CSC3206 ARTIFICIAL INTELLIGENCE

ACADEMIC SESSION: MARCH 2021

ASSIGNMENT (1)

DEADLINE: 21ST MAY 2021 5:00pm

Lecturer Name	:	Dr Richard Wong Teck Ken	
Group Member	:	Tan Wyyee	17013673
		Kok Ming Ho	19024272
		Sin Jun	18087619
		Leong Yen Loong	18076083

Contents

1 Introduction

1.1 Problem description

In this assignment, we are required to create agents or players with implementation of different search algorithms in the snake game to achieve the challenges provided. There are 3 challenges given to us to solve in this assignment by using 2 methods, uninformed search algorithm and informed search algorithm. The purpose of this report is to investigate how we apply the search algorithms and problem formulation in the "player" file. This report covers discussion of problem formulation, search algorithms applied in our program, results and lastly a conclusion.

Snake is common game concept in which players control a line (snake) which grows when it eats the target (food). The goal of the game is for the snake to eat randomly spawned-in foods on the map and survive for as long as possible. The food's location is decided randomly by the game and it is only considered "eaten" when the snake's head collides with the food. The scores increased by 1 every time the snake successfully eaten a food. The snake dies when it runs into the borders of the map or itself. The game progressively becomes harder as the snake becomes longer and which makes it harder to avoid.

In this project, our goal is to create two agents, one using uniformed search algorithm, another using informed search algorithm, to navigate the snake to the food while avoiding death. We will be comparing how different search algorithms performs while playing snake.

1.2 Problem formulation

Initial State: at coordinate of snake's head, (x, y), where x, y must be less than the size of the maze.
Actions: (North "n", South "s", East "e", West "w"), snake's head are unable to move in the direction it's body is at

Transition Models: e.g. For coordinate of Snake's Head = (5,5), Result(Snake's Head, North) = (5, 4)

Goal test: is coordinate of snake's head same as food's coordinate, If snake's head(x, y) = food(x, y)

Path cost: each node traverse cost 1 action

2 Case study of the search algorithms

We have picked Breadth-First search algorithm, Depth-First search algorithm, A* search algorithm and Greedy best-first algorithm for our case study. These algorithms will be compared to determine the most suitable algorithm to implement for the snake game.

For Image shown in Breath-First search, A* Search and Best-First Search, I used PathFinding.JS by qiao, and AI-Snake-Game by Louisbourge for Depth-First Search.

https://www.louisbourque.ca/AI-Snake-Game/

https://github.com/qiao/PathFinding.js/wiki

2.1 Uninformed search

Also known as blind search, are strategies that have no additional information provided to them other than those in the problem definition. Search operations are performed blindly until the goal is found.

2.1.1 Breadth-First search (BFS)

The BFS as its name implies, is a technique that prioritized exploring all the node on a depth first before moving on to the next depth level. It is a very effective technique that guarantees to find a solution if the goal node is somewhere at a finite depth level. Due to its nature of fully exploring a depth level first, the solution returned will always be shallowest. However, if the path cost between every node is not of equal value, the shallowest node does not mean the optimal node. The solution is only optimal if:

- The path cost is non-decreasing function of the depth of the node
- All actions have the same cost

To conclude, BFS is complete but it is not optimal. It uses a first-come-first-out queue to store the unexplored nodes.

Breadth-First search in a snake game will guarantees that the returned solution is optimal. This is because the snake's actions (traversing each nodes) have equal cost. Other than that, if there is a safe path to reach the food, BFS will always be able to find it. One scenario that may cause BFS to fail is if the snake is surrounded by its own body or borders.

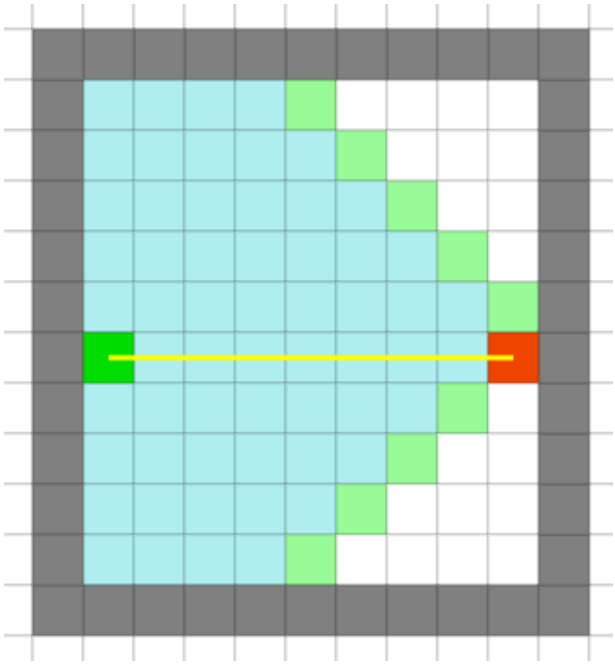

Figure 1. Breadth-First Search in Snake

As shown in Figure 1, the green block represents the initial node, or in snake's term, the snake's head. The red block is the food, and the yellow line is the solution (path). Other than that, the light blue blocks are explored nodes and light green block for the frontier. As mentioned, BFS will explore nodes at the present depth first, so the nodes surrounding the

initial node are explored first. Then the algorithm will move on to the adjacent nodes of the previously explored nodes. Hence, it will look like the light blue blocks is expanding outward from the green block.

2.1.2 Depth-First Search (DFS)

DFS is an algorithm that picks a branch and traverse down as deep as possible before backtracking. This process is repeated until the goal node is found. DFS uses a stack to store the nodes, which does the opposite of BFS. New unexplored nodes are stacked on top of previously unexplored nodes and are the first one to be expanded. This means that newer nodes will be expanded first before old nodes, indicating a last-in-first-out characteristic.

DFS are complete when used in a graph-search but will not always finds a solution for tree-search. The solutions acquired from DFS are also unlikely to be optimal.

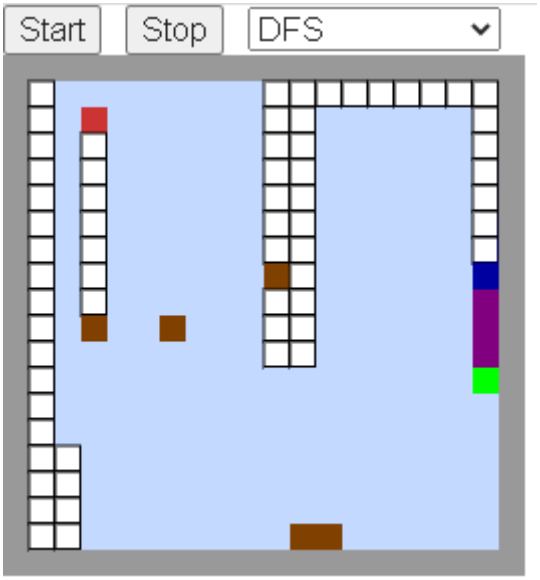

Figure 2. Depth-First Search in Snake

In Figure 2, the green block represents the head of the snake, and everything north of the green block is the body of the snake. Red is the food, and browns are obstacles. The light blue blocks are the “paths” that the snake must follow to get to the food. This is the result of DFS. When the food is found very deep down the branch, the solution returned will also be a long path. Since DFS will always go down a branch as far as possible, often that when it located the food, the path will also be very deep.

