

ARTIFICIAL INTELLIGENCE ASSIGNMENT 1

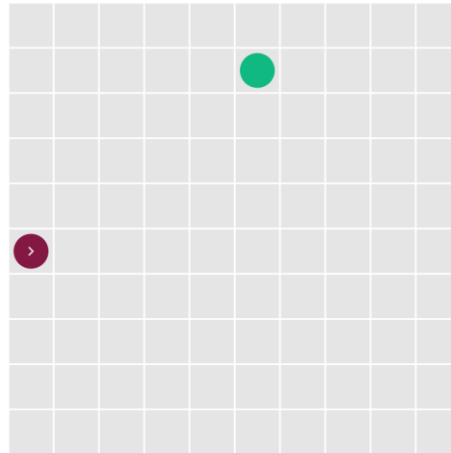
Group Name: Caffeine

Group Member:

Student ID	Name
17071549	Pan Chen Soon
16087520	Pan Chen Pong
18037259	Tan Sze Qin

a. **What is the problem the algorithm needs to solve?**

In this snake game, the initial snake location (red circle) would be the initial state and the food location (green circle) would be the goal state. The algorithm is supposed to search for the food location based on the position of the snake. And a new food location will be generated randomly within the maze once the snake ate the food.



b. **How do the different parameters of the problem fit into the algorithm?**

The first parameter of the problem is “snake_locations” and “snake_locations” is a list of coordinates which is passed in from the frontend code. However, in our case, the length of the snake is static, hence there will only be one coordinate passing in from the “snake_locations” parameter. When the run function is called, it will prepare all the relevant variables and it will append the “snake_locations” coordinate into the frontier as shown in diagram 2. The coordinate will then be used to start the search algorithm. For instance, if the current coordinate of the snake is at [0,5], then it will start expanding from [0,5] to get the coordinates of the nodes around [0,5]. Therefore, the snake location (root node) is appended into the frontier as shown in line 60 and the root node will then become the initial state for the algorithm to be expanded.

What you have written is "how do the code uses the input arguments"

```
36 def run(self, problem):
37     # problem = {
38     #     snake_locations: [[int,int],[int,int],...],
39     #     current_direction: str,
40     #     food_locations: [[int,int],[int,int],...],
41     # }
42     solution = []
43     children = []
44     directions = "nswe"
45     explored = []
46     frontier = []
47     expansionSequence = 1
48     search_tree = []
49 > search_tree.append({...
50
51
52
53
54
55
56
57
58
59     #append the snake location into the frontier
60     frontier.append(Node(problem["snake_locations"][0], None))
```

there is nothing about the algorithm

The “food_locations” parameter will be used as the goal node checker. The search algorithm will keep looping the coordinates using a while loop and it will stop the loop when the child coordinate is the same as the “food_locations” coordinate. Furthermore, both parameters of

“snake_locations” and “food_locations” will be inserted into a heuristic function to estimate the distance from the node to the goal. Nevertheless, once the snake has successfully eaten a food, a new coordinate for both “snake_locations” and “food_locations” will be passed in from the frontend again to run the search algorithm again, and this cycle will keep on looping until the targeted score is reached or when the user stops it.

c. How do you make the algorithm understand and be able to solve this particular problem?

Uninformed Search (Breadth First Search)

Breadth First Search algorithm is an algorithm where the root node is expanded first, then all the successors of the root node will be expanded next and this cycle continues until the goal is found/reached. Furthermore, it is an algorithm that expands the shallowest node that is unexpanded first.

```
2 class Node:
3     def __init__(self, state = None, parent=None, direction=None):
4         self.state = state
5         self.parent = parent
6         self.direction = direction
7         self.children = []
8
```

First, we define the Node class and it has multiple parameters available in the Node Class, for instance, store the state of the node, its parent of the current node's state, it's direction to reach the node and its children. These parameters will have their own value when a new Node object is created.

```
59 #append the snake location into the frontier
60 frontier.append(Node(problem["snake_locations"][0], None))
61 found_goal = False
62 # actions dict to be loop later when expanding the node
63 actions = {"n": [0, -1], "s": [0, 1], "w": [-1, 0], "e": [1, 0]}
64
```

Then we append the “snake_locations” into the frontier and the algorithm will start expanding the first node (initial state) in the frontier and defined the “actions” dictionary at line 63 and it consists of 4 letters “N”, “S”, “E”, “W” as the key and the coordinates of where it should move as the value.

After preparing all the needed parameters and data, we start the search by running the while loop in line 61, the initial state (frontier[0]) will be expanded by looping the “actions” dictionary as shown in line 68 and 69. As shown in line 70, the newly generated position will be checked if the coordinate exceeds the maze size and if the generated coordinates do not exceed the maze, it

don't explain the code by just stating the code in sentences.

will be appended into the children list as shown in line 71 and the relevant data such as the state of the node, its parent's node and the directions will also be appended in to the children list.

```
61 while not found_goal:
62     children = []
63     print("Expansion Sequence", expansionSequence)
64     expansionSequence = expansionSequence + 1
65     print("Current Node: ", frontier[0].state)
66
67     # loop all the actions in the actions dict to expand the node.
68     for x in actions:
69         new_position = [frontier[0].state[0] + actions[x][0], frontier[0].state[1] + actions[x][1]]
70         if not (new_position[0] < 0 or new_position[0] > self.maze_size[1]-1 or new_position[1]<0 or new_position[1]
71             > self.maze_size[0]-1):
72             children.append(Node(new_position, frontier[0].state, x))
73
74     frontier[0].addChildren(children)
75     explored.append(frontier[0])
76     print("Children of Current Node: ", [f.state for f in children])
77
78     for child in children:
79         if not (child.state in [e.state for e in explored]) and not (child.state in [f.state for f in frontier]):
80             if child.state == problem["food_locations"][0]:
81                 found_goal = True
82                 goalie = child
83                 frontier.append(child)
84
85     del frontier[0]
```

Then, for each of the children of the expanded node will go through a checking of whether the child state is already in the frontier or explored list to ensure there is no loopy path or any redundant state as shown in line 78. If the child node is not in the frontier and the explored list, it will then append the child node into the frontier. Once the node has been expanded successfully, the expanded node will be removed from the frontier and it will be stored in the explored list as shown in line 74 and 84. This process will continue until the child state is equal to the food_location coordinate, then the "found_goal" boolean variable will be checked to True and set child node to the "goalie" variable to back variable to stop the while loop at line 61. Hence, if the "found_goal" boolean variable is still false, it will continue to run the while loop and expand the next node in the frontier.

```
93     path = [goalie.direction]
94     while goalie.parent is not None:
95         path.insert(0, goalie.direction)
96         for e in explored:
97             if e.state == goalie.parent:
98                 goalie = e
99                 break
100     del path[-1]
101     solution = path
102
103     return solution, search_tree
```

Once the goal state is found, it will exit the while loop and proceed to extract the solution using the Node class. Then, using the goal node, it will trace back to the parent node, using the parent property as defined in the Node Class as shown in line 6 at the image below. Then, the parent node will use the same method to get the predecessor node and this process will continue until the initial state that has no parent to obtain the solution (lists of directions).

