Modern AI for Games: Individual Assignment

Mathias Brandt, mfli@itu.dk, IT University of Copenhagen

I. Introduction

This paper serves as documentation for the individual assignment in the course Modern AI for Games at the IT University of Copenhagen.

The paper describes two separate Ms. PacMan controllers that have been built using different techniques. The first controller has been implemented using Monte Carlo Tree Search and tries to predict what will happen later in the game. The controller uses this information to select the best move in the current situation. The second controller is based on a Behavior Tree and only uses information from the current game state to decide on which move to make next. The performance of the Behavior Tree has been improved by using an Evolutionary Algorithm to optimize the parameters that the tree uses to make decisions.

It should be noted that my implementation of the Monte Carlo Tree Search algorithm is loosely based on the implementation made by Stewart MacKenzie-Leigh from the Computer and Information Sciences department of the University of Strathclyde [1].

The source code is available in a public repository on GitHub [2].

October 9, 2015

II. Monte Carlo Tree Search

The Monte Carlo Tree Search (MCTS) algorithm can be used as a way of selecting the best next move from the set of possible moves given the current game state. The algorithm tries to predict the future using simulation to decide on the most optimal next move.

The algorithm has been implemented in such a way that it uses as much time as possible to build the search tree. Each node in the search tree represents a single move that can be taken at a specific junction in the game. Simultaneously with building the search tree, the algorithm simulates the game, copying the game state at each junction.

A. Algorithm Description

When the game first starts, the algorithm selects one of the possible moves from Ms. PacMan's initial position. The first move is also used to create the root of tree. From this point on, each time the game requests the next move the algorithm will simultaneously traverse and expand the search tree. Starting from the root node, the most promising direct child is selected until a leaf is reached. To decide where to go next, the leaf node is expanded to reveal the next possible moves. These moves are added as children to the leaf node. One of the new children is selected and the game is simulated until it ends by applying the move that the child represents to the current position. The game ends when either Ms. PacMan dies, the level changes, or a predefined simulation count is reached. When the simulation finishes, the end game score is retrieved along with a path bonus score. The sum of the two scores is applied to the current child and is backpropagated up through the search tree to the root. The process is repeated until most of the computation time has been used, and results in different score weights of the children of the root.

The remaining computation time is used to select the best child of the root. The child represents the most promising move that can be taken from the current position. The move is returned to the framework.

B. Selecting the Best Child

To traverse the tree, the algorithm must select the best child at each node. The best child is defined by having the maximum Upper Confidence Bound (UCB) value of all the available children. The UCB value is calculated for each child by using the following formula:

$$A + C * \sqrt{\frac{2 * \log n}{T(n)}}$$

Where *A* is the average score of the child, i.e., the child's total score divided by the number of times the child has been visited, *C* is a predefined constant, *n* is the number of times the child's parent has been visited, and *T(n)* is the number of times the child has been visited.

The fraction in the equation makes sure that the algorithm explores new child nodes from time to time. Had the UCB value simply been represented by *A*, the same child would always be selected, since the other, unexplored, children would have an average score of 0. The value of *C* determines how much weight should be put on the fraction, i.e., how often the algorithm should explore the other children that may not seem to be best. The higher C value, the more exploration is performed. A *C* value of 3 has been manually selected by trial and error.

C. Calculating the Path Bonus Score

When simulating the game, most of the time the simulation count will be reached before the level is completed or Ms. PacMan dies. As such, the game is ended prematurely and the game score provided by the framework alone might not be fully representative of the quality of the chosen move that initiated the simulation.

Consequently, a path bonus is added to the game score to better reflect the actual quality of the chosen move. The path bonus is calculated based on these events: *a*) was a pill eaten,

20 points; *b*) was a power pill eaten, *100 points*; *c*) was path with no pills traveled, *-100 points*; *d*) was a ghost eaten, *500 points*; *e*) is a ghost that cannot be eaten too close, *-300 points*; *f*) is a ghost that can be eaten close enough, *300 points*; *g*) did Ms. PacMan die, *-1000 points*; and *h*) was the level completed, *5000 points*.

These values have all been selected manually through experimentation.

