Modern AI for Games
Developping Pac-Man controllers

Roman Sahel, *Exchange Student at ITU*

Abstract—In this paper, we present and explain the development and implementation of controllers for the Pac-Man agent, using three different AI techniques.

Index Terms—Artificial Intelligence (AI), Pac-Man, Behavior Tree, Neural Network, Genetic Algorithm, Monte-Carlo Tree Search (MCTS)

1 Introduction

For the last few years, implementing Artificial Intelligence methods in game has been a problem: how to create a smart and realistic but stil playful AI?
Also, because games try to reproduce the real world in a simple and controllable way, they are a good way to test, experiment and develop AI.
This paper will thus describe our attempt to understand and implement three different AI algorithms applied to the PacMan game to control the player's agent.

2 Behavior Tree

2.1 Description

In order to be free to create an advanced AI using a Behavior Tree, the first step was to implement the actual structure of the tree and its different node types. The implemented structure contains only what is needed for the final Behavior Tree so that it remains very simple, as seen below:

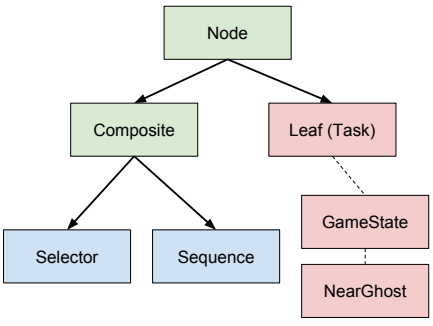


Fig. 1. Class hierarchy for the Behavior Tree implementation

The Behavior Tree is thus composed of *Node* objects which can either be a *Task* or a *Composite*. The only requirement is for the nodes to implement a *DoAction* method which returns a *boolean*. A *Task* is a node that has a direct action: it can either test a condition or modify variables (including the move to return, for example). A *Composite* is a node that doesn't do anything by itself but that executes its children under some conditions and in a certain order. Here, the only composite node that were implemented were the *Selector* and the *Sequence*.

A GameState class is also used analyze the state of the game at each step and keep the important information such as the nearest pill, the nearest power pill, and a list of the ghosts, sorted by distance. All these characteristics are used by the tasks of the Behavior Tree.

The final Behavior Tree (see representation on the last page) is a succession of condition checking and often simple reaction. The only special case is the task that checks if the path to the nearest power pill is safe.

2.2 Algorithm Parameters

The actual behavior tree revolves around the distance between the nearest ghost and the pacman, and more precisely if the Pacman is currently in a danger zone which corresponds to the distance between the player and the nearest ghost being less than a threshold. This threshold was determined through experiment: by using the runExperiment method for 2000 trials with different values (from 0 to 100), the optimal value was found.

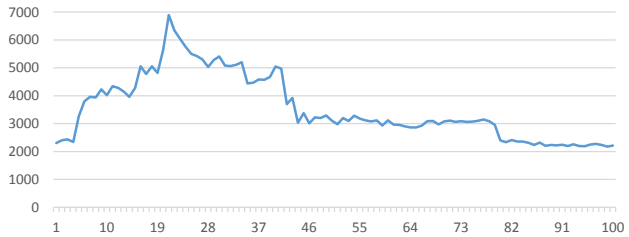
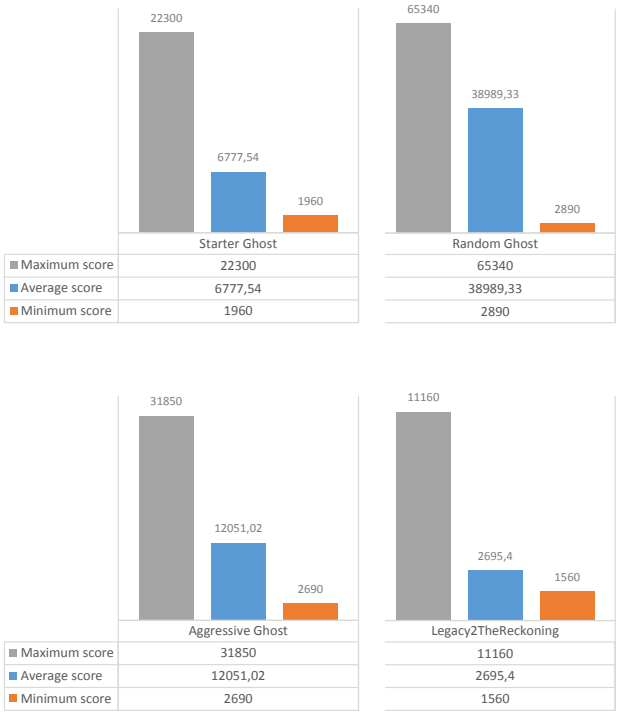


