SCHOOL OF ENGINEERING AND TECHNOLOGY

COURSEWORK FOR THE BSC (HONS) COMPUTER SCIENCE

ACADEMIC SESSION APRIL 2021;

CSC 3206: ARTIFICIAL INTELLIGENCE

DEADLINE: 21ST MAY 2021 (FRIDAY), 5:00 PM

NAME	STUDENT ID
Yan Yi Cheng	17109505
Ian How Yu Xuan	18032649
Christine Law Han Ni	18056796
Bryan Eeo Zhe Yu	20016960

Navigating Snake Game using Search Algorithms

1.0 Introduction

This report introduces 2 Python programs developed with the purpose of completing 3 different challenges of a snake game using 2 different search algorithms, one being an uninformed search and another being an informed search. There are a multitude of uninformed search algorithm and informed search algorithm that exists as of the current times. However, Breadth First Search (referred to as BFS in future usage) and Greedy Best First Search (referred to as Greedy Search in future usage) was chosen to be uninformed and informed search algorithm respectively for the 3 challenges that are to be tackled. The reasoning and justification for the algorithms chosen will be further elaborated in their respective sections. The 3 challenges to be tackled are:

1. Traversing the maze with one food at any time as a fixed snake length of one and achieving at least 15 points.
2. Traversing the maze with one food at any time as an increasing snake length and achieving at least 10 points.
3. Traversing the maze with two foods generated when no food is available in the maze as an increasing snake length and achieving at least 10 points.

* Note: Each food eaten is counted as 1 point.

BFS, as mentioned above is an uninformed search. BFS is a simple yet effective strategy whereby the root node is expanded first, then all the successors of the root node are expanded next. It works in a similar manner to a FIFO queue. This would almost ensure that the goal node will always be reached albeit incurring a relatively high search cost.

On the other hand, Greedy Search is an informed search whereby it uses problem-specific knowledge beyond the problem definition. This knowledge is derived from past experience and common knowledge. Greedy Search as opposed to BFS would incur only a minimum search cost as it expands only the path to the goal. The implementation of these search algorithms will be further discussed in the following sections of this report.

The remainder of the report is as follows, implementation of BFS in Section 2, followed by implementation of Greedy Search in Section 3. Section 4 discusses on the performances and results of the algorithms. Lastly, Section 5 concludes the paper.

2.0 Breadth-First Search

This section will contain 3 subsections with the first one mainly covering on the problem description, problem formulation and how BFS is implemented to tackle the problems. The following subsection would be a flowchart as a visualization of the algorithm for better understanding. A sample of the search tree will then be showed in the final subsection to prove that the algorithm is indeed BFS.

2.1 BFS Implementation

Before we go into the implementation of the BFS algorithm, we would have to first understand the problem at hand that is to be tackled. We have,

Initial State, of [0, 5] in a 10×10, zero-indexed maze.

Actions, that are available to us is to move in the direction of North, South, West and East. Or easier understood as up, down, left and right.

Transition Model, would bring us to a new location within the maze based on the actions taken. For example, taking the action of up at [0, 5] would transition to [0, 4].

Goal Test, is to determine whether the current location within the maze holds the food.

With the understanding and knowledge of the search problem, we can now go into how BFS solve this problem. First, we have to understand a vital concept of a class name Node in order the grasp the idea of how our algorithm is able to understand the problem and solve it. The Node class contains few properties, most notably its ID which is used to identify the Node. The state of it, which represent the coordinate of that Node within the maze, the children of it, which is an array of IDs and the parent of it, which should just be a single ID. Apart from that, as mentioned above, in every state/coordinate within the maze, only 4 actions can be taken at most, which is to expand/move in the direction of North, South, West and East. Therefore, within the Node class, there also exist a property named actions, containing a predefined array of these 4 actions.

Our algorithm starts off by accessing the food location from the problem dictionary which is passed on from the frontend. This food location is then made as our goal state whereby the BFS is supposed to find this state within the maze and direct the snake towards. The snake location which is again passed on from the frontend as a problem dictionary is use as our initial state within the maze. This same value is used to create the first node using the aforementioned Node class. With this, we initialize our first Node/location within the maze.

Using this first Node of state [0,5], it is expanded whereby it perceives its environment and return an array of Nodes which are its children. This is done through a function that removes unavailable actions to the current Node when it takes one of the 4 predefined actions. An action is deemed unavailable when it would result in the snake hitting the wall or biting itself and hence removed from the array of actions. Using this same Node as an example,

Going North, would result in the snake going upwards to [0, 4]. Which is a valid square.

Going South, would result in the snake going downwards to [0, 6]. Which is a valid square.

Going West, would result in the snake going left to what would be [-1, 5], which is not a possible square.