2.2 Informed Search

Also known as heuristic search. The Informed search utilise problem-specific knowledge that is beyond the problem definition. In this case, the algorithm knows the coordinates of the food.

2.2.1 Greedy best-first search

This algorithm follows one rule when picking a node to expand, which is the node that is closest to the goal. How close a node is to the goal is determine by a heuristic function, which is a logically estimation. Since it's just an estimation, it may not necessarily be accurate, but just a rough idea.

Since only the heuristic function is used for greedy best-first search, the formula for it is:

$$f(n) = h(n)$$

Figure 3. Greedy best-first search formula.

The Greedy best-first search has minimum search cost because it only expands the solution path. It has completeness when it comes to graph search with finite spaces, but not with tree search due to possible loopy path. It also does not always return the optimal solutions, as it doesn't calculate the cost to travel to a node, sometimes a node that is closest to the goal may be very far away from the root node.

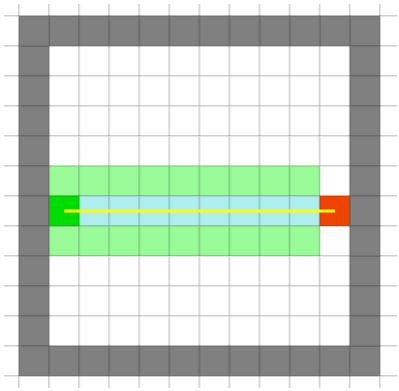

Figure 4. Greedy best-first Search in Snake

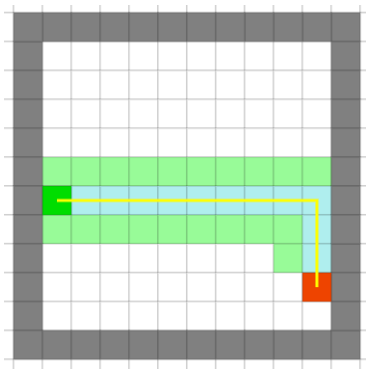

Figure 5. Greedy best-first Search in Snake

In Figure 4 and Figure 5, we can see that the algorithm went straight towards the food. It only considers 1 solution path and disregard other alternatives solution. The heuristic function used for this is Manhattan Distance. The Manhattan Distance is the distance between two coordinates. The formula for this function is: $h(n) = |x_1 - x_2| + |y_1 - y_2|$.

In a snake game, the Manhattan distance can be used to estimate the distances between a node and the goal in order to determine the closest node to goal.

2.2.2 A* Search

A*, is a variant of the best-first search. The difference is that A* not only looks at the estimated cost of a node to goal, it also looks at the cumulative cost needed to go to said node. The formula for A* search is:

$$f(n) = g(n) + h(n)$$

Figure 6. A search evaluation function*

Where g(n) is the path cost needed to go from the root node to node n. h(n) is the heuristic function, same as the one we discussed in Greedy Best-first search, which is the estimated cost for node n to the goal. Finally, f(n) is the combination of both these values. The A* can also uses Manhattan Distance discussed in Best-first search.

A* search is a complete search, which means it will guarantee the return of a solutions if there is one, and the returned path is always optimal if the conditions are met. The two conditions required are:

- If h(n) is admissible heuristic
- If h(n) is consistent heuristic

Admissible heuristic means that the estimation never exceeds the actual cost. Secondly, a heuristic can be consistent if its estimate, h(n), is always equal or lower than the estimated distances, h(n') + c(n, a, n'), from neighbouring vertex to the goal, adding the cost of reaching that neighbour.

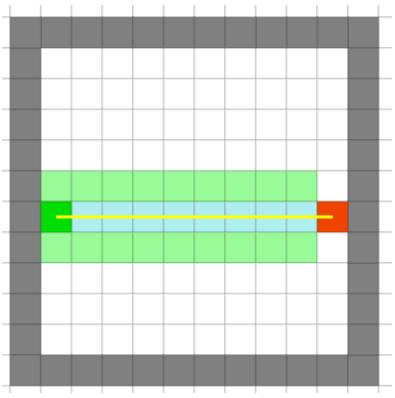

Figure 7. A search in Snake*

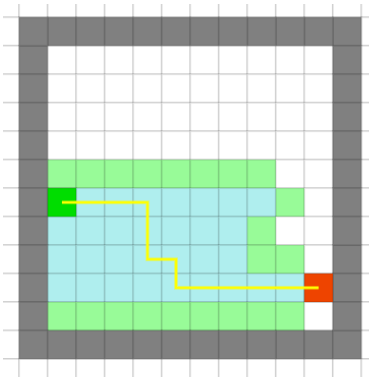

Figure 8 A search in Snake*

From Figure 7, we can see that it is the same as Figure 4, Greedy best-first search. This is because that there the f(n) is lower east (right) of the initial node. Therefore, there are no reason to expand the other nodes. However, in Figure 8, we can see that unlike in Figure 5, where the Greedy best-first search did a sharp turn in the end, A* algorithm starts to explore alternative paths which yields the same amount of f(n). This allows A* to always return the most optimal solutions.

3 Selecting the most suitable algorithm to implement

For uninformed search, we can see that BFS will give us the optimal solutions because all the nodes are of equal distance, whereas DFS may return very long paths. Therefore, for our uninformed search, we will do Breadth-first search.

As for informed search, we can see from Figure 4, Figure 5, Figure 7 and Figure 8, both algorithms returned the optimal solutions, but best-first search managed to have much lesser search time and operations. However, while is it true that best-first search will return optimal solution if the snake is small, this may not be the case with a bigger snake.