Informed Search (Greedy Breadth First Search)

Compared to uninformed search, informed search algorithms have information on the goal state, allowing the search process to be more efficient. There are a few types of informed search, nonetheless, we have chosen to use Greedy Best-First Search to solve this problem. Greedy Best-First Search uses a heuristic function to estimate the distance between the node and the goal.

In our Greedy Best-First Search algorithm to solve the snake game, the heuristic function is calculated using the Manhattan Distance to define the distance between the snake location and the food location. The formula for Manhattan Distance is $|x_2 - x_1| + |y_2 - y_1|$, the first coordinate (x_1, y_1) would be the snake location and (x_2, y_2) would be the food location.

```
61 found_goal = False
62 actions = {"n": [0, -1], "s": [0, 1], "w": [-1, 0], "e": [1, 0]}
63 expansion_sequence = 0
64
65 goal_node = problem["food_locations"][0]
66
67 self.frontier.append(Node(problem["snake_locations"][0], None, None))
68
69 while not found_goal:
70     current_node_children = []
71     if self.frontier[0].state == problem["food_locations"][0]:
72         found_goal = True
73         goalie = self.frontier[0]
74         break
75
76     for x in actions:
77         new_position = [self.frontier[0].state[0] + actions[x][0], self.frontier[0].
78                         state[1] + actions[x][1]]
79         cost = (abs(goal_node[0] - new_position[0]) + abs(goal_node[1] - new_position
80                  [1]))
```

In the snake game, the snake can only move to one coordinate at a time, either north(n), south(s), east(e) or west(w). Hence, line 62 from the code shows the actions dictionary to define the snake's movement. Furthermore, one of the most important parts of informed search is to define the goal, thus, the food location is defined as the goal node as shown in line 65. And the snake location is appended into the frontier as shown in line 67.

After knowing the food location and the snake location, the algorithm can calculate the children nodes of the snake location by looping the actions dictionary. After each child node is identified as new_position shown in line 77. The algorithm will carry on to calculate the cost from each child node to the goal node by using the Manhattan Distance theory as shown in line 78.

```

80         if not (new_position[1] < 0 or new_position[1] > self.grid_size[0]-1 or
81               new_position[0] < 0 or new_position[0] > self.grid_size[1]-1):
            children.append(Node(new_position, self.frontier[0].state, x, cost))

```

The algorithm will also verify whether each child node is within the maze by comparing the child's coordinate and maze's coordinate as shown in line 80 before appending each child node into the children list shown in line 81. In addition, each child node appending into children would contain its coordinate, parent node, direction and the cost from the goal node.

```

86     self.frontier[0].addChildren(children)
87     explored.append(self.frontier[0])
88
89     for child in children:
90         if not (child.state in [e.state for e in explored]) and not (child.state in
91               [f.state for f in self.frontier]):
92             self.sort(child)
93         for f in explored:
94             if f in self.frontier:
95                 self.frontier.remove(f)

```

After the children have been verified, it will be added to the expanding node's children parameter in line 86 as well as appending into the explored list in line 87.

Thereafter, for each child of the expanded node, a checking is done to make sure the ~~child~~ state has not been in the frontier or explored list. This checking is to ensure there is no ~~loopy~~ path or any redundant state as shown in line 90. After ensuring the child is not in the frontier or explored list, the child will be sorted in the frontier by calling the sort function as shown in line 91. In addition, if the child node has been previously explored (in the explored list). It will be removed from the frontier as shown in line 92 - 94. This is to prevent the algorithm to keep expanding the parent node.

```

37     def sort (self, child):
38         duplicated = False
39
40         for i, f in enumerate (self.frontier):
41             if f.state == child.state:
42                 duplicated = True
43
44         if not duplicated:
45             index = len(self.frontier)
46             for i, f in enumerate(self.frontier):
47                 if child.cost < f.cost:
48                     index = i
49                     break
50             self.frontier.insert(index, child)

```

The code shown above is the sort function mentioned previously. When the sort function is called, the algorithm will sort the child by first checking whether there is another node in the frontier that has the same state as the child, if yes, duplicated will be True as shown in the first

for loop from line 40 - 42. However, if there is no duplicate, the algorithm will sort/rearrange the frontier in an ascending order based on the cost, allowing us to always expand the first node of the frontier as shown in line 44 - 50.

```
68     while not found_goal:
69         current_node_children = []
70         if self.frontier[0].state == problem["food_locations"][0]:
71             found_goal= True
72             goalie = self.frontier[0]
73             break
```

