AI Assignment 1

Group Name: 1G3I

Group Members:

Ng J-Han	18042507
Koh Jia Yi	17063470
Gan Kai Wen	16007940
Peng Shao Wei	19084227

1. Introduction

In this report, we will address the problems associated with the snake-game from a perspective of solving it using search algorithms. Then, we will discuss how we managed to implement BFS and A* search to tackle the snake game by making the snake search for the food on its own. Finally, we will compare both algorithms to see which algorithm is more performant and efficient. We intend to discuss why this is so and consider the possible tweaks and additions that can be made to improve performance.

1.1 Maze size

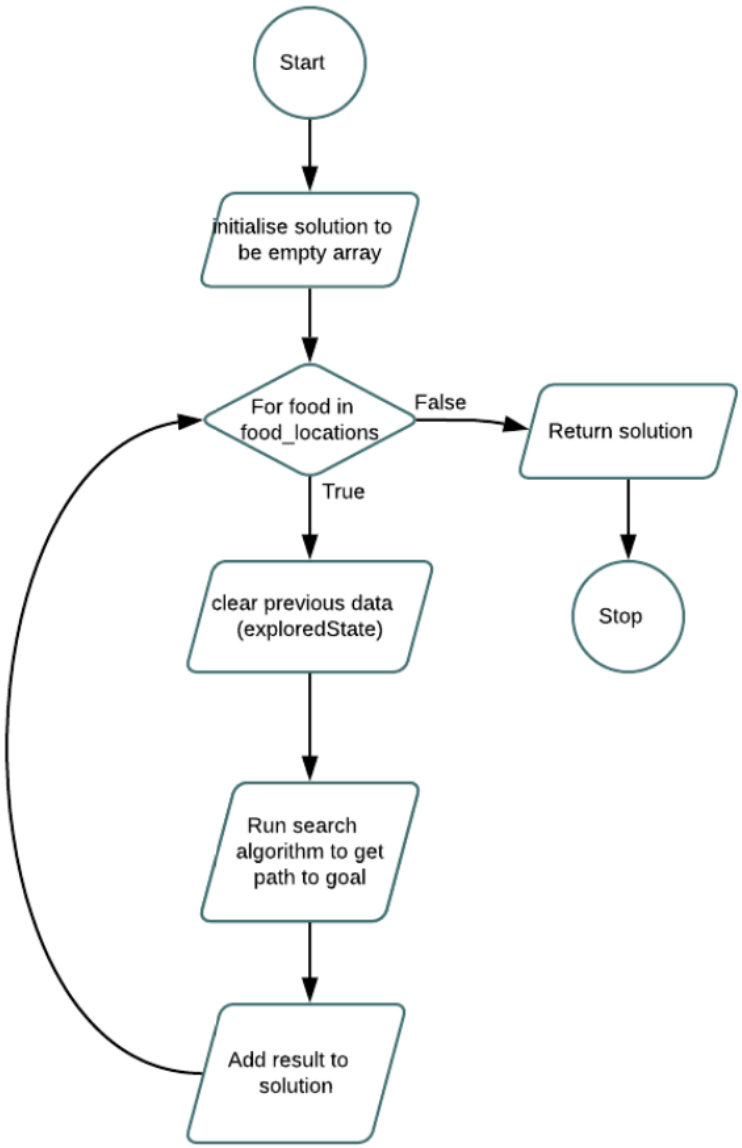
In this assignment, the maze size is a parameter that adjusts the size of the board. The maze size is dynamic which means it can be adjusted by the user in the UI. By default, the maze size is 10 x 10.

1.2 Snake

The snake will be moving on the board and looking for food. The goal is to let the snake eat as much food as possible without hitting itself or the walls. The snake does this by taking into account the current locations of its body and direction to determine the potential moves it can make without bumping into a wall or intersecting with itself. Once the foods are eaten, new foods will be spawned randomly on the board. The snake will also grow by one unit for every food it eats. The initial location of the snake is at [0,5] and the snake will travel based on the algorithm implemented. The game will end when the snake has nowhere to go other than hitting himself or a wall. Each food eaten is counted as 1 mark and the final score is returned when the game ends.

1.3 Food

Our code is made to handle multiple food locations which are adjustable by the user.



Above represents the logic behind handling multiple food locations.