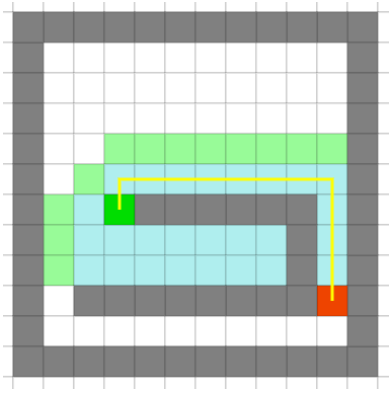

Figure 9. A search in Snake*

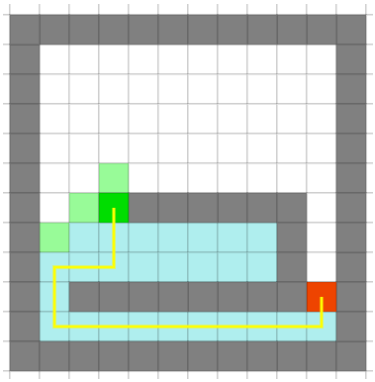

Figure 10 Greedy Best-first search in Snake

We can see in Figure 9 and Figure 10, the greedy best-first search takes the longer path to reach the goal while A* still picks the optimal path. This is because BFS does take the path cost from root node into consideration when expanding. Finally, we decided that we will be doing A* search for our informed search.

To conclude, we pick Breadth-First search for uninformed search and A* search for informed search.

4 Implementation

4.1 Breadth-first search

In this Snake game, there are many ways we can use to find out the final path to the food location, and we use Breadth First Search (BFS) for uninformed search and A* search for informed search.

As we had learned BFS before, this search algorithm will search horizontally before search vertically, and for the grid diagram like Snake game, we need to explore the adjacent node, which is the left, right, up and down of our current node to find out the best path to get the location of food, then all the successors of the root node are expanded next, then their successors, and so on. The queue of BFS works First In First Out (FIFO) or First Come First Out and underlies a queue will ensure that those things that were discovered first will be explored first. Before we figure out the least path, the shallowest unexpanded node between the initial and end coordinates is always chosen for expansion and it will have a FIFO queue for the frontier. At the end of BFS, we will choose the least path of coordinates of the snake to eat the food.

When the way to find the path from snake to location is the Breadth First Search (BFS), the first coordinate we insert into the start list is the location of our snake at first of the game, and the list is marked as frontier. After that, we also will let the snake locations be inserted into a queue which is called explored so that we will not let the snake eat itself in the path to find the food locations. While the found goal is still false, we use expandAndReturnChildren function to find the adjacent nodes of current nodes and then we let the adjacent coordinates in queue is dequeued, all the adjacent nodes surround the current node will be pushed onto the queue which are called explored. For every child in the children list which were not expanded previously, do the goal test to check whether one of them is the goal coordinate or not. Subsequently, we will repeat the above steps over and over again until we get the path from the initial start node to the food, we will store the coordinate action and the coordinate

of the node into the solution and path, and return the final path at the end of the whole function of BFS.

```
def bfs(problem, setup):
    print("this is breadth-first search.")
    idIterator = itertools.count(1)
    expansionSequenceIterator = itertools.count(1)
    trees = []
    frontier = []
    explored = []
    found_goal = False
    goalie = Node()
    solution = []
    frontier.append(Node(problem["snake_locations"][0], None, None, next(idIterator)))
    trees.append(searchTree(frontier[0],next(expansionSequenceIterator), False))
    for x in problem["snake_locations"]:
        explored.append(Node(x, None))
    while not found_goal:
       # expand the first in the frontier
       children = expandAndReturnChildren(setup["maze_size"], frontier[0], idIterator)
       # add children list to the expanded node
       frontier[0].addChildren(children)
       # add to the explored list
       explored.append(frontier[0])
       # remove the expanded frontier
       del frontier[0]
       # add children to the frontier
       for child in children:
           trees.append(searchTree(child))
           # check if a node was expanded or generated previously
           if not (child.state in [e.state for e in explored]) and not (child.state in [f.state for f in frontier]):
               # goal test
               if child.state == problem["food_locations"][0]:
                   found_goal = True

                   goalie = child
               frontier.append(child)
               for tree in trees:
                   if tree.node.id is child.id:
                       tree.removed = False
                       break

               for tree in trees:
                   if tree.node.id is frontier[0].id and tree.expansionsequence is -1:
                       tree.expansionsequence = next(expansionSequenceIterator)
                       break
       print("Explored:", [e.state for e in explored])
       print("Frontier:", [f.state for f in frontier])
       print("Children:", [c.state for c in children])
       print("")
       solution = [goalie.action]
       path = [goalie.state]
       while goalie.parent is not None:
           solution.insert(0, goalie.parent.action)
           path.insert(0,goalie.parent.state)
           for e in explored:
               if e.state == goalie.parent.state:
                   goalie = e
                   break
    print("path: ",path)
    del solution[0]
    search_tree = []
    for node in trees:
        search_tree.append(node.returnTreeNode())
    return solution, search_tree
```

