SCHOOL OF SCIENCE AND TECHNOLOGY

CSC3206: ARTIFICIAL INTELLIGENCE

ASSIGNMENT 1 REPORT

GROUP:

ISLAND CODERS

GROUP MEMBERS:

MUHAMMAD NABEEL RASHEED VARSALLY 19025634

RAVEESH SHIBCHURN 19053156

ELTIGANI HAMADELNIEL ELFATIH 17087610

YUTASHNA GUNNOO 19063304

SNAKE GAME SOLUTION THROUGH AI SEARCH ALGORITHMS

Abstract — In this report, we will present informed and uninformed search algorithms that are being implemented as agents in an automated snake game. Greedy best-first search and Breadth-first search are the algorithms we selected for informed and uninformed search algorithms respectively. In this report, we will describe and explain the main problems we encountered , and will evaluate the performance of each algorithm.

INTRODUCTION

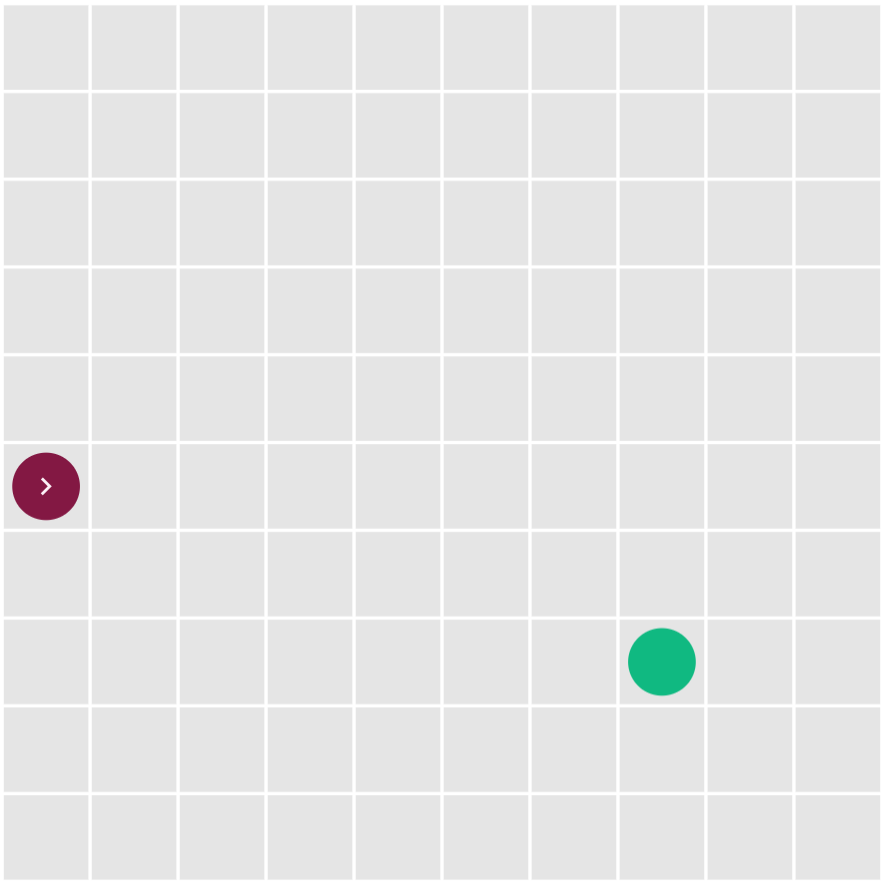
Snake is a rather common video game that dates back many decades. Our project aimed to create and test agents using both informed and uninformed search algorithms to solve the game. The execution of the snake game is such that the snake, which is our agent, must search for its food in a grid. The chosen algorithm for informed search was Greedy Best First Search while for uninformed, Breadth First Search.

Additionally, one important changing feature of the snake is its length - which can either be static or dynamic. If the snake has a static length, the amount of food eaten will not affect the size of the snake. However, if it has a dynamic length, the snake will be longer by one unit with every food eaten. Both agents, from each search algorithm, should be designed to cater for these two cases.

This report covers each aspect of the project, starting from the problem description, in section I, to describing the approach of the selected algorithms, in section II. The results and discussions of each approach are also included in section III of the report.

SECTION I: PROBLEM DESCRIPTION

Before we can further breakdown and explore the search algorithms applied in detail, we must first have a complete understanding of the problem definition and the possible actions that the agent can perform.

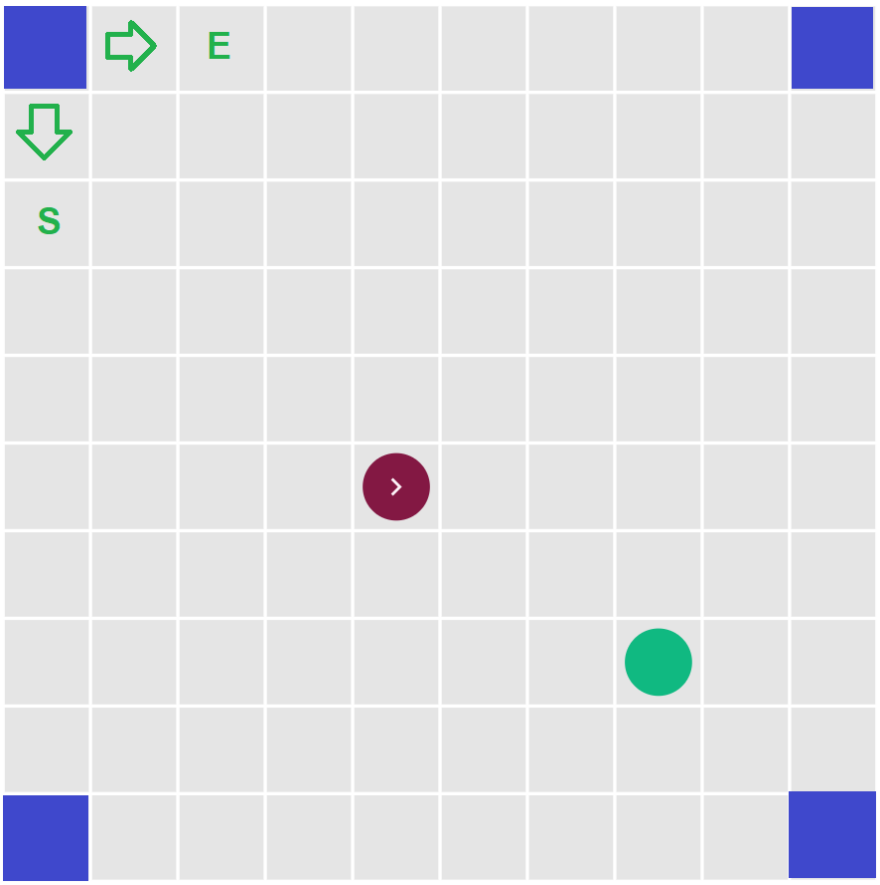


Our initial state consists of a maze with 10 rows and 10 columns with the initial state of the snake being [5,0] and the food that is randomly generated.

Each time a node is expanded it has a minimum of 2 children and a maximum of 4 children depending on the snake's location in the grid.

The following are visual examples that portray the different positions on the grid that when expanded will generate 2,3 or 4 children as well as the actions that need to be performed to reach these goals.