2. Problem Description

The snake location will be initiated, and the food location will be spawned randomly. The snake will need to move to the food location spawned each time without being controlled by the user. We need to find the path of the given snake location to the spawned food location using an uninformed and informed search algorithm. For each time the snake eats the food successfully, one point will be awarded, and the length of the snake will be grown by one. The snake should be able to obtain at least 10 points, in other words, at least 10 foods should be eaten by the snake.

2.1 Problem Formulation

How BFS and A* can be implemented to find the correct path from the current location to the food location.

3. Algorithm

In this assignment, we decided to use breadth first search as the uninformed search and A* search as the informed search to automate the snake. We will cover the reasons why we chose these specific algorithms in more detail down below.

3.1 Breadth First Search

Breadth First Search Algorithm consists of two arrays, which are frontier, and explored. This algorithm expands from a root node and traverses the graph layer wise thus exploring and adding the neighbour nodes(children) into the frontier. The BFS algorithm traverses the graph breadthwise as it moves horizontally and visits all the nodes of the current layer, then only it moves to the next layer.[1] Therefore, the next node that will be expanded is the first node in the frontier. Once a node is explored and expanded, it will be removed from the frontier and added into explored. If the children of the node are previously explored, it will not be added to the frontier as it is a redundant path. Once the goal is found, it will stop expanding and return the path of starting point to the goal state. We decided to use BFS over DFS because we felt that since the search tree for this snake-game problem is rather deep, DFS would take a longer time to process and traverse through the tree to find the goal node as compared to BFS.

3.1b Implementation of the BFS algorithm

In this case, we implemented BFS on the snake. The root node will be the starting point of the snake. The algorithm expands from the snake until it reaches the goal state which is the food location. Once the path is found, the snake will start moving to the food and eat it. When the snake eats a food, the nodes in frontier and explored will be cleared and the algorithm restarts. The game will end when the snake has no other choice than biting himself or hitting a wall.

To illustrate this algorithm, we will travel from snake location [0,5] to location [2,4].

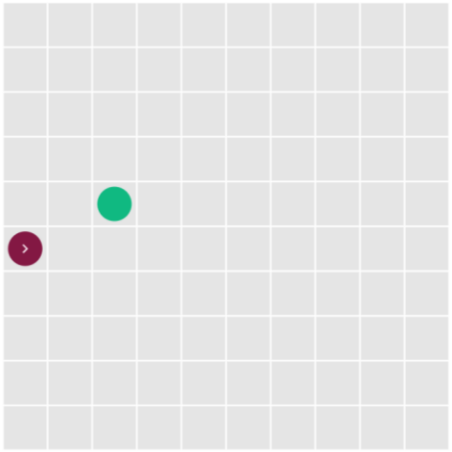


Figure 1

The root node will be [0,5] and its neighbour nodes, which are [1,5], [0,4],[0,6], will be added into the frontier list. [0,5] will now be added into the explored list.

Frontier = [[0,4], [0,6], [1,5]]
Explored= [[0,5]]

Figure 2

The first node in the frontier list is [0,4], therefore we start exploring from [0,4]. We expand and add its neighbouring nodes into the frontier, which are [1,4],[0,3],[0,5].Since [0,5] is already explored, it will not be added to the frontier as it is a loopy path. Once [0,4] is explored, it will be removed from frontier and added to the explored list.

Frontier = [[0,6], [1,5], [1,4], [0,3]]
Explored= [[0,5], [0,4]]

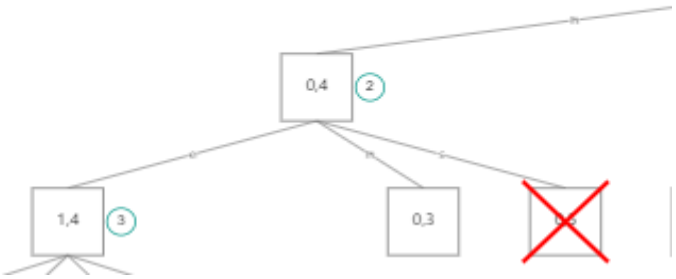


Figure 3

The next node in the same level of [0,4], or the first node in the frontier list is [0,6]. The neighbouring nodes of [0,6] is [1,6], [0,5], [0,7], since [0,5] has already been explored, we will only be adding [1,6], [0,7] to the frontier. Remove [0,6] from the frontier and add it into the explored list.

