AI pathfinding in Snake

Elias May

December 12th, 2018

1 Introduction

Snake is a famous video game in which a player controls a snake in a flat playing field. The performance and efficiency of several AI algorithms were evaluated by using them to play the game of Snake. This contained the creation of the game mechanics and five distinct algorithms to play the game. This paper will explain the differences between the algorithms and determine their viability through extensive experimentation and theoretical analysis.

The inspiration for the game Snake can be tied back to a 1976 game called Blockade [1]. The first popular version of the game was created in 1997 when Nokia put Snake on one of its phones called the Nokia 6110. From there, variations of the game became popular and quickly spread.

In the game of Snake, a player uses the left and right arrow keys to turn a snake traversing a grid-like board. The snake moves forward when the player does not hit either key. The goal is to direct the snake to food pieces, which causes it to grow and speed up. Finally, if you manage to expand the snake to cover every square in the board, you win the game.

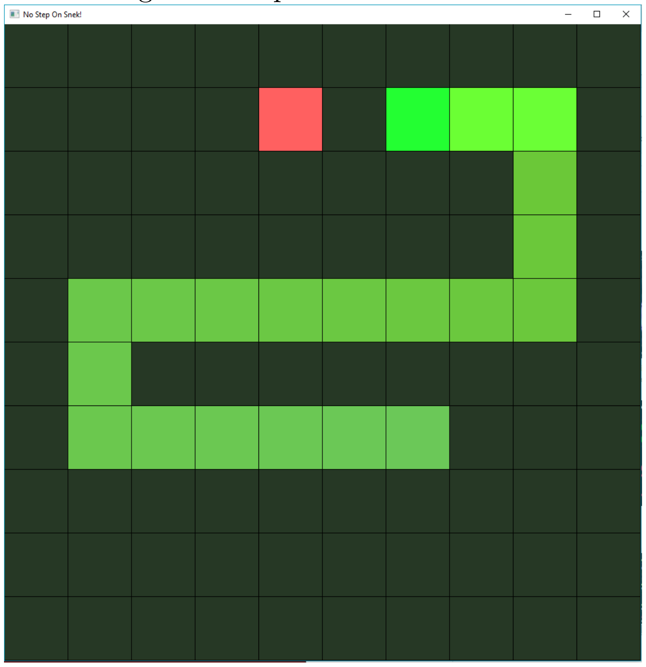
Originally, Snake was designed to be played by humans, but the code has been modified to allow selected algorithms to move the snake for us. This is a good opportunity to test different path finding and search algorithms. Because there are many variations to the rules of the game, the detailed implementation that was used will be covered in Section 2. The algorithms will be introduced in Section 3, and experiments will be covered in Section 4. Section 5 will discuss the experiment results. Finally, future work will be discussed in Section 6.

2 Game Mechanics

This section will cover implementation of the Snake game.

A snake starting with a length of 3 blocks moves across a square board in an attempt to eat as many pieces of food without running into itself or a wall. Upon eventually colliding with itself or a wall, a score is returned.

Figure 1: Implementation of Snake



The following rules are followed by the implementation:

- The final score is calculated as the length of the snake, or pieces of food eaten - 2 since the snake starts at a length of 3.
- The snake can only move orthogonally across the board. This means the snake must be moving either north, south, east, or west. Because the snake will always have a tail piece behind it, it can not change directions at a 180 degree angle.
- As the snake moves forward, the tail follows it. The snake moving 1 space ahead causes to tail to recede by 1 space, unless food is eaten and the snake grows.
- Upon eating a piece of food, the snake grows by one square, which is appended to the end of its tail.
- When the snake eats a piece of food, a new one is randomly placed on the board. The new food can only be placed in an empty board tile.

3 Algorithms

In this section, the five algorithms that are used to play Snake are presented.

3.1 Random Move

This algorithm serves as a base for comparison to other algorithms. Just like the name suggests, the next move for the snake is chosen randomly. Only one exception is made, the snake will try to avoid running into itself or a wall in the next move. The snake will take a long time searching for food as the location of food is not considered in this algorithm. The algorithm can be thought of as a blind snake, not knowing where it is going. This makes it easy for the snake to run into a dead end if it loops around and traps itself with its tail. Although the snake can't kill itself initially, as soon as it randomly finds 2 pieces of food, the tail is long enough to create a dead-end. At this point, it is destined for failure.