D. Performance and Evaluation

This algorithm was entered into the first Ms. PacMan competition that was held in class. The solution performed very poorly, achieving an average score of only 1818 points. Just 5 other entries performed worse. Measuring the performance by average game length makes the solution look a bit better, with a score of 1069. 10 other entries performed worse on this metric. The solution's average score per time stamp is as low as 1.2 which is the 6th worst performance out of all the entries. Interestingly, though, one of the best solutions among the entries (*BEYA, with an average score of 12862*) has an average score per time stamp of only 0.2 – which is significantly lower. However, combining these different metrics reveal a quite poor performance of the algorithm.

Prior to the competition, the bonus scores were manually tweaked to provide the seemingly best performance of the algorithm. It should be noted that the performance was manually measured against the StarterGhosts controller, which was not used as opponent in the competition. Also, the same ghost controller is used internally in the algorithm when simulating the game. This could potentially have a big impact on the performance, and might explain the poor performance in the competition.

Assuming that the algorithm is implemented correctly, the only factors that influence the current implementation are the size of the bonuses awarded to different events. Manual adjustments of these numbers is definitely not preferred since the combination of bonus values is almost endless and the process is very slow. Even if some reasonable bounds were to be put on the bonus values, there are still too many combinations to try out manually. Instead, using an evolutionary algorithm to adjust the bonus values could have been used. This could potentially have resulted in much more optimal bonus values. Evolutionary algorithms is explored in the implementation of the Behavior Tree.

III. Behavior Tree

Unlike the MCTS algorithm, the behavior tree does not try to look into the future to determine which move will be the best one. Instead, the algorithm uses knowledge about the current state of the game – such as ghost positions, if ghosts can be eaten, and distance to pills and power pills – to determine which move should be the next one.

A. Algorithm Description

Each time a move is requested by the framework, the algorithm traverses a static decision tree to determine the most appropriate behavior given the current game state. The decisions tree can consist of sequence and selector nodes, inverters, and leaves, among other node types. A leaf is a node that performs a specific action, such as determining which ghosts that are currently threatening the player. A leaf returns either *true* or *false* and cannot have any children. Sequences and selectors have a finite number of children (the node types can be "nested" in any combination, so a sequence can have another sequence as a child, etc.). They each evaluate their children in turn, however, there is one key difference: the sequence will return *false* immediately when one of its children evaluates to *false*. If all children evaluate to *true*, the sequence will also return *true*. The selector is opposite in the sense that if a child returns *false*, the selector will try to evaluate the next child. If a child returns *true*, the selector stops evaluating its children and returns *true* itself.

B. Implementation Walkthrough

The full behavior tree is shown in figure 1. The root of the tree is a selector that determines Ms. PacMan's overall strategy for the current move. The order of the root's children is crucial: the most important thing will always be to stay alive. If Ms. PacMan dies, no points are gained. Next, the algorithm should choose the strategy that will possibly provide her with the most points – eat ghosts, if this is possible. Lastly, Ms. PacMan should go after the nearest pill, where power pills are more attractive than regular pills.

To stay alive, the algorithm must first determine if any threats are near. If this is the case, that algorithms checks if the ghosts are actually threatening – i.e., if they are not edible. This is achieved by inverting the result of the *Are ghosts edible?* leaf. If the nearest ghosts are considered a threat, the algorithm find the move that will help Ms. PacMan escape the ghosts. This behavior is implemented in the *flee sequence*.

If the flee sequence fails, the algorithm will check if there are any edible ghosts nearby by evaluating the *attack sequence*. The sequence is straightforward: find any possible victims, check if they are edible, and if so, attack the nearest victim.

Lastly, if Ms. PacMan is neither threatened nor able to consume nearby ghosts, the algorithm will send her in the direction of the nearest power pill if one such is within range. If not, the nearest regular pill will be consumed instead. The pills sequence has no conditions to run (like all the other sequence), and as such it serves as a fallback. If no other options are viable, simply move towards the next regular pill.

C. Improving the Behavior Tree Using Evolution

Three parameter values lie at the core of the Behavior Tree algorithm: *a*) minimum distance to a ghost before it is considered a threat *(10)*; *b*) minimum distance to a ghost before it should be chased when attacking *(75)*; and *c*) minimum distance to a power pill before Ms. PacMan should move towards it *(30)*. These parameter values are used to evaluate the conditionals in the flee sequence, attack sequence, and power pills sequence, respectively. Initially the parameters were set manually to the values in the parentheses.