Frontier = [[1,5], [1,4], [0,3], [1,6], [0,7]]
Explored= [[0,5], [0,4], [0,6]]

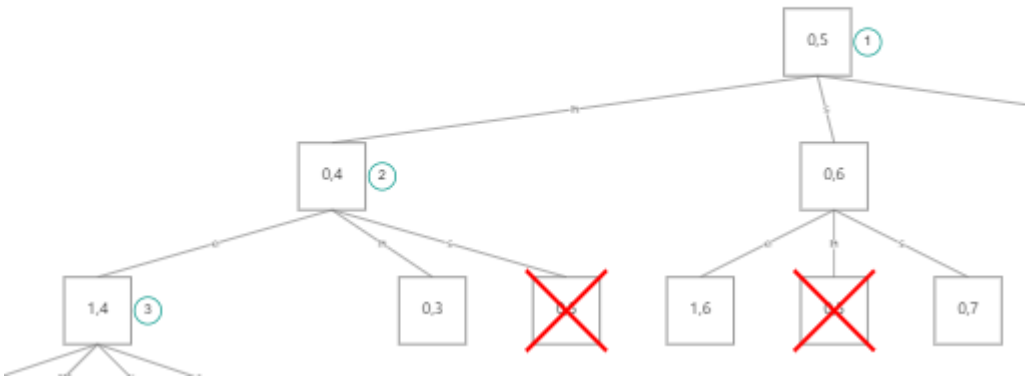


Figure 4

The next node in the same level or the first node in the frontier list is [1,5]. The neighbouring nodes of [1,5] is [2,5],[0,5],[1,4],[1,6] since [0,5] has already been explored and [1,4],[1,6] is already in the frontier, we will only be adding [2,5] to the frontier. Remove [1,5] from the frontier and add it into the explored list.

Frontier = [[1,4], [1,6], [0,3], [0,7], [2,5]]

Explored= [[0,5], [0,4], [0,6], [1,5]]

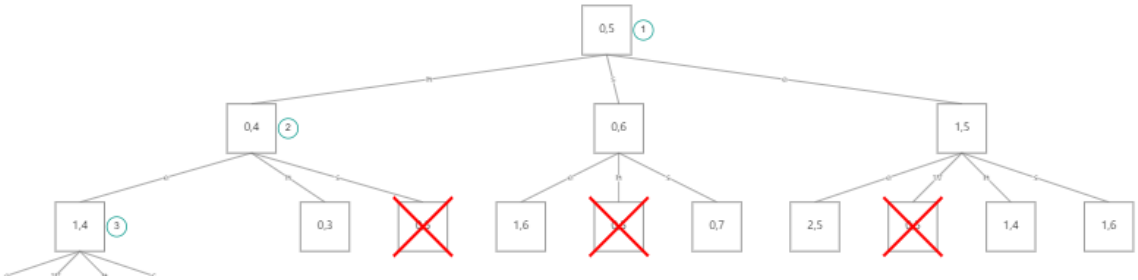


Figure 5

The first node in the frontier list is [1,4]. The neighbouring nodes of [1,4] is [2,4],[0,4],[1,3],[1,5] since [2,4] is the goal state, the algorithm will stop and return the path. Once we reach the goal state, the frontier and explored list will be emptied and the algorithm restarts with [2,4] as the root node to find the second food. The directions from [0,5] to [2,4] will be stored in an array and the snake will follow the sequence in order to reach the food.

Path = [[0,5],[0,4],[1,4],[2,4]]

Solution = [‘n’,‘e’,’e’]

