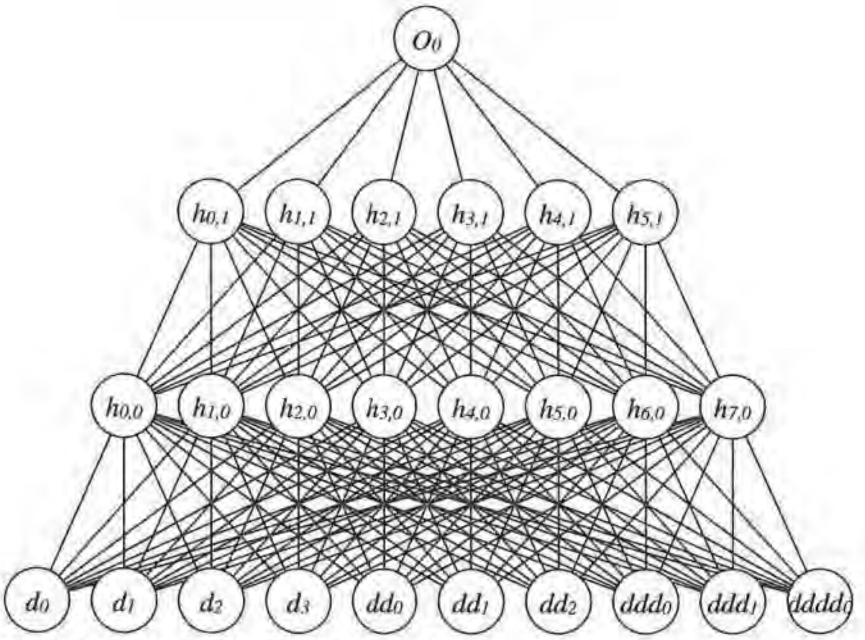


Figure B.2: Structure of the neural network.

Appendix C: Implementation of genetic programming

This appendix will document our implementation of some of the algorithms of the genetic programming system used in this thesis. First, we explain the algorithm used to calculate the minimum number of nodes required to construct an object of a given type of node (nodes of either abstract or concrete types). The algorithm to create new individuals will use this number. Finally, the algorithm used to perform a crossover operation on two strongly typed individuals is discussed.

1 The create algorithm

A simple algorithm to construct an individual of a given type *T* in a genetic programming system with strong hierarchical typing is shown below:

```
Function Create(NodeType T)
  Construct set S of all non-abstract types derived from T
  Select a type t from S
  N = Arity(t)
  For i = 1 to N do
    NodeType $t_i$ = the type required for child i of t
    Node $c_i$ = Create($t_i$)
  End for
  Return ConstructNodeOfTypeWithChildren(t, $c_1$, …, $c_N$)
End
```

While this algorithm creates a syntactically correct individual, it is difficult to predict the size of the created individual, or even if the algorithm will terminate at all (the size of an object is the number of nodes of that object). To control the size of new individuals when they are created, it is possible to add the maximum allowed size of the new individual to the Create function. To enforce this maximum size *M*, it must be possible that an object of size *M* or less can be created from the type *t* selected from the set *S* in the algorithm above. The size of the smallest object of a type *t* that can be created using a given set of primitives will be called MinInstanceSize(*t*). The algorithm used to calculate these values is described in the next section. Using the function MinInstanceSize, the new Create function is shown below:

```
Function Create(NodeType T, MaxSize M)
  Construct set S of all non-abstract types derived from T
  where MinInstanceSize(t) <= M
  Select a type t from S
  N = Arity(t)
  For i = 1 to N do
    NodeType $t_i$ = the type required for child i of t
    $w_i$ = MinInstanceSize($t_i$)
    $r_i$ = random()
  End for
  W = $\Sigma_i w_i$
  R = $\Sigma_i r_i$
  For i = 1 to N do
    // Distribute the available MaxSize (M - 1 - W) over
    // all children and add MinInstanceSize
    $m_i$ = $(r_i/R)*(M - 1 - W) + w_i$
    Node $c_i$ = Create($t_i$, $m_i$)
  End for
  Return ConstructNodeOfTypeWithChildren(t, $c_1$, …, $c_N$)
End
```

First, the algorithm selects a type that is able to construct an object smaller or equal to *M*. If the selected type is not a terminal type, the available size will be distributed over the child nodes. The available size is $M - 1$, as the parent node of type *t* has a size of 1. First, the minimum required size of every child node is calculated. This leaves $(M - 1 - W)$ nodes that can be distributed randomly over the *N* child nodes. The child nodes are created recursively with a size constraint m_i. The child nodes are then used to construct the node of type *t*. The use of MinInstanceSize removes the need to construct any "types possibilities tables" as described by Montana in [72].

2 Calculating MinInstanceSize of primitives

Table C.1 shows a simple example of a strongly-typed hierarchy of primitive types used for genetic programming. Primitive types are derived from the primitive type to the right of them. The number of children and their types are given between brackets. Abstract primitive types have no brackets.

The algorithm of the previous section requires the minimum possible size of an object of a given type. For example, objects of type Constant have a minimum size of 1 because they are terminals. Objects of type RealBase also have a minimum size of 1, because Constant is of type RealBase and has a minimum size of 1. The type Add has a minimum size of 3, because the minimum size of both its children is 1 and the Add-node has a size of 1.

Base	RealBase	Add(RealBase, RealBase)
		Sub(RealBase, RealBase)
		Constant()
	GraphicsBase	Line(RealBase)
		ScaleGraphic(RealBase, GraphicsBase)

Table C.1: A simple strongly-typed hierarchy of primitives.

To calculate the MinInstanceSize of all primitive types, the primitive types use two variables MinSize and InRecursive. MinSize contains the current minimum size of the object, and is initialized with $+\infty$. InRecursive is a Boolean flag that prevents infinite recursion and is initially False. The algorithm is presented below. The MinSize value must only be calculated once for each type, and a non-recursive call to MinInstanceSize will calculate the minimum size of the type, but not necessarily of other types for which MinInstanceSize is called recursively.

```
Function MinInstanceSize(NodeType T)
  If not T.InRecursive and T.MinSize = $+\infty$ then
    T.InRecursive = True
    Res1 = $+\infty$
    For every NodeType t derived from T do
      Res1 = MIN(Res1, MinInstanceSize(t))
    End for
    If not IsAbstract(T) then
      Res = 1  // Size of this node
      For i = 1 to Arity(T) do
        // Add MinSize of child nodes
        Res = Res + MinInstanceSize(ChildType(T, i))
      End for
      Res1 = MIN(Res, Res1)
    End if
    T.MinSize = Res1
    T.InRecursive = False
  End if
  Return T.MinSize
End
```

3 Strongly-typed crossover algorithm

The standard crossover algorithm for typeless genetic programming selects a random node in two individuals and swaps the subtrees originating at the selected nodes. In strongly-typed genetic programming, the selected subtrees must be compatible with the parent node of the selected subtrees. The crossover operator implemented by Montana [72] selects a random node from the first individual. The node in the second individual is selected from those nodes that return the same type as the first node.

The strongly typed crossover operator used in this thesis first selects a type that is found in both individuals, using the algorithm described below. Then, a node that is compatible with the selected type is selected in both individuals, and the subtrees originating at these nodes are exchanged to form two new individuals.

The algorithm to select a type that occurs in both individuals is given below. It is assumed that all individuals of the population must be derived from a type *T*.

```
For i = 1 to 2 do
  // Create a map $M_i$ of the types of the nodes of
  // individual $O_i$. This map is a set of tupels
  // (Type, Occurences), where Occurences is the number
  // of times a node of Type occurs in the individual.
  $M_i$ = MapTypes($O_i$, T)
End for
// Create the division of sets $M_1$ and $M_2$
M = {(Type, Occurences)|(Type, $O_1$) $\in$ $M_1$ and (Type, $O_2$) $\in$ $M_2$
                        and Occurences = MIN($O_1$, $O_2$)}
Select a type from M where types are weighted by their
occurences

Function MapTypes(Node N, NodeType T)
  // Add one occurrence of the root node
  Map M.add(T, 1)
  For i = 1 to Arity(N) do
    M.add(MapTypes(ChildNode(N, i), ChildType(N, i)))
  End for
  Return M
End
```

Appendix D: Complexity of the *N*-parity problem

The *N*-parity problems, described in section 5.2.2 of chapter 5 test if either an even or odd number of 1's are present in *N* input Boolean variables.

A simple solution for the odd *N*-parity problem, containing 2*N*-1 nodes, can be constructed when the xor function is allowed to be used with program (D.1):

$$(d_0 \text{ xor } d_1 \text{ xor } \dots \text{ xor } d_{n-1}) \tag{D.1}$$

When only the function set {and, or, nand, nor} is allowed, it is still possible to create every possible Boolean function. The function xor(a, b) can be constructed with equation (D.2):

$$\text{xor}(a, b) = ((a \text{ nand } b) \text{ and } (a \text{ or } b)) \tag{D.2}$$

When the xor functions of program (D.1) are replaced by the construction of equation (D.2), and N is 2^n for some $n \in \mathbb{N}_0$, $2N^2$-1 nodes are required. This can be demonstrated by induction:

***N* = 2:**

The solution for the odd 2-parity problem is ((d_0 nand d_1) and (d_0 or d_1)). This solution contains 7 nodes and therefore satisfies $2N^2-1$.

