SCHOOL OF SCIENCE AND TECHNOLOGY

COURSEWORK FOR THE BSC (HONS) COMPUTER SCIENCE; YEAR 3

CSC3206: ARTIFICIAL INTELLIGENCE

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

NO.	STUDENT ID	STUDENT NAME
1	18011551	PHYLLIS CHONG YEE TENG
2	18032052	KHIRRTINI MURALEETHARAN
3	18045872	KHAIRUL AKMAL BIN KHAIRULANWAR

INSTRUCTIONS TO CANDIDATES

- This assignment will contribute 15% to your final grade.
- This is a group assignment.

Academic Honesty Acknowledgement

"We, **PHYLLIS CHONG YEE TENG [18011551], KHIRRTINI MURALEETHARAN [18032052], and KHAIRUL AKMAL BIN KHAIRULANWAR [18035872],** verify that this paper contains entirely our own work. We have not consulted with any outside person or materials other than what was specified (an interviewee, for example) in the assignment or the syllabus requirements. Further, we have not copied or inadvertently copied ideas, sentences, or paragraphs from another student. We realize the penalties *(refer to the student handbook and undergraduate programme handbook)* for any kind of copying or collaboration on any assignment."

[illegible] [illegible] [illegible]

21ST MAY 2021

Contents

Introduction

In today's games, pathfinding algorithms are one of the many crucial key to creating an artificially intelligent bots to go against human players. Thus, pathfinding algorithms are commonly used in solving mazes. To solve the pathfinding problem, the simplest example can be seen in a snake game.

A classic snake game is a game where one is required to navigate around the maze to find the shortest path to the food. Each food that the snake managed to eat will earn the player a point. The key point of the game is to earn as many points as possible before dying. The following are the conditions ~~to where~~ in which a snake can die:

- Running into a wall
- Running into itself

As the snake game can generate random obstacles over time with its growing length, it is crucial for the pathfinding algorithm to constantly update the obstacles after every food is consumed. Thus, pathfinding algorithms, which is also known as search algorithm are needed to solve the classic snake game problem. In search algorithms, it is categorised into two major parts – uninformed and informed search algorithms. Examples of uninformed search algorithms are uniform-cost search, breadth-first search (BFS) and depth-first search (DFS) whereas examples of informed search algorithms are greedy best-first search and A* search algorithms.

As different algorithms has different complexity, efficiency and performance; choosing the algorithms with priority on the said attributes is important. This can also be observed from the difference between uninformed and informed search algorithm. Compared to uninformed search algorithm, the informed search algorithms also calculates the heuristic values of a node to the goal state to determine the best possible route to take.

For this project, the efficiency of the pathfinding is taken into highest priority when choosing which algorithm to implement. Thus, BFS algorithm and A* search algorithm is chosen for uninformed and informed search algorithms respectively. The main objective of this project is to investigate the efficiency between BFS and A* search algorithm,

Problem

The project requires to create two agents, specifically performing uninformed and informed search algorithms to play the game automatically. The game given here is the class snake game where the agent will pose as the snake that constantly eats food at a given location. For each food that it consumes, one point is rewarded. However, there are conditions where the game will be over such as running into itself and running into the wall of the maze.

To start off by creating this agent, a couple of problems are faced but pathfinding will be the main issue. When the starting location of the snake is determined, there are various ways that the snake can reach its target – the food. This is to be solved by getting a sequence of actions that will guide the snake to reach its goal. In other words, this can be solved by using various search algorithms out there. The challenge here is to determine and implement the better algorithm in terms of performance and efficiency to ensure that an optimal pathfinding algorithm is used for the snake to navigate around.

Besides that, there are also a few issues faced along the project. The following are some problems encountered:

1. Studying and understanding the various search algorithms available. As there are many different algorithms known in both informed and uninformed search algorithms, it is crucial to understand them in depth before attempting to implement it.
2. Decide a suitable algorithm each for uninformed and informed search algorithms. This is to ensure that both algorithms that are chosen has the best efficiency among the algorithms that were studied.
3. Implementation of the finalised algorithms. This is another issue that slowed down the progress of the project. Refer to no. 4 for communications explanation.
4. Communications with group members. The COVID-19 pandemic has made it difficult for any means for face-to-face discussion. This makes it difficult for one to make a point to another individual. Although this issue can be easily overcome by using online communication platforms, discussing face-to-face is still the most efficient way.

Algorithms and Application

Breadth First Search Algorithm

The Breadth First Search (BFS) algorithm is an uninformed algorithm that searches by traversing from its starting node to its neighbour nodes. As the name itself has suggested, BFS explores the nodes breadthwise before moving on to the next layer.