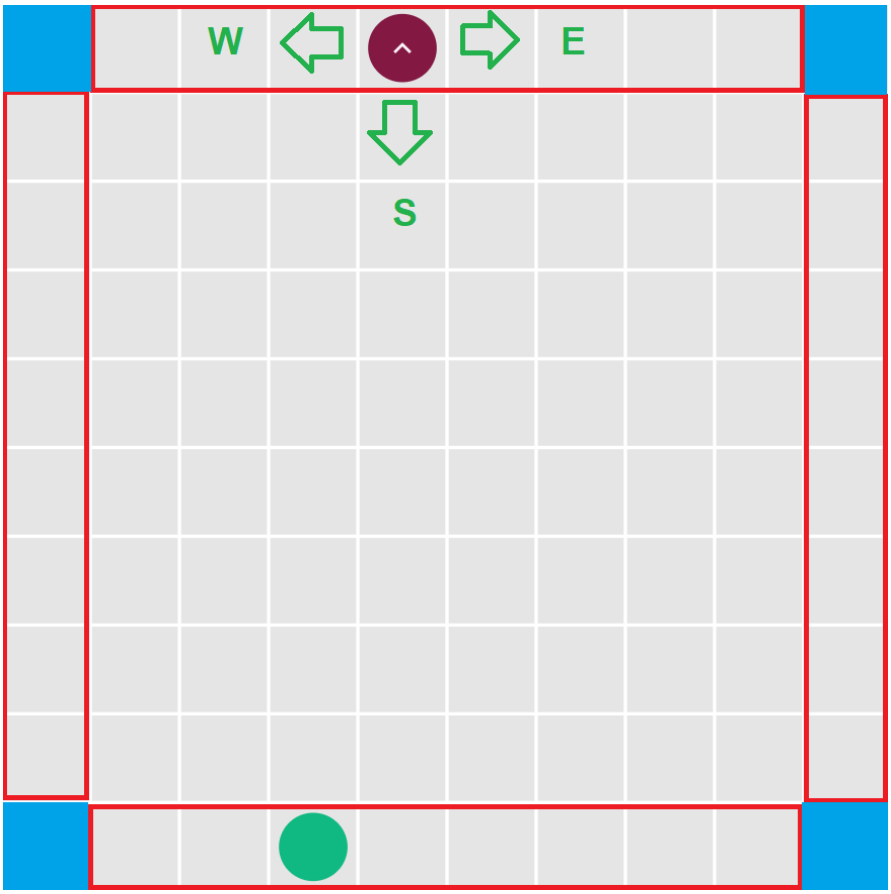
2 Children:



The four corners within the grid highlighted in blue can only generate 2 children. These are generated through the following actions:

- Top left corner: East and South
- Bottom left corner: East and North
- Top right corner: West and South
- Bottom right corner: East and North

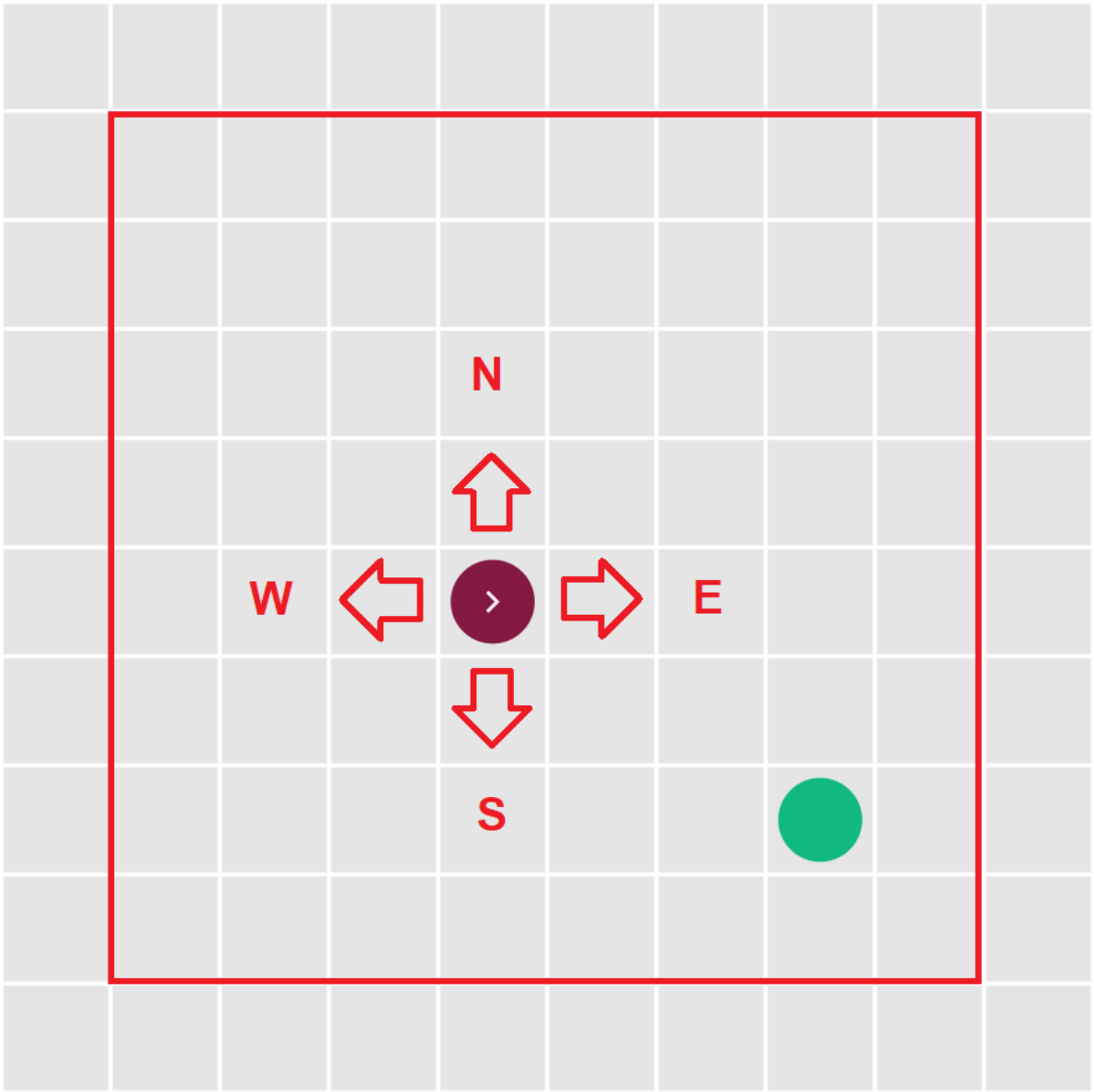
3 Children:



The tiles around the corners at the 4 edges of the maze highlighted in the red border can generate 3 children. These are generated through the following actions:

- East
- South
- West

4 Children:



This is the most common scenario in which 4 children can be generated through the following actions:

- North
- East
- South
- West

Now that we have visualized all the possible locations along with the corresponding number of children that can be generated from those states, we can dive deeper into the details of the algorithms applied.

Considering the agent that should follow uninformed search:

In this scenario, the agent is unaware of the location of the food and therefore the agent must explore all the possible actions until the goal state is generated. In this strategy, the agent essentially performs a blind search, meaning that the agent has no additional information about the states beyond that of the problem definition. Which in this case are the snake's initial position or state, its current direction, the size of the maze and the goal state i.e. the food which is also generated randomly.

Considering the agent that should follow informed search:

In this case, the snake knows where the food is located and should move to the said destination. Using informed search, the agent has the knowledge about the coordinate of the food location. Thus, the algorithm implemented will calculate the distance between the goal and the node and move to the closest food available.

SECTION II: ALGORITHMS

UNINFORMED: Breadth-first search:

Breadth-first search can be defined as a search algorithm or strategy in which the root node or parent is expanded first, then all the children of the parent node are expanded next, their children and so on. Each time a node is generated a goal test is performed to check whether the node is the goal node or not. The algorithm is terminated when the goal node is generated.