Fig. 2. Average score after 2000 trials, depending on the danger threshold: the high peak is for the 20th value.

2.3 Results



	Starter Ghost	Random Ghost	Aggressive Ghost	Legacy2TheReckoning
Maximum score	22300	65340	31850	11160
Average score	6777,54	38989,33	12051,02	2695,4
Minimum score	1960	2890	2690	1560

Fig. 3. Results for different Ghost Controllers, after 2000 trials

2.4 Performance

In the final Behavior Tree, most of the tasks are testing conditions. The only task that requires lookahead is the case when the player is in danger and we need to check if the path to the nearest power pill is safe. This still takes very little time as the distance between Pacman and the pill is often very small. Thus, the performance of the behavior tree is really good and its execution is very fast.

2.5 Conclusion

The score obtained with the Behavior Tree are rather satisfying. The power of this algorithm resides in its scalability and its adaptability so that we might be able to obtain better scores with a more sophisticated tree. For example, a custom A* Algorithm could be used to penalize path that are more dangerous (paths with only one exit, with ghosts too close, etc.).

3 Genetically Evolved Neural Network

3.1 Description

The first step of implementing an evolving PacMan controller was to choose what kind of controller would be evolved. Here, we chose to evolve a Neural Network rather than a Finite State Machine or a Behavior Tree, because it seemed at once more interesting and more challenging.

The idea is thus to combine a neural network, which can – given certain characteristics of the current game state – assess the quality of the situation the player is in, with a genetic algorithm that will evolve the said neural network in order to make it better.

3.1.1 *The Artificial Neural Network*

First, for the neural network, we used a multi-layer perceptron: simple yet powerful enough for this application. The implementation in itself of the perceptron is pretty straightforward, especially since we don't need any back-propagation: we propagate the given inputs to each neurons using a weighted sum then normalized using the activation function (here, the sigmoid function), until it reaches the output neurons.
This perceptron is used by the PacMan controller to decide what move to make: at each game step, we consider every possible moves (four at most). We then use these locations to extract the relevant parameters (i.e. the inputs, that we will detail later) and pass them to the neural network. The MLP will then combine those values and compute a score representing the quality of the situation. The move to make is the one that got the highest score.

- Distance to the x nearest ghosts ($[0; +\infty]$)
- State of the x nearest ghosts (-1 if normal, 1 if edible)
- Distance to the nearest pill ($[0; +\infty]$)
- Distance to the nearest power pill ($[0; +\infty]$)
- The number of lives remaining ($[0; 3]$)

3.1.2 *The Evolutionary Algorithm*

The genetic algorithm is thus used to evolve those perceptrons from a random population to a population as fit as possible. The algorithm then goes as follow:

1) We generate a population of n Perceptron based PacMan controllers. The perceptrons have random weights.
2) Each individual runs the game multiple times using their unique neural network so that we can get a representative average score for each of them.
3) We only keep the k best individuals as parents to replace the dead ones.
4) We combine two-by-two those parents to create offspring by taking each gene (here, each particular weight) randomly from one of the parent or the other. There is also a chance to mutate the gene, in order to insure diversity and to avoid getting stuck on a local maximum.
5) We repeat from step 2 until the population becomes stable.