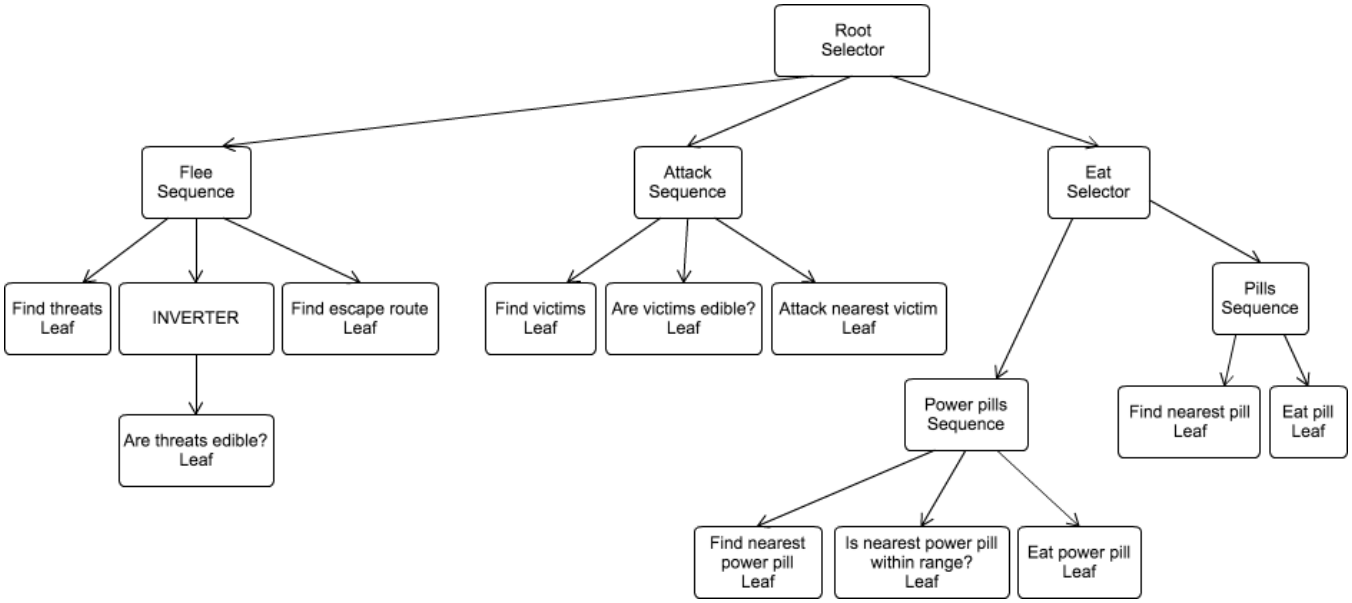

Fig. 1. The full behavior tree as it is implemented in the algorithm

These values seemed to provide a pretty decent result, with an average score of 3704 points across five games. The scores ranged from 2730 to 4320.

An evolutionary algorithm has been implemented in an effort to tune the parameter values. The algorithm starts by creating a base population of size 40. Each gene is created with three random genotype values ranging from 0 to 100, mapping to the parameter values described above. The population is then evaluated 500 times. Each evaluation simulates the game with each gene in the population four times, and uses the average end game score as the new fitness value for the gene. After each evaluation, the best gene from the population is extracted and compared to the all-time best gene – the gene with the highest fitness value of the two is kept as the all-time best gene. Afterwards, the best half of the population is selected for breeding. The genes are bred pair-wise, and each breeding results in two offspring that are added to the population. When the breeding process is completed, the population size is back to its original level: half of the population consists of the best half of the previous population, while the other half are new offspring. The offspring are generated using single-point crossover.

The result of running the evolutionary algorithm is the best gene across all the evaluated populations. Running the game with the parameter values from the best gene's genotype shows that the evolution process has found a set of values that provide a better game score. As such, across five games an average score of 4752 was attained with scores ranging from 3390 to as high as 9700.