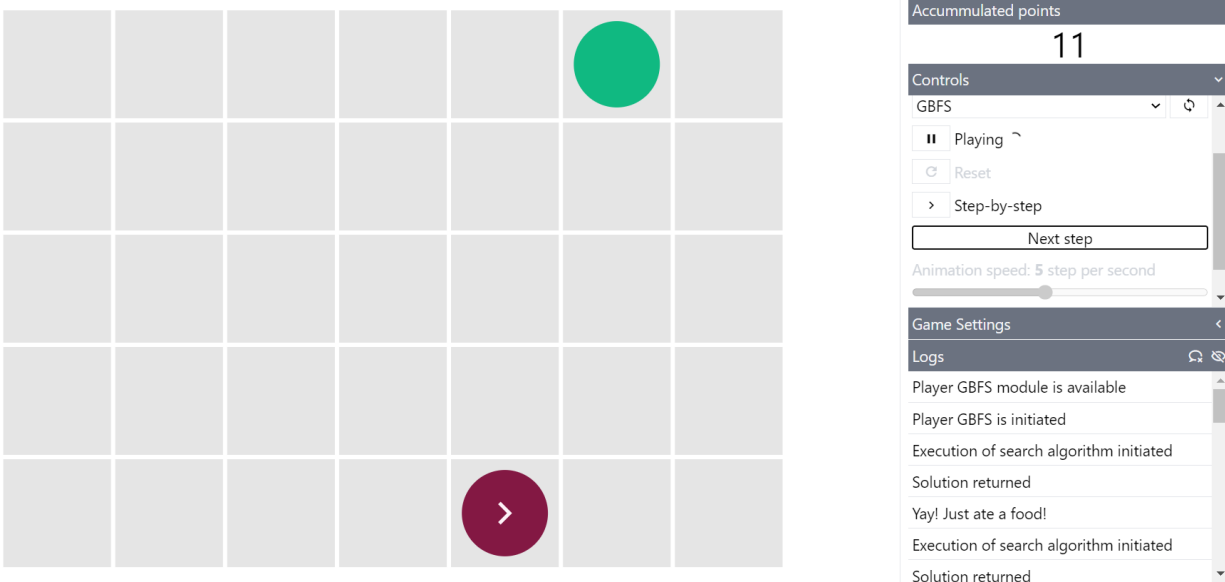
This process will keep looping until the child state is equal to the food location, hence the goal has been found. When the goal is found, the “found_goal” boolean variable will be checked to True and set the child node to the “goalie” variable to back to stop the while loop shown at line 68. Therefore, while the “found_goal” boolean variable is still False, it will continue to run the while loop and expand the next node in the frontier as well as the cost calculation and sort function.

```
107     path = [goalie.direction]
108     while goalie.parent is not None:
109         path.insert(0, goalie.direction)
110         for e in explored:
111             if e.state == goalie.parent:
112                 goalie = e
113                 break
114         del path[-1]
115         print(path)
116         solution = path
117
118     return solution, search_tree
```

Once the goal state is found, it will exit the while loop and proceed to extract the solution using the Node class. Similar to our Breadth-First search algorithm mentioned above.

- d. **Do the results achieve the aim of solving the problem (can you obtain the solution)?**

Yes. By running both of the Greedy Best First Search and Breadth First Search algorithms, the snake is able to reach 15 points and above. The initial snake location (red circle) is able to search the food location (green circle) based on its current location using the actions (north, west, east, south) to move one coordinate at a time until it reaches the goal node which is the food location. The snake location (red circle) is able to search the path by itself to reach the food location. Once the snake location (red circle) has reached the food location and eats the food, another food will be generated in the maze. While continuing with the loop, once a food is eaten by the snake, the accumulated points will plus 1 until the user stops it.



Uninformed Search (Breadth First Search)

The solution can be obtained, and the algorithm has shown the principles of the breadth first search algorithm where it expands the first node in the frontier, and it expand horizontal before going vertical.

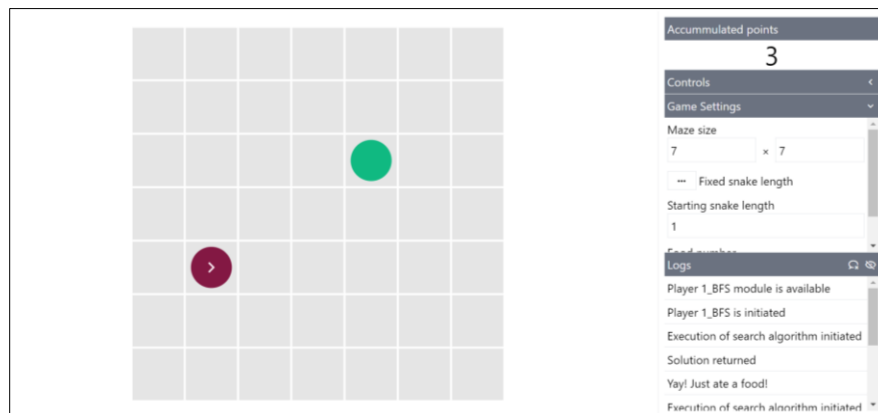


Figure 1: UI of the snake game before the Search Algorithm is initiated (Snake: [1,4]; Food: [4,2])

In the case shown above, the snake location is [1,4] as shown on the UI in figure 1 as well as the message on figure 2.1. Figure 2.1, Figure 2.2, Figure 2.3, and Figure 2.4 is the data printed on the Command Line Interface (CLI) and these data are directly printed from the code. As shown in Expansion Sequence 1 in Figure 2.1, the algorithm has expanded the node [1,4] and generated [1,3], [1,5], [0,4], and [2,4]. Then the initial node will be moved from the frontier to the explored list and the generated nodes will then be appended into the frontier after it has check that these states are not in the frontier or the explored list as shown in Expansion Sequence 1 in Figure 2.1.

Then, for the second expansion sequence as shown in Figure 2.1, the algorithm will expand the first node in the frontier, which is [1,3] as the node of [1,4] has been removed and moved to the explored list. The nodes generated from [1,3] is [1,2], [1,4], [0,3] and [2,3], in this case the [1,4] node will not be expanded as this state is a loopy path as it exists in the explored set. Hence, only [1,2], [0,3] and [2,3] will be expanded into the frontier.

For Breadth First Search Algorithm, it expands horizontally (breadth) instead of vertical (depth). Hence, it will not continue to expand on node [1,3] but it will expand node [1,5] as shown in Expansion Sequence 3 in Figure 2.1. Furthermore, to prove the algorithm is correct, the search tree in Figure 4 shows the visual representation of Breadth First Search with the Expansion Sequence. In this case, the Expansion Sequence 3 in Figure 2.1 shows that the node to be expanded is [1,5] and the search tree in Figure 3 also shows that Sequence 3 is the node to be expanded proves that it is using Breadth First Search algorithm.

This process will continue until the goal node is found as shown in Figure 2.2, Figure 2.3, and Figure 2.4. The Expansion Sequence stops at 24 because while expanding node [3,2], the goal node is found. The children generated from [3,2] are [3,1], [3,3,], [2,2] and [4,2] and [4,2] is the food location coordinates, hence it stops expanding and return the solution to the front end and the snakes moves towards the food as shown in Figure 3.