Going East, would result in the snake going right to [1, 5]. Which is a valid square.

From the example above, the first Node of state [0, 5], would have south(s) removed from its actions array leaving only 3 actions, north(n), west(w) and east(e), within the actions array which was once 4. Using the actions left

within the actions array, 3 children Nodes are now generated and stored in an array in the order of them being generated. This is something worth noting as it is vital to obtain the solution later on.

The first Node of state [0,5] is now appended into an explored array and deleted from the frontier. The explored array here functions as a closed list to keep track of the Nodes that has been expanded to remove any future loopy paths.

Each of the newly generated children nodes will now be checked for redundancy and loopy paths. Each of these children nodes are compared within the nodes that are within the frontier array to check for redundancy and then compared within the explored array to check for loopy paths. The state property within each node is used as the basis of comparison and checking.

From there, if the newly generated node is found to be either redundant or loopy, it is removed. If it is proven to be an entirely new node that has never been generated before, the node will be checked whether it is the goal node using the state as the basis again. If it is an entirely new node but not the goal, it will be appended to the frontier array.

This process is then repeated by expanding the first element in the frontier array until the goal is found. It is due this nature that take the first element of the frontier array, expands it, removes it from the frontier array and adding new nodes to end of the frontier array that makes every cycle of the process unique. No nodes will be expanded twice, and the nodes are selected to be expanded in a fixed manner. Our algorithm demonstrates the concept of First In, First Out (FIFO) which is also the core idea of BFS, making it a BFS algorithm.

Up till this point, our algorithm is able to take on the first 2 challenges that it aims to tackle. However, with the third challenge that may have more than 1 food in the maze, adaptations have to be done. We overcame this problem by constantly accessing the first food within the problem dictionary that is pass on from the frontend. This allows our algorithm to fixated upon a single goal state instead of being confused by multiple goal states.

Once the goal node has been found, the algorithm would have to backtrack to find the actions taken from the initial state leading to the goal state. This is done through searching for the parent node of the goal node within the explored array using ID as an identifier.

Once the parent is found, the index of the goal node within children property, which is an array, of the parent is identified. (This is done because if the parent node is the parent of the goal, hence, the goal is definitely one of the children of the parent.) Now that the index has been identified, the same index within the actions property array of the parent node is inserted into a solutions array. (This can be done because child nodes are generated in the order of actions within the array). The goal is now then set to be the parent and the process repeats back to the initial state which does not have a parent node. The pseudocode below might provide a clearer picture.

```
Loop
        Find for parent of goal in explored
        index = parent.children.index(goal.id)
        solution.insert(0, parent.actions[index]
        goal = parent

End if goal does not have parent
```

2.2 BFS Flowchart

Below is a simplified flowchart showing the general idea and sequence of the algorithm.