IV. Discussion

From the game scores provided in this paper, it is clear to see that the implementation of the Behavior Tree algorithm outperforms the Monte Carlo Search Tree. The average score was increased by a factor of 2. Improving the Behavior Tree with values from the evolutionary algorithm further improves it performance an increases the average score by a factor of 2.5 compared to the MCTS implementation. The highest score of the improved behavior tree (9700) might have been an outlier, but proves that the behavior tree has potential to be improved even further.

A. Future Improvements to the Behavior Tree

The behavior tree that has been implemented is quite simple in its nature, and what follows are some ideas for future expansion and improvements.

1) Calculating a Better Escape Route When Threatened: As mentioned earlier, staying alive is – not surprisingly – necessary to gain points, and thus possibly one of the most important aspects of the game. To find an escape route, the current implementation first defines three sets: *a*) the set of possible moves from the current position of Ms. PacMan (A); *b*) the set of moves that are being blocked by threatening ghosts (B); and *c*) the set of valid escape moves (C), defined by

$$C = A \setminus B$$

The set *B* consists of the next moves *towards* each of the threatening ghosts. But in reality, the next move towards the ghost might not lead directly to it. In the situation shown in figure 2, the algorithm will result in Ms. PacMan (yellow) being blocked from going right, assuming that the ghost (red) is considered to be a threat. Ms. PacMan will instead escape the ghost by going down. However, if another ghost was present below Ms. PacMan, no valid escape moves would exists. It is clear to see that going right would be the best move in that specific situation.

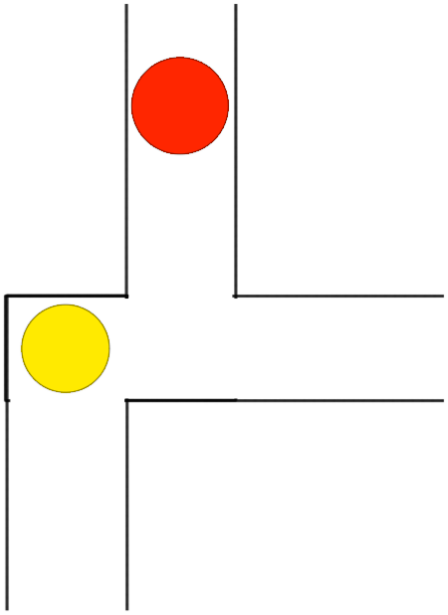

Fig. 2. A situation where Ms. PacMan (yellow) might choose a sub-optimal escape route, assuming that the ghost (red) is heading downwards and is within range to be determined a threat

2) Moving Towards Edible Ghosts or Pills are Not Considered While Fleeing: While staying alive might be the most important strategy, points can still be collected while fleeing. As long as the algorithm makes sure to intelligently escape the ghosts and keep them behind Ms. PacMan, pills should still be collected. The current implementation does not take this into account when fleeing. Furthermore, when the set of valid escape moves has been computed, the first entry is always selected. This often leads to undesirable outcomes, as Ms. PacMan may enter into a loop around a short wall as long as she is being chased. If the ghosts are merely somewhat intelligent, performing a pincher move is easy and will lead to the death of Ms. PacMan.

Two different approaches could mitigate this issue: *a*) randomly selecting from the set of valid escape moves to avoid loops; and *b*) the next move when fleeing should be the one towards the nearest pill. Combining these two approaches would lead to improved performance of the algorithm.

3) Power Pills are Being Eaten Even When a Power Pill is Already Active: If a power pill is within range, the algorithm will always send Ms. PacMan in its direction, assuming that no edible ghosts are also within range. This can lead to wasting the effect of the power pill. This could be mitigated by tweaking the parameter values of the behavior tree – e.g., by increasing the minimum distance before a victim is attack, and possibly decreasing the minimum distance before a power pill is eaten. Another solution would be to simply not actively go towards power pills when a power pill is active, but instead towards the nearest regular pill. This could, however, still lead to accidental consumptions of power pills.

REFERENCES

[1] Stewart MacKenzie-Leigh, *https://github.com/stewartml/montecarlo-pacman*

[2] maig-f2015 GitHub repository, *https://github.com/MathiasBrandt/maig-f2015*