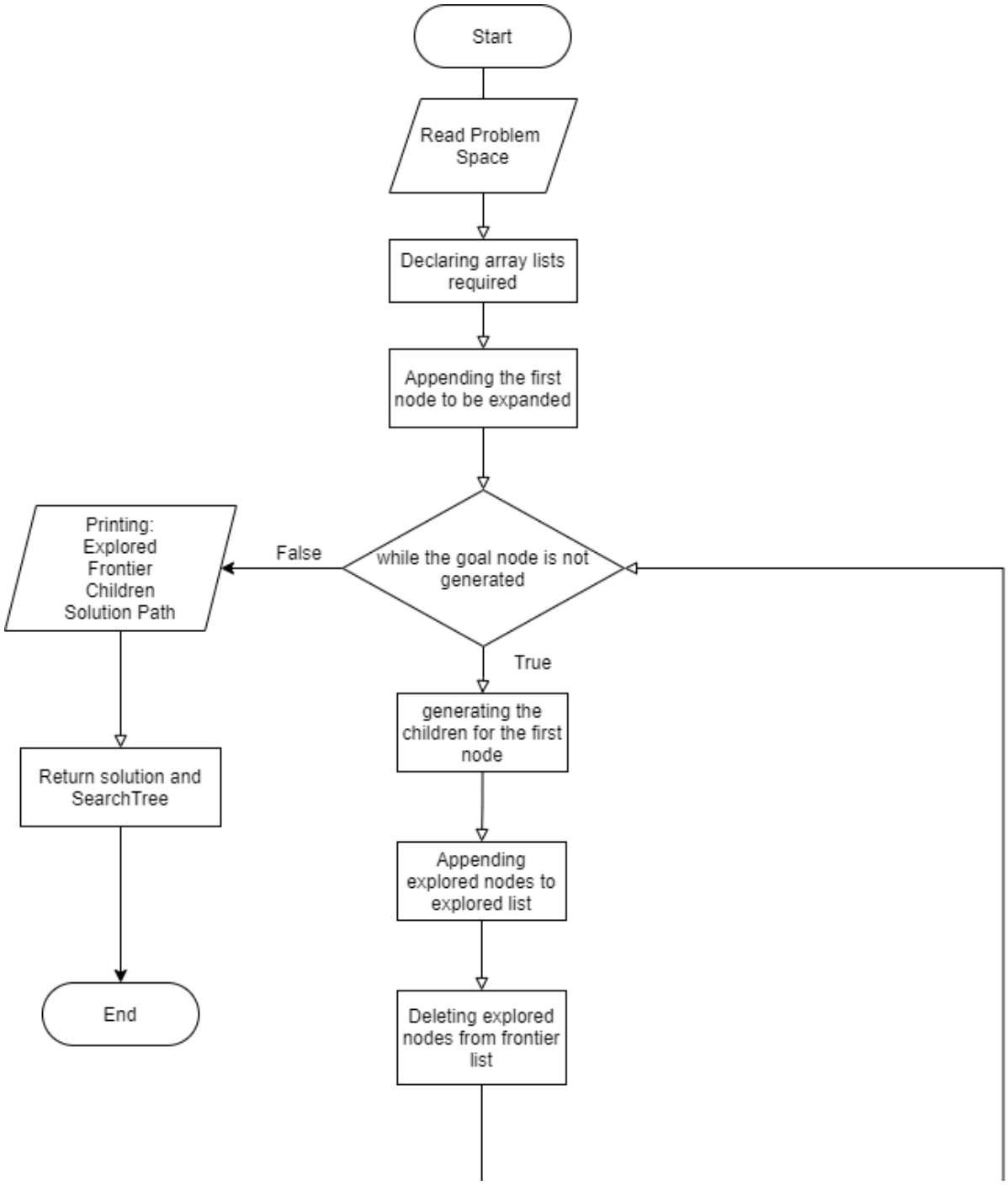
Since we are applying a search algorithm to generate a solution path for the snake to reach the goal, it is important to highlight that the solution must consist of the series of actions or moves that the agent must make to move the snake to the food. Therefore, when generating a child node from the parent node we must also take note of the action required to go from the parent node to the child node. Furthermore, when generating children, we must take note of the snake locations, specifically the snake's body such that if a child state is already occupied with the snake's body, that child is not generated.

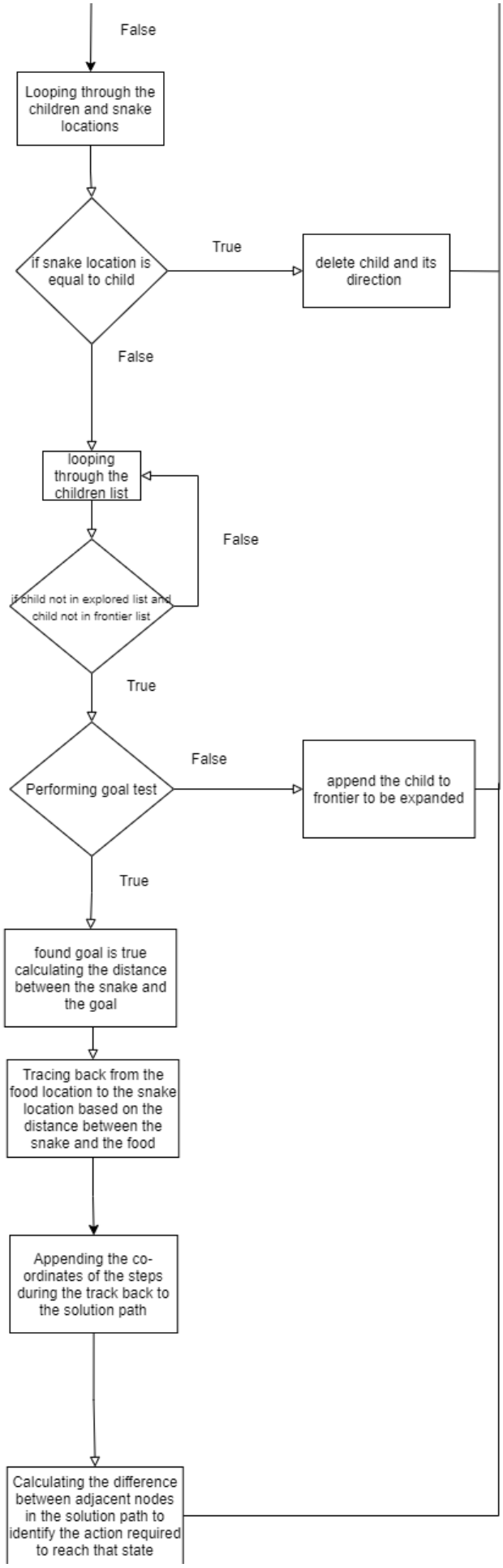
It is important to also understand how and when nodes are selected for expansion within our algorithm, since we are applying a breadth-first search horizontal node take priority in the expansion order before vertical nodes this because we need to expand all the children's nodes of a parent node before expanding before further expanding the children of the child node.

In our case when children are generated, they generate a clockwise manner of direction from north to east to south to west. Hence the child node with the direction of north is always expanded first followed by east and so on. Furthermore, when a child node is expanded a goal test is performed which compares the location of the child node to the food location on the grid. If they align then we have succeeded our goal test and the algorithm will assign the solution path and direction and terminate.

Presented below is the flow chart of the algorithm that provides an overview of the logic and approach taken for the algorithm.

