Comparing Pathfinding Algorithms

Michael Lu Han Xien, Garv Sudhir Nair, Anjali Radha Krishna, Yong Tze Min

Department of Science and Technology, Sunway University
Subang Jaya, Malaysia
18081588@imail.sunway.edu.my
19073535@imail.sunway.edu.my
16009847@imail.sunway.edu.my
19079748@imail.sunway.edu.my

***Abstract* - This document contains a study on which of the path finding algorithms performs best in the environment of the Snake game. Each approach is implemented through Python and is analyzed to determine the pros and cons of the algorithms in the scenarios.**

***Keywords* – Pathfinding Algorithms, Artificial Intelligence, Python, Uniform-Cost Search, A* Search**

I. INTRODUCTION

This report aims to compare the efficiency of two different pathfinding algorithms in a simple Snake game environment with variable conditions. The two algorithms compared are the uniform-cost and the A-star searching algorithms. This report will go over the concept, implementation, and discussion of the algorithms.

The game of Snake is a popular video game that originated from an arcade game known as Blockade [1]. The objective of the game is for the player to control an object, known as the snake, on a plane and search for food. Upon reaching and consuming the food, the length of the snake is increased, and a new food is randomly generated on the plane. The game is lost when the snake hits the borders or intercepts its own body. The only win condition is to keep eating and growing until the snake covers the entire area of the plane. The game is a simple concept that gets progressively more difficult the longer the snake's body becomes and the less space there is to manoeuvre through.

As with this concept, there are methods in which an autonomous game of Snake can be conducted with the help of pathfinding algorithms. However, the efficacy and function of each algorithm vary from one another. To determine which is better, two different algorithms shall be tested on the game and note how well they respond and carry out the program.

II. ALGORITHM CONCEPTS

The first algorithm is known as uniform-cost search (hereafter referred to as UCS), also known as a variant of the Dijkstra's algorithm [2]. The algorithm functions by only exploring the closest frontier until the goal node is explored. When exploring a frontier, the child nodes are added into a priority queue of frontiers where they are sorted by their traversal cost (also known as path cost) from the starting node. Every explored node path is confirmed to be the shortest path from the starting node. Therefore, if the child node of a frontier has already been explored, it is forgotten as any new path discovered for an explored node is guaranteed to be longer. If the child node of the frontier has already been added into the frontier queue, the path cost of the similar nodes is compared. The node with the least path cost is kept while the other is forgotten. The process of adding, deleting, and sorting frontier nodes is repeated for every expansion. When the goal node is discovered, the algorithm treats the goal node like any other frontier and continues running. Any other path to the goal node discovered will be compared to the one in the queue and the longer path is forgotten. When the goal node reaches the front of the queue and is explored, the algorithm ends. This helps to guarantee that the path discovered will be the shortest path from the starting node to the goal node.

The time complexity of UCS can be summarized as follows:

$$O(n^2)$$

The second algorithm is known as the A star search (hereafter referred to as A*) which functions similarly to the UCS algorithm. Unlike the UCS algorithm, the A* is an informed search algorithm. The difference between the two is that informed search algorithms make use of existing information, often referred to as a heuristic, to reach the goal state which in turn helps in improving search efficiency [3][4]. In this case, both UCS and A* algorithms take into consideration the cost of each branch but A* search adds an estimated cost from node n to the goal node. The cost is calculated as follows:

F(n) = Estimated cheapest cost to the goal

G(n) = Estimated cost from the starting node till n

H(n) = Estimated cost of n till the goal based on Heuristic function

$$F(n) = G(n) + H(n)$$

H(n), also referred to as the heuristic function, would then be used to sort the frontiers in the priority queue rather than just the cost of the frontier from the initial node. This method of search takes into consideration the cost before and after the expansion so that the shortest cost path is always chosen and prevents expansion in the wrong direction.

The time complexity of A* can be summarized as follows:

$$O(b^d)$$

Where *b* refers to the "branching factor", i.e., the number of children at each node, and *d* being the depth of the occurring search tree [5].

III. ALGORITHMS IMPLEMENTATION

The implementation of both UCS and A* algorithms within our code primarily depends upon three common parts with each aspect of the code further explained alongside their components and functions.

A. `Player` class

The first part of the game is to initialize the Snake game with a `Player` class. When the game begins and the code begins running, it creates a `Player` object where details about the algorithm itself (name, informed search, etc.) and details obtained about the game (maze size, static snake length, etc.) are initialized.