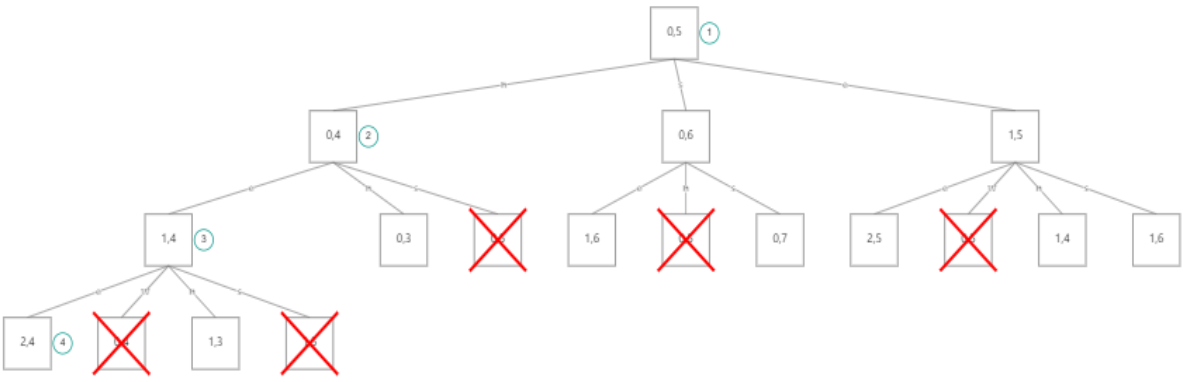
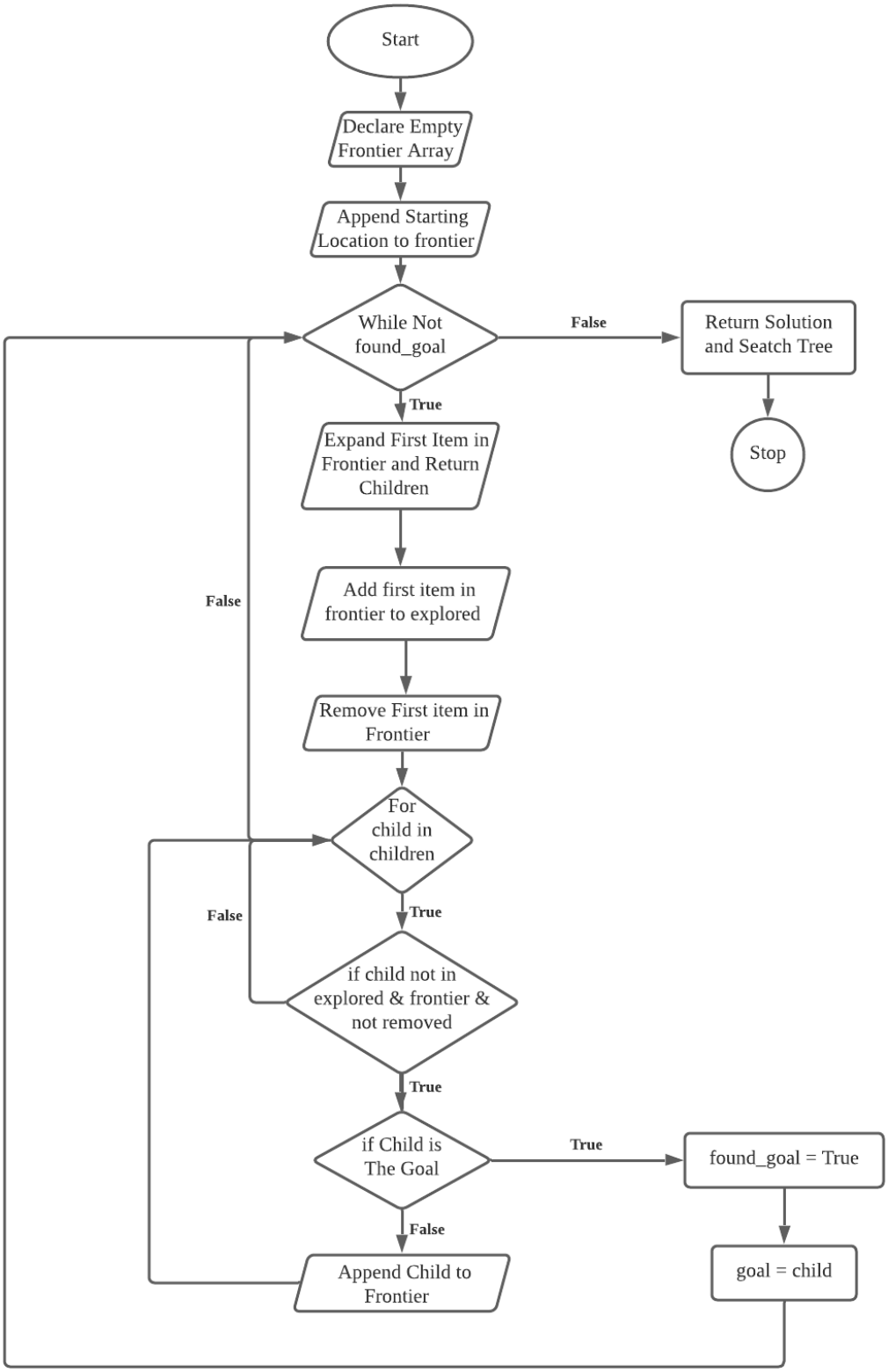


Figure 6

BFS FLOW CHART



Above is the flow chart of the BFS program. This represents the overall flow of the program throughout its execution.