3.2 Greedy Algorithm

The Greedy Algorithm is very simple, it always tries to move the snake closer to the food whenever a choice of multiple moves represents itself. In order to do so, it uses the location of the food. Each possible move that doesn't kill the snake is generated, and the Manhattan distance after each move to the food is calculated. Whatever move has the lowest distance is selected. While this algorithm works great in the begging of the game, after the snake grows to the size of 15 or 16 tiles, the greedy algorithm often causes the snake to trap itself.

3.3 A* Search

A* is a computer algorithm that is widely used in pathfinding and graph traversal, which is the process of finding a path between multiple points, called ”nodes”. [2] A* does this by assigning each possible location in a given space a node, while recording the starting position and goal position. In order to find the fastest path to the goal, each movement to a new node is given a cost, which is denoted with G(n). A heuristic function is designed to estimate how close the node is to the end goal, and applied to each node, denoted with H(n). The equation

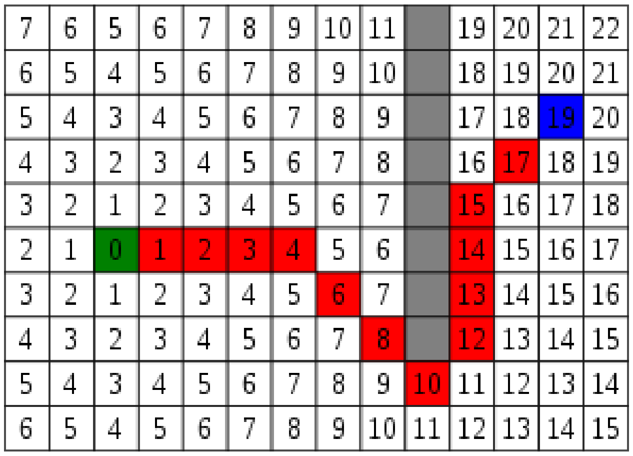
$$F(n) = G(n) + H(n) \tag{1}$$

then determines to cost of moving to each node. In a very simple explanation, the starting node resembling the current location is added to an array. The node with the lowest cost f is popped from the array, and each successor or possible move away from it is generated and added to the array. This process repeats until the goal is found. In figure 2 you can see the order of moves the A* algorithm would take from the green (start) location to the goal (finish) location. Each number inside a tile represents the number of moves it would take to get to that tile.

In Snake, the goal is the location of the food, the start is the snake's head, and obstacles or nodes that are impassable consist of the snake's tail or the border of the game board. Manhattan distance, which is the distance that would be traveled to get from one data point to the other if a grid-like path is followed [3] is what was used as a heuristic function to determine H(n), or how close to the food the snake was estimated to be. G(n) was incremented by 1 for every node generated away from the snake head, as it would take 1 more move to get there.

Without memory or time constrains, this algorithm will always find a path if it exists. This algorithm does not account for the tail of the snake moving when the snake moves, so it will try to move around the tail, making it in-efficient. In the scenario that an path the the food is not found, the move going closest towards the food is executed instead. It is easy for the snake to run into a dead-end using this strategy, as food could spawn in such a way that the snake head is trapped after eating it.