B. `Node` class

Each object of class `Node` contains important details, such as unique id, position, cost, children, parents, etc., and are used to keep track of the expansion path. In both algorithm's code, the `Node` class contain a `remove()` function and `expand()` function. The purpose of the `expand()` function is to explore and generate child nodes of the current nodes. The purpose of the `remove()` is to mark the node as forgotten when the node is not qualified for expansion.

C. `run()` function

The `run()` function exists within the Player class and is the meat of the program as this is where the algorithm is implemented and how the program will run the algorithm. The `run()` function takes in values from the game, such as snake locations, food locations, etc., and returns the direction in which the snake needs to move to as well as the search tree for the searching process.

The major deviation between both algorithms emerges when implementing A* due to the existence of the heuristic function mentioned in the previous section. To efficiently calculate a heuristic cost, the Player class is initialized with a two-dimensional array `grid[]` and makes use of two new functions `generateGrid()` and `spreadFoodDistance()`. The purpose of these new functions is to generate a replica of the snake maze and estimate the distance from the food for every node.

2	1	●	1	2	3	4	5	4	5
3	2	1	2	3	4	5	4	3	4
4	3	2	3	4	5	4	3	2	3
5	4	3	4	5	4	3	2	1	2
6	5	4	5	4	3	2	1	●	1
›	6	5	6	5	4	3	2	1	2
8	7	6	7	6	5	4	3	2	3
9	8	7	8	7	6	5	4	3	4
10	9	8	9	8	7	6	5	4	5
11	10	9	10	9	8	7	6	5	6

Fig 1. Grid values generated when game begins.

12	›	›	9	8	7	6	5	4	5
11	10	9	8	7	6	5	4	3	4
10	9	8	7	6	5	4	3	2	3
9	8	7	6	5	4	3	2	1	2
8	7	6	5	4	3	2	1	●	1
9	8	7	6	5	4	3	2	1	2
10	9	8	7	6	5	4	3	2	3
11	10	9	8	7	6	5	4	3	4
12	11	10	9	8	7	6	5	4	5
13	12	11	10	9	8	7	6	5	6

Fig 2. Grid values generated after few iterations.

The `generateGrid()` function is called every iteration of the algorithm, effectively changing the values of grid at every move. When multiple food locations are present in `grid[]`, it favors the smallest possible value for each individual slot and overwrites the value of the slot given it finds a more optimal heuristic cost (compare grid from fig.1 and fig.2). Using the `grid[]`, the `Node` could simply access the next node position to obtain the grid value and add them to the heuristic function.

IV. TESTING THE ALGORITHMS

The algorithms were tested on a 10 by 10 grid with each of the game settings adjusted according to the cases and their requirements. Runtime was not an immediate factor in these tests as they were running at a fixed pace of 10 steps per second, thus, running at the same time regardless of how long the games lasted. While runtime is not an immediate factor, it should be noted that there was some noticeable slowing down as the program ran. Scores and average elapsed time were used for comparison for more accurate data. Elapsed time is the time it takes for the algorithm to calculate and obtain the results. Thus, the elapsed timer begins when the `run()` function is called and stops right before the results are returned.

First Case: *Fixed Length, One Food Generated.* In this case, the goal was to reach at least 15 points for both algorithms. The algorithms were only tested once as there was no chance of the snake dying. Therefore, in this case, the time and speed of the

snake pursuing the food points were considered for analysis instead. Each search was left to reach a maximum of 150 points.

A. *Uniform-cost Search*: The average elapsed time for the algorithm is 2.361 milliseconds. However, the program can be observed to slowdown heavily as the program ran.

B. *A* Search*: The average elapsed time for the algorithm is 11.003 milliseconds. Similar slowing down can be observed as the program ran. However, the slowdown was not as intense as in the case of UCS.

Second Case: Dynamic Length, One Food Generated. In this test case, the goal was to reach at least 10 points for both algorithms. The program ran seven (7) times consecutively, resetting at each death to find the number of points and average elapsed time.

Run	UCS	A* Search
1	23, 2.269ms	32, 11.048ms
2	38, 2.264ms	19, 11.238ms
3	30, 2.363ms	27, 11.453ms
4	13, 1.785ms	32, 11.674ms
5	18, 2.377ms	39, 11.885ms
6	25, 2.133ms	27, 12.471ms
7	30, 2.512ms	31, 13.302ms
Average	25.29, 2.243ms	29.57, 11.867ms

TABLE I: Second Case results

A. *Uniform-cost Search:* The number of points collected by the UCS algorithm over seven rounds is 177 points. The number of points collected on average per round is 25.29 points. The lowest number of points collected by the snake in a single round is 13 points. The speed of the algorithm is consistently averaging around 2.243 milliseconds per iteration.