3.2 A* Search

A* search is an informed algorithm that is originally derived from best-first search. It shares a lot of similarities with its sibling algorithm, greedy best-first search, with one key difference being the inclusion of the path cost from the start node to the current node being visited (denoted by g(n)) on top of the cost path given by a heuristic function (denoted by h(n)). The A* algorithm can be broken down into a few general steps. Firstly, the algorithm considers the starting node and the goal node. Then, starting from the initial node, the algorithm expands each node and keeps track of the estimated cost from the current node to the goal node as well as the cost from the starting node to the current node for each child node. Then, based on the information that can be derived from these 2 factors, the algorithm will sort the children in the frontier such that the child nodes with the lowest costs will be expanded first. The algorithm will then rinse and repeat these actions until the goal node is discovered upon **expansion**. This is important to note, because there will be cases in which the path itself is longer, but the cost is shorter, so determining if a node is a goal node during generation instead of expansion would give you an invalid solution.

Both the A* search algorithm and the Greedy best-first search algorithm share the same basic principles, which means they both function in very similar ways. However, the difference between greedy best-first search and A* search lies in its accuracy. Because the A* search considers the additional information given by g(n), it is both complete, meaning to say that there is little to no risk in taking a path that does not result in the goal node, and optimal, which means that the path found has a higher probability of being the optimal one [2]. Therefore, we decided to implement the A* algorithm over Greedy best-first search algorithm to solve our problem.

3.2b Implementation of the A* algorithm

To implement the A* algorithm to solve our snake-game problem, we must provide the algorithm with the necessary parameters in order for it to understand the problem and function optimally. To do so, we can make a few connections. Among them are:

- The initial node is analogous to the initial snake location on the board and the goal node is analogous to the food location on the board.
- The possible children that can be generated during expansion is analogous to the directions that the snake can travel given the snake's current locations on the board. For example, if the snake is at [0,5], the snake can travel, north, south, and east. As a result,

the children's nodes will be [1,5], [0,6], and [0,4]. The snake cannot travel west in this case due to there being an obstruction in the form of a wall. Moreover, if the snake grows in length, the body of the snake will also be counted as an obstruction to ensure that the algorithm only generates children's nodes that do not intersect with the locations of the snake on the board. Thus, allowing the snake to avoid eating itself.

- The path cost from the starting node to the current node (g(n)) is analogous to the moves taken between the current position of the snake on the board and the initial position of the snake. For example, if the snake's initial position is [0,5] and its current position is [2,5], then the value of g(n) will be 2, because the snake must travel 2 steps east to get to [2,5].
- The estimated cost of the current node to the goal node (h(n)) is analogous to the Manhattan distance between 2 points on the board. Manhattan distance in this case refers to the sum of the absolute differences between two vectors. It is given by the formula: $|a-c|+|b-d|$. For example, if the food location is [2,4] and the snake is at [0,5], then the value of h(n) will be 1, because the absolute value of [0-2] + [5-4] = 3.

Using all of the above connections, the algorithm will be kept up to date on the state of the game with each turn and will be able to decide the order of which nodes are expanded till it expands the goal node.

To illustrate the operation of the algorithm, we will travel from [0,5] to [2,4]

Frontier = [[0,4], [1,5], [0,6]]
Explored = [[0,5]]

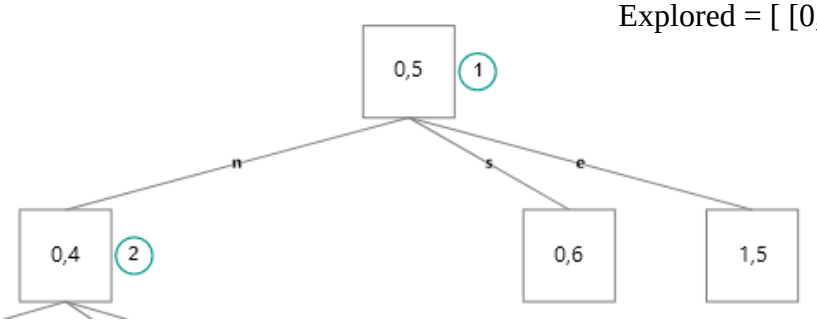


Figure 7

First, the root node will be expanded. Based on the location, the algorithm determines the possible directions to travel and generates the children nodes [0,4] for north, [0,6] for south, and [1,5] for east. For this expansion, the algorithm cannot travel west because there is a wall. Then, the algorithm will sort the frontier to ensure that the child with the lowest cost path is expanded first. In this case, [0,4] has the lowest cost path, so the algorithm will expand that node first, and add the children of that node into the frontier. Once again, because [0,4] is still against a wall, the algorithm can only expand north, south, east.