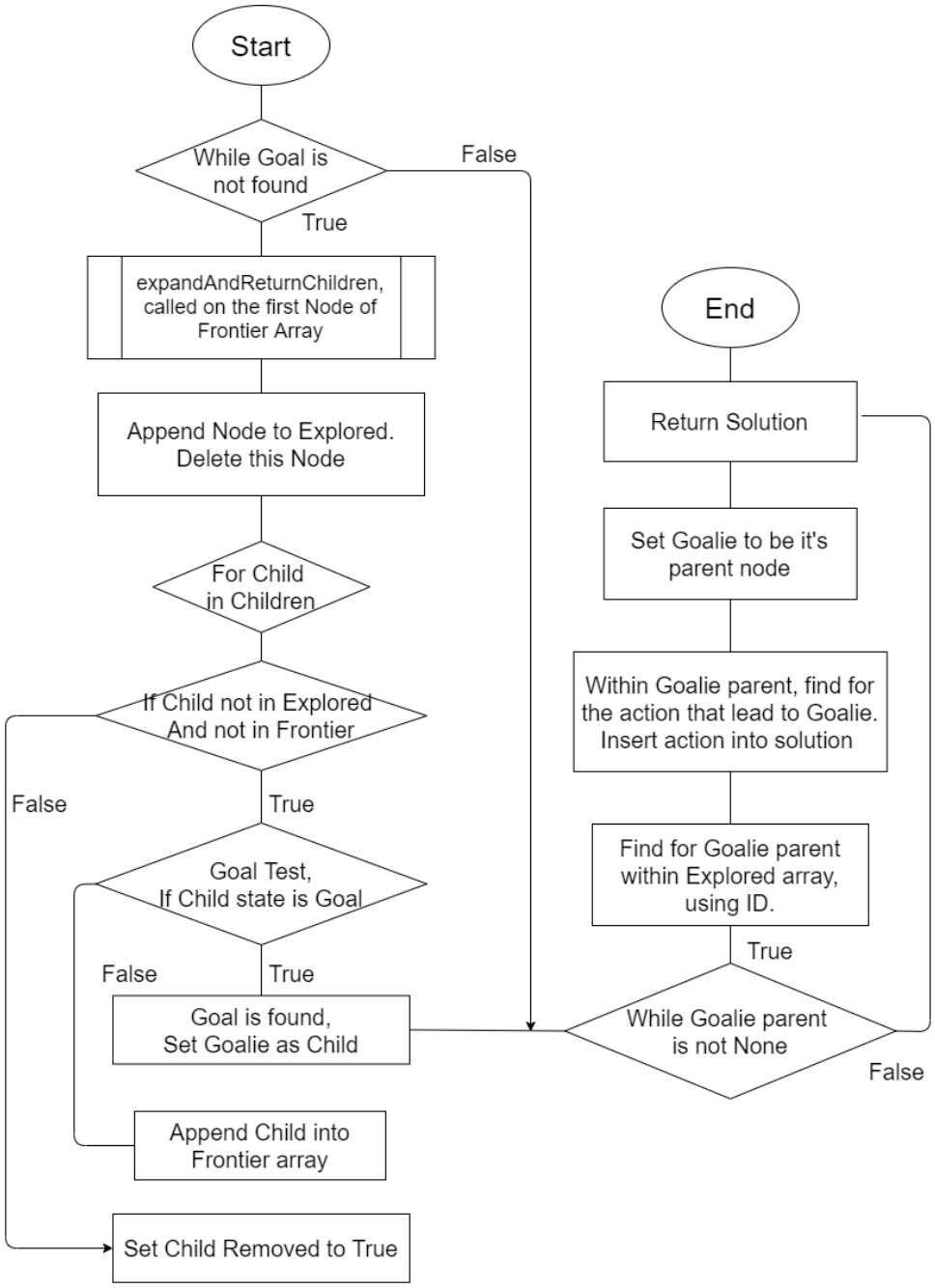


Figure 1: BFS Flowchart

2.3 BFS Justification

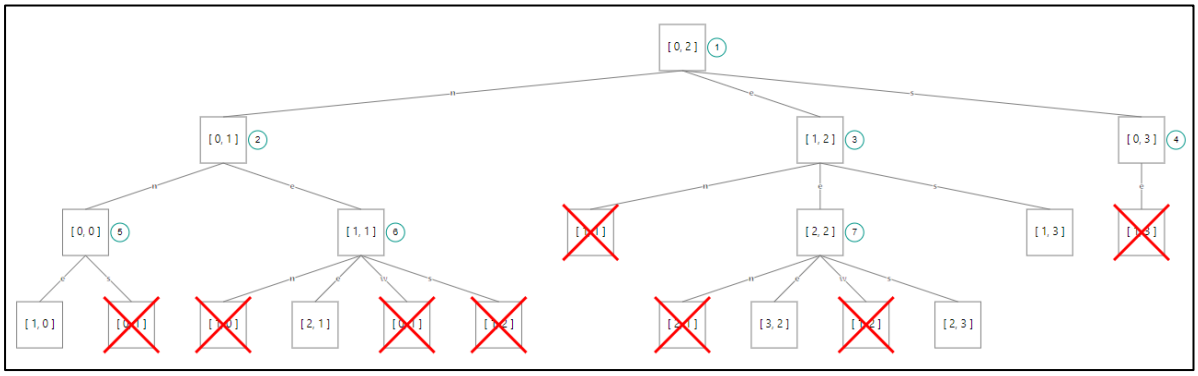


Figure 2: Sample BFS Search Tree

Figure 2 above is a BFS search tree generated from the algorithm. This sample is done on a 4×4 maze with the initial state of [0, 2] and goal state of [2, 3]. As can be seen from the search tree and its expansion sequence, our algorithm is a BFS algorithm that expands horizontally across the same depth before going into the next depth.

The reason that BFS was chosen as the best algorithm for uninformed search is due to its nature of expansion. The chances to be expanded for each node is relatively more even compared to Depth-First Search, allowing it to the goal node with a lower search cost. The figure below roughly illustrates the expansion pattern of BFS.

