NTNU – Trondheim
Norwegian University of
Science and Technology

Multi-Objective Neuroevolution in Super Mario Bros.

Ole-Petter Olsen
Lars Solvoll Tønder

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Pauline Haddow, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Multi-Objective Neuroevolution in Super Mario Bros.

TDT4900 - Computer and Information Science, Master Thesis

Artificial Intelligence Group
Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering

Lars Solvoll Tønder & Ole-Petter Olsen

Supervised by **Pauline Haddow**

June 9, 2013

Abstract

This thesis explores how to use Multi-Objective Evolutionary Algorithms (MOEA) to solve problems that are not explicitly defined as multi-objective problems. A neuroevolution technique consisting of combining a multi-objective evolutionary algorithm called NSGA-II and artificial neural networks (ANN) based on NeuroEvolution of Augmented Topoligies (NEAT) were used to develop a system that created controllers for a version of the Super Mario Bros game called Mario AI. Experiments were conducted to measure different ways to define objectives for MOEAs in Mario AI, how using these objectives as a basis for a scalar fitness function would affect a genetic algorithm and to examine how to use ensembles to combine individuals of a pareto front into a single controller that would be able to display the strengths of all of the individual controllers.

The results show that adding sub-goals as objectives together with the main goal could have a positive effect for a MOEA and that the same sub-goals could also give a positive effect when applied to the scalar fitness of a genetic algorithm. It is however not trivial to decide which sub-goals to use, as most of the chosen objectives were found to have a negative impact on the controllers, even when selected based on the authors' expert knowledge about the game domain. Using basic behaviours that the controller has to use in order to play well as objectives had a negative effect on the controllers and the controllers were able to learn these behaviors even without using them as objectives.

Preface

This report concludes the research and findings done in the authors' master thesis at NTNU(Norwegian University of Science and Technology). We would like to thank our superviso Pauline Haddow for her patience and invaluable feedback throughout the project. We would also like to thank the other members of the lab at Gemini-centre for applied artificial intelligence for all of the interesting discussions on several subjects related to our thesis and the field in general.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives an introduction to the master thesis by presenting some background information and some motivation for doing this work. It also states the goals and research questions that this thesis attempts to answer.

1.1 Background and Motivation

This thesis is largely based on the work done by the authors in their specialization project during autum 2012. The project explored usage of evolutionary algorithms for creating auomated controllers for video games. It was found that there were several advantages of using evolutionary algorithms, such as being able to create controllers with dynamic behaviors without much need for expert knowledge about the game.

Multi-objective evolutionary algorithms were found to be popular because it had been shown to create controllers that do well at multiple objectives simultaneously and could create controllers that took advantage of tradeoffs between objectives. This technique was used by the authors during the project for creating controllers for a video game and results showed that it could perform well at playing the game at the same time as it was able to display intelligent behavior.

Most examples of problems where multi-objective evolutionary algorithms are used have explicitly defined objectives that have to be optimized. These objectives are usually conflicting, so that the MOEA can find a tradeoff between the objectives. In some cases there is no such obvious objectives when analyzing a problem. This was the case when the authors designed the controllers for the game. In these cases it can be very challenging to define which objectives to use in the algorithm.

This thesis explores how MOEAs can be used to solve problems that are not explicitly defined as multi-objective problems with obvious contradicting objectives by using MOEAs together with various other techniques to create controllers for a video game with no clear division between objectives. The goal is not to create as good controllers as possible, but to compare how different approaches affect the performance of solving the problem.

1.2 Goals and Research Questions

The goals and research questions of this thesis can be summarized as follows:

- **Goal** Explore how using MOEAs can be used to solve problems that are not explicitly defined as multi-objective problems with obviously contradicting objectives.
- **Research Question One** How to define objectives for a MOEA in a problem without explicitly defined objectives?
- **Research Question Two** How does incorporating objectives into a scalar fitness function affect a GA, compared to rewarding objectives separately using a MOEA.
- **Research Question Three** How can ensembles be used to combine individuals of a pareto front into a controller that is able to perform well at all the objectives of the front?

1.3 Research Method

In order to achieve the goal of this thesis and answer the research questions, a video game was used as test bed for various techniques and approaches suggested in the literature and by the authors. A system was designed and implemented that used these approaches and experiments were conducted in order to measure and compare how these approaches and techniques affected the performance in the game.

1.4 Report Structure

The required background theory for understanding the approaches used in this thesis is explained in Chapter 2. The motivation behind the approaches and its rationale is discussed in Chapter 3. Chapter 4 describes the details of the system that was designed and implemented for the experiments. The experimental plan and setup is explained in Chapter 5 and the results of these are

presented in Chapter 6. The evaluation of these results are described in Chapter 7, together with the conclusion of this thesis. Some suggestions for further work are described in Chapter 8.

Chapter 2

Background Theory

The system developed in this thesis is based on several techniques from biologically inspired artificial intelligence such as evolutionary algorithms and artificial neural networks and this chapter gives an introduction to how this technology works. This chapter also gives an introduction to the game used as test bed for the experiments.

2.1 Evolutionary Algorithms

An evolutionary algorithm (EA) is a population-based meta-heuristic optimization algorithm that uses mechanics borrowed from biological evolution to solve a predefined problem. There are several types of EAs such as gentic algorithms (GA), genetic programming (GP) and evolutionary strategies (ES) and they have in common that they use most of the same components in the artificial evolutionary process which can be seen in Figure 2.1.

The first step is to create an initial population. A population consists of individuals and each individual is defined by their genotype. The representation of the genotype can vary depending on the problem they are used to solve and the type of EA used. Some common representation types are binary strings (used in GA) and tree structures (used in GP). The individuals in the initial population are usually randomly generated, but can also be predefined if a previous good solution to the problem is known.

The next step is to create phenotypes from the genotypes in the population. The phenotype is a usually a higher level representation of the individuals, but they can also be the same as the genotype. The phenotypes are used as solutions to the problem the algorithm is trying to solve. The mapping between genotypes and phenotypes can vary depending on which type of EA is used and the complexity of the solutions to the problem.

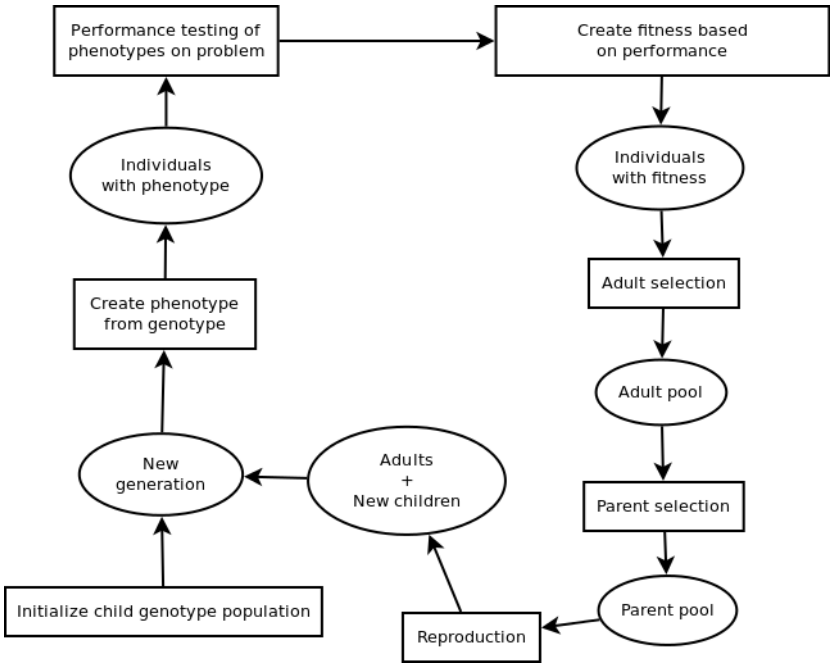

Figure 2.1: Flow of individuals (ovals) through an evolutionary algorithm. Individuals begin with a genotype that maps to a phenotype which then obtains a fitness value after performance testing. The selection of both adults and parents are then based on the fitness of the individuals and genetic operators are applied to the selected parents to create new individuals.

The next step is to evaluate the fitness of the phenotypes. This is done in two steps. First the phenotype is applied to the problem the EA is trying to solve and then the results of its performance is used to calculate a numerical score which is its fitness value. Depending on how the fitness value is calculated, the individuals with either the highest or lowest fitness value will be considered the best individuals. The fitness values are used in the selection process in the next steps.

Selection is done in two steps. The first step is adult selection where the newly evaluated individuals compete for becoming adults. Only the fitness score is used for this competition and the criteria to win can vary between being among the best individuals in a local group of randomly selected individuals or among the best individuals globally.

After this process is completed the adult population competes for membership of the mating pool. The winners of this selection will mate and create new offspring for the next generation. This selection mechanism can also be local or global and fitness may be scaled before the selection begins. When the parents

have been selected the reproduction process begins.

In the reproduction process, parents copy their genotypes and use these to create new individuals. If crossover is used, two parents combine their genotypes according to a predefined rule and if not, the genotype is just copied to the child. Before the child is created a mutation operator can be applied to its genotype where random changes are made to some of its components.

Mutation and crossover is called genetic operators. Mutation is done on a single gene which is either a copy of a single parent or the resulting gene of a crossover operation. The mutation prodecure makes a random change to one or more components of the genotype. If the genotype is a bit-string then the mutation could be to flip some of the bits.

Crossover is done by taking the genotype from two or more parents and combining these into one or more new individuals. A common way to do this is 1-point and 2-point crossover. This is done by swapping parts of the genotype of the parents, as shown in Figure 2.2. The crossover operation becomes more complicated for more complex genotype representations.

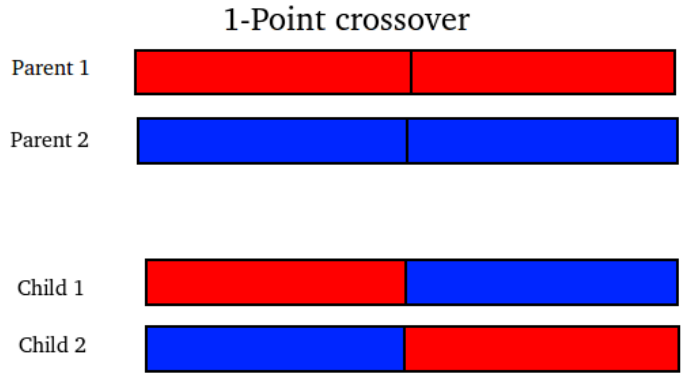

Figure 2.2: 1-Point crossover operation on two genotypes. If the genotype is a bit string then the first child consists of the bits before the crossover point from the first parent followed by the bits after the crossover point from the second parent and reversed for the second child.

After the new individuals has been created the cycle repeats with the next generation. This cycle repeats until a maximum number of generations have been created or another criteria, such as whether the best individual of the current generation has a fitness over a certain threshold, has been met. When the last generation is completed the individual with the highest fitness is presented as the solution to the problem.

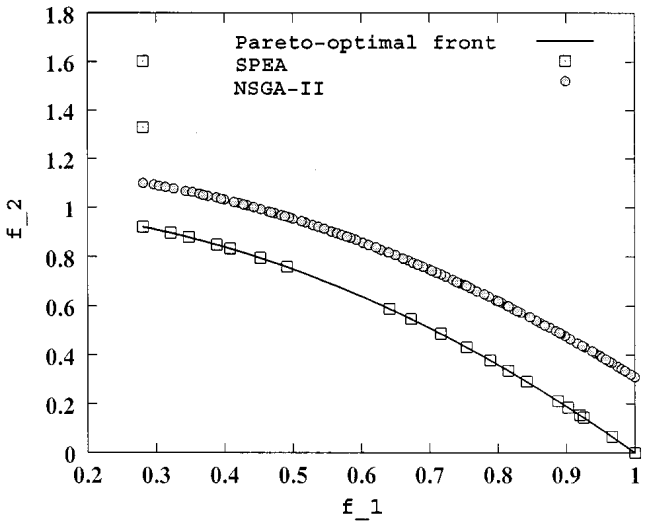

Figure 2.3: True pareto front and approximated pareto front by two different multi objective evolutionary algorithms (Deb et al. [2]).

2.2 Multi-Objective Optimization

Multi-objective optimization is the problem of finding a solution to a problem with several objectives where all objectives are optimized simultaneously. An example of a multi-objective problem could be trying to maximize profits while minimizing risk when it comes to stock trading. Objectives like these are often contradictory in that maximizing the results of one objective will hurt the results of the other. There is therefore no single optimal solution to the problem, but instead an array of valid nondominated solutions with different tradeoffs that constitutes the pareto front.

The pareto front consists of solutions that are pareto optimal, or nondominated. A solution is nondominated if there is no other solution that improves any of the objectives without impairing the result of another objective. Without any additional information about the preference of the different objective functions, either of the solutions in the pareto front can be considered optimal. Multi-objective optimization can therefore be understood as approximating the pareto front. A true pareto front with approximated solutions can be seen in Figure 2.3. The true pareto front shows all the solutions for minimizing both of the objectives. The circles and squares show the solutions obtained by approximating the pareto front by using two different multiobjective evolutionary algorithms.

Evolutionary algorithms have become a popular way of approximating the pareto front, much due to the fact that EAs already generate a population of differ-

ent solutions that can be used to approximate several points of the true pareto front in a single simulation. Evolutionary algorithms can be converted to solve multi-objective problems by replacing the fitness calculation with a two-step approach. This leads to a multi-objective evolutionary algorithm (MOEA) and the main loop of a MOEA with this new approach can be seen in Figure 2.4 with the new steps colored.

The first step is to calculate objective values for each individual by running performance testing of phenotypes on the problem they are going to solve. The second step is to calculate fitness. In a regular EA, the fitness would be calculated by a function of the objective values, but in a MOEA the fitness is calculated using a pareto based ranking scheme to sort the individuals according to their non-dominated count. Each group of individuals with the same ranking becomes a front, where the individuals who are not dominated by any other solution construct the pareto front. Individuals are then given a ranking based on which front they inhabit, often accompanied by a secondary ranking strategy. A common secondary ranking strategy is to give a worse ranking to individuals that are close in the objective function space, enforcing a wider spread of solutions and an approximation that covers a large part of the true pareto front instead of solutions being focused in a narrow area. The fitness is finally calculated based on these rankings where the fitness of an individual with a lower non-domination count always has a higher fitness than one from another front. Selection is then performed as usual and the rest of the MOEA loop is similar to that of an EA.

2.2.1 Non-Dominated Sorting Genetic Algorithm II

The Nondominated Sorting Genetic Algorithm-II (NSGA-II) (Deb et al. [2]) is a multi-objective evolutionary algorithm created to solve multi-objective optimization problems using the notion of pareto dominance and crowding distance.