Screenshots that show it uses the Breadth First Search Algorithm to look for the solution and goal:

```
Anaconda Prompt (anaconda3) - conda activate snake-game - uvicorn serverapp
Data received from frontend {'purpose': 'next step', 'data': {'snake_locations': [[1, 4]], 'current': [[4, 2]]}
Expansion Sequence 1
Current Node: [1, 4]
Children of Current Node: [[1, 3], [1, 5], [0, 4], [2, 4]]
Direction: ['n', 's', 'w', 'e']
States in Frontier: [[1, 3], [1, 5], [0, 4], [2, 4]]

Expansion Sequence 2
Current Node: [1, 3]
Children of Current Node: [[1, 2], [1, 4], [0, 3], [2, 3]]
Direction: ['s', 'w', 'e', 'n', 'w', 'e']
States in Frontier: [[1, 5], [0, 4], [2, 4], [1, 2], [0, 3], [2, 3]]

Expansion Sequence 3
Current Node: [1, 5]
Children of Current Node: [[1, 4], [1, 6], [0, 5], [2, 5]]
Direction: ['w', 'e', 'n', 'w', 'e', 's', 'w', 'e']
States in Frontier: [[0, 4], [2, 4], [1, 2], [0, 3], [2, 3], [1, 6], [0, 5], [2, 5]]

Expansion Sequence 4
Current Node: [0, 4]
Children of Current Node: [[0, 3], [0, 5], [1, 4]]
Direction: ['e', 'n', 'w', 'e', 's', 'w', 'e']
States in Frontier: [[2, 4], [1, 2], [0, 3], [2, 3], [1, 6], [0, 5], [2, 5]]

Expansion Sequence 5
Current Node: [2, 4]
Children of Current Node: [[2, 3], [2, 5], [1, 4], [3, 4]]
Direction: ['n', 'w', 'e', 's', 'w', 'e', 'e']
States in Frontier: [[1, 2], [0, 3], [2, 3], [1, 6], [0, 5], [2, 5], [3, 4]]

Expansion Sequence 6
Current Node: [1, 2]
Children of Current Node: [[1, 1], [1, 3], [0, 2], [2, 2]]
Direction: ['w', 'e', 's', 'w', 'e', 'e', 'n', 'w', 'e']
States in Frontier: [[0, 3], [2, 3], [1, 6], [0, 5], [2, 5], [3, 4], [1, 1], [0, 2], [2, 2]]
```

Figure 2.1: Command Line Interface of BFS

```
Expansion Sequence 7
Current Node: [0, 3]
Children of Current Node: [[0, 2], [0, 4], [1, 3]]
Direction: ['e', 's', 'w', 'e', 'e', 'n', 'w', 'e']
States in Frontier: [[2, 3], [1, 6], [0, 5], [2, 5], [3, 4], [1, 1], [0, 2], [2, 2]]

Expansion Sequence 8
Current Node: [2, 3]
Children of Current Node: [[2, 2], [2, 4], [1, 3], [3, 3]]
Direction: ['s', 'w', 'e', 'e', 'n', 'w', 'e', 'e']
States in Frontier: [[1, 6], [0, 5], [2, 5], [3, 4], [1, 1], [0, 2], [2, 2], [3, 3]]

Expansion Sequence 9
Current Node: [1, 6]
Children of Current Node: [[1, 5], [0, 6], [2, 6]]
Direction: ['w', 'e', 'e', 'n', 'w', 'e', 'e', 'w', 'e']
States in Frontier: [[0, 5], [2, 5], [3, 4], [1, 1], [0, 2], [2, 2], [3, 3], [0, 6], [2, 6]]

Expansion Sequence 10
Current Node: [0, 5]
Children of Current Node: [[0, 4], [0, 6], [1, 5]]
Direction: ['e', 'e', 'n', 'w', 'e', 'e', 'w', 'e']
States in Frontier: [[2, 5], [3, 4], [1, 1], [0, 2], [2, 2], [3, 3], [0, 6], [2, 6]]

Expansion Sequence 11
Current Node: [2, 5]
Children of Current Node: [[2, 4], [2, 6], [1, 5], [3, 5]]
Direction: ['e', 'n', 'w', 'e', 'e', 'w', 'e', 'e']
States in Frontier: [[3, 4], [1, 1], [0, 2], [2, 2], [3, 3], [0, 6], [2, 6], [3, 5]]

Expansion Sequence 12
Current Node: [3, 4]
Children of Current Node: [[3, 3], [3, 5], [2, 4], [4, 4]]
Direction: ['n', 'w', 'e', 'e', 'w', 'e', 'e', 'e']
States in Frontier: [[1, 1], [0, 2], [2, 2], [3, 3], [0, 6], [2, 6], [3, 5], [4, 4]]
```

Figure 2.2: Command Line Interface of BFS (continue)

```

Expansion Sequence 13
Current Node: [1, 1]
Children of Current Node: [[1, 0], [1, 2], [0, 1], [2, 1]]
Direction: ['w', 'e', 'e', 'w', 'e', 'e', 'e', 'n', 'w', 'e']
States in Frontier: [[0, 2], [2, 2], [3, 3], [0, 6], [2, 6], [3, 5], [4, 4], [1, 0], [0, 1], [2, 1]]

Expansion Sequence 14
Current Node: [0, 2]
Children of Current Node: [[0, 1], [0, 3], [1, 2]]
Direction: ['e', 'e', 'w', 'e', 'e', 'e', 'n', 'w', 'e']
States in Frontier: [[2, 2], [3, 3], [0, 6], [2, 6], [3, 5], [4, 4], [1, 0], [0, 1], [2, 1]]

Expansion Sequence 15
Current Node: [2, 2]
Children of Current Node: [[2, 1], [2, 3], [1, 2], [3, 2]]
Direction: ['e', 'w', 'e', 'e', 'e', 'n', 'w', 'e', 'e']
States in Frontier: [[3, 3], [0, 6], [2, 6], [3, 5], [4, 4], [1, 0], [0, 1], [2, 1], [3, 2]]

Expansion Sequence 16
Current Node: [3, 3]
Children of Current Node: [[3, 2], [3, 4], [2, 3], [4, 3]]
Direction: ['w', 'e', 'e', 'e', 'n', 'w', 'e', 'e', 'e']
States in Frontier: [[0, 6], [2, 6], [3, 5], [4, 4], [1, 0], [0, 1], [2, 1], [3, 2], [4, 3]]

Expansion Sequence 17
Current Node: [0, 6]
Children of Current Node: [[0, 5], [1, 6]]
Direction: ['e', 'e', 'e', 'n', 'w', 'e', 'e', 'e']
States in Frontier: [[2, 6], [3, 5], [4, 4], [1, 0], [0, 1], [2, 1], [3, 2], [4, 3]]

Expansion Sequence 18
Current Node: [2, 6]
Children of Current Node: [[2, 5], [1, 6], [3, 6]]
Direction: ['e', 'e', 'n', 'w', 'e', 'e', 'e', 'e', 'e']
States in Frontier: [[3, 5], [4, 4], [1, 0], [0, 1], [2, 1], [3, 2], [4, 3], [3, 6]]

Expansion Sequence 19
Current Node: [3, 5]
Children of Current Node: [[3, 4], [3, 6], [2, 5], [4, 5]]
Direction: ['e', 'n', 'w', 'e', 'e', 'e', 'e', 'e']
States in Frontier: [[4, 4], [1, 0], [0, 1], [2, 1], [3, 2], [4, 3], [3, 6], [4, 5]]

Expansion Sequence 20
Current Node: [4, 4]
Children of Current Node: [[4, 3], [4, 5], [3, 4], [5, 4]]
Direction: ['n', 'w', 'e', 'e', 'e', 'e', 'e', 'e']
States in Frontier: [[1, 0], [0, 1], [2, 1], [3, 2], [4, 3], [3, 6], [4, 5], [5, 4]]

Expansion Sequence 21
Current Node: [1, 0]
Children of Current Node: [[1, 1], [0, 0], [2, 0]]
Direction: ['w', 'e', 'e', 'e', 'e', 'e', 'e', 'w', 'e']
States in Frontier: [[0, 1], [2, 1], [3, 2], [4, 3], [3, 6], [4, 5], [5, 4], [0, 0], [2, 0]]

Expansion Sequence 22
Current Node: [0, 1]
Children of Current Node: [[0, 0], [0, 2], [1, 1]]
Direction: ['e', 'e', 'e', 'e', 'e', 'e', 'w', 'e']
States in Frontier: [[2, 1], [3, 2], [4, 3], [3, 6], [4, 5], [5, 4], [0, 0], [2, 0]]

Expansion Sequence 23
Current Node: [2, 1]
Children of Current Node: [[2, 0], [2, 2], [1, 1], [3, 1]]
Direction: ['e', 'e', 'e', 'e', 'e', 'w', 'e', 'e']
States in Frontier: [[3, 2], [4, 3], [3, 6], [4, 5], [5, 4], [0, 0], [2, 0], [3, 1]]

Expansion Sequence 24
Current Node: [3, 2]
Children of Current Node: [[3, 1], [3, 3], [2, 2], [4, 2]]
Direction: ['e', 'e', 'e', 'e', 'w', 'e', 'e', 'e']
States in Frontier: [[4, 3], [3, 6], [4, 5], [5, 4], [0, 0], [2, 0], [3, 1], [4, 2]]

