Snake Game AI Optimization and Comparison

Scott Gordon
Dept. of Computer Science
University of Massachusetts Lowell
Scott_Gordon@student.uml.edu

Chris Ober
Dept. of Computer Science
University of Massachusetts Lowell
Christopher_Ober@student.uml.edu

Abstract – Snake is a classic game that most people will be familiar with, but for those that aren't, the rules are simple; you play as a snake, and navigate the game board in search of apples. For each apple you eat, your snake increases in length. The game ends when you either run into the end of the game board, or collide with your own tail. Our goal with this project and paper is to investigate and compare different AI algorithms in how efficiently and quickly they can play the game snake.

I. INTRODUCTION

To start out our project, we simply created a playable implementation of the game snake from scratch using C++. Our implementation included a game board of size 10x10 with walls surrounding it. The apples in our game are represented on the grid with an 'a', and are generated randomly at the start of the game and after an apple has been eaten. The snake in our game is represented by two different characters; an 'e' marking the head and an 'o' representing the body/tail. Empty spots in the game are represented by an underscore. Finally the walls are represented by an 'X". For each apple eaten, you would gain one point, and the game would also keep track of the number of steps taken. Figure 1 shows the initial state of the game board as shown to the user. Our implementation gave us a good foundation to implement any algorithm we wanted and just hook it up to the inputs to the game.

Figure 1 – Game Board in the initial state when a game is started and the first move is made.

II. ALGORITHMS

We had many algorithms to choose from to try and figure out which algorithm would be the quickest. We ended up wanting to try an algorithm we could use to experiment with different heuristics, which is how we ended up on A star. We also wanted to experiment with reinforcement learning and ended up implementing Q-learning for that reason.

a. A Star

In our approach of A star pathfinding we created an implementation that was based upon two different heuristics. One being the Manhattan distance, the other being euclidean distance. The snake's tail gave an interesting design to how heuristics would be calculated as it was essentially be a moving wall, and because of this we wanted to test how Manhattan and euclidean distances would differ in getting our snake to a goal state.

For our implementation for our A star search we implemented a heuristic board that modeled the playing board 10x10 that held, for each position, how far away that was using the corresponding equation for each heuristic. For manhattan we implemented this by calculating the distance for the x and y values from the snake head to the apple and then finding the absolute value and assigning that to each board state. We do the same thing for euclidean however we use the formula (snake_x - apple_x)^2 + (snake_y - apple_y)^2. Doing this allows us to now have a heuristic at each board space which we can then use to run our search algorithm with. We accomplished this by creating a list which we stored unexpanded states inside, then doing a search to find the lowest value heuristic inside this list and expanding upon that one. We store all states continuously until we

expand the goal state. As we are expanding states, we keep track of which direction we came from in order to later obtain the path that the snake should follow. When constructing this path, we work backwards from the apple, going to the 'previous state' of each state we are at until we reach the snake head. Then we reverse those movements to finally obtain the steps that the snake needs to take in order to reach the apple. Then the process starts over again with a new apple.

b. Reinforcement Learning

In our implementation of Q-learning we wanted to see if having reinforcement learning would help navigate around walls in a more efficient manner than simply calculating a euclidean or manhattan distance to our end goal.

For our implementation of Q learning it was a little similar to how we handled A star search. At first we define a gamma to be .9 and a learning rate of .1. After this we run the learning for a certain number of episodes that can be adjusted, for ours we chose to run one million episodes. In each episode we pick a random coordinate with a random action and plug it into the Q-learning equation to come up with a q-value. With these q values we then represent the board again but at each state we hold a list that corresponds to [up, right, down, left]. So we then assign the q value we received to the action we took, and do this for our chosen number of episodes. If the action we are taking causes the game to end (IE Hitting a wall or hitting its own tail) we assign the q value a score of -999. After that we can convert the board to actions that the snake is going to take by finding the highest index in the list at each state and taking that action. We can then place the snake anywhere on this board that holds all the q values and go in the direction of the highest q-value in each list.

III. EVALUATION

To evaluate each of the algorithms we implemented, we decided to run each algorithm ten times and average the number of steps and points scored. With these numbers we could get the average number of steps taken per point scored, and also the average number of points per step taken. We utilized these metrics as well as the total points and steps to compare the different algorithms.

IV. RESULTS

We found that all three algorithms performed very similarly, and none of the three stood out as a clear best option. The highest average score was achieved by the A star algorithm utilizing euclidean distance as the heuristic, but it also had the highest average steps taken. In terms of efficiency, the A star algorithm utilizing manhattan distance achieved the best score per step. These results are not too surprising to us since the pathfinding is relatively simple given that the only obstacle is the tail of the snake. In future work we can expand the game board and add obstacles to see how the algorithms perform under more complex scenarios. Table 1 shows these results in more detail to expand on the comparison.

	A star Manhattan	A star euclidean	Q-Learning
Average Score	20.2	21.9	20.2
Average Steps	153.6	179.5	159.2
Score per Step	0.132	0.122	0.127
Steps per Score	7.604	8.196	7.881

Table 1 – Results from running each algorithm ten times, and averaging the results.

V. CONCLUSION

In our evaluations, each of the three algorithms had similar results. The highest scoring algorithm was A Star using euclidean distance, and the most efficient in terms of steps per score was also the euclidean distance A Star. The best algorithm in terms of average score per step however was the Manhattan distance A Star, which implies that it took a better path to each apple, even if it ultimately achieved a worse final score.

We believe The similar results from each algorithm is a result of the limited scope of our game board. Since the board is only ten by ten, there are only so many ways that the snake can reach the apple in any given situation. Similarly, without obstacles other than the snakes' tail, the path to the apple will be similar for each of the algorithms. In order to achieve more meaningful results, in future work we plan to expand the game board to allow for more of a variety in movement from the snake, and also experiment with obstacles such as walls in the middle of the game board to force different paths from the snake.