3.2 Algorithm Parameters

We already discussed the inputs we chose but what about the rest of the network's structure? Indeed, a multi-layer perceptron has hidden layers of neurons so that we had to decide how many layers and how many neurons per layer were needed.
In order to find the best values, we had to experiment. The final neural network thus has one hidden layer with eight neurons. We also concluded that using only one 'nearest ghost' as input gave better results than using two or more.

The evolutionary algorithm also has parameters that can influence the results:

- **Mutation chances:** This is the chance of mutating genes for new offsprings. We tried multiple values and we found out that with a value too high the score would constantly fluctuate but that without any mutation, it often (but not always though) got stucked on a local maximum.
  10 percent thus gave good results.
- **Size of the population:** This is the size of the population per generation. We need a population big enough to have enough diversity to evolve but not too big otherwise we would have too many parents (not necessarily good ones then) and a too long computing time.
  A population of 100 individuals gave good results while keeping performance at a reasonable state.
- **Number of trials per generation:** obviously, the more trials we do per generation, the more accurate the scores are. However, it also greatly increases the computing time so that we have to find the right balance.
  15 trials per generation was a good balance between accuracy and performance.

3.3 Results

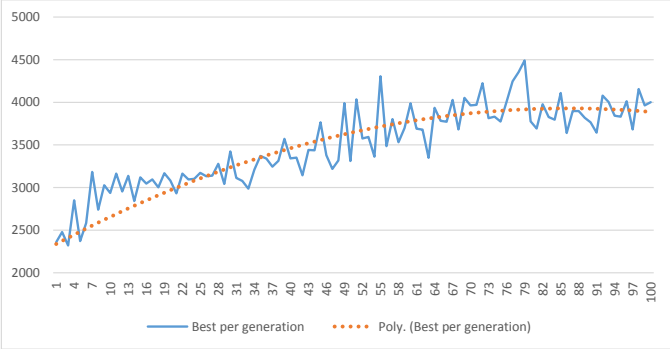


Fig. 4. Evolution of results depending on the training: we can see the scores improving before reaching a 'stable' state

3.4 Conclusion

The results were rather disappointing: while we can see the influence of training since the average score does get slightly better over time, we don't actually reach a high result. One possible way to improve the results would be to not only evolve the weights of the neural network but also to evolve its inner structure (i.e. the number of hidden layers and hidden neurons), thus applying a NEAT algorithm. Another possibility would be to change the inputs we take: maybe the chosen characteristics are not sufficient or not relevant for the neural network to give a correct output. With those results, we can wonder if this technique is adapted to such a game? Is it just a matter of changing the inputs or should we actually use a different neural network algorithm?

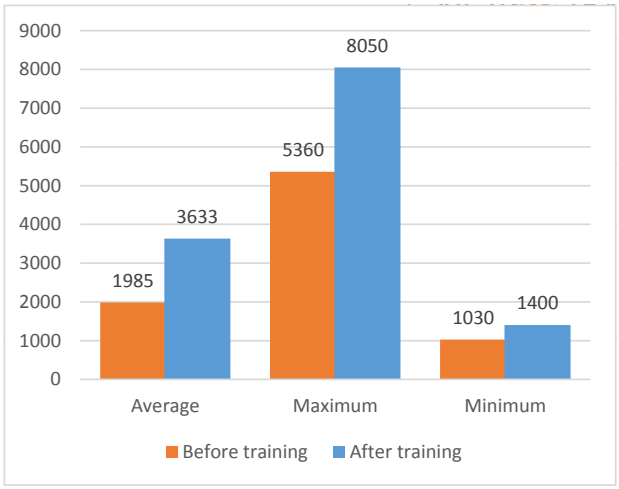


Fig. 5. Results of 2000 trials before training and after training.