```

Figure 2.3: Command Line Interface of BFS (Continue)

```

Expansion Sequence 20
Current Node: [4, 4]
Children of Current Node: [[4, 3], [4, 5], [3, 4], [5, 4]]
Direction: ['n', 'w', 'e', 'e', 'e', 'e', 'e', 'e']
States in Frontier: [[1, 0], [0, 1], [2, 1], [3, 2], [4, 3], [3, 6], [4, 5], [5, 4]]

Expansion Sequence 21
Current Node: [1, 0]
Children of Current Node: [[1, 1], [0, 0], [2, 0]]
Direction: ['w', 'e', 'e', 'e', 'e', 'e', 'e', 'w', 'e']
States in Frontier: [[0, 1], [2, 1], [3, 2], [4, 3], [3, 6], [4, 5], [5, 4], [0, 0], [2, 0]]

Expansion Sequence 22
Current Node: [0, 1]
Children of Current Node: [[0, 0], [0, 2], [1, 1]]
Direction: ['e', 'e', 'e', 'e', 'e', 'e', 'w', 'e']
States in Frontier: [[2, 1], [3, 2], [4, 3], [3, 6], [4, 5], [5, 4], [0, 0], [2, 0]]

Expansion Sequence 23
Current Node: [2, 1]
Children of Current Node: [[2, 0], [2, 2], [1, 1], [3, 1]]
Direction: ['e', 'e', 'e', 'e', 'e', 'w', 'e', 'e']
States in Frontier: [[3, 2], [4, 3], [3, 6], [4, 5], [5, 4], [0, 0], [2, 0], [3, 1]]

Expansion Sequence 24
Current Node: [3, 2]
Children of Current Node: [[3, 1], [3, 3], [2, 2], [4, 2]]
Direction: ['e', 'e', 'e', 'e', 'w', 'e', 'e', 'e']
States in Frontier: [[4, 3], [3, 6], [4, 5], [5, 4], [0, 0], [2, 0], [3, 1], [4, 2]]

```

Figure 2.4: Command Line Interface of BFS

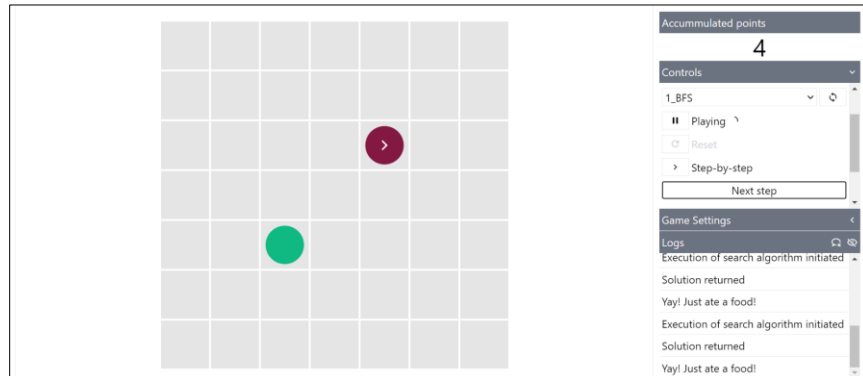


Figure 3: UI of the snake game (Solution returned and food eaten)