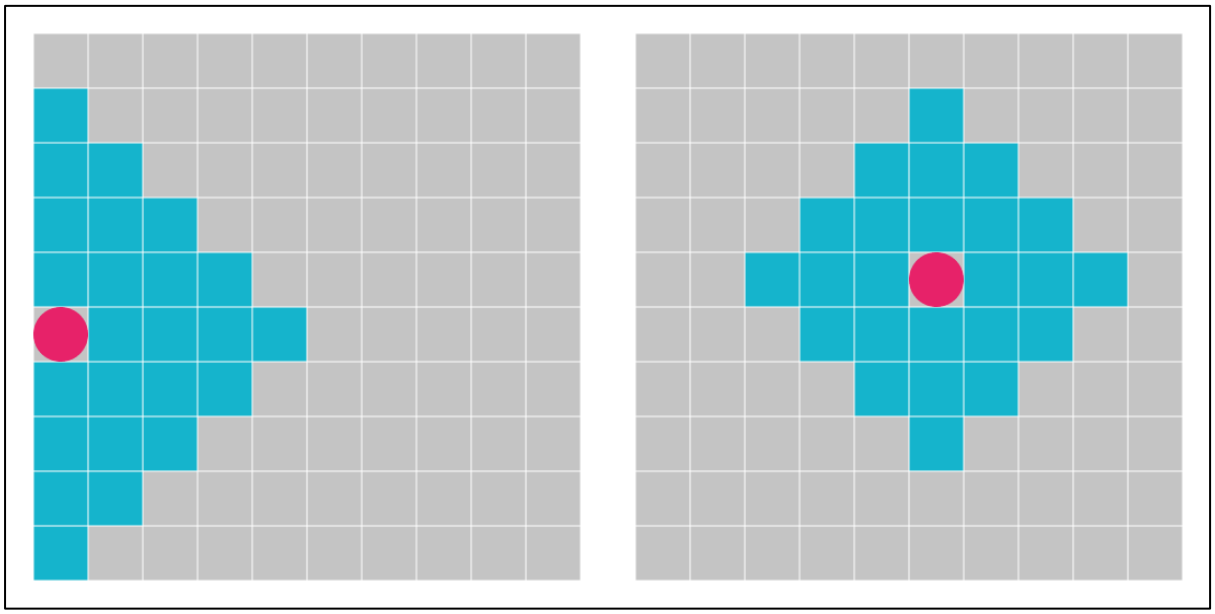


Figure 3: BFS Expansion Pattern

The red dot in the figure above shows the position of the snake whereas the blue squares indicate the expansion pattern of BFS. It is precisely this expansion pattern that allows the snake to explore the maze in an unbiased way, finding its way to the goal regardless of where the goal is. Although one may argue that DFS would be better if the goal happens to be aligned with the first node expanded, however over the course of a game, BFS would still be the superior search algorithm, incurring lesser search cost overall.

3.0 Greedy Best-First Search

This section contains 3 subsection which focuses on the implementation of Greedy Search, followed by the flowchart of the algorithm and concluding with the justification of Greedy Search.

3.1 Greedy Search Implementation

The problem definition and description for this search is identical to the one of BFS as the challenges that are to be tackled is identical. As such, the initial state, actions, transition model and goal test are the same as BFS. However, due to Greedy Search being and informed search, there ought to more knowledge fed to the agent (snake) that is beyond the problem definition. Therefore, a heuristic function is required to estimate the distance between the goal and the current location of the snake within the maze. This extra knowledge is then fed to the Greedy Search algorithm to take the closest path to the goal.

The heuristic has to be applied to every node that is generated and therefore is saved into the Node class as a property. This Node class is identical to the one in BFS only now with the addition of the heuristic property, which saves the distance between the current node and the goal node.

The heuristic of our algorithm is calculated by adding the differences between the x-coordinate and y-coordinate of the goal and the current location. The formula is as shown below:

$$Heuristics = |x\ of\ goal - \ x\ of\ current| + |y\ of\ goal - y\ of\ current|$$

The general flow of the algorithm is as follows,

1. Retrieves goal and current location from the frontend. Create first Node.
2. Goal test before expansion. If goal, stops algorithm.
3. If not goal, expand the first node of the frontier array.
4. The node is expanded, returning an array of children nodes. Same node is appended to explored array and removed from frontier.
5. Each child node in the children array has its heuristics calculated and stored within its heuristic property if it is not a redundant nor a loopy path. The child is then appended into the frontier array.
6. The frontier array is sorted in ascending based on the heuristic property of each node within the array.
7. Repeat steps 2-6 until the goal is found.

The details of how each step works out can be found in Section 2.1 BFS implementation of this report. The main characteristics of an informed search and Greedy Search is displayed in Steps 2, 5 and 6. Step 2 runs a goal test before expansion which is a characteristic of informed search whereas Step 5 displays the usage of heuristics within this algorithm. Step 6 reorders the frontier array so that the closest node to the goal is expanded first which is the core idea of Greedy Search. The algorithm also sets the goal to be the first food from the problem dictionary that is passed on from the frontend to allow it to complete all 3 challenges. The effects of doing so for both BFS and Greedy Search will be further discussed in Section 4.