$N = 2^{n+1}$, $n > 1$

Let the function S_1 be a solution for the odd 2^n-parity problem for the inputs d_0, ..., d_{2^n-1} and S_2 a solution for the odd 2^n-parity problem for the inputs d_{2^n}, ..., $d_{2^{n+1}-1}$. Per induction, S_1 and S_2 contain $2(2^n)^2-1$ nodes. Then xor(S_1, S_2) is a solution for the odd 2^{n+1}-parity problem. Using equation (D.2), this results in the program ((S_1 nand S_2) and (S_1 or S_2)). This solution contains the three nodes {nand, and, or}, and four sub-expressions of size $2(2^n)^2-1$. The total number of nodes is thus $3+4*(2(2^n)^2-1) = 8*(2^n)^2-1 = 2*(2^2(2^n)^2)-1 = 2*(2^{n+1})^2-1 = 2N^2-1$.

While this does not demonstrate that it is impossible to find a smaller solution for the *N*-parity problem, it gives an indication that a solution is more complex without the xor-operator.

References

1. Andre, D.; Teller, A.: *A Study in Program Response and the Negative Effects of Introns in Genetic Programming.* In Genetic Programming 1996, Proceedings of the First Annual Conference, July 28-31 1996, Stanford University, MIT Press, pp. 12-20.
2. Andre, D.; Teller, A.: *Evolving Team Darwin United.* In Robocup-98: Robot Soccer World Cup II (Lecture Notes in Artificial Intelligence Vol. 1604). Springer-Verlag, New York, NY, 1999.
3. Angeline, P.: *Genetic Programming and Emergent Intelligence.* In Advances in Genetic Programming. MIT Press (1994) pp. 75-97.
4. Banzhaf, W.; Langdon, W. B.: *Some Considerations on the Reason for Bloat.* In Genetic Programming and Evolvable Machines, 3, 81-91, 2002.
5. Blickle, T.: *Evolving Compact Solutions in Genetic Programming: A Case Study.* In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, Hans-Paul Schwefel (Eds.): Parallel Problem Solving from Nature IV. Proceedings of the International Conference on Evolutionary, Berlin, September 1996. LNCS 1141, Heidelberg: Springer Verlag.
6. Blickle, T.; Thiele, L.: *Genetic Programming and Redundancy.* In Hopf, J., ed., Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken). Saarbrücken, Germany: Max-Planck-Institut fur Informatik (1994) pp. 33-38.
7. Blumberg, B. M.: *Old Tricks, New Dogs: Ethology and Interactive Creatures.* Ph.D. thesis, MIT Media Laboratory, Boston, MA, 1996.
8. Bryson, S.: *Developing Advanced Virtual Reality Applications.* In chapter Approaches to the Successful Design and Implementation of VR Applications, no. 2 in Course Notes for SIGGRAPH '94, ACM.
9. Capin, T. K.; Pandzic, I. S.; Noser, H.; Magnenat Thalmann, N.; Thalmann, D.: *Virtual Human Representation and Communication in VLNet Networked Virtual Environments.* IEEE Computer Graphics and Applications, Special Issue on Multimedia Highways, 1997.
10. Carlsson, C.; Hagsand, O.: *DIVE – a Platform for Multi-User Virtual Environments.* Computer & Graphics, vol. 17(6), pp. 663-669, 1993.
11. Chao, D.: *Doom as an Interface for Process Management.* In Proceedings of the Conference on Human Factors in Computing Systems (CHI), March 31 – April 5 2001, Volume 3, Issue 1, pp. 152-157.
12. Chen, M.; Foroughi, E.; Heintz, F.; Huang, Z. X.; Kapetanakis, S.; Kostiadis, K.; Kummeneje, J.; Noda, I.; Obst, O.; Riley, P.; Steffens, T.; Wang, Y.; Yin,

X.: *RoboCup Soccer Server.* Users manual for Soccer Server Version 7.07 and later, June 11, 2001.

13. Choset, H.: *Sensor Based Motion Planning: The Hierarchical Generalized Voronoi Graph*. Ph.D. thesis, California Institute of Technology, 1996.
14. Cohen, P. R.; Levesque, H. J.: *Teamwork.* In Nous, 25(4): pp. 487-512.
15. Coninx, K.: *Hybrid 2D/3D Human-Computer Interaction Techniques in Immersive Virtual Modeling Environments*. Ph.D. thesis, Limburg University Center, Diepenbeek, Belgium, 1997.
16. Coninx, K.; Van Reeth, F.; Flerackers, E.: *A Hybrid 2D/3D User Interface for Immersive Object Modeling.* In Proceedings of Computer Graphics International '97, Hasselt and Diepenbeek, BE, pp. 47-55, 1997.
17. Cossement, N: *Developing low-level skills for Robocup.* Master thesis, Department of Computer Science, University of Leuven, Belgium (1998).
18. Darwin, C.: *On the Origin of Species by Means of Natural Selection*. John Murray, 1859.
19. De Jong, E.; Watson, R.; Pollack, J.: *Reducing Bloat and Promoting Diversity using Multi-Objective Methods.* In Proceedings of the Genetic and Evolutionary Computation Conference, San Fransisco, USA, July 7-11 2001, pp. 11-18.
20. Di Girolamo, S.; Di Nardo, W.; Piciotti, P.; Paludetti, G.; Ottaviani, F. et al.: *Virtual Reality in Vestibular Assessment and Rehabilitation.* In Virtual Reality, vol. 4(3), pp. 169-183, 1999.
21. Dix, A.; Finlay, J., Abowd, G.; Beale, R.: *Human-Computer Interaction.* Prentice Hall, 1998.
22. Ekárt, A.: *Controlling Code Growth in Genetic Programming by Mutation.* in Late Breaking Papers of EUROGP'99, Göteborg, 26-27 May 1999, pp. 3-12, ISSN 1386-369X.
23. Ekárt, A.; Németh, S. Z.: *A Metric for Genetic Programs and Fitness Sharing.* in Genetic Programming, Proceedings of EUROGP'2000, Edinburgh, 15-16 April 2000, LNCS volume 1802, pp. 259-270, ISBN 3-540-67339-3.
24. Everett, S. S.; Wauchope K.; Pérez Q. M. A.: *Creating Natural Language Interfaces to VR Systems.* In Virtual Reality, vol. 4(2), pp. 103-113, 1999.
25. Gathercole, C.; Ross, P.: *An Adverse Interaction between the Crossover Operator and a Restriction of Tree Depth.* In Genetic Programming 1996, Proceedings of the First Annual Conference, July 28-31 1996, Stanford University, MIT Press, pp. 291-296.
26. Giallorenzo, V.; Bannerjee, P.; Conroy, L.; Franke, J.: *Application of Virtual Reality in Hospital Facilities Design.* In Virtual Reality, vol. 4(3), pp. 103-113, 1999.
27. Goldberg, D. E.: *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

28. Greenhalgh, C.; Benford, S.: *MASSIVE: a Distributed Virtual Reality System Incorporating Spatial Trading.* In Proceedings of the 15th International Conference on Distributed Computing Systems (DCS'95), Vancouver, Canada, May 30-June 2, 1995, pp. 27-34, IEEE Computer Society Press, Los Alamitos, California.
29. Handley, S.: *On the Use of a Directed Acyclic Graph to Represent a Population of Computer Programs.* in Proceedings of the 1994 IEEE World Congress on Computational Intelligence, pp. 154-159, Orlando, Florida, USA: IEEE Press.
30. Haynes, T. D.; Schoenefeld, D. A.; Wainwright, R. L.: *Type Inheritance in Strongly Typed Genetic Programming*. In Advances of Genetic Programming Volume 2, MIT Press (1996), pp. 359-375.
31. Hertz, J. A.; Krogh, A. S.; Palmer, R. G.: *Introduction to the Theory of Neural Computation.* Addison-Wesley, 1991.
32. Hill, R.; Chen, J.; Gratch, J.; Rosenbloom, P.; Tambe, M.: *Intelligent Agents for the Synthetic Battlefield: A Company of Rotary Wing Aircraft.* Innovative Applications of Artificial Intelligence (IAAI-97), 1997.
33. Hix, D.; Swan, J. E., Gabbard, J. L.; McGee, M.; Durbin, J. et al.: *User-Centered Design and Evaluation of a Real-Time Battlefield Visualization Virtual Environment.* In Proceedings of IEEE Virtual Reality '99, Houston, TE, USA, pp. 96-103, 1999.
34. Holland, J. H.: *Adaptation in natural and artificial systems.* Ann Arbor, MI: The University of Michigan Press (1975).
35. Hsu, W. H.; Gustafson, S. M.: *Genetic Programming for Layered Learning of Multi-agent Tasks.* In Late Breaking Papers of the 2001 Genetic and Evolutionary Computation Conference, July 9-11, 2001, pp. 176-182.
36. Iba, H.; De Garis, H; Sato, T.: *Genetic Programming Using a Minimum Description Length Principle.* In Advances in Genetic Programming. MIT Press (1994) pp. 265-284.
37. Jayaram, S.; Wang, Y.; Jayaram, U.; Lyons, K.; Hard, P.: *The Software Architecture of a Real-Time Battlefield Visualisation Virtual Environment.* In Proceedings of IEEE Virtual Reality '99, Houston, TE, USA, pp. 29-36, 1999.
38. Julier, S.; King, R.; Colbert, B.; Durbin, J.; Rosenblum, L.: *The Software Architecture of a Real-Time Battlefield Visualisation Virtual Environment.* In Proceedings of IEEE Virtual Reality '99, Houston, TE, USA, pp. 29-36, 1999.
39. Kalawski, R. S.: *The Science of Virtual Reality and Virtual Environments.* Addison-Wesley, 1993.
40. Keijzer, M.: *Efficiently Representing Populations in Genetic Programming*. In Advances of Genetic Programming Volume 2, MIT Press (1996), pp. 259-278.
41. Keller, R. E.; Banzhaf, W.: *Explicit Maintenance of Genetic Diversity on Genospaces*. Unpublished manuscript, June 1994, available at http:// citeseer.nj.nec.com/keller94explicit.html .