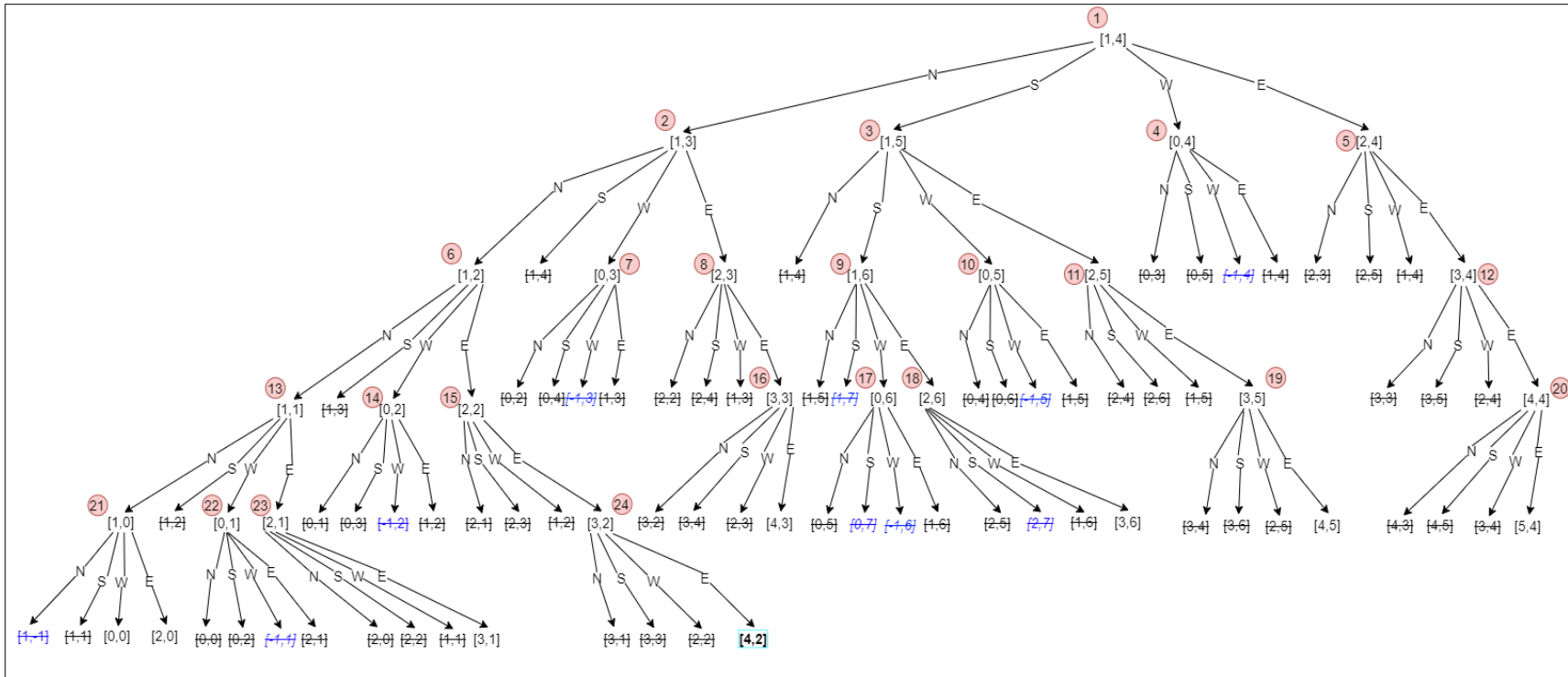


Figure 4: Search Tree of BFS

Informed Search (Greedy Best First Search)

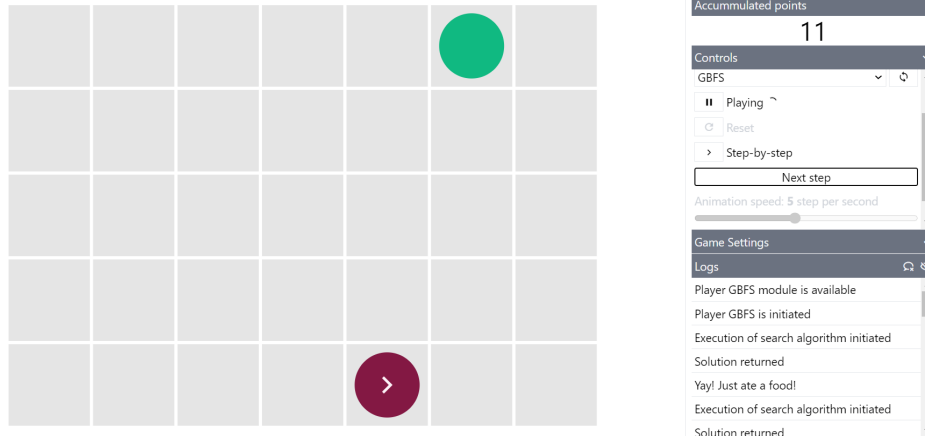


Figure 5: Snake Game UI with Initial Node = [4,4], goal node= [5,0]

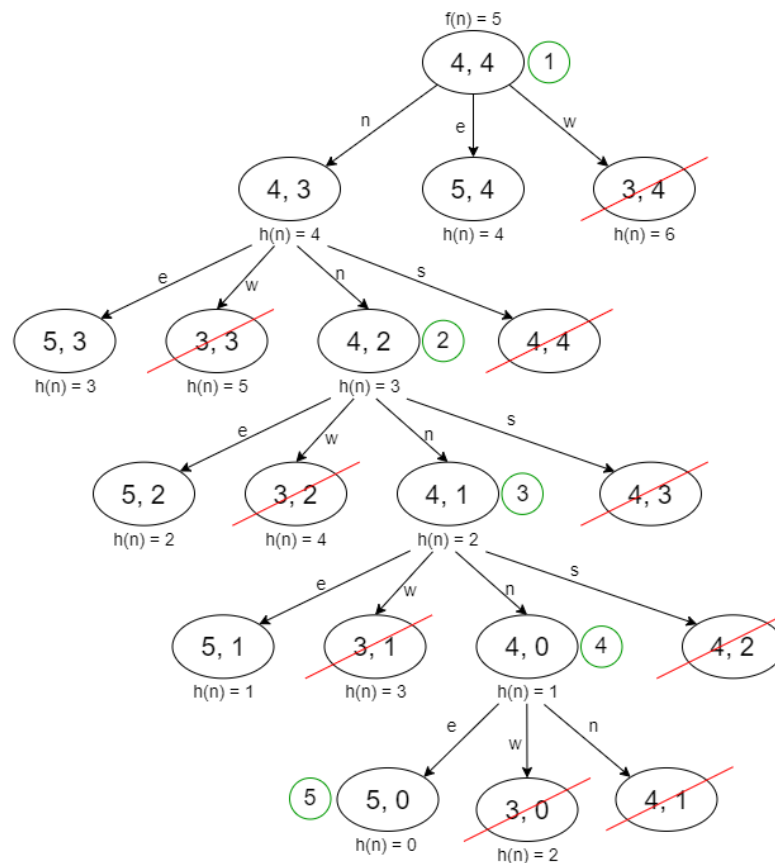


Figure 6: Search Tree of Greedy Best-First Search

```

Data received from frontend {'purpose': 'next step', 'data': {'snake_locations': [[4, 4]], 'current_direction': 'e', 'food_locations': [[5, 0]]}}
expansion sequence: 1
parent: [4, 4]
children: [[4, 3], [3, 4], [5, 4]]
frontier: [[4, 3], [5, 4], [3, 4]]
cost: [4, 4, 6]
direction: ['n', 'e', 'w']

expansion sequence: 2
parent: [4, 3]
children: [[4, 2], [4, 4], [3, 3], [5, 3]]
frontier: [[4, 2], [5, 3], [5, 4], [3, 3], [3, 4]]
cost: [3, 3, 4, 5, 6]
direction: ['n', 'e', 'e', 'w', 'w']

expansion sequence: 3
parent: [4, 2]
children: [[4, 1], [4, 3], [3, 2], [5, 2]]
frontier: [[4, 1], [5, 2], [5, 3], [5, 4], [3, 2], [3, 3], [3, 4]]
cost: [2, 2, 3, 4, 5, 6]
direction: ['n', 'e', 'e', 'e', 'w', 'w', 'w']