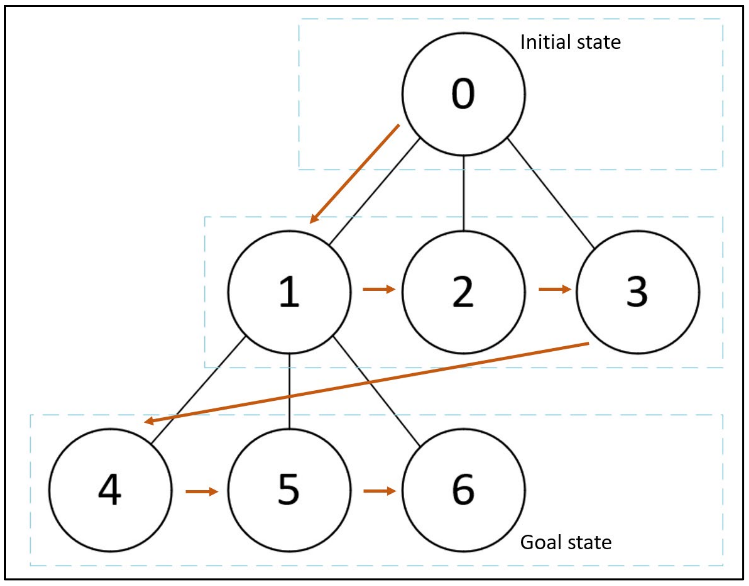


Figure 1: Example of BFS

The figure shown above is an illustration on how BFS algorithms works to search its goal state from its starting node (also known as initial state). In other words, all nodes in a layer must be explored before moving on to the next layer in BFS algorithm.

When compared to Depth First Search (DFS) algorithm, BFS algorithm uses lesser moves in total to reach its goal state. This is because DFS algorithm focuses on depth wise search first where if there is an infinite depth in the problem, it is less likely to be solved or reach the goal state. Using BFS algorithm ensure that every node in a layer are explored before searching in deeper into the search tree. This ensures that no nodes are missed out during the search. Thus, BFS algorithm is selected to be implemented as a representative for the uninformed agent.

A* Search Algorithm

The A* algorithm is an informed searching algorithm that searches for the shortest path between the initial state and goal state. The A* algorithm uses a heuristic into regular search algorithms. This helps the algorithm to plan ahead at each step so a more optimal decision is made. The key difference of the A* algorithm when compared to the uninformed searches, like BFS and DFS, is that for each node, A* algorithm uses a function $f(n)$ that gives an estimated total cost of path using that node. The A* algorithm expands paths that are already less expensive by using this function:

$$f(n) = g(n) + h(n)$$ ✓

Where,

- $f(n)$ = total estimates cost of path through node n
- $g(n)$ = cost so far to reach node n
- $h(n)$ = estimated cost from n to goal.

One of the methods to calculate the $h(n)$ is by using the Manhattan Distance. The Manhattan Distance is the sum of absolute values of differences in the goal's x and y coordinates with the respective x and y coordinates of the node.

$$h = | x_{start} - x_{destination}| + | y_{start} - y_{destination} |$$ ✓

This heuristic is only used when the program is allowed to move only in four directions in a straight line: north, east, south, or west.

The A* Algorithm is more optimal than using the Greedy BFS Algorithm. This is because the Greedy BFS Algorithm expands the nodes using only the estimated distance to the goal, $h(n)$, and first expands the smallest. It does not keep into consideration of the past knowledge, $g(n)$, which is why the algorithm is considered 'greedy'. Therefore, the path that is found by the Greedy BFS Algorithm may not be the optimal path.

Code Explanation

`class Player`

The **`Player`** class contains the information about the algorithm. The **`informed`** variable informs the program of whether the algorithm uses an informed search or uninformed search. The **`constructor`** takes in the setup parameter, which contains the maze size information and whether the snake length is static or not. This class also contains the **`run`** function that takes in the snake location, the direction currently faced by the snake, and the location of the foods. The method calls the respective algorithm (BFS or A*) and returns the solution for the game along with its search tree.

In the BFS algorithm, the codes has the **`informed`** variable set to `False` as it is an uninformed search algorithm. In the A* algorithm, the codes has the **`informed`** variable set to `True` as it is an informed search algorithm.

`class Node`

The **`Node`** class consist of the information of each node in the maze. The constructor takes in the parameters where the self in this case, defines the properties of a node object. The **`self.children`** variable is a list that stores the children nodes of the current node object. The **`addChildren`** function is used to update the children of the nodes (which is into the **`self.children`** list).

In A* search algorithm, there are additional parameters and variables in the Node constructor. It is added for the use of heuristic function which will be described later.

`class searchTreeNode`

The **`searchTreeNode`** class is created to generate and update the search tree for the respective search algorithms. The constructor takes in the parameters of self, node, whether or not the node was removed, and the expansion sequence of the node, which is set to -1.

The **`updateTree`** function is used to update the children nodes in the search tree It contains a dictionary of information regarding to the node, which are the id, the state (the co-ordinates), the expansion sequence number, the children, the action (the directions), and whether or not it is removed. The children nodes will be linked to their parent nodes if the parent node exists. This class is the same for both BFS and A*.

