Modern Artificial Intelligence
Individual Assignment

Adrian Radziszewski
IT University of Copenhagen
Email: adra@itu.dk

I. Introduction

This report analyses and evaluates various techniques for the individual assignment in the Modern Artificial Intelligence course. The work carried out for the assignment consists of implementations of three selected techniques introduced during the course. The techniques were implemented in Python using the Pac-Man framework used in the course lab sessions. More specifically, the framework was used to implement various agents controlling the Pac-Man in order to solve specific levels. The code for this assignment was written by me, however I did cooperate, with a fellow student, Julia Beryl van Straaten (juva@itu.dk).

The following techniques were selected for this assignment:

- A* Path finding
- Behaviour Trees
- Deep Q-Learning

The report is divided into a section for each technique. Each technique section begin with a description of the approach for implementing the related technique and the relevant algorithm parameters. The section continues with adjustments and experiments performed to obtain the optimal agent behaviour, and lastly a conclusion summarizing the results.

The report then continues with a discussion in which the approaches are compared and possible improvements considered. Lastly, the final conclusion on the overall results and the preceding discussion is presented.

II. A* Path finding

This section covers the analysis of the path finding approach using A* and other path finding algorithms that were developed, namely Depth First Search and Uniform Cost Search, otherwise known as Dijkstra's algorithm.

A. Approach

The implementations of the path finding algorithms required levels in the Pac-Man framework to be represented as graphs in order to traverse them. Thus, each implementation requires a SearchProblem object that encapsulates the level representation along with the start and goal states. In the case of A*, the algorithm requires an additional parameter in form of a heuristic function used to traverse the graph towards the goal. The output of each algorithm is a list of actions (North, South, East, West) that lead the Pac-Man to a goal state. The framework already provided a SearchAgent implementation, which allowed to change the relevant parameters, and was used to execute and evaluate the various algorithms.

B. Experiments

The aim of the experiments was to compare the various path finding algorithms and adjust the A* parameters to get the optimal result in the Pac-Man environment. To begin with, the results of Depth First Search, Uniform Cost Search, and A* with a naive setup are presented. Then the parameters of A* are discussed and their results presented. The evaluation is performed purely based on the path taken, as the score optimisation is not the goal of path finding.

In order to provide a simple visualisation of the results of each approach, the level 'boxSearch' was chosen. Each algorithm was executed with the pacman starting at a position close to the middle, and had the goal of reaching the lower left corner. As Depth First Search and Uniform Cost Search do not use any adjustable parameters, they were executed based on the start position and the goal. A* on the other hand requires a heuristic function used to evaluate the distance between a node and the target. For the initial test a null heuristic, which always returns 0, is used. Later, two heuristics were tested: euclidean distance and Manhattan distance. The initial results are presented on figure 1. The graph traversal performed on the level by each algorithm was visualised by a diminishing red colour on the visited level tiles, starting from the initial Pac-Man position. The final path taken by the Pac-Man is visualised by a green line going from the start position to the goal position.

Depth First Search: Considering the first result, produced bu Depth First Search, it is easy to see that the algorithm naively traversed the level from left to right, going towards the bottom until reaching the goal state. Since no optimizations are performed by the algorithm, the path is constructed from the first graph traversal that reaches the goal. With the overall goal of finding the shortest path, this algorithm does not perform well.

Uniform Cost Search: The result of Uniform Cost Search shows that the algorithm performs the search by expanding in all directions around the starting point, until the goal state is reached. A shortest path to any visited node is assured as the algorithm prioritizes the graph traversal by visiting the nearest unexplored nodes every time, and storing the path to them. The result also shows a major improvement over the path found by Depth First Search.