```

Figure 7.1: Greedy Best-First Search Command Line Interface

```

expansion sequence: 4
parent: [4, 1]
children: [[4, 0], [4, 2], [3, 1], [5, 1]]
frontier: [[4, 0], [5, 1], [5, 2], [5, 3], [3, 1], [5, 4], [3, 2], [3, 3], [3, 4]]
cost: [1, 1, 2, 3, 3, 4, 4, 5, 6]
direction: ['n', 'e', 'e', 'e', 'w', 'e', 'w', 'w', 'w']

expansion sequence: 5
parent: [4, 0]
children: [[4, 1], [3, 0], [5, 0]]
frontier: [[5, 0], [5, 1], [5, 2], [3, 0], [5, 3], [3, 1], [5, 4], [3, 2], [3, 3], [3, 4]]
cost: [0, 1, 2, 2, 3, 3, 4, 4, 5, 6]
direction: ['e', 'e', 'e', 'w', 'e', 'w', 'e', 'w', 'w', 'w']

['n', 'n', 'n', 'n', 'e']

```

Figure 7.2: Greedy Best-First Search Command Line Interface

The initial node which is the snake location at the beginning is at the node [4, 4] and the goal node which is the food location at is at the node [5,0]. At the node [4, 4], it has the estimating promise path of 5 nodes in the heuristic evaluation function $f(n)$ as the information gathered by the search up to the final node. The node [4,4] has 3 children nodes that can be the next node which are [4,3], [5,4] and [3,4]. If moving with the action “n”, the next node will be [4,3] with the heuristic value of 4, while moving with the action “e”, the next node will be [5,4] with the heuristic value of 4 and moving with the action “w”, the next node will be [3,4] with the heuristic value of 6. Therefore, the node [3,4] will be ignored due to the higher heuristic value. However, although both the nodes [4,3] and [5,4] have the exact same heuristic value of 4, [5,4] it is still ignored due the fact that the algorithm would only expand to the first node of the frontier, and after sorting, [4,3] has appeared to be placed in front of [5,4] as shown in figure 7.1. Therefore, the algorithm has chosen node [4,3].

When the snake has moved with the action “n” from the node [4,4] to the node [4,3], the node [4,3] has 4 children nodes and they are [5,3], [3,3], [4,2] and [4,4], and each of their heuristic value are 3, 5, 3 and 5. Due to the fact that child node [4,4] is a loopy path, it will be ignored as well as child node [3,3] with the highest heuristic value among other children nodes. Child node [4,2] has been chosen to be expanded instead of [5,3] because it is placed as the first node in the frontier with the lowest cost. With the same theory, the search tree will keep expanding until it reaches

the food location, goal node [5,0]. When the snake reaches the goal node, the snake location will become an initial node, ready to search for the next food location (goal node).

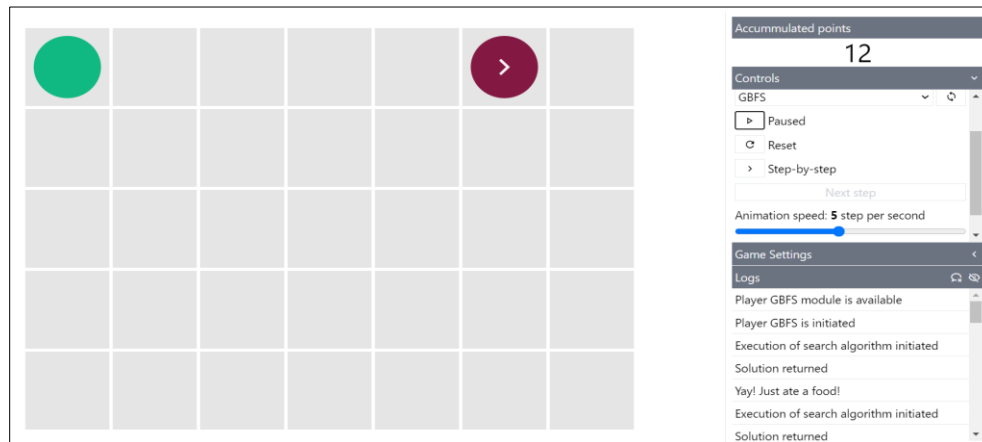


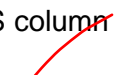
Figure 8: UI of the snake game (Solution returned and food eaten)

e. How is the performance of the algorithms (how good are the solutions)?

Both of the search algorithms did return the solution well by moving the snake to look for food and increase the point after eating it. However, the performance of the Greedy Best First Search (GBFS) is better than the performance of Breadth First Search (BFS) as Greedy Best First Search expands the node that is closest to the goal and this helps minimise the search cost. Furthermore, GBFS uses the heuristic function to find the closest path to the goal node,, hence it makes the search process more efficient.

On the contrary, the Breadth First Search algorithm is a blind search, it will keep on expanding the node without the knowledge of whether the generated node is closer to the goal. Furthermore, it will keep on expanding the shallowest node one by one regardless of the search cost and only stops when a goal is found. Therefore, the performance of Breadth First Search will be poor in a large state space as compared to informed search algorithms (GBFS) because it may continue to expand the nodes that are further away from the current state to the goal state.

An example is shown in table 1, where we run the BFS and GBFS algorithm on the same maze size [6x6], same food location [2,0] and the same snake location [0,3]. The result shows that the GBFS column shows that it expanded 5 times whereas the BFS column shows that it expanded 16 times to get the search for the goal node.