4 Monte-Carlo Tree Search

Once the basic Monte-Carlo Tree Search algorithm is understood, the implementation is rather straightforward: we build a tree with the nodes that are selected by the *tree policy*. *Tree policy*'s goal is to balance exploration and exploitation: to achieve that, we use the *Upper Confidence Bound 1 applied to Trees* (UCT) formula.
We then run a simulation from this node. The rules of this simulation – i.e. the moves made by the player and the ennemies – are defined by the *default policy*. At the end of the simulation – which can either be caused by a game over or by a custom rule, like a lost life or a new level reached – we obtain a score. This score is then backpropagated to all the nodes on the path from the selected node to the root of the tree. This is the score that will be used by the UCB1 algorithm through the *tree policy*.

The main trouble is to find what the nodes are and how to represent the game states. For our implementation, we chose to keep the whole *Game* object. This choice has the advantage of making the simulation really easy, since we can just use the *advanceGame()* method; on the other hand, it demands more computing power, which can hugely affect the algorithm's results.

4.1 Algorithm Parameters

The only variable is the exploration parameter in the UCT formula. We tried a few values ranging from 1 to 300 without seeing much difference. We will give a possible explanation in the section below.

4.2 Results

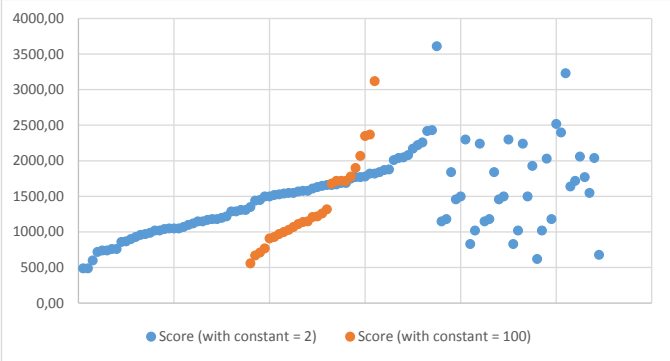


Fig. 6. Results obtained with two different exploration parameter values. Maximum Score: 3610; Average Score: 1487; Minimum Score: 490

4.3 Performance

The MCTS algorithm is *really* compute-intensive due to the fact it tries to simulate multiple games. Besides, the game representation we chose does not simplify, in any way, the simulation... The consequence of this is that the algorithm does not have time to explore many nodes and go as deep as it would need in order to find a good solution. That is why the exploration parameter has no influence on the results and that's why the scores we get with this controller are not really satisfying.

4.4 Conclusion

As we said and as we can see in the section above, the average score obtained with this controller are low. In order to improve it, we would need to speed up the algorithm. The way to do that would be to change the game state's representation in order to do the simulations without using the *advanceGame()* method which seems too slow to be used here. With these changes, the MCTS should give much better results and we know we can expect extremely high scores with this algorithm.

5 Discussion

Between the Behavior Tree, the Evolved Neural Network and the Monte-Carlo Tree Search, the first one was the one which got the best results. The Behavior Tree is a really interesting and powerful way to develop an realistically intelligent AI. It is also challenging because it is the developer's job to think and find the different behavior and reactions the controller should have. In this way, it is very different from the Neural Network and the Monte-Carlo Tree Search which are AI which plays very intelligently, as a computer. For that reason, it seems that the MCTS algorithm is particularly powerful and will probably always get a better score – when implemented right. Another strength of the Behavior Tree is that it can be very fast and does not need any training, which makes it particularly adapted – if not perfect – for games.

In the end, the choice of the AI technique we use depends on the game and the result we want to reach: While MCTS is the go-to algorithm if we want a perfect agent for a board-like game, Behavior Tree seems like a solution that is adapted to all sorts of games.