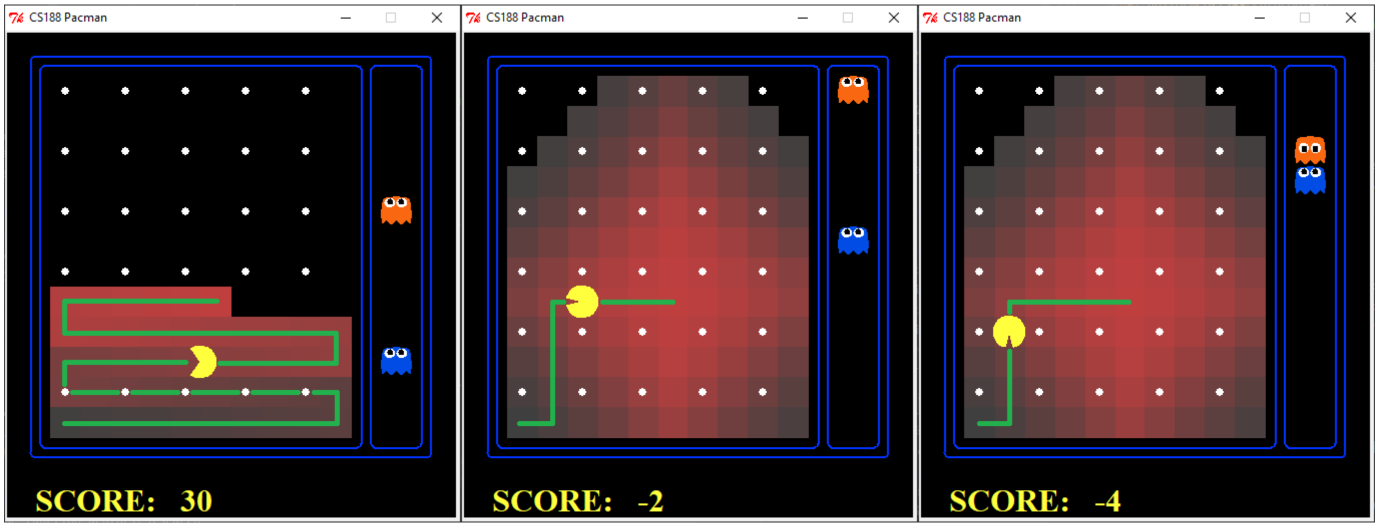

Fig. 1. Results of Depth First Search (left), Uniform Cost Search (middle), and A* with a naive setup (right).

A - Null Heuristic:* The result of A* shows that without a heuristic, the algorithm performs as good as Uniform Cost Search. The shortest path is still found, but the the traversal of the graph could be optimized. Therefore, it is interesting to examine how heuristics can affect the algorithm.

A - Euclidean Distance Heuristic:* As seen on the left result on figure 2, the euclidean distance heuristic improved the performance of the algorithm by prioritizing nodes closer to the target and thus avoiding to visit nodes further away. The small difference with this heuristics what the path taken, however it was still a shortest path.

A - Manhattan Distance Heuristic:* The result of the Manhattan distance heuristic is even more efficient, where fewer irrelevant nodes were visited. In the case of Pac-Man, the Manhattan distance heuristic also seems to be more appropriate compared to the euclidean distance due to the grid layout of each level and the set of allowed actions (where the Pac-Man cannot move diagonally).

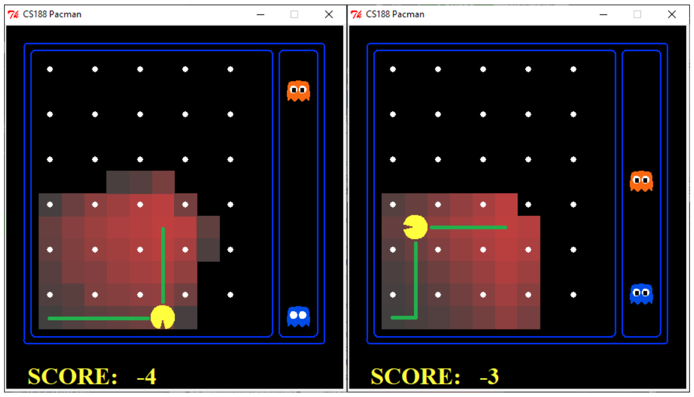

Fig. 2. Results of A* with the Euclidean Distance heuristic (Left) and Manhattan Distance (Right).

