Development of AI Ms. Pac-Man Controllers

Modern AI for Games, Fall Semester 2013, IT-University of Copenhagen

Julian Møller (jumo@itu.dk)

Abstract— **In this paper, successful implementations of two Ms. Pac-Man controllers and unsuccessful implementation of one Ms. Pac-Man controller is described. The methodologies used are genetic evolution of state machines, Monte Carlo Tree Search for Microplanning and Neural Network with multilayered perceptron. The results of the implementations as well as their shortcomings are described. Comparisons are be made with the benchmark controllers *StarterGhosts*, *Legacy* and *Legacy2TheReckoning*. Maximum scores of approximately 14.000 (with means around 6.500 points) are obtained against the toughest ghosts.**

I. Introduction

In this paper I will describe the controllers I have developed for the real-time arcade game Ms. Pac-Man (henceforth named Pac-Man).

I will first describe the state based controller which was evolved via genetic programming. Then, I will describe the Monte Carlo Tree Search controller and lastly, I will describe the controller backed by a multilayer perceptron network. I will then present my results. To end with, I will discuss and evaluate the results as well as the project in general.

II. Genetically Evolved State Machine

The basis of the genetically evolved state machine controller (the SMC) is the division of labour into the following distinct states:

- Go to pill
- Go to powerpill
- Hunt ghosts
- Go to safest node

The states change the behaviour of Pac-Man. Based on the state of the game, a preferred state is chosen, and a move is performed based on the analysis made by the state controller.

To determine which state should be used, all states implement a method which is used to find out if the state thinks it can handle the current game situation. The states also have a priority which changes the order in which the states are checked for validity.

The behaviour states all contain a number of parameters. These are used both to determining if the state can handle the current game situation, but also when an appropriate move is calculated.

The parameters used for the different states are:

- **Go to pill**: Priority, distance to nearest ghost, distance to nearest pill
- **Go to power pill**: Priority, minimum distance to nearest ghost, maximum distance to nearest ghost, distance to nearest power pill
- **Hunt ghosts**: Priority, minimum remaining edible time, distance to nearest edible ghost
- **Go to safest node**: Priority

Evolution

Both the priority of the states as well as their individual parameters are evolved during an evolution experiment.

The evolution of the parameters was performed by creating a population of 100 genomes. Each iteration these were evaluated. Of these, the 50 worst were eliminated. They were replaced with 20 entirely new genomes, 20 mutations of the best 20 genomes and 10 crossovers based on pairs from the 20 best genomes. 100 evolutions were run. The average score assigned to a genome was calculated based on the average of the previous iterations and the average of the newest iteration.

The different numbers for how the population should be "split" were determined in part by trial and error. I started with a set of values, ran an experiment. Then I changed a value and ran the experiment again. If the average score went up I either kept the value or changed it more in the same direction. I did this a couple of times until it seemed as if the average score had stabilized.

Problems and ways to improve

The controller was evolved against single opponents. This led to overfitting of the problem, witnessed by the fact that none of the evolved controllers were consistently good against all other controllers. They were too specialized with regards to the ghost behaviour.

On top of that, the search space for the genome too large. Every part of the genome started in the range 0-200. For some of the parts of the genome this might not have been a sufficiently large range, whereas for others the range was too big. The result of this is that ranges of values for the genome parts might have been superfluous and a more efficient search could have been made.

The states could perhaps also perhaps have been implemented so the result of the planning was a more long term strategy instead of a "greedy" short term approach.

The parameters for the evolution itself (how a population should be composed) could have been found with a more rigorous approach instead of the one I used. The approach could perhaps even have been to run an automated search for optimal parameters, maybe even with running an evolution on the evolution parameters.

III. Monte Carlo Tree Search

Monte Carlo Tree Search is a 'best first' way to get an estimate of the results of actions in a search space, where the space is too large to construct a full tree of actions and outcomes.

It works by recursively selecting actions and playing out their outcomes based on random actions. One of the advantages of Monte Carlo Tree Search is that the algorithm can be terminated at any time, because the current best choice will always be known. Given infinite time, the outcome simulation will converge towards an optimal solution. For Pac-Man this is an advantage, because the allotted time can be fully utilized.

My implementation uses the default policies (*TreePolicy* and *DefaultPolicy*) for expansion and playout. The implementation constructs a node for each possible move for Pac-Man. The moves are based on neighbouring, short moves Pac-Man has available to her at a given position.

The *DefaultPolicy* is modified a bit from completely random playout. Pac-Man is only allowed to go straight ahead when in corridors, except when a ghost has just been eaten or when a ghost is very near by. Otherwise, all direction changes take place at junctions. Pac-Man isn't allowed to go backwards at junctions either, to avoid the search space becoming larger than it already is.

Problems and ways to improve

Because the nodes in the search tree is based on individual moves, Pac-Mans strategy becomes a micro-strategy and doesn't allow for Pac-Man to look properly ahead in the future. The first an best way to improve the implementation is to change the tree from being based on moves to being based on junctions like described in [2]. This would decrease the search space considerably, thus allowing Pac-Man to better plan her moves.

Another good way to improve the implementation is to change the selection of child nodes to use a heurestic function instead of relying on randomness.

When the "lowest hanging fruit" has been dealt with, strategies could be implemented, trying to handle specific situations such as pincer moves[1], endgame situations with few pellets as well as other situations.

IV. Multilayered Perceptron Network

My implementation of this controller succeeded. But the behaviour generated by the controller was not successful. I created a neural network that successfully learned various boolean functions[2].

However, when I tried to train the network to play Pac-Man based on data recordings of myself, nothing really happened. Because I got the neural network working very late, I didn't have time to find the errors. I also didn't have time to network run for longer periods to see if the problem was merely too short a maximum training time[3].

So, this is a description of what I did, which unfortunately won't be accompanied by any results.

Multilayered perceptron networks is a machine learning technique, where a number of inputs are passed through a network of sums, weights and functions that map the values onto the $[0; 1]$ domain. The result is a single or a number of numbers, that (hopefully) represent some learned knowledge.

This can for instance be utilized in Pac-Man to determine a 'goodness' score for a given state, and by feeding the neural network each possible move as a distinct state, determine which is the better action to perform. A neural network can also be used to determine which state it is favorable for Pac-Man to be in (such as flee, eat pill, seek power pill or hunt ghost).

A general problem with a MLP network is determining the topology of the network, as there is no known way to determine beforehand if a topology will be a good fit for a certain problem space. This could be 'solved' by utilizing genetic programming and letting an algorithm evolve the topology based on the achieved scores in the game.

Another problem is that MLP networks suffer from overfitting, which requires multiple distinct sets of data for training, test and verification. To get consistent results, this requires a consistent controller used for training.

Implementation

The neural network is implemented directly based on the algorithms found in [1] and [3].

I tried to make the neural network have four outputs, each representing a direction that Pac-Man can take. The inputs are everything that is available from the supplied `DataTuple` class:

- Distance to every one of the four ghosts
- Whether each of the four ghosts are edible
- The number of pills left
- The number of power pills left
- The current score
- The current level time
- The total game time

This represented my input layer. Left, is the hidden layer which for number of nodes had the mean of the number of input nodes and number of output nodes.

Primary problems

Had I gotten the MLP controller to function at an earlier point in time, I would also have liked to explore the following possibilities for input values:

- Is Pac-Man at a junction?
- Current score
- Number of ghosts eaten
- Is (the nearest) ghost ahead of or behind Pac-Man
- Where in the maze is Pac-Man?

I would also have liked to run the training for longer periods of time, to see if my problems would solve themselves, if the training process was merely slow.

Yet another problem could possibly be my topology, more specifically the number of output nodes. I also considered having the network try to classify the "goodness" of a state, instead of having it try to predict the road chosen with four different outputs. The network would have been fed each of

[1]When the ghosts are about to approach Pac-Man from all possible directions, thus trapping her.

[2]AND, OR, NOT, NOR, XOR, T, F

[3]Which I had at 15 minutes

the possible game states, *after* a Pac-Man move had been simulated. The different scores from the network would then have been used to determine the best action.

V. Results

A. Evolution of genomes for the SMC

Figures 1, 2 and 3 show the average score of the controller as the evolution progressed. The controller reached a quite stable score against all three ghosts. The StarterGhosts were more forgiving with regards to genome parameters than the Legacy and Legacy2 ghosts, as the average score was quite high to begin with. For all three simulations, a 'maximum' fitness was reached quite early.

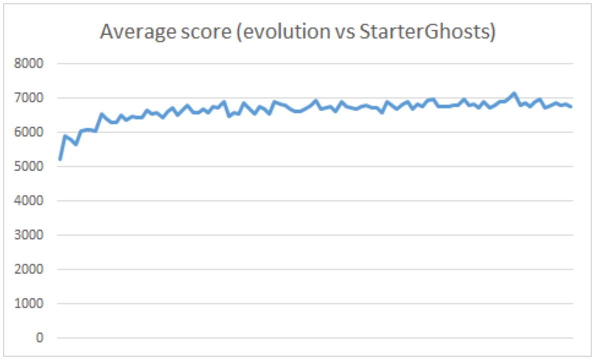

Fig. 1. Evolution vs. StarterGhosts

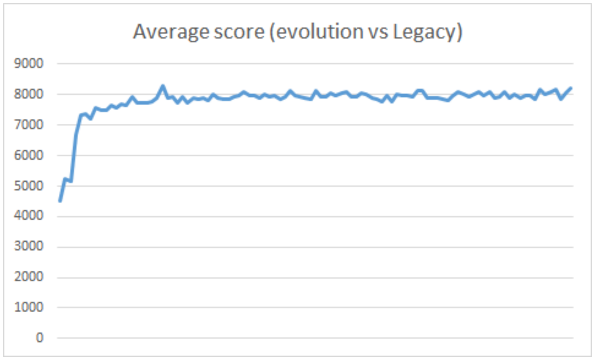

Fig. 2. Evolution vs. Legacy

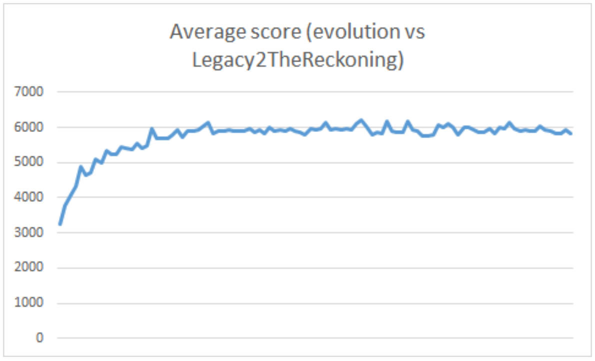

Fig. 3. Evolution vs. Legacy2TheReckoning

B. Learning rate for the neural network

Figure 4 show the mean squared error for the different learning rates 0.2 and 0.5. Both errors stabilize very fast, but at different levels. That the error is lower for a lower learning rate is to be expected.

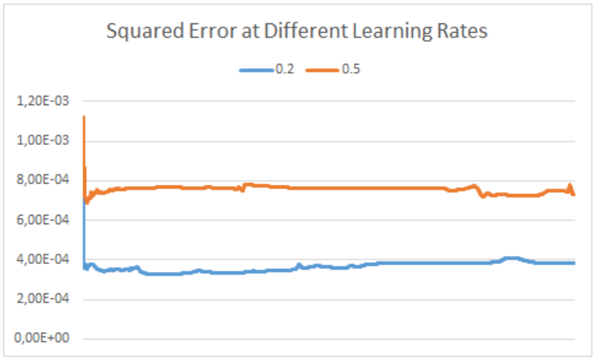

Fig. 4. Mean, squared error for different learning rates

C. Performance against ghosts

Figure 5 shows the performance against the starter ghosts. *S*, *L* and *L2* represent that the values for the genome are the ones found to be best against StarterGhosts, Legacy and Legacy2TheReckoning, respectively.

The most stable performer is the MCTS controller, but due to its micro- instead of macro-strategy, it doesn't score very high. All three genomes for the state machine performs better than StarterPacMan, but both StarterPacMan and all three state machines have a quite large standard deviance. The L2 version performs about as well as the version evolved against the ghosts specifically, but has a higher standard deviation.

Figure 6 show the same pattern: The controllers perform about the same, with an advantage for the SMC that was specifically evolved to handle the Legacy ghosts.

Another picture is painted in figure 7, where the SMC specifically evolved show a much higher average score than the rest of the controllers. I believe this can both be because the ghosts have a very distinct behavior, but also because the ghosts are less forgiving, so a specialized version will perform better.

Figure 8 show the average scores for the MCTS controller based on different values for the C-component, that determine the rate of exploration vs exploitation when the search tree is traversed. The most solid value is $C = 0.2$, which displays the lowest variance of all of my controllers, but an optimal C-value requires more extensive testing than I have performed.[4]

VI. Conclusions

I set out to create a SMC that performed better than the starter ghosts, which I did, so I am happy about that. I would have liked to create a macro-based MCTS controller instead

[4]Because the MCTS utilized all available time given, the games were very slow, so it wasn't feasible to run that many simulations.

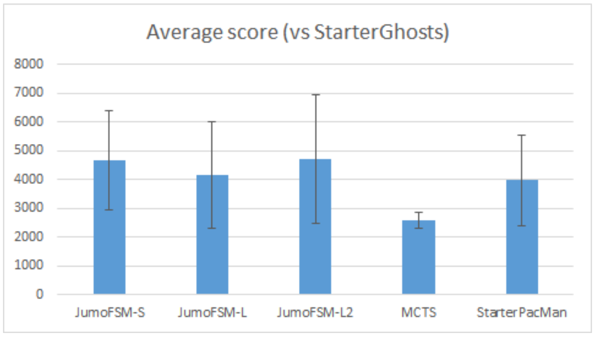

Fig. 5. Average score against StarterGhosts

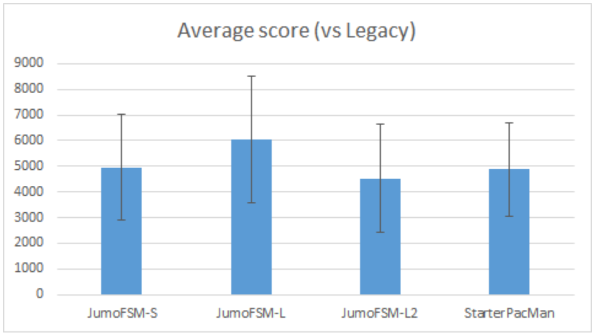

Fig. 6. Average score against Legacy

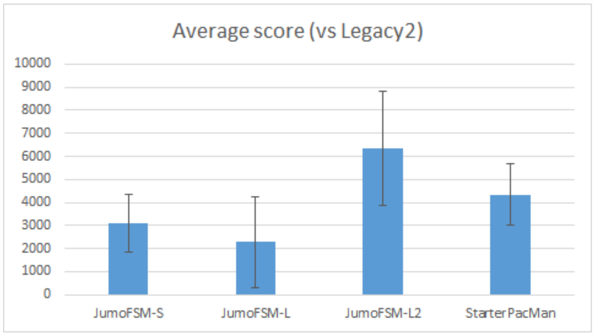

Fig. 7. Average score against Legacy2TheReckoning

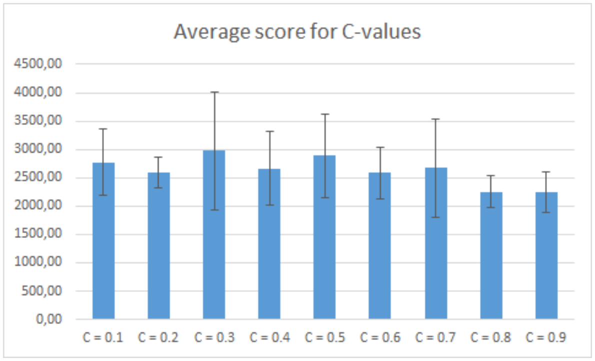

Fig. 8. Average score against StarterGhosts for different C-values

of the micro-based one I implemented, to score better, but I am happy with the high stability and low standard deviance the controller exhibited.

I am not satisfied, that I couldn't get the MLP network to work, as that is the solution I think is the most interesting. I underestimated how long time it could take to debug the MLP implementation. I also wasted too much time on improving the state based controller, that I should have used on the MLP and on making the MCTS better.

References

[1] S. Russel, P. Norvig *Artificial Intelligence - A Modern Approach*. Pearson, 2010.

[2] T. Pepels, M.H.M. Winands, "Enhancements for Monte-Carlo Tree Search in Ms Pac-Man,"

[3] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques (3rd ed.)*, 2011. Morgan Kaufmann Publishers Inc.