The algorithm starts by creating an initial random population of size N that is sorted based on nondominance. This means that each individual is given a ranking based on how many individuals it is strictly dominated by, such that the individuals that are not dominated by any other gets highest rank. Binary tournament selection is then used to select parents and crossover and mutation operators are applied to these parents to produce a new offspring population of the same size as the old population.

For each generation, the parent and child populations are combined into a single population that is used for creating the next generation. First, rank is assigned to each individual based on nondominance, such that the individuals gets sorted into different fronts where the 0th front is the best individuals. Then the crowding distance is calculated for each individual and the individuals get sorted within their front based on their crowding distance. This means that

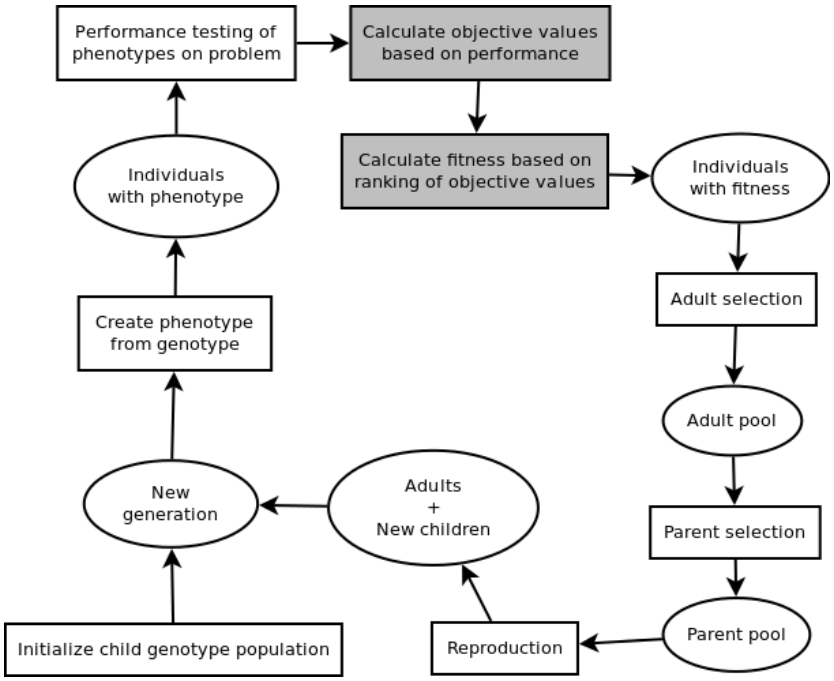

Figure 2.4: The general flow of a Multi-Objective Evolutionary Algorithm. The grey boxes shows the difference between a MOEA and the regular EA in Figure 2.1.

an individual is considered better than another if it is either in a lower front than the other individual or in the same front with a higher crowding distance. This two-step sorting process can be seen in Figure 2.5. The crowding distance of an individual is calculated as the sum the cubic distance of the objective values between the neighbouring solutions of the individual. This means that individuals in a sparsely crowded area in the solution space are preferred over solutions in crowded areas. At the end of each generation the N best indivuals are selected as parents and a new child population is created similar to in the initial population. These two populations are then combined and used for the next generation.

After the final generation is completed, the individuals with a nondomination count of zero are presented as an approximation to the true pareto front of the problem.

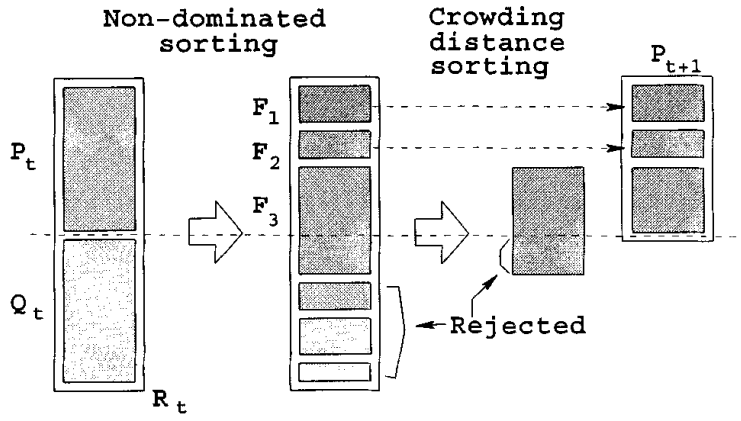

Figure 2.5: Flow diagram with the sorting procedure in NSGA-II. Pt is the parent population and Qt is the offspring population at generation t. F1, F2 and F3 are the three best fronts after nondominated sorting. Since there are too many individuals in F3 the front is sorted based on crowding distance, and the best individuals in F3 are added to the final population together with all individuals in F1 and F2 (Deb et al. [2]).

2.3 Artificial Neural Networks

Artificial neural networks (ANNs) are computational models that attempt to capture the behavior and adaptive features of biological neural networks. These models are implemented in software or hardware and are used to approximate some complex function. ANNs are often used in order to solve problems where it is difficult or impossible to create an analytical solution.

An ANN consists of individual computational units called neurons that are interconnected by weighted connections. These neurons are usually divided into layers and the number of layers depend on the architecture of the network. The layers are usually categorized as input layer, output layer and hidden layer. The neurons in the input layer receive information from the environment, the neurons in the output layer emits signals back to the environment and all internal neurons are said to be in a hidden layer. Most networks have one or more hidden layers, but it is also possible to not have any hidden layers. A generic ANN architecture can be seen in Figure 2.6. This network has five input neurons, one hidden layer with three hidden neurons and two output neurons. This network can be considered as a function that takes five parameters as input from the environment and returns two values back to the environment. If the network was used for something like controlling a robot, the input values could be values from proximity sensors and the output could be values for the speed of the left and right wheels of the robot.

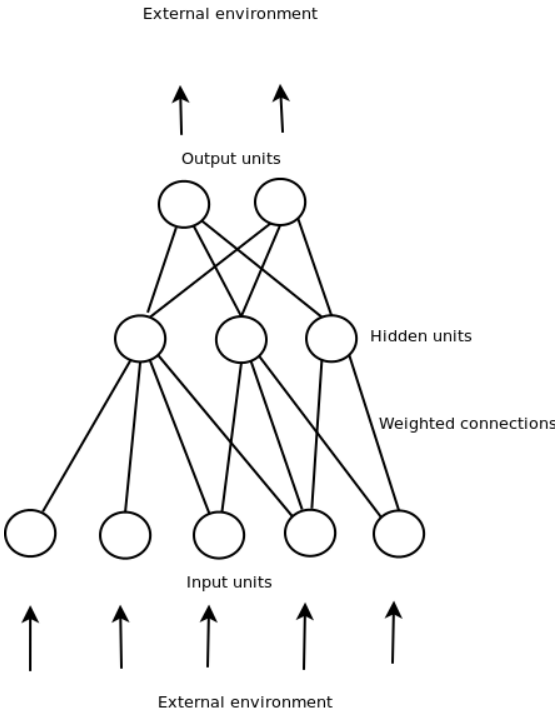

Figure 2.6: The architecture of a generic ANN with three layers. The circles represent neurons and the lines represent weighted connections.

Each neuron has input connections, output connections and an activation function. The activation function is a mathematical function such as a linear function, step function or sigmoid function. Each neuron use the net input, which is defined as the sum of all its input values, as input to the activation function. The output signal from the neuron can be either discrete or continous. In the discrete case, the output is either 0 or 1 depending on whether the output of the activation function is above a certain threshold or not. In the continous case, the output signal of the neuron is simply the output of the activation function used on the net input. The neurons emits an output signal to all other neurons it has outgoing connections to. The receiving neuron receives an input signal with a value equal to the output signal of the emitting neuron multiplied by the connection weight.

The architecture of a neural network affects its functionality. The most common architectures are feedforward neural networks and recurrent neural networks. Feedforward neural networks are divided into layers where each layer only emits signals to the next layer. These type of networks are also called multi-level perceptrons and are the simplest type of networks. Feedforward neural networks cannot detect or produce temporal sequences unless they are composed of dynamic neurons. Recurrent neural networks have connections between neurons in the same layer and between neurons in upper layers back to neurons in lower layers. These networks can detect and produce temporal sequences even with static neurons, since a neuron can get input from the previous time step from a recurrent connection.

The response from the neural network on the environment when it receives input depends on the activation functions of the neurons, the connections between the neurons and the weights of the connections. The connections can be modifying by using a learning method such as backpropagation or by using an evolutionary algorithm.

Most learning methods are done using examples and can be divided into supervised, unsupervised and reinforcement learning. Supervised learning, such as backpropagation, is based on comparing the actual output of the ANN with a desired output, given a predefined set of input values. In backpropagation, the difference between the two output values are considered the error value and this error is propagated back through the network and the weights are changed based on the learning rule used. Reinforcement learning is a special case of supervised learning where it is only known whether the output is correct or not. Unsupervised learning is based on correlations between input values, with no knowledge about correct output values available.

2.4 Neuroevolution

Neuroevolution is the use of evolutionary algorithms applied to artificial neural networks. Evolutionary algorithms can be used to change the behavior of ANNs by evolving connection weights, network architecture, learning rules and activation functions (Yao [11]).

The most common way of evolving neural networks is to evolve the connection weights of the ANN. This can be done by encoding the connection weight values into the genotype of the individuals of the population and give fitness to individuals according to the performance of their ANN phenotype. Mutation and crossover is used to create networks with new connection weights. This technique usually use fixed architecture and activation functions.

Another way to evolve ANNs is to evolve the architecture. This enables the ANNs to adapt their topologies to the task they are trying to solve without the need of the design of a human. When evolving architectures of ANNs it is common to only encode some of the characteristics of the network, such as number of nodes, probability of connections between them and type of activation function instead of the connection weights. This is known as indirect encoding of the network, since it is not a one to one mapping between the genotype and phenotype of the individuals and gives the evolution more freedom to find the optimal topology of the network.

It is also possible to evolve learning rules and activation functions. Evolution of different parts of ANNs, such as connection weights and architectures, can also be combined together.

2.4.1 Neuroevolution of Augmented Topologies

NeuroEvolution of Augmenting Topologies (NEAT) (Stanley and Miikkulainen [9]) is a genetic algorithm for evolving both weights and architectures of artificial neural networks. NEAT solves some of the difficulties with evolving neural networks, such as competing conventions and having to specify the architecture of the network before evolution.

Competing conventions is when individuals with different genotypes can result in ANN phenotypes that are behaviorally equivalent. This makes evolutionary seach more difficult because the genotypes create seperate hills on the fitness landscape and it also makes crossover more difficult because the two genotypes can disrupt the positive traits of each other, resulting in offspring with low fitness.

The genetic encoding of a NEAT individual can be seen in Figure 2.7. The genes are separated into node genes and connection genes. Each node gene defines a node in the network by a unique number and type (input, hidden or output). The connection genes define the connections between the nodes

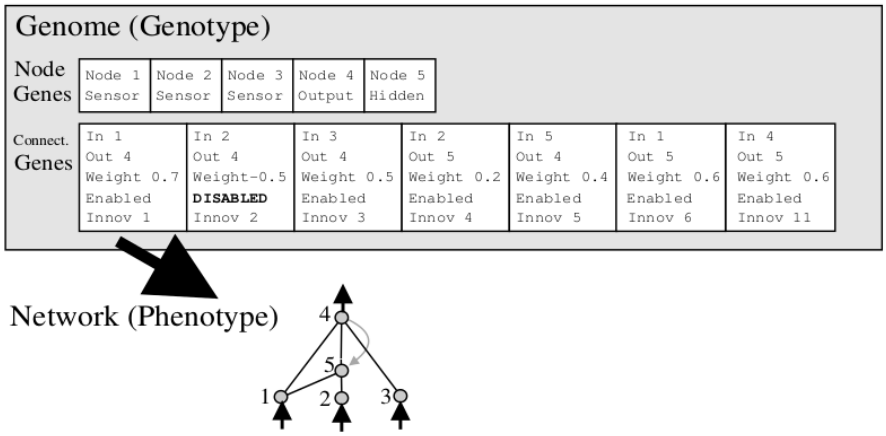

Figure 2.7: Genotype of NEAT individuals. The genotype is divided into node genes and connection genes. The node genes defines the nodes in the network and the connection genes defines the connections between them. Connection genes can be disabled and have a innovation number in addition to the usual connection weight. (Stanley and Miikkulainen [9])

and each gene contains information about which node the connection goes in and out of as well as the weight of the connection. In order to make mating easier, the connection genes also contains an enable bit that specifies whether the connection is expressed in the network or not and a innovation number. If a connection gene is not expressed, it is not included in the ANN of the phenotype.

NEAT uses three different mutation operators which is mutation of connection weights, adding new connections and adding new nodes. The last two can be seen in Figure 2.8 and these types of mutation allows evolution to find the best architecture for the network itself, instead of having to specify the architecture beforehand, which is often the case when evolving ANNs. Mutation of connection weights is simply to change the weight value of a connection gene. Adding a new connection consist of creating a new connection gene and adding it to the genotype, as seen in the upper half of the figure. Adding a new node is done as part of splitting a connection. This is done by adding a new node gene to the genotype, disabling a connection gene, then finally adding two new connection genes, one from the input of the disabled connection gene to the new gene and one from the new gene to the output of the disabled connection gene, as shown in the lower half of the figure.

NEAT starts with a uniform initial population where the individuals have very simple architectures with no hidden nodes. Through mutation, the genotypes will gradually become larger, causing genomes of varying size to appear with different connections at the same positions. In order to perform meaningful

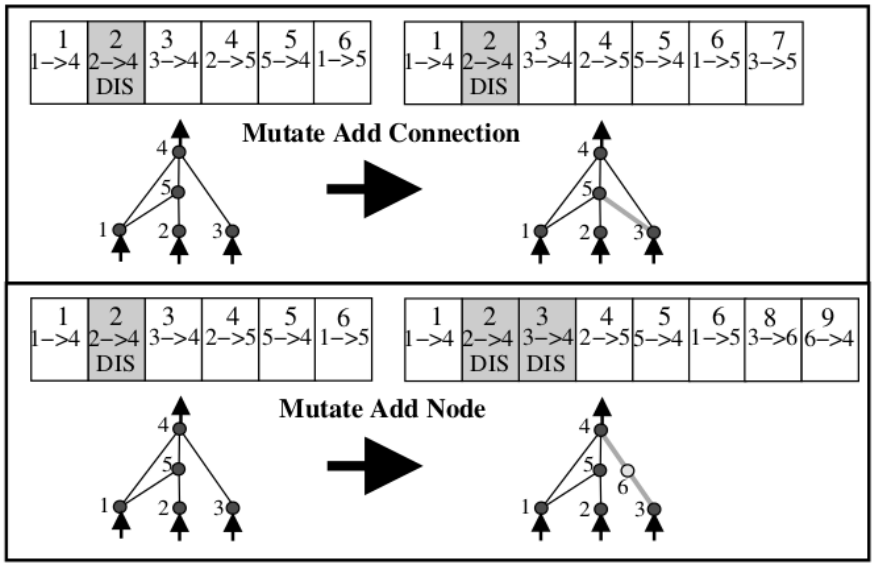

Figure 2.8: Two of the mutation operators in NEAT. The top half shows how adding a new connection gene to the genotype (7) adds a new connection to the phenotype network between node 3 and 5. The lower half shows how two new connection genes (8 and 9) replaces the old disabled gene (3) when adding a new node (node 6). (Stanley and Miikkulainen [9])

crossover between such different individuals, NEAT uses the notion of historical markers called innovation numbers. Whenever a new connection gene appears, a global innovation number is incremented and assigned to that gene. The structure of that gene is then stored in a global list of innovations and each new gene that matches one already placed in the innovation list is assigned the same innovation number as the one found in the list. When crossover is performed between two parents, the connection genes in both genomes with the same innovation numbers are lined up, and a new offspring is created containing genes that are randomly chosen from either parent at matching genes and all of the other genes from the most fit parent.