C. Conclusions

The tests of various algorithms have shown that A* with an appropriate heuristic has the best performance. However, in order to find a path using A* a target position must be provided. In the case of Pac-Man, an obvious goal would be a pellet. However, the classical level contains more than one of them, and determining the closest pellet may not directly correspond to the closest one in terms of the path. Therefore it is still interesting to use the Uniform Cost search, which terminates as soon as the first pellet is encountered, and thus guaranteeing to find a pellet that the Pac-Man can get fastest to.

III. Behaviour Trees

This section explains and evaluates the Behaviour Tree approach to control the Pac-Man. First the exact approach and implementation is explained, followed by a description of experiments. The goal of the behaviour tree approach for the Pac-Man game was to complete levels where a more sophisticated logic was required. That is levels with the ghost adversaries that complicate the solving of the level and can lead to losing the game. The behaviour tree agent implementation was used in the Pac-Man competition.

A. Approach

As the Pac-Man framework did not define a behaviour tree structure, a custom implementation was required. The implementation was based on the description of behaviour trees in [1], with some small differences. The behaviour tree structure from the book was kept intact, but the logic of execution was slightly modified. In the custom implementation, nodes do not have a state nor return a value (run/success/failure) but a tuple with the status whether the execution succeeded or failed (True/False) and an action for the Pac-man (in the case of a success). That is, when the tree is executed, it immediately traverses its nodes and returns the success and action tuple.

The following data structures were implemented in order to build behaviour trees:

Node
: A base node which can contain other nodes as children, and an abstract 'execute' method. Used by the other, advanced nodes.

Query Leaf
: A leaf node that executes a query in form of a function returning a True or False value.

Action Leaf
: A leaf node that executes an action function, always succeeds and returns an action.

Sequence
: A node that executes all its children in sequence. Fails when any of the child node fails.

Selector
: A node that executes all its children and succeeds at the first succeeding child.

With the basic behavior tree nodes in place it was possible to define a specific tree instance for the Pac-Man agent. The main goal was to complete the classic level of Pac-Man with two ghost adversaries. Before constructing the tree, a few issues needed to be considered. First of all, in order for the agent to win, it needed to gather all of the pellets in the level. That includes the logic for finding a way to reach the pellet, thus a path finding algorithm was required. Moreover, the agent should also somehow be aware of the adversaries and avoid them to not lose the game. Therefore some kind of indicator whether an action was safe was needed, along with an appropriate logic to handle the behaviour in dangerous situations.

B. *Experiments*

The tree constructed had the overall goal to reach the closest pellet and run away if a ghost is encountered on the path. Thus, the tree consisted of a selector root node and 3 nodes used to find a path, move, and run away. The path finding algorithm used for this agent approach was Uniform Cost Search as it does not require manually finding a pellet, and guarantees to find the closest one, as discussed in the pathfinding section. The behaviour tree structure is presented on figure 3. The tree structure was very simple and the implementation required more emphasis on the action and query function implementations. A more detailed description of each node and the corresponding functions is provided below.

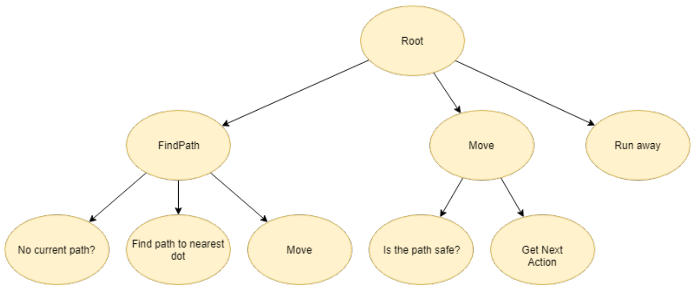

Fig. 3. Structure of the behaviour tree.