Frontier = [[1,4], [0,3], [1,5], [0,6]]

Explored = [[0,5], [0,4]]

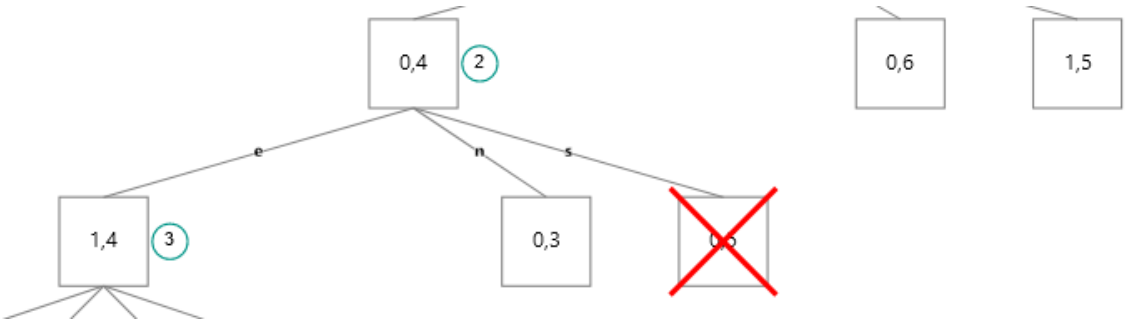


Figure 8

Then, the algorithm will once again sort the frontier and continue to expand the node with the lowest cost path. In this case, the node [1,4] has the lowest cost path, so the algorithm will expand that node.

Frontier = [[2,4], [1,3], [1,5], [0,3], [1,5], [0,6]]

Explored = [[0,5], [0,4], [1,4]]

