DEPARTMENT OF INFORMATION SCIENCE AND MEDIA STUDIES

MASTER THESIS

Using Genetic Programming for Agents in Non-Deterministic Games

Bjarte Johansen

August 1, 2012

Contents

Abstract

The aim of this thesis is to explore the possibility of using Genetic Programming to create agents that play non-deterministic games at a human level. The game that is used to test the hypothesis is Ms. Pac-Man; an old arcade game. The reason for choosing this game is that there is a competition whose object it is to create an agent for the game only using the visual data of the game. This allows me to compare my results against human players and, others who have been trying to solve the same problem. The best score a human has achieved is 921,360 points by Abdner Ashman in 2005, while the agents in the competition have only received a score of 36,280 points.

A modified emulator is used to run the original binary of the game. The modifications are made to allow the agents easier access to key properties of the game. They also make it possible to run the game as a background process, which allows more than one emulator to run in parallel over multiple machines. This makes it easier to run larger populations over many generations.

The ideas around how to solve the problem sees large changes during the development of the project. A path-finding algorithm, A*, is introduced to decrease the search space for the GP Library. The algorithm is modified by the agents to control where to move.

Even though the results from the thesis are not as good as hoped, we do see agents that perform near competition level. The late discovery of a bug in the game-API made progress difficult. Especially since a large amount of time was used on different solutions to the problem that proved ineffective. However, the thesis does show that it is possible to make agents, through Genetic Programming, that improves when playing non-deterministic games. Further research is needed to show if this approach may prove better than humans at playing Ms. Pac-Man.

Acknowledgments

I would like to thank my supervisor Weiqin Chen for excellent help during the work on the thesis and for not giving up on me when motivation was low. I would also like to thank my fellow master students for listening and discussing with me when I was stuck on a topic and, for keeping me sane with distraction when I needed it.

Chapter 1

Introduction

In 2007 a competition for creating agents that play Ms. Pac-Man started at the "IEEE Congress on Evolutionary Computation 2007". The goal was to beat the standing world record (for humans) of 921,360 points made by Abdner Ashman in 2005. No machine has beaten that score yet; only receiving 36,280 points in the game.

Ms. Pac-Man is a difficult game, even for most human players. Nobody even knows if it is possible to make a software agent that can beat the current standing high score, though, I believe that it this is highly likely.

In this thesis I will be looking at one approach to solve the problem; namely through Genetic Programming. I will also use some other algorithms and methods to aid the Genetic Programming in making agents for the problem, as the A*-algorithm used for search in graphs.

However, the focus of this thesis is not Ms. Pac-Man, it is to see if the use of Genetic Programming in "non-deterministic"[1] games is a viable strategy. The reason for using Ms. Pac-Man as a test bed for this problem is that it is a small enough problem domain with enough variation that it becomes an interesting problem to work on.

What is meant by non-determinism is randomness. Or more specifically, that there are things that can not be predicted. Not that all things can not be predicted. If that was the case then the game would not be interesting as there would not

[1]Non-deterministic is written in quotes as the non-deterministic behavior is dependent on how random the pseudo-random number generator is.

exist a strategy that would be better then any other. Another word for this would be “stochastic”. Problems that follow rules but at the same time allows for some random choice will require that the agents are dynamic enough to follow the changes that occur. This is something that humans are good at, but that machines still lack.

1.1 Motivation

The motivation for the research is to explore how modern Machine Learning and AI algorithms can compete at the same level as humans in non-deterministic games. It also presents itself as a good opportunity to learn more about how to use this type of algorithm in a real and largely unexplored space, and maybe discover something new along the way.

That is not the only reason for doing this as it also is an excellent opportunity to experiment with distributed parallel processing. Which is interesting in itself as processors come with more and more cores and we need to distribute the work we do over more machines to complete heavy calculations within a reasonable amount of time. As the web page for the Ms. Pac-Man competition states:

> This is a great challenge for computational intelligence and machine learning, and AI in general.

This project will aim to show, through Genetic Programming, that modern AI algorithms can be as good as humans on non-deterministic problems and that computers are not restricted to deterministic problem spaces like chess and checkers where it is possible to calculate every possible move (though impractical).

The approach for the project also has the opportunity to be transferred into other similar problem spaces. An example could be the “pacman-vs-ghosts” competition where the aim is to create an agent for Pac-Man and another team makes agents for the ghosts. The Pac-Man agent then tries to get the most points, while the ghosts try to stop Pac-Man from getting any points. More information can be found here http://www.pacman-vs-ghosts.net/.

1.2 Organization of the thesis

The thesis is divided into 6 distinct chapters. Each chapter will present a different topic. The first chapter is the introduction; this chapter. It describes the motivation behind the research and why it has been done.

The next chapter talks about the domain of the thesis, namely Ms. Pac-Man. It will discuss what Ms. Pac-Man is, what the problems in Ms. Pac-Man are, and what others people have done in the same area.

The third chapter will introduce the theory that the thesis builds on. It gives an overview of what Genetic Programming is and the different topics within Genetic Programming. It will also give a quick overview of some of the other algorithms and techniques that have been used during or in the development of the project.

The fourth chapter will show how the system has been design and will go in depth into how the development has progressed through the different iterations of the project. It will also describe the different tools that have been used. Lastly, it will show some samples of the code that have been created for the project.

In the fifth chapter there will be an evaluation of the different parts of the system according to the criteria that was set in the development stage of the project and some flaws in the design will be highlighted. It will also present some of the experiments that have been conducted in the different iterations of the project.

The last chapter will give a brief summary of what has been done before it explains what conclusions can be drawn from the research. Then, at the end, it will look at what further research can be done to improve and expand the research presented in the thesis.

Chapter 2

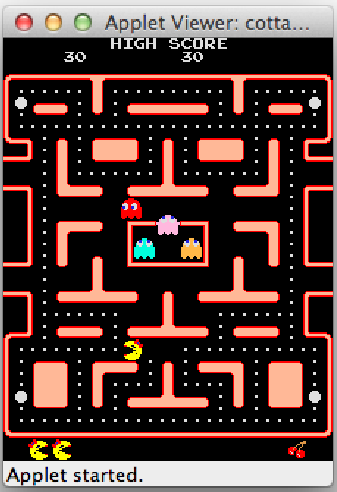
Ms. Pac-Man

Ms. Pac-Man is an arcade game released by Midway in 1982 and is a modification of another popular game, Pac-Man, made by Toru Iwatani for the Namco Company in 1980. The original game is completely deterministic and can be cleared with a perfect route. While in Ms. Pac-Man the ghosts will sometimes randomly change direction. This makes it a lot more interesting to create agents for the game, as the agents have to decide what path it should take instead of just finding the optimum path. The general game play is well described by Lucas (2005):

> The player starts with three lives, and a single extra life is awarded at 10,000 points. While it is never a good idea to sacrifice a life, it may be better to take more risks when there are lives to spare. There are 220 food pills, each worth 10 points. There are 4 Power Pills[1], each worth 50 points. The score for eating each ghost in succession immediately after a power pill starts at 200 and doubles each time. So, an optimally consumed power pill is worth 3,050 (= 50 + 200 + 400 + 800 + 1,600). Note that if a second power pill is consumed while some ghosts remain edible from the first power pill consumption, then the ghost score is reset to 200. Additionally, various types of fruit appear during the levels, with the value of the fruit increasing with each level. The fruit on level one is worth only 100 points, but this increases to many thousands of points in higher levels. It is not necessary to eat

[1]This thesis uses the term super pill.

(a) The start screen of Ms. Pac-Man



(b) A game of Ms. Pac-Man

any fruit in order to clear a level.

The interesting thing about this game compared to other versions of it, or other arcade games, is that the agents can not just find an optimal path that can be followed to clear the map, like in Pac-Man. The path of the ghosts move in a non-deterministic way and the agent has to be highly dynamic to be able to avoid the ghosts to receive a high score in the game.

In contrast to what some other researchers have done when making agent that play games like this, and especially the others who have been researching the same as me, I will not make my own clone of the game (Lucas, 2005, Robles and Lucas, 2009). I will be using an emulator and using the old arcade binaries. I will in other words play the original game and not a clone that might not behave in the same way as the original game.

2.1 Research on the Ms. Pac-Man Problem

The Ms. Pac-Man problem is superficially very simple, but under the surface it becomes incredibly complex. This complexity stems from the non-determinism of the game.

There are some papers that are interesting for the work done in this thesis and this section will be looking at some of them. Even though none of these papers use Genetic Programming in their attempt to solve the problem they are still interesting to look at to find ideas for how to represent the problem and to get some other perspectives on the problem.

2.1.1 Evolving a Neural Network Location Evaluator to Play Ms. Pac-Man

This paper, by Lucas (2005), is probably the closest paper to the topic of this thesis. It is also using evolution to evolve agents for Ms. Pac-Man. The difference though is that this paper evolves Neural Networks instead of computer code. Another difference is that the game that is used in the experiments is only an approximation of the original game and is implemented by the researcher. Though, some of the differences make the game harder than the original, the behavior of the ghosts are very different from the original behavior in the game. The way the agent handles control over Ms. Pac-Man is that it evaluates all the possible next locations, given the current node. The control algorithm will then move into the node that receives the best score. It uses a feature vector designed by the author, but says it would be better if it was possible to use all of the features in the game to control where to move. This vector is input to the neural network. The neural network was then evolved over generations where the overall objective was to evolve the best agent possible. The quality of play was measured by averaging the score over a significant number of games (e.g. 100). The results from the research is a bit disheartening to see as they were only able to perform at the level of a "reasonable novice human." Around 4,500 points. Especially because this paper uses an approach that is somewhat similar to the one in this thesis.

2.1.2 Learning to play using low-complexity rule-based policies: illustrations through Ms. Pac-Man

The method used in this paper, by Szita and Lõrincz (2007), is reinforcement learning. They make the argument that Ms. Pac-Man meets all the criteria of a reinforcement learning task. They say that

- The agent has to make a sequence of decisions that depend on its observation.
- The environment is stochastic.
- There is a well-defined reward function.
- Actions influence the rewards to be collected in the future.

The controller of the game is encoded by a compact decision list. The components of the list is selected from a large pool of rules. The rules are either hand crafted or automatically generated. A selection is learned by using a method called cross-entropy, a global optimization algorithm.

The game that is used in this paper is also implemented by the researchers. The behavior of the ghosts in this implementation is different from the original ghosts. In this version the ghosts will move towards Ms. Pac-Man 80% of the time and move randomly the other 20% of the time.

The results from this paper shows better results than the previous paper and manages a score of around 9,000 points. The problem however is that there is a significant difference in the implementation of the game, and that could have a lot to say. This is not a problem for the paper though as the score in Ms. Pac-Man is not the end goal of the research. The interesting part for the research in this paper is however the different rules that are used in the controller for Ms. Pac-Man. It is interesting to see what they are focusing on and that they also focus on the immediate surroundings of Ms. Pac-Man.

2.1.3 Pac-mAnt: Optimization Based on Ant Colonies Applied to Developing an Agent for Ms. Pac-Man

In this paper, by Emilio et al. (2010), an optimization algorithm based on ant colonies is used to develop competitive agents for Ms. Pac-Man. A genetic algorithm is then implemented to optimize the parameters for the artificial ants.

The aim of the paper is to verify if Ant Colony Optimization can be used to develop competitive agents for real-time video games. The paper also uses its own implementation of Ms. Pac-Man, but also tests it against the original game.

The way the controller works is that it tries to optimize the score it can achieve in any given situation. It calculates the score by defining 2 types of ants. The first is the *collector ant*. This type of ant is used for collecting points. The second type is the explorer ant. It tries to find safe paths to prevent Ms. Pac-Man from being caught. The ant colony has a wide range of parameters that the genetic algorithm can optimize. Some of the parameters include the size of the colony, and the maximum and minimum distance the ants will travel.

The system presented in the paper shows interesting results. It is much higher than the previous papers and can show the best score of 20,850 points. The interesting part in this paper is that a genetic algorithm was implemented and showed that it is possible to use evolutionary methods to get higher score than what has been achieved earlier.

2.1.4 Ms. Pac-Man competition results

To get a better understanding of the difficulty of the Ms. Pac-Man problem we will look at the performance of the agents at the first and latest competition event.

We can see the first results in table 2.2. The results from the first competition was rather poor with the maximum score of 3,810 points made by the default agent. Though, one of the contestants could not get their agent to run at the event, but would later manage to get over 17,000 points.

In the latest competition in 2011 we can see from table 2.2 that there is a significant improvement from the first time the competition was held. Now the highest scoring agent is over 36,000 points. That is almost 10 times as much as the

	Default	Wirth	Handa	Elno
	1,330	1,120	1,000	650
	230	820	1,300	1,040
	1,940	780	2,170	1,310
	2,390	1,250	1,760	1,790
	2,990	3,370	1,880	1,360
	2,060	1,700	1,310	680
	3,810	1,490	2,270	620
	3,140	1,990	2,270	620
	3,149	1,990	2,210	1,830
	1,010	1,380	1,700	1,370
	1,990	2,830	1,910	1,160
Max	**3,810**	**1,673**	**2,270**	**1,181**
Mean	**2,269**	**3,370**	**1,751**	**1,830**

Table 2.1: Results from the 2007 competition (Lucas, 2007).

best scoring agent in the first competition. The score is however a far cry from the best a human has achieved.

Something to note is that all of the agents have wild differences in how they score in different runs. I think this really shows us the effect the stochastic nature of the game has on the performance of the agents.

	Bruce	Kyong-Joong	Nozomu	Ruck	TsungChe
Prior 10 runs	12,180	5,070	23,870	12,290	6,350
	13,700	8,300	20,150	19,330	8,420
	12,640	**19,900**	30,200	21,250	7,770
	3,240	6,560	**36,280**	23,690	16,790
	7,570	9,020	21,410	13,660	**20,300**
	5,370	13,900	16,830	9,110	13,880
	7,960	8,549	32,310	15,760	7,310
	12,180	7,690	20,640	24,060	8,350
	3,880	16,600	31,940	25,420	14,520
	6,790	12,750	24,580	15,730	19,810
Live session runs	6,800	*(15,760)*	24,460	21,860	*(12,840)*
	5,930	*(12,360)*	16,130	**27,240**	*(4,920)*
	5,180	12,380	18,530	5,440	5,710
Max	**13,700**	**19,900**	**36,280**	**27,240**	**20,300**
Mean	**7,955**	**11,448**	**24,341**	**18,065**	**11,305**

Note: Entries in parentheses was run after the live event.

Table 2.2: Results from the 2011 competition (Lucas, 2011a).

Chapter 3

Genetic Programming and related concepts

This chapter discusses the theories and literature that the research relies on. It will present an extensive review of Genetic Programming. In addition it will also explain some of the problems that may occur when using Genetic Programming to solve computational problems. It will also discuss the other algorithms and techniques that were used in the thesis.