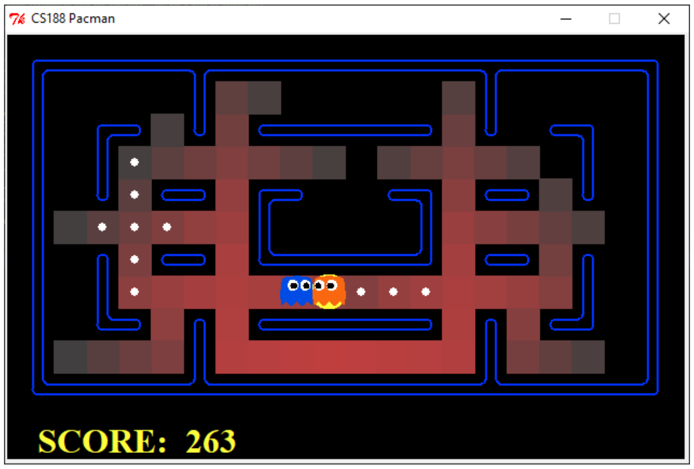

Fig. 4. One of the problems where the agent gets stuck between the ghosts and lose.

Root
: The root of the tree. Is a selector.

FindPath
: A sequence that finds a path and returns the first action for that path.

Move
: A sequence that first checks whether the next action is safe, and if that is the case, returns it.

Run away
: Action node that returns a Pac-Man action that goes in the opposite direction of the ghosts.

With the aforementioned behaviour tree instance, tests were run in order to determine its efficiency. The performance indicator for the tests was the win/lose ratio, and the average score. A total of 10 games were run, with the win ratio being 5/10 and the average score 929,3. The exact results of each run are given below.

Run #	Score	Win/Loss
1	1545	Win
2	506	Loss
3	492	Loss
4	1543	Win
5	1324	Win
6	1326	Win
7	1338	Win
8	417	Loss
9	488	Loss
10	314	Loss

Besides the results, some issues have been spotted with the Pac-Man behaviour during the runs. More specifically, there is a bug with the ghost danger detection, and sometimes the agent ignored a ghosts and moved into it, thus losing the game. Moreover, a general flaw in the tree design was spotted.

During some of the runs, the agent would walk into a corridor which was then blocked by ghosts from both sides and thus trap itself, without any chance of getting out. This issue is related to the greedy approach of getting to the nearest pellets without checking the surroundings. An improvement to the tree design would be to extend it with logic for estimating whether some parts of the level are safe to go to. The most important part of it would be evaluating whether the ghosts can be a danger after taking some action or amount of actions. Further improvements would also include making the agent take advantage of the big pellets which can improve the score when a scared ghost is eaten. The current implementation only makes the agent ignore the ghosts if they are scared, when checking if the path is safe.

C. Conclusions

Overall, the behaviour tree approach looks promising and, even with the current implementation having some flaws, can get a fine score and win some of the game runs. Further work on it could yield even better results, however that would also include other AI techniques for the specific cases in the game to evaluate and solve them.

IV. Deep Q-Learning

This section covers the Deep Q-Learning approach used to solve a small Pac-Man level without any adversaries. First, the approach and the implementation is explained,

followed by a description of tests and results

The approach is based on the article "Reinforcement Learning Tutorial Part 3: Basic Deep Q-learning"[1].

A. Approach

In standard Q-Learning, a Q-table is used to store and update the Q values for actions in a specific state. The problem of Q-tables is scalability, their size depends on action and state spaces, which can be very large in more advanced game. In order to avoid this problem, the Q-table can be replaced with an artificial neural network that learns the values of actions for each state. Replacing the Q-table with a neural network can save space, but at the cost of accuracy. Therefore, it was interesting to see how the introduction of neural networks to Q-Learning would affect the learning process, and what parameters could improve it. The Pac-Man framework provided a partially implemented setup of an Deep Q-Learning agent. The most important parts to implement was the initialisation of the neural network, the backward pass, and network updating. The network updating was the core part of Deep Q-Learning, where the network predicts the Q-values of an old and a new state, which are then used to train the model. The implementation had a few parameters that needed adjustments in order to improve the results. Specifically, the espilon value, discount factor, and parameters related to the network: number of layers, units in each layer, activation functions, and the learning rate. Additionally, the approach with a target network was also used for comparison.