Figure 11. Python code for. Breadth-first search

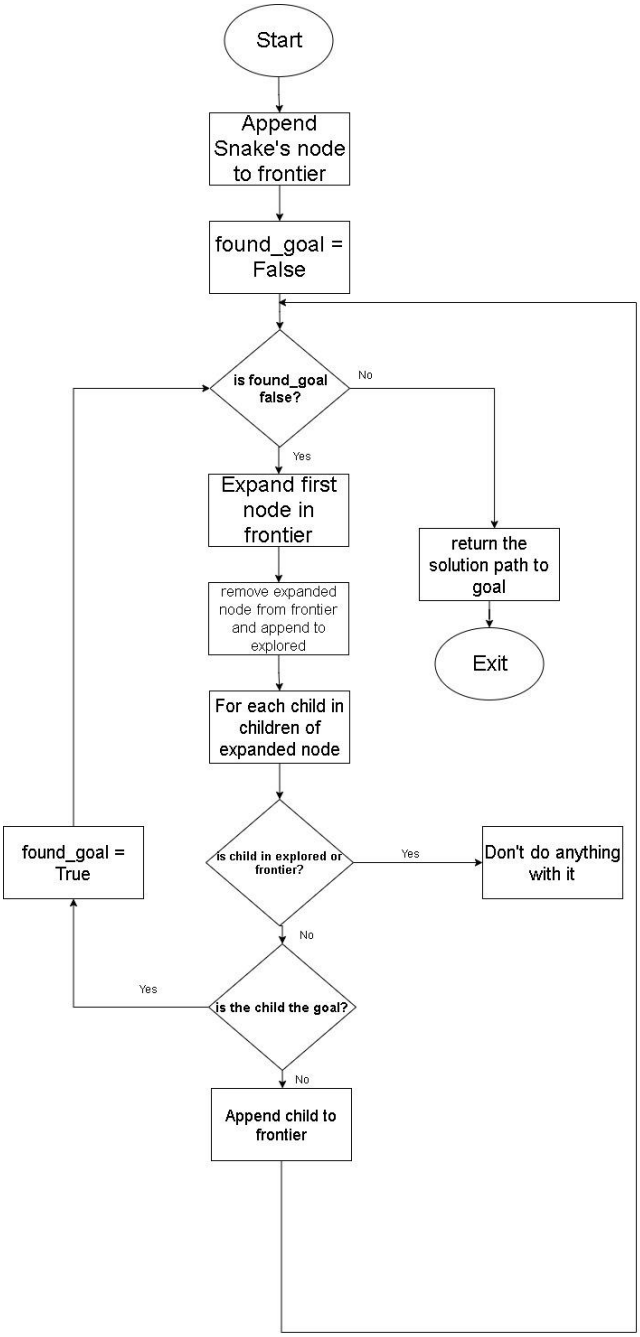

Figure 12. Flowchart for Breadth-first search

4.2 A* search

For the A* search we chose for the informed search, A* search is a bit similar as the BFS algorithm we used before because it can be said as an improved version of BFS (breadth first search). We know that in the Snake game, the adjacent nodes of the current node are needed to find out, and all the g and h of all those adjacent nodes will be used to compare their distance between the initial node and the current node and the distance between the current node and the final food.

In our function, we use the expandAndReturnFunction to find the adjacent nodes of the current coordinate we insert and all the g and h of the adjacent nodes will also be calculated by using the returnManhattanDistance and we will used them to compare each other, the

coordinate which has the lowest f will be chosen as the next node, we will repeat and repeat again until we get the final path. Same as the BFS function we used at uninformed search, the initial node of the snake will be inserted into the explored list first and all the adjacent nodes which surround the current node will be appended into the children list and explored list, the reason we insert them into the explored list is because we will only choose the coordinate which has the lowest f. For every child which was not generated before, the goal test is also necessary to check whether one of them is the goal coordinate or not. If the current node is not the equal coordinate of the food, we skip the step to make the current node equal to goalie. For this algorithm, instead of simplifying appending the nodes, we use appendAndSort function to print out the fcost of the current node and insert newly generated child nodes into a specific index. The rule for the index is based on the f value of the nodes, lower f value will be inserted before those with higher f value. Subsequently, we will repeat the above steps over and over again until we get the path from the initial start node to the food, we will store the coordinate action and the coordinate of the node into the solution and path, and return the final path at the end of the whole function of A* search algorithm.

```
def astar(problem, setup):
    print("this is A* search.")
    idIterator = itertools.count(1)
    expansionSequenceIterator = itertools.count(1)
    trees = []
    frontier = []
    explored = []
    found_goal = False
    goalie = Node()
    solution = []
    frontier.append(Node(problem["snake_locations"][0], None, None, next(idIterator), 0,
                         returnManhanttanDistance(problem["snake_locations"][0], problem["food_locations"][0])))
    trees.append(searchTree(frontier[0],-1, False))
    for x in problem["snake_locations"]:
        explored.append(Node(x, None))
    while not found_goal:
        # expand the first in the frontier
        children = expandAndReturnChildren(setup["maze_size"], frontier[0], idIterator,problem)
        # add children list to the expanded node
        frontier[0].addChildren(children)
        for tree in trees:
            if tree.node.id is frontier[0].id:
                tree.expansionsequence = next(expansionSequenceIterator)
                break
        # add to the explored list
        explored.append(frontier[0])
        # remove the expanded frontier
        del frontier[0]
        # add children to the frontier
        for child in children:
            trees.append(searchTree(child))
            # check if a node was expanded or generated previously
            if not (child.state in [e.state for e in explored]) and not (child.state in [f.state for f in frontier]):
                # goal test
                if child.state == problem["food_locations"][0]:
                    found_goal = True
                    goalie = child
                frontier = appendAndSort(frontier, child)
                for tree in trees:
                    if tree.node.id is child.id:
                        tree.removed = False
                        break
                #frontier.append(child)
        print("Explored:", [e.state for e in explored])
        print("Frontier:", [f.state for f in frontier])
        print("Children:", [c.state for c in children])
        print("Trees:", [t.node.id for t in trees])
        print("")
        solution = [goalie.action]
        path = [goalie.state]
        while goalie.parent is not None:
            solution.insert(0, goalie.parent.action)
            path.insert(0,goalie.parent.state)
            for e in explored:
                if e.state == goalie.parent.state:
                    goalie = e
                    break
    print("path: ",path)
    del solution[0]
    search_tree = []
    for node in trees:
        search_tree.append(node.returnTreeNode())
    return solution, search_tree
```

Figure 13. Python code for A search*

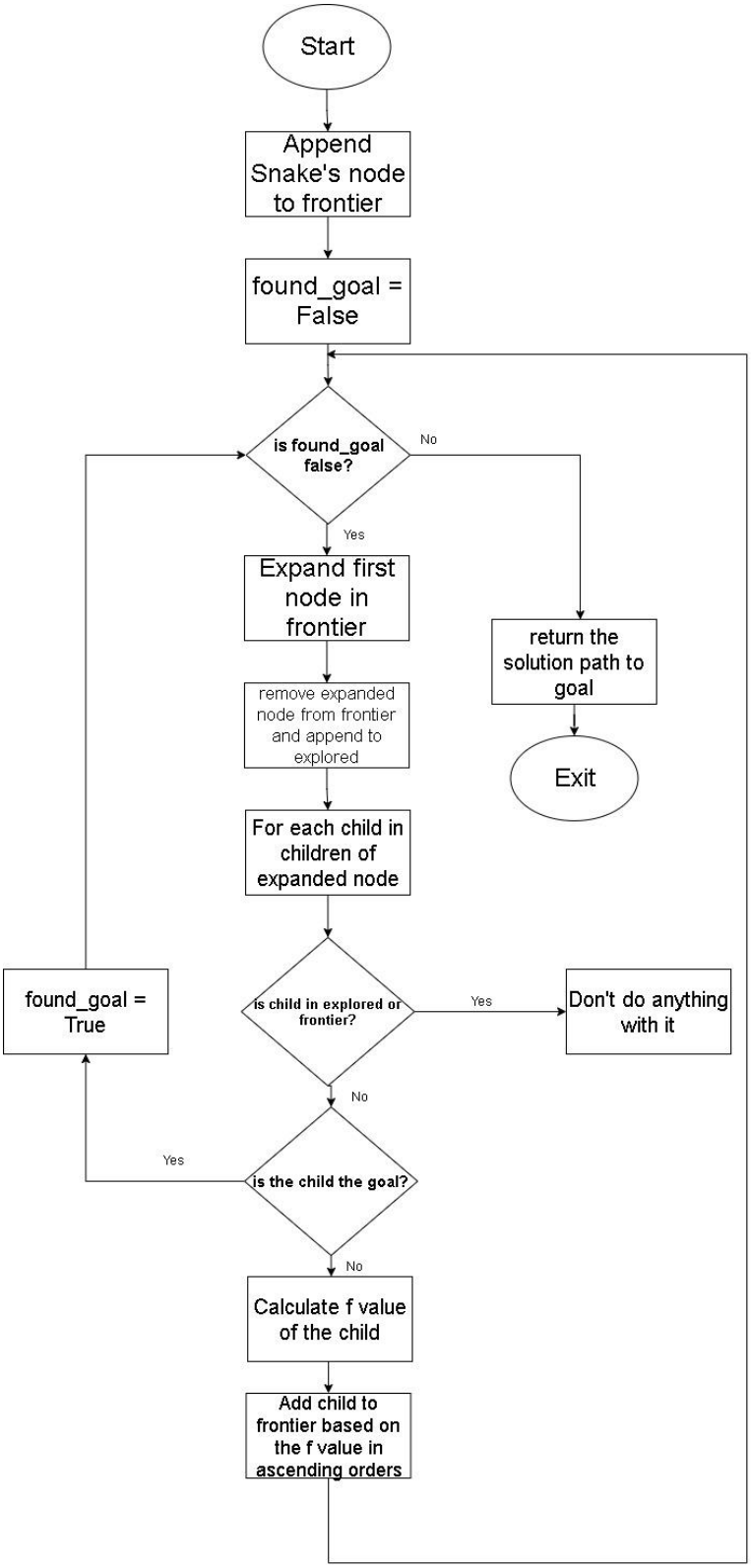

Figure 14. Flowchart for A search*

5 Results and Analysis

5.1 Challenge 1

Attempt	A* search		Breadth First Search	
	Result	Scores	Result	Scores
1	achieved	50	achieved	50
2	achieved	50	achieved	50
3	achieved	50	achieved	50
4	achieved	50	achieved	50
5	achieved	50	achieved	50

Table 1. Results for challenge 1 (One food at any time; Non-increasing snake length; Snake length of one; reaching at least 15 points)

For challenge one, our algorithms achieve to make the snake impossible to die as the length of snakes does not increase. Therefore, we stop the test when the snake scores 50 points.

5.2 Challenge 2

Attempt	A* search		Breadth First Search	
	Result	Scores	Result	Scores
1	achieved	17	achieved	24
2	achieved	25	achieved	20
3	achieved	24	achieved	16
4	achieved	25	achieved	20
5	achieved	24	achieved	18

Table 2. Results for challenge 2. (One food at any time; Increasing snake length with food; Starting snake length of one; reaching at least 10 points)

5.3 Challenge 3

Attempt	A* search		Breadth First Search	
	Result	Scores	Result	Scores
1	achieved	21	achieved	41
2	achieved	18	achieved	23
3	achieved	22	achieved	34
4	achieved	22	achieved	20
5	achieved	24	achieved	17

Table 3. Results for challenge 3. (Two food generated when no food is available on the maze; Increasing snake length with food; Starting snake length of one; reaching at least 10 points)

5.4 Comparison of performance between Breadth-first search and A* search

To test the performance of the algorithm, we set the maze into 30 x 30 maze and animation speed into 5 steps per second. Then we take the average time taken for the snake to score 10 food. When the snakes ate the 10^{th} food, immediately stop the timing. The test is repeated 5 times to get a more accurate result.

Attempt	Result (s)
1	53.12
2	35.46
3	49.84
4	35.65
5	55.79
Average (total results in s / attempts)	45.97

Table 4. Performance of A search*

Attempt	Result (s)
1	58.23
2	62.84
3	64.66
4	52.59
5	37.56
Average (total results in s / attempts)	55.18

Table 5. Performance of Breadth-first search

As the result in Table 4 shown, the average time taken for the snake to gain 10 scores of the A* search algorithm is 45.97s, which is shorter than the Breadth First Search algorithm shown in Table 5, which is 55.81s. This has concluded that the performance of the A* search algorithm is better than the Breadth First Search algorithm. During the test, we found out that when we execute the game, the Breadth First Search algorithm player will take longer to think which path it will take.

5.5 Analysis

Both of our search algorithms can work indefinitely when snake's length is one and fixed. From this we can conclude that the algorithms work as intended and have not encountered any bug. As for challenges 2 and 3, there is not much difference between the scores. This is because the time taken is not considered and how many foods can the snake eat before game over is highly dependent on the locations the food randomly spawned on. Sometimes when trying to eat the food, the snake may accidentally corner itself. This situation is more likely as the snake gets bigger. In our algorithm, we have not coded any workarounds to prevent the snake from cornering itself. Lastly, we can see that A*takes fewer operations and times to come out with a solution. Since, both A* search and breath-first search both return optimal solutions in snake game, the time difference can only be due to search cost.

6 Conclusion

In this research, we concluded that breadth-first search is better than depth-first search for snake game. Depth-first search will very likely return path that is non-optimal. Other than that, while greedy best-first search has less search cost compared to A* search, it doesn't always turn the optimal solution. Hence, we determined the best algorithms to implement is breath-first search and A* search. Finally, we successfully completed three of the challenges provided and compared the performance between the two algorithms.

7 Appendices

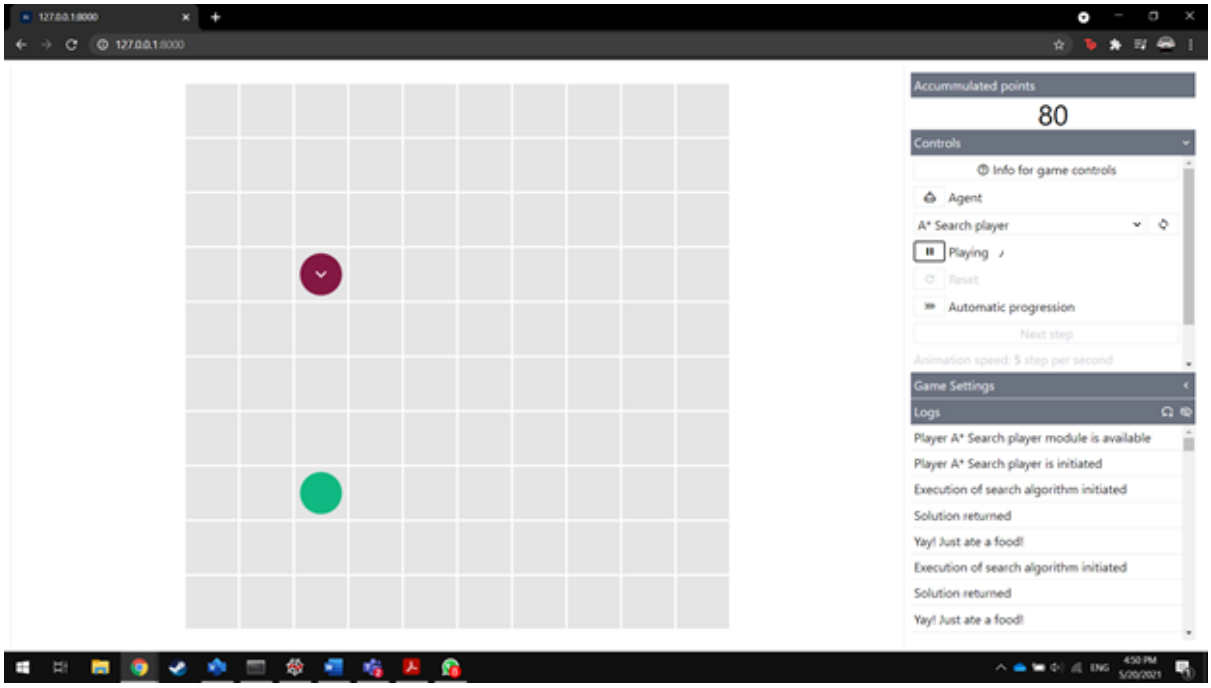

Figure 15. Challenge 1 screenshot

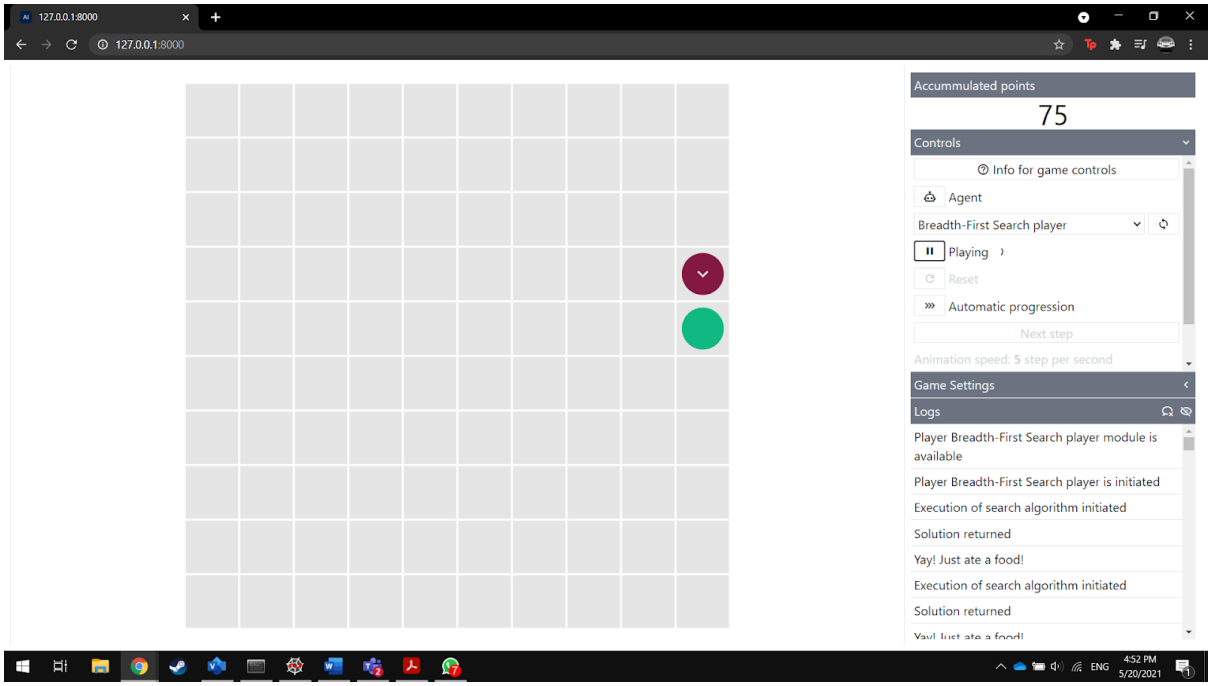

Figure 16. Challenge 1 screenshot

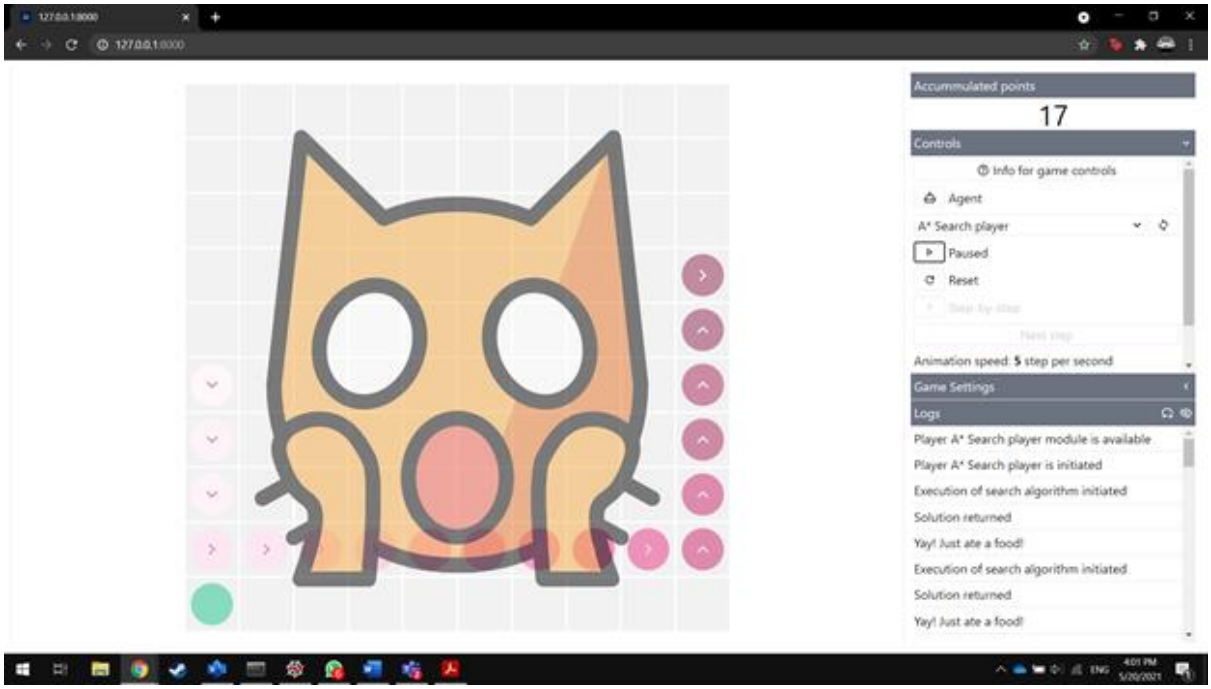

Figure 17. Challenge 2 screenshot