BFS Algorithm

`def expandAndReturnChildren`

This function expands the children of a given node based on four directions – north, south, east, and west.

Using a temp list, -1 and 1 is used to determine the actions of a node. For example, given the x-coordinate of a node is x + i, the child node is facing east from the current node. The same procedure applies to the y-coordinates of the current node.

After that, the function will get the children node and its information and append it to the children list in the function. From there, the function returns the children list.

`def bfs`

This function is where the algorithms for BFS will take place entirely. It starts off with a variables declarations for the id and expansion sequence counters for the search tree; **`frontier`**, **`explored`**, **`solution`**, **`searchTree`**, **`tempSearchTree`** lists; **`isFound`** and **`foodLocation`**.

The function takes in the **`problem`** parameter which consist of the default location of the snake, the current direction it is facing and the food spawn location on the maze. Other than that, it also takes in the **`setup`** parameter in which holds the information regarding the size of the maze.

The frontier list is initialised with the starting node of the snake. Based on it, the algorithm expands to get the children of the node. It will then run through the nodes repeatedly with the while loop until the **`foodLocation`** is True (meaning the food is found).

Then, the algorithm checks for any repeated nodes in the search tree and marks it as removed to prevent unnecessary backtrackings and expand the nodes accordingly.

At the end of loop, if the **`foodLocation`** is found, the solution list, where the directions headed by the snake will be updated as well as the path list, which indicates the coordinates of each movement from the snake.

The function will then return the solution and **`searchTree`** list.

A* Algorithm

`def manhDistance`

This function contains the following formula to calculate the $g(n)$ and the $h(n)$

$$d = abs(toNode.x - fromNode.x) + abs(toNode.y - fromNode.y)$$

`def sorter`

The **`sorter`** function is used to sort the frontier list in ascending order of the **`fn`** value of the nodes. This is because the A* function iterates and expands on depending on the lowest **`fn`** value.

`def expandAndReturnChildren`

In A* search algorithm, the only difference is that when appending the child node information, the Manhattan Distance (**`manhDistance`**) of the nodes is called and calculated.

`def astar`

The structure of the **`astar`** function is similar to the **`bfs`** function stated above in the BFS Algorithm. The difference in the code block is that it calls the **`sorter`** function to sort the frontier list in ascending **`fn`** order before carrying out the process.

Result

The algorithm managed to fulfil the requirements of the assignment. The main requirement were to create one uninformed search algorithm agent and one informed search algorithm agent. The uninformed search algorithm chosen was Breadth First Search, and the informed search algorithm chosen was A* Algorithm.

There were three challenges proposed, and all three challenges were carried out. The challenge requirements were as follows:

1. One food at any time; Non-increasing snake length (Static); Snake length of one; Reaching at least 15 points.
2. One food at any time; Increasing snake length (Dynamic) with food; Starting snake length of one; Reaching at least 10 points.
3. Two food generated when no food is available on the maze; Increasing snake length (Dynamic) with food; Starting snake length of one; Reaching at least 10 points. ✓

Below are the proof and explanation of completing all three challenges for both, the uninformed BFS search algorithm and the informed A* search algorithm.

Uninformed Agent: Breadth First Search

To obtain a shorter and easier path for explanation, a 5 by 5 maze is used as an example instead of its default setting. The following shows the two solution examples for the given location of the snake and the food.

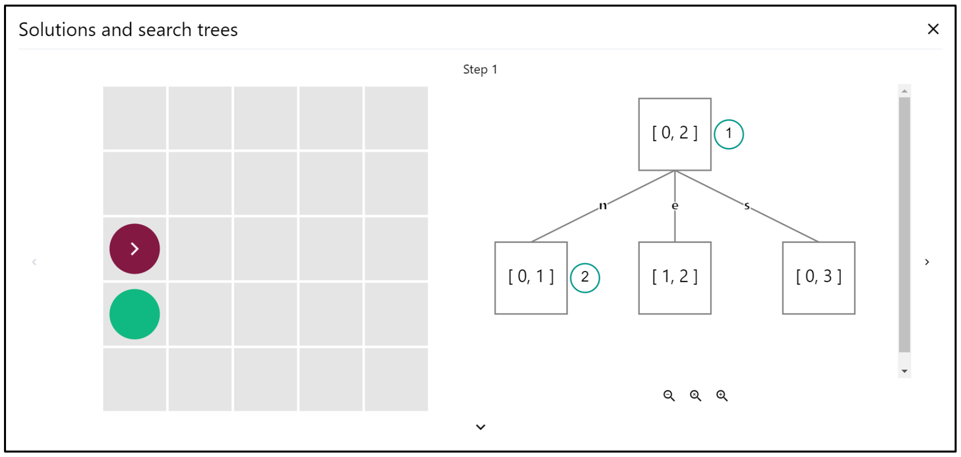


Figure 2: BFS Sample Solution 1

The setup information of from the interface of the game is first loaded by the constructor in the **`Player`** class. From there it loads the **`run`** function where the **`bfs`** function is called to start searching for an optimal path for the snake to reach its food. The **`bfs`** functions will call the **`expandAndReturnChildren`** function to locate the children of the current node. In the case of Figure 2, the current node location is [0,2] and its children are [0,1], [1,2] and [0,3]. Thus, the optimal solution is to just head south one step since the food is located right below the snake by one step.

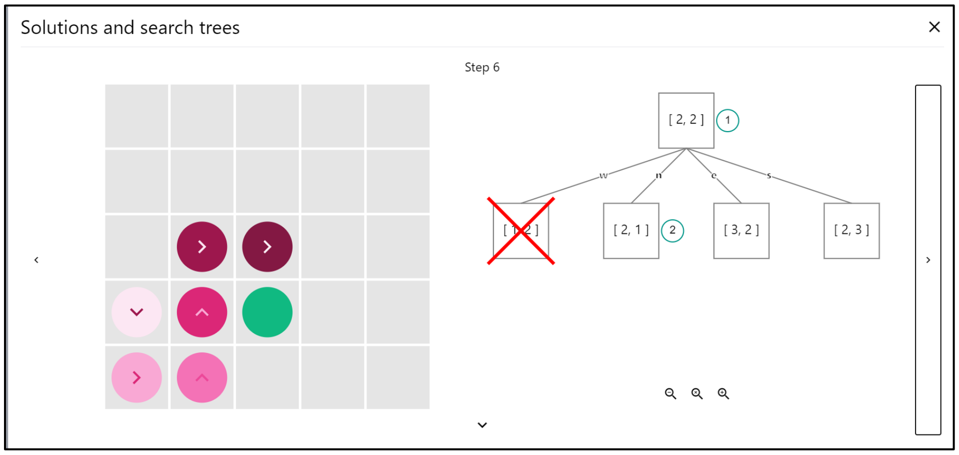


Figure 3: BFS Sample Solution 2

Figure 3 shows another example of the BFS algorithm. In this example, the nodes occupied by the snake (body and head) are removed. As [2,2] is the position of the snake head, the snake cannot move backwards to avoid running into itself. Thus, the node [1,2] is removed from the search tree so that it will not be considered as one of the available paths to reach the food.

Informed Agent: A* Search

Just like the Breadth First Search example, to obtain a shorter and easier path for explanation, a 5 by 5 maze is used as an example instead of its default setting. The following shows the two solution examples for the given location of the snake and the food.

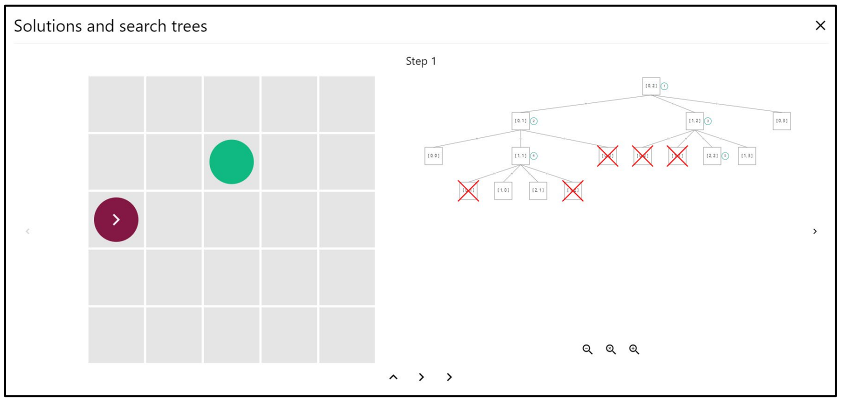


Figure 4: A* Sample Solution

The setup information of from the interface of the game is first loaded by the constructor in the **`Player`** class. From there it loads the **`run`** function where the **`astar`** function is called. The **`astar`** function first calls the **`sorter`** function to sort the frontier by the **`fn`** value. Then, the **`astar`** function will call the **`expandAndReturnChildren`** function to locate the children of the current node. The **`manhDistance`** function is called to calculate the heuristics of the nodes and find the optimal path. In the case of Figure 4, the current node location is [0,2] and its children are [0,1], [1,2] and [0,3]. The location of the food is at [2,1]. By calculating the f(n) value, which is g(n) + h(n), the tree expands first on the lowest f(n) values of the children. According to the tree, the optimal solution is getting to node [1,1]. From there, the food node is can be found.

Challenge #1

Using a 7 by 7 maze, the location of the snake and food are initialised. As per stated in the challenge, the length of the snake will be set to static. In other words, the snake will not grow after consuming the food. The algorithm has successfully completed the first challenge. The results can be seen in the figures shown below.

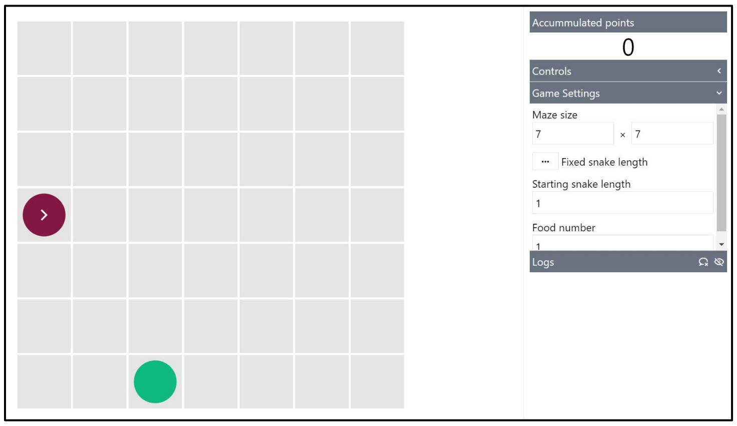
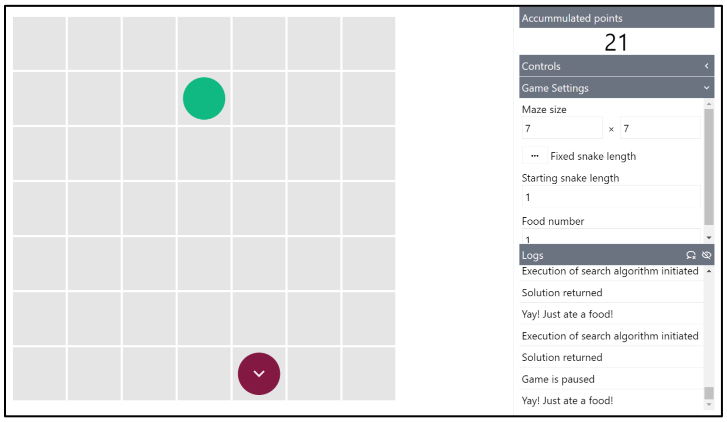

Accummulated points
0
Controls
Game Settings
Maze size
7 × 7
Fixed snake length
Starting snake length
1
Food number
1
Logs

Figure 5: BFS Initial positions of snake and food location


Accummulated points
21
Controls
Game Settings
Maze size
7 × 7
Fixed snake length
Starting snake length
1
Food number
1
Logs

```
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
Execution of search algorithm initiated
Solution returned
Game is paused
Yay! Just ate a food!
```

Figure 6: BFS Current positions of both snake and food location (on Pause)

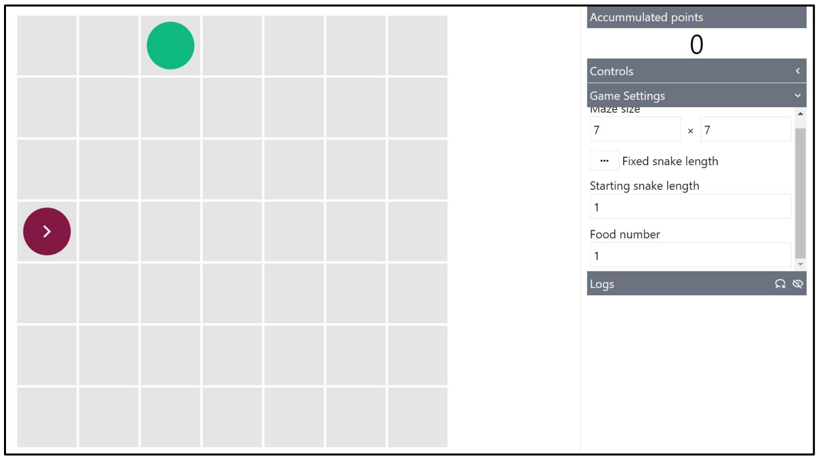
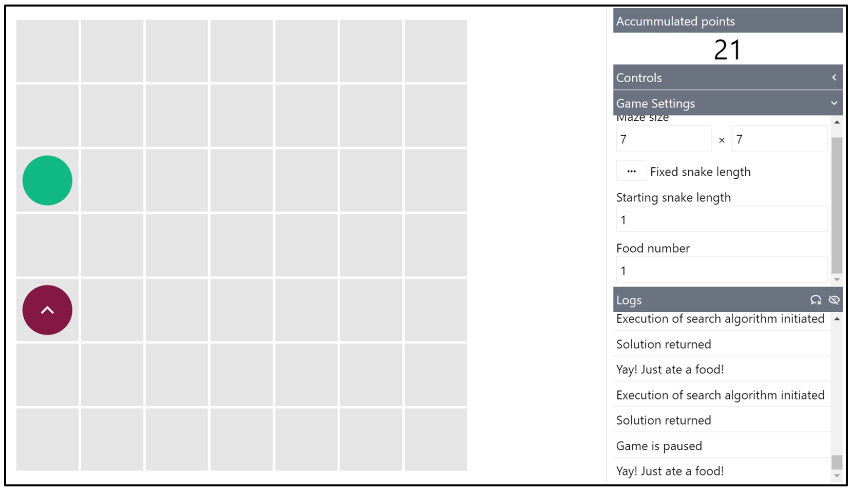


Figure 7: A* Initial positions of snake and food location



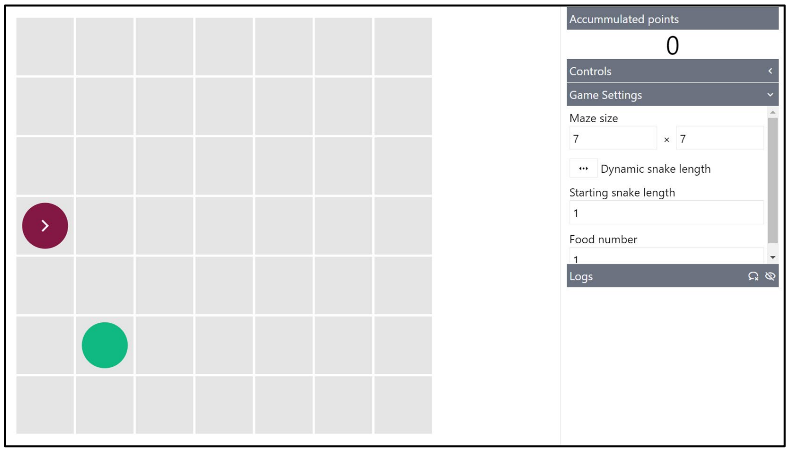
```
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
Execution of search algorithm initiated
Solution returned
Game is paused
Yay! Just ate a food!
```

Figure 8: A* Current positions of both snake and food location (on Pause)

As seen in Figure 6 and Figure 8, for BFS and A* respectively, the accumulated point collected by the snake is twenty-one points without dying. The current situation is put into paused mode to show that it has already made past the minimum requirements (fifteen points) for this challenge.

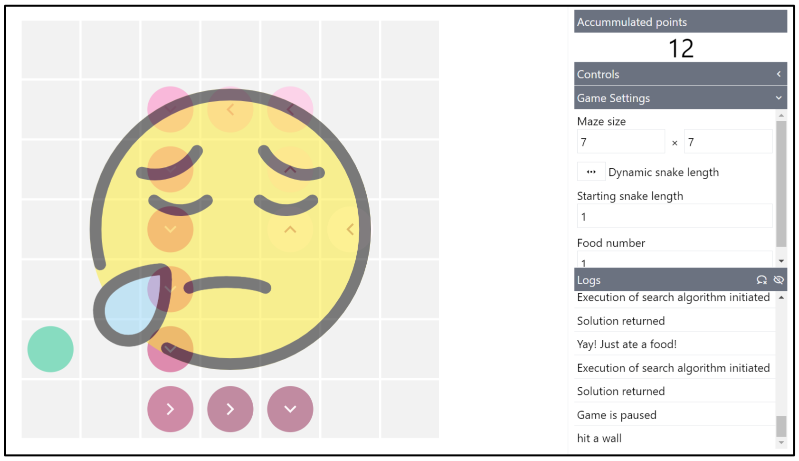
Challenge #2

Similar to Challenge #1, the setup for the maze size is the same. The only difference in the setup setting is the snake length. In this challenge, it has been changed to dynamic to allow the snake to grow in length upon eating the food. This sets the **static_snake_length** variable in the setup to False. The setting on the interface can be seen in the figures below.



Maze size
7 × 7
Dynamic snake length
Starting snake length
1
Food number
1

Figure 9: BFS Initial positions of both snake and food location



Maze size
7 × 7
Dynamic snake length
Starting snake length
1
Food number
1

```
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
Execution of search algorithm initiated
Solution returned
Game is paused
hit a wall
```

Figure 10: BFS Final positions of both snake and food location (upon Death)

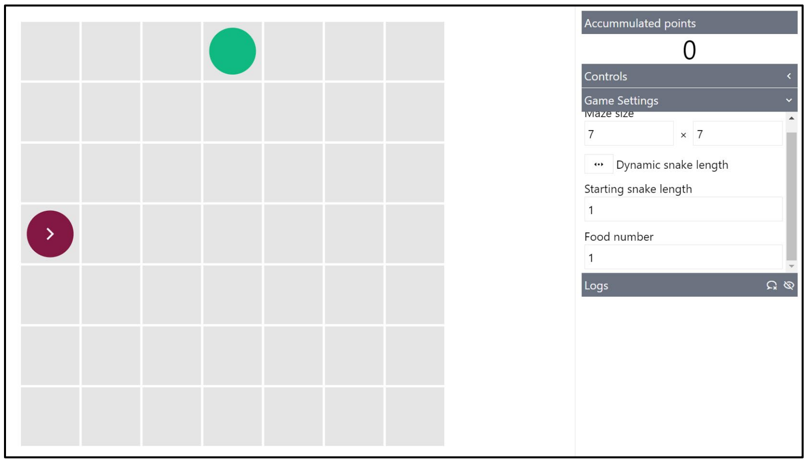
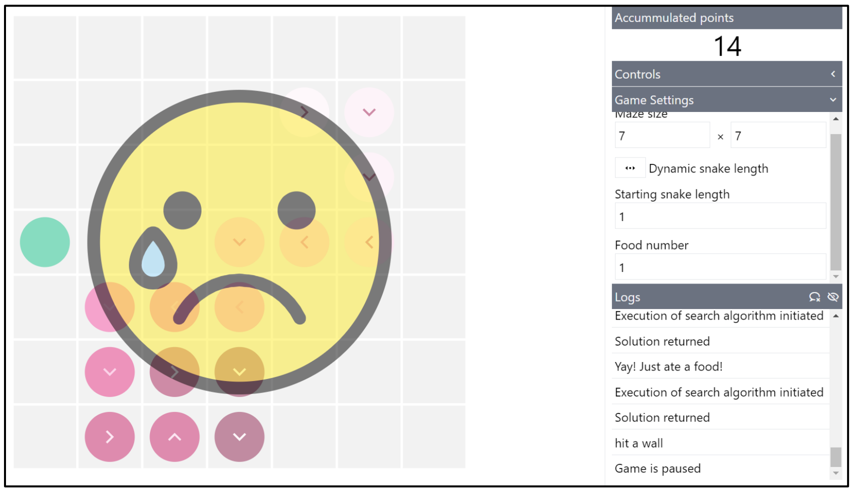


Figure 11: A* Initial positions of both snake and food location



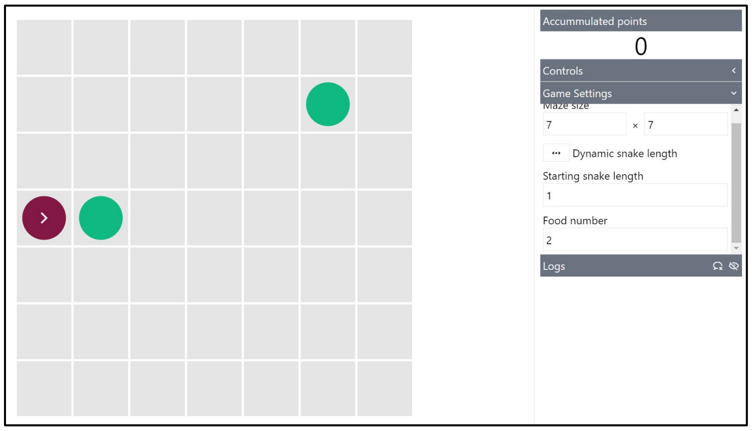
```
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
Execution of search algorithm initiated
Solution returned
hit a wall
Game is paused
```

Figure 12: A* Final positions of both snake and food location (upon Death)

As observed in the Figure 10, the snake is able to accumulate a total of twelve points before death. As observed in Figure 12, the snake is able to accumulate a total of fourteen points before death. Despite running into the wall or running into itself due to a less efficient pathfinding algorithm, the snake managed to achieve more than ten points with dynamic snake length activated. This completes Challenge #2.

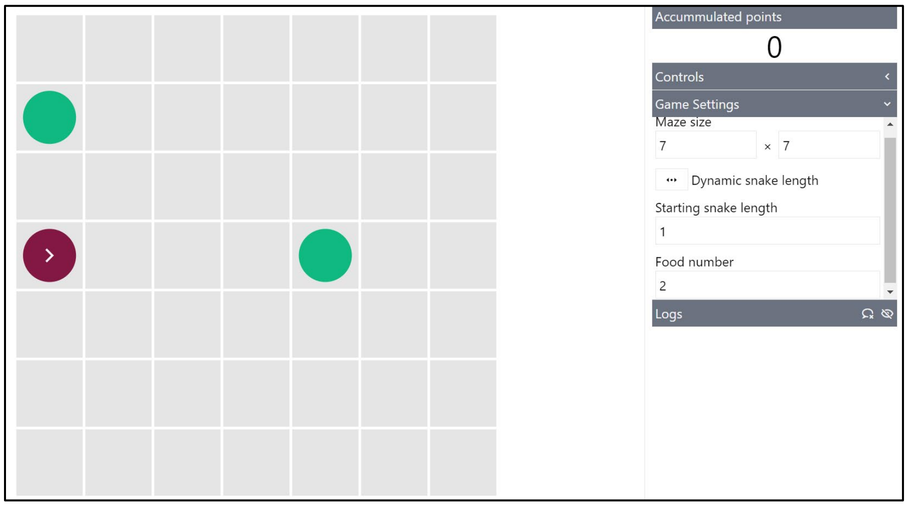
Challenge #3

For this challenge, the settings are also in correspondence with the setting in Challenge #2 except for the number of food spawn. To fit the requirement of Challenge #3, the food spawn for every run is set to two.



Maze size
7 × 7
Dynamic snake length
Starting snake length
1
Food number
2

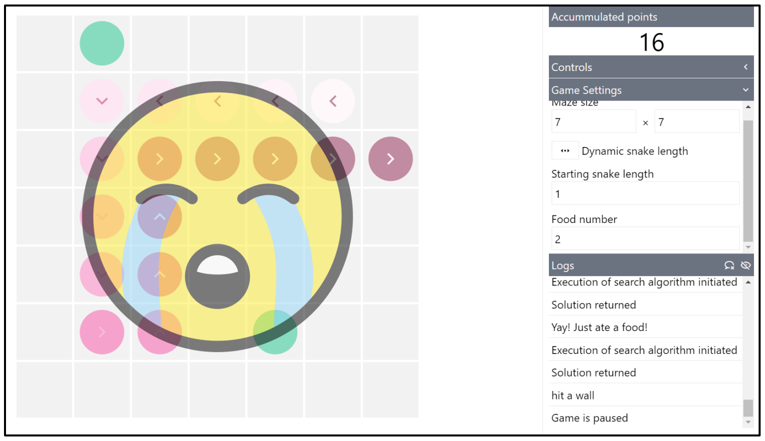
Figure 13: BFS Initial positions of both snake and food location



Maze size
7 × 7
Dynamic snake length
Starting snake length
1
Food number
2

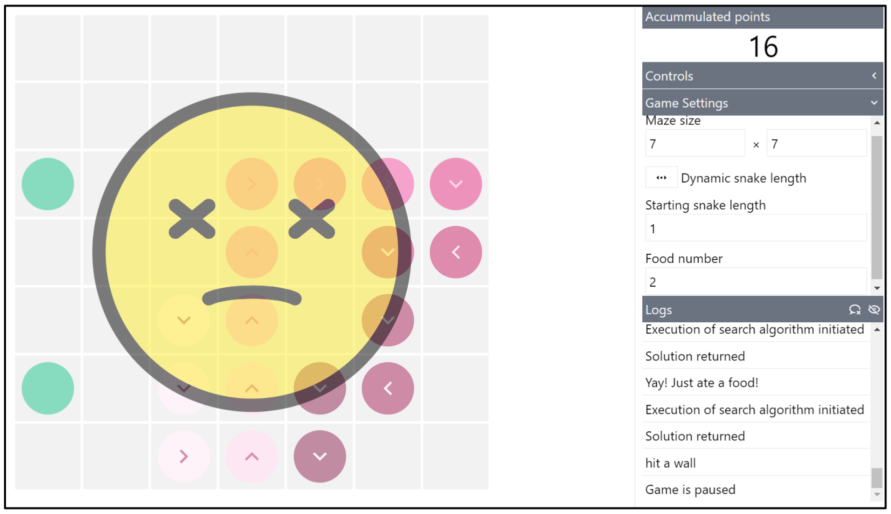
Figure 14: A* Initial positions of both snake and food location

Based on Figure 13 and Figure 14, it is observed that two food has been spawned instead of the default one from the changes in the setting. Thus, the **food_locations** list in the **problem** variable from the interface will now have two elements in the list. The snake will now have to clear the two foods for new food spawns to appear.



```
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
Execution of search algorithm initiated
Solution returned
hit a wall
Game is paused
```

Figure 15: BFS Final positions of both snake and food location



```
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
Execution of search algorithm initiated
Solution returned
hit a wall
Game is paused
```

Figure 16: A* Final positions of both snake and food location (upon Death)

Despite the complexity of the game increasing, the challenge is still completed with the snake only hitting a wall at sixteen points. This fulfils the requirement of accumulating ten points at least. This can be seen in Figure 15 and 16 above.

Conclusion

According to the implementation of the search algorithms for this project, BFS and A* search algorithm, it is observed that both algorithms have their pros and cons. For BFS, it has a faster performance as the algorithm searches through every possible node breadthwise (refer to Figure 1 and 2 for visualisation). This ensures that every possible path to the goal state are explored without having to navigate to unnecessary nodes that are too far. In return, the pathfinding efficiency is reduced since the algorithm takes into consideration all possible nodes within range instead of prioritising the shortest possible path. Meanwhile, the A* search algorithm has a better pathfinding efficiency as it is calculated heuristically to estimate the distance of the initial state and the goal state. However, this algorithm comes with a performance drawback as it has to calculate the heuristic values of every node and takes that value into consideration when selecting the most optimal path.

References

Abiy, T., Pang, H., & Tiliksew, B. (2016, April 29). *A* Search*. Retrieved from Brilliant: https://brilliant.org/wiki/a-star-search/

Lateef, Z. (2019, September 06). *All You Need To Know About The Breadth First Search Algorithm*. Retrieved from edureka: https://www.edureka.co/blog/breadth-first-search-algorithm/

Wenderlich, R. (2011, September 29). *Introduction to A* Pathfinding*. Retrieved from raywenderlich: https://www.raywenderlich.com/3016-introduction-to-a-pathfinding