42. Kinnear, K. E.: *Generality and Difficulty in Genetic Programming: Evolving a Sort.* In proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93, july 17-21, pp. 287-294.

43. Kitano, H.; Asada, M.; Kuniyoshi, Y.; Noda, I.; Osawa, E.: *Robocup: The Robot World Cup Initiative.* Proceedings of the First International Conference on Autonomous Agents, Marina del Rey, CA, USA, February 5-8, 1997. ACM Press, New York, pp. 340-347.

44. Koza, J. R.: *Genetic Programming*. MIT Press, Cambridge, MA, 1992.

45. Koza, J. R.: *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, MA, 1994.

46. Kuffner, J. J. Jr; Latombe, J.-C.: *Fast Synthetic Vision, Memory, and Learning Models for Virtual Humans*. In proceedings Computer Animation 1999.

47. Langdon, W. B.: *Data Structures and Genetic Programming.* In Advances of Genetic Programming Volume 2, MIT Press (1996), pp. 395-414.

48. Langdon, W. B.: *Directed Crossover within Genetic Programming*. Research Note RN/95/71, University College London, September 1995.

49. Langdon, W. B.; Poli, R.: *Fitness Causes Bloat*. In Chawdhry. Soft Computing in Engineering Design and Manufacturing. Springer-Verlag London (1997) pp. 13-22.

50. Langdon, W. B.; Poli, R.: *Fitness Causes Bloat: Mutation.* In Genetic Programming: Proceedings of the First European Workshop, EuroGP'98, Paris, France, April 14-15 1998. Springer-Verlag, Berlin, pp. 37-48.

51. Langdon, W. B.: *Fitness Causes Bloat: Simulated Annealing, Hill Climbing and Populations.* Technical report CSRP-97-22, available at ftp://ftp.cs.bham. ac.uk/pub/authors/W.B.Langdon/papers/CSRP-97-22.ps.gz .

52. Langdon, W. B.; Poli, R.: *Genetic Programming Bloat with Dynamic Fitness.* In Genetic Programming: Proceedings of the First European Workshop, EuroGP'98, Paris, France, April 14-15 1998. Springer-Verlag, Berlin, pp. 97-112.

53. Langdon, W. B.: *Quadratic Bloat in Genetic Programming.* In GECCO-2000: Proceedings of the Genetic and Evolutionary Computation Conference, july 8-12 2000.

54. Lee, D.; Vredevoe, D.; Kimmick, J.; Karplus, W. J.; Valentino, D. J.: *Ophthalmoscopic Examination Training Using Virtual Reality.* In Virtual Reality, vol. 4(3), pp. 184-191, 1999.

55. Levesque, H. J.; Cohen, P. R.; Nunes, J. H. T.: *On Acting Together.* In Proceedings of Eighth National Conference on Artificial Intelligence (AAAI-90), Boston, MA, USA, pp. 94-99, 1990.

56. Luke, S.; Hohn, C.; Farris, J.; Jackson, G.; Hendler, J.: *Co-evolving Soccer Softbot Team Coordination with Genetic Programming.* In Proceedings of the

First International Workshop on Robocup, at the International Joint Conference on Artificial Intelligence, Nagoya, Japan, 1997.

57. Luke, S.: *Code Growth is Not Caused by Introns.* In Late Breaking Papers of the 2000 Genetic and Evolutionary Computation Conference, July 8, 2000, Las Vegas, Nevada, USA, pp. 228-235.

58. Luke, S.: *Evolving Soccerbots: a Retrospective.* In Proceedings of the 12th Annual Conference of the Japanese Society for Artificial Intelligence, 1998.

59. Luke, S.; Spector, L.: *Evolving Teamwork and Coordination with Genetic Programming.* In Genetic Programming 1996, Proceedings of the First Annual Conference, July 28-31 1996, Stanford University, MIT Press, pp. 150-156.

60. Luke, S.: *Genetic Programming Produced Competitive Soccer Softbot Teams for Robocup97.* In Genetic Programming 1998: Proceedings of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin, USA, July 22-25 1998, Morgan Kaufman, pp. 214-222.

61. Luke, S.: *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat.* Ph.D. dissertation, Department of Computer Science, University of Maryland, College Park, Maryland (2000).

62. Mahwinney, D.: *Prevention of Premature Convergence in Genetic Programming.* Honours Thesis, RMIT, Department of Computer Science, 2000.

63. McPhee, N. F.; Miller, J. D.: *Accurate Replication in Genetic Programming.* In Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95), july 15-19 1995, San Francisco, California, USA, pp. 303-309.

64. McPhee, N. F.; Poli, R.: *A Schema Theory Analysis of the Evolution of Size in Genetic Programming with Linear Representations.* In J. Miller et al. (Eds.): EuroGP 2001, LNCS 2038, pp. 108-125, 2001.

65. McPhee, N. F.; Hopper, N. J.; Reierson, M. L.: *Impact of Types on Essentially Typeless Problems.* In Genetic Programming 1998: Proceedings of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin, USA, July 22-25 1998, pp. 232-240.

66. Mine, M. R.: *Working in a Virtual World: Interaction Techniques Used in the Chapel Hill Immersive Modeling Program.* Technical report 96-029, University of North Carolina, 1996.

67. Monsieurs, P.: *Developing High Level Skills for Robocup.* Master thesis, Department of Computer Science, University of Leuven, Belgium (1998).

68. Monsieurs, P.; Coninx, K.; Flerackers, E.: *Collision Avoidance and Map Construction Using Synthetic Vision.* In Proceedings of the Second Workshop on Intelligent Virtual Agents, University of Salford, UK, September 13, 1999, pp. 33-45.

69. Monsieurs, P.; Coninx, K.; Flerackers, E.: *Collision Avoidance and Map Construction Using Synthetic Vision.* In Virtual Reality (2000) 5: pp. 72-81.

70. Monsieurs, P.; Flerackers, E.: *Reducing Bloat in Genetic Programming*. In Computational Intelligence, Proceedings of 7th Fuzzy Days, Dortmund, Germany, October 1-3 2001, pp. 471-478, ISBN 3-540-42732-5.

71. Monsieurs, P.; Flerackers, E.: *Increasing the Diversity of a Population in Genetic Programming*. In Proceedings of the Genetic and Evolutionary Computation Conference, San Francisco, California, USA, July 7-11, 2001, p. 185.

72. Montana, D. J.: *Strongly Typed Genetic Programming*. In Evolutionary Computation, 3(2) pp. 199-230.

73. Muslea, I.: *A General Purpose (AI) Planning System based on the Genetic Programming Paradigm*. In Late Breaking Papers of the 1997 Genetic Programming Conference, July 13-16, 1997, pp. 157-164.

74. Muslea, I.: *SINERGY: A Linear Planner based on Genetic Programming*. In Proceedings of the Fourth International Conference on Planning, Toulouse, France, September 24-26, 1997.

75. Neumann, P.; Siebert, D.; Schulz, A.; Faulkner, G.; Krauss, M. et al.: *Using Virtual Reality Techniques in Maxillofacial Surgery Planning*. In Virtual Reality, vol. 4(3), pp. 213-222, 1999.

76. Nienhuys-Cheng, S.-H.: *Distance Between Herbrand Interpretations: a Measure for Approximations to a Target Concept*. In N. Lavraĉ, S. Dẑeroski (eds): Proceedings of the 7th International Workshop on Inductive Logic Programming, volume 1297 of LNAI, pp. 213-226. Springer-Verlag, 1997.

77. Nordin, P; Banzhaf, W.: *Complexity Compression and Evolution*. In Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA 95), Pitssburg, PA, USA, july 15-19, 1995, pp. 310-317.

78. Nordin, P.; Banzhaf, W.; Francone, F. D.: *Introns in Nature and in Simulated Structure Evolution*. In Bio-Computation and Emergent Computation, World Scientific Publishing, Skovde, Sweden (1997).

79. Noser, H.; Renault, O.; Thalmann, D.; Magnenat Thalmann, N.: *Navigation for Digital Actors based on Synthetic Vision, Memory and Learning*. In Computers and Graphics, Pergamon Press, Vol. 19, No. 1, 1995, pp. 7-19.

80. O'Neill, M.; Ryan, C.; Nicolau, M.: *Grammar Defined Introns: An Investigation Into Grammars, Introns, and Bias in Grammatical Evolution*. In Proceedings of the Genetic and Evolutionary Computation Conference, San Fransisco, USA, July 7-11 2001, pp. 97-103.

81. O'Reilly, U.-M.: *An Analysis of Genetic Programming*. Ph.D. Dissertation, Carleton University, Ottawa-Carleton Institute for Computer Science, Ottawa, Ontario, Canada (1995).

82. Pandzic, I. S.; Capin, T. K.; Lee, E.; Magnenat Thalmann, N.; Thalmann, D.: *A Flexible Architecture for Virtual Humans in Networked Collaborative Virtual Environments*. In Proceedings Eurographics '97, Budapest, Hungary, 1997.

83. Pandzic, I. S.; Capin, T. K.; Lee, E.; Magnenat Thalmann, N.; Thalmann, D.: *Autonomous Actors in Networked Collaborative Virtual Environments.* In Proceedings of Multimedia Modeling '98, Lausanne, Switzerland, October 12-15, 1998, pp. 138-145.

84. Poli, R.; McPhee, N. F.: *Exact Schema Theorems for GP with One-Point and Standard Crossover Operating on Linear Structures and Their Application to the Study of the Evolution of Size*. In J. Miller et al. (Eds.): EuroGP 2001, LNCS 2038, pp. 126-142, 2001.

85. Poli, R.; McPhee, N. F.: *Exact Schema Theory for GP and Variable-length Gas with Homologous Crossover.* In Proceedings of the Genetic and Evolutionary Computation Conference, San Fransisco, USA, July 7-11 2001, pp. 104-111.