B. *A* Search:* The number of points collected by the A* algorithm over seven rounds is 207 points. The number of points collected on average per round is 29.57 points. The lowest number of points collected in a single round is 19 points. The speed of the algorithm is consistently averaging around 11.867 milliseconds per iteration.

Third Case: Dynamic Length, Two Food Generated. In this case, the goal was to reach at least 10 points for both algorithms. The program ran seven (7) times consecutively, resetting at each death to find the number of points and average elapsed time.

Run	UCS	A*Search
1	29, 2.231ms	23, 13.480ms
2	37, 1.837ms	31, 12.611ms
3	19, 2.088ms	37, 12.171ms
4	15, 1.327ms	36, 12.300ms
5	44, 1.255ms	27, 12.049ms
6	24, 1.315ms	25, 11.531ms
7	29, 1.990ms	17, 12.069ms
Average	28.14, 1.720ms	28, 12.316ms

TABLE II: Third Case results

C. *Uniform-cost Search:* The number of points collected by the UCS algorithm over seven rounds is 197 points. The number of points collected on average per round is 28.14 points. The lowest number of points collected by the snake in a single round is 15 points. The speed of the algorithm is consistently averaging around 1.72 milliseconds per iteration.

D. *A* Search:* The number of points collected by the A* algorithm over seven rounds is 196 points. The number of points collected on average per round is 28 points. The lowest number of points collected in a single round is 17 points. The speed of the algorithm is consistently averaging around 12.316 milliseconds per iteration.

V. DISCUSSION AND ANALYSIS

Program Speed: In the first test case, both algorithms can survive for a very long time and could theoretically keep running forever. However, the reason behind the observed slowing down is unknown and no solid hypothesis about the algorithm code could be made to explore any reasonings. It could be possible that the reason behind the slowing down is the Snake game platform itself. This could be a solid hypothesis as when comparing the average elapsed time for the second and third test cases, where the program's speed was more consistent with little to now slow down was observed, with the first case for both algorithms, the average elapsed time is almost similar even though the program is significantly slower for the first test case.

Score Comparison: In the second test case, A* search obtained a higher average score before dying compared to UCS. However, in the third test case, both results were very similar. A solid hypothesis can be made that due to the nature of how the food location was generated, the scores are highly dependent on luck. Depending on the placement of the food, the worst-case scenario could cause the generated food to create a circular path to loop the snake around then generate a food within the circle of the snake loop. Thus, trapping the snake and killing it early on. In fact, this worst-case scenario is the only scenario capable of causing the snake to trap itself within its body or against a border. Thus, the hypothesis behind the scores being dependent on luck could be proven true.

Algorithm Speed: By theory, A* search should be obviously much faster than UCS. However, from all three test

cases, all results show that A* is consistently about five times slower than UCS. A hypothesis could be made that the significant increase in time could be due to the `generateGrid()` function. To confirm this theory, all three test cases were repeated but this time, the timer begins after the grid was generated. As predicted, the average elapsed time was recorded to be 0.544ms, 0.585ms, and 0.556ms respectively, more than three times faster than UCS. Thus, proving the hypothesis that the `generateGrid()` function was the cause of the significant increase in time and that A* search is in fact faster than UCS.

VI. CONCLUSION

In conclusion, two different algorithms, namely the Uniform Cost Search and A-star Search, were implemented into the game of Snake to observe and analyze the difference between the two algorithms. After multiple tests being run on both pathfinding algorithms, all the requirements and challenges were fulfilled. In the first test case, the program slowed down significantly when it ran for a long period of time. However, it was inferred that it was not caused by the algorithms themselves, as when the elapsed time of the algorithm was compared to other more consistent test cases, the elapsed time was found to be quite similar. Scores obtained by both algorithms were not a good factor for comparing performance as it is highly dependent on luck. A surprise occurred when comparing the average elapsed time for all three test cases. The A* algorithm, which was theoretically faster than UCS, performed five times slower than the UCS algorithm. It was later discovered that the algorithm is in fact more than three times faster than UCS but due to external implementation issues (i.e.: the `generateGrid()` function), performed much slower.

REFERENCES

[1] G. Goggin, Global Mobile Media, Illustrated ed., Taylor & Francis, 2010, p. 101.

[2] m. k. 2. r. andrew1234, "Uniform-Cost Search (Dijkstra for large Graphs)," 3 February 2021.

[3] R. Belwariar, "A* Search Algorithm," 1 February 2021.

[4] t. pp_pankaj, "Difference between Informed and Uninformed Search in AI," 26 May 2021.

[5] P. N. Stuart Russel, Artificial Intelligence: A Modern Approach, 3 ed., Prentice Hall, 1995, p. 103.