Figure 2: The A* Algorithm



7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

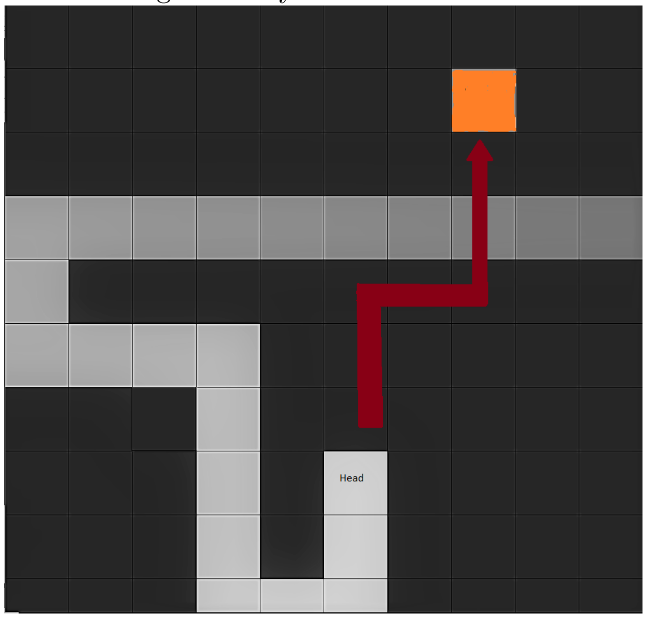
3.4 Dynamic A* Search

This is a slightly modified version of A* Search to allow the snake to predict when its tail is going to move. We account for the tail of the snake following the tail piece before it during each iteration. This lets us plan ahead and move according to where the snake is going to be. In figure 3 you can see the path the snake will take when accounting for the fact that its tail will move to the left as it moves upwards. Normal A* would not be able to find a path to the food in this scenario.

The only change to the code from the regular A* function was the method to generate possible successor nodes from a given node. Each successor node represents a move from one location to the next. In A*, any node that is generated as a successor is discarded if it resembles the location of a snake tail. In Dynamic A*, we instead look at the G value of the node, and if the snake tail is within length G from the last tail piece, the node is allowed to be considered. Since G is the number of moves to get to the given node, the snake tail will recede by G spaces before the snake head arrives.

Although the snake is now a lot more effective at getting to the food in the quickest possible way, the snake still has the same problems as with the normal A* algorithm, which is that it can easily be tricked into a dead-end.

Figure 3: Dynamic A* in action



3.5 Dynamic A* Search with Forward Checking

Dynamic A* Search with Forward Checking tries to tackle the other algorithm's inability to predict walking into dead-ends. To avoid this, it is equipped with a Breadth First Search that is used to check for available moves after the food is eaten. Once an iteration of A* ends, the Breadth First Search algorithm starts at the goal state left from the A* algorithm. Each possible move is expanded recursively to check for a dead end. If no moves are found, a dead end was located, and we discard the result from that particular A* iteration. In the implementation, 20 moves after the food was eaten were analyzed to see if the snake would run into a dead end afterwards.

Another feature of Dynamic A* Search with Forward Checking is its ability to survive when A* doesn't find a path to the food. Instead of deploying greedy search, it uses its BFS search to find the longest possible path to death. After every move, A* is executed again in hopes of now finding a new path to the food. In the implementation, this part of the algorithm could check up to 40 moves ahead.

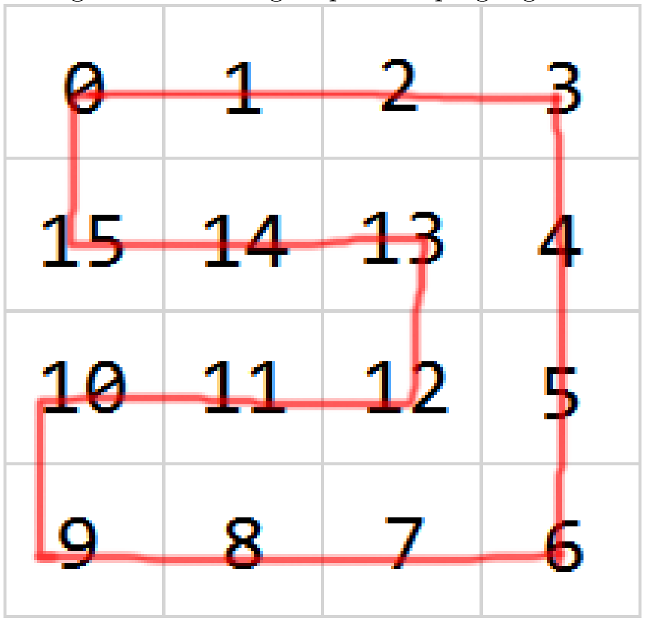
3.6 Longest Path Algorithm

This algorithm is arguably the safest, as it makes the snake immortal. The snake loops around the board continuously, in the same repeating pattern. With this algorithm, the snake will always eat the maximum number of food pieces and win the game by filling the board with itself. The downside is efficiency in terms of moves to get to a new piece of food. Because the same pattern is always followed regardless of where the snake is in regards to food, the snake can move right past the food without eating it. Figure 4 shows an example of what this loop would look like on a 4x4 board. The numbers and red line on each tile represent the path the snake would follow continuously.