86. Poli, R.: *Introduction to Evolutionary Computation.* Online tutorial: http://www.cs.bham.ac.uk/~rmp/slide_book/slide_book.html .

87. Rajani, R.; Perry, M.: *The Reality of Medical Work: The Case for a New Perspective on Telemedicine*. In Virtual Reality, vol. 4(4), pp. 243-249, 1999.

88. Raymaekers, C.: *Haptic Feedback in Virtual Environments: Towards a Multi-modal Interface.* Ph.D. dissertation, Department of Computer Science, Transnationale Universiteit Limburg, Belgium, 2002.

89. Raymaekers, C.; De Weyer, T.; Coninx, K.; Van Reeth, F.; Flerackers, E.: *ICOME: An Immersive Collaborative 3D Object Modelling Environment*. In Virtual Reality (1999) 4: pp. 265-274.

90. Reynolds, C. W.: *Not Bumping Into Things*. Notes on "obstacle avoidance" for the course on Physically Based Modeling at SIGGRAPH 88, August 1-5, Atlanta, Georgia, 1988.

91. Rosca, J. P.: *An Analysis of Hierarchical Genetic Programming*. Technical Report 566, University of Rochester, Rochester, NY, USA, 1995. http://citeseer.nj.nec.com/rosca95analysis.html.

92. Rosca, J. P.: *Entropy-Driven Adaptive Representation.* In Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications, Tahoe City, California, USA, july 9, 1995, pp. 23-32.

93. Rosca, J. P.: *Generality versus Size in Genetic Programming.* In Genetic Programming 1996, Proceedings of the First Annual Conference, July 28-31 1996, Stanford University, MIT Press, pp. 381-387.

94. Ryan, C.: *Pygmies and Civil Servants*. In Advances in Genetic Programming. MIT Press (1994) pp. 243-263.

95. Sastry, L.; Boyd, D. R. S.: *Human Factors in Virtual Environments.* In Virtual Reality, vol. 3(4), pp. 223-225, 1998.

96. Singhal, S.; Zyda, M.: *Networked Virtual Environments: Design and Implementation*. SIGGRAPH Book Series, ACM Press, 1999.

97. Smith, P. W. H.; Harries, K.: *Code Growth, Explicitly Defined Introns, and Alternative Selection Schemes.* In Evolutionary Computation, Vol. 6, No. 4, pp. 339-360, 1999.

98. Soule, T.; Foster, J. A.; Dickinson, J.: *Code Growth in Genetic Programming.* In Genetic Programming 1996, Proceedings of the First Annual Conference, July 28-31 1996, Stanford University, MIT Press, pp. 215-223.
99. Soule, T.; Foster, J. A.: *Effects of Code Growth and Parsimony Pressure on Populations in Genetic Programming.* In Evolutionary Computation, 6(4) (1998) pp. 293-309.
100. Soule, T.; Foster, J. A.: *Removal Bias: a New Cause of Code Growth in Tree Based Evolutionary Programming*. In 1998 IEEE International Conference on Evolutionary Computation. Anchorage, Alaska, USA: IEEE Press (1998) pp. 181-186.
101. Soule, T.; Heckendorn, R. B.: *An Analysis of the Causes of Code Growth in Genetic Programming.* Genetic Programming and Evolvable Machines, 3, Kluwer Academic Publishers, pp. 283-309, September 2002.
102. Stone, P.; Veloso, M.: *A Layered Approach to Learning Client Behaviors in the Robocup Soccer Server.* In Applied Artificial Intelligence (AAI), Volume 12, 1998.
103. Tackett, W. A.: *Recombination, Selection, and the Genetic Construction of Computer Programs*. Ph.D. Dissertation, University of Southern California, Department of Electrical Engineering Systems (1994).
104. Tambe, M.: *Implementing Agent Teams in Dynamic Multi-agent Environments.* In Applied Artificial Intelligence (AAI), Volume 12, 1998.
105. Teller, A.; Veloso, M.: *Neural Programming and an Internal Reinforcement Policy.* Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996.
106. Teller, A.: *The Internal Reinforcement of Evolving Algorithms.* In Advances in Genetic Programming Volume 3, MIT Press(1999), pp. 325-354.
107. Terzopoulos, D.; Rabie, T. F.: *Animat Vision: Active Vision in Artificial Animals.* In Journal of Computer Vision Research Fall 1997, Volume 1, Number 1, The MIT Press.
108. Thrun, S.: *Learning Metric-topological Maps for Indoor Mobile Robot Navigation*. In Artificial Intelligence, 99 (1) (1998) pp. 21-71
109. Thrun, S.; Bücken, A.: *Integrating Grid-Based and Topological Maps for Mobile Robot Navigation*. In Proceedings of the Thirteenth National Conference on Artificial Intelligence AAAI, Portland, Oregon, August 1996.
110. Westerberg, C. H.; Levine, J.: *Optimizing Plans Using Genetic Programming.* In Proceedings of the 6th European Conference on Planning, Toledo, Spain, September 2001, Springer Verlag.
111. Wineberg, M.; Oppacher, F.: *The Benefits of Computing with Introns.* In Genetic Programming 1996, Proceedings of the First Annual Conference, July 28-31 1996, Stanford University, MIT Press, pp. 410-415.
112. Woolridge, M.; Jennings, N.: *Agent Theories, Architectures, and Languages: A Survey.* In The Knowledge Engineering Review, 10(2), 1995.

113. Zhang, B.-T.; Mühlenbein, H.: *Balancing Accuracy and Parsimony in Genetic Programming*. In Evolutionary Computation, Vol. 3, No. 1, 1995, pp. 17-38.
114. Zhukov, S.; Iones, A.; Kronin, G.: *Navigation of Intelligent Characters in Complex 3D Synthetic Environments in Real-time Applications*. Proc. of WSCG'98 - Central European conference on Computer Graphics and Visualization 1998, pp. 456-463.
115. *FIDE Swiss Rules*. Fédération Internationale des Échecs, available online at http://www.fideonline.com/official/handbook.asp?level=C04.

Samenvatting

In dit gedeelte wordt een korte Nederlandstalige samenvatting van de gehele thesis gegeven. Voor een meer gedetailleerde beschrijving wordt verwezen naar het Engelstalige gedeelte van de thesis.

1 Inleiding en overzicht

Deze thesis behandelt agenten in virtuele omgevingen. Virtuele omgevingen zijn werelden die bestaan in het geheugen van een computersysteem. Gebruikers kunnen met behulp van grafische hardware deze wereld waarnemen en interageren met deze wereld. Agenten zijn zelfstandige computerprogramma's die eenvoudig te automatiseren taken uitvoeren voor gebruikers of andere agenten. Een agent die bestaat in een virtuele omgeving is een virtuele agent.

In het eerste gedeelte van deze thesis wordt behandeld hoe een agent op een natuurlijke manier kan navigeren in een virtuele omgeving, en wordt robotvoetbal geïntroduceerd als een virtuele multi-agent omgeving. Het tweede gedeelte van de thesis behandelt het aspect van leren van virtuele agenten. Hiervoor wordt genetisch programmeren geïntroduceerd. Vervolgens worden enkele optimalisaties toegepast op genetisch programmeren die nuttig zijn voor het trainen van virtuele agenten. Ten slotte worden deze optimalisaties toegepast op het domein van robotvoetbal om uit te testen of deze optimalisaties ook werken op een complex probleem.

2 Agenten en virtuele omgevingen

Door de steeds beter wordende grafische hardware van computersystemen worden virtuele omgevingen steeds vaker gebruikt in uiteenlopende toepassingen. Deze toepassingen omvatten medische en militaire applicaties, ontwerp en constructie, modelleren, virtuele gemeenschappen en ontspanning. In al deze toepassingen is de gebruiker ondergedompeld in een computergegenereerde wereld om zo op een meer natuurlijke manier te interageren met de applicatie. In een aantal gevallen kan het nuttig zijn om een aantal taken te laten uitvoeren door een zelfstandig computerprogramma, een virtuele agent genaamd. De kenmerken van agenten zijn dat ze zelfstandig werken, reageren op gebeurtenissen in hun omgeving en, wanneer gepast, zelf initiatieven nemen, en kan communiceren met gebruikers en andere agenten. Een virtuele agent heeft, net als een gebruiker, een representatie in een virtuele omgeving die een avatar wordt genoemd.

Bij het ontwerpen van een virtuele agent treden verschillende moeilijkheden op. Vermits de virtuele agent een representatie heeft in de omgeving, is het nodig dat de agent op een natuurlijke manier kan navigeren in de omgeving. Dit wordt behandeld in sectie 3 van deze samenvatting.

Vervolgens is het in een multi-agent systeem nodig dat de verschillende agenten met elkaar en met de gebruikers van het systeem kunnen communiceren. Dit onderwerp wordt niet behandeld in deze samenvatting, maar wordt kort behandeld in Appendix A van het Engelstalige gedeelte van deze thesis.

Ten slotte is het nodig dat virtuele agenten leren een taak te verrichten in hun omgeving. In deze thesis wordt genetisch programmeren beschouwd als een manier om agenten te trainen. Genetisch programmeren heeft echter enkele problemen, die belangrijk zijn in een domein zoals virtuele agenten waar evaluatie van programma's zeer lang kan duren. Wanneer genetisch programmeren wordt gebruikt, wordt vastgesteld dat de grootte van programma's zeer snel groeit, zonder veel bij te dragen aan de kwaliteit van het programma. Vaak zorgt deze groei er zelfs voor dat de evolutie van programma's sterk vertraagd. Ook is het nodig om gebruik te maken van een grote populatie van kandidaat oplossingen om de diversiteit van de populatie te behouden. Omdat dit voor zeer lange evaluatietijd zorgt, is het nodig dat de grootte van de populatie verminderd kan worden zonder de diversiteit ervan aan te tasten. Oplossingen voor deze twee problemen worden besproken in secties 6 en 7 van deze samenvatting. Verder worden deze oplossingen ook toegepast op het multi-agent domein van robotvoetbal, om de efficiëntie van deze oplossingen te testen op een complex leerprobleem. Dit zal worden besproken in sectie 8.

3 Navigatie in virtuele omgevingen

Een virtuele agent kan op verschillende manieren navigeren in een virtuele wereld. Als de agent toegang heeft tot de interne representatie van de omgeving kan hij een kaart opstellen van de omgeving. Met behulp van deze kaart kan hij dan routes plannen naar andere plaatsen en vaste obstakels in de omgeving vermijden.

Indien de agent echter geen toegang heeft tot de interne representatie zal hij zelf een representatie van de omgeving moeten opbouwen. Dit moet gebeuren via de sensors waarmee hij de omgeving kan waarnemen, zoals een synthetische visuele sensor die een beeld genereert vanuit het oogpunt van de agent.

3.1 Synthetische visie

In deze sectie wordt verondersteld dat de agent de omgeving enkel kan observeren met behulp van het gerenderde beeld van de omgeving. Vermits dit gerenderde beeld de diepte-informatie van elke pixel bevat die gebruikt werd om het beeld te genereren, kan hier gebruik van worden gemaakt. Dit proces is vergelijkbaar met een mobiele robot die een dieptesensor heeft. In dit gedeelte wordt enkel gebruik gemaakt van de diepte-informatie op dezelfde hoogte van de agent. De diepte-informatie geeft informatie over de afstand tot objecten in de omgeving in een hoek van 90° voor de agent. Om de informatie te krijgen van de volledige omgeving wordt de diepte-informatie opgeslagen in een gezichtsbuffer. Vervolgens roteert de agent om zijn eigen as om een nieuw stuk van de omgeving te observeren. Al deze stukken worden samen opgeslagen in de gezichtsbuffer, tot de volledige omgeving is geobserveerd. Na elke beweging van de agent worden de gegevens in de gezichtsbuffer aangepast aan zijn nieuwe positie. De opgeslagen diepte-informatie kan nu worden gebruikt voor twee doeleinden: het ontwijken van obstakels en het construeren van een kaart van de omgeving.

3.2 Ontwijken van obstakels

Als de afstand tot obstakels voor de agent gekend is, kan de agent deze gegevens gebruiken om een botsing met deze obstakels te vermijden tijdens het bewegen. Dit gebeurt zowel voor het ontwijken van obstakels op korte als op middellange afstand. Op korte afstand wordt rekening gehouden met de beweging van de agent tijdens één volgende tijdstap. Als deze beweging zou leiden tot een botsing, wordt de beweging in voldoende mate aangepast om een botsing te vermijden.

Op middellange afstand wordt gekeken naar de huidige doelpositie waarnaar de agent aan het bewegen is. Deze doelposities worden gegenereerd door functies van de agent die werken op een hoger niveau, zoals het niveau dat een kaart van de omgeving genereert. Als de doelpositie niet kan bereikt worden in een rechte lijn omdat er een obstakel is, wordt gekeken of het na een kleine aanpassing van het pad naar de doelpositie mogelijk is om hier toch voldoende dicht bij te komen. Als dit zo is zal een nieuwe tijdelijk doelpositie worden gegenereerd op dit aangepaste pad. Wanneer de oorspronkelijke doelpositie zichtbaar wordt, wordt de tijdelijke doelpositie vervangen door de oorspronkelijke doelpositie. Als een aanpassing van het pad naar de doelpositie echter niet mogelijk is, zal het hogere niveau dat de doelpositie genereerde hiervan op de hoogte worden gebracht. Het is dan de taak van het hogere niveau om dit probleem op te lossen.

3.3 Constructie van een kaart van de omgeving

Als de diepte-informatie van alle obstakels rondom de agent gekend is, kan een kaart worden gemaakt van het gebied waarin de agent zich bevindt. De langs elkaar liggende dieptewaarden worden benaderd door een aantal rechte lijnen waar dit mogelijk is. Deze lijnen worden opgenomen in de kaart van dit gebied. Op sommige plaatsen zullen echter discontinuïteiten voorkomen in deze lijnen. Deze open punten stellen openingen voor waarlangs het mogelijk is om aangrenzende gebieden te betreden. De open punten worden eveneens opgeslagen in de kaart van het gebied, en stellen mogelijke plaatsen voor om de kaart van de omgeving uit te breiden met nieuwe gebieden. Om deze discontinuïteiten te detecteren wordt gebruik gemaakt van een neuraal netwerk.

Om nieuwe gebieden toe te voegen aan de kaart worden de open punten van een bestaand gebied beschouwd. Als dit open punt leidt naar een reeds bestaand gebied, wordt het beschouwde open punt verwijderd en wordt een verbinding tussen deze twee gebieden toegevoegd. In het andere geval beweegt de agent naar het nieuwe gebied en detecteert de diepte-informatie in het nieuwe gebied dat wordt toegevoerd aan de kaart. Wanneer alle open punten in alle gebieden van de kaart zijn verwijderd, is de volledige kaart van de omgeving voltooid. Op dat moment kunnen overbodige verbindingen tussen gebieden worden verwijderd om de kaart te vereenvoudigen.

3.4 Het vinden van paden met behulp van de kaart

Met behulp van de kaart van de omgeving is het mogelijk om het kortste pad te vinden tussen twee punten in de omgeving. In een eerste stap worden de twee gebieden van de kaart gevonden waarvan hun centrum het dichtst bij de begin- en eindpositie van het pad liggen. Vervolgens kan een standaard padvinderalgoritme, zoals het A*-algoritme, worden gebruikt om het kortste pad tussen deze twee gebieden te vinden. Het gevonden pad is dan een aaneenschakeling van rechte lijnen, waarvan de verbindingspunten worden gebruikt als doelposities van het navigatiesysteem van de agent. Zodra een volgend punt op het pad zichtbaar wordt, wordt de huidige doelpositie vervangen door de volgende doelpositie. Op deze manier kunnen hoeken in het gevonden pad worden afgesneden.

Padvinden en het construeren van een kaart kunnen worden gecombineerd om een positie te bereiken in een onbekende omgeving. In dit geval worden eerst de open punten in een gebied onderzocht die het dichtst liggen bij de te bereiken doelpositie.

4 Robocup

Robocup is een onderzoeksdomein naar multi-agent systemen dat gebruik maakt van het domein van robotvoetbal. Het doel van Robocup is in de eerste plaats om een team van echte robots te ontwikkelen die een voetbalwedstrijd kunnen spelen. Om echter ook onderzoek te kunnen doen naar de multi-agent aspecten van robotvoetbal, zonder aandacht te moeten besteden aan de complexe problemen van robotica, is ook een software simulator ontwikkeld. In deze simulator wordt het besturen van robots geabstraheerd door eenvoudige primitieve commando's zoals draaien en vooruit bewegen.

4.1 Uitdagingen van Robocup

Het Robocup domein bevat een aantal interessante uitdagingen voor multi-agent systemen, waar elke speler wordt bestuurd door een agent. De omgeving is uiterst dynamisch, waardoor lange termijn plannen constant moeten worden aangepast aan de veranderende situaties. De sensorgegevens die de spelers ontvangen bevatten ook ruis en zijn soms onvolledig. Bijgevolg moeten de agenten fouttolerant zijn om deze fouten op te kunnen vangen. De communicatie tussen de spelers is ook beperkt, waardoor een gedistribueerde vorm van samenwerking noodzakelijk wordt. De spelers moeten ook rekening houden met een beperkte hoeveelheid uithoudingsvermogen, waardoor inspanningen gedoseerd moeten worden. De simulatie verloopt in real-time, waardoor het snel nemen van beslissingen noodzakelijk is. Tenslotte weet een speler ook niet of een uitgevoerde actie geslaagd is. De agent moet de veranderingen in de omgeving bestuderen om het effect van de actie te observeren.

4.2 Werking van Robocup

De simulator werkt volgens een client-server principe. Verschillende agenten verbinden met de simulator via een netwerk. Op regelmatige intervallen ontvangen de agenten visuele en auditieve informatie over de objecten die worden geobserveerd door de speler. Elke tijdstap kan de agent commando's sturen naar de simulator om de speler te besturen. Dit zijn primitieve commando's zoals draaien rond de as van de speler, vooruit bewegen of de bal in een bepaalde richting trappen als deze zich dicht genoeg bij de speler bevindt.

Om het ontwerp van een Robocup agent te vereenvoudigen wordt de implementatie opgedeeld in een aantal verschillende lagen. Op het laagste niveau bevinden zich eenvoudige taken zoals het onderscheppen van de bal of het bewegen naar een positie op het veld. Deze taken maken gebruik van de primitieve acties van de simulator. Daarboven zijn er taken om samen te werken met één andere speler, zoals het geven van een pas. Nog hoger zijn taken in verband met meerdere andere spelers, zoals het ontwijken van tegenstanders. In de volgende laag wordt bepaald welke actie van een lager niveau de agent op een bepaald moment gaat uitvoeren. Hier wordt bijvoorbeeld beslist of het beter is om met de bal te dribbelen of een pas te geven. De hoogste laag behandelt teamwerk. Hier wordt bijvoorbeeld bepaald welke posities de spelers innemen op het veld en wat hun taak is.

De moeilijkste taak van een agent in het Robocup domein is het leren van de actie selectie. In deze thesis zullen evolutionaire algoritmes worden gebruikt voor deze taak. Zowel genetische algoritmes als genetisch programmeren worden hiervoor beschouwd. Deze technieken worden besproken in sectie 5, en worden toegepast op het Robocup domein in sectie 8 van deze samenvatting.

4.3 Ontwijken van obstakels in Robocup

De technieken om obstakels te vermijden die beschreven werden in sectie 3.2 kunnen ook worden gebruikt in Robocup. Vermits de agent afstandsinformatie ontvangt over de andere spelers die zichtbaar zijn, kan een gezichtsbuffer worden bijgehouden van de andere spelers. Deze gezichtsbuffer kan dan worden gebruikt om te bewegen in een richting die andere spelers vermijt, en waarbij vooral afstand wordt gehouden van tegenstanders. De gezichtsbuffer wordt ook gebruikt om te bepalen in welke richting een veilige pas kan worden gegeven. In dit geval worden de richtingen en afstanden van medespelers en tegenstanders vergeleken.

5 Genetisch programmeren

Evolutionaire algoritmes zijn zoektechnieken die gebaseerd zijn op de natuurlijke evolutietheorie beschreven door Darwin in de 19^e eeuw. Volgens deze theorie worden in de natuur eigenschappen die een organisme helpen overleven vaker doorgegeven aan hun nageslacht, waardoor deze goede eigenschappen in meer individuen van een populatie gaan voorkomen. Dit gebeurt omdat de organismen met goede eigenschappen meer kans hebben om lang genoeg te leven om zich te kunnen voortplanten. Om dezelfde reden zullen organismen met slechte eigenschappen zich minder vaak kunnen voortplanten, waardoor de slechte eigenschappen verdwijnen uit de populatie. Bijgevolg worden de organismen na verloop van tijd steeds beter in het overleven in hun omgeving.

5.1 Evolutionaire algoritmes

De principes van de evolutietheorie kunnen ook worden gebruikt voor het oplossen van problemen in de informatica. In plaats van organismen in de natuur worden nu oplossingen voor problemen beschouwd. Meerdere kandidaat oplossingen worden bijgehouden in een populatie. Afhankelijk van hoe goed een beschouwde kandidaat oplossing het probleem kan oplossen wordt bepaald hoeveel kans deze kandidaat oplossing heeft om zich voort te planten. Betere kandidaat oplossingen zullen zich vaker voortplanten, en door het combineren van verschillende kandidaat oplossingen kunnen nieuwe kandidaat oplossingen gevonden worden die beter zijn dan elk van hun ouders.

Om evolutietheorie toe te passen op informaticaproblemen moeten de volgende principes worden gesimuleerd:

- **Selectie:** Kandidaat oplossingen die het probleem beter oplossen moeten vaker worden geselecteerd om nieuwe kandidaat oplossingen te genereren. Bijgevolg is het nodig dat de kwaliteit van een kandidaat oplossing kan bepaald worden. Hiervoor wordt een fitness functie gebruikt. Deze functie evalueert een kandidaat oplossing, en kent een fitness waarde toe aan de kandidaat oplossing. Deze fitness waarde wordt dan gebruikt om de kans te bepalen dat de kandidaat oplossing wordt geselecteerd.
- **Reproductie:** Het creëren van nieuwe kandidaat oplossingen wordt reproductie genoemd. Er wordt een onderscheid gemaakt tussen seksuele en aseksuele reproductie. In het eerste geval worden twee ouders gecombineerd tot één nieuwe kandidaat oplossing, in de hoop de goede eigenschappen van beide ouders te combineren in het kind. In het tweede geval wordt één enkele ouder gebruikt om kind te creëren. Op deze manier worden goede eigenschappen bewaard in de populatie.
- **Mutatie:** Een vorm van aseksuele reproductie is mutatie. In dit geval wordt een gedeelte van de kandidaat oplossing op een willekeurige manier veranderd. Het doel van mutatie is om elementen te introduceren in de populatie die voordien niet aanwezig waren, en om de diversiteit van de populatie te vergroten.

Het evolutionaire algoritme werkt door een willekeurige populatie van kandidaat oplossingen te genereren. Vervolgens wordt een nieuwe populatie gegenereerd met behulp van reproductie, waarna deze nieuwe populatie de oude vervangt. Deze stap wordt een generatie genoemd. Dit proces wordt herhaald tot een voldoende goede oplossing is gevonden, of een maximum aantal generaties is bereikt.

5.2 Genetische algoritmes

Een genetisch algoritme is een evolutionair algoritme waar een oplossing wordt gezocht voor een bepaald probleem, zoals een verzameling optimale parameters voor een proces. Een kandidaat oplossing in een genetisch algoritme wordt meestal voorgesteld door een reeks elementen. De lengte van deze reeks is dikwijls contant voor alle kandidaat oplossingen van de populatie. De elementen van de reeks kunnen bits, getallen, symbolen enz. zijn, afhankelijk van het op te lossen probleem.

De meest gebruikte vorm van seksuele reproductie bij genetische algoritmes is crossover. Bij deze operator worden de reeksen van twee kandidaat oplossingen geselecteerd. Vervolgens wordt een crossover punt gekozen in de reeks van elementen. Twee nieuwe kandidaat oplossingen worden gecreëerd door de elementen van de eerste ouder voor het crossover punt te combineren met de elementen achter het crossover punt van de tweede ouder, en vice versa. Er zijn ook varianten mogelijk waarbij meerdere crossover punten worden geselecteerd. Mutatie gebeurt meestal door een willekeurig element van de reeks te vervangen door een nieuw element.

5.3 Genetisch programmeren

Genetisch programmeren is een specialisatie van genetische algoritmes waarbij de kandidaat oplossingen programma's zijn die een verzameling van problemen kunnen oplossen. Een kandidaat oplossing van genetisch programmeren wordt meestal voorgesteld door een boom van primitieve elementen. De primitieve elementen worden opgedeeld in twee verzamelingen: terminale en niet-terminale elementen. Niet-terminale elementen bevatten een aantal kinderen die worden gebruikt bij de evaluatie van het element. Een voorbeeld van een niet-terminaal element is optelling, dat twee getallen als kinderen bevat. Terminale elementen zijn constanten, functies zonder argumenten of variabelen.

Mutatie in genetisch programmeren gebeurt door een willekeurige knoop te selecteren in een kandidaat oplossing. Deze knoop kan dan worden vervangen door een andere willekeurige knoop of een volledig nieuwe subboom. De crossover operator wordt ook toegepast bij genetisch programmeren. In elk van de twee ouders wordt een willekeurige knoop geselecteerd als crossover punt. De subbomen die beginnen bij deze geselecteerde knopen worden dan uitgewisseld tussen de twee ouders om twee nieuwe kinderen te vormen. Er moet hier worden opgemerkt dat de grootte van de kinderen na een crossover operatie meestal anders is dan de grootte van hun ouders: één kind zal groter zijn en het andere zal kleiner zijn. Er kan nu experimenteel worden vastgesteld dat het grotere kind meer kans heeft om een betere fitness waarde te hebben dan het kleinere kind.

Dit heeft tot gevolg dat de gemiddelde grootte van de elementen van de populatie groeit over verschillende generaties. Vermits deze groei tot problemen kan leiden, wordt in sectie 6 besproken hoe dit effect kan worden tegengegaan.

5.4 Uitbreidingen op genetisch programmeren

Verschillende uitbreidingen op genetisch programmeren zijn mogelijk om de efficiëntie ervan te verbeteren. Enkele belangrijke uitbreidingen zijn:

- **Sterk getypeerd genetisch programmeren:** In het standaard algoritme voor genetisch programmeren staan er geen beperkingen op de manier waarop terminale en niet-terminale elementen met elkaar kunnen worden gecombineerd. Voor complexe zoekproblemen kan het echter nodig zijn om beperkingen in te voeren. Het heeft bijvoorbeeld geen zin om een optelling uit te voeren op twee auto's. Door types toe te kennen aan de terminale en niet-terminale elementen, en beperkingen te leggen op de toegelaten types van kinderen van niet-terminale elementen, kunnen complexe problemen worden voorgesteld.
- **Automatisch gedefinieerde functies:** Sommige problemen kunnen efficiënter worden opgelost door het gebruik van subroutines. Als kandidaat oplossingen subroutines kunnen ontwikkelen tijdens hun evolutie, kunnen deze problemen beter worden opgelost.
- **Voorstelling van de populatie door een gerichte acyclische graaf:** Een populatie bevat een groot aantal deelbomen die gebruikt worden door meerdere elementen van die populatie. Door alle identieke deelbomen in de populatie voor te stellen door éénzelfde object in het geheugen kan de populatie compacter worden voorgesteld. Andere voordelen zijn dat de evaluatie van identieke deelbomen slechts één keer moet gebeuren, en het is eenvoudiger om overeenkomsten tussen verschillende elementen te detecteren.

6 Het probleem van de sterke groei van programma's

In sectie 5.3 werd vermeld dat de gemiddelde grootte van de elementen van een populatie groeit na een crossover operatie. Een voor de hand liggend nadeel hiervan is dat meer geheugen vereist is voor het opslaan van de elementen, en dat de evaluatie ervan langer duurt. Andere nadelen zijn dat grotere oplossingen over het algemeen minder goed generaliseren voor algemenere problemen, hetgeen de evolutie van de populatie sterk kan vertragen.

Als de genetisch geëvolueerde programma's worden onderzocht, blijkt dat grote stukken ervan nauwelijks of geen effect hebben op het eindresultaat. Verschillende theorieën zijn ontwikkeld die verklaren waarom grotere programma's meer kans hebben op een hoge fitnesswaarde:

- **"Lifters" (hitchhikers):** Volgens deze theorie worden stukken code die geen of weinig effect hebben op het resultaat meegenomen met stukken zeer goede code, waardoor de neutrale code zich gemakkelijk kan verspreiden.
- **Verdediging tegen crossover:** Een crossover operatie op een element met een hoge fitness heeft meestal tot gevolg dat een stuk van de goed werkende code wordt verwijderd. Dit heeft tot gevolg dat de fitness waarde van de kinderen drastisch daalt. Als stukken code die een positief effect hebben op de fitness van een element gegroepeerd zijn en gescheiden worden door inactieve code, zal er een grotere kans zijn dat crossover in de inactieve code gebeurt. Bijgevolg zullen de individuele goede stukken code samen blijven, hetgeen een positief effect heeft op de fitness van de kinderen.
- **Voorkeur van verwijdering (removal bias):** Deze verklaring richt zich op stukken code in een element die nooit een effect hebben op het resultaat van dat element. Een crossover die plaats heeft in dit soort code zal geen effect hebben op het individu en dit zal bijgevolg de crossover operatie overleven. Het gevolg hiervan zal zijn dat de gemiddelde fitness van een populatie verhoogt door deze te vullen met functioneel identieke kopieën van het beste element. Wanneer dit gebeurt zal de evolutie van de populatie sterk verminderen en treedt er premature convergentie op.
- **Diffusie:** Het is mogelijk om verschillende programma's te schrijven die eenzelfde probleem even goed oplossen. Omdat, in de oneindige zoekruimte van programma's, er meer grote oplossingen voor een probleem bestaan dan kleine, is het statistisch gezien waarschijnlijker dat een grote oplossing wordt gevonden.

Verschillende manieren bestaan om de sterke groei van programma's af te remmen. Eén van de eenvoudigste manieren is het opleggen van een maximale diepte voor ontwikkelde programma's. Deze manier is echter niet flexibel en verhindert niet de snelle groei in het begin van de evolutie. Een andere manier is om de grootte van programma's op te nemen in de fitness functie, zodat grotere programma's een slechtere fitness waarde krijgen. Het probleem met deze methode is dat het moeilijk is om grootte en kwaliteit ten opzichte van elkaar te balanceren, en het is mogelijk dat er op deze manier optima ontstaan in het fitness landschap waaruit het onmogelijk is te ontsnappen. Een derde manier om de grootte te beperken is door gebruik te maken van een heuvelklim techniek. Na een crossover operatie wordt gekeken of de kinderen beter en/of kleiner zijn dan hun ouders. Als dit het geval is, worden de kinderen toegelaten in de populatie, maar in het andere geval worden ze verwijderd. Dit heeft tot gevolg dat de programma's niet nodeloos groeien.

Vermits een groot deel van een programma bestaat uit inactieve code of code die weinig effect heeft, lijkt een voor de hand liggende manier om de groei van programma's te verminderen om de inactieve gedeeltes van de code te detecteren en te verwijderen. Dit werd geprobeerd door voor verschillende niet-terminale elementen de invloed te meten die hun kinderen hebben op het eindresultaat. Als de invloed van een kind onder een bepaalde drempelwaarde valt, wordt het kind verwijderd van het programma.

Deze techniek werd uitgetest op de problemen van multiplexer en symbolische regressie. In het geval van de multiplexer werd een redelijke vermindering van de grootte van de programma's vastgesteld na het gebruik van deze techniek. In het geval van de symbolische regressie werd er echter nauwelijks verbetering vastgesteld, ondanks het feit dat inactieve code niet meer voorkomt. In deze gevallen werd de inactieve code vervangen door andere vormen van code die een meer geleidelijk effect hebben op het resultaat van het programma. In het geval van de multiplexer was de techniek meer succesvol omdat inactieve code veel gemakkelijker gevormd wordt.

Wegens het beperkte succes van de vorige techniek werd nog een tweede techniek ontwikkeld. Deze techniek legt een maximale grootte op voor nieuwe programma's. Deze maximale grootte wordt echter dynamisch aangepast aan de huidige populatie. De grootte van het beste element van de populatie wordt vermenigvuldigd met een getal (1.33 in dit geval), en deze waarde wordt de maximale grootte van nieuwe elementen. Deze grootte wordt na elke generatie aangepast.

Ondanks de grote eenvoud is deze techniek zeer succesvol. In een experiment werd het gebruik van de dynamische grootte, het verwijderen van inactieve code en het gebruik van de heuvelklim techniek die eerder werd besproken met elkaar vergeleken. Deze technieken kunnen ook met elkaar gecombineerd worden. De experimenten in het domein van symbolische regressie toonden aan dat de dynamische grootte en de heuvelklim techniek de grootte het beste beperkten. Combinatie met het verwijderen van inactieve code had weinig effect op de resultaten. De heuvelklim techniek had echter tot gevolg dat de evolutie zeer sterk vertraagde, waardoor oplossingen veel minder snel werden gevonden. Het dynamisch aanpassen van de grootte had echter een positief effect op de snelheid van evolutie. Er werd dan ook geconcludeerd dat deze techniek, in het bijzonder gezien zijn eenvoud, een zeer goede manier is om de grootte van programma's te beperken.

7 Meten en behouden van de diversiteit van een populatie

Een vaak voorkomend probleem bij evolutionaire algoritmes is dat na verloop van tijd de meeste elementen van een populatie sterk op elkaar lijken. Dit gebeurt wanneer een individu dat fitter is dan de rest van de populatie de ouder wordt van de meeste andere individuen die bijgevolg ook een hoge fitness hebben. Als dit gebeurt is de diversiteit van de populatie verloren gegaan, en kunnen nieuwe elementen enkel worden geïntroduceerd door mutaties. Vaak wordt dit probleem tegengegaan door de populatie zeer groot te maken zodat het langer duurt voor de diversiteit verdwijnt uit de populatie. Het nadeel hiervan is echter dat de evaluatie van de populatie langer duurt.

Andere methodes om de diversiteit van een populatie te behouden kunnen op twee manieren werken. Enerzijds kan erop worden gelet dat elementen die worden toegevoerd aan de populatie voldoende verschillen van de huidige elementen. Anderzijds kunnen elementen die sterk gelijken op andere elementen in de populatie verwijderd worden. Beide manieren moeten op één of andere manier kunnen meten hoe sterk een element verschilt van een ander element of van al de elementen van de populatie.

De manier die wordt geïntroduceerd in deze thesis verwijdert elementen die sterk gelijken op de bestaande elementen van de populatie. Om de overeenkomst van een element met de populatie te meten wordt gebruik gemaakt van de representatie met behulp van de gerichte acyclische graaf, besproken in sectie 5.4. Eerst worden alle elementen van de populatie gerangschikt volgens fitness waarde. Vervolgens worden alle elementen van de populatie geëvalueerd, te beginnen met het beste element. Initieel zijn alle elementen van de gerichte acyclische graaf, die dus de volledige populatie voorstellen, ongemarkeerd. Tijdens de evaluatie van een element wordt het totaal aantal knopen van dat element vergeleken met het aantal gemarkeerde knopen van dat element. Als het element voldoende ongemarkeerde knopen bevat, wordt het element niet verwijderd uit de populatie en worden alle knopen van dat element gemarkeerd. Omdat identieke deelbomen worden gedeeld door alle elementen van de populatie, worden ook deelbomen van andere elementen gemarkeerd. Als het element te veel gemarkeerde knopen bevat wordt het element verwijderd uit de populatie en worden geen knopen gemarkeerd. De test die bepaalt of een element voldoende ongemarkeerde knopen bevat hangt af van de verhouding met het totaal aantal knopen, en met de rang van het element in de populatie.

Experimenteel werd vastgesteld dat deze techniek nuttig was voor het oplossen van problemen. Gemiddeld gezien werd een oplossing voor het probleem sneller gevonden, hetgeen zeer duidelijk was voor het probleem van AI planning dat redelijk veel generaties nodig had om een probleem op te lossen.

8 Evolutionair programmeren toegepast op Robocup

Om te testen of de technieken beschreven in secties 6 en 7 ook werken op een complexer probleem werden de technieken getest op het Robocup domein. Eerst wordt ook nog onderzocht of genetische algoritmes kunnen worden gebruikt om een Robocup speler te ontwikkelen.

8.1 Genetische algoritmes

In een eerste poging om een speler voor Robocup te ontwikkelen wordt gebruik gemaakt van een reactionair netwerk van acties en sensors. Alle spelers van een team maken gebruik van hetzelfde netwerk. De acties stellen laag niveau acties voor van een voetbalspeler, zoals trap naar de goal of pas naar een andere speler. De sensors geven bijvoorbeeld de afstand naar de bal aan. Als een bepaalde actie een voldoende sterk signaal krijgt om geactiveerd te worden, wordt deze actie uitgevoerd. De verbindingen in het netwerk verbinden de sensors, acties en verschillende controleknopen. Elk van deze verbindingen heeft ook een gewicht dat de invloed van een knoop op een andere knoop aangeeft. Het leerproces bestaat erin om geschikte gewichten voor deze verbindingen te vinden. Genetische algoritmes lijken een geschikte manier om deze gewichten te leren.

De evaluatie van een set gewichten voor een speler kan op twee manieren gebeuren. De eerste manier maakt gebruik van een vast referentieteam van spelers, en elke kandidaat oplossing speelt een wedstrijd tegen dit referentieteam. Het resultaat van die wedstrijd bepaalt de fitnesswaarde van een kandidaat oplossing. De tweede manier laat alle kandidaat oplossingen een toernooi spelen tegen elkaar. In dit geval is er geen referentieteam nodig, en bepaald de positie van een team op het einde van een toernooi de fitness waarde van dat team. In dit geval spreekt men van co-evolutie.

Een vergelijking leert dat het gebruik van een vast referentieteam leidt tot een sneller beter worden van de ontwikkelde teams, maar de ontwikkelde teams zijn specifiek ontworpen om het vaste referentie team te verslaan en kunnen minder goed veralgemenen tegen andere teams. In het geval van co-evolutie werd vastgesteld dat evolutie bijzonder traag was, en dat nauwelijks verbetering was vast te stellen. Mogelijk is de vaste netwerkstructuur die gebruikt wordt niet flexibel genoeg om gemakkelijk tot grote verbeteringen te leiden.

8.2 Genetisch programmeren

Om de flexibiliteit van het leerproces te verhogen, en om de ontwikkelde technieken van secties 6 en 7 uit te testen, werd gebruik gemaakt van genetisch programmeren om een Robocup speler te ontwerpen. De implementatie van de gebruikte primitieven is gebaseerd op het werk van Sean Luke. Na zijn werk over genetisch geprogrammeerde Robocupspelers beschreef hij de volgende problemen die zijn aanpak had:

- De grootte van de populatie moest erg klein worden gehouden om de evaluatietijd beperkt te houden.
- De evaluatie van alle teams hangt af van een enkele wedstrijd van elk team. Dit kan leiden tot een grote willekeurigheid van de fitnesswaarde van een team.
- Bij de evolutie werden teams van identieke spelers ontwikkeld in plaats van een team van verschillende gespecialiseerde spelers.
- De gebruikte verzameling primitieven was sterk bevooroordeeld naar het gedrag van menselijk voetbal. Er werd ook geen gebruik gemaakt van een interne staat om lange-termijn-denken mogelijk te maken.
- De grootte van de ontwikkelde programma's werd na ongeveer 40 generaties bijzonder groot.

Deze problemen worden door ons op de volgende manier aangepakt:

- Door het gebruik van de diversiteitsmaat besproken in sectie 7 kan de grootte van een populatie beperkt worden gehouden zonder de diversiteit ervan aan te tasten. Bijgevolg is er geen probleem meer om de grootte van de populatie klein te houden.
- De evaluatie van de verschillende teams kan gebeuren met behulp van een "Swiss" toernooi systeem. In dit systeem speelt elk team een vast aantal wedstrijden, en wordt getracht om elk team tegen een ander team te laten spelen dat gelijkaardig presteert. De rankschikking van de teams na het toernooi bepaalt dan de fitnesswaardes van de teams.
- De spelers van elk team kregen een rol toegewezen (doelman, verdediger, middenvelder, aanvaller) en de verzameling van primitieven werd uitgebreid met een functie om deze rol op te vragen. Bijgevolg is het mogelijk om één enkel programma te ontwikkelen dat gespecialiseerd is voor verschillende spelers.
- Omdat de gebruikte verzameling primitieven gebaseerd is op deze van Sean Luke is deze nog steeds bevooroordeeld. Er wordt wel gebruik gemaakt van

een beperkte interne staat waardoor het mogelijk is om enkele primitieve acties in opeenvolgende tijdstappen uit te voeren.
- De techniek besproken in sectie 6 beperkt de groei van programma's.

Om de effecten van de technieken van secties 6 en 7 te onderzoeken werden de volgende experimenten uitgevoerd:

- Evolutie over 131 generaties waarbij zowel het dynamisch beperken van de grootte als het behouden van de diversiteit werden toegepast.
- Evolutie over 121 generaties waarbij enkel het behouden van de diversiteit werd toegepast.
- Een tweede evolutie over 119 generaties waarbij zowel dynamisch beperken van de grootte als het behoud van de diversiteit werden toegepast.
- Evolutie over 71 generaties waarbij enkel het dynamisch beperken van de grootte werd toegepast.

In deze experimenten wordt de gemiddelde groei van de individuen van de populatie beschouwd, de evolutie van de kwaliteit van oplossingen en de evolutie van de specialisatie van de spelers. Er moet echter wel opgemerkt worden dat het moeilijk is om algemene conclusies uit deze resultaten te trekken wegens het beperkte aantal experimenten.

De evolutie van de fitness van een experiment kan worden bestudeerd door de beste individuen uit elke generatie met elkaar te vergelijken met behulp van een toernooi. In alle vier experimenten bleek dat deze vergelijking bijzonder veel ruis bevat. Gemiddeld blijkt de fitness wel te stijgen over het verloop van verschillende generaties.

In de drie experimenten waar de grootte van nieuwe individuen dynamisch werd beperkt bleef de gemiddelde grootte van individuen beperkt. In het andere experiment daarentegen groeide de gemiddelde grootte snel.

De specialisatie van spelers kan worden onderzocht door de individuen te bekijken die het "MyType" primitief bevatten. Deze individuen bevatten code die specifiek is voor een bepaald soort speler. In enkele van de experimenten werd een succesvolle specialisatie ontdekt. In dit geval werd deze succesvolle code ook verder verspreid over de populatie. In het geval waar de diversiteit van de populatie niet werd onderzocht werd een succesvolle specialisatie echter zo dominant dat drie kwart van alle individuen in een populatie deze specialisatie bevatten. In dit geval werd de diversiteit van de populatie verlaagd, en het vinden van goede en originele oplossingen werd moeilijker.

De resultaten van de vier verschillende experimenten kunnen met elkaar worden vergeleken door alle teams van de experimenten samen een toernooi te laten spelen. Het resultaat van dit toernooi bevat opnieuw zeer veel ruis, maar het eerste experiment lijkt hier toch de beste resultaten te geven. In dit geval blijft de gemiddelde fitness groeien tot het einde van het experiment. In de andere experimenten wordt ook een groei vastgesteld, maar minder sterk. In het geval van het derde experiment is dit vooral door de relatief hoge fitness in het begin van het experiment. De gebruikte optimalisaties lijken een positief effect te hebben op de gemiddelde fitness van de populatie.

9 Conclusies en bespreking

Het werk over virtuele agenten in deze thesis kan worden opgesplitst in twee delen. In het eerste deel werd het deel van navigatie en het ontwijken van obstakels behandeld. Om dit probleem op te lossen, maakt de virtuele agent gebruik van een virtuele optische sensor die de afstanden tot geobserveerde objecten detecteert voor de agent. De agent maakt vervolgens gebruik van deze afstandsinformatie om een kaart te construeren van zijn omgeving door om zijn as te roteren. Om een kaart van de volledige omgeving te construeren beweegt de agent door openingen in het huidige gebied, waarna het proces wordt herhaald voor het volgende gebied. Dit gebeurt tot de volledige omgeving bezocht is. De geconstrueerde kaart kan vervolgens worden gebruikt voor het vinden van paden in de omgeving, en kan worden uitgewisseld met andere agenten of gebruikers. De afstandsinformatie over obstakels rond de agent wordt ook gebruikt om botsingen te voorkomen tijdens het bewegen in de omgeving. Als een beweging in een richting zou leiden tot een botsing met het gedetecteerde object wordt het bewegingscommando aangepast om deze botsing te vermijden. Dit kan gebeuren door rond het obstakel te bewegen indien mogelijk, of door de agent te stoppen.

Er zijn verschillende voordelen verbonden aan het gebruik van een virtuele sensor om de omgeving te detecteren vergeleken met het gebruik van de interne representatie van de virtuele omgeving. Ten eerste is een virtuele sensor een meer realistische simulatie van de reële wereld, omdat mobiele robots ook dikwijls gebruik maken van dieptesensors voor navigatie. Een tweede voordeel is dat het op deze manier niet nodig is dat de virtuele agent toegang heeft tot de interne representatie van de omgeving. Wanneer enkel een visualisatie van de omgeving nodig is kan de virtuele agent in elke omgeving werken waar zulk een beeld beschikbaar is.

Het tweede deel van de thesis behandelt het trainen van virtuele agenten om een bepaalde taak uit te voeren. In deze context werd het gebruik van genetisch programmeren beschouwd. Twee belangrijke problemen werden aangetroffen bij het evolueren van virtuele agenten met genetisch programmeren. Ten eerste heeft de grootte van de geëvolueerde genetische programma's de neiging om zeer snel te groeien. Ten tweede, omdat de evaluatie van geëvolueerde virtuele agenten meestal veel tijd vergt, moet het aantal evaluaties zo laag mogelijk

worden gehouden. Helaas heeft genetisch programmeren meestal een grote populatie van kandidaat oplossingen nodig om goede resultaten te ontdekken.

Verschillende technieken werden ontwikkeld om deze problemen op te lossen. Eerst werden twee nieuwe methodes ontwikkeld om de grootte van geëvolueerde programma's te verminderen, en deze werden vergeleken met bestaande technieken. De eerste methode detecteert en verwijdert inactieve code uit de genetische programma's. De tweede methode legt een dynamisch veranderende maximale grootte op aan nieuwe individuen. Deze grootte hangt af van de grootte van het beste element van de huidige populatie. De tweede methode bleek beter te zijn dan de bestaande methodes, en het is zelfs mogelijk om beide methodes samen te gebruiken.

Het tweede probleem werd opgelost door een algoritme te ontwerpen dat het aantal individuen in een populatie verlaagt, zonder de diversiteit van de populatie te verminderen. Het algoritme werkt door identieke deelbomen in de individuen van de populatie te detecteren, en vervolgens die individuen te verwijderen die hoofdzakelijk bestaan uit deelbomen die ook elders in de populatie voorkomen. Er werd ook aangetoond dat het verwijderen van deze individuen uit de populatie de convergentiesnelheid verbetert van de evolutie.

Tenslotte werden deze verbeteringen op het genetisch programmeersysteem toegepast op het virtuele multi-agent systeem van robotvoetbal. De verbeteringen lijken een positief effect te hebben op de evolutie. Wegens de zeer lange evaluatietijd in dit domein was het echter niet mogelijk om een voldoende aantal verschillende testen uit te voeren om doorslaggevend te zijn.