In order to protect innovation of new topologies in NEAT, a mechanism called speciation is used to make individuals compete primarily within their own niches instead of with the global population. This protects topological innovations, which might have low fitness at first, by allowing them to optimize their structure within a niche.

Speciation is implemented using the innovation numbers of the individuals to compute a distance between individuals such that individuals that find themselves over a certain threshold away from eachother are defined as belonging to a different species and thus not able to mate. Each existing species is represented by a random genome inside the species from the previous generation, preventing overlapping species within a population. NEAT also uses explicit fitness sharing where all individuals of a species must share the fitness of their niche. This has the effect of limiting the size of any given species so that it does not take over the entire population.

2.5 Mario AI

Mario AI [8] is an open source version of Nintendo's classic platformer Super Mario Bros. based on the Infinite Mario Bros implementation by Markus Persson. It is created as a framework to allow AI researchers to benchmark AI methods and algorithms in a game that is easy to learn, hard to master, visually pleasing and can be played by both AI and players.

Mario AI is a clone of Super Mario Bros., which is a platform game where the player has to control a character through side-scrolling 2-dimensional levels. For each level, the player starts at the left side of the level and the goal of the game is to get to the exit at the right side. The player controls a character that can move left and right as well as run and jump. The player need to navigate Mario through each level which is filled with platforms, pitfalls, blocks and enemies with different skills and properties. The player loses the game if Mario is hit by enemies enough times to get killed, falls into a pit or runs out of time before reaching the exit.

The player can collect power-ups that changes the attributes of Mario. The

mushroom power-up allows Mario to take one extra hit from enemies before getting killed and enables him to break regular blocks by jumping under them. The fire-flower power-up gives Mario the ability to shoot fireballs that can kill most enemies if they get hit. Getting hit by an enemy will revert Mario to the previous state before the last power-up or cause Mario to die and lose one life if he has no more power-ups.

The game gives the player a score based on how well the game is played. Points are awarded for picking up power-ups and coins, killing enemies by jumping on top of them or shooting them with a fireball and reaching the goal of the level in the shortest amount of time. Each level of the game has a timer which shows the remaining time before the player automatically dies on the current stage and points are awarded based on how much time is remaining when reaching the exit. An example of a level from the original Super Mario Bros. can be seen in Figure 2.9. The level consists mostly of platforms and pitfalls. There are enemies on the level as well, but they are not shown in the figure.

Figure 2.9: Level 5-3 in Nintendo's Super Mario Bros. The player starts at the door on the left side of the level and has to get to the exit, represented by the flagpole at the right side of the level.

In Mario AI, the player has to control Mario with the use of 2 buttons for running and jumping and 4 buttons for moving in each direction. Mario AI runs at 24 frames per second and the AI controller for the Mario AI framework needs to give an output at each timestep for whether or not each of these buttons or directions are pressed, yielding a combination of $2^6 = 64$ different actions that Mario can perform at any given time. However some of these combinations are nonsensical as pressing both left and right, or up and down at the same time which will cancel each other out.

The Mario AI API gives the developers the ability to automatically generate new levels through the use of different parameters such as the difficulty of the level, type and length. It is possible to generate the same levels each time by providing the same random seed each time levels are generated. Levels with higher difficulties include more and tougher enemies than easier levels. It is also possible to set whether or not the level should have pits or whether or not the level should have enemies or obstacles and even if enemies can move or not.

The AI playing the game is presented with information about the game environment at each time frame. Information about the environment and game state can be extracted in several ways such as parameters about the state of Mario and number of enemies on the screen and through a receptive field. A screenshot from the game with the receptive field shown can be seen in Figure 2.10. The

receptive field is a matrix of values based on what each cell contains. In the field shown in the figure (the grid overlay), most of the cells are empty and contains a zero value, but the cells with enemies, mario, item blocks, terrain, etc. all have a nonzero value depending on what they contain. Mario is centered in the middle of the receptive field and the default resolution of the receptive field is 19 * 19 blocks. The figure also show information about the level and game state such as level difficulty, length of the level (Mario's current position plus total length), number of coins collected, number of enemies killed and remaining time. The figure also show information about intermediate reward, which is the score Mario AI gives the AI or player for the performance at playing the game.

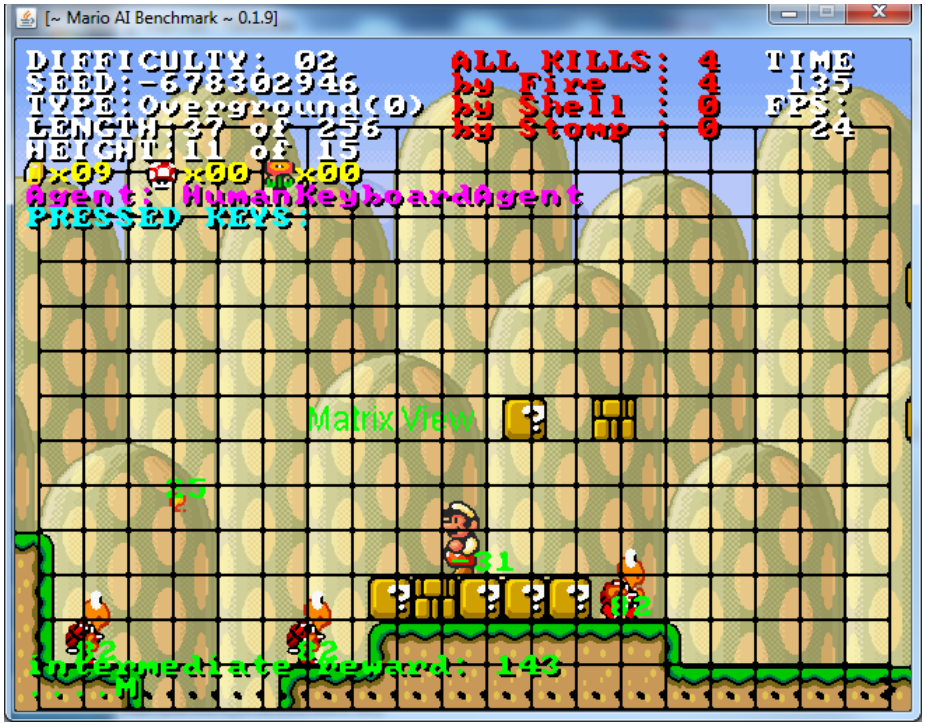

Figure 2.10: A screenshot of Mario AI showing the receptive field (grid overlay) and several other pieces of data available for the controller. Each cell in the receptive field contains a value depending on what it contains. In this case, the cell with Mario in it contains the value -31 and the cells with enemies contains the value 82. Empty cells contain a zero value.

2.5.1 Challenge in Mario AI

From an AI perspective, platform games and Super Mario Bros. in particular serve as an interesting challenge as it has a very high-dimensional state and observation space (Sergey Karakovskiy, Julian Togelius [8]). It also has a relatively high action space where the effect of several actions are highly dependent on being executed in proper sequence. An example of this is jumping, where the jump button has to be continuously held for several frames in order to control the height of the jump, and has to be released after landing on the ground in order to be able to jump again later on. This makes it nearly impossible for an AI to generate specific rules for each game state, and it instead forces the AI to learn to generalize and find patterns and similarities between different states and which set of actions that enables it to get from these states and into more favourable ones.

In order to make sensible decisions and generalize properly, an AI also has to be able to learn that several different game objects have the similar behaviour or physical properties as others. This means that there are many objects within the game that demand the same type of interaction in order for the AI to be able to traverse the level. Similar objects such as blocks and item boxes have very similar properties in the game, but are represented with different values in the receptive field. This is also the case for different types of enemies. Most enemies can be killed by jumping on top of them or shooting them with a fireball, but it can be difficult for the AI to know what are enemies or not when the enemies are represented with different values in the receptive field. In order to get good gameplay performance, the AI therefore has to be able to approximate input-output functions that yield similar results for several different input values.

A big problem for an AI playing Mario AI without the use of hard coded actions is figuring out how the controller works. As mentioned earlier, there are 64 different combinations of button presses that can be performed at each timeframe. It is therefore very difficult for an AI to learn how the different combinations of key presses actually affect the state of the game. This becomes more difficult when some combinations of key presses do exactly the same as another or when some key presses do different things based on which game state you are in. Examples of these are pressing left and right at the same time which is essentially the same as pressing neither key, or pressing down while in the air which does nothing. The ambiguous effects of such combinations of button presses therefore end up as noise in the output space of the AI, making it more difficult to decode the interplay between combination of button presses and game state changes. This also becomes more of a problem considering the fact that some actions such as running require a sequence of button presses over several frames, but where it is also possible to perform the action while also pressing additional buttons.

Super Mario Bros. is a game of dynamically changing environments, where the

player can only see a small portion of the level at any time. This means that the AI has to be able to cope with an environment that radically changes as Mario traverses the level. The game also runs in real-time, giving the AI very little time to calculate which move to do next, making it difficult to calculate the next move by looking many time-steps ahead. For human players and AI alike, Super Mario Bros. has a smooth learning curve between levels, both in terms of which behaviours are necessary in order to complete a level and their degree of refinement. This means that very simple levels with no enemies and only a few holes and obstacles might only require the AI to be able to keep running to the right while jumping whenever it reaches an obstacle. However in order to excel at the game and beat every level, the AI has to learn the different attributes of the terrain and enemies and how to get past or interact with them. This includes for instance learning that some enemies are impervious to stomping or that bricks can be broken by jumping beneath them as long as Mario has any power-ups.

Chapter 3

Motivation

This chapter summarises the specialization project which this thesis is based on and describes some of the related work that was used during the work of this thesis.

3.1 Specialization project

This thesis is based on the work done by the same authors in their specialization project.

The project resulted in an approach for creating automated controllers for the Mario AI framework using multi-objective evolutionary algorithms. The goal was to create controllers with a diverse set of behaviors and to combine these behaviors into individuals that can play the game well and behave more intelligently than the individual behaviors. A system for creating such controllers was developed using the non-dominated sorting genetic algorithm-II (NSGA-II) combined with artificial neural networks. The controllers were tested in the Mario AI framework and the results showed that the developed approach could produce controllers with different behaviors and that combining them could lead to more intelligent behavior.

Several different techniques were investigated before the initial system was developed. Both evolutionary algorithms and artificial neural networks had been used individually in earlier works. Using these techniques individually for controlling agents in video games where found to have their strengths and weaknesses. In the approaches investigated, it was found that ANNs needed learning data from human players and GAs needed to have predefined behaviors created. Other researchers had avoided or solved these problems by using neuroevolution techniques. Neuroevolution of augmented topologies and multi-objective evolutionary algorithms were found to be popular choices of techniques for evolving

neural network controllers for video games. Multi-objective optimization for evolving neural networks with fixed topology was chosen as the main technique because it allowed the problem of playing the game well to be divided into sub-problems and could create a diverse set of individuals.

Experiments were conducted to measure the ability of the system to create controllers that could perform well at playing the game, as well as to create controllers with different behaviors and whether these could be combined into controllers with intelligent behavior, benefitting from the different behaviors. The results showed that the controllers produced by the system was able to complete over 50% of some test levels, which means they had reasonably well performance. By visual inspection it was noticed that the controllers often got stuck when trying to jump because they continuously held down the jump button or did not try to jump at all when reaching small walls or pipes. It was also noticed that the system could produce controllers with different behaviors, such as standing still and avoiding enemies, running fast towards the goal and taking damage, shooting many enemies and jumping on enemies depending on which objectives were used. Some combinations of objectives such as to get far and to avoid taking damage resulted in controllers that combined the different behaviors and behaved rather intelligently such as running towards the goal in order to get far while shooting and jumping on enemies in order to get high score from killing enemies. It was concluded that the system was able to develop controllers with both good performance and intelligent behaviors, but some changes were suggested in order to improve the system further. The system is described together with the implemented improvements in chapter 4.

3.2 Related work

This section includes some of the related work that was used as inspiration and guidance for this thesis. The related work is grouped together into the relevant categories and given a short explanation as to why they are included and how they affected the work of this thesis.

3.2.1 Evolution of behaviors

This section describes some approaches for evolving controllers that display different behaviors in video games.

Hong and Cho [3] used a GA to evolve controllers with different behaviors in a game called Robocode. In Robocode, a tank controlled by a program provided by a player battles with other tanks. The tank has to deal damage to the opponent and prevent the opponent from damaging it. The authors evolved controllers with a GA that is used to evolve behavior parameters that are used to select a combination of simple predefined hand-coded strategies.

These evolved controllers are tested against opponents with different behaviors and the result shows that the GA is able to evolve controllers with different behaviors depending on what behavior it is faced against and win against the opponent. Some examples of evolved behaviors includes running away from the opponent, shooting at random, accurately tracing donw the opponent while shooting and bumping the opponent. This shows that evolutionary approaches, such as GAs, can be used to create controllers with a variety of intelligent behaviors and even find the appropriate behavior for different opponents. One of the weaknesses with this approach is that the evolved behavior is not very general. An evolved behavior may be good against the opponent it trained against, but if the opponent changes its behavior, then the evolved controller will not necessarily be able to adapt to this change. Another weakness with this particular approach is that simple hand-coded strategies needed to be designed and implemented before the evolutionary process could begin and the set of possible behaviors for the agent is defined and limited by these strategies. If the optimal strategy is not well known, it can be difficult to be certain that the best strategies are achievable by the evolutionary process because it is not known which strategies such an optimal behavior consists of. The system developed in this thesis is different from the system by Hong and Cho because it does not need create any predefined behaviors before the evolutionary process, but it similarly takes advantage of genetic algorithms by evolving individuals based on continuous feedback about its performance from the environment.

Schrum and Miikkulainen [5] used multi-objective optimization together with constructive neuro-evolution to evolve controllers for a game called Battle Domain. In this game, several NPCs have to destroy a player bot by dealing damage to it, while at the same time avoid taking damage from the player. These conflicting goals are difficult for GA approaches with scalar fitness values, such as the one by Hong and Cho [3] , because solutions tend to get stuck in local optimas where they do well at one part of the task at the expense of the other instead of performing reasonably well at both parts. A multi-objective evolutionary approach where the task was divided into separate objectives that are rewarded separately was compared against such a single objectibe GA approach. The results showed that the MOO approach resulted in populations that was able to mix between charging and baiting behaviors, which lead to better performance for the NPCs. This shows that MOO can be very successful for creating intelligent behaviors for controllers in domains with multiple contradictory objectives and that rewarding objectives separately and then combining them is a good way of achieving multi-modal behavior.

3.2.2 Multi-modal behavior

This section describes some approaches for creating multi-modal behavior for controllers in video games.

Schrum and Miikkulainen [6] used multi-objective optimization and neuroevolution to create controllers for a game called Fight or Flight. This game is divided into two separate tasks and the controllers are playing one task at a time. The fight task is similar to the Battle Domain game used in [5], where the NPCs have to deal damage to a player while avoiding being hit by the player. In the flight task the player tries to escape from the NPCs and the NPCs have to surround the player and deal damage to it. These two tasks requires different modes of behavior in order for the NPCs to be successfull and a method is used to evolve networks that are able to change behavior when the situation changes.

This method is called Mode Mutation and works by adding a mode mutation operator to the evolutionary process which works by adding a new set of output nodes to the ANN. The ANNs created by the Mode Mutation method has a set of output nodes for each mode which include the nodes needed to control the NPC in addition to a preference node. Which mode gets selected when an action is to be taken is decided by which preference node has the highest value. This way the networks should be able to switch between modes based on the current situation. In Mode Mutation networks the features learned in the hidden layer is the same for all modes, but the output nodes are different. The reasoning for this is that the networks should not need to learn the basic behavior more than once.

The Mode Mutation method was compared against a 1Mode method, with only one set of output modes, on the Fight and Flight tasks and the results showed that Mode Mutation was able to evolve different behaviors for different situations. Mode Mutation was better at completing the task than 1Mode. The results also showed that Mode Mutation had good performance at all objectives, while 1Mode is good at one objective at the expence of others. Complex behavior was observed in the trials where Mode Mutation was able to clear the task. For example, in the fight task, a behavior where one NPC was used as bait for the player bot and the others chased the player from behind was observed. In this case the NPCs evolved one set of output nodes for baiting and another for chasing and switched between the modes depending on where the player bot was positioned. In the Flight task, a behavior where the NPCs was able to surround the player bot and hit it between each other was observed. For the 1Mode method, where individuals were focusing on only one objective, an example of observed behavior was the NPCs attacking the bot in order to deal damage, but all individuals eventually died because they did not care about avoiding damage.

These results shows that Mode Mutation is a promising way to make the NPC controllers able to do well at multiple objectives at the same time and that simply using individuals from the pareto front, as in 1Mode can lead to controllers that only focus on one objective at the expense of others. The authors suggests that it might be possible to taking advantage of the diverse population with specialized individuals by treating the population as an ensemble. The

reasoning for this is because a diverse population is likely to have at least one member that is suitable for different situations, and that having a way to change between which individuals controls the NPCs will lead to more appropriate behavior. This approach will be explored further in one of the experiments in this thesis and compared against the Mode Mutation approach. One difference between the experiments in this thesis and from the ones in [6] is that the task division is less explicit in this thesis. This means that the game is not divided into seperate tasks with contradictory objectives, but instead there are separate tasks which require some of the same basic behavior as well as some behavior specialized for each task. The reasoning for why Mode Mutation is appropriate is still applicable for the less explicit task division, since the basic behavior required for all tasks should be learned in the networks hidden layers while the specialized behavior is learned in the output layers of each mode.

In [7], Schrum and Miikkulainen extended the work in [6] by including more techniques and applying each method to two separate games. Three different techniques are used. *Multinetwork* learns separate controllers for each task in the game, and then combines them manually before they are used by the NPCs. *Multitask* evolves separate output units for each task and shares information within the hidden layer of the network. *Mode Mutation* (as explained in [6]) evolves new output modes, and selects the mode to use at any given time by using preference nodes. In contrast to Multinetwork and Multitask, Mode Mutation does not require that the task division is known.

These techniques are used in two separate games, Front/Back Ramming and Predator/Prey. Both of these games are multitask games because they are divided into two separate tasks and the controller are evaluated based on performance in both tasks. In Front/Back Ramming the NPCs have to battle with the player bot (similar to the Fight task in [6]), but under two slightly different circumstances. In one task, the NPCs have their bat equipped at the front of their body and in the other they have the bat equipped at the back. This forces the NPCs to be able to change behavior between tasks such that the bat is facing the player bot. In the Predator/Prey game the NPCs have to learn both offensive and defensive behavior. In the Predator task they have to surround the player bot and hit it between each other and in the Prey task they have to escape from the player bot.

In the Front/Back ramming game, Multitask and Multinetwork gave the best result because they do well in all objectives rather than focusing on the extreme regions of the trade-off surface of the pareto front. Mode Mutation networks get lower scores than Multitask and Multinetwork and one of the reasons could be because they lack explicit knowledge about what task they are doing.

In the Predator/Prey game Multinetwork and Mode Mutation gave the best results and was able to do both tasks well, but Multitask was not very good at this game. One of the reasons why could be because the two tasks are not equally difficult to learn and Multinetwork might put too much attention at learning the easiest task when it should be focusing mostly at the harder task. These

results also indicate that Mode Mutation could be a good technique when the task division is not well known, since it does not have to be told explicitly which mode it is currently in. These results showed that it was able to switch mode based on environmental cues and that this mode switch became permanent for some time due to the recurrent connections added by mutation. This makes it an attractive approach for the game used in this thesis because the task division is not explicit and therefore not well known.

Overall, Mode Mutation and Multitask was good at one game each, while Multinetwork was good at both games. The fact that Multinetwork always performs well indicates that combining controllers that are good at their seperate tasks is a good way to achieve good behavior across different tasks simultaneously. This indicates that an ensemble of the best individuals on the pareto front could be could be a good approach for performing well at multiple tasks at the same time. This is explored further in this thesis by training each of the networks simultaneously with a multi-objective evolutionary algorithm, instead of training each network on its own task, and then combining them into an ensemble.

3.2.3 Ensemble methods

This section describes some of the ensemble methods used in the literature for combining multiple ANNs into a single controller in video games.

Tan et al. [10] used a game called Ms. Pac-Man as test bed for comparing performance of controllers consisting of a single ANN and ensembles of ANNs. An ensemble consist of five different ANNs that are combined by Winner Takes All voting. Each ANN outputs an action of the game playing agent with a value indicating how much they want to take that action. The output of the ensemble is simply the action with the highest value among the members of the ensemble. The ANNs are evolved using multi-objective optimization and the best ANN is selected from the pareto front. The performance of the two approaches are then compared in the game and the results show that the ensemble manages to achieve a higher average and maximum score than the single network. This indicates that ensembles with Winner Takes All policy is a good way to combine several ANNs into a single controller for game playing agents. A similar approach is explored in this thesis, where a pareto front is evolved using multi-objective optimization and individuals are used in an ensemble, but the game and objectives used are different.

Chaperot and Fyfe [1] used a Motocross game to evaluate several approaches for improving the performance of the agent controlling the bike. The bike is controlled by an ANN which is trained on human data. One of the approaches was to create several ANNs where each ANN was trained on unique set of the training data. These individual ANNs were then combined into an ensemble. The output of the ensemble was a combination of the average of all the individual ANNs and the output of the most confident ANN, which is the output with the

highest magnitude. This approach was tested using ten ANNs and the results showed that the ensemble was better than each individual ANN, but not as good as another ANN trained using the full data set. This indicates that training ANNs on seperate sets of training data and then combining them does not lead to better performance for game playing agents in video games, at least without careful consideration to how they are combined.

Jang et al. [4] used NEAT to evolve multiple species of ANN controllers for a Real-Time Strategy Game. The controllers had one input neuron for each grid on the map and output neuron for each of four possible actions. The output value represents how much the controller prefers that action. One of the differences between these controllers and the ones created by the system used in this thesis is that this controller only makes one decision, that is which of the different actions it should take, whereas the controllers used in this thesis have to make a decision of whether or not to press each of the buttons. They do have a similar input layers, however, as both controllers get input based on a grid of values that represents the environment surrounding the agent. Five different species are combined into an ensemble and several methods of combining the individuals are tested in the game and compared. The following methods were used:

- Voting: Each individual votes for the action with highest output value. The action with most votes is selected by the ensemble.
- Average: The action with the highest average value among the individuals is selected by the ensemble.
- Winner Takes All: The action with the highest value among all the individuals is selected by the ensemble.
- Borda Count: Each individual gives points to each action based on its rank. The action with the highest score from all the individuals is selected by the ensemble.
- Single Gene: The prefered output of the individual with the highest fitness is selected by the ensemble.

The results showed that the methods that used more than one individual was more successful. Winner Takes All and Average got the best maximum scores while Borda Count and Average got the best average scores. Single Gene and Voting both got the worse maximum scores and average scores. This result shows that Average, Winner Takes All and Borda Count are good candidates for combining individuals in an ensemble with different individuals. Average seems particularly good, since it was among the two best methods in both maximum and average score. The authors also mentions that these results only counts for the domain they tested it in and that the results could be different for some other domain. In the system developed for this thesis, the individuals in the ensemble are members of a pareto front instead of species created by NEAT and there is a similarity between them in that both ensembles consists of members with

different strenghts and weaknesses instead of homogenous individuals. Therefore it is fair to assume that this type of ensemble will be useful for members of a pareto front as well, especially if its action is chosen based on Average, Winner Takes All or Borda Count policies.

Chapter 4

System

This chapter describes the original system developed in the specialization project and the new improvements implemented for the experiments in this thesis.

4.1 Original system

The original system was developed by the authors in a specialization project from the semester before this thesis. The original system evolved artificial neural networks using multi-objective neuroevolution, and these networks were used as controllers in Mario AI. The topology of the ANNs were static and were designed by the authors. The ANNs are modified by the multi-objective evolutionary algorithm by evolving the values of the connection weights. The details of the system will be explained further in this section.

4.1.1 Controlling Mario using Artificial Neural Networks

The controllers for the Mario AI game are defined by their respective artificial neural network. At each timestep the controller receives information about the environment and game state from the game and responds with an action. This action consists of whether each of the six different buttons should be pressed or not. Four of the buttons are for moving left, right, up and down and the two others are for jumping and running/shooting. The controller uses its ANN to decide what action to take based on the information it receives from the environment.

The topology of the ANN is fixed for all the controllers and is fully connected. Each ANN has 6 output nodes, and each output is used to decide whether one of the six buttons should be pressed or not. The controllers receives information

about the environment through a receptive field which has a resolution of 19 × 19 blocks, as explained in section 2.5. The ANN has one input node for each block in the receptive field, resulting in a total of 19 × 19 = 361 input nodes. The ANN also has 10 hidden nodes because having a hidden layer was found to give better results than without hidden layer and increasing the number of hidden nodes did not significantly improve the results further. This topology was chosen because it is a straightforward way to connect the input data to the controller buttons and because the ANN is only used as a tool for controlling the agent in the game and the job of solving the task is given to the evolutionary algorithm.

At each timestep the controller receives a receptive field of values from the environment and feeds this into its ANN. The ANN then outputs a value for each of the buttons and the controller responds to the game by pressing each button with an output value over a given threshold. This threshold was set to 0.5 in the initial system after testing was done with different values. Each neuron in the ANN has a sigmoid activation function with real output values, which gives each neuron an output value in the range [0.0, 1.0]. An illustration of this interaction between the game and the ANN of the controller can be seen in Figure 4.1. Each cell in the receptive field is sent to its corresponding input node, which is then processed through the network until the network emits an output signal for each button.

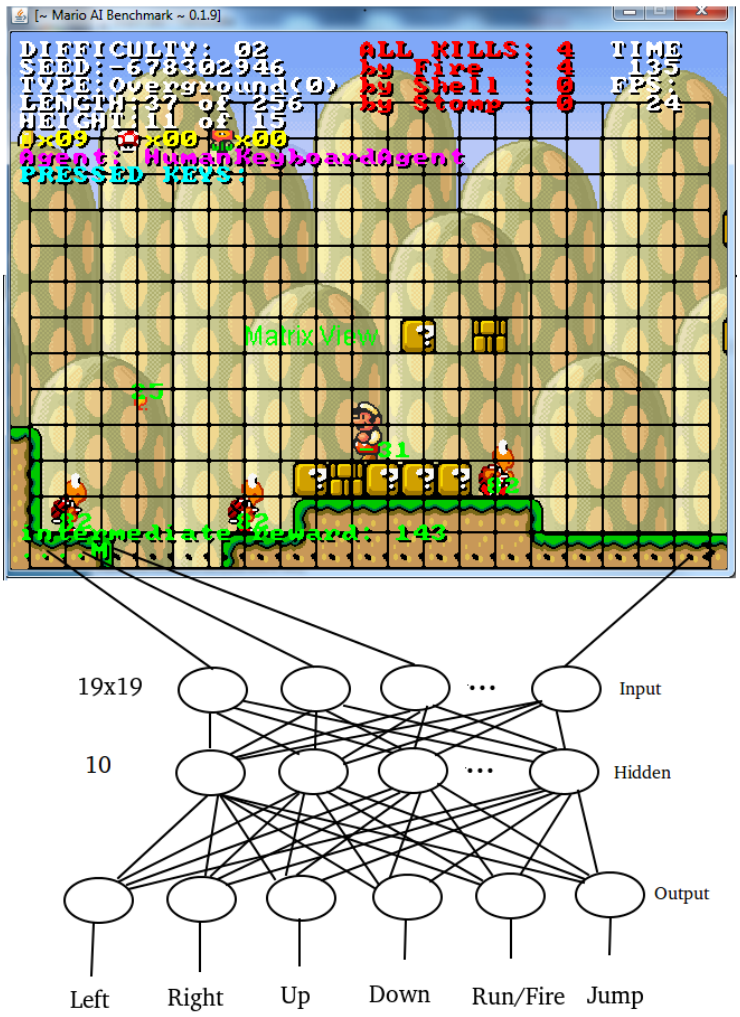

Figure 4.1: The ANN of a controller where the receptive field data is sent to the input nodes and the output nodes represent button presses.