UNINFORMED ALGORITHM FLOWCHART:




False
Looping through the children and snake locations
if snake location is equal to child
True
delete child and its direction
False
looping through the children list
False
if child not in explored list and child not in frontier list
True
Performing goal test
False
append the child to frontier to be expanded
True
found goal is true calculating the distance between the snake and the goal
Tracing back from the food location to the snake location based on the distance between the snake and the food
Appending the co-ordinates of the steps during the track back to the solution path
Calculating the difference between adjacent nodes in the solution path to identify the action required to reach that state

Coding implementation:

Our BFS algorithm contains of 3 processes that have been implemented as functions within the code and help us achieve the desired outcome:

1. Generating children of the parent node
2. Getting the direction of all the adjacent nodes in the solution path
3. Generating the Search Tree

Generating child nodes:

This is implemented as the function getChildren() in our algorithm and is defined to take in two parameters **X and Y** which are the coordinates of the nodes we would like to generate children for. Within this function we initialize two arrays' children and children direction which will be used to store the children and direction once generated. In addition, the maximum number of rows and columns are defined, and these are used along with the x and y parameters to perform checks on whether the parent node can generate 2 ,3 or 4 children. We use conditional if and else statements to identify the position of the node in the grid and based on this we can append the number of children and their direction. For example, if the location is any of the 4 corners of the grid it will have 2 children if it is within the top or bottom rows or the left most and right most columns it will have 3 children else it has 4 children.

```
if x == xx:
    if y == 0:
        pass
    elif y == yy:
        children.append([xx, yy - 1])  # N
        children_direction.append("n")
        children.append([xx - 1, yy])  # W
        children_direction.append("w")
    else:
        children.append([x, y - 1])  # N
        children_direction.append("n")
        children.append([x, y + 1])  # S
        children_direction.append("s")
        children.append([x - 1, y])  # W
        children_direction.append("w")
if y == yy:
    if x == 0:
        pass
    elif x == xx:
        pass
    else:
        children.append([x, y - 1])  # N
        children_direction.append("n")
        children.append([x + 1, y])  # E
        children_direction.append("e")
        children.append([x - 1, y])  # W
        children_direction.append("w")
if (0 < x < xx) & (0 < y < yy):
    children.append([x, y - 1])  # N
    children_direction.append("n")
    children.append([x + 1, y])  # E
    children_direction.append("e")
    children.append([x, y + 1])  # S
    children_direction.append("s")
    children.append([x - 1, y])  # W
    children_direction.append("w")

return children, children_direction
```

Generating the direction of all nodes in solution path:

When the agent moves the snake, the agent requires only the direction of the actions required to move from one state to another as it approaches the goal rather than the coordinates. After finding coordinates of the child nodes, we append it in the solution path array. From this array, we need to find the series of actions that the snake needs to do to reach the goal node. Thus, using the solution path array, we subtract the adjacent nodes to generate an array of directions (solution) to reach the food.

```
def getsolutionDirection(self,path_x,path_y,sol_x,sol_y):
    r1 = sol_x - path_x
    r2 = sol_y - path_y
    if r1 == 1:
      return "e"
    if r1 == -1:
      return "W"
    if r2 == 1:
        return "s"
    if r2 == -1:
      return "n"
```

Generating the Search Tree:

To generate the search tree, dictionaries are used. Since in both of our algorithms, the snake_locations were used as coordinates, it had to be converted to string using this line of code:

```
parent_state = str(px) + "," + str(py)
```

The coding for the search tree was done in two parts - one for the root node (has no parent) and one for all the successive nodes that have parents.

Explaining the approach for the successive nodes

```
parent_state = str(px) + "," + str(py)
# if the node is not the first one to be expanded and has a parent which is already in the search tree
if len(Player.search_tree) != 0:
    for st in Player.search_tree:
        if  st["state"] == parent_state:
            st["actions"] = children_direction
            st["expansionsequence"] = Player.xpansionsequence
            for c in children:
                child_state = str(c[0]) + "," + str(c[1])
                Player.search_tree.append(
                    {"id": Player.ids, "state": parent_state, "expansionsequence": -1, "children": [],
                     "actions": [], "removed": False, "parent": st["id"]})
                st["children"].append(Player.ids)
                Player.ids += 1
```

Id is a unique identification for every node. This starts with one and will keep on incrementing as the expansion happens. The id and the xpansionsequence are attributes of our Player Class.

Provided that the node has a parent, the parent_state, id and xpansionsequence are appended to the search tree. However, at this point, the children and their directions are not known yet. Thus, they are empty at the beginning but as the expansion proceeds, the details are updated in the search tree. After adding one child to the search tree, ids are incremented for the next children.

Explaining the approach for the root node

```
# if the node is the first one to be expanded and has no parent
else:

    first_node = {"id": Player.ids, "state": current_state,
                  "expansionsequence": Player.xpansionsequence,
                  "children": [], "actions": children_direction, "removed": False, "parent": None}
    Player.search_tree.append(first_node)
    Player.ids += 1
    for c in children:
        child_state = str(c[0]) + "," + str(c[1])
        Player.search_tree.append(
            {"id": Player.ids, "state": parent_state, "expansionsequence": -1, "children": [], "actions": [],
             "removed": False, "parent": first_node["id"]})
        first_node["children"].append(Player.ids)
        Player.ids += 1
```

This part of the code starts by appending the details of the node with the snake_location in the search tree. The rest of the code follows the same concept as explained for the successive nodes.

In addition to our function defined above, we run the function within our code that performs the actual running of our BFS search algorithm within our code implementation. This function takes in the **problem space** as its only parameter. Within this function we initialize the array lists required specifically the solution, frontier, frontier direction, explored, explored direction and the solution path. Since our first node is the current location of the snake we append that node along with the current direction that the snake is facing to our frontier list and frontier direction list respectively.

```
def run(self, problem):
    problem = {
      snake_locations: [[int,int],[int,int],...],
      current_direction: str,
      food_locations: [[int,int],[int,int],...],
    }

   #declaring required variables
   found_goal = False
   solution = []
   frontier = []
   frontier_direction = []
   explored = []
   explored_direction = []
   solution_path = []


   #appending initial state
   frontier.append(problem["snake_locations"][0])
   frontier_direction.append(problem["current_direction"])
```

We declare a while loop with the base condition while the goal is not found using a found goal variable of type Boolean to continuously expand the search tree until the found goal condition is true. Within the loop body generate the children and the children direction of the element at index 0 of the frontier list with the help of the getChildren () function that we have defined previously, at this point the first frontier is considered to be expanded and is therefore moved to the explored array and the explored direction array. Since the frontier has already been expanded and appended to the explored list, we delete that element from index 0 of the frontier list.

```
while  found_goal == False:
    #generating the children for the frontier at index 0

    children, children_direction = self.getChildren(frontier[0][0],frontier[0][1])

    #appending expanded frontier
    explored.append(frontier[0])
    explored_direction.append(frontier_direction[0])


    #deleting expanded frontier from the list
    del frontier[0]
    del frontier_direction[0]
```

Within our conditional while loop we loop through our snake locations and children checking to see if any of the children have the same location as the snake body. If it is true, then that child must be deleted. This is very essential as if this step is not performed the search tree will expand that node and it can cause the snake to bite itself.

```
#checking if snake location is the same as the children locations
for s in enumerate(problem["snake_locations"],start=1):
    for i in range(len(children)-1):
        if s == children[i]:
            del children[i]
            del children_direction[i]
```

The next section of the code within the while loop performs the goal test on each element from the children list at each iteration. Firstly, we also perform a check to see if the child at iteration was previously expanded or generated. If this condition is false, we proceed to check if the child has the same state as the goal node. If that is true, then we set the found goal to true which helps to terminate the while and proceed to traceback the steps required for the

snake to reach from its initial position to the goal node. To do this we first calculate the difference in distance between the x and y coordinates of the goal node and the initial node. If the difference in x is positive, it means that the snake needs to move further east to get closer to the goal. On the other hand, if the difference in x is negative it means that the snake needs to move further west to approach the location to the goal node. Similarly, if the difference in y is positive it means that the snake needs to move further south, and it is negative it needs to move further north to approach the goal node.

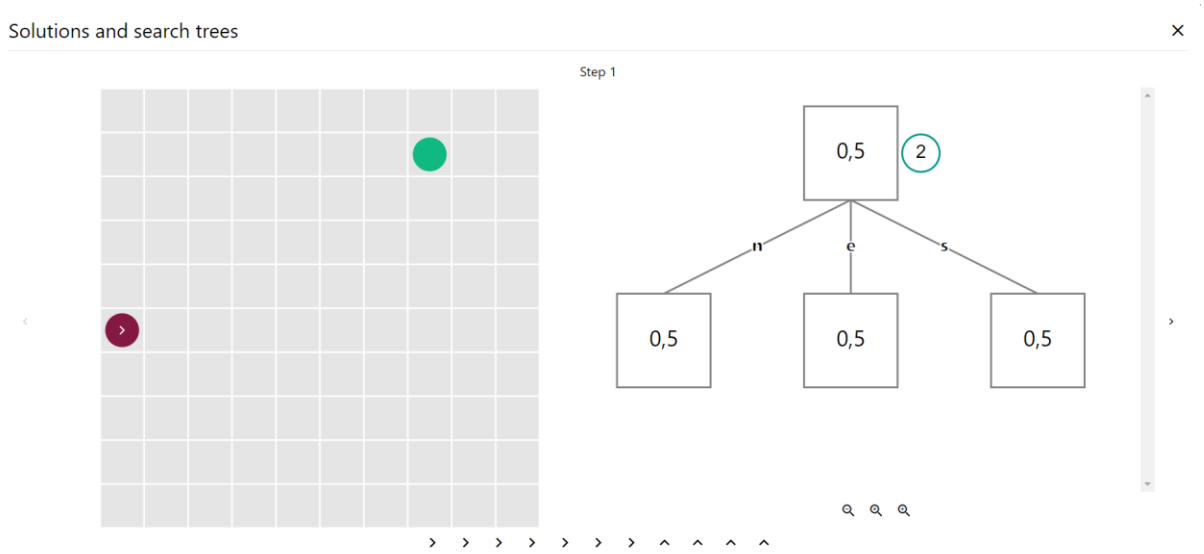
```
#looping through the children list
for child in children:
    # check if a node was expanded or generated previously
    if not (child in explored) and not (child in frontier):

        #check if child is goal
        if child == problem["food_locations"][0]:
            found_goal = True
            #after goal is found, calculate the distance between the snake and the goal
            distance_x = problem["food_locations"][0][0] - problem["snake_locations"][0][0]
            distance_y = problem["food_locations"][0][1] - problem["snake_locations"][0][1]
            sp = problem["snake_locations"][0][0]
            last_element_x = sp + distance_x
```

Based on the condition above we initialize for loops that iterate for the length of distance x and y and repeatedly increment or decrement the x and y in the snake location coordinate. Once incremented we check if the coordinate is still within the bound of the grid if so, we append that coordinate to the solution path.

```
#trackback the sets required to reach the goal
if distance_x < 0:
    for row_distance in range(1,abs(distance_x)+1):
        dis1 = [problem["snake_locations"][0][0] - row_distance, problem["snake_locations"][0][1]]
        if dis1[0] >= 0 and dis1[0] <= 9:
            solution_path.append(dis1)

else:
    for row_distance in range(1,abs(distance_x)+1):
        dis1 = [problem["snake_locations"][0][0] + row_distance, problem["snake_locations"][0][1]]
        if dis1[0] >= 0 and dis1[0] <= 9:
            solution_path.append(dis1)

if distance_y < 0:


    for col_distance in range(1,abs(distance_y)+1):
        dis2 = [last_element_x,problem["snake_locations"][0][1]-col_distance]
        if dis2[1] >= 0 and dis2[1] <= 9:
            solution_path.append(dis2)
else:
    for col_distance in range(1,abs(distance_y)+1):
        dis2 = [last_element_x,problem["snake_locations"][0][1]+col_distance]
        if dis2[1] >= 0 and dis2[1] <= 9:
            solution_path.append(dis2)
```

Now that we have generated the solution path that the snake needs to move through to reach the goal we also need to generate the series of actions or directions required to reach these

states. Firstly, we need to append the snake's current direction that the snake is facing to the solution as this is the first step of the solution that will be followed by the series of actions to reach the goal state. With the help of the getsolutionDirection function defined above we use a for loop that iterates through the solution path array getting the direction required to move between the adjacent states in the solution path until it reaches the goal and appending that direction to the solution array at each iteration. With this we have completed our goal test and generation of our solution path and solution we proceed to print the explored list, frontier list, children list solution path list and their respective directions at each iteration of the loop. Finally, we use the generateSearchTree function previously defined to update the search tree each time the run function is called.



INFORMED: Greedy best-first search:

Greedy search is a variant of best first search where each node in the graph has an evaluation function:

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

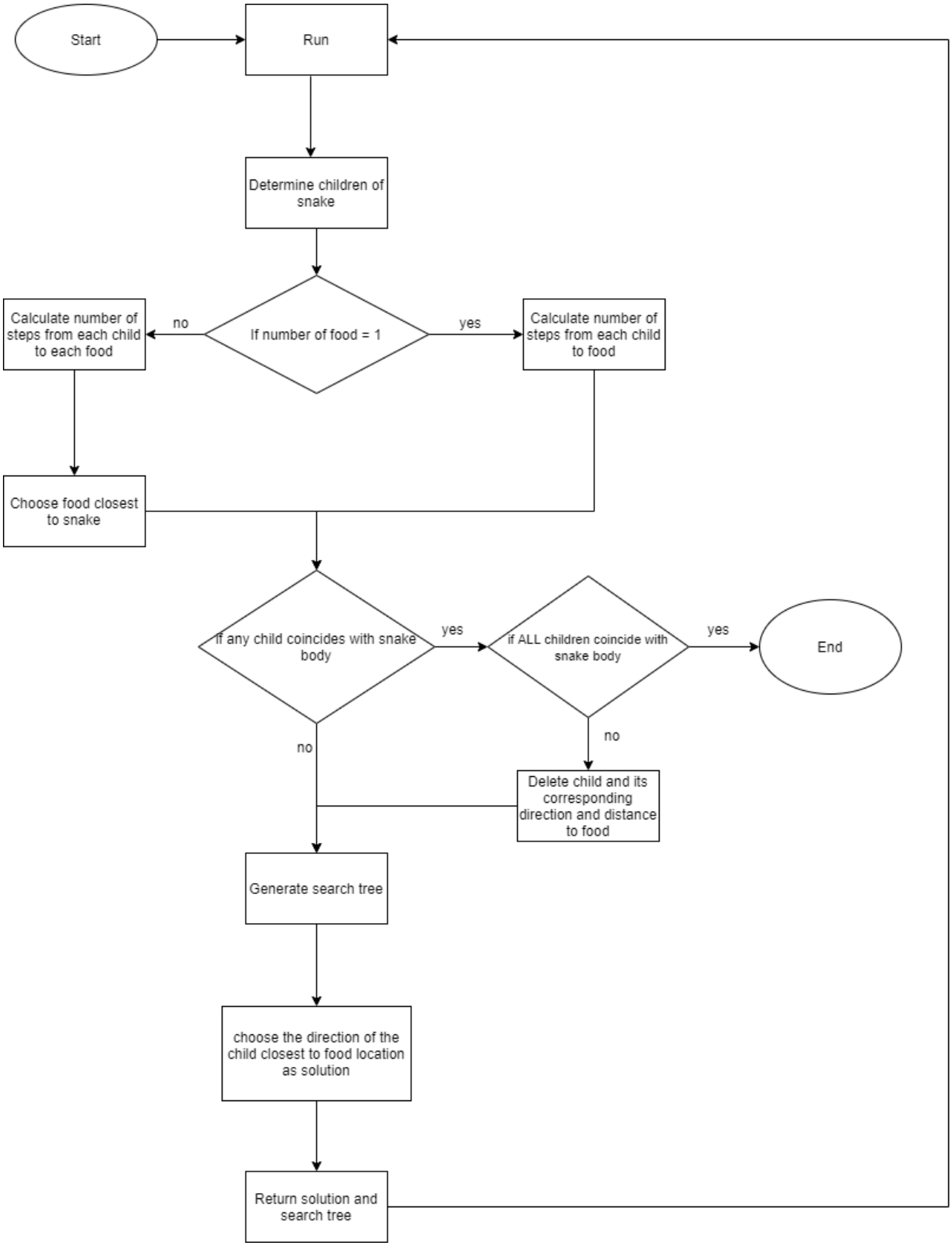
The above function is made up of the heuristics function [h(n)] which provides knowledge about the problem to come up with the best solution. Furthermore, it is used to evaluate the estimated total cost of the nodes and thereby the node with the lowest evaluation will be expanded first. The Greedy best-search algorithm will select the best solution at a particular point in time. Thus, the algorithm will expand the node closest to the goal with the minimum cost.

This variant of snake game uses a game area that is the maze within which the snake can move. Since we are using dynamic mode, the snake size increases each time it finds a food. Per contra, when the snake hits either the border or itself, it dies.

In this situation, the position of the snake head is represented as a node which when expanded have either 4 children, 3 children or 2 children depending on the position of the snake head. Each node consists of information about the location of the snake head and the distance to the goal.

In this case, we are using the heuristic function to calculate the distance between the goal and the node. The code is implemented in such a way that the heuristic function finds the distance with the lowest cost between the head of the snake and the food. Presented below is the flow chart of the algorithm that provides an overview of the logic and approach taken for the algorithm.

INFORMED ALGORITHM FLOWCHART:



Whenever a child is generated, the direction required to move to that child is determined as well. Each child is appended to an array named “children” and its corresponding direction is appended to an array named “children_direction”.

Finding the number of steps from each child to food:

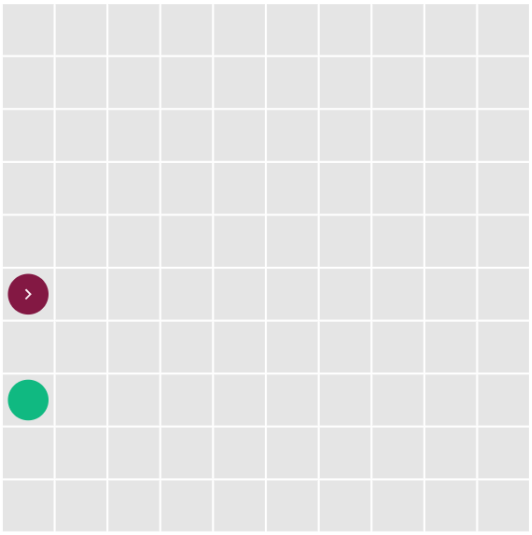
```
solution = []
distance_to_food_list = []

#calculate distance from each child to each food location and choose the closest food
for c in children:
    distance_to_food1 = abs(c[0] - problem["food_locations"][0][0]) + abs(c[1] - problem["food_locations"][0][1])

    if len(problem["food_locations"]) == 2:
        distance_to_food2 = abs(c[0] - problem["food_locations"][1][0]) + abs(c[1] - problem["food_locations"][1][1])
        if distance_to_food1 < distance_to_food2:
            distance_to_food_list.append(distance_to_food1)
        else:
            distance_to_food_list.append(distance_to_food2)
    else:
        distance_to_food_list.append(distance_to_food1)
```

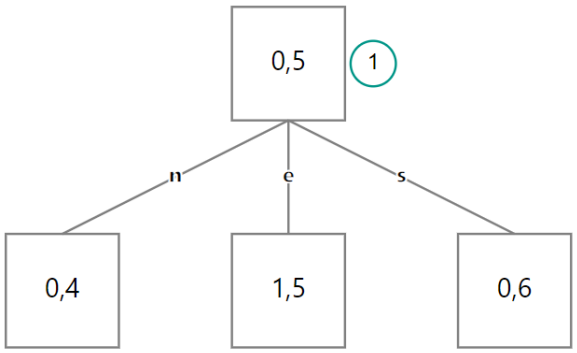
For each child of the snake, the number of steps to the 1st food in food locations is calculated. This is performed by finding the number of steps horizontally and the number of steps vertically and adding them. If there are 2 foods, the algorithm also calculates the number of steps from the child to the 2nd food. The program then compares the two numbers and chooses the food with the least number of steps to the child. For each child, this distance is appended to the array “distance_to_food_list”. Hence, the distance at index i of “distance_to_food_list” will correspond to the child at index i or the array “children”.

Issue in generating the search tree:

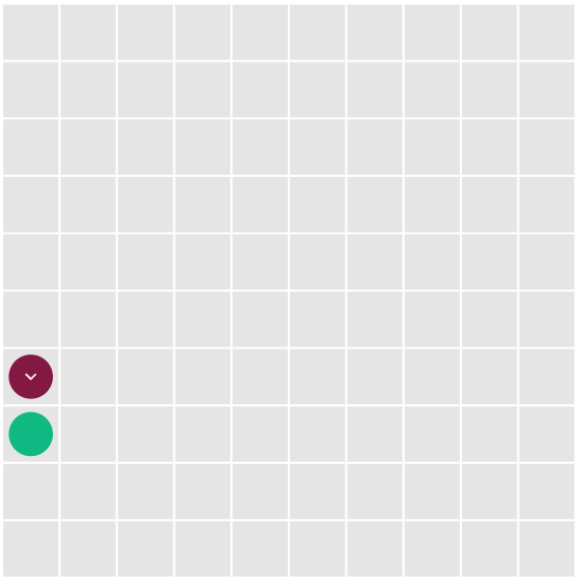
The search tree generated becomes exponentially huge after some iterations of the Run function, thus it massively slows down the frontend of the interface and may even cause it to become unresponsive. Thus, we have ignored the returning of the search tree to the interface. Some example solutions of the search tree are found below.

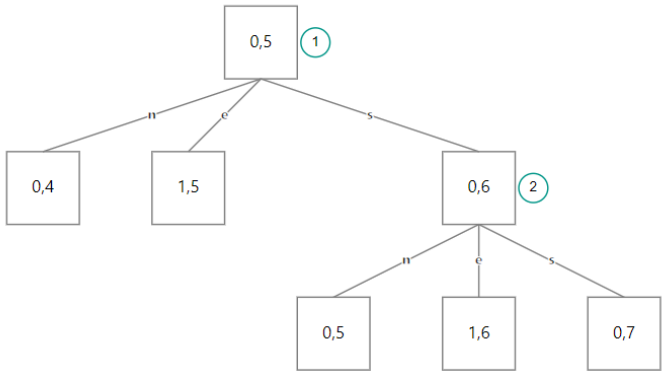
Step 1

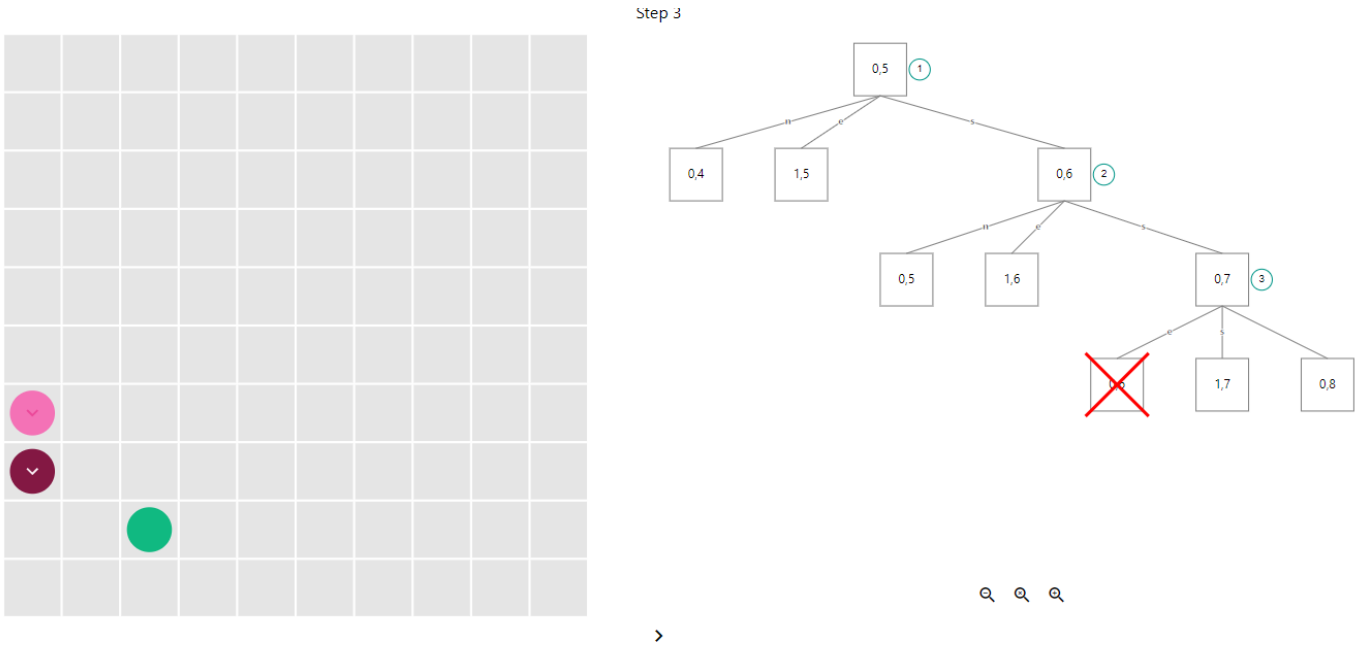




Step 2







Deleting children that will bring snake to eat itself:

```
#delete child if the snake body is in the same location (dynamic length)
 for s in problem["snake_locations"]:
    for i in range(len(children)):
         if s == children[i]:
             for st in Player.search_tree:
                 if st["state"] == (str(children[i][0])+ "," + str(children[i][1])) and st["expansionsequence"] == -1:
                     st["removed"] = True
             del children[i]
             del distance_to_food_list[i]
             del children_direction[i]
             break
```

This block of code checks if any of the children generated coincide with the snake of the body which is found in snake_locations. If a child is found to coincide with the snake body, it is deleted from the children list. Its corresponding value in “children_direction” and “distance_to_food_list” are also deleted.

Choosing the solution:

```
#choose child closest to food location and choose its respective direction in children_direction as solution
 for i in range(len(children)):
     if distance_to_food_list[i] == min(distance_to_food_list):
         solution = [(children_direction[i])]
         break
```

The child with the minimum number of steps to the food is chosen as the solution. Its corresponding direction is thus the solution that is returned.

SECTION III: RESULTS AND DISCUSSIONS

Statistics has been used to support and explain the efficiency of our algorithms.

Below is a Histogram showing the distribution of the outcomes of the informed search algorithm, with a sample of 40 trials.

The 40 iterations were done in four sets of tens on four different computers.

The results collected for both cases are those obtained with a snake having a dynamic snake length, eating food that are generated twice at a time.

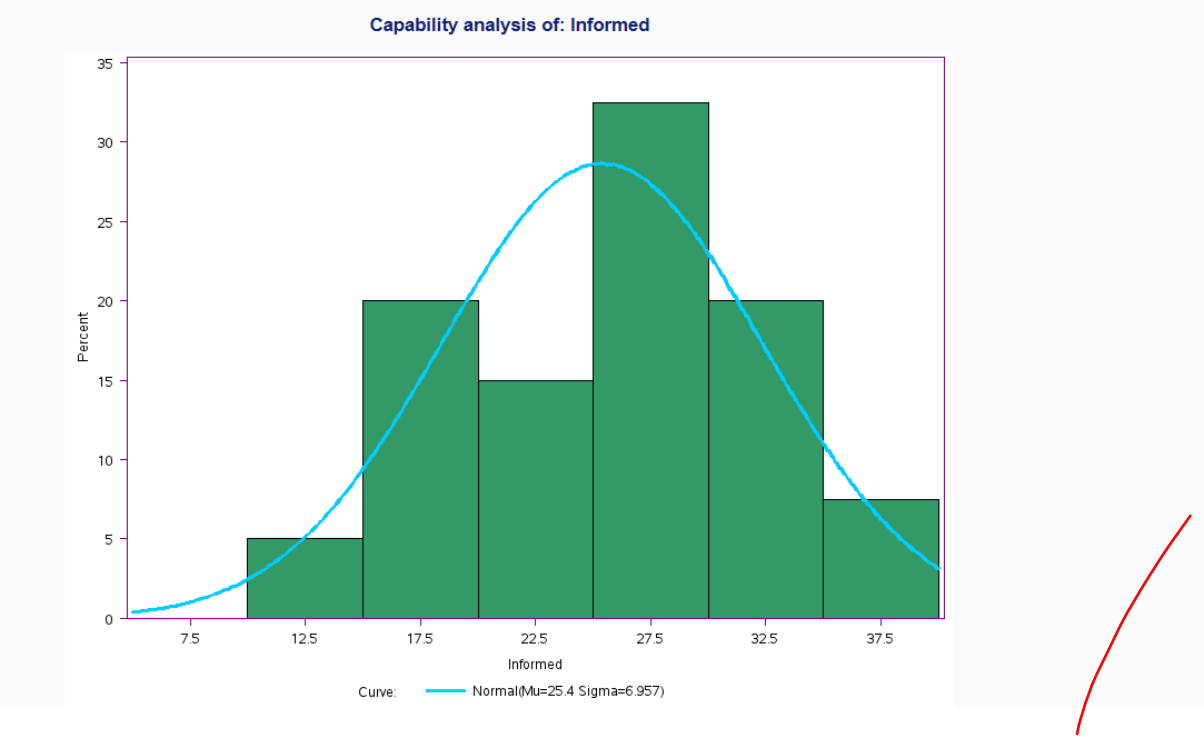


Diagram 1

From the above diagram, the normal curve (blue line) demonstrates that the results are normally distributed. Similarly, to have an overview of the estimate distributed values, the table (Table1.1) below has been generated.

Capability analysis of: Informed

The CAPABILITY Procedure
Variable: Informed

Basic Statistical Measures			
Location		Variability	
Mean	25.40000	Std Deviation	6.95701
Median	26.00000	Variance	48.40000
Mode	29.00000	Range	28.00000
		Interquartile Range	11.00000

Table 1.1

The above table shows the summary statistics of the combined results obtained from the informed search algorithm.

The table comprises central tendencies- mean, mode and median, as well as the measure of dispersion - the quartiles and the standard deviation.

The mean is useful to compare the sets of data and is the only central tendency that uses all data. The mode is only useful to see what datum has been repeated most.

The standard deviation value shows by how much the values in the data set are deviated from the mean.

The value of the mean (25) shows the completion of the challenges b, and c (In instruction manual) which specified a minimum score of 10 points.

Likewise, Diagram 2 shows the histogram of the distribution with the uninformed search algorithm. The conditions for the 40 samples were kept the same as with the informed one. This time as well, the distribution is found to be normal (Illustrated by the blue curve in the diagram).

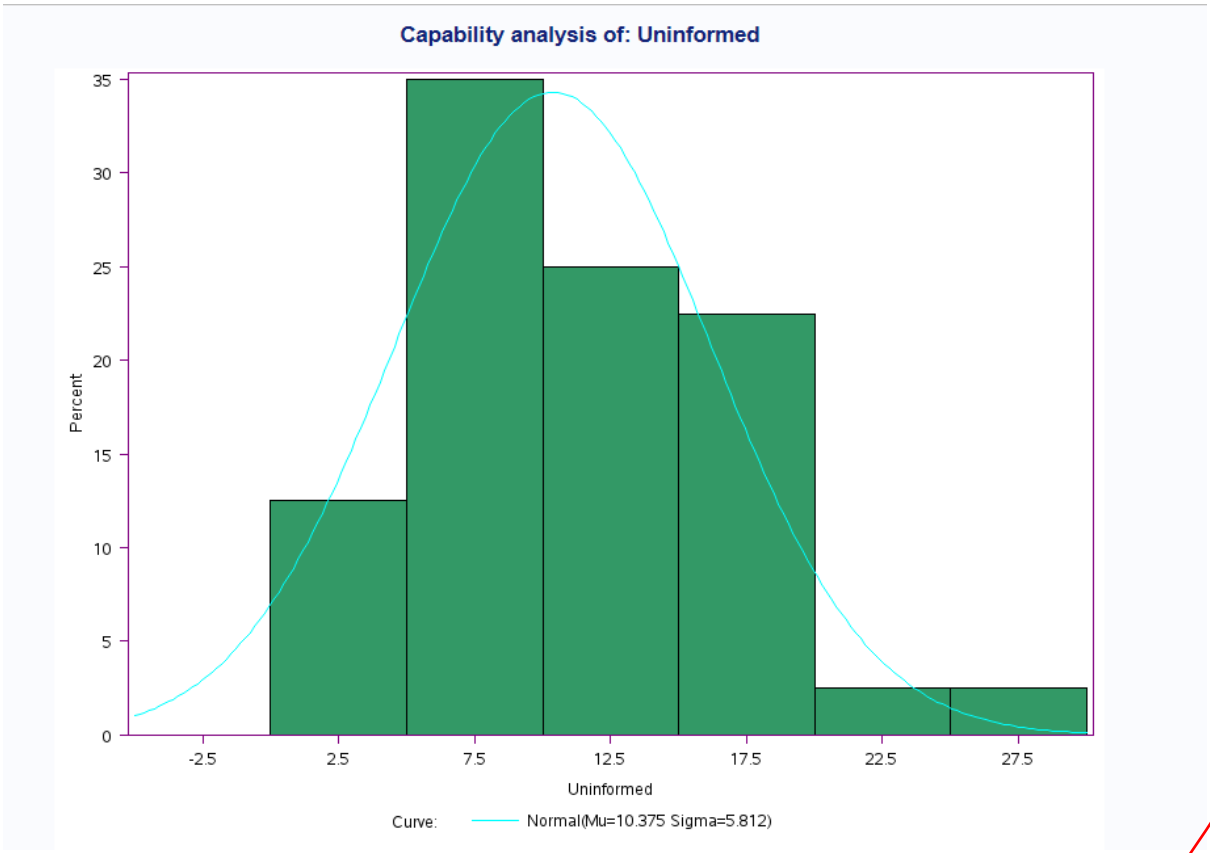


Diagram 2

Capability analysis of: Uninformed

The CAPABILITY Procedure
Variable: Uninformed

Basic Statistical Measures			
Location		Variability	
Mean	10.37500	Std Deviation	5.81196
Median	10.00000	Variance	33.77885
Mode	10.00000	Range	27.00000
		Interquartile Range	9.00000

Note: The mode displayed is the smallest of 2 modes with a count of 5.

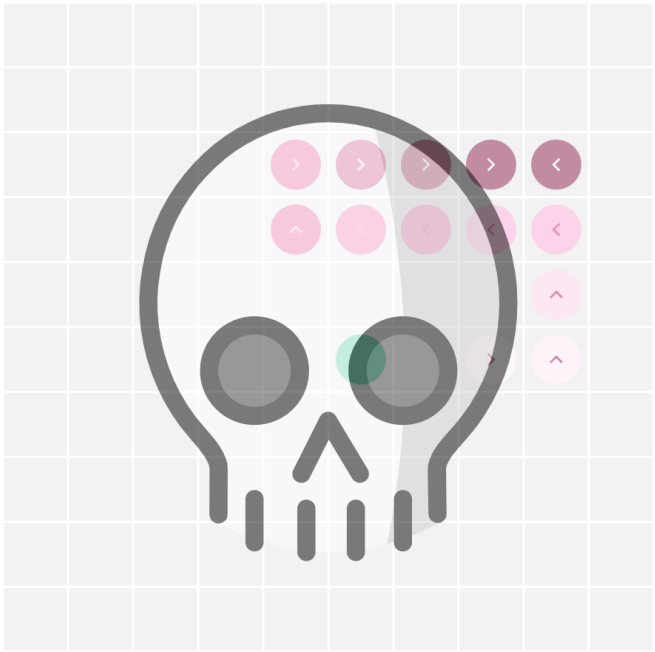
Table 2.1

Table 2.1 shows the summary statistics with the uninformed algorithm. A mean and mode of 10 points shows the correctness of the solution. Additionally, the range of 27 shows the highest score of the approach.

Error patterns and points of improvement:

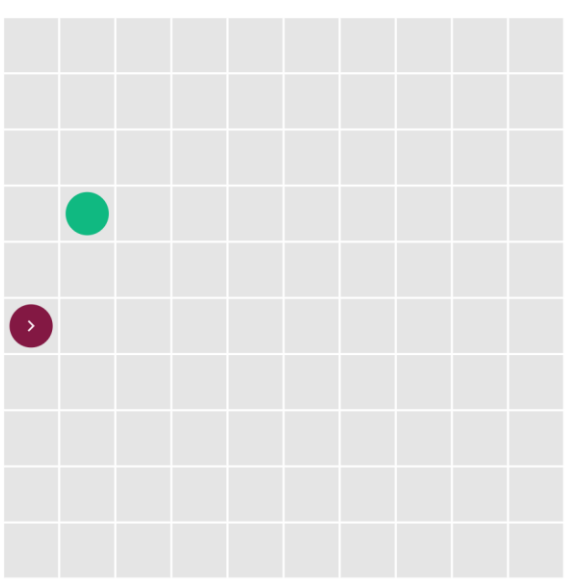
BFS algorithm:

