MAIG Individual Assignment

Mikkel Stolborg
Games and Technology
IT-University of Copenhagen
Copenhagen, Denmark
Email: msto@itu.dk

***Abstract*—The abstract goes here.**

1. Introduction

This report contains the dissertation of 3 methods used to create an AI for the *Ms. PacMan Vs Ghosts* framework. The 3 methods in question are an AI using a *Behaviour Tree*, one using a *Monte Carlo Tree Search* algorithm, and one using a *State Machine* optimized by a *Genetic Algorithm*.

For each of the methods and their algorithms I will detail the approach to the problem, which considerations were made before tackling the task at hand, the parameters of the algorithm, the parameters the final algorithm uses for the AI, and the performance measure of the algorithm, the scores the AI achieves over a number of games, totalling the average value, the minimum, and the maximum score.

Finally I will compare the performance of each of the AI's to the performance of the AI' included in the *Ms. PacMan Vs Ghosts* framework.

2. Behaviour Tree

In this section I go over the approach for the behaviour tree based AI, the parameters the AI uses, and how well it performs on average.

2.1. Description of Approach

A behaviour tree is a tree structure that you traverse using its own logic to find a action to take. The nodes in the ends of the tree are called leafs, and they decide which action the AI takes. The structure of the tree contains several types of nodes which helps run the correct leafs and executing the correct action. Each of the leafs returns one of 3 values, running, failure, or success. The leaf executes its code and depending of whether it failed, no ghost nearby, succeed, ghost are nearby, or running, when it is still processing its action, moving towards nearest pill. Each of the other nodes are controlled by the responses from the leaves. The categories for the nodes, other than leaf, are *Decorator* and *Composite*.

The composite node is capable of having several child nodes. A good example of a composite node is the sequence node. The sequence node runs through all of its children, starting from the first added, until it reaches the end or when a child fails. On failure of a child the sequence returns failure, it returns running if the current child it is investigating returns running, or success if all its children returns success. This type of sequence could be described as an and-loop. There exist the opposite loop, which could be described as an or-loop.

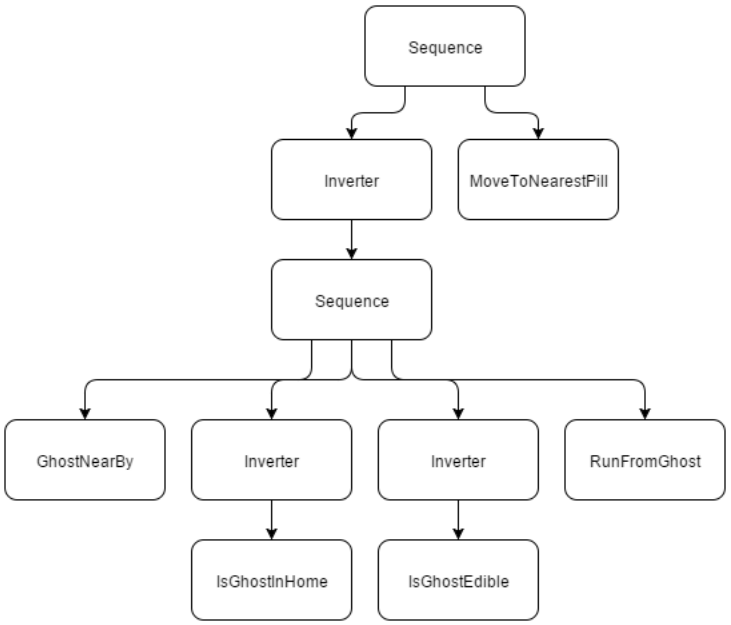

Figure 1. Behaviour tree for final AI. The top sequence is the entry point. The left branch returns true if there are no ghost nearby.

The decorator node is capable of having only a single child. A good example of a decorator node is the inverter used in our algorithm. The inverter takes the input from its child and simply inverts it, returning success if the child failed or failure if the child succeed.

The strength of a behaviour tree is its modulation. Each of the leaf can be used independently of each other, and adding steps and conditions before an action, is merely the result of adding new leafs in between the existing ones.

The first approach was to create a simple AI, which simply tried to go to the nearest pill and eat it. This was

achieved just by having a leaf in the tree which set the move of the tree result to send PacMan to the nearest available pill. This leaf was called MoveToNearLeaf.

Now this AI obviously did not perform that well, as it completely ignored the ghost. In order to improve this I changed the root of the tree to a sequence and added logic to check for ghosts. The branch for getting the pills remained unchanged and the new branch for the ghost detection was added. The ghost detection were comprised of a sequence which would first check if there were ghost nearby, then if it were true, it would run from the ghost.

This improved upon the AI, however, it still did not behave as expected. It kept running from ghost, even edible ones and the ones in the lair. In order to fix this two new leafs were created, one to check if the ghost were edible and one to check if they were within their lair. The final tree can be seen in figure 1.

2.2. Algorithm Parameters

The only parameter the algorithm uses is the distance of how close the ghost is allowed to be, before the AI will run from it. The parameter were set from the same as what the starter AI used, which proved to be efficient enough for the AI to execute its logic.

2.3. Performance measure of the algorithm

In order to test the performance of the AI algorithm, we run 10 simulated games and collect the data of the experiment. The results can be seen in table 1.

Max Score	5560
Min Score	2010
Average Score	2914.0

TABLE 1. PERFORMANCE OF THE BEHAVIOUR TREE. THE TABLE INCLUDES THE MIN SCORE, MAX SCORE, AND THE AVERAGE SCORE.

The performance of the AI varies a lot, and it does not perform that well. This can be attributed to the fact that the AI does not chase the ghost, which are a source for greater points than simply succeeding in gathering pills.

In order to improve the score of the AI, a branch could be included in the behaviour tree which is responsible for hunting and eating ghost when applicable. This could even be done using some of the leaves already in the current tree.

3. Monte Carlo Tree Search

In this section I go over the approach for the *Monte Carlo Tree Search* based AI, the parameters the AI uses, and how well it performs on average.

3.1. Description of Approach

A Monte Carlo Tree Search algorithm creates a partial tree of the entire game and from the limited knowledge choosing the best action to perform at the current state of the game.

The way it achieves this is as follows. First it takes the current state as the root node of the search tree. Then while the tree still has time to compute, use the *TreePolicy* on the node find the next node to investigate.

The TreePolicy expands all the possible moves from the current state of the game. Then it return the *BestChild* of the expanded nodes using an *Upper Confidence Bound for Trees* (UCT) algorithm to estimate which child is the best.

The algorithm then uses *DefaultPolicy* to determine a value for the chosen game node. This keeps making random choices till the game reaches an end state or some other predefined condition is achieved. Then it returns the reward of the end state to the main function. The reward is determined by what the developer wishes the AI to achieve. In this case the score achieved were used as reward.

Finally the algorithm walks back up the tree and adds the reward from the DefaultPolicy to each of the parent nodes in the *Backup* function.

The result of the algorithm is the BestChild of the root node based on the total reward propagated back up through the tree.

The way the algorithm were implemented were to set the reward from the DefaultPolicy to be the reward. This way we tried to optimize the path taking into account the greater the score the more pills and the more ghost would be eaten.

3.2. Algorithm Parameters

The algorithm's only parameter is the parameter for how much the algorithm favours exploration. The parameter were determined solely through trial an error, and was settled on the value of 10.

3.3. Performance measure of the algorithm

In order to test the performance of the AI algorithm, we run 10 simulated games and collect the data of the experiment. The results can be seen in table 2. The algorithm

Max Score	780
Min Score	300
Average Score	543.0

TABLE 2. PERFORMANCE OF THE MCTS AI. THE TABLE INCLUDES THE MIN SCORE, MAX SCORE, AND THE AVERAGE SCORE.

does not perform very well. This can be attributed to the fact that the AI is not punished significantly for being eaten by ghost. What is not shown in the table is the time it takes for the AI to die. It is actually able to avoid ghost because of the score wont increase if it is eaten. But the simple requirements does mean that it often just walks around in circles or back and forth. You could improve the AI significantly by punishing the reward if pacman gets eaten or if it keeps walking back and forth.

4. State Machine and Genetic Algorithm

In this section I go over the approach for the State machine based AI which were optimized using a Genetic Algorithm. I look in to the final parameters and how they were attained and I test how well it performs on average.

4.1. Description of Approach

First part of the AI were to create a suitable state machine. The machine had to take parameters which could be manipulated through a genetic algorithm. Second part were to create a genetic algorithm and have it optimize the variables available.

State machine works by having several states of actions, which jumps to the other states depending on different criteria. A special quirk with this state machine is that it first checks whether it should change state, rather than executing its own state first.

The chosen states were as follows, an overview can be seen in figure 2. RunFromGhost is, as the name implies, the state of running from the ghosts. If the distance to a power pill is within its limits it will try to run towards it, by changing state to MoveToNearestPowerPill. If it simply evades the ghost it will revert to MoveToNearestPill.

MoveToNearestPowerPill moves towards the nearest power pill. The state will revert back to RunFromGhost if a ghost comes to close to it, as in a new ghost appear to kill it. It can go to the EatGhost state when it achieves eating a power pill. It defaults back to MoveToNearestPill if there are no ghost close.

EatGhost tries to eat any nearby ghost, in effect chasing after them. The state will revert back to MoveToNearestPill, if there is no edible ghost nearby.

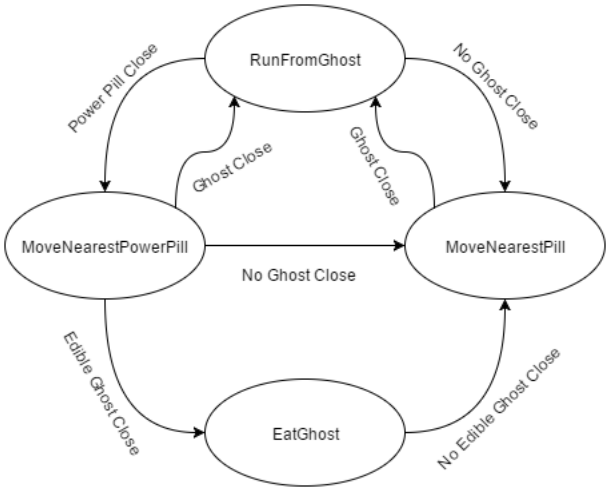

Figure 2. State machine diagram. The 4 different state of the machine and the conditions for the changes between them.

After the state machine were in place, a genetic algorithm were to optimize its parameters.

A genetic algorithm takes a population of genes and creates a new population based thereupon. The mechanisms for creating the new population varies, but mostly is done through reproduction and/or mutation. In this case we use both of the concepts as the simplicity of the genes, but the importance of the variables required both. The genes map to what is called a phenotype, which is what the real representation of the population, in our case the state machine. Each gene has an associated fitness value, the value which determines how well the gene does in its environment. In our case the score which the state machine achieved is used as the fitness. The population, a number of genes, is usually kept constant. This is done through discarding members after each cycle. This will be explained below.

Reproduction is when you take members of your population and use them to create new individuals. The number of individuals created and the method through which it is done usually varies from each algorithm. The members used are usually the fittest members of the current population, but for diversity one could also include some of the worst members.

The way members were reproduced in this case were through taking, at random, a piece of the chromosome from a parent and setting to be the child value. The reproduction method in this case also only creates one new member with two parents as input.

Mutation is when you take an individual in your population and make a small random change in it. In this case a gene's chromosome, chosen at random, is varied by either increasing it by 1 or decreasing it by 1.

In order to keep a constant population, usually only the fittest members survive each iteration of the algorithm. A gene can be discarded usually either due to age or to low fitness. In this case half of the population is discarded based solely on their fitness value.

The cycle of creating a new population is done until either a threshold in fitness is reached or a number of generations has occurred.

The algorithm were set to use the parameters of the state machine as its chromosome within its genes.

4.2. Algorithm Parameters

The genetic algorithm uses the parameters of the state machine as its chromosomes for gene representation. The parameters in question are those which decides when the states should be changed and are as follows:

DistFromNonEdible. The distance of non edible ghost. Used to determine when to run.
DistToPowerPill. The distance to the power pill. Used to determine if power pills are close enough.
DistToEdible. The distance to edible ghost. Used to determine if it is worth chasing ghost.

The parameters are randomized to a start value between 1 and 100. This max limit is chosen based on the general idea of the game size, and the minimum due to the requirement for it to be a positive non-negative number. Furthermore the value is limited within the iteration of the algorithm to never go below 1.

Gen	Avg. Fitness	Min Fitness (Chrom)	Max Fitness (Chrom)
0	3286.7	1255.0 ([31,46,4,])	6228.0 ([50,46,39,])
9	5990.85	4860.0 ([50,42,43,])	6959.0 ([48,43,46,])
19	5952.925	3742.0 ([42,37,39,])	8064.0 ([43,42,37,])
29	5827.35	3180.0 ([42,39,39,])	7569.0 ([43,41,41,])
39	6189.475	4199.0 ([45,38,39,])	7893.0 ([43,41,41,])
49	6023.125	3656.0 ([42,40,42,])	7336.0 ([43,41,40,])
59	6163.5	4027.0 ([42,42,40,])	8476.0 ([43,42,42,])
69	6316.675	4666.0 ([42,43,40,])	7785.0 ([43,42,41,])
79	6239.3	3847.0 ([42,44,39,])	8568.0 ([43,43,42,])
89	6258.0	3724.0 ([45,39,41,])	8087.0 ([43,44,41,])
99	6202.125	3946.0 ([42,43,43,])	7595.0 ([43,43,42,])

TABLE 3. EVERY TENTH OUTPUT OF THE GENETIC ALGORITHM. FIRST COLUMN IS THE GENERATION NUMBER, SECOND IS THE AVERAGE FITNESS, THIRD IS THE MINIMUM FITNESS OF THE POPULATION AND ITS CHROMOSOME, AND THE FOURTH IS THE MAXIMUM FITNESS OF THE POPULATION AND ITS CHROMOSOME. IT IS NOTICEABLE THAT THE ALGORITHM QUICKLY REACHES A PLATEAU.

The final value of the parameters of the algorithm and the start values can be seen in table 3. In the table you can see that the values stabilize quickly around a certain chromosome. This either means that this is the optimal value or that the algorithm does not optimize the population correctly. But judging from the increase in the average fitness, it is a good guess that the algorithm improves the parameters.

4.3. Performance measure of the algorithm

In order to test the performance of the AI algorithm, we run 10 simulated games and collect the data of the experiment. The results can be seen in table 4. As seen

Max Score	11590
Min Score	2670
Average Score	7514.0

TABLE 4. PERFORMANCE OF THE STATE MACHINE AI. THE TABLE INCLUDES THE MIN SCORE, MAX SCORE, AND THE AVERAGE SCORE.

in the table the algorithm performs way better than the previous algorithms. But even though the average score is significantly higher, the span between high and low is as well. Yet the algorithm is clearly superior to the other algorithms. If the score is to improve, the logic of the state machine could be improved to have more parameters and better chances between states. This might improve the overall score, and increase its survive ability.

5. Experiments

In this section I will compare the performance of the different AIs to the standard AI of the *Ms. PacMan Vs Ghosts* framework. I will compare the different scores for the AIs and give a small discussion regard its relative performance.

In table 5, you see the different algorithm scores. Given the data found under the performance test, I will not dive deeply in to the performance of the MCTS algorithm as much as the other two. The MCTS algorithm handles very poorly compared to the starter PacMan AI. The performance of the algorithm, as discussed earlier, could be optimized by punishing the reward function for deaths and movement in a more intelligent way.

Contestant	Min Score	Max Score	Average Score
StarterPacMan	2390	7320	4207.0
MCTS	300	780	543.0
BehaviourTree	2010	5560	2914.0
StateMachine	2670	11590	7514.0

TABLE 5. THIS TABLE COMPARES THE PERFORMANCE OF THE DIFFERENT AIS. THE TABLE INCLUDES THE MIN SCORE, MAX SCORE, AND THE AVERAGE SCORE FOR EACH OF THE ALGORITHMS.

The behaviour tree algorithm fares better, yet still not as good as the StarterPacMan AI. This might be explained by the fact that the behaviour tree does not chase and eat ghost, which is a source of a lot more points than eating pills. The tree could with few improvements possible out play the starter AI easily, just by changing the focus to chase ghost.

The state machine is better than the starter AI, especially seen in the differences between their average scores. The state machine is better at getting points by eating the ghost compared to the starter PacMan. The logic behind both of the AIs are visually similar, but with the state machine being better at hunting ghost.

6. Conclusion

In this section I will go over the conclusions gained from the experiments and work done with the different methods for getting the highest score in the *Ms. PacMan Vs Ghosts* framework.

The most successful AI was definitely the state machine optimised with a genetic algorithm. The optimization of the parameters in the state machine, yielded a good example of an algorithm capable of getting a high score. As discussed in earlier sections, the algorithm could be optimized by improving the state machine logic and adding more parameters which could be optimized by the genetic algorithm.

The second most successful AI was the behaviour tree. It managed to perform averagely, despite only trying to eat as many pills as possible. The AI could be optimized by both adding logic to chase ghost and optimized the logic for detecting and avoiding ghost and choke points. The primary reason for the failure of the AI were that it died at the second level at the choke points, because of the way it detect ghost. Due to time constraints the AI optimizations were not implemented.

The least successful AI was the one using MCTS. This AI could be optimized immensely by changing the reward function. Changing how to calculate the reward, by punishing the number of moves it takes, as well as changing the punishment for getting eaten by ghost. Again the optimizations were not implemented due to time constraints.

7. Discussion

In this section I will discuss the different algorithms and their strengths and weaknesses, and which generally performed best in this experiment.

Based on the results given from each of the algorithms, the one which was easiest to achieve the best result was the usage of the genetic algorithm. Its ability to optimize a problem simply by trying out values quickly in a clever manner, gave the best results and it was able to achieve this quickly. Perhaps applying the approach to another problem than a state machine, would perhaps yield even better results. You could perhaps with some work create a genetic algorithm which could combine some sort of planning strategy, to have a greater space to optimize for the AI.

The MCTS approach proved the to be the worst implementation of the algorithms. However, its ability to guess the outcome of the game, might prove invaluable to create a powerful AI. In general with optimization it might outperform the other algorithms.

The behaviour tree proved to be a powerful tool for creating an AI which could easily be visualized. The tree structure and the simple logic behind it, and the re-usability of the action nodes, creates an AI which is easy to understand and rework. Its greatest feature is the ease of which it could be rewritten, whilst still using the code already produced. The downside is that much of the logic has to be produced from a human design point of view, rather than through algorithm optimization. This could be migrated by using an optimization algorithm such as the genetic algorithm used above.

Final thoughts on the subject is that there is no clear winner in terms of which AI will solve any job best. But in given situations one could take pretence above others, simply by using its strengths.

Acknowledgments

I would like to thank Claus Bdker Wind and Mats Stenhaug for help with the programming and algorithm construction.