4.1.2 Evolving controllers using NSGA-II

The system evolves controllers by using a multi-objective evolutionary algorithm called NSGA-II, as described in 2.2.1. The system use a version of the algorithm provided by the jMetal framework, which has been slightly modified to be able to evolve the specific controllers used in the Mario AI game. These modifications are related to genotype representation and mutation operators and not to the logic of the algorithm itself.

The flowchart of the evolutionary process used to create controllers can be seen in Figure 4.2.

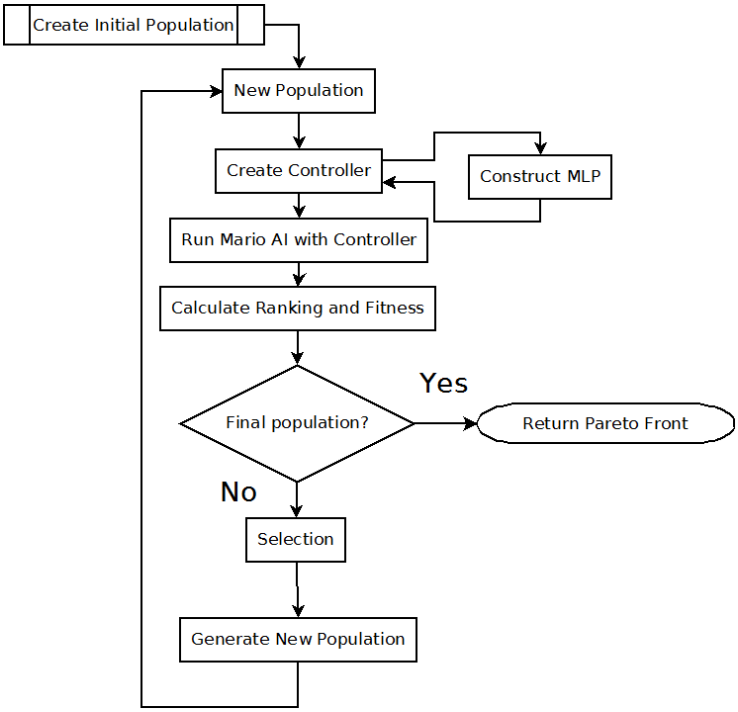

Figure 4.2: Flowchart of the system.

The system starts by creating an initial population of individuals with randomly assigned values to their genomes. Each individual has a genome that is a list of real numbers in the range [-1.0, 1.0] which are used as weights in the ANNs. The phenotypes are created by using the values of the genotypes as weights in their neural networks, which means that there is a one-to-one mapping between genotypes and phenotypes. The phenotype individuals are used as controllers in the Mario AI game to obtain statistics about their playing performance. This performance data is used to calculate the objective values of each individual. Fitness is rewarded based on the ranking of objective values, as described in

2.2.1. The following objectives were used in the system:

- KILL_MANY: The number of enemies the controller is able to kill during performance testing.
- MANY_COINS: The number of coins the controller is able to obtain during performance testing.
- LOW_DAMAGE: The number of hits the controller takes from enemies during performance testing.
- HIGH_REWARD: The score Mario AI rewards the controller with during performance testing.
- GET_FAR: The number of cells the controller is able to traverse during performance testing.
- FAST: The amount of time the controller spends playing during performance testing.

These objectives were chosen by the authors because they are either directly related to how to play the game well or subgoals within the game. All the objectives, except for LOW_DAMAGE and FAST, are objectives the controller is trying to maximize.

Parents and adults are selected based on the fitness of the individuals and new individuals are created by applying the genetic operators. The only genetic operator used in this system is mutation. The mutation works by randomly changing the value of some of the weights. The chance of having a weight mutated and the ammount it changes can be set when starting experiments. Crossover is not used in the system because of how the individuals are represented as genotypes. Since the genotypes are used as artificial neural networks, it is difficult to perform meaningful crossover between them. One of the reasons for this is because the knowledge of a neural network is usually spread throughout most of the network and combining one half of one network with another half of another is more likely to do damage than to combine the good traits of both networks. This is especially true with the size of the networks used in the controllers of this system, with a total of $19 \times 19 \times 10 + 10 \times 6 = 3670$ connections. The new population is used as the starting population for the next generation in the evolutionary process. This routine is repeated until a final number of generations has been reached, where the system returns the non-dominated individuals of the final population as the pareto front of solutions. These individuals are stored to file and can be loaded by the system later for visual inspection if needed.

4.1.3 Limitations

Based on the results obtained from the experiments in the specialization project, several limitations with the controllers of the system were observed.

One potential weakness with the system was that it had one input node for each of the cells in the receptive field with a resolution of 19 × 19. This makes for a rather big input space where it can be difficult for the ANN of the controller to find which input nodes has the greatest importance. Another weakness is that the actual values of each cell in the receptive field can be difficult to interpret because similar game objects can be represented by different values. Most enemies have very similar behavior, but the values they are represented with by the receptive field can be in the range of [80, 100]. The same issue applies to level objects as well. An unbreakable brick have basically the same properties as normal ground, but they are represented by the values -22 and -62 respectively, which is quite different. This makes it possible for the game playing agent to separate the different game objects from each other, but when they are used as input in an ANN, it would be preferable to have more similar values for similar objects. An attempt to solve these issues is explained in section 4.2.

One issue with the controllers produced by the system was that they tend to get stuck besides walls and objects. One of the reasons why they got stuck was because they were holding down the jump button continuously, indicating that they want to jump to get further into the level, but failing to do so. In this scenario, Mario will not jump because in order to make a second jump he first needs to release the jump button and then press it again. Since the controller does not have any information about its prior actions, it does not know that it was already pressing the jump button the last time frame, and therefore do not that it first needs to be released. Memory was later added to the system by allowing the system to evolve its own topology, including recurrent connections. This addition is explained in section 4.3.

4.2 Converted receptive field

In order to reduce the input space and represent similiar objects with more similar values, a way to convert the old receptive field into a smaller one was implemented. The new receptive field have fewer cells and each cell can have fewer possible values. This conversion loses some of the information provided by the original receptive field, but should provide less irrelevant information and make it easier for the controller to interpret the information. The conversion is done in two steps. The first step is to convert the old cell values into new values with fewer possible values and bigger difference between different types of objects. The second part is to reduce the number of inputs by combining the input of adjacent cells that are far away from the center of the receptive field.

The idea behind the conversion of values is to make it easier for the controller to interpret the input. Instead of having a unique value for each type of enemy, all enemies have the same value. This is also done for level objects, so that different types of traversable terrain and objects have the same value as well.

Object	**Value**
Enemy	-100
Mario	-30
Empty cell	0
Fireball	10
Level object	30
Power up	70
Coin	100

Table 4.1: New receptive field values.

The old receptive field can have over 30 different values in its cells depending on what object it contains. The new receptive field only have 7 different values, which can be seen in Table 4.1. All enemies have been grouped together and have a maximum negative value. Coins and power ups have a high positive value in order to make it easier for the controller to separate objects to avoid and objects to collect.

One of the reasons for reducing the number of inputs is to reduce the complexity of the artificial neural networks of the controllers. Since the ANNs have one input neuron for each cell in the receptive field, the number of connection weights for the evolutionary algorithm to optimize is greatly reduced by reducing the number of cells. The new receptive field only has 61 input nodes instead of 361, which for a fully connected ANN with 10 hidden neurons and 6 output neurons means reducing the total number of connections from 3670 to 670. Another reason for reducing the number of inputs is because the cells that are far away from the center of the receptive field are not as important for the controller as the rest. This is because the receptive field is centered on Mario, and it is most important to react to objects that are close to him.

The original receptive field, provided by the Mario AI framework has a resolution of 19×19 cells with Mario centered in the middle of the matrix. The new receptive field remain the same for the 5×5 cells in the middle of the old field, but outside of that area adjacent cells are combined into one. The new cells is given a value based on what was the most important value in the old cells. The values are ranked as follows:

1. Enemies
2. Coins
3. Level objects

If the cell does not contain any values that represent these type of objects the value is set to the same as if it was an empty cell. This only applies for the combined cells, not for the single cells in the middle of the receptive field. The pattern of the converted receptive field can be seen in Figure 4.3. Mario is located in the red cell in the center. The white cells are of the same size as

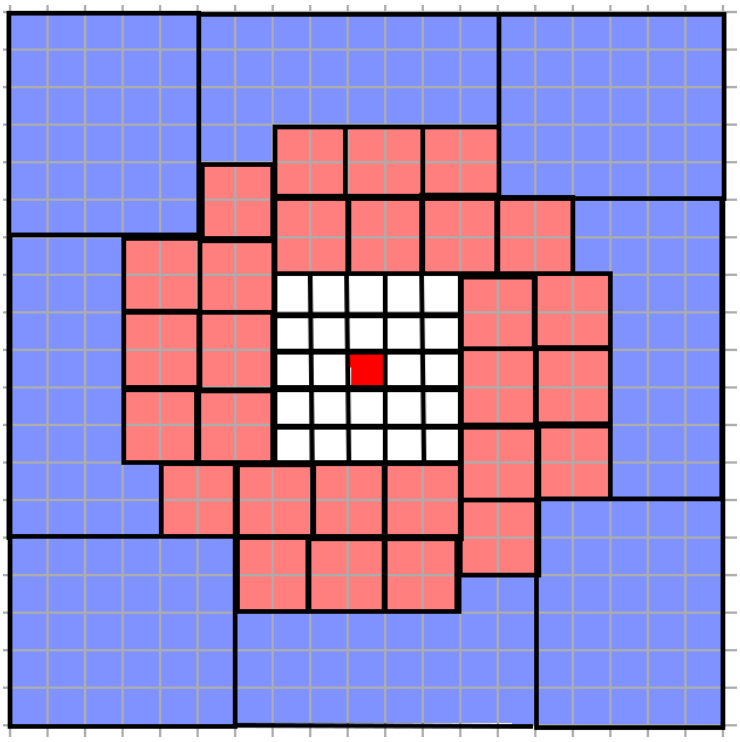

Figure 4.3: Pattern of the converted receptive field. Mario is located in the red cell in the center of the field. The resolution of the cells increase farther away from the center of the receptive field because they contain less important information than the ones in the center.

the cells in the original receptive field. The red cells combines 4 regular cells and have only one value which is calculated as explained above. The blue cells combine even more regular cells and also have only one value. The pattern shown can be considered as a mask that is applied to the original receptive field and translates it into the new one through the mechanics described here.

4.3 Evolving topologies and recurrent connections

As explained earlier, one of the issues with the original system was that it did not have a way for the controllers to remember its prior button presses, making it difficult to perform actions such as jumping, which require continous input over several time frames. This was also one of the reasons why the controllers sometimes got stuck, since they did not release the jump button before press-

ing it again. In order to allow the controllers to handle continous actions, the evolutionary process used in the system was modified to allow the evolutionary algorithm to modify the topology of the artificial neural networks of the controllers. These include modifications to the genotype representation of individuals and the genetic operators. Both these modifications are heavily inspired by NEAT, and have also been used in [5].

The genotypes are now represented by a list of neurons and a list of connections, instead of just one list of connections because the topology of the network is no longer static. This allows both nodes and connections to be added and removed. This representation is very similar to the representation in NEAT, shown in Figure 2.7. The difference is that this system uses three separate lists for input, hidden and output node genes instead of combining them insto a single list. The connection genes include information about input node, output node, weight and if it is disabled or not, but does not have innovation numbers. This is because this system does not use the speciation mechanics from the NEAT algorithm. This and crossover is not used because it was found to lead to overly homogenous populations without fitness sharing in [5]. Fitness sharing is not used because it is difficult to define with multiple objectives.

The individuals in the initial population start with a simple network without hidden neurons. This is different from the original system where each individual had 10 hidden neurons and remained this way through all generations. The mutation operator in the new system allows new connections and nodes to be added in addition to modifying existing connections. Existing connections can be modified by changing its weight, as well as to disable or enable the entire connection. These operations are illustrated in Figure 2.8. Adding a connection is done by either enabling a disabled connection or adding a new connection gene to the connection list. Adding a node is done by adding a new node gene to the hidden node list and adding a new connection from an existing gene to the new gene and from the new gene to an existing gene. These connections can be recurrent, which makes it possible for the network to remember previous actions.

4.4 Evolving controllers using GA

In order to compare the effects of rewarding objectives seperately through multi-objective evolution and combined by using scalar fitness function, a way to evolve controllers through a simple genetic algorithm was developed. The representation of the individuals are kept the same as the ones used in the multi-objective evolutionary algorithm. The evolutionary process used by the GA is the same as the shown in Figure 2.1. Tournament selection and elism were used because it was found to give best performance after some initial testing. Crossover was not used for the same reasons mentioned in 4.1.2.

4.5 Multi-modal network

A multi-modal network approach was implemented in order to measure the ability of the system to create controllers that are good in different modes represented in the experiments as different types of levels, such as levels with and without enemies and levels with and without platforms. This approach is called Mode Mutation and is based on the work by Schrum and Miikkulainen in [6] and [7]. In Mode Mutation, the artificial neural networks of the controllers are multi-modal because they have explicit distinction between the different modes of behavior it has evolved. A mode is defined as a specific behavior for that network and the network is able to switch between each of its modes. Each mode is defined by a unique set of output nodes for that mode in addition to a preference node. The number of output nodes for each mode is the same as for the regular networks. The preference node is used for the network to decide which mode to use. The mode with the highest preference is chosen and the output nodes from this mode is used as output from the network. Each network can have one or more such modes, depending on how they are evolved by the evolutionary algorithm.

Each network starts with one mode when the evololutionary algorithm is initiated. The genotypes of the individuals have been slightly modified to have a list of output nodes for each mode instead of a single list. Connections and nodes can be mutated as explained in section 4.3. New modes can be added through a mode mutation operator which adds a new set of output nodes as well as a preference node for this mode. This mutation process can be seen in Figure 4.4. A multi-modal network with one mode with two output nodes is shown in (a). In (b) the same network is shown after a new mode has been added after mutation. The grey nodes are preference nodes. The red arrows are the new connections that have been added at random between the old network and the new mode. Modes can also be removed by mutation, which is the reversed process of adding a new mode.

The main motivation for using this technique in this system is to allow the controllers to be able to change between behaviors such as collecting coins, avoiding enemies and jumping over obstacles when the situation changes. In the experiments where this technique is used the task division is not very explicit, as some of the same behavior is needed in all the different tasks, but the authors mention in [7] that it may work well in games where the task division is dynamic and overlapping.

4.6 Pareto ensemble

In order to combine individuals from the same pareto front into a single controller, a way to create an ensemble of several individuals was implemented.