Greedy Best First Search

```
Data received from frontend {'purpose': 'setup', 'data': {'maze_size': [6, 6], 'static_snake_length': True}}
Data received from frontend {'purpose': 'next step', 'data': {'snake_locations': [[0, 3]], 'current_direction': 'e', 'food_locations': [[2, 0]]}}
expansion sequence: 1
parent: [0, 3]
children: [[0, 2], [0, 4], [1, 3]]
frontier: [[0, 2], [1, 3], [0, 4]]
cost: [4, 4, 6]
direction: ['n', 'e', 's']

expansion sequence: 2
parent: [0, 2]
children: [[0, 1], [0, 3], [1, 2]]
frontier: [[0, 1], [1, 2], [1, 3], [0, 4]]
cost: [3, 3, 4, 6]
direction: ['n', 'e', 'e', 's']

expansion sequence: 3
parent: [0, 1]
children: [[0, 0], [0, 2], [1, 1]]
frontier: [[0, 0], [1, 1], [1, 2], [1, 3], [0, 4]]
cost: [2, 2, 3, 4, 6]
direction: ['n', 'e', 'e', 'e', 's']

expansion sequence: 4
parent: [0, 0]
children: [[0, 1], [1, 0]]
frontier: [[1, 0], [1, 1], [1, 2], [1, 3], [0, 4]]
cost: [1, 2, 3, 4, 6]
direction: ['e', 'e', 'e', 'e', 's']

expansion sequence: 5
parent: [1, 0]
children: [[1, 1], [0, 0], [2, 0]]
frontier: [[2, 0], [1, 1], [1, 2], [1, 3], [0, 4]]
cost: [0, 2, 3, 4, 6]
direction: ['e', 'e', 'e', 'e', 's']

['n', 'n', 'n', 'e', 'e']
```

Breadth First Search

```
Data received from frontend {'purpose': 'next step', 'data': {'snake_locations': [[0, 3]], 'current_direction': 'e', 'food_locations': [[2, 0]]}}
Expansion Sequence 1
Current Node: [0, 3]
Children of Current Node: [[0, 2], [0, 4], [1, 3]]
Direction: ['n', 's', 'e']
States in Frontier: [[0, 2], [0, 4], [1, 3]]

Expansion Sequence 2
Current Node: [0, 2]
Children of Current Node: [[0, 1], [0, 3], [1, 2]]
Direction: ['s', 'e', 'n', 'e']
States in Frontier: [[0, 4], [1, 3], [0, 1], [1, 2]]

Expansion Sequence 3
Current Node: [0, 4]
Children of Current Node: [[0, 3], [0, 5], [1, 4]]
Direction: ['e', 'n', 'e', 's', 'e']
States in Frontier: [[1, 3], [0, 1], [1, 2], [0, 5], [1, 4]]

Expansion Sequence 4
Current Node: [1, 3]
Children of Current Node: [[1, 2], [1, 4], [0, 3], [2, 3]]
Direction: ['n', 'e', 's', 'e', 'e']
States in Frontier: [[0, 1], [1, 2], [0, 5], [1, 4], [2, 3]]

Expansion Sequence 5
Current Node: [0, 1]
Children of Current Node: [[0, 0], [0, 2], [1, 1]]
Direction: ['e', 's', 'e', 'e', 'n', 'e']
States in Frontier: [[1, 2], [0, 5], [1, 4], [2, 3], [0, 0], [1, 1]]

Expansion Sequence 6
Current Node: [1, 2]
Children of Current Node: [[1, 1], [1, 3], [0, 2], [2, 2]]
Direction: ['s', 'e', 'e', 'n', 'e', 'e']
States in Frontier: [[0, 5], [1, 4], [2, 3], [0, 0], [1, 1], [2, 2]]

Expansion Sequence 7
Current Node: [0, 5]
Children of Current Node: [[0, 4], [1, 5]]
Direction: ['e', 'e', 'n', 'e', 'e', 'e']
States in Frontier: [[1, 4], [2, 3], [0, 0], [1, 1], [2, 2], [1, 5]]

Expansion Sequence 8
Current Node: [1, 4]
Children of Current Node: [[1, 3], [1, 5], [0, 4], [2, 4]]
Direction: ['e', 'n', 'e', 'e', 'e', 'e']
States in Frontier: [[2, 3], [0, 0], [1, 1], [2, 2], [1, 5], [2, 4]]

Expansion Sequence 9
Current Node: [2, 3]
Children of Current Node: [[2, 2], [2, 4], [1, 3], [3, 3]]
Direction: ['n', 'e', 'e', 'e', 'e', 'e']
States in Frontier: [[0, 0], [1, 1], [2, 2], [1, 5], [2, 4], [3, 3]]

Expansion Sequence 10
Current Node: [0, 0]
Children of Current Node: [[0, 1], [1, 0]]
Direction: ['e', 'e', 'e', 'e', 'e', 'e']
States in Frontier: [[1, 1], [2, 2], [1, 5], [2, 4], [3, 3], [1, 0]]

Expansion Sequence 11
Current Node: [1, 1]
Children of Current Node: [[1, 0], [1, 2], [0, 1], [2, 1]]
Direction: ['e', 'e', 'e', 'e', 'e', 'e']
States in Frontier: [[2, 2], [1, 5], [2, 4], [3, 3], [1, 0], [2, 1]]

Expansion Sequence 12
Current Node: [2, 2]
Children of Current Node: [[2, 1], [2, 3], [1, 2], [3, 2]]
Direction: ['e', 'e', 'e', 'e', 'e', 'e']
States in Frontier: [[1, 5], [2, 4], [3, 3], [1, 0], [2, 1], [3, 2]]
```

	<div>Expansion Sequence 13 Current Node: [1, 5] Children of Current Node: [[1, 4], [0, 5], [2, 5]] Direction: ['e', 'e', 'e', 'e', 'e', 'e'] States in Frontier: [[2, 4], [3, 3], [1, 0], [2, 1], [3, 2], [2, 5]]</div> <div>Expansion Sequence 14 Current Node: [2, 4] Children of Current Node: [[2, 3], [2, 5], [1, 4], [3, 4]] Direction: ['e', 'e', 'e', 'e', 'e', 'e'] States in Frontier: [[3, 3], [1, 0], [2, 1], [3, 2], [2, 5], [3, 4]]</div> <div>Expansion Sequence 15 Current Node: [3, 3] Children of Current Node: [[3, 2], [3, 4], [2, 3], [4, 3]] Direction: ['e', 'e', 'e', 'e', 'e', 'e'] States in Frontier: [[1, 0], [2, 1], [3, 2], [2, 5], [3, 4], [4, 3]]</div> <div>Expansion Sequence 16 Current Node: [1, 0] Children of Current Node: [[1, 1], [0, 0], [2, 0]] Direction: ['e', 'e', 'e', 'e', 'e', 'e'] States in Frontier: [[2, 1], [3, 2], [2, 5], [3, 4], [4, 3], [2, 0]]</div>
--	---

Table 1: GBFS and BFS Performance Comparison.