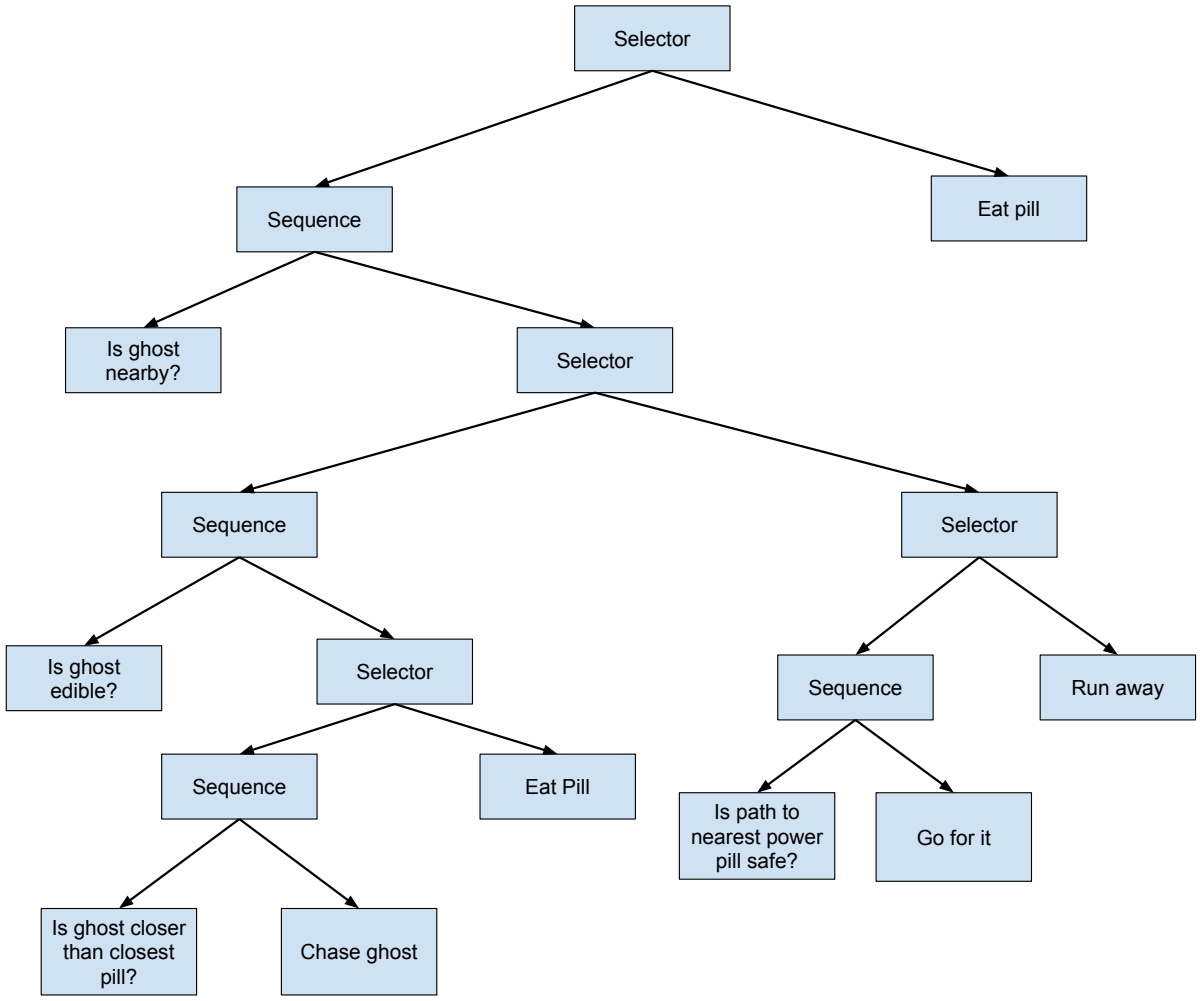


Fig. 7. The final Behavior Tree