4 Methods

Experiments performed consisted of multiple tests for the 6 different algorithms: Random move, Greedy, A*, Dynamic A*, Dynamic A* with forward checking, and looping. Due to time and memory constraints, a 10x10 game board was used to facilitate our testing. This means there are a total of 100

Figure 4: The longest path looping algorithm



tiles, and since the snake starts with a length of 3, 97 pieces of food have to be eaten to win the game.

Because of the randomness of the food being generated, each algorithm was run 1,000 times, and data such as total food eaten and total moves was exported for every iteration. This information is used to determine the viability of each algorithm.

5 Results

Figure 5 displays the experiment results in a table.

- The Random Move algorithm performed just as poorly as expected. It ate about 3.3 pieces of food per trial before dying. It also took an average of 119 moves to get each piece of food. Considering there are only 100 tiles on the game board, this is extremely inefficient.

- The Longest Path Loop algorithm did indeed always eat every one of the 97 pieces of food. It had a very high moves taken per food eaten though. On average, it ate a piece of food every 25 moves. As the game progressed, the moves taken per food eaten significantly decreased as more of the board became part of the snake's tail, leaving less tiles for the food to spawn on.

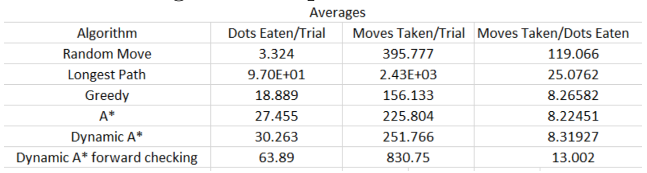
- The Greedy Algorithm ate food quickly at one piece every 8.26 moves, but it didn't get very far after that, eating only 18 pieces of food per trial. It also had the lowest number of moves per game, meaning it was the fastest snake to die.

- A* can be considered a strictly better version of the Greedy Algorithm. It ate food at roughly the same rate of Greedy at 8.22 moves/food, but it stayed alive much longer, eating an average of 30.2 pieces of food before dying.

- Dynamic A* was surprisingly similar to regular A*. The ability to predict the tails movement only marginally increased the snake's food eaten/trial to 30.2 pieces. It also only stayed alive marginally longer with 251.8 moves per trial.

- Finally, Dynamic A* with Forward Checking was a major improvement to A* without forward checking. It ate on average 63.9 pieces of food per trial and stayed alive for 830 moves. That means it took 13 moves per piece of food. Although it still ate less food than Longest Path did, it grew much quicker.

Figure 5: Experiment Results



Averages

Algorithm	Dots Eaten/Trial	Moves Taken/Trial	Moves Taken/Dots Eaten
Random Move	3.324	395.777	119.066
Longest Path	9.70E+01	2.43E+03	25.0762
Greedy	18.889	156.133	8.26582
A*	27.455	225.804	8.22451
Dynamic A*	30.263	251.766	8.31927
Dynamic A* forward checking	63.89	830.75	13.002

6 Future Work

In conclusion, Dynamic A* with Forward Checking can still be improved. A potential idea is to equip it with a method that allows it to find a path to its tail piece when it can't find food. This would allow it to stay alive longer and give it later opportunities to grab the food. Another possibility is to merge Dynamic A* with Forward Checking with Longest Path Loop. The snake would start out using the A* part to grow quickly, then change to the much safer looping approach later on to ensure victory.

References

[1] Smith, Nina M. “A Brief History of Snake.” *Digit*, Digit Www.digit.in, 13 June 2017, www.digit.in/mobile-phones/a-brief-history-of-snake-33913.html.

[2] “A* Search Algorithm.” , Geeksforgeeks, 7 Sept. 2018, www.geeksforgeeks.org/a-search-algorithm/.

[3] “Manhattan.” Distance Metric, Improvedoutcomes, 4 Oct. 2016 www.improvedoutcomes.com/

[4] “Snake.” *Wikipedia*, Wikimedia Foundation, 13 Nov. 2018, en.wikipedia.org/wiki/Snake.