Once the goal has been found, the steps taken to return an array of actions for the snake to move towards the goal is as follows,

1. Goes through the elements within the explored array.
2. Find for a match in ID property between the nodes in explored array and goalie's parent.
3. Once matched, find for the action within the parent node that led to the generation of the child(goalie).
4. Insert this action into the solution array.
5. Set goalie now to be parent. (Goalie is now the parent of the initial goal)
6. Repeat steps 1-5 until the current node has no parent.

However, due to the characteristic of our Greedy Search algorithm, under certain specific conditions the search cost might be slightly higher than expected. An example is shown in the Figure below with the conditions of Challenge 2.

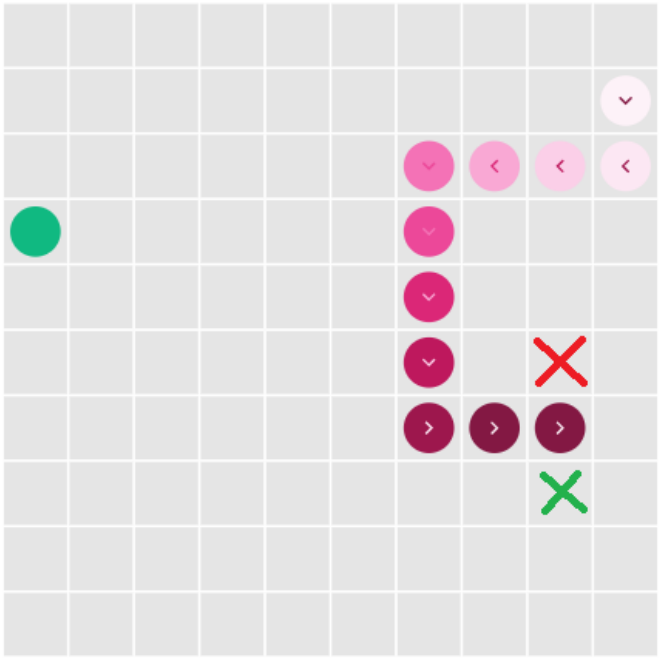


Figure 4: Greedy Search

The snake head is currently at [8, 6] whereas the goal/food is at [0, 3]. The snake produces three children nodes, [8, 5] (red cross), [9, 5] and [8, 7] (green cross). Based on the heuristics formula, the heuristics of the red cross would be 10 whereas the heuristics would be 12 for the green cross. Therefore, our algorithm would expand the node with the red cross (future reference as red node) before expanding the node with the green cross (future reference as green node), resulting in higher search cost. The red node would be expanded to a certain depth until the algorithm figures that this closest path to the goal that it is taking would result in it either being trapped by itself or biting itself making it an invalid solution before going on to take the path starting with the green node. Ultimately, the algorithm is still able to find the path to goal albeit incurring a slightly higher search cost.

3.2 Greedy Search Flowchart

Below is the flowchart of the Greedy Search Algorithm.