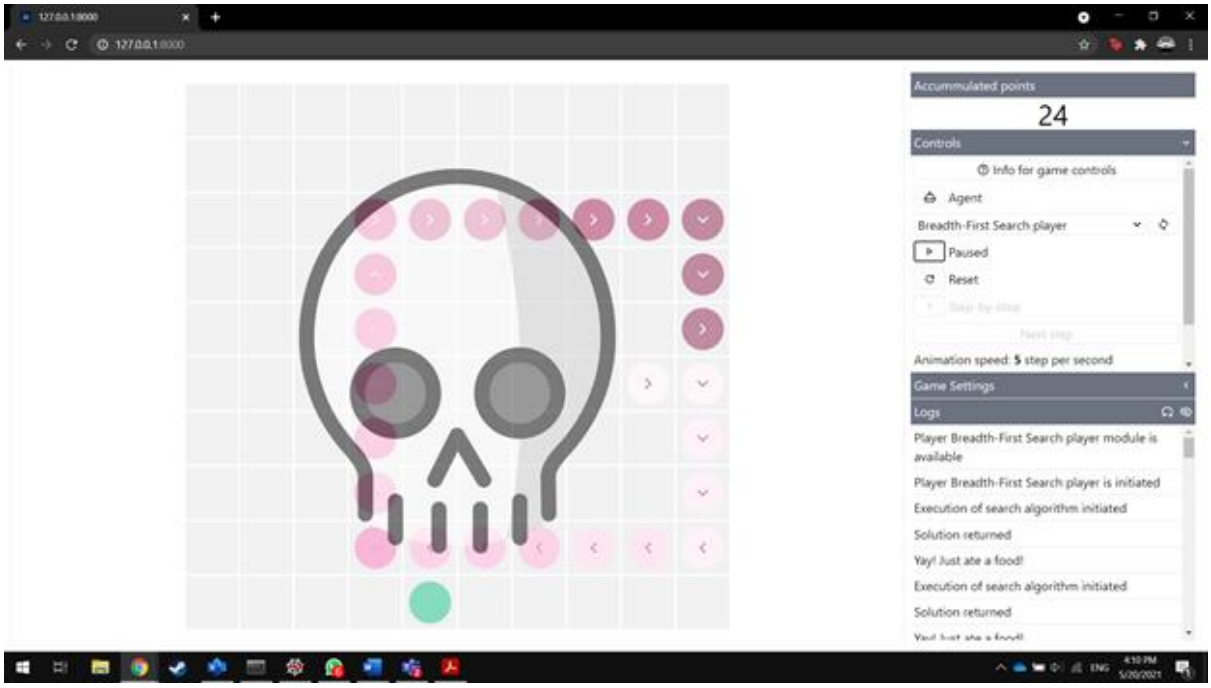

Figure 18. Challenge 2 screenshot

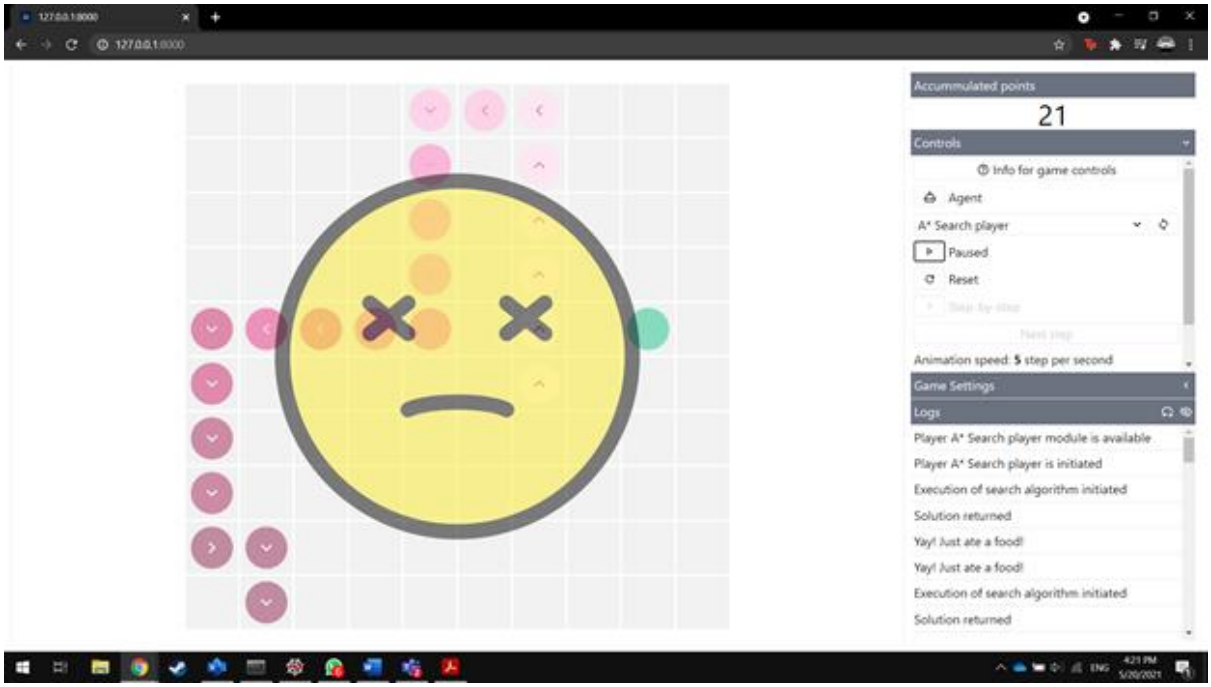

Figure 19. Challenge 3 screenshot

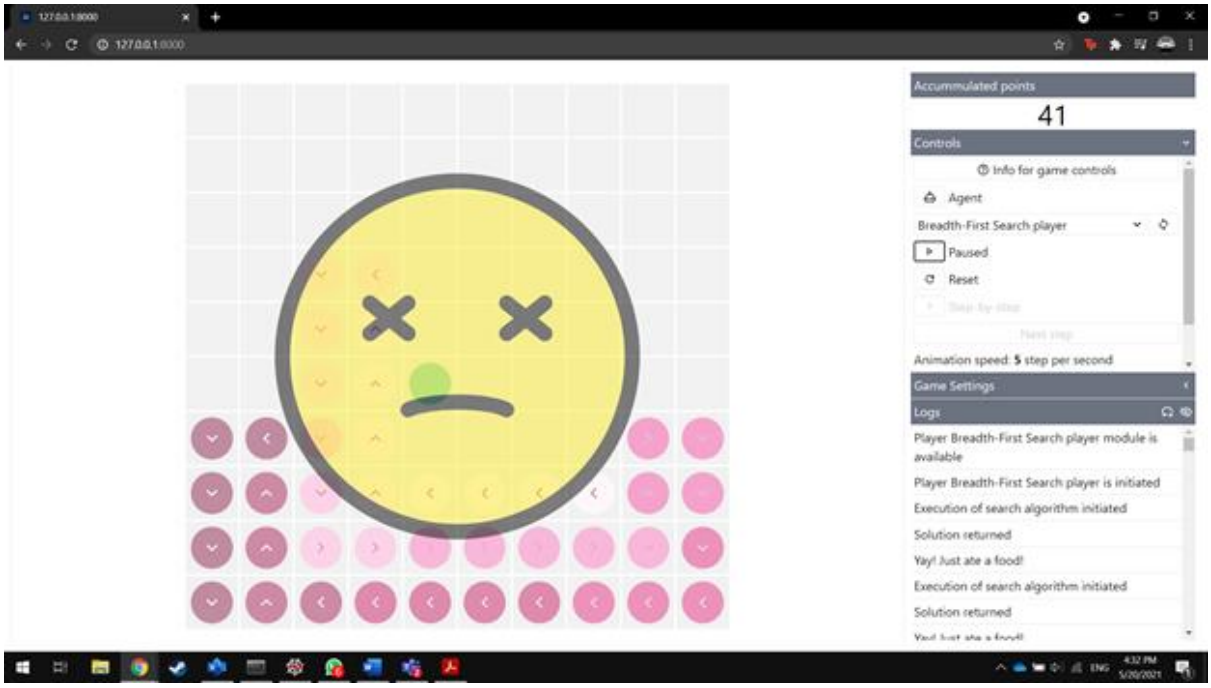

Figure 20 Challenge 3 screenshot

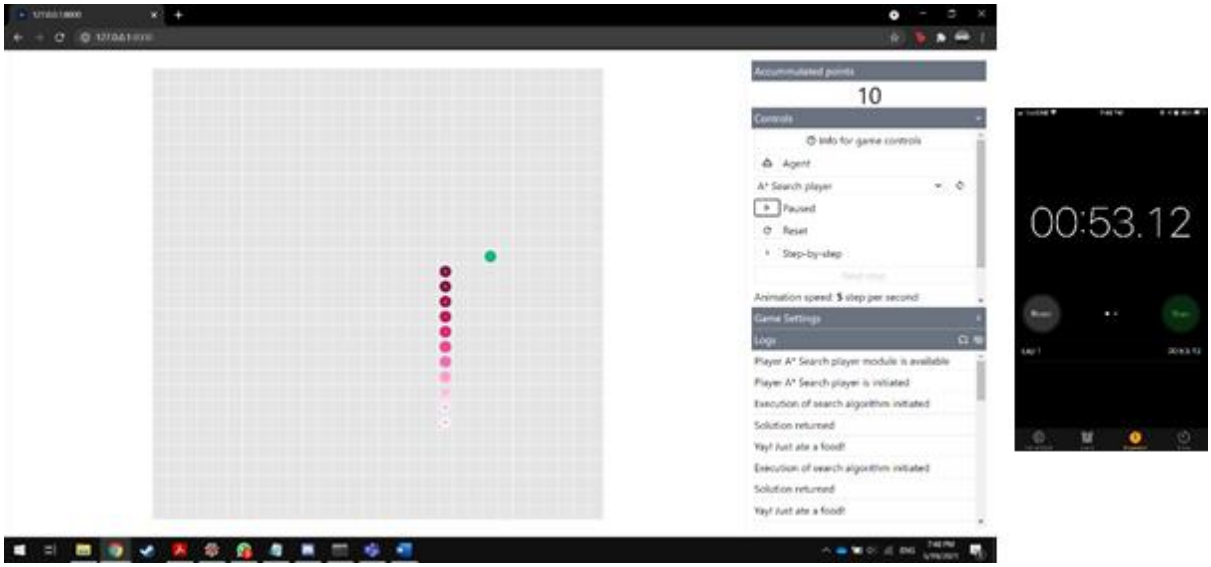

Figure 21. Performance test for A screenshot*

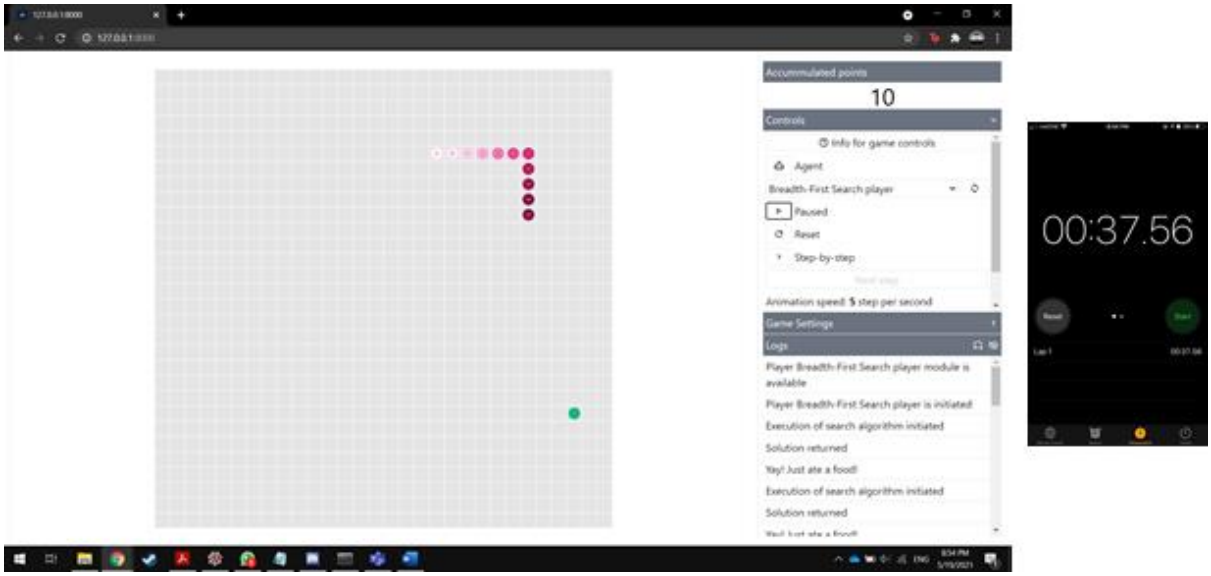

Figure 22. Performance test for Breadth-first search screenshot

Breadth-first search screenshot