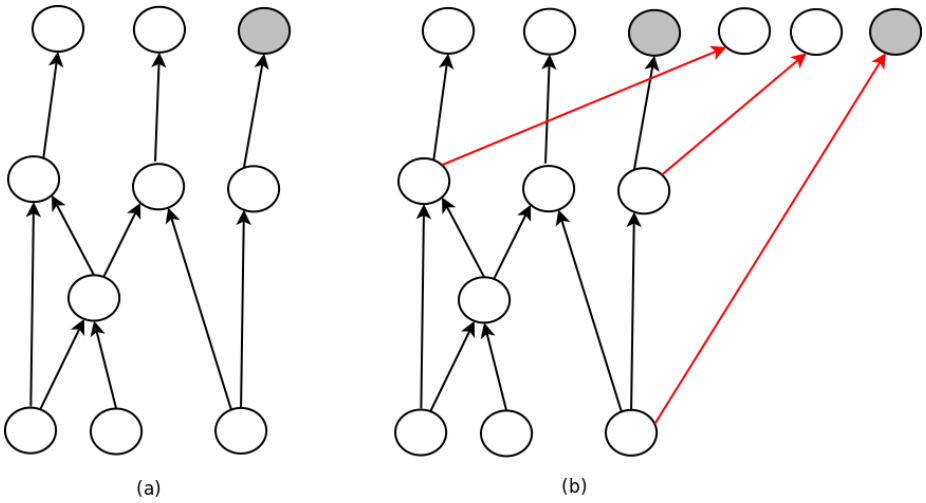

Figure 4.4: Mode Mutation.

The individuals in the ensemble are evolved by using the system with the improvements described in section 4.3. The ensemble controller is created by collecting and combining all the individuals from the resulting pareto front of the evolutionary run. When the ensemble controller receives input from the game, it feeds this input into each of its individual controllers. Each individual controller then process the input through its ANN and sends an output back to the ensemble controller. The ensemble controller then sends output to the game based on the output of each of the individuals. The structure of an ensemble with three individuals can be seen in Figure 4.5. The ensemble controller receives data from the receptive field which it feeds to the corresponding input nodes of each of its individual controllers. The individual controllers process the data through its ANN and sends the output to the corresponding output node of the ensemble controller.

Two ways to combine output from each individual in the ensemble was implemented. The first is *average*, which works by taking the average of each of the six outputs across all individuals and sending this average as output for each output neuron. This means that if there are three individuals in the ensemble, and they output a value of 0.5, 0.6 and 0.7 for the jump output respectively, the ensemble will have a value of 0.6 for its jump output. The same procedure is performed for each set of output nodes. This technique was also used by Jang et al. [4] where results showed it was among the best techniques for both maximum and average scores. The controllers in [4] only made one decision, whereas the ensemble controller described here have to make a decision for each of the six output neurons and this could affect the performance of this technique.

The second way to combine output from each individual is *voting*. Voting is

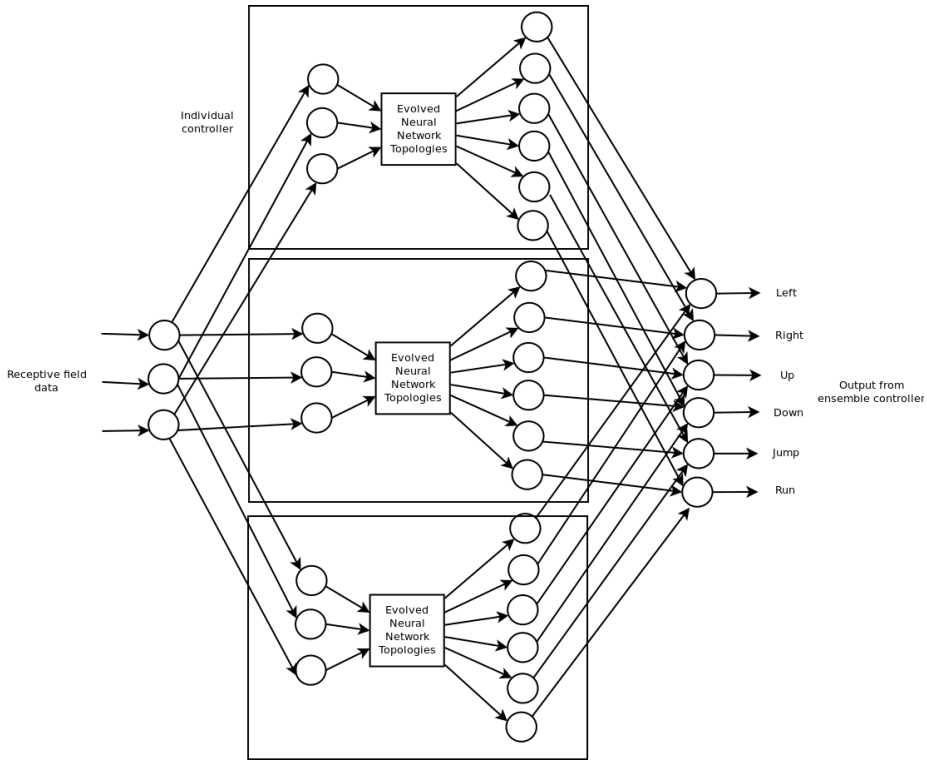

Figure 4.5: The structure of an ensemble controller which consists of three individual controllers taken from the pareto front.

implemented so that each individual in the ensemble sends a vote for each of its output nodes about whether or not the corresponding button should be pressed. The ensemble controller then press each button where the majority voted for it to be pressed. If there are three individuals in the ensemble, and two of them votes for pushing the jump button and one votes for not pushing it, then the ensemble will push the jump button. The same procedure is performed for each button. The same technique was used by Tan et al. [10] , where the ensemble got better performance than the individual ANNs. Similarly to [4], the ensemble used by Tan et al. was used to make only one decision, so this difference could also affect the performance of this technique.

Chapter 5

Experiments

This chapter introduces the experimental plan and the hypotheses used to answer the research questions of the thesis. In this context, a general controller means a controller that is able to traverse levels it has not previously trained on. In all of the experiments, the individuals of interest are those that maximize the main objective of getting far and how these are affected by the other objectives used in their population.

In order to answer **Research Question 1**, the following hypotheses were used:

- **Hypothesis 1** Using the sub-goals of the game as objectives in addition to the main objective of the game will create populations with more general controllers.
- **Hypothesis 2** Using behaviour-changing objectives in addition to the sub-goals and main objective of the game will create populations with more general controllers.

The following hypothesis was used to answer **Research Question 2**:

- **Hypothesis 3** Creating a scalar fitness function comprised of the objectives that makes the most general controllers in a MOEA, also creates more general controllers for a standard GA.

The following hypothesis was used to answer **Research Question 3**:

- **Hypothesis 4** Combining all the individuals of the pareto front into a single controller using an ensemble, creates a controller that is able to maximize all the objectives of the pareto front.

In order to test these hypotheses, a series of experiments were designed. The goal of each of these experiments are to confirm or disconfirm its corresponding hypothesis, and thus help answering the research questions of the thesis. For

Main Objective	Sub-Objective
Reach the end of the level	Kill as many enemies as possible
	Pick up as many coins as possible
	Take as little damage as possible

Table 5.1: Objectives used in the first experiment

each run of the system a list of the objective scores of each of the controllers in the resulting pareto front is kept and it is these scores that were used when comparing the different controllers to each other.

5.1 Experiment One

The goal of this experiment was to test Hypothesis 1, regarding if adding sub-goals as objectives would create more general controllers. In order to test this we used NSGA-II using only the game's main objective and compared it to controllers evolved using the main objective of the game in addition to one of the sub-objectives of the game. The different objectives used can be seen in Table 5.1 which were all found using domain knowledge and the fact that all of these objectives contribute to the score of the original Super Mario Bros. game.

5.1.1 Phase One

In the first phase of the experiment, benchmark controllers were created using only the game's main objective. These were then compared to controllers created using the main goal of the game in addition to one of the sub-goals of the game. The following list displays step by step how the experiment was carried out:

1. Created a benchmark by using NSGA-II with only the main objective.
2. Compared the benchmark to a run with the added objective of killing as many enemies as possible.
3. Compared the benchmark to a run with the added objective of picking up as many coins as possible.
4. Compared the benchmark to a run with the added objective of taking as little damage as possible.

5.1.2 Phase Two

In the second phase of the experiment, the objectives that were found helpful in creating general controllers from the first phase of the experiment would all be used in the same population, to see whether or not this would further improve the generality of the individuals of interest. The following list details step by step how the experiment was carried out:

1. Inspected the results of the phase one and selected the objectives that led to the most general controllers.
2. Run the system using the selected objectives and compared them to the benchmark and the controllers using only one of these additional objectives.

5.1.3 Expectations

From this experiment, the results were expected to show that for each objective added to the run, the controllers of interest would be more general as well as showing improvements on the new objective added to the population.

5.2 Experiment Two

The goal of this experiment was to test Hypothesis 2, concering adding objectives that are specifically tailored to changing the behaviour of the controllers. These objectives would then be used one by one together with the objectives that created the most general controllers from the first experiment. The objectives that were used for this experiment were found using expert knowledge of the game and by analyzing which actions are generally helpful or harmful for the controller to perform. The objectives selected were as follows:

1. Jump as much as possible
2. Shoot as much as possible
3. Duck as little as possible

The following list details step by step how the experiment was carried out:

1. NSGA-II was used with the objectives from the most general controller of the first experiment as well as the objective of jumping as much as possible.
2. NSGA-II was used with the objectives from the most general controller of the first experiment as well as the objective of shooting as much as possible

3. NSGA-II was used with the objectives from the most general controller of the first experiment as well as the objective of ducking as little as possible.

4. Compared each controller with the controllers from the first experiment that used the same sub-goals as objectives.

5.2.1 Expectations

The results from this experiment were expected to show that each of the behaviour-changing objectives would further improve the generality of the controllers, but not in an as high degree as using the sub-goals of the game as evolution might have been able to figure out these behaviours in order to maximize the sub-goals of the game.

5.3 Experiment Three

The goal of this experiment was to test Hypothesis 3, concerning if a standard GA with a scalar fitness function comprised of the same objectives used in the MOEA would also create more general individuals than those created using a fitness function that only considers the main goal of the game. In order to test this, two seperate runs of a standard GA was run, both using a different fitness function. The following list details step by step how the experiment was run:

1. Ran the GA with a fitness function that was only based on getting as far into the levels as possible.

2. Ran the GA with a fitness function that was based on the all of the objectives from the most general controller found in the first two experiments.

3. Compared both controllers to those created using NSGA-II with the same objectives.

5.3.1 Expectations

The results from this experiment were expected to show that using a scalar fitness function comprised of the objectives that created the most general controllers using NSGA-II, would help guide the evolution of the individuals in the standard GA towards making individuals that were able to get further into the levels than those not using all of the objectives.

5.4 Experiment Four

The objective of this experiment was to test Hypothesis 4, concerning the ability of an ensemble to maximize all objectives used in the pareto front. For this experiment a separate set of test and training levels were designed to specifically test each of the different sub-goals of the game. This meant that there were some levels that were completely flat with different types of enemies where the controllers started with and some without the fire flower powerup, some levels with coins and some longer levels with different terrain the controller would have to scale without enemies. A run of the system was done using NSGA-II with the game's main objective as well as all of the sub-goal objectives and combined all the individual of the pareto front into an ensemble that uses voting as a mechanism to decide what action to take at a given time-step. In order to compare the effectiveness of this approach to other state of the art approaches for creating controllers that are able to combine several different behaviours into one controller, a controller using a multi-modal network evolved on the same training levels as the ensemble was used. These two controllers were then compared to each other as well as the single individual from the pareto front that maximized the main objective of the game. The different steps of the experiment were as follows:

1. Used NSGA-II with the game's main objective as well as all of its sub-objectives.
2. Created an ensemble out of all the individuals in the pareto front created in the previous step
3. Created a controller using a multi-modal network evolved on the same levels as the ensemble controller.
4. Compared the multi-modal controller, ensemble controller and the single individual controller to each other.

5.4.1 Expectations

The results from this experiment were expected to show that the ensemble controller would be able to match or exceed the multi-modal controller on all the objectives of the pareto front, with the single individual controller displaying worse results on every objective.

5.5 Experimental Setup

This section introduces the experimental setup, including the general setup that was used for all of the experiments as well as the specific values that were used for each individual experiment.

For all of the experiments, the algorithm in question was run 20 times per set of objectives with a population of 50 individuals over 300 generations. These parameters were chosen in order to let the algorithms produce decent results without spending too much time to complete the run. The experiments were all ran on the Clustis3 cluster of computers at NTNU, where each run of 20 different populations would take approximately 3 hours. None of the experiments used crossover and the selection mechanism found to be the most effective was tournament selection where the winner of each tournament had a chance of 0.9 to be selected as a parent for the next generation. All networks were also instantiated as a feed-forward network with 10 hidden nodes with random connection weights with a minumum value of -1 and a maximum value of 1. All experiments except experiment four uses the same set of training and test data, which were randomly generated by using the same seed and parameters.

5.5.1 Experiment One

NSGA-II was used with a 0.8 chance of having its connections mutated, 0.2 chance of adding an additional node to the network and a 0.3 chance of adding an additional connection to the network. Given that an individuals connections were going to be mutated, for each connection there was a 0.05 chance of perturbing the value of its weight, a 0.1 chance of deactivating the connection and a 0.05 chance of reactivating the connection. In the first phase, the experiment was ran with 4 different sets of objectives. First with the objective of getting as far as possible alone, and then 3 different sets containing both getting as far as possible and killing as many enemies, collecting as many coins and taking as little damage as possible. Finally in phase two of the experiment the algorithm was ran using the objective set of getting as far as possible, collecting as many coins as possible and taking as little damage as possible.

5.5.2 Experiment Two

NSGA-II was used with the same parameters as in experiments one, but with different objectives. The algorithm was ran with 3 different sets of objectives, all containing the objective of getting as far as possible and collecting as many coins as possible, where the different runs used the additional objective of jumping as much, shooting as much, and ducking as little as possible.

5.5.3 Experiment Three

In order to select the parameters used by the GA, several tests were ran and the preliminary results showed that a high degree of elitism and a high multiplier for the objective of picking up many coins was beneficial for getting individuals that were able to traverse the levels while also picking up the coins that were

scattered through the levels. A standard GA was used with an elitism rate of 0.5 and where each tournament covered 5% of the population. The mutation parameters were the same as those used in the previous experiments. The fitness function chosen was $Distance + (20 \times Coins)$.

5.5.4 Experiment Four

A different set of test and training levels were used in this experiment. 2/3 of the levels are completely flat with different types of enemies while the rest of the levels are without enemies but with varying terrain. In 1/2 of the levels with enemies Mario starts with the fire flower powerup while in the others he starts with the super mushroom power-up.