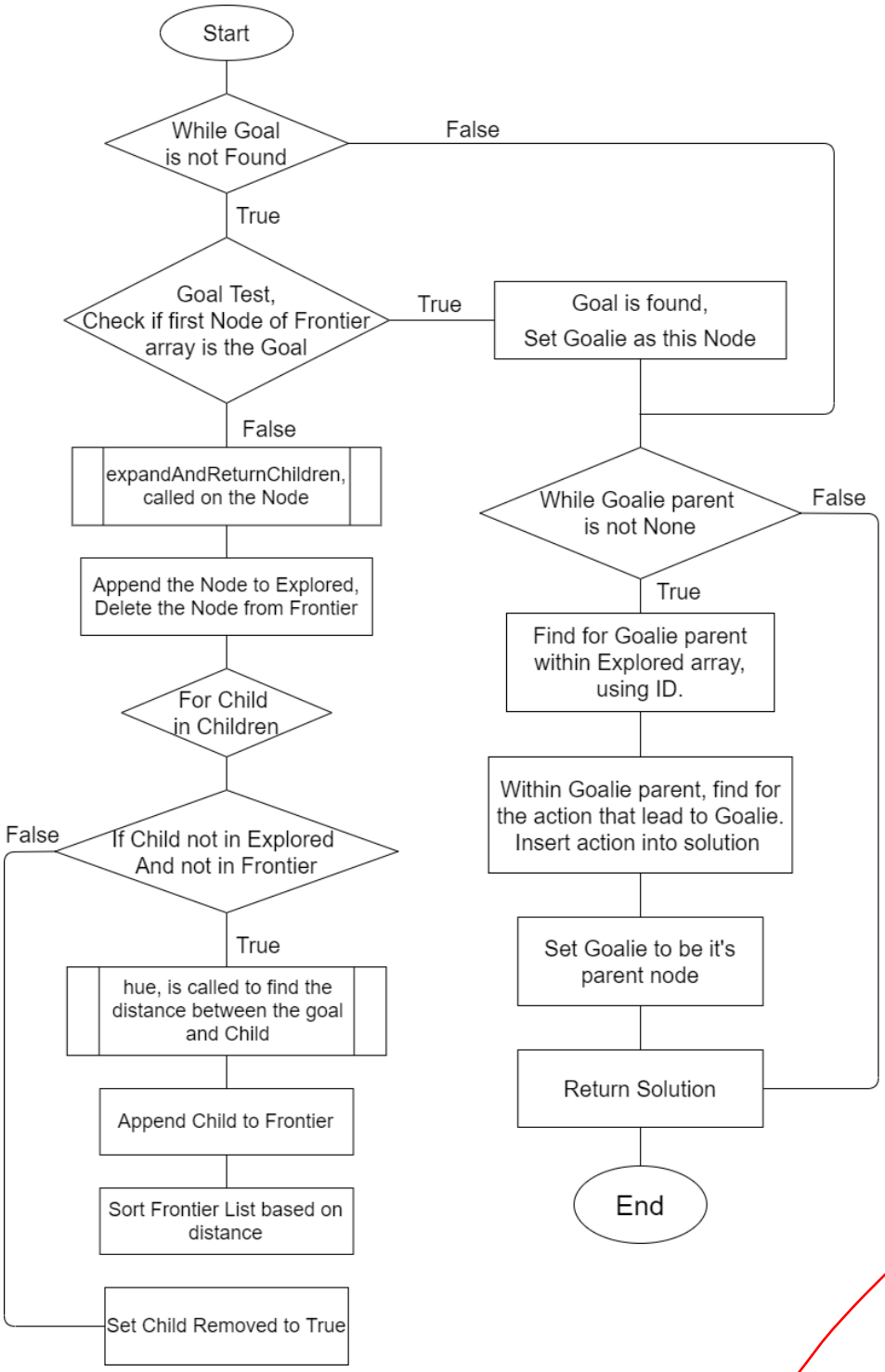


Figure 5: Greedy Search Flowchart

3.3 Greedy Search Justification

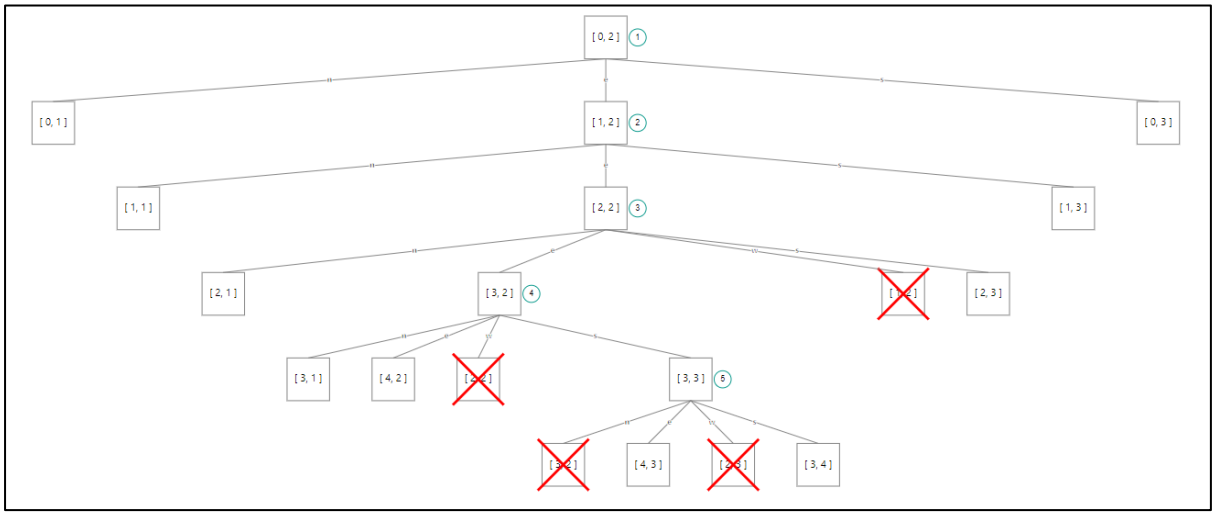


Figure 6: Sample Greedy Search Tree

Figure 6 above is a Greedy search tree generated from the algorithm. This sample is done on a 5×5 maze with the initial state of [0, 2] and goal state of [4, 3]. As can be seen from the search tree and its expansion sequence, our algorithm is indeed a Greedy Search algorithm whereby only the path to the node is expanded. The heuristics has also been proven to work function correctly. The heuristics of the initial state would be 5 and it took 5 steps/expansion to arrive at the goal.