[1]https://towardsdatascience.com/reinforcement-learning-tutorial-part-3-basic-deep-q-learning-186164c3bf4

B. Experiment

A few tests were performed on different parameter setups. Each test was performed in the 'tinyMaze' level. The first test was performed on a setup with 3x256 hidden layers, ReLU activation functions, and a learning rate = 1e-6, and both setups with a single and two neural networks. Both setups ran 125 times. The games' results can be seen on figure 5. Later,

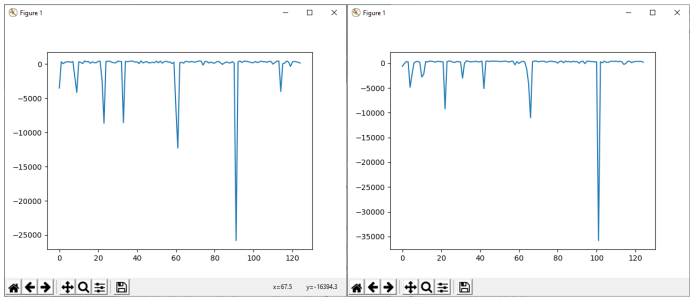

Fig. 5. Results of the DQNAgent for 125 games, with a single network (left), and two networks (right).

various setups were performed, however many of those ended in huge negative scored, and did not seem to learn anything. The most interesting behaviour happened by reducing the learning rate, as it resulted the score increasing after an amount of games. This happened for instance in a setup with a single network, 3x64 hidden layers, ReLU activation functions, and a learning rate = 1e-9. The result can be seen on figure 6.

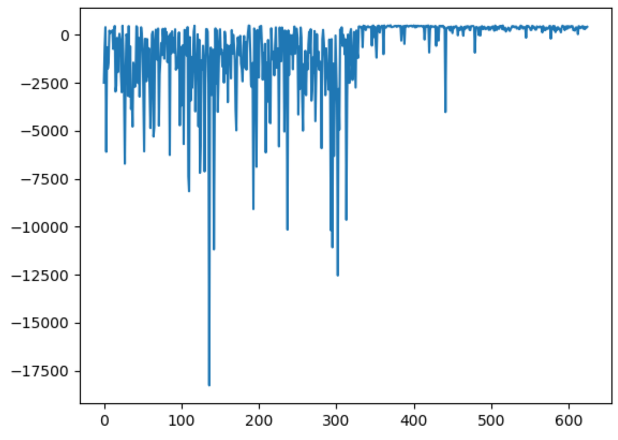

Fig. 6. Results of the DQNAgent for 600 games with a single network.

In general, the adjusting of the parameters turned out to be very tricky and required a lot of guesswork. Many of the results were arbitrary and it did not seem that anything was learned by the network. One of the considerations after watching the results was that there could be a bug in the added implementation, however it was not spotted.

C. Conclusions

The Deep Q-Learning approach is very interesting but needs a lot of adjustments in order to work. The performed experiments only considered a level without any adversaries which is not useful for a final Pac-Man agent. This approach could be extended to also include adversaries, by adjusting the state representation. Before that however, it should be made sure that the simple DQNAgent can learn solving a simple level and keep the scores throughout games stable.

V. Discussion & Conclusion

The implemented approaches had shown potential in the context of developing Pac-Man agents. Most notably the Deep Q-Learning and Behaviour Tree approaches, which can produce an agent that can solve various levels. In the case path finding, a simple agent is possible, even with a logic to avoid adversaries, however it is not advanced enough to alone perform as good as the other approaches. The Deep Q-Learning approach could not get a stable score in many of the setups, and more importantly, the current implementation would not work at all in an environment with adversaries, as they would be completely ignored and hard to learn by the algorithm without a proper state representation. The Behaviour tree approach did solve the adversary problem, however the main disadvantage of the approach is that the tree needed a manual setup, thus enforcing a specific behaviour defined by the developer. While this does not necessarily mean that the behaviour tree approach will perform worse, it makes it important to be aware of many edge cases that need to be handled. This problem could be overcome by introducing evolutionary strategies for behaviour tree construction.

References

[1] J. T. Georgios N. Yannakakis, *Artificial Intelligence and Games*. 2018.