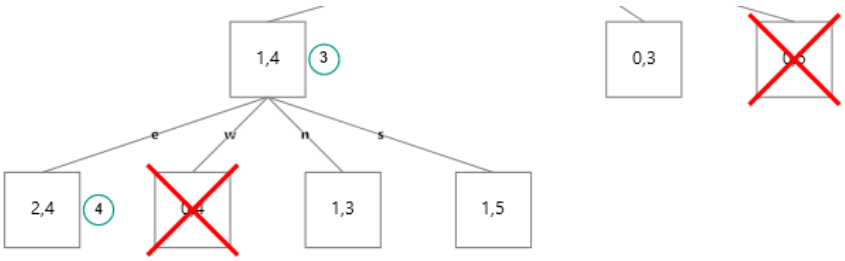
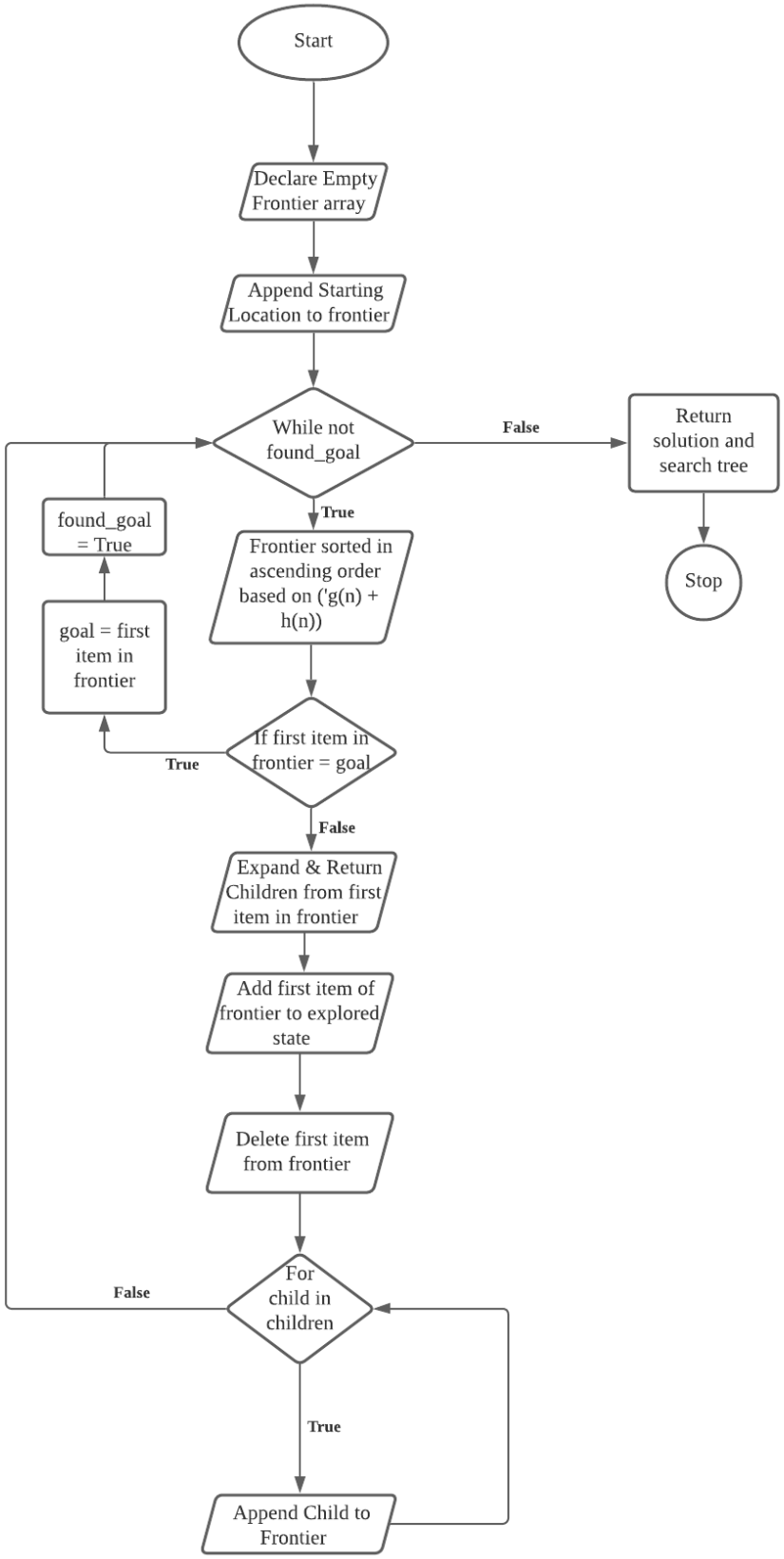


Figure 9

Finally, the algorithm sorts the frontier and expands the node [2,4] and realises that this is the goal node. The algorithm then traces back from this node to the initial node to return the solution.

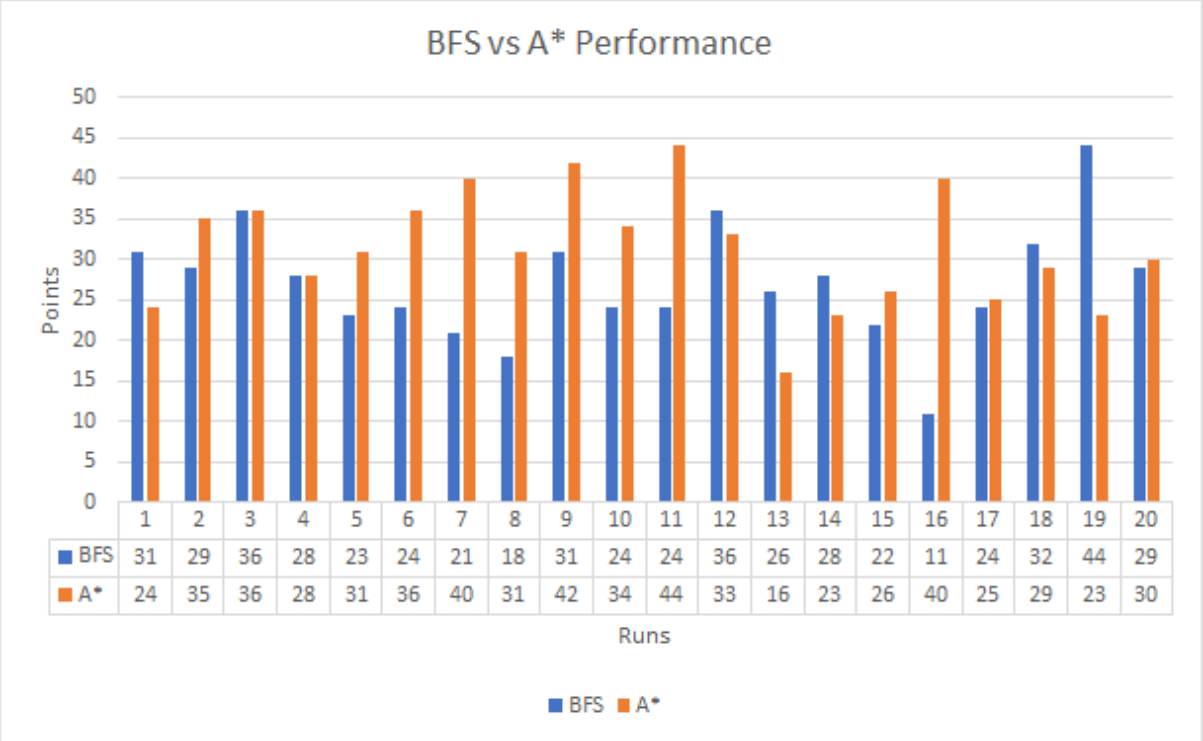
A* Flow Chart



Above is the flow chart of the A program. This represents the overall flow of the program throughout its execution.*

4. Experiment and Results

In order to obtain an average result, we ran each algorithm 20 times with a board size of 10 x 10 and each run was recorded. Figure 12 shows the score of 20 runs.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
BFS	31	29	36	28	23	24	21	18	31	24	24	36	26	28	22	11	24	32	44	29
A*	24	35	36	28	31	36	40	31	42	34	44	33	16	23	26	40	25	29	23	30

Figure 10: Performance between BFS and A*

Based on the graph above, we can see that the A* will have a higher average score which is **31.3** compared to BFS which only have an average score of **27.05**. This indicates that A* is much more consistent in finding the best solution.

5. Discussion and Evaluation

Based on the search tree that is generated by both algorithms with the same initial node and goal node which is [0,5] for the initial node and [2,4] for the food location, we can see that A* search is more efficient compared to BFS as A* search expands fewer nodes compared to BFS. This is because A* search has a heuristic function which serves as the guide for the snake to find the shortest path to the food location without expanding all of the nodes. However, BFS doesn't have any prior knowledge about the nodes so it will expand all of the nodes until the goal node is found.

As an analogy for these two algorithms, say that in a scenario whereby there is a mountain and a goal to find gold with a robot, with BFS, the robot will have no prior knowledge about the gold location on the mountain so it will simply dig 10cm deep within a radius. If the gold isn't found within the radius, the robot will dig another 10cm deeper until it finds the gold. As for A* search, the metal detector is built-in originally so the robot can detect the food location using the metal detector. However, the efficiency of the robot to find the gold depends on the quality of the metal detector where the higher the accuracy of the metal detector, the faster the robot can find the gold. In this case, the snake indicates the robot, the food location indicates the gold location, and the metal detector indicates the heuristic function. However, the A* search algorithm is only as good as the heuristic function provided. If the heuristic function is poorly defined, A* search may perform worse than BFS. This is because a poor heuristic function may lead the algorithm to look for the wrong search tree and expand unnecessary nodes which will cause the algorithm to look for a longer path instead of the shortest path.

Based on the experiments and analysis above, there is an improvement that we can make in order to enhance the algorithm's performance while tackling this snake-game problem. We can add a forward checking for the algorithm to avoid a dead-end path. Forward checking is an algorithm which detects the inconsistency in the search tree and cuts off the branches of the search tree which may lead to failure [3]. This allows the algorithm to look for the shortest and most suitable path which will ignore the path that will lead to a dead-end.

6. Conclusion

In this report, we have defined and addressed the problems that need to be solved to implement search algorithms. The biggest problem we faced was how to convert the parameters of the game into information that the algorithms could use. After defining the problem, we managed to implement BFS and A* search and discovered that the A* search algorithm is far more efficient than BFS due to the informed nature of the A* search algorithm that includes a heuristic function that allows the algorithm to more accurately determine the best path to take based on the cost of the path rather than the total destinations along the way.

References

[1] https://en.wikipedia.org/wiki/Breadth-first_search

[2] Russell, S. and Norvig, P., 2016. *Artificial intelligence*. 3rd ed. Harlow: Pearson, p.92.

[3] Roman Bartak, 1998, "Guide to Propagation"