Greedy Search is chosen to be informed search to tackle the 3 challenges because Greedy Search only takes into account of the heuristic function whereas A* search takes in both the heuristic function and the known cost. Although intuition may tell us that A* is the better algorithm to be used since it takes into account more parameters thus producing a more accurate result. However, for the application of this snake game, the cost of moving from one node to another is a constant value of a single action taken. Therefore, taking into account both heuristic function and the known cost can be redundant at the same time drastically increasing the complexity of the algorithm without significant improvement to the snake's overall performance.

4.0 Results and Discussion

This section will cover the results of the snake traversing using the algorithms mentioned above and discuss on the performance of the algorithms individually and comparatively as well as the optimality of the algorithms. Each algorithm was developed with the purpose of solving the 3 challenges as mentioned in the Introduction section of this report, therefore the algorithms will be tested based on the challenges.

Each algorithm is tested 50 times in each of the challenge conditions. Overall, 300 tests were carried out and its performance is evaluated. 3 performance measure were introduced to evaluate the performance of the algorithm. These performance measures are the average score achieved, the maximum score obtained and the minimum score obtained.

A higher value of average score would indicate that the algorithm, under that challenge condition, is able to solve a similar problem of path finding more. On the other hand, the value of max and min score indicates that the range of results that the algorithm may produce. Therefore, average score would be the better indicator in terms of evaluating the performance of the algorithm. This is because max and min score are isolated instances that may be heavily influence by external factors such as the food location that is randomly generated.

The table below shows the average score (rounded down), max and min score over 50 trials for each challenge.

	U1	U2	U3	I1	I2	I3
Average	>50	25	24	>50	22	22
Max	∞	40	40	∞	34	43
Min	∞	15	15	∞	13	11

U refers Uninformed Search where I referred to Informed Search. The numbers 1, 2 and 3 corresponds to each challenge condition.

For challenge 1, each algorithm is run till the snake achieves more than 50 points and then manually stopped by the tester. This is because, due to challenge 1 being a fixed length snake, ideally it is able to go on forever. Hence, the maximum and minimum for U1 and I1 is assumed to be infinity as the algorithm is displaying an ideal scenario where it can run forever.

The remaining test for uninformed search, in our case using BFS, yielded an average score of 25 for U2 and 24.68 for U3. Both achieving the highest score of 40 once, out of 50 tests for each challenge condition. On the other hand, challenge 2 and 3 for informed search produced an average of 22.82 and 22.88, respectively. A high score of 43 was achieved in one of the tests for I3. The min score achieved by both algorithms in challenge 2 and 3 exceeds 10, which satisfy the requirements to complete the challenge.

It is worth noting that in all the test cases for U2, U3, I2 and I3, the algorithm only stops functioning as it should when the snake length gets long enough. This is because the snake body ‘divides’ the maze into 2 parts and the food is generated on the other part of the maze where the snake head is not at. This results in the algorithm not being able to find a valid path to the food as the understands that it would be biting itself.

This is largely due to the fact that our algorithm does not update the position of the snake as it explores the maze, therefore leaving a shadow of the snake’s tail even if the snake has moved resulting in a false positive that the path to the goal will result in the snake biting itself. Hence not being able to find a solution to the problem. There are multiple approaches to handle this issue, however, none of the following discuss is implemented in our algorithm.

One of the possible approaches to counter this issue is to have the snake’s tail as a temporary sub-goal until the maze is not ‘divided’ anymore. Once the maze is open again, the snake will have a clear path and solution to the goal. Besides that, the algorithm can also be programmed to traverse the outer edge of the maze until the maze is ‘undivided’, thus providing a solution once again. Both of these approaches use the concept of a sub-goal to allow the snake to be able to reach the intended goal/food.

Up till the point, the results displayed have only been instances of testing the algorithm as itself. The results strongly suggest that the algorithms that were developed, BFS and Greedy Search are complete algorithms to a certain extent and are able to tackle the 3 challenges. The following paragraphs will focus on the performance of the algorithms comparatively when presented with identical problems. The aim of this test is to understand the search cost incurred by each algorithm in solving the same problem.

The setting of the test are default settings, with a maze size of 10×10 and starting position of [0, 5] in the maze. The test was carried out in the following steps:

1. Run BFS at the starting position to the food. Record the coordinates of the food/goal, the number of generated nodes and expanded nodes.
2. Using the Greedy Search, reset the maze until the food is generated on the same coordinates as step 1.
3. Run Greedy Search. Record the number of generated nodes and expanded nodes.

* Note: Generated nodes here refers to the nodes generated regardless of it being redundant or loopy.

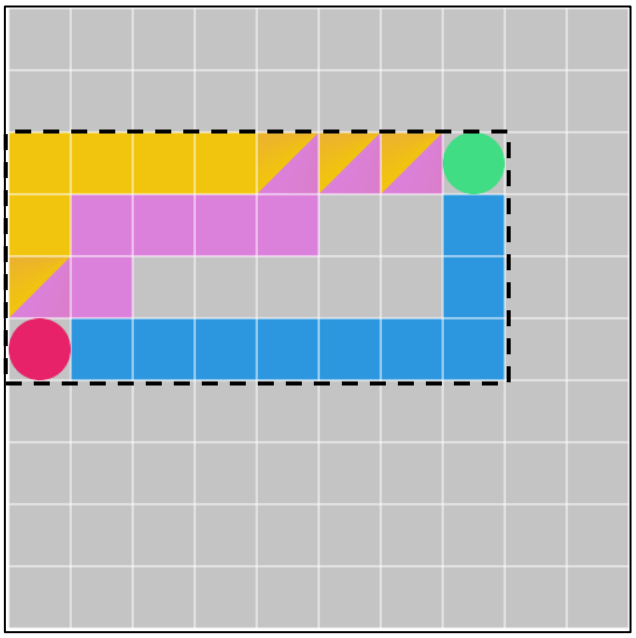
Due to the difficulty of having the second food generated on the same coordinates after the first food eaten, all tests were limited travelling from the initial position the first food generated within the maze. It is also due to the same reason; the sample size is only limited to 7. The results are shown in the table below.

		BFS		Greedy Search	
Test No.	**Food Location**	**Generated**	**Expanded**	**Generated**	**Expanded**
1	(0,3)	6	2	6	2
2	(1,6)	9	3	7	2
3	(0,8)	29	9	9	3
4	(2,3)	36	11	13	4
5	(4,3)	102	29	21	6
6	(4,7)	114	32	23	6
7	(8,5)	186	51	31	8
Average	-	68.86	19.57	15.71	4.43

Table is ordered in relative distance between initial state and food

As can be seen in the table above, although both algorithms are complete and is able to find a solution to the same problem, it is evident that the search cost incurred by Greedy Search is significantly lower than the BFS. BFS requires more than 4 times of the number of generated nodes and the number of expanded nodes to reach the goal compared to Greedy Search. However, it is also observed from the table above that when the food/goal is closer to the initial state, for example in test 1 and test 2, the difference in search cost between both algorithms are not apparent.

Lastly, the optimality of both algorithms is to be discussed and evaluated. Although no test was done to evaluate the performance of the algorithms in this aspect but it is suggested to be optimal in both algorithms, BFS and Greedy Search alike. This is owing to the fact that the cost of moving from one square to another is a non-increasing fixed cost and the maze size is a symmetrical state space. The illustration below will demonstrate the optimality.



The figure shows the snake (pink circle) at the initial state of [0, 5] and the food/goal (green circle) at [7, 2]. With both of these coordinates, a rectangle (dotted black lines) can be constructed with the snake and goal being diagonal vertices. It can be said that as long as the snake travels directly to the food within the rectangle, it is an optimal solution. This idea is further strengthened by the suggested yellow, purple and blue paths all taking 10 steps to reach the goal. There are many other paths that the snake can take within the rectangle, going directly to the food and yet the steps taken will also be 10, producing an optimal solution.

Therefore, it can be concluded that any algorithm, not limited to BFS and Greedy Search as it is in this report, that travels directly towards the goal within an imaginary rectangle is an optimal algorithm.

The effects, if any, of constantly accessing the first food from the food locations array for challenge 3 on any of the aforementioned performance measure should also be clarified. Although, it is possible for the snake to set the food that is further away as the goal but we believe that the effects of doing so on the performance of the snake to obtain the food is mitigated to minimum due to sample size of the test cases. This is supported by the scores between Challenge 2 and 3 being very close in both algorithms. In line with that, we also believe that the effects on the optimality of algorithm is also at a minimum. This is because although the overall optimality of the algorithm may be reduced but the solution provided by the algorithm going from the current snake location to the food it has set as its goal is still optimal. However, the search cost or space complexity incurred by both algorithms may be slightly higher than in an ideal scenario, as the snake may travel to the food further away first, before turning around for the other food thus generating and expanding more nodes over the course of eating 2 foods.

5.0 Conclusion

Our algorithms, BFS and Greedy Search, created using Python programming language is able to solve the problems and the 3 challenges that was posed in a snake game. The algorithms are also complete to a certain extent providing optimal solution to all problems it is able to solve. However, there are still shortcomings in our algorithms that either have not been identified at this point or known flaws that have yet to be solved. Some of the findings through the test of the algorithms is that although informed search is expected to generate and expand lesser nodes on its way to the goal compared to uninformed search, the difference between the 2 algorithms is not significant when the initial state and goal is closer. Future efforts can look into the direction of eating more than 1 food in a single path in the scenario of multiple foods within the maze.