NSGA-II was used, first with NEAT and the same parameters as in experiment one except that all of our objectives were used, which includes getting as far as possible, taking as little damage as possible, killing as many enemies as possible and collecting as many coins as possible. Then secondly, a multi-modal network was used with a 0.1 chance of perturbing any given weight, a 0.1 chance of deleting a mode, a 0.2 chance of adding a new mode, a 0.5 chance of adding a new connection to the network, a 0.3 chance of removing a connection from the network, a 0.25 chance of adding a node to the network and a 0.1 chance of removing a node from the network.

Chapter 6

Results

6.1 Experiment One

As can be seen in Figure 6.1 adding an additional objective other than the main objective of the game did not significally change the increase of the controllers score at the main objective of the game during evolution, except for the objective of not taking any damage which seemed to have a slightly more negative impact than the other objectives. This could be due to the fact this is the only objective that is conflicting to the main objective. In early stages of evolution, most of the controllers that are able to maximize this objective does so through going as far as they can to the left before crouching down and waiting for the timer to go out, something that causes the controllers to traverse an as small part of the level as possible. An screenshot of this behaviour can be seen in Figure 6.2.

Table 6.1 shows the differences between the different objective scores on the training and test levels for the different populations. Here we see that even though the performance on all objectives except taking little damage plummits and none of the controllers with a second objective are able to achieve better scores at them than the other controllers. The two populations using the objectives of not taking damage and collecting coins on however do display better results at getting far than the benchmark on the test data. The replays of the individuals show that many of them are much worse at scaling the different obstacles in the test levels. Such behaviour can be seen in Figure 6.4 where the controller tries to jump over an obstacle but is unable to, due to the fact that it never releases the jump button. Not being able to traverse the test levels properly also has a distinct negative effect on the controllers secondary objectives, as they are all reliant on the controllers ability to go scale obstacles. This is not true for the objective of not taking damage though, as the controller is exposed to more enemies the further he is able to get on each level, something that explains why most of the controllers are able to improve on this objective

		Distance	Kills	Damage Taken	Coins collected
Benchmark	Training	**13700**	48	11	144
	Testing	**5700**	17	9	45
Low damage	Training	**13000**	48	**9**	143
	Testing	**5950**	18	**10**	55
Kills	Training	**13200**	**60**	9	143
	Testing	**5400**	**20**	8	44
Coins	Training	**13800**	47	10	**152**
	Testing	**6400**	20	10	**50**
Combined	Training	**13200**	57	**10**	**148**
	Testing	**5350**	17	**8**	**44**

Table 6.1: The average scores on the objectives by the best individual of each population. The objectives that were used in the different populations are marked in bold

when they traverse smaller parts of the test levels. On the training levels however, the controllers from the populations with the objective of collecting coins would some times jump ontop of platforms in order to collect the coins that were there even though jumping on top of the platform would not help it advance the level in any way. An example of this behaviour can be seen in Figure 6.3. It is possible that this behaviour is what makes these controllers more general than the others.

As only the objective of collecting coins and not taking damage did better than the benchmark, only these were used for phase two of the experiment and the results from both the training and test levels can also be seen in Figure 6.1 and Table 6.1 under the name "combined". Instead of displaying results similar to those of the populations using the same objectives, these controllers displayed results more similar to those of the controllers with the objective of killing enemies.

These results indicate that using some of the sub-goals of the game as objectives can be beneficial for a MOEAs ability to create general controller, but that not all of them necessarily makes for more general controllers. They also indicate that using several of these objectives together can create controllers that behave very differently from the controllers using the objectives seperately.

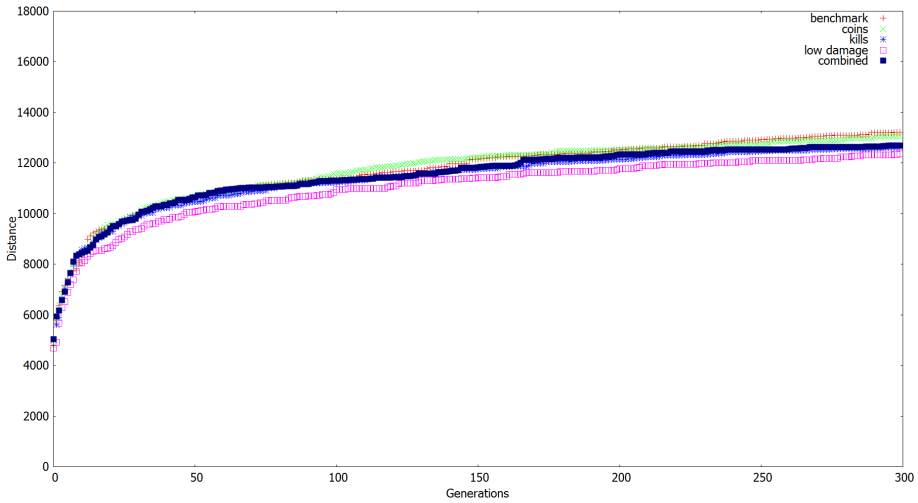

Figure 6.1: Average score at getting far of the best individual in each of the populations in experiment one.

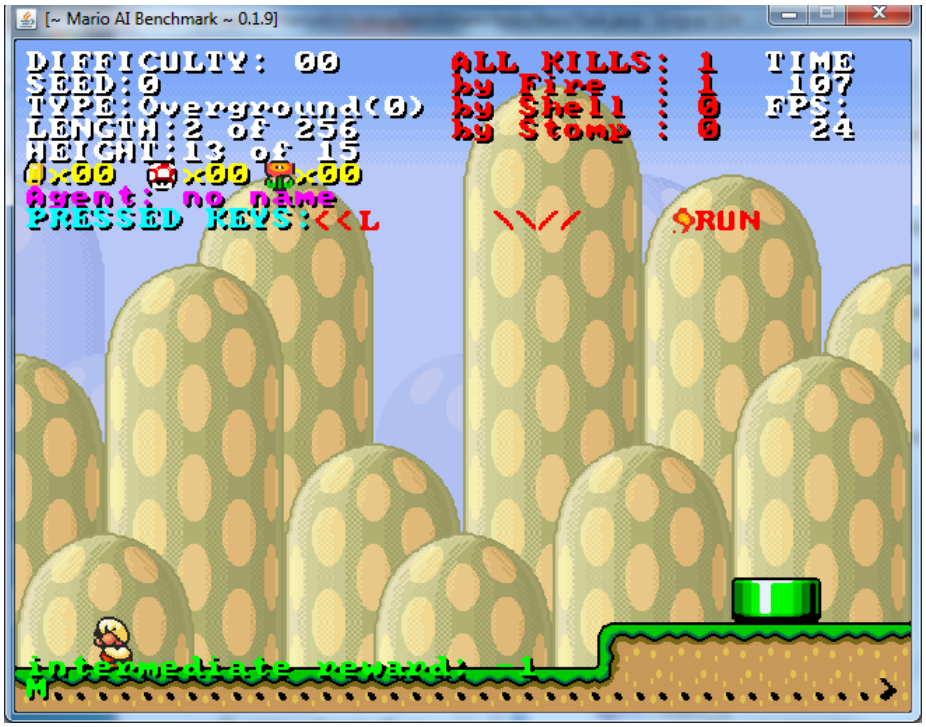

Figure 6.2: An individual maximizing the objective of taking as little damage as possible by crouching at the far end of the level

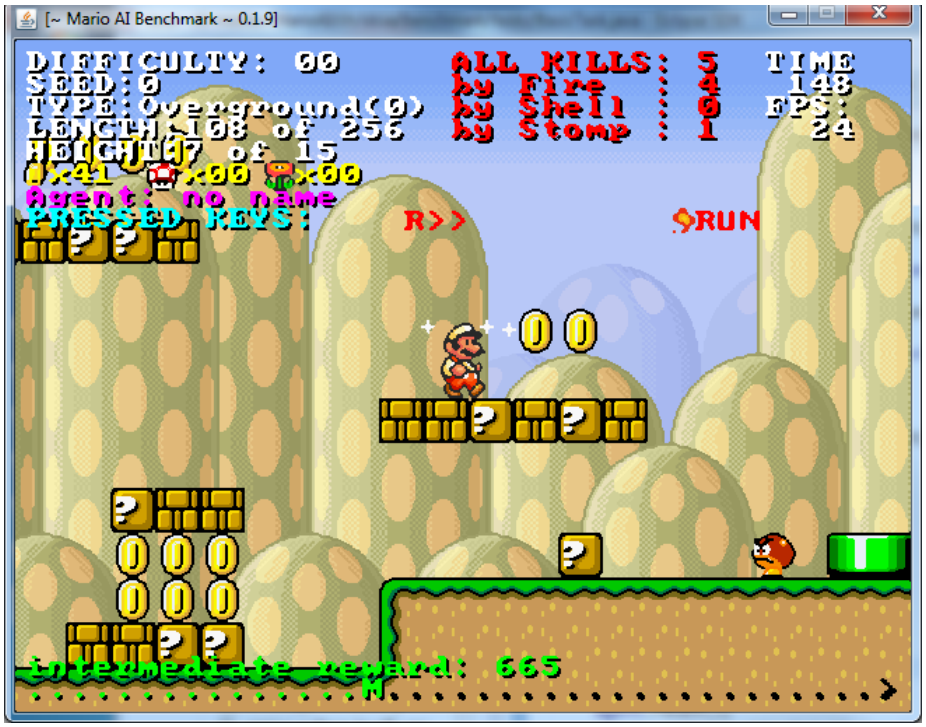

Figure 6.3: An individual jumping on top of a platform to collect the coins that are there

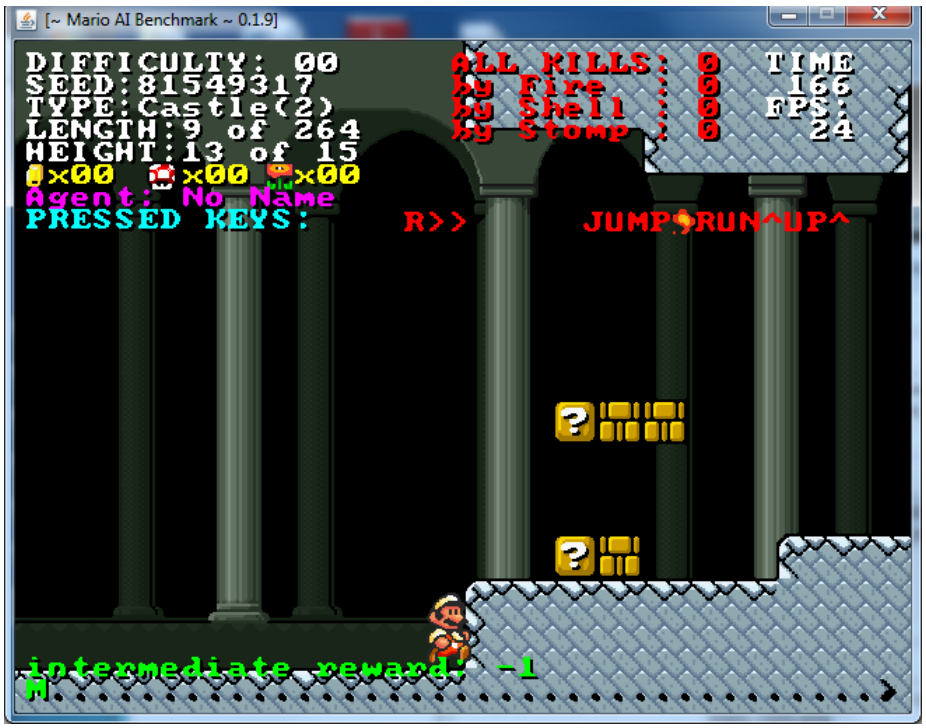

Figure 6.4: A controller stuck in one of the test levels. It keeps holding right and jump but is unable to jump because it was holding the jump button the last frame

		Distance	Jumps	Ducks	Shots
Coins	Training	**13800**	NA	NA	NA
	Testing	**6400**	NA	NA	NA
Jump	Training	**11840**	**146**	515	228
	Testing	**5690**	**155**	1629	470
Duck	Training	**11910**	114	**866**	95
	Testing	**5830**	153	**1956**	160
Shoot	Training	**11850**	143	326	**175**
	Testing	**6142**	184	985	**182**

Table 6.2: The average scores on the objectives by the best individual of each population. The objectives that were used in the different populations are marked in bold.

6.2 Experiment Two

As can be seen in Figure 6.5, on the training data all of the different behaviour changing objectives had a similar increase in distance traveled over the generations. None of them were however able to get better results than the controller not utilizing a behaviour changing objective. This is partially explained in Table 6.2 where we can see that the controllers that got the furthest in the populations with jumping and shooting as objectives, are not jumping or shooting drastically much more than the ones not maximizing these functions. This indicates that there is a cap on many times it is beneficial for the controllers to jump or shoot and that you do not need to have these behaviours as objective functions in order to get controllers that use these behaviours where they are needed. It is however interesting to note that none of the controllers that were the best at getting far were able to dominate the other populations on their behaviour changing objective. This could mean that the controllers are able to find out when to do these behaviours themselves without having them as a secondary objective, and that performing these actions when they are not needed ends up hurting the performance of the controller.

On the test levels none of these controllers were able to get better results than the controller not using a behaviour changing objective, but the controllers from the population where shooting was an objective were still significantly better than the benchmark used in the first experiment. This could stem from the fact that even if the controller spends a lot of time shooting, he can still move exactly like he would without shooting as the action doesn't have any impact on Mario's movement.

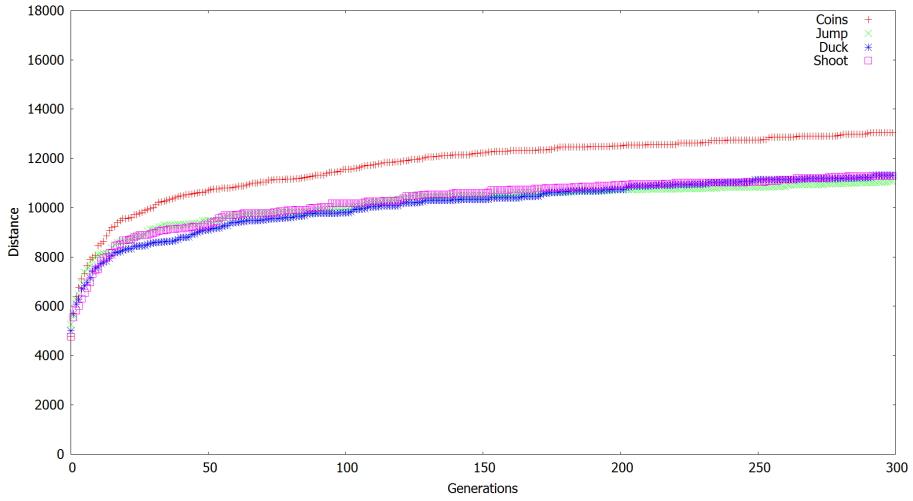

Figure 6.5: Average score at getting far of the best individual in each of the populations in experiment two.

		Distance	Coins collected
Benchmark	Training	**13700**	144
	Testing	**5700**	45
Coins	Training	**13800**	**152**
	Testing	**6400**	**50**
GA with one objective	Training	**11052**	107
	Testing	**5514**	48
GA with two objectives	Training	**11540**	**121**
	Testing	**5890**	**51**

Table 6.3: The average scores on the objectives by the best individual of each population. The objectives that were used in the different populations are marked in bold.

6.3 Experiment Three

In Figure 6.6 similarities can be seen between the two populations using NSGA-II and the two populations using the standard GA. In both cases the populations utilizing the objective of collecting coins in addittion to getting far get slightly higher distance scores than the ones not using the objective. This similarity continues on the test levels as can be seen in Table 6.3 where these controllers are stil able to get further than the others.

Even though the two sets of populations display similar characteristics, it is evident that the ones developed using the GA do not proffit as much as the MOEA by adding the extra objective. This could stem from the way pareto dominance and ranking works NSGA-II which ensures that the individuals of the population are as spread out as possible in the solution space, while a standard GA is more prone to getting stuck in local maximas and where the different objectives will have to be scaled properly in the fitness function in order for it to have any effect. Even so, as both populations were able to create more general controllers by utilizing this objective, the results indicate that incoroprating the objectives used in a MOEA also can improve the results of the individuals of a standard GA.

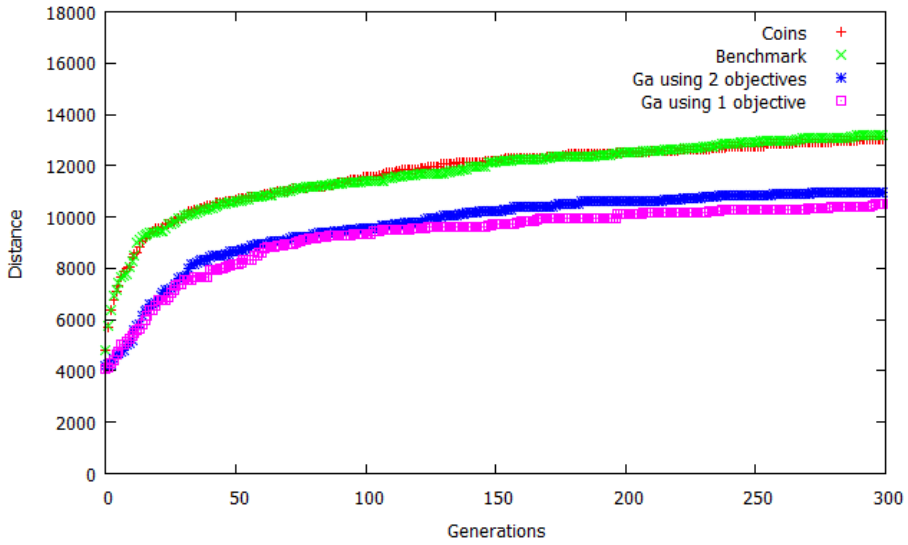

Figure 6.6: Average score at getting far of the best individual in each of the populations in experiment three.

		Distance	Kills	Damage Taken	Coins collected
Ensemble	Training	2320	4	16	6
	Testing	2320	4	16	6
Single	Training	21140	142	10	225
	Testing	21090	141	10	226
Multi-Modal	Training	26720	146	9	301
	Testing	16930	101	12	195

Table 6.4: The average scores on the objectives by the best individual of each population

6.4 Experiment Four

The results of this experiment can be seen in Table 6.4. It is apparent that the controllers created using an ensamble attain much worse results on all objectives than both the single individual of the pareto front and the multi-modal networks maximizing the main objective. By inspecting replays of the ensemble controllers it was found that they would stand still most of the time without pressing any buttons. This could be because of how the ensemble is structured, requiring the average value of each button to be above a certain threshold. This means that even though there might be a time period of only a few frames where all the individuals of the ensemble would have pressed the jump button, the ensemble might not press the button at all, as only some of the controllers would press any given button at the exact same frame.

The multi-modal controller was able to attain better results than the single controller on the training levels on all objectives. On the test levels however the multi-modal controller suffers a drop in performance comparable to the ones suffered by the other controllers in the previous experiments. Surprisingly the same is not the case with the single controller where the results hardly change between the training and test levels. Due to the way the levels were designed for this experiment with fewer elements in each respective level, there were fewer differences between them than there were in the previous experiment which would explain why the objective scores would not drop as much from the training to the test levels. The fact that the multi-modal controller still suffers such a big loss in objective values indicates that even though this network representation is better at maximizing several different objectives on training data, it is very vulnerable to overfitting and is thus less suited for creating general controllers.

Chapter 7

Evaluation and Conclusion

This chapter evaluates, discusses and concludes the results from the experiments.

7.1 Evaluation

This section discusses the results of the experiments of the thesis and how the research questions were answered.

7.1.1 Overfitting

A common denominator for all the experiments ran except for those from the multi-modal experiment was that the performance on almost all objectives had a huge drop from the training levels to the test levels. It could be argued that this stems from the fact that there were only 10 different training levels and that the controllers would overfit to these levels. In order to test whether or not this was the case we created a controller using the objectives of getting far and collecting coins on 100 different training levels in a run that took approximately 30 hours. This test showed that there was only a minor(5%) increase in the performance on the test levels while the performance on the training levels dropped by around 10-15%. This indicates that the different controllers would not have faired much better had a massive ammount of training levels been used instead.

7.1.2 Objectives

The first research question of the thesis was finding out how to define objectives for a MOEA in a domain without explicitly defined objectives. From the results of the first two experiments it became apparent that adding the game's sub goals as objectives could both increase and decrease the ability of the controllers to generalize depending on which objective was chosen. Using several objectives that alone create more general controllers were also found to create individuals with much worse objective scores than the controllers using these objectives separately. These results indicate that while it is possible to use sub-goals of a game as objectives for a MOEA, research still has to be put into figuring out which objective is beneficial for the MOEA, which isn't and if it is possible to use these objectives together. It was also discovered that adding behaviours as objectives in addition to sub-goals had a negative effect. This indicates that evolution is able to find these behaviours on its own given that these behaviours are beneficial to its other objectives. Adding basic behaviours as objectives then only makes it harder for the MOEA to progress on the other objectives when it has to keep several individuals that maximize superflous objectives instead of individuals that get good results on other objectives.

7.1.3 Objectives in a scalar fitness function

The second research question of the thesis was figuring out how incorporating objectives into a scalar fitness function would affect a GA compared to using them in a MOEA. In the third experiment of the thesis it was found that both populations that used a standard GA without the added objectives that were found to be beneficial in NSGA-II had poorer performance than one that had an extra objective incorporated into the fitness function. Even though populations created using a GA would display poorer results at the main objective of them game during evolution, it was the final controllers that used the extra objective from both the GA and the MOEA that had the best performance on both objectives on the test levels. This indicates that it is possible to incorporate objectives found to be beneficial for a MOEA into fitness functions for other types of GAs. Adding such objectives to a scalar fitness function however is not trivial as care has to be taken when scaling the different objectives in order to avoid one objective taking control of the entire fitness function.

7.1.4 Ensemble

The final research question asked how an ensemble could be used to combine individuals of a pareto front into a controller that is able to perform well on all objectives of the front. In order to answer this question it was decided to test creating an ensemble consisting of all the individuals of a pareto front in hopes that they would be able to cooperate in such a way that the ensemble would

be able to change behaviours according to which behaviour would be the most appropriate. In the 4th experiment, special training and test levels were used to test the controllers performance on three different types of levels where each type required different skills from the controllers in order to succeed. The results from the experiment showed that using an ensemble that uses voting created controllers that would hardly ever move and that were worse at every objective than both single individuals and multi-modal networks that were trained on the same levels. A problem with using ensembles as a controller is that even though all of the individuals of the ensemble would have issued a button press during a small number of frames, as long as the majority doesn't do so on the same frame, the ensemble will not issue the button press. This is an inherent problem of using ensembles in a domain where agents have to use combination of button presses over several time steps in order to make an action as it is very difficult to cooperate on pressing the exact same buttons on the exact same time steps over a longer period of time. This problem could be solved by processing the output of the networks used in this thesis instead of using a threshold to decide whether or not a certain button would be pressed based on the value of the corresponding output node. An example of this could be creating a set of actions where the output values of the network could be used to decide which action to take instead of six different button presses where the function of the button changes based on the state of the agent.

7.2 Conclusion

The main goal of the thesis was to explore how one could use MOEAs to solve problems that are not explicitly defined as multi-objective problems. In this thesis, a neuroevolution technique consisting of combining a multi-objective evolutionary algorithm called NSGA-II and NEAT-inspired ANNs were used to develop a system that created controllers for a version of the Super Mario Bros. game called Mario AI. This game was used as a test bed in order to answer the research goals and hypotheses of the thesis. Experiments were conducted to find ways to define objectives for MOEAs in Mario AI, how using these objectives as a basis for a scalar fitness function would affect a GA and to find a way to use ensembles to combine individuals of a pareto front into a single controller that would be able to display the strenghts of all of the individual controllers.

The results of the experiments showed that it is possible to use main and sub-goals of a game both as objectives for a MOEA and as the basis for a scalar fitness function to be used in a standard GA. It is however not trivial to find which goals to use, and even with the useage of the authors' expert knowledge of the domain, most of the chosen objectives were found to have a negative impact on the controllers. It was however discovered that rewarding basic behaviours that the controller has to use in order to play well were not needed in order to create controllers that were able to utilize these behaviours at the right times.

The results also showed that it is possible to create a controller using an ensemble of the individuals of a pareto front, but that care has to be taken when designing the controllers so that it is easy for the controllers to cooperate on finding which action to issue at every time-step. These results suggests that more research should be spent on creating a technique for finding objectives to be used to create agents for video games in order for it to be easier to create controllers that are able to do well on several aspects of problems such as these.

Chapter 8

Further Work

This chapter contains both the suggested improvements on the system used in this thesis and other techniques that could be used to reach the same goal as the one presented in this thesis.

8.1 Reducing input

The input of the ANNs used in the system created for this thesis consisted of a receptive field with a size of 61 blocks. Even though this receptive field was much more compact than the one used in the authors' specialization project, the size of the input layer greatly increases the difficulty of evolution to find good weights and topologies of the networks. This is not only due to the large size of the resulting network, but also due to the difficulty of finding relevant features from the input. This could be improved by processing the receptive field before feeding it to the network, only returning values such as the distance to the closest enemies, gaps and walls, as well as which powerup the agent currently has and how much time it has remaining. This would greatly reduce the size of the network and as these are all relevant features for assessing the current situation, it would also mean that each input node will always be of importance.

8.2 Reducing output

The controllers used in the system created for this thesis each output a set of button presses at each time-step. There are several combinations of button presses that result in the same action and several keypresses that cancel each other out, such as pressing left and right at the same time. Some actions such

as running also requires the controller to hold the run button over several continuous frames, and jumping requires the controller to release the jump button after landing in order for it to be able to jump again. This makes it hard for evolution to figure out what combination of key-presses causes the agent to perform a given action. It is possible to instead create a set of actions the agent is able to perform where some actions could go over several time steps. This would not only make it easier for evolution to figure out when to perform each action, but it would also make it easier to combine individuals into an ensemble where it is feasible for the different individuals to cooperate on selecting a single action that is beneficial at the given time.

8.3 Other domains

The game of Super Mario Bros. is a domain with multiple ways of reaching the end of each level, several different types of terrain and enemies with different properties, making it a very complex domain for evolutionary algorithms. The techniques detailed in this paper should be tested on less complex video games with simpler controls and fewer obstacles. This would make it easier to compare the results of using different objectives as the agents would have an easier time learning to play the game and interacting with the world around them, making them less likely to get stuck and not being able to complete their other objectives.

8.4 Multi-Modal Networks

In the final experiment of the thesis, ensembles were compared to multi-modal networks. These networks showed promising results on the training data, displaying much better scores on all objectives of the training data than both the ensemble and single individuals from the pareto front consisting of the NEAT-inspired networks used in the other experiments. As these networks are specificly designed to display several different behaviours, it would be interesting to further explore the use of these networks to find objectives to be used for MOEAs in other games or problems that are not explicitly stated as multi-objective problems with obviously contradicting objectives.

Bibliography

[1] Benoit Chaperot and Colin Fyfe. Improving artificial intelligence in a motocross game. In *Computational Intelligence and Games, 2006 IEEE Symposium on*, pages 181 –186, may 2006. doi: 10.1109/CIG.2006.311698.

[2] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, 2002. ISSN 1089-778X. doi: 10.1109/4235.996017.

[3] Jin-Hyuk Hong and Sung-Bae Cho. Evolution of emergent behaviors for shooting game characters in robocode. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 1, pages 634 – 638 Vol.1, june 2004. doi: 10.1109/CEC.2004.1330917.

[4] Su-Hyung Jang, Jong-Won Yoon, and Sung-Bae Cho. Optimal strategy selection of non-player character on real time strategy game using a speciated evolutionary algorithm. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 75 –79, sept. 2009. doi: 10.1109/CIG.2009.5286490.

[5] Jacob Schrum and Risto Miikkulainen. Constructing complex npc behavior via multi-objective neuroevolution. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2008)*, pages 108–113, Stanford, California, 2008. URL `http://nn.cs.utexas.edu/?schrum:aiide08`.

[6] Jacob Schrum and Risto Miikkulainen. Evolving multi-modal behavior in npcs. In *IEEE Symposium on Computational Intelligence and Games (CIG 2009)*, pages 325–332, Milan, Italy, September 2009. URL `http://nn.cs.utexas.edu/?schrum:cig09`. (Best Student Paper Award).

[7] Jacob Schrum and Risto Miikkulainen. Evolving multimodal networks for multitask games. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):94–111, June 2012. URL `http://nn.cs.utexas.edu/?schrum:tciaig12`.

[8] Sergey Karakovskiy, Julian Togelius. The mario ai benchmark and competitions. 2012.

[9] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. URL `http://nn.cs.utexas.edu/?stanley:ec02`.

[10] Tse Guan Tan, J. Teo, P. Anthony, and Jia Hui Ong. Neural network ensembles for video game ai using evolutionary multi-objective optimization. In *Hybrid Intelligent Systems (HIS), 2011 11th International Conference on*, pages 605 –610, dec. 2011. doi: 10.1109/HIS.2011.6122174.

[11] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87 (9):1423–1447, sept. 1999.