3.1 Genetic Programming

Genetic Programming is a method that uses the mechanisms of evolution to *evolve* code into a form that is *good enough* as a solution for the problem at hand. It is not expected to produce a perfect solution, but an approximation towards it. This can be handy in situations were one does not know what a prefect solution is. As an approximation of natural evolution on computer programs the method is part of a larger set of algorithms that are called Evolutionary Computation or Evolutionary Algorithms.

Before continuing, to make it easier to follow the text onward, it would be helpful to define a few terms. First, there should be a distinction between the terms "code" and "program". From here on out I will use the term "program" as the instructions an individual created by the Genetic Programming Library consists of. Even tough I could use the term individual, I feel it is a way to disambiguate between the instruction set defining the program in an individual and the full individual defined by its program, fitness and other data.

This does however create a new problem when addressing the instructions the system consists of. To resolve this I will be using the term "code" when discussing these parts of the system, but, to keep confusion to a minimum, I will try to use the name of the part or address it as the system or otherwise try to avoid using the term "code" when possible.

By solution, in this context, it is meant a means of solving a problem, not a correct answer. This thesis is not looking for correct answers, and neither is Genetic Programming. I want to find approximations to *good enough* solutions.

Even though these two things are the same, both are instructions for the machine to perform, I think it will be easier to understand the overall text if we differentiate between them as it will no longer be needed to figure out what is being discussed from the context.

3.1.1 Representation

The most common representation for Genetic Programming is a syntax-tree where a function is a node and variables and constants are terminals (Poli et al., 2008a). To create the programs GP uses a set of primitives, which is a combination of predefined functions and terminals.

Function set

The function set is the set of predefined functions that the genetic programming library is allowed to use during creation and mutation of the individuals in the population (Koza, 1992). What this set consists of is dependent on the problem at hand. A function set to approximate the function $x^2 + 3$ might be $\{*, +\}$. Observe that the constants and the x-variable is not in the function set. This is because the constants and variables are not part of the function set.

Terminal set

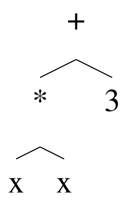
The terminal set is the set of terminals, or leaf nodes, for the individuals. These are there to ensure that the functions have access to values for their arguments. Terminals can be constants or variables (Poli et al., 2008a). If they are variables they are usually there to be used by the fitness function to test the program or meant as the input of the function being created. To represent the terminal set in the example used earlier in explaining the function set, a valid terminal set could be $\{1...5, x\}$. As we can see here all the numbers between and including 1 and 5 are in the set, but if we look at the problem it only contains 2 and 3, and we could have just put them in there, but this is to highlight the fact that the reason for using GP is that we are unsure of the best solution to a problem and need something that works. So usually when working with GP one will be working without knowing what is best, therefore one should put a range of variables and constants in the set to ensure that it covers the basis of the problem.

Sufficiency

It is important that the representation of the individuals together with the primitive set is sufficient for the problem at hand. By sufficiency it is meant that the representation is capable of expressing most, if not all, of the solutions to the problem (Langdon et al., 2008). If it is not able to do that the Genetic Programming Library will not find a satisfactory solution. Proving sufficiency can be very difficult, especially if the problem is poorly understood. It is not strictly necessary to prove that the primitive set is sufficient, but one should at least have a general idea if it is the case or not.

Trees

The most popular way of representing programs in Genetic Programming is to represent them as syntax-trees. This is because it is both easier for the researcher to reason about and is easily machine-readable (Back et al., 1997). Looking again at the previous example, the tree-representation of the solution to the problem could look something like this:



Normally this would be represented as a list inside the Genetic Programming Library and could look like this:

```
(+ (* x x) 3)
```

As we can see, this looks unmistakably like lisp code. This makes lisp an excellent candidate for Genetic Programming and many libraries that already exists for Genetic Programming are written in a lisp (see: Koza (1992)).

Linear

Linear representation of programs in Genetic Programming has also been used as it is closer representation of imperative programming and how many of the most popular programming languages works (Brameier and Banzhaf, 2007). Again, the same example as before can be represented as follows if one wants to use this type of representation:

$R_0 := I_0 * I_0$;
$O_0 := R_0 + 3$;

Here the representation is in a C-like syntax where R_0 represent a registry address, I is the input registry and O is the output registry. This representation is closer to the imperative programming paradigm and might be easier to comprehend and use than the tree-based representation for some users.

3.1.2 Initialization

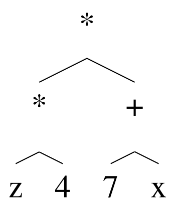
Initialization of a new population can happen in several ways, but the most common is random creation of new individuals (Poli et al., 2008a). The reason for this is that one does not always know what a good solution is and a random population is as good as any guess. Another way of initialization would be to prepare a population by hand based on the functions provided in the function set.

The two simplest ways of generating the population is to use either the methods called grow or full. We will be discussing these two and a third, very popular one, called *ramped half-and-half*.

Full

In the full-method the tree is created by starting at the tree's root node and creating nodes from the function set until the tree is at full depth (Poli et al., 2008a). When full depth is reached only terminals can be chosen. The reason for the name "full" is that it creates trees where every node is filled to the full starting depth.

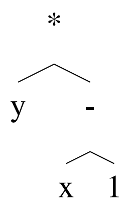
Using the primitive set $\{+,-,*,1...5,x,y,z\}$ and the depth 2 a sample tree could be:



Grow

The grow method allows for more shapes than the full method. Instead of selecting nodes from the function set it selects its nodes from the full set of primitives. But as with the full-method, it can only select terminals when the specified depth is reached (Poli et al., 2008a).

Using the same primitive set and depth as in the full-method *grow* could produce a tree like this:



As we can see, this tree is also expanded to the specified depth, but if we look at the left leaf of the top node we can see that it is a terminal while the right node is a function. When the right node reached the specified depth we can see that it also only contains terminals, as with the full method.

Ramped half-and-half

Ramped half-and-half is a method proposed by Koza (1992). It is a combination of the two previous methods where half of the population uses *full* and the other half uses *grow*. It is also normally used with a range of depth limits. This is to ensure that a large pool of tree of various shapes and sizes are generated for the initial population.

3.1.3 Fitness Evaluation

The fitness evaluation is the part of the program that determines how well a single program is performing. There has to be a way to evaluate the programs and this is the function that does that. Usually, this happens by deciding a metric on how to evaluate the performance of a solution. There are two ways of doing this depending on what the goal is.

If the goal is to achieve an approximation of a value, e.g, one tries to find a better compression algorithm for an image, then the interesting value is not the bit-string representing the image but the difference between the original and the compressed version. In this instance we can make the metric of the fitness function the absolute difference between the image produced by the program and the image from the known algorithm (Back et al., 1997). In other words we would like a value as close to 0 as possible.

If on the other hand the goal is to do as good as possible when playing a game we may want to choose the high-score of the game as the metric for the fitness function. In this case the best individual is the one with the highest score in the game. This is a type of competitive fitness score that fits well in problems that are naturally competitive and where it is difficult to compute an absolute metric for the individuals (Back et al., 1997).

In the end the way the fitness score is evaluated depends heavily on the task at hand. The fitness function is also the part of GP that drives how the population evolves. It is therefore crucial to create a fitness function that can put a metric to how good a program performs the given task (Koza, 1992).

3.1.4 Selection

Selection happens after the fitness function has evaluated the individuals within the current population. Selection then sorts and selects individuals that are acceptable for inclusion in the new generation. There are more than one way of doing this selection and we will discuss some of them.

Rank-based selection

Rank-based selection is normally not used as the *only* selection form if it is used at all. What rank-based selection does is that it selects a predefined percentage of the best of the previous population to be used for generation of the new generation (Back et al., 1997). It is however often used as a supplement to the other selection methods in that a small part of the best of the previous population is included in the new generation. In conjunction with other selection strategies this type of selection is often called elitism. Elitism should be used sparingly, but can prove to be beneficial to the overall health of the population (Poli et al., 2008b). In this kind of situation the selected individuals are usually not used in the recombination step, but instead just copied into the next generation.

Why rank-based selection is not used as the sole selection method is because it will most likely lead to a rapid loss in diversity for the population. The best of the population will very quickly take over the population and squeeze out all variety. This makes it so that further advancement comes at a much higher cost than with other selection methods.

Fitness proportionate selection

A popular form of selection is fitness-proportionate selection. It is popular because it is easy to implement and understand. It also usually gives satisfactory results. The problem, however, is that it can be slow on large populations.

The way it works is that it calculates the chance of selecting an individual from the population based on the proportion of the total fitness of the population the individual holds (Holland, 1975). Or in mathematical terms:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

Where p_i is the probability that the i-th individual is selected, N is the amount of individuals and f is the fitness of an individual.

Tournament selection

Tournament selection is another popular form of selection. It is usually much faster than fitness proportionate selection because it does not have to calculate the probability of selection for every individual every generations.

Instead, tournament selection selects n individuals randomly from the population and then selects the best individual to be used for the new generation (Blickle and Thiele, 1995). The following pseudo code shows how it is implemented:

```
tournament($t_{size}$, $population$):
    for $i \rightarrow 1$ to $t_{size}$ do:
        $a'_i \rightarrow$ chose individual with best fitness from $t_{size}$ randomly
            chosen individuals from $population$.
    return $\{a'_1, ..., a'_{t_{size}}\}$
```

Where t_{size} is the size of the tournaments.

3.1.5 Recombination

Recombination is the name for the function that handles the creation of new individuals for the next generation. There are three main ways that recombination can happen; mutation, reproduction and crossover. Reproduction is merely taking an individual from the population and copying it into the next generation. Mutation and crossover however are a bit more advanced and the next sections will therefore explain them with more depth.

Mutation

Mutation is usually a rare occurrence but can be really beneficial when it happens. This is because it has the potential to introduce new bits of code that the population doesn't already contain. This also makes it helpful in preventing a homogeneous population.

There are 3 main ways for mutation to happen: Replacing, removing and constant mutation. The names of the mutation types are rather self-explanatory, but there some things that might not be that intuitive so we will still describe the different types.

A replacing mutation is a mutation that *replaces* a part of the program with a new that is (usually) generated from the function and terminal sets (McKay et al., 1995). The generation happens much in the same way the generation of new individuals, but with more restrictions like shorter depth and width.

A removing mutation will *remove* something from the program. This depends a bit on the structure of the representation of the program. E.g., in a tree representation the mutation can remove a node from any point in the tree. One can of course set restrictions on how a removing mutation behaves as well (Angeline, 1997).

A constant mutation is a bit different from the rest of the mutation operators. It will instead of only mutating an individual, affect all individuals in the new generation that contains that constant. What happens is that a constant is chosen and changed either randomly or by design (Schoenauer et al., 1996).

These are the general mutation operator that exists, though there are other specializations of these operators.

Mutation is not necessary to have a working GP process, as shown by Koza in Koza (1992) and Koza (1994), but it can be advantageous to include low rates of mutation (Koza et al., 1996).

Crossover

Another type of recombination is crossover. This type of recombination is there to represent how sexual reproduction happens in biological processes. Crossover is done by exchanging nodes in two or more individuals (Poli et al., 2008a). The selection of which nodes to exchange depends on the restrictions on how programs can be put together. It is also possible to put other restrictions on the crossover function. The easiest way to implement crossover is to select a random node in each parent. The nodes are then exchanged, putting the selected node in place of a node in the other parent. This type of crossover is called uniform crossover (Poli and Langdon, 1998). There are other types of crossover that are more specialized on were the crossover happens within the parents.

What one has to be aware of when doing crossover is the types of input the different functions can receive. The danger her is that a crossover might replace a node with a node that returns something that the parent can't handle. This can lead to type errors. E.g., if a node that returns a number, like +, is replaced with a node that returns a list of numbers, like `(map inc [1 2 3])` and the parent node is $\star$ then, when the parent node is evaluated, it will throw an error that states that a list cannot be thrown to a number and stop working.

It is important to be aware of this type of problem when building the crossover selector, and to some extent, the primitive set.

3.1.6 Problem areas

Other than the problems already discussed there are some aspects of Genetic Programming that tend to be problematic. We will discuss the most prevalent ones here.

3.1.7 Bloat

Bloat is a common problem within Genetic Programming. It is also difficult to prevent as there are many different aspects of Genetic Programming that in synergy can affect the emergence of bloat in the programs, but there are some areas that might be more worth to investigate than others.

Especially in the definition of the problem domain and how the GP Library builds the programs and how it combines them into new ones. It is here possible to define restrictions on the way the program grows and how it evolves. But the easiest area to control bloat is by introducing new selection criteria based on not just the fitness score the individuals receive. E.g., one could look at the size of the programs and give the programs that are smaller but the same fitness a bigger advantage than the larger programs. This is called the *parsimony pressure* method (Koza, 1992).

There are plenty of other ways to control bloat in the population both by creating mutation operators that ensure smaller children and putting size and depth limits to how large an individual can grow (Poli et al., 2008a), but the measures that need to be taken depends on the project and the specific problems that occur.

3.1.8 Diversity

Ensuring diversity in the programs can be a real challenge. Even though there are ways to detect homogeneity in the population, it is difficult to decide on a good solution to the problem that doesn't introduce new problems itself.

It is possible to estimate diversity by looking at the variety of the population. If the number of distinct individuals falls below an acceptable threshold (usually around 90%) there might be problems with diversity in the population. However, a high variety is not necessarily indicative of a diverse population. The reason for this is that there might be programs that look different, but mainly do the same thing (Poli et al., 2008a). Also known as introns.

3.2 Other research

Here we will be looking at some of the different sources for the theory behind Genetic Programming and other concepts that is used in the thesis; like the A*-search algorithm.

3.2.1 A*-search algorithm

A* is a heuristics search algorithm that is excellent for finding paths in graphs. Invented at the Stanford Research Institute (Hart et al., 1968), it is an extension of Edsger Dijkstra's 1959 algorithm (Dijkstra, 1959). It tries to find the shortest path through a graph by looking one node on the shortest path it has found for now and choosing the node that has the shortest distance towards the goal. This strategy ensures that A* will always find the shortest path through the graph.

3.2.2 Memoization

Memoization is a technique for optimizing pieces of code. It does this by limiting the amount of repeat calculations that can happen in a computer program. This is done by storing the calculations it does, and then the next time the function needs to do the same calculation for the same input it will use a look-up table to retrieve the previously calculated value from memory (Michie, 1968).

Memoization can easily be done automatically as long as the function that it is done for does not affect state elsewhere in the program and has no side effects (Norvig, 1991). It is especially efficient on recursive code that generally has a large potential of recalculating previous results. An often used example of memoization is the calculation of the Fibonacci-numbers, which, when done naively, will calculate all past Fibonacci-numbers for every Fibonacci-number calculated.

The problem with memoization though is that it will use a lot more space than what the function would do otherwise. On machines with plenty of memory this is generally a good trade of, but if one is willing to do this depends on the magnitude of memory the memoized version of the function needs versus the speeds gains received.

3.2.3 Socket programming

Sockets are a tool for enabling programs to communicate over computer networks. The way it works is that a server listens to a socket on its end of the line waiting for a connection from another machine. A client will then connect to that machines socket and if it receives a reply that the connection is open and ready it will send a message. The client will then usually wait for a response from the server. The server will take the message from the client and either do some calculations on the message or pass it a long to another machine. After the server has finish the calculation it will respond back to the client. After the client has received the response it needs, the connection between the two machines are usually closed by the client (Stevens et al., 2004).

Chapter 4

Design and Development

The topic for this chapter is the design and development of the system. The chapter will go over the requirements, the design and will show a diagram over the system. The development of the system has required the use of a lot of different tools and the chapter will therefore also be presenting these as well.

There are 3 main parts of the system, the Genetic Programming Library (GP Library), the Emulator and, the Distribution Framework, each with their own sub-parts. I will be trying to explain in detail what each of these parts do and how they interconnect with each other.

4.1 Requirements

It is important to know what is needed to implement for the Genetic Programming Library and how the emulator should behave. Firstly, there are some rules for the competition the agents should follow. The rules are:

1. The program should interact with the game by capturing screen pixels.

2. The program should not noticeably slow down the game.

3. The program will have three attempts at playing the game.

This means that the system should not access or acquire any information directly from the memory of the machine running the game. The agent should find *all* the information it needs in the bitmap capture of the screen.

Secondly, the Genetic Programming library should be able to do some operations. It should be capable of

- creating random individuals.
- mutation of individuals.
- crossover of two individuals.
- fitness-proportionate selection.
- elitism on a generation.
- running a fitness test.

Some of these requirements are obvious, but it is important to know what needs to be done to get an overview of how long time the development will take. Other things are also good to know, like what selection process to use. The earlier these decisions are made the easier it is to plan for the eventual implementation of that part of the program. Though, one could argue that if one does not know what the actual implementation is going to be then one would have to keep a more overall open design in the whole system to later accommodate for these decisions.

The emulator should also be able to do some basic things that are needed to succeed with the project. It should be able to

- run the original binaries of Ms. Pac-Man.
- run with and without a graphical user interface.
- provide an API for access to key areas in the game.

The reason I want to run the original binaries is that it ensures that the project is using a version of the game that is as close as possible to the original behavior of the game and to the parameters of the competition. It also ensures that no bugs are introduced into the behavior of the ghosts, which is one of the most important parts of the problem. The only problem here though is that the emulator might already have some unknown bugs. I also want the emulator to be able to run in the background instead of with a GUI so fewer resources are used, and it can run on a server that has no way of displaying a GUI. The last thing is an API, and there is at least the need for a minimal API the agents can use to get the state of the screen and send commands to the game to move Ms. Pac-Man. Without this it will be very hard to do anything.

The Distribution Framework should be as simple as possible, but there are some criteria that it should follow. The framework is divided into two parts, the client and the servers.

The client should be able to

- distribute and set up servers on a set of machines.
- check which servers are available to process the individuals.
- split a population up into equal parts and send them to the servers for processing.
- Receive back the processed individuals and put them back together before sending them back to the GP Library.

The servers should

- Look to see if anyone is using the machine it is running on.
- Tell the client that it can not process anything if this is the case.
- Run the fitness function for the individuals.
- Return the processed individuals back to the client.

The reason for looking to see if there is anyone using the machine is to avoid using up the resources that are available. This is to make it less likely that a machine is turned off while running the fitness test and data is lost.

4.2 System diagram

Here is a diagram showing how the different parts of the system interconnects. A more detailed version of the Genetic Programming library can be found in the section for the development of the Genetic Programming library.

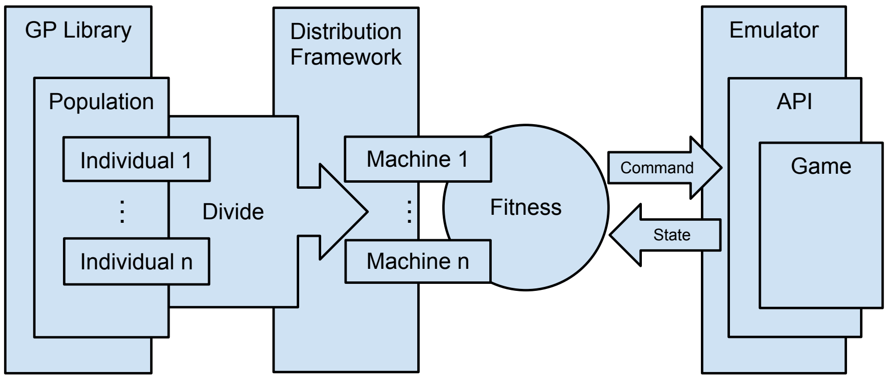


Figure 4.1: A diagram over the full system.

What figure 4.1 shows is that the GP Library creates a population. The population is divided by the Distribution Framework and sent in equal pieces to the different machines in the machine park of the Distribution Framework. Each machine then runs a fitness test for each of the machines. During the fitness-test the individuals send commands and receive the state of the program from the API wrapped around the game running on the emulator.

4.3 The Genetic Programming Library

The Genetic Programming Library consists of 3 sub-parts: The part that creates the generations, the part that runs the generations and, the specification of the individuals. It has gone through many changes to how it works, but there has been a general design idea throughout the process.

4.3.1 Diagram over the Genetic Programming library

This is a diagram over the flow of the Genetic Programming library. It show a simplified overview of how the population is transformed through the generations and what happens at the different stages of the generations. It does, however, not show the connection it has to any of the other parts of the system. Most of the implementation specific details are glossed over. This is especially evident in how the fitness of the individuals are calculated as this is done through the Distribution

Framework.

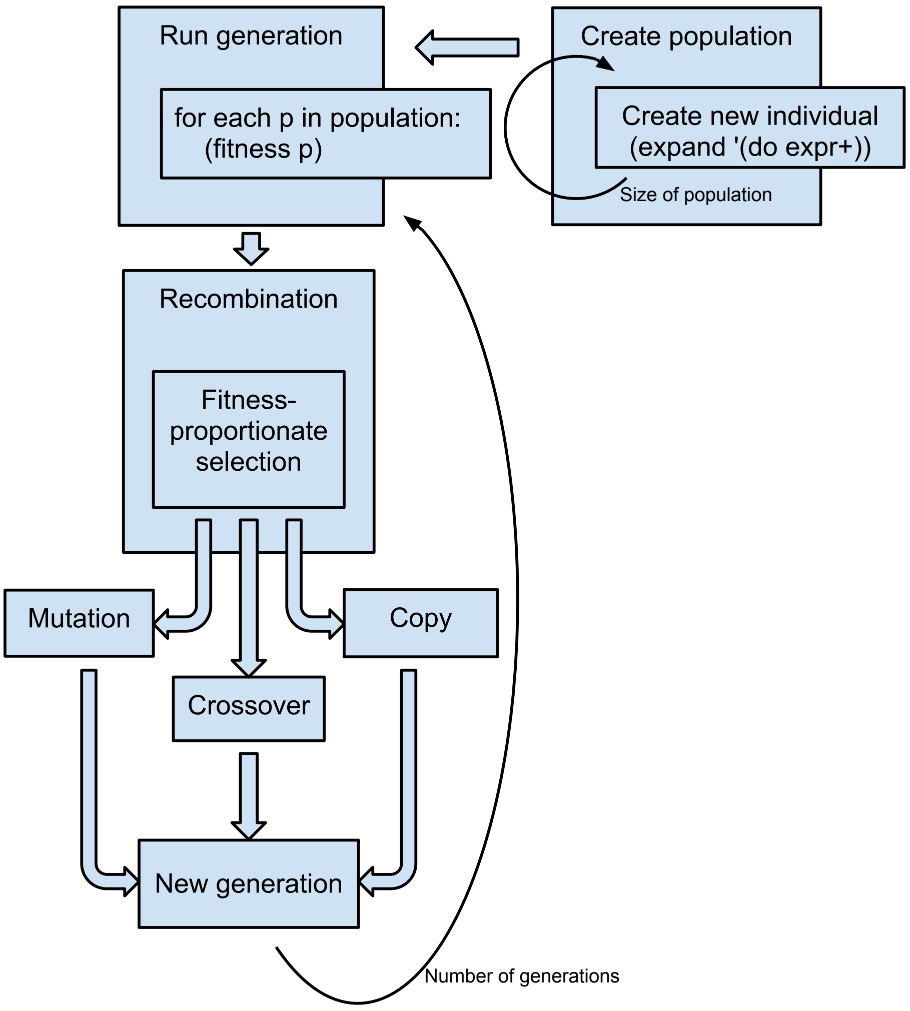


Figure 4.2: A diagram over the Genetic Programming library.

If we look at figure 4.2 we can see that first, a population is created. This is done by running the expand function for each new individual that should be created, i.e., for the size of the population. The population is then passed to fitness evaluation. After fitness evaluation the population is pass to recombination were the individuals that are fit for the next generation are selected through fitness-proportionate selection. There are 3 paths for the selected individuals to take. Either they passed through mutation, crossover or copying. The result of this is a new generation. The new generation is then passed back for fitness evaluation and the whole things starts over from there. This process will happen until it is aborted or a set number of generations is reached.

4.3.2 Individuals

How the individuals behave have changed considerably during the develop of the system, but this section will only be discussing the last iteration of the individuals.

The representation of individuals is with a map that contains the fitness and the program that the fitness-function runs. The programs are represented as a tree, or more specific lists-within-lists; which is the preferred representation of data in Clojure. This makes sure that instead of defining an evaluator for the "randomly" created program, it is possible to use the evaluator already present in the Clojure language. The only problem is that the programs have to conform to the Clojure syntax and language specifications, but this is really a feature as it therefore will be no difference between the system code and the created individuals.

The direction the individuals move in is controlled by a simple A*-search to the currently most interesting point. The interesting points are the points that has something Ms. Pac-Man wants to eat on it. To determine the most interesting of these points, the individual modify a set of weights it gives each point. The point with the lowest weight is the most interesting point. The weights are also used by the A*-search to determine where it does not want to go. Every point on the map starts at 0 and then additional weights are given or detracted depending on criteria found through Genetic Programming. In other words, the A*-algorithm calculates the shortest path to the interesting point (pill, super-pill, blue ghosts, fruit) with the lowest weight, and moves in the direction of that path.

```clojure
(do (adjust-point (get-pinky) 1000.0)
    (adjust-point (get-superpill 2) 4)
    (adjust-point (get-superpill 0) 0)
    (adjust-circle (get-blinky) 103 1.0E10)
    (adjust-point (get-blinky) 10.0)
    (adjust-point (get-sue) 5))
```

Using the above individual as an example of what the programs can do, we see here that the ghosts have their value adjusted such that the points they occupy have a higher cost than going another way (if the adjustment happens twice it is an additive process, the value can never become lower). The adjust-circle function will adjust everything around the point in a decreasing manner until it reaches the circumference of the circle.

4.3.3 Generation of generations

The generator first creates a random generation from a specified set of functions and terminals. This set is also further controlled by setting limitation in what functions and terminals can interconnect with one another. The generator creates the new individuals by selecting a random function that it expands in a guided but semi-random way using the grow method of expansion. When the generator creates a new generation from an old one it does many of the same things, except for some small differences.

The population is moved to recombination where the individuals are select by fitness-proportionate selection. A small change has been made to the way the selection method selects individuals and now gives a small advantage to programs that are smaller but have the same fitness as another program. The individuals that are selected are then chosen for mutation, crossover or just copying.

Crossover requires two individuals. A node is selected in each individual as the swap point. These nodes are swapped and two new individuals are moved into the new generation.

Mutation has 3 different ways it can happen as well. It can either replace, remove or insert a node in the code tree. Each of the three have an equal chance of happening.

Copying, mutation and, crossover have a set percentage chance of happening. The most likely is usually crossover. Then mutation and lastly copying with a small percentage of the individuals moved to the new generation.

A small percentage of the best individuals in the population is can also be moved into the new generation through elitism.

4.3.4 Testing a generation

To test a generation a fitness-function is mapped onto each individual in the population. There are two modes in which the fitness function can run. Either in graphics mode or in server mode. In graphics mode it requires a GUI and it shows the game as the agents plays it, but it requires a lot of resources and does not scale well over multiple machines. The server mode runs without a graphical interface and over multiple machines, this allows for more individuals to run simultaneously and faster. The problem however is that it is not possible to see what is going on.

For the most part the generations runs in server mode, but when a problem occurs that is too difficult to debug without visual confirmation of what is happening, it is good to have graphics mode. This also means that there should be as little as possible difference between the graphics mode and the server mode. To solve this the graphics mode is now only a small layer on top of the server mode. It runs in a thread alongside the emulator and agent and receives the pixel-map in the same way that the agent does and for every tick of the Ms. Pac-Man world it will update the visuals with the new pixel-map.

Keeping the graphical part of the emulator in a different thread also makes sure that it has as little as possible impact on the performance of the agent as the agent will not be dependent on the GUI to finish drawing the scene before it gets the pixel-map itself.

4.3.5 Configurations

The GP Library has many configuration options that affect everything from the size of the population, to how the population is created, to how many times the fitness-function should run. A small change to any of these parameters could give large changes to the behavior of the population and to the chances of finding a good solution to the problem at hand. Because of the uncertainty in the configurations it is very difficult to know if the reason for a poorly performing population is that there is a problem with the understanding of the problem domain or just a problem with the chosen configurations. It is also entirely possible that the current population is just *unlucky*[1] for the time being. This section will be looking at the different configuration options in the Genetic Programming Library.

Size of population

The most obvious configuration is the population size. A larger population size can give more variety, but at the cost of having to run more fitness tests. A problem however is that if the population starts to become homogenized, variety will decrease much faster in a large population than in a smaller one. This is because a small amount of different individuals will have a larger impact on a small population than on a large one.

Rate of elitism

The elitism rate ensures that the highest performing individuals are not lost in the next generation because they are unlucky. How elitism does this is that for each generation a set percentage of the best performing individuals are copied into the next generation. Individuals selected through elitism are also generally allowed in the normal selection pool for the next generation. The elitism-rate should in any case be low, around 1-5% of the population. Any higher and the population risk loosing diversity. The risk of loosing the best performing individuals when not using elitism can be quite high, especially with a selection method like fitness-proportionate selection, which is design to also allow low performing individuals to survive into the next generation. This means that there is a possibility for the population to be unlucky and only low performing individuals survives into the next generation.

However, there is the problem that keeping the high performing individuals can over time remove diversity from the population as the same individuals are copied every time into the new population. If these individuals are in a local maxima it might be very difficult to get out of that maxima as the more there are of the same individual the more likely it is to be chosen.

[1]By unlucky it is meant that a higher performing individual has been removed or the random combination of new mutations or crossovers has not found a good match yet.

Max starting depth and width of programs

It is possible to control how deep and wide the beginning programs can be. This is both to ensure that the individual-creator does not go into an infinite loop, and to control the size of the programs.

The size of the programs can be important as too small programs might lead the GP Algorithm into a local maxima that can be difficult to get out of as the next maxima demands a much larger program then what is currently possible.

Starting with very large programs is also bad as the larger the programs get, the more random elements are introduced that are most likely bad for performance. These bad performing elements then have to be removed from the population by the Genetic Programming algorithm through the generations.

It is also generally a better idea to allow the programs to grow naturally through the generations then starting with large programs, as randomly generated programs can introduce a lot of unnecessary code. This means that the Genetic Programming algorithm has to use more time pruning the code trees before being able to gain performance by growing.

Mutation rate

The mutation rate controls the amount of mutations that happens during the recombination. Depending on what one wants to achieve this configuration can either be set really high, or, as it normally is, really low. A normal mutation rate would be around 1.5%. It is also possible to cut out crossover as a recombination operator and only use mutation (and copying).

The reason for a high mutation rate is to introduce more variety into a population that has a tendency to drift into local maxima and produce homogeneous populations. A larger variety will allow the population to escape the local maxima. A low mutation rate is set if the population is able to hold a reasonable variety, but sometimes needs a push in a new direction.

A mutation rate that changes with the population could also be set if it is possible to have a good measure of the variety of the population. The mutation rate will be high if the population is homogeneous, but low if the population is more heterogeneous. This is similar to how Simulated Annealing works in that the configuration changes over time in accordance with how the population performs.

Crossover rate

This configuration is normally set very high. Most recombinations happens by crossover. A normal rate would be around 90%.

Reproduction rate

This rate is not explicit on the source code, but is an aggregate with the crossover and mutation rate. It is the missing percentage after crossover and mutation is added together. If we look at normal mutation rate and the crossover rate we can see that these percentages are quite low and around 8% of the recombinations.

Fitness runs

The last configuration that can be done to the Genetic Programming library is how many times the fitness test should be run for the individuals. Getting the right amount of fitness-runs depends a lot on the problem. For a functional problem problem, i.e., that it is known that the generated function will always return the same results, it is not necessary to test the fitness more than once for each fitness case. If it is a stochastic problem however, like the Ms. Pac-Man problem, the amount of fitness runs should be determined by how many times it takes to get a good statistical sample of the average run of an individual. This can however be problematic if it takes a really long time to run the fitness-function. If this is the case, then a compromise should be met between the time it takes to run the fitness-function and the accuracy needed to find a good average.

4.3.6 Restrictions on the GP Library

I have tried to keep the GP library as general and configurable as possible, but as it stands now there are many things that are specific to the Ms. Pac-Man-problem domain. Or, by that I mean that to write an extension to a new problem one will have to change the code for the definition of the individuals. This definition is now tightly coupled with the GP library. It is not difficult to do, but will take some time.

It would be better if the individuals were defined in a domain specific language, but this was not possible with the time limitations on the project.

4.3.7 Sample code of the Genetic Programming library

To get a deeper understanding of how the library does some of the more complicated tasks we will be looking at some of the code that has been implemented. We will be looking at the expand and fitness functions. The reason for looking at these two is that the expand function will show how the individuals are created and therefore gives some insight into the process of how the programs behave. The fitness function on the other hand will give a deeper understanding of how the programs play the game and what is needed to make a functioning fitness function for the Ms. Pac-Man problem. The full and uninterrupted code can be found in the appendix.

`expand`

`expand` is a recursive function. The recursion is ended if either a specified depth, symbol, number or, empty node has been reached. The function then passes that part of the individual to `atomize` to stop the expansion of new nodes. The function takes a list of expressions together with the remaining depth it can go to in the tree.

```
(defn expand [exprs depth]
  (if (or (symbol? exprs)
          (number? exprs)
          (empty? exprs)
          (< depth 1))
    (atomize exprs)
```

If the function is none of these things then we can continue with the execution. The first expression is always a function and doesn’t need an expansion; it stays where it was as the first part of the list.

```
    (cons (first exprs)
```

It is the rest of the `exprs`-list that has a possibility of expansion. We loop through them and provide an accumulator for the new list. If there are no more terms we return the accumulator.

```
         (loop [terms (rest exprs)
                acc ()
                expr-width (rand-int MAX-STARTING-WIDTH-OF-EXPR
                   )]
           (if (empty? terms)
             acc
```

For the rest of the list we check if it is any of these symbols and according to what symbol it is we replace it with a random function from the set of function that are of that type.

```
             (let [term (first terms)
                   exp (case term
```

If the term is of type `expr` or `expr+` we select a random function from the function-list and expand it. We also decrement the depth expand can go.

```
                             (expr expr+)
                             ,(expand (rand-nth ind/FUNCTION-LIST)
                                      (dec depth))
```

If it is `expr?` we check if a new randomly chosen number is smaller than the `EXPR?-RATE`, it is then a new expanded random function is returned from the function list. Else an empty list is returned.

```
                             expr?
                             ,(if (< (rand) EXPR?-RATE)
                                (expand (rand-nth ind/FUNCTION-LIST
                                   )
                                        (dec depth))
                                ())
```

For `point`, we return an expanded function from the `point`-list.

```
                             point
                             ,(expand (rand-nth ind/POINT-LIST)
                                      (dec depth))
```

If it is none of these we return an atomized version of the term.

```
                             ,(atomize term))]
```

Then the function loops on the list. If the term is of type `expr+` we return the terms as long as the allowed `expr`-width is greater than 0 (e.g., positive). In the other cases we loop on the rest of the terms. The accumulator is then concatenated with the expression created by the expansion of the terms. Lastly we decrement the `expr`-width.

```
                (recur (if (and (= term 'expr+)
                                (pos? expr-width))
                           terms
                           (rest terms))
                         (concat acc (list exp))
                         (dec expr-width)))))))
```

fitness

`fitness` is the function that tests the fitness of the programs created through the genetic programming. It takes the amount of times the test should run and the program as arguments. It returns the sum of the different fitness-scores it achieves divided by the amount of tries it does (e.g., the average).

```
(defn fitness [tries code]
```

`msp` (a global variable) becomes bound to a new object, the Game, which is the Java code that runs the game for the fitness test.

```
  (binding [msp (Game.)]
    (loop [score 0
           times 0]
```

If the set amount of times has been run or, it has run for 3 times and the average score is still 120, the fitness function returns. This number is considered somewhat special in that it represents the point were the program has in reality done nothing.

```
      (if (or (<= tries times)
              (and (<= 3 times)
                   (= (/ score times) 120)))
```

It then returns the score averaged over time.

```
        (int (/ score times)
```

If it does not end it starts the game and updates the game until it can receive commands.

```
        (do (.start msp)
            (.update msp)
```

It then adds the new score to the previous score, and increments the amount of times it has run.

```
            (recur (+ score
```

This part of the program is the real running of the fitness test. Here it updates the game state and runs the program in quick succession. It finishes with returning the score that the code managed to achieve.

```
                      (do (while (not (.isGameOver msp))
                            (eval `~code)
                            (.update msp))
                          (.getScore msp)))
                   (inc times)))))))
```

4.4 The Emulator

The Emulators only function is to run the Ms. Pac-Man binary, but it has received many modifications to make it run faster and smoother. An API was also built over the emulator to make it easier to interface the GP Library with the emulator, and to make the problem space the GP Library has to search through smaller. The API has received drastic changes throughout the development. In this part only the most resent version of the API will be discussed.

The API contains the code for starting the game and transforming the pixel-map created by the emulator into discrete structures such that the programs created by the GP Library more easily can understand the map. The map finds the positioning of the different objects within the game and contains a path finding algorithm, namely A*, that the GP Library utilizes.

After the rendering of the map the API will collect the pixel-map and process the image. Before the game starts the full graph of the map is calculated. The graph is then used for the remainder of that stage to calculate where all the entities are on the map.

It finds the graph by looking in 16*16 pixel blocks, which is the size of an entity on the map. Starting in the uppermost corner of where it is possible for anything to move, it looks in every corner of the 16*16 block to see if any of the four corners are not black. If this is true, it is not possible for an entity to move there and the point gets removed from the graph (, i.e. all corners needs to be black for it to be possible for any entity to move into the position). The API needs a set of different functions to look for each entity, but the method to produce them is basically the same: For each point on the graph it looks in the 16*16 block for a pattern matching any of the entities available on the map. If a block matches a pattern the entity is in that block.

4.5 The Distribution Framework

The reason for the Distribution Framework is that during development I discovered that running a large population over many generations was very slow. It could take days to run experiments and tests that needed multiple generations to give meaningful results. After some deliberation, I decided that I could use the many idle machines that are stationed in computer labs around the university.

The Distribution Framework was therefore made to make it possible to run a generation over multiple machines. It works by creating a server on each machine in the machine park. The server is responsible for receiving individuals from the controller and running the fitness test on them. This means that for each individual it receive from the client it will start a new emulator and run the individual against it.

The client is responsible for receiving a new generation from the Genetic Programming library. It will split the generation up into equal sizes and send one portion to each of the servers in the machine park.

The way the individuals are distributed is that a socket is opened to each of the servers. Each portion of the population is sent as a character stream to the server. The server receives the character stream and converts the character stream back into individuals. This is not the fastest way to do this, but it is fast enough.

4.5.1 Sample code for the Distribution Framework

In this section we will be looking at the implementation of how the Distribution Framework splits a population into equal parts, and then sends the individuals for processing on the servers. We will also look at how the server receives the individuals and how it runs the fitness calculation over them and afterwards sends the individuals back to the client. The full and uninterrupted code can be found in the appendix.

Server/client

The client creates the individuals and splits them up into equal sizes, it then sends each set of individuals to a machine.

```
(defn- send-inds-to-machine [individuals machine]
  (try
```

It opens a socket to the machine and creates a reader to read the reply from the machine.

```
    (let [socket (Socket. (format ``%s.klientdrift.uib.no''
       machine) 50000)
          rdr (LineNumberingPushbackReader.
                (InputStreamReader.
                 (.getInputStream socket)))]
      (try
```

It then creates an output stream and prints the individuals to the stream.

```
        (binding [*out* (OutputStreamWriter.
                          (.getOutputStream socket))]
          (prn individuals))
```

In the end it wait for the reply from the server, converts it from a string to Clojure data and returns it.

```
        (read-string (.readLine rdr))
```

Finally it shuts down the connection to the server.

```
        (finally
         (shutdown-socket socket))))
```

The function ignores all raised exceptions and if they happen we might lose some data because of this.

```
    (catch Exception e nil)))
```

The server receives the input and output stream of the connection.

```
(defn- run-fitness [ins outs]
```

It creates a new reader that gets read and the output string from the reader is converted to Clojure data, which happens to be a set of individuals.

```
  (let [rdr (LineNumberingPushbackReader.
              (InputStreamReader. ins))
        inds (read-string (.readLine rdr))
```

It then runs the fitness test on the individuals.

```
        out (gp/run-fitness-on inds)
    (binding [*out* (OutputStreamWriter. outs)]
```

and prints the result from the fitness test to the connection.

```
      (prn out))))
```

4.6 Other considerations

In this case, and as it usually is with GP, there a considerable amount of problems with defining the problem space and finding out how to approach the problem. There are many things to take into consideration to make a functioning system that also performs well.

4.6.1 Bloat

One of the ways the system tries to control bloat is to give programs that are shorter an advantage over the longer programs. It has a slightly higher chance of getting selected over the longer program.

Another way is that it restricts the way the GP Library creates and combines new programs. How this is done is by adding constraints on the types for the input and output of the functions. This helps to control bloat by avoiding configurations of agents where parts of the calculation is always thrown away. It is usually much simpler to just allow any input to all functions and discard the input that the function can not handle. The problem here is that this can build large patches of code that does nothing.

4.6.2 Diversity

Diversity is more difficult to control. It is important to maintain diversity, but there is no silver bullet that can *auto-magically* do it. Instead I have been trying to look for this problem in the population by monitoring the scores achieved by the programs. The thinking is that if there is over all the same scores for the programs the diversity is low. However if there is a large difference in the scores, the population is more diverse.

This is not foolproof and there are situations were it is not true that the population holds a high diversity even though there is a large difference in the fitness scores. E.g., if the way GP produces programs that produce highly disparate fitness scores for each run then the same program can appear to be the best program and the worst (in the most extreme cases). A way the system tries to diminish this problem is by running the programs several times and then averaging the score. This way it can make a much more balanced view of how the program performs. There is a performance cost to this though so there has to be a balance between running the program many times for accuracy and not running the fitness function for a too long time.

It is also highly possible that different programs get very close to the same scores. However, an optimal population will usually have some individuals that are better then the rest, and some that are worse. If this line starts to get blurred, there is something wrong with the population.

4.6.3 Emergent and unexpected behavior

Some emergent behavior will always appear when doing GP, and unexpected behavior is per definition unexpected. I have been trying to minimize both these two types of behavior from happening, though both forms have been observed during the development. E.g., there was a problem were the most immediate solution to the problem at hand was to grow the program in size. If the program was bigger it had more time to decide what it wanted to do and therefore it would do better. This is a bad solution for the problem, even though it provides a better score in the game, because performance is also tied to time constraints set by the contests for the Ms. Pac-Man problem. The way this was solved was by putting a constraint on the score such that a smaller program is always beneficial given the same score in the game. This is not a foolproof plan, but gets rid of the most extreme cases. The problem described here was also solved by making the agent run *after* the game instead of at the same time.

4.6.4 Other

A problem that haunted the project for a long time happened when I was trying to parallelize the system. The problem was that usually one program would not finish and instead run forever. This seems like a problem that could be easily fixed, but the problem was that this would happen maybe once during 10-15 generations of 500 programs. The program would also be distributed over approx. 40-50 machines. That is a possible 7,500 agents running over 50 machines, and there is only one program exhibiting the problem!

The first thought was that there had to be a problem with the program, but after retrieving the program and running it locally the problem disappeared. The focus then went over to that the machine might be the problem, that there had to be some faulty software on that machine or hardware that interfered with the operation of the fitness-run, but after removing the machine from the machine park and again running the GP process for around the same amount of time as last time the problem would reappear. This time with another program.

In the end I discovered that the problem was that in some extreme conditions a thread that started the game after reaching the start screen would run and finish before the game had created the next frame. This caused the game to not start and there would be a program stuck at the start screen. The problem was solved fairly easy after discovering the issue by making a loop that ensured the game had gone past the start screen.

This problem would also appear later, but for a completely different reason.

This time an agent was created that managed to find a configuration that would always be able to avoid the ghosts, but would not pick up the pills and finish the game. The problem was solved by putting a timer on the fitness function.

4.7 Tools

In this section I will discuss the different tools I have used to complete the project. Why I chose the different tools and some of their strength and weaknesses.

4.7.1 Java

The system is partly programmed in Java because it is a language I was already quite comfortable with, and that the Emulator the system uses to run the Ms. Pac-Man game also is written in Java. Java is a nice language, but has a large amount of syntax (compared to some other languages). It was therefore decided to use another language that could intermingle with the java code for the Genetic Programming library.

The problem with Java though is that it can be slower then other languages, but it has good editor support and debugging utilities.

4.7.2 Emulator – CottAGE

CottAGE is an emulator built in Java. It has drivers for many old games, including Ms. Pac-Man. In this project I am using a stripped down version of the CottAGE emulator that now has the possibility to run in a headless mode, e.g., without a graphical user interface, and that also can run multiple instances of the games next to each other in the same JVM. CottAGE needed to start a new JVM each time one wanted to run a new instance of Ms. Pac-Man, but during the development the code was modified such that it is now possible for many games to run in the same JVM without disturbing each other. This removes overhead when running the GP Library and preserves resources.

Unnecessary code was also mostly removed so that the remaining code is only the code that is needed to run Ms. Pac-Man. The reason for this is to both create a smaller footprint for the code and to make it easier to find where in the code certain things happened, like where the score of the game is stored. Removing things also made sure that I became more familiar with the code and had fewer places to look for bugs if I needed to.

CottAGE is open source, but currently unsupported, software. It is the basis for a new emulator, but it seems unsupported as well. The reason for using CottAGE instead of the new emulator was that during the preliminary research for the project I could not figure out how to run Ms. Pac-Man on the new emulator and there was no available examples of it running Ms. Pac-Man or Pac-Man anywhere on the web. The web-version of Ms. Pac-Man that the competition recommends is also based on CottAGE.

4.7.3 Clojure

To develop the Genetic Programming library I decided to use Clojure as it is a dialect of Lisp that works well together with Java. The reason I wanted something that worked well with Java was that the Emulator chosen for the project is written in Java. I also wanted to work with a lisp as it is a good match for Genetic Programming in that it is easy to treat code as data and data as code. This allows us to create code for the Genetic Programming task that is real code, and I don't have to create my own language that runs on top of everything else. In other words the Genetic Programming library creates code that is Clojure-code. This helps considerably as I also use standard Clojure functions inside the code of the generated code.

4.7.4 Distribution – SSH – clojure-control

I soon discovered that running the system took considerable amounts of resources and time and that I needed a way to either increase the speed of the code or to allocate more resources. There is a limited amount of speed ups that can be done to the code so I decided to see if there was a way to get more available resources. I discovered that there were plenty of idle machines with linux installed all over the university. This lead to finding another open source project, clojure-control, and made a distribution platform, through SSH, to distribute the work load onto all of the available machines. This worked for some time, but, because of instability of the connection between the machines, another solution was devised.

In the end I ended with implementing a client/server-architecture where the client would create and manipulate the generations and each machine in the network would run a server capable of running the fitness test. The client would then split and distribute the generation over the available servers.

4.7.5 Git – github

The project uses git as its version control software. Git is well suited for the task in that it is easy to work with and easy to find a host for the code. I decided to use github as the host because, at the time the project started, github said they would provide a closed repository free of charge for students working on projects for their education.

4.8 Iterations

There have been many changes during the coding of the problem as new ideas have formed and new problems have been discovered. This chapter will be looking at these problems and changes and how the program evolved during the development. It will however not go into any depth for the experiments that were done through the iterations. This topic will be presented at a later stage.

4.8.1 Iteration 1

Iteration 1 of the development consisted almost exclusively of trying to figure out how the emulator worked so that I could make a version of it that runs in the background and is not dependent on a GUI and, making a preliminary Genetic Programming Library.

For the GP Library I focused on a minimal function set that I thought might be able to create increasingly better programs. The functions were at this point defined as:

- `(move-left)`, `(move-right)`, `(move-up)`, `(move-down)`
  These functions would be responsible for changing the direction Ms. Pac-Man would move in.

- `(do expr+)`
  `do` would allow for sequential execution of commands.

- `(get-pixel int int)`, `(get-pixels)`
  These would allow access to the value of a specific pixel or all of them.

- `(if expr expr expr?)`
  `if` allows for branching.

- `(rand-int 288)`
  `rand-int` would allow for random integers to the height of the map.

- `(= expr+)`, `(msp> expr+)`, `(msp< expr+)`
  Comparison operators are important for the programs to discover distances and find things in the images provided by the emulator.

- `(or expr+)`, `(and expr+)`
  `or` and `and` gives an alternative to using the branching operator for everything.

- `(msp-sleep)`
  `msp-sleep` puts the running thread to sleep. This is for the program to time specific events.

- `int`
  `int` is not an operator but a placeholder for a constant. On program creation the expand-function will transform this into an integer and use it as a terminal.

- `()`
  The program uses the empty list as a terminal.

Here, `expr` stands for a single expression and, `expr+` is series of expressions. An expression is one is either a function or terminal.

All functions are also programmed in such a way so that if they take in an argument of a type that it can not handle it will drop the argument, as we can see from the msp>-function:

```
(defn msp> [& keys]
  (let [l (remove #(not (instance? Number %)) keys)]
    (if (empty? l)
      true
      (apply > l))))
```

Here we can see that msp> removes any instance from the input that is not a number, it then checks of the remaining list is empty, if it is it returns true, else it checks if the numbers in the list is ordered from largest to smallest.

In this iteration the fitness function(s) are also radically different from what they will become in later iterations and are heavily dependent on timing instead of proper synchronization. This both became a source of bugs and much time spent in debugging because of it.

I would also implemented mutation as the only recombination operator for this iteration. The implemented mutation operator is a bit interesting though, in that it would iterate through the nodes of the program and each node would have a chance of mutating. If it mutated a node it would stop the mutation process such that only one node would ever mutate in a given program. What this entailed though was that the bigger the program the more likely it was that the program would mutate. This got changed in the next iteration, because it became unpredictable over time and there was a needed for more control over what goes on when a program mutates.

The reason for only implementing mutation as a recombination operator was to get something functioning quickly and other recombination operators can take more time and requires more thought to implement correctly.

The last thing that was implemented in the GP Library for this iteration was fitness-proportionate selection as the selection method of choice. The reason for this instead of tournament selection is that fitness-proportionate selection is easy to implement and has shown great results for other people in the past. I also wanted a selection method that did more than just selection based on elitism as that generally leads to a homogeneous population, and that is undesirable. Even though I wanted to have a functioning GP library (and something to run it on) as soon as possible

I though that the time would be best spent if it wasn't spent on implementing something that was not going to be use later.

Plenty of code was also removed from the emulator, though there was also some new code to support headless mode. This way it was possible to run the code in the background without disturbing the user of the machine the process was running on. It also makes it possible to run faster as the emulator does not have to show the graphics. An attempt was made to make the emulator run even faster, but because of a bug in the emulator this was not possible before completion of iteration 1. The bug would be revisited in iteration 3.

4.8.2 Iteration 2

The system went through drastic changes for iteration 2. What was discovered from iteration 1 was that the function-set that was used for the generation of a population was too small as I observed what could only be describe as random search for programs. It was therefore decided to add more functions in the hope that it was not the basis for the current idea that was bad, but just that the current problem space was too large and had to be made smaller for the GP library to be able to find a satisfactory solution in an adequate time frame.

The first thing that was done was that the entities and how to find them was added. This was done to make it easier for the programs to know what the interesting items there are to operate on in the game. The list of the items are as follows:

- `mspacman`
- `blinky`
- `pinky`
- `sue`
- `pills`
- `walls`
- `walkway`

To make the search space even smaller the program generator was modified by sorting the functions into types. Sorting them into types allows the generator to ignore the other functions when selecting a node for an argument that needs a specific type. This can be seen here as the functions that return a boolean are sorted into one list:

- `(msp-check-area-below entity)`,
  `(msp-check-area-above entity)`,
  `(msp-check-area-leftof entity)`,
  `(msp-check-area-rightof entity)`
  These functions would check if there was a ghost in a direction from the entity. Later, they where also made to support if Ms. Pac-Man was in the direction from the ghosts if a ghost was passed as the entity.

- `(msp-closer? entity item)`
  `msp-closer?` is a method more than a function and relies on state. It receives an entity to check for and if it has check for that entity and item before (in this context all entities are also items) it checks if the distance between them is smaller than before, and if it is it returns true else false. If it has not check for the entity/item-pair before it returns false by default.

- Others
  The old functions from iteration 1 that return booleans are also in this list.

Another new function that was added was the `(msp-relative-distance entity item)`. This function, as its name suggest returned the relative distance between the `entity` and `item`.

At this time there was a shift in the way I was thinking about how the programs should behave. I could split this into another iteration, but I feel that the ideas overlap in such a way that to split them would change the understanding of how the development progressed.

The shift in thinking was with the behavior of the programs. The old way from iteration 1 was that the program should execute behavior as soon as it needed it. This would had the problem of creating both large and erratic programs. I therefore opted to change the way the program would give commands to Ms. Pac-Man, now the program would return a direction (or none), instead. This would lead to programs that would not have to time things correctly and instead just worry about current state. It would still have to run the program in parallel with the game, though. The idea was that running the programs in parallel would ensure small programs that responded within the time constraints in the problem without having to specify it in the fitness-test; it becomes implicit. The more one can make these things implicit for the GP algorithm the less problems with undefined and emergent behavior there should be.

I also wanted to remove the dependence on timing in the fitness test, which was becoming a source of bugs and slowed down the fitness tests considerably. Since the desire was also to have the programs run in parallel with the game I had to resort to object locking and other ways to ensure that the program and game were on the same page. Therefore I needed to ensure that the agents would wait until the game had started before it started to send commands to the game. There were some small problems here, but nothing colossal. A problem that took some time to figure out was that to start a game it needs to receive a series of key presses, but the emulator will not accept key presses unless the thread it is in is running. In the end the problem was solved by spawning a new thread that would send the necessary key presses to the emulator while the emulator was running. Using a lock on an object also ensured that when the game ended and a new game was starting the program could be paused while waiting to start again.

When the tests for iteration 2 started a frustrating problem was discovered. Running the fitness test the amount of times needed for an accurate account of the fitness of a program was taking to long when running it over large populations. The first solution that was tried was to run the fitness tests in parallel. The thinking was that since each machine in the machine park have a 4 core processor available it should be able to run at least 4 individuals in parallel as the 4 core processor has 8 thread available, 2 for each core; one thread for the game and one for the individual. In the end I discovered that the game was running in normal game speed and therefore the game thread would spend a significant amount of time in a pause it was possible to run 8 individuals in parallel without losing any performance on any of the individuals.

This was still not fast enough. The progress through the generations was still unbearably slow. Especially since a population as large as 5-7 hundred individuals was wanted and the current framework could only support up to 100 individuals.

The solution to this came in that the university has a number of computer labs that run Linux with access via ssh. It therefore became possible to split the population up into sizable chunks and send them for processing to different machines. It was possible to do this since no programs fitness can affect another programs fitness. To build the distribution framework I decided to first try to find an existing open source library that I could use or re-purpose for what was needed.

The problem was that there was some restrictions in what could be done with the machines. First of all, having only normal user access to the machine meant that I had no admin and was not able to install any new software to the machines. So, after searching for some time I found the clojure-control library; a library meant to deploy commands from a local machine to a set of remote machines. It uses ssh to connect to the machines, which is available, and is written in Clojure, which is good for me since all my affected code is also in Clojure. Another thing that made this framework good for the project is that when a user logs in to a university machine one gets access to ones private storage area. This meant that it was possible to start the fitness test on a remote machine if only a command was sent to the machine containing the programs it was going to test.

Modifying clojure-control didn't take long, but a problem soon arose. Crippling problems with the ssh-daemon running on the machines would occur daily to hourly. The ssh-daemon would either crash and new requests would go into a hang-up loop or something would freeze and not respond. This meant that if the system sent a request to a machine with this problem the process sending the request would also hang until the process was killed. The next problem came with authentication. I was using public key authentication to authenticate the connection to the machines, the problem however was that one part of the authentication to the servers is with Kerberos. The Kerberos authentication does not accept public key authentication. This is not a problem in and of itself, but what happens when an authentication request does not authentication with Kerberos is that Kerberos never responds with a deauthentication. It just never authenticates. This means that any connection will just hang forever. These problems would be intermittent and it was impossible to know when they would appear. This is why it took a long time to debug. When the problem was discovered I filed a bug report with the IT-department at the university, but in the end it could not be fixed in time and I had to find a new solution that proved to be much simpler and more robust than the modified clojure-control approach.

The new solution was thought up after a discussion with some fellow student that are working on networks and they asked why I didn't do the whole client/server business by myself through socket programming. In the end this was a much better idea. Especially after a new library that wrapped around Javas internal socket API for Clojure was found, which made much of the socket programming considerably easier than it would have been otherwise. Having no earlier experience with socket programming I sat out to learn and discovered that it was much easier to build this sort of thing myself since the problem with the distribution of programs was a rather simple one.

In the end the whole system consisted of one client, the part that generates and divides the population, and the servers, the machines where the fitness of the individuals is calculated. This is done by dividing the individuals in the population into equals sets and then sent to each machine in the cluster as a string. The client then reads this string as Clojure-data, and after that the fitness tests evaluates it as normal. This is as simple as it can get and gets the job done in an efficient and satisfactory manner.

In the end, the problem with a too large search space was still prevalent and the GP library could still not find a sufficient solution in a satisfactory amount of time. This leads to the next iteration were a new idea comes into light and the discovery of the real reason for the unsatisfactory state of the programs appears.

4.8.3 Iteration 3

As was said, for iteration 3 I was again thinking of how I could narrow down the problem-space further. The problem space was still too large for the GP Library to find any acceptable solution in a reasonable amount of time. For this round I would go through 2 stages of ideas and I will explain them in-depth, but before this is done the real problem that has been plaguing the development with insufficient run times and poor performance will be revealed.

The real reason for all these problems with programs that perform in an insufficient way stems from a bug in the API for the Emulator. This bug does not show up when playing the game normally as a person because a person will not do what the programs does and relies on. The problem is that when the programs "press" a button to move, the button does not stay "down". It releases the moment after the program has "stopped" pressing the button, which is right away. That is not totally true, but it is in effect what happened between the programs and the emulator. Things would still work and the generations would still do better over time, but would stop improving after a couple of generations. How this happened is that the agents could up to a certain point rely on timing the button presses for exactly when it was needed, but in the long run this would not be enough. This problem was even thought to have been eliminated earlier, but obviously there was a second component to the problem. I felt this was important to note before continuing as it is very important for the direction of the project.

Iteration 3 saw the removal of running the game and the program in two different threads. Now the game would run and then the program would decide its course of action. The reason for this was that there was still a problem with seemingly random behavior in the agents. A change of tactic was therefore needed to see if there was a way to figure out why this happened. After discovering the real reason, as described earlier, there was no longer any time to change back to the original plan.

This also lead to revisiting the bug discovered in the emulator in iteration 1. The reason for this was that since the game and agent would run alternately, it should be possible to just run the emulator at full speed without having any sleeps and pauses since it was no longer necessary to worry about timing. The problem however was that when the sleep cycle was removed from the emulator it would crash with a divide-by-zero exception. This error happened because the emulator calculates how long it should pause to keep a constant frame-rate of 24 fps, but when the emulator is no longer throttling the speed the frame-rate is set to 0. This will later trigger a divide-by-zero exception. After some diving into the emulator code the flaw was found and fixed with a try-catch that would just ignore the error. This seemed to work fine.

In the first stage of this iteration the plan was to "pre-calculate"[2] most of the code that was deemed unnecessary for the GP to find. To do this there first had to be a refactoring of the old Java code to make it easier for the Clojure code to look at the different parts of the emulator. This also lead to a refactoring and rethinking of the function-set for the GP as well, since it needed functions to deal with the new emulator API. This is a messy time in the development and much of the work is done because I was moving back and forth between ideas, which makes it difficult to explain in an efficient manner. I will therefore gloss over some of the gritty details.

Another problem was that the programs were not able to figure out a good path-finding algorithm by itself and if it can't find a path-finding algorithm it will never behave in a good way. With that in mind I decided that implementing A* would be a good idea as A* is one of the top performing path-finding algorithms. With A* the idea was to calculate the distance to all the "interesting points" in the map; the ghosts (and the blue ghosts when they appear as Ms. Pac-Man eats a super pill), the pills and the super pills, and make the program decide where it wants to go. The way the programs was supposed to do the decision was that it would first sort the goal points, all the pills and different things it can eat, according to the real distance, computed by A*, from Ms. Pac-Man. The Program would then rotate and change the list as it pleased. It would also remove points from the map where it did not want to go. It could remove any point that did not contain a pill (, as it has to eat all the pills before it can advance to the next level). When the fitness-functions run it will then use the newly created map as a way to calculate the A* distance to the target that the program selects. This way it could have a path-finding algorithm that is efficient and that the program could still have control over where it should go. In the end there was a rather large problem with the approach.

The problem was that even though A* is one of best path-finding algorithms to use I found that it took to long to calculate A* from Ms. Pac-Man to all of the interesting points. This coupled with an already heavy load of finding all the object on the map made the time it took before the program could even try to calculate the direction Ms. Pac-Man wanted to go, just too slow. An attempt to try and optimize

[2]And by pre-calculation it is meant that the system calculates the operations so the GP Library can focus on finding good programs for the important parts of the problem.

the A* algorithm such that all the paths were not calculated each time was made, but to no avail.

In stage 2 a new idea emerged that was designed to solve the problems in stage 1. The idea is that instead of figuring out what the best point to go to is, the bot should figure out what the least interesting areas are and weight them for the A*-algorithm. This means that the bot only has to decide where it does not want to go, not where it wants to go. This demands less processing from both the calculation of the target and the amount of work the programs has to do.

The program now could add a circle around a point (that could be relative to an entity on the map) and add the desired desirability, or more undesirability, to the points captured in the circle. The A*-algorithm will then take this desirability and use it as weights for its calculations. A higher weight means that a path has to be considerably shorter to weigh up for the undesirability of going there.

4.8.4 Iteration 4

The final iteration was more about optimization, debugging and fixing problems that had cropped up in the previous iteration. The first problem was that the agents would in some rare cases go into an infinite avoid loop. What I mean by that is that the agent would be able to always avoid the ghosts. The way this would happen is that the ghosts would chase the agent out the tunnel on one side of the map, when the agent reappeared on the other side the ghosts would turn to that side of the map and chase the agent out of the map again and to the other side. In some configurations this behavior would continue forever. The problem was solved by using a timer to determine how long the agent was allowed to play the game before being stopped. The time is determined by how long on *average* the previous generation used to run through the game and then adding two standard deviations to that number. The first generation is given half a minute to finish the game.

The second problem was that the game was running too slow. A profiler was therefore utilized to find the worst spots in the code. The profiler showed that the A*-implementation was one of the worst offenders. This lead to the re-factoring of the A*-implementation. It went from one of the slowest pieces of code in the project to admittedly still being significant, but was no longer anywhere near the amount of time spent in this part of the code as before. This was done by making as much of the code static as possible. There was also an overall clean-up of the code that certainly removed some hot spots as well.

The second large optimization that was done was on the code that creates the map for the agents to read. Finding the different entities and items on the map was taking a really long time. There was a lot less room for optimization in this part of the code as most of it is needed to find the different entities on the map. However, there were some areas where the code could be streamlined into a single loop instead of running through the same data every time. This brought down the run-time of this piece of code by a significant amount. Even though it is still one of the heaviest pieces of the code base, the time it is using now is much closer to acceptable then what it was before.

After the discovery of the bug where the agents had to accurately time when they should turn or risk missing a corner, I wanted to do a new experiment with one of the *old* ideas. The system has change significantly since then, especially how the data from the game is collect, so the code had to be re-written. This was however surprisingly easy as the code and libraries have been designed quite modular and there was not that much change needed for the old design to work.

Looking back at iteration 2 and the function that where defined in there, a lot of the same functions are available again. The `msp-check-area-*`-functions are all there. The distance functions are now gone and replaced with some other functions that work a bit differently. Since A* is now available for use just the direction to the entity is recorded instead of the distance. It would probably be easy to also implement a distance based on the A*-path as well, but because of time constraints it was not implemented. Instead, the opposite direction is calculated.

Some entities where removed and some added. The new entities now include, as before, Ms. Pac-Man and the ghosts, but there are also some new, the closest pill, super pill and blue ghosts, as well as the furthest blue ghosts and super pill.

The agent is calculated, as in the last iteration, after the map is calculated and returns the direction to move Ms. Pac-Man in. This is again, so that we don't have to worry about threads being parked and can just fully utilize the power we have available on the machines.

The reason for not splitting these two strategies into two different iterations is that there is no architectural change behind the choice in strategy. Both rely on the same basic background that does most of the work.

Chapter 5

Evaluation

In this chapter we will be evaluating the system and look at experiments that were done during different stages of the development. We will see that during the development there is a significant improvement in the performance of the programs created by the Genetic Programming library, but that it does not achieve the better-than-human goal. Why this is the case will not be explained here, but we will revisit it later when we look at the conclusion of the thesis and further work.

5.1 Genetic Programming library

The Genetic Programming library performs well for the task that it is supposed to. It conforms to all of the criteria mentioned in the requirements for the library. There are some complaints though. It would be better if the way to adapt it for new situations was through a Domain Specific Language (DSL). This would mean that if one wanted to use GP library in a different setting and for a different problem than what it is now, one would only have to give it a new set of configurations instead of how it is now where one has to go into the source of the library and change things to match a new situation.

Another feature that would be nice is if the different parts of the library would be easier to change into a new implementation. E.g., one should be able change or add new mutation operator by writing a new one and present it to the library. This could be a part of the DSL mentioned earlier.

There is also room for speed ups in the execution of creating new generations. This is not a big complaint however as the fitness function is at least a couple of orders of magnitude slower and time spent optimizing for execution speed should be spent here instead. Though, this depends heavily on the problem the fitness function is implemented for.

5.2 The Emulator

The emulator does everything that it should do. It runs the original binaries for Ms. Pac-Man, it has a background mode, and contains an API for the agents to have easy access to the properties the agents need to play the game. The API does however much more than just provide an easy interface for receiving movement commands from the agents and sending the image of the screen. It is responsible for calculating the position of the items within the game and to calculate the distance between them. All of this is implemented in the API to remove as much as possible from the problem space and make the Genetic Programming library focus on the important parts of the problem instead of getting bogged up in discovering the different items and where they are in the game.

A problem though is that the implementation of the API is a bit slow. It needs to be faster so more of the processing can be spent figuring out what to do instead of what is on the map and where they are.

The emulator is also slow. It could be beneficial to the agents, as the system is now, if some time was spent optimizing the calculations for the game. The reason for this is that the agents can not run before the emulator is finished with producing the image. The agent then uses the space between finishing the previous screen and starting to calculate the next one. If producing the image was faster more time could be spent calculating the actions of the agent.

5.3 The Distribution Framework

The Distribution Framework also conforms to all the criteria that was set for it. However there are some complaints. Most were resolved by switching away from using ssh as the communication protocol and communicating by sending strings over sockets, but some remain.

The first is that since this is client machines and used by other students as well, some data will be lost some of the time. The reason for this is that sometimes the other students will turn these machines off. There is no fail-safe built in to retrieve the lost data. Instead the system will continue working with the remaining available population. The individuals running on that machine however will be lost. This is probably the most glaring mistake in the Distribution Framework.

It is also much more difficult to debug code that is running on multiple machines. Especially when bugs that seem to only appear on some of the machines, but all of the machines are running mostly the same hardware. The difficulties can however be mitigated with good logging of the behavior of the code.

There is also no security implemented. This means that if anyone knows the machine the server is running on and the port it is listening to, he can send and execute commands on that server. Possibly disrupting the process. Though, because of good security at the network level, only people with access to the network will be able to do so. This has not been a problem for the development during this thesis, but should be taken into consideration in further studies. The reason for not implementing any security was that the risk involved was considered low and the time spent on implementing security was therefore better spent on other parts of the project.

5.4 Experiments

There have been conducted more experiments then the ones represented here, but because of the large number of experiments I have decided to only present some of the experiments in light of the four developmental iterations.

5.4.1 Experiment design

There is large variation in the experiments and they differ throughout the evolution of the software, but there is a red line through all of them. The experiments consist of making configurations to the software and running them through the genetic programming library. Most of the time the experiments had to run for several days to give any meaningful results.

Because of the nature of what I am trying to do and how I am doing it the code for the programs has changed considerably, this means that it can be very difficult to compare between the different iterations. I will try to compare where it is appropriate, but will otherwise abstain from doing so.

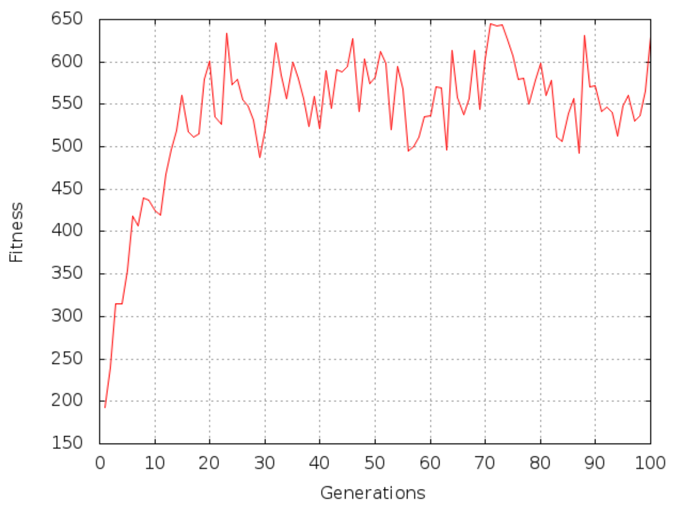
The experiments were done by doing multiple runs with variation on key variables in the GP framework. I would change the mutation rate, reproduction rate, population size, rate of elitism, the starting depth and width of the programs and the amount of fitness-tests ran for each program. I would also change the way individuals were created and some times seed the initial population with programs that I knew were doing well. The types of mutations that can happen also changed from one to three.

Each experiment will visualize the data collected from the fitness tests. The fitness of an agent will be the average score achieved from the fitness tests. This means that the graphs showing "best score" in reality is the best average score of the population.

5.4.2 Experiments in iteration 1

In the first iteration the point of the experiments was not to get a high score, but to prove that the Genetic Programming framework was working and that I had a platform to build the rest of the system on. I wanted to see that the programs were interfacing with the game well and that they could improve over time. Anything outside of this goal was considered a bonus.

If we look at figure 5.1 we can see that the population is gradually getting better. Even though the numbers might not be that high, the first couple of runs showed much promise as it is possible to see from this that the programs are actually doing something. It is possible to say this, even from just looking at the scores. The reason for this is that 120 is the score the program would score if the program would do no actions at all. Ms. Pac-Man will always start moving left and if no action is taken she will collect 12 pills worth 120 points. There are other ways of achieving this score, but usually it would represent a program that did little or



Pop. size:	20
Fitness runs:	5
Elitism:	N/A
Mutation:	0.02
Crossover:	N/A

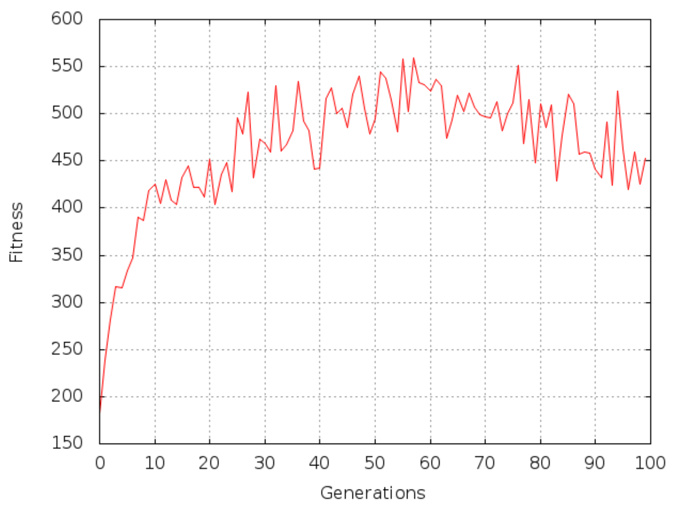
Figure 5.1: Average score of *test* population, 25.11.11

next to nothing. Therefore a score lower than 120 is actually more interesting than a score of 120. Individuals with a fitness score lower than 120 is actually doing something, something wrong, but at least something.

However, there are two clear problems that the graph shows. The first is that it plateaus quite quickly; after only 15 generations. The other problem is that after it has plateaued the average score is erratic. This signals that there is something wrong with either how the fitness score is run or how the problem is encoded. As we know from previous discussions this is probably the problem with only changing direction when the "button"[1] is pressed at the same moment as it is possible to move in that direction. This is reoccurring problem that won't be properly solved before late in the third iteration.

After developing the program further into a better working state a new test was made as the previous test was only to test if everything was working and how long it would take. Based on the previous test, in this test the population size was modestly increased to see what effect this would have on the run time. Two new functions where added, `get-pixelxy` and `find-colour`, together with two variables, `x` and `y`, to give the agents some state they could manipulate. In this case the `x` and `y` variables represent position in the *x*- or *y*-plane of the map.

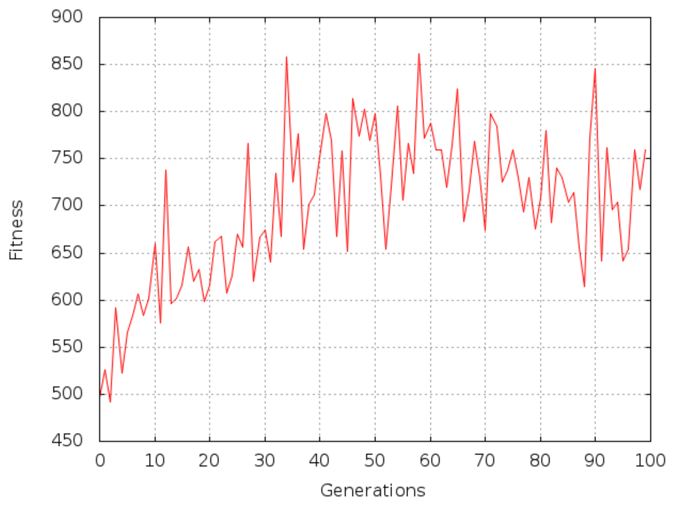
[1]Putting button in quotes here because the programs doesn't actually push buttons but sends commands that the emulator interprets as button presses.



Pop. size:	25
Fitness runs:	5
Elitism:	N/A
Mutation:	0.02
Crossover:	N/A

Figure 5.2: Average score of population, 25.11.11

Looking at figure 5.2 we can see the average score of the population and what we see is that it takes a sharp turn upwards in the couple of first generations, but after a while it stabilizes around 480. This is actually a lower score than the first test run, but even though this is the case there is some interesting things we can see in this graph.



Pop. size:	25
Fitness runs:	5
Elitism:	N/A
Mutation:	0.02
Crossover:	N/A

Figure 5.3: Best average of population, 25.11.11

The first interesting point is that the graph uses a much longer time to plateau than the previous test run. It is also a bit less erratic than the previous graph. The strange thing though is that it seems to be falling in fitness towards the end. There are several reasons for this to happen. One is that this is only 100 generations. There is a chance that the individuals will start to climb again. It is quite normal to see this kind of oscillation in the fitness of a population after it has plateaued and is "waiting" for a new recombination to happen to make a better individual. This is especially true when the generations are run, like this one is, without elitism. This means that if a generation is particularly unlucky it might, through bad recombination and selection, end up with only individuals that are worse than the previous generation. Whole genomes might actually disappear in this sort of situation.

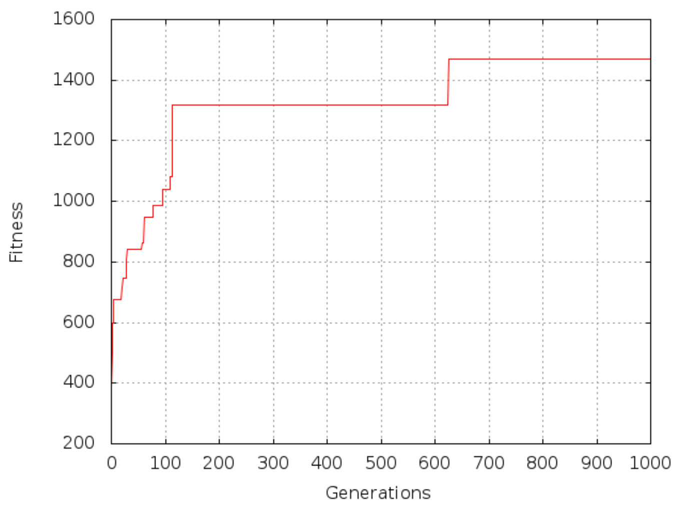
If we also look at figure 5.3, which shows the best individuals for the same generation, we can see that throughout the generations it has a very erratic behavior. Comparing this to the average score we can see that the average has to be responsible for evening out this erratic behavior. It is also evident that the best score is partly responsible for the decline in fitness towards the later generations, but that probably there are worse individuals than there were before in previous generations.

An interesting observation is also that the best individuals in a population does not control the average as much as one might think. This is good as that would mean that the fitness score is more evenly distributed throughout the population.

5.4.3 Experiments in iteration 2

Iteration 2 saw one of the longest running experiments and perhaps even one of the more interesting ones. Maybe not interesting from the point of solving the problem, but interesting from the perspective of how Genetic Programming works. We can see this in figure 5.4.

In this experiment the population ran for a 1,000 generations. What I mean by interesting for how Genetic Programming works is that if we look at the graph we can see this stepping of how good the best of the generation is. The reason for this stepping is that in the experiment, elitism is used to preserve a set percentage of the population and that a new optimization entered the Genetic Programming library.



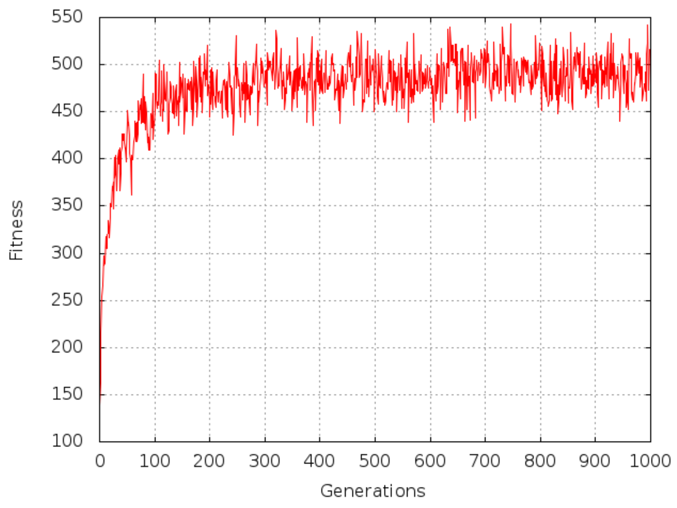
Pop. size:	400
Fitness runs:	20
Elitism:	0.05
Mutation:	0.15
Crossover:	0.65

Figure 5.4: Best average of population, 10.01.12

Now, instead of running every individual every generation, the library would use the previous score from the previous generation if the individual was preserved to the next generation either through copying or elitism. What the stepping represents are where new combinations of individuals or new beneficial mutations occur. The problem however is that these beneficial recombinations seem to happen slower and slower during the generations. Though, there might not be enough data points to say that.

The problem seems to be that rest of the population does not take benefit of these mutations and changes that are happening to the best of the population. We can see this if we look at figure 5.5. This is probably the overall problem in this experiment and why we don't see better scores. The average score of the population is much lower than that of the best in the population.

This is quite strange though and might be indicative of a larger problem that is in the Genetic Programming library and maybe a problem with the new optimization idea that was employed. What one might expect from a population like this is that a new beneficial change would spread rather quickly through to the overall population. Most likely the reason this is not happening is that the best scoring individual was lucky during one generation and since the score achieved at this point was not a good representation of the score the individual would normally



Pop. size:	400
Fitness runs:	20
Elitism:	0.05
Mutation:	0.15
Crossover:	0.65

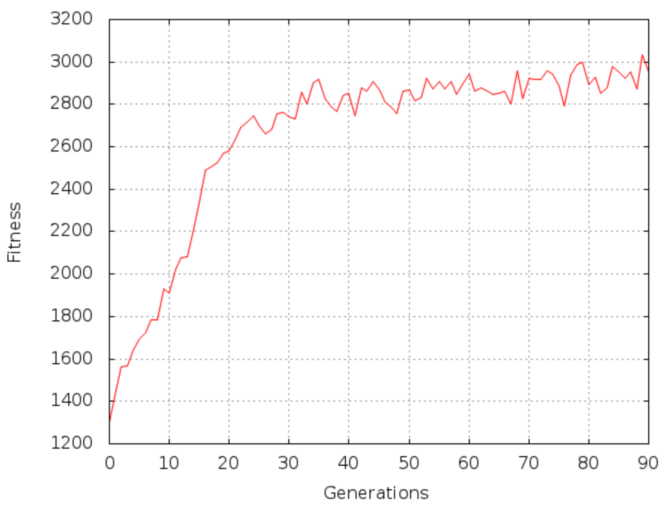
Figure 5.5: Average score of population, 10.01.12

achieve. Either the new optimization technique has to be removed, or the fitness test has to be run more times. In the end it as decided to remove the optimization technique as it running the fitness tests the required amount to get a good fitness score would be too slow.

5.4.4 Experiments in iteration 3

In the third iteration there was a drastic increase in the fitness of the individuals as we can see from figure 5.6. It also showed a much better curve of improvement than what has been seen before. The overall erratic behavior of the average population is also gone. This is indicative of a healthier population than what has been achieved before.

We can still see the sharp rise in the beginning generations, but it does not taper off as fast as it used to do in the past. It resembles much more the usual development we see in Genetic Programming. When we look at the best individuals we see the same; there is a steady increase in fitness over the generations. We still see some erratic behavior in the best individuals, but that is generally to be expected. Especially since each individual is only ran a handful of times to find the average fitness. This would probably be much smoother if we ran each individual more times.



Pop. size:	400
Fitness runs:	5
Elitism:	0.05
Mutation:	0.15
Crossover:	0.75

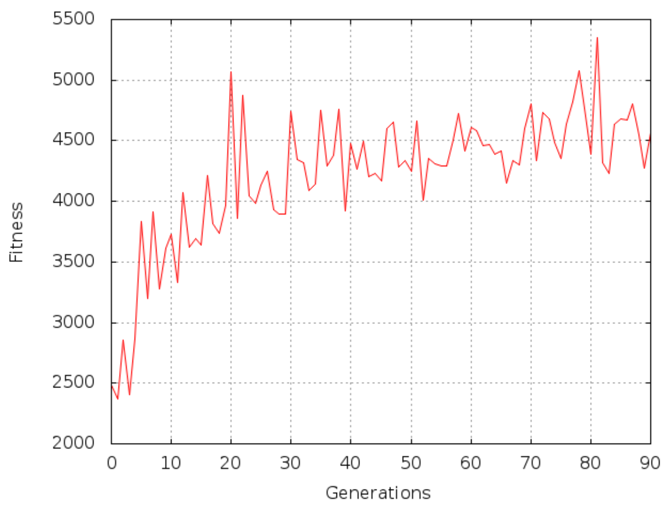
Figure 5.6: Average score of population, 19.04.12

In this experiment the long standing bug where the agents had to time their change of direction to the exact corner or risk not being able to turn, is fixed. I would like to say that we can see this from the data, but since there is such a huge difference in the overall way the programs function from the previous generation to this one that would be a stretch. We can however make the observation that the generations seem to be running much smoother if we look at the shape the graphs, than what we have seen earlier. A large improvement has also been made on the scores achieved by the agents. The starting score is now around 2,000 points, and we can see a peak of above 5,000 points.

It might be wishful thinking, but if the overall trend continues in a way that one could expect from Genetic Programming, we would be able to start seeing scores that rival that of other algorithms used to create solutions for this problem.

5.4.5 Experiments in iteration 4

During iteration 4 two experiments were conducted. The first one was with the same as in iteration 3, but with some bug-fixes. The second was a new version of the ones conducted in iteration 2. This was to show that the idea was not wrong, but that it was a problem with a long standing bug that kept it from becoming better.



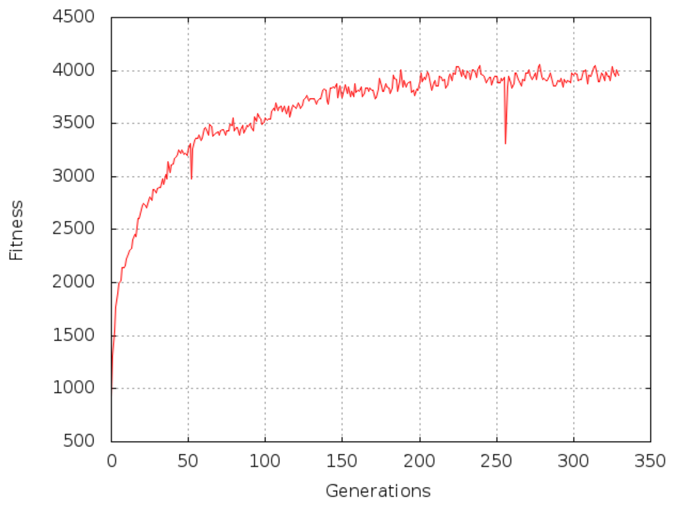
Pop. size:	400
Fitness runs:	5
Elitism:	0.05
Mutation:	0.15
Crossover:	0.75

Figure 5.7: Best average of population, 19.04.12

Experiment 1

Looking at figure 5.8 it looks pretty much as a continuation of figure 5.6 in the experiments in iteration 3. This is perfectly acceptable as they are based on the same code, except for a couple of bug-fixes. The most prominent one being the one where the agent could go into an infinite loop on the map preventing the game from finishing. What we see is that the population is in a steady rise up until around 3,700 points where it seems to taper off. This seems to go against the expectations from iteration 3 where it was suggested that this configuration should be able to get as good scores as what other researchers have achieved. If we however look at the score of the best individuals in figure 5.9 we can see that the best recorded individual is almost at 10,000 points with 9,848 points. Which scores better than most others in the competition, except for the one held in 2011.

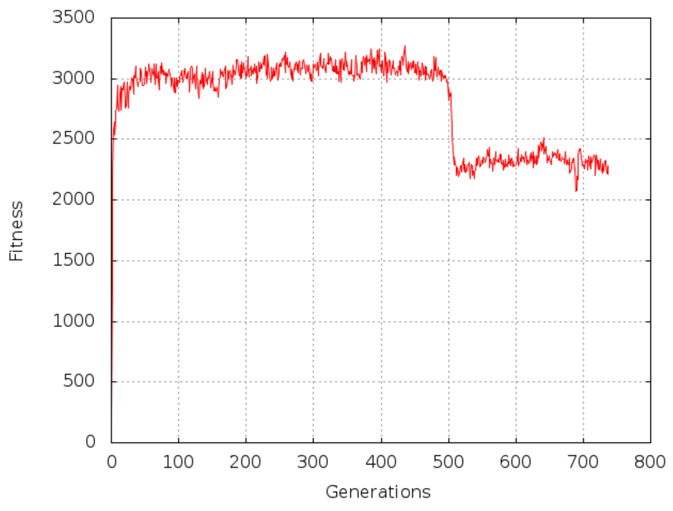
When looking at the individuals in the population for this run it is evident what has happened. The population has lost almost all diversity and consists mostly of the same individual in different configurations. If this is the main problem of the population or if there are other limitations that prevent the population from getting better needs further experimentation.



Pop. size:	400
Fitness runs:	5
Elitism:	0.05
Mutation:	0.15
Crossover:	0.75

Figure 5.8: Average score of population, 18.07.12

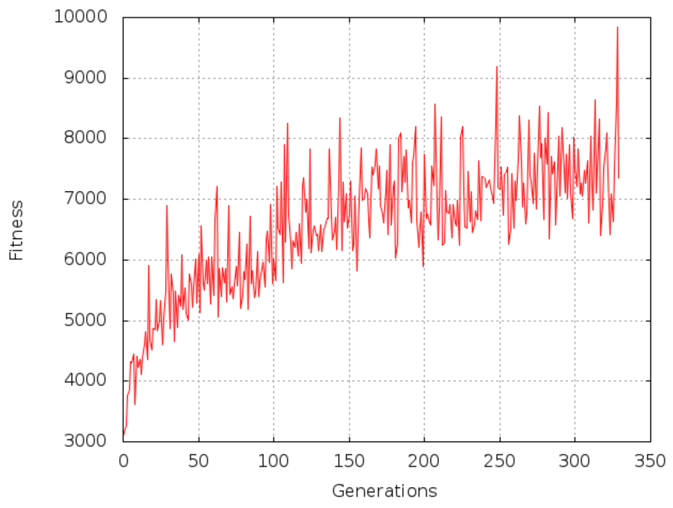
Experiment 2



Pop. size:	300
Fitness runs:	8
Elitism:	0.00
Mutation:	0.15
Crossover:	0.75

Figure 5.10: Average score of population, 23.07.12

For the second experiment the individuals where changed back to how they where made in iteration 2. This was to confirm that the previous approach was also valid after the bug fix. If we look at figure 5.10 we can see that the individuals rise very sharply and then halts at around 3,000 points. This is probably because of a too



Pop. size:	400
Fitness runs:	5
Elitism:	0.05
Mutation:	0.15
Crossover:	0.75

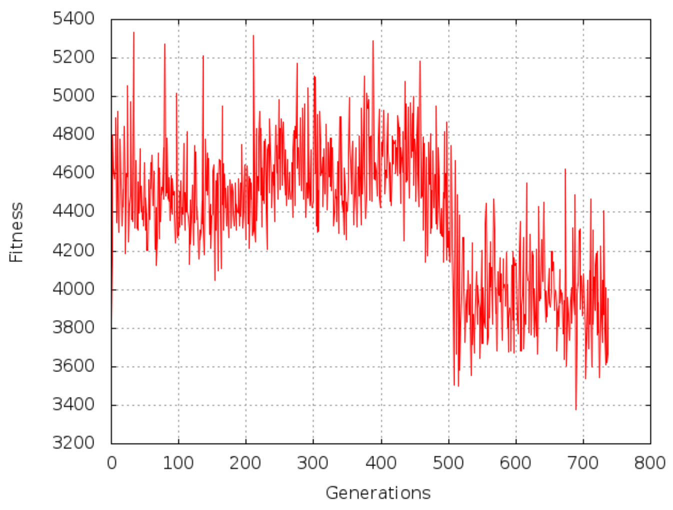
Figure 5.9: Best average of population, 18.07.12

small function set that does not contain all of what the agent needs to do any better. It does however show that the idea has potential, even though there obviously are some problems that need to be addressed.

We also see a pretty serious dip around 500 generations. We will get back to that after looking at the best individuals in figure 5.11. In this graph we can see some wild fluctuations in the best individuals, and the same dip that we see in figure 5.10. The reason for the fluctuations is probably because of two things. The first is that the fitness test is run too few times. The other is that this experiment was done without any elitism. The mutation rate and reproduction rate is also changed for this experiment. The reasoning behind these changes is that the previous experiment had problems with diversity in the population. Making sure copying into the next generation happens rarely or not at all while crossover and mutation happens much more can help create a more diverse population.

And that gets us to the reason for the dip at the end of the graphs. I believe it is simply because we do not have any elitism to keep the best individuals in the population over generations and the population was *unlucky* and lost a lot of the performing individuals.

Overall the graphs in this experiment has a rather strange shape. This is probably also because of the too small function set and that the GP library managed



Pop. size:	300
Fitness runs:	8
Elitism:	0.00
Mutation:	0.15
Crossover:	0.75

Figure 5.11: Best average of population, 23.07.12

to create one of the highest performing individuals possible when randomly creating individuals for the first generation.

This experiment needs to be run again with a larger function set and more fitness tests to truly validate if the strategy will be useful.

Chapter 6

Conclusion and further work

This chapter will sum up the results from the research and look at how it relates to the goal of the thesis. It will also look at what the results from the Ms. Pac-Man problem shows. After that it will discuss what there is to improve on in the research and areas where further research could go deeper then the focus of this study. I will also speculate on what I could have done different knowing what is known now.

6.1 Conclusion

I have created a Genetic Programming Library that creates agents for the old Ms. Pac-Man arcade game and tries to best other people that have tried the same using other techniques. I have created a way for the GP created agents to access and find the different objects within the Ms. Pac-Man game. A distribution framework for the genetic programming library has also been made so that it is possible to utilize the processing power that is available through the linux machines that are standing idle throughout the university.

In the experiment section I presented an individual that performed almost at the level of current competitive play with an average over 5 runs of 9,848 points. Looking at the scores the agent received we see that it has managed to get a top score of 11,890. This is almost 1 third of the way to the best scoring agent in the system with 36,280 points, but still almost 2,000 points below the least scoring agent with its highest score of 13,700 points. The interesting part though is that my average is still higher then the least scoring agents average of 7,955 points, and close to two of the others who achieved averages of 11,448 and 11,305 points.

Even though the results from this thesis is not better then the currently top scoring agents, it can still show agents that are scoring much better then what the first agents that where created and that has a higher average score then one of the agents from the latest competition. If we look at previous competitions we can see that my agents are better then many of them. I therefore believe that it is possible to say that I have built a system that shows that Genetic Programming is a valid technique for creating agents that play non-deterministic games. A long standing bug really prevented research from going any further. Had the bug been discovered earlier I might been able to show better results. The problem here really came down to time constraints. If more time had been available there might have come results out of the research that could have rivaled the highest performing algorithms that have been used to solve the problem.

6.2 What could have been done differently?

Looking back at the development of the system I can see mistakes that could have been avoided had I known what I know now. Firstly some of the work could have been avoided in that I found a competing Genetic Programming Library written in Clojure after most of the development for my version was written. There had been an extensive search done for Genetic Programming and Clojure in the preliminary research for the thesis, but somehow this library had escaped my attention. This was not a total loss as I now have a much better understanding of the implementation specific problems of Genetic Programming. There is also a difference in how the two libraries have been implemented. E.g.: In my library multi-argument functions are allowed, but not in the other library where the number of arguments has to be explicitly stated. The workaround to this problem is to define the same function with all the different number of arguments that are needed. The most important thing here however is that having implemented my own library meant that I know all the implementation specific details. This allows me to prototype new features as they are needed much faster then what I would have otherwise been able to.

Another thing I should have done was to spend more time in the beginning looking at how the emulator works. The problem that emerged from this mistake haunted the project all through the development. Discovering the bug that has really impeded the score of the agents I believe would have had a profound effect on the results of the project. In the end though, it is not automatically true that the bug would have been found even if I had spent more time working on the emulator. What should have been done though, is to look deeper into the root of this problem before making the assumption that it had been solved. The problem however was that at the time it seemed that it was my own code that was the problem and not the implementation of the emulator.

The distribution through the clojure-control library should also have been dropped much earlier in favor for the current system. It is much simpler and has a much larger tolerance for failure than the system based on clojure-control. The problem in itself was not clojure-control, it did its job beautifully, the problem was the instability of the ssh servers and the authentication issues through Kerberos.

6.3 Further work

In the future, if one wants to continue the work that this thesis represents, there is a couple of interesting points that one could look at. There currently two immediate paths that future research could explore. The first is to continue the research on the agents that use configurations of A* to find its way around the map from iteration 3. The other is the strategy from iteration 2. Both have shown great promise. A problem however, is that the emulator and fitness function is quite slow. If any of these strategies are to be pursued some effort should be put into making these parts of the program faster. It would be beneficial to be able to run larger populations faster.

Another thing would be to use the Genetic Programming framework in other situations to see if it can perform better there. It might require some work to replace the areas of the code that is dependent on the current problem, but it should not take too much time. Something that should be easier is to test it with another Ms. Pac-Man competition were the contestants either program Ms. Pac-Man (as I have done already) or program the ghosts. It would be interesting to see how the current programs do in this situation instead of with just the standard ghosts.

It would also be interesting to revive the effort to run the agent and the game simultaneously in a different thread. This could ensure that the agents get more time to run and could therefore potentially make better decisions. It would also guarantee that the agent will not slow the game down. The problem however is that it would also increase the time needed for every fitness test to run. It is however possible to somewhat mitigate this problem by running the game at normal speed until it has received a direction for that frame and then stop waiting to create the next frame.

One could also use the stripped down emulator to test other methods to solve the Ms. Pac-Man problem. An interesting method could be to use Neural Networks, especially in combination with Genetic Programming. If it at the same time is possible to program the Neural Network for General-purpose computing on graphics processing units[1] (GPGPU), then it would be possible to fully utilize the parallel nature of Neural Networks. This could serve as a way to cut the processing time of the API and agent dramatically allowing for more processing for the decision making process. Neural Networks are also excellent in combination with Genetic Programming. The only things that has to be done is to change the primitive set to neurons and weights instead of functions and terminals.

Another interesting approach could be within the immune algorithms. Inspired by the biological immune system, immune algorithms tries to protect a host organism from pathogens and toxic substances. Treating the ghosts as pathogens and Ms. Pac-Man as the organism. The immune algorithms were not designed for this sort of task, but could prove a novel way of using the methods. The algorithm that is best fit for this kind of task might be the Clonal Selection Algorithm, which was designed as a general machine learning algorithm that has already been applied to pattern recognition, function optimization and combinatorial optimization (Brownlee, 2011).

In the end we can see that even though the library is capable of creating agents that are performing at a level close to other researchers agents, there are plenty of improvements that can be done on the research in the thesis and several new directions available for creating agents in non-deterministic games.

[1]Also know as graphics cards.

References

P. J. Angeline. Subtree crossover: Building block engine or macromutation? In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17. Morgan Kaufmann, 13-16 July 1997.

T. Back, D. B. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*. IOP Publishing Ltd., Bristol, UK, UK, 1st edition, 1997. ISBN 0750303921.

T. Blickle and L. Thiele. A mathematical analysis of tournament selection. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 9–16, 1995.

M. Brameier and W. Banzhaf. *Linear genetic programming*. Springer-Verlag New York Inc, 2007.

J. Brownlee. *Clever Algorithms: Nature-Inspired Programming Recipies*. Lulu Enterprises Uk Ltd, 2011.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

M. Emilio, M. Moises, R. Gustavo, and S. Yago. Pac-mAnt: Optimization based on ant colonies applied to developing an agent for Ms. Pac-Man. In *IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 458–464. IEEE, 2010.

P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

J. H. Holland. *Adaptation in natural and artificial systems*. University of Michigan press, 1975.

J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-11189-6.

J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane. Four problems for which a computer program evolved by genetic programming is competitive with human performance. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, volume 1, pages 1–10. IEEE Press, 1996.

W. B. Langdon, R. Poli, N. F. McPhee, and J. R. Koza. Genetic programming: An introduction and tutorial, with a survey of techniques and applications. In John Fulcher and Lakhmi C. Jain, editors, *Computational Intelligence: A Compendium*, volume 115 of *Studies in Computational Intelligence (SCI)*, chapter 22, pages 927–1028. Springer-Verlag, 2008.

S. M. Lucas. Evolving a Neural Network Location Evaluator to Play Ms. Pac-Man. *IEEE Symposium on Computational Intelligence and Games*, 2005.

S. M. Lucas. IEEE CEC 2007 Results. `http://dces.essex.ac.uk/staff/sml/pacman/CEC2007Results.html`, 2007. Accessed: 2012-07-27.

S. M. Lucas. IEEE CIG 2011 Results. `http://dces.essex.ac.uk/staff/sml/pacman/CIG2011Results.html`, 2011a. Accessed: 2012-07-27.

S. M. Lucas. Ms Pac-Man Competition. `http://cswww.essex.ac.uk/staff/sml/pacman/PacManContest.html`, March 2011b. Accessed: 2012-03-25.

B. McKay, M. J. Willis, and G. W. Barton. Using a tree structured genetic algorithm to perform symbolic regression. In A. M. S. Zalzala, editor, *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, volume 414, pages 487–492, Sheffield, UK, 12-14 September 1995. IEE.

D. Michie. Memo functions and machine learning. *Nature*, 218(5136):19–22, 1968.

P. Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, 1991.

R. Poli and W. B. Langdon. On the search properties of different crossover operators in genetic programming. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 293–301. Morgan Kaufmann, 22-25 July 1998.

R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Lulu Enterprises Uk Ltd, 2008a. ISBN 1409200736.

R. Poli, N. F. McPhee, and L. Vanneschi. Elitism reduces bloat in genetic programming. In Maarten Keijzer, Giuliano Antoniol, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Nikolaus Hansen, John H. Holmes, Gregory S. Hornby, Daniel Howard, James Kennedy, Sanjeev Kumar, Fernando G. Lobo, Julian Francis Miller, Jason Moore, Frank Neumann, Martin Pelikan, Jordan Pollack, Kumara Sastry, Kenneth Stanley, Adrian Stoica, El-Ghazali Talbi, and Ingo Wegener, editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1343–1344. ACM, 12-16 July 2008b.

D. Robles and S. M. Lucas. A simple tree search method for playing ms. pac-man. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 249–255. IEEE, 2009.

M. Schoenauer, M. Sebag, F. Jouve, B. Lamy, and H. Maitournam. Evolutionary identification of macro-mechanical models. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 23, pages 467–488. MIT Press, 1996.

W. R. Stevens, B. Fenner, and A. M. Rudoff. *UNIX Network Programming: The Sockets Networking API*, volume 1. Addison-Wesley Professional, 2004.

I. Szita and A. Lõrincz. Learning to play using low-complexity rule-based policies: illustrations through Ms. Pac-Man. *Journal of Artificial Intelligence Research*, 30(1):659–684, 2007. ISSN 1076-9757.

Appendix A

Listing of code

A.1 fitness

```
(defn fitness [tries code]
  (binding [msp (Game.)]
    (loop [score 0
           times 0]
      (if (or (<= tries times)
              (and (<= 3 times)
                   (= (/ score times) 120)))
        (int (/ score times))
        (do (.start msp)
            (.update msp)
            (recur (+ score
                      (do (while (not (.isGameOver msp))
                             (eval `~code)
                             (.update msp))
                           (.getScore msp)))
                   (inc times)))))))
```

A.2 `expand`

```
(defn expand [exprs depth]
  (if (or (symbol? exprs)
          (number? exprs)
          (empty? exprs)
          (< depth 1))
    (atomize exprs)
    (cons (first exprs)
          (loop [terms (rest exprs)
                 acc ()
                 expr-width (rand-int MAX-STARTING-WIDTH-OF-EXPR)]
            (if (empty? terms)
              acc
              (let [term (first terms)
                    exp (case term
                          (expr expr+)
                          ,(expand (rand-nth ind/FUNCTION-LIST)
                                   (dec depth))
                          expr?
                          ,(if (< (rand) EXPR?-RATE)
                             (expand (rand-nth ind/FUNCTION-LIST)
                                     (dec depth))
                             ())
                          point
                          ,(expand (rand-nth ind/POINT-LIST)
                                   (dec depth))
                          ,(atomize term))]
                (recur (if (and (= term `expr+)
                                (pos? expr-width))
                         terms
                         (rest terms))
                       (concat acc (list exp))
                       (dec expr-width)))))))
```

A.3 Server/client

```
(defn- send-inds-to-machine [individuals machine]
  (try
    (let [socket (Socket. (format ``%s.klientdrift.uib.no''
       machine) 50000)
          rdr (LineNumberingPushbackReader.
                (InputStreamReader.
                 (.getInputStream socket)))]
      (try
        (binding [*out* (OutputStreamWriter.
                          (.getOutputStream socket))]
          (prn individuals))
        (read-string (.readLine rdr))
        (finally
         (shutdown-socket socket))))
    (catch Exception e nil)))
```

```
(defn- run-fitness [ins outs]
  (println ``Run-fitness'')
  (let [rdr (LineNumberingPushbackReader.
              (InputStreamReader. ins))
        inds (read-string (.readLine rdr))
        out (gp/run-fitness-on inds)]
    (println ``finished'')
    (binding [*out* (OutputStreamWriter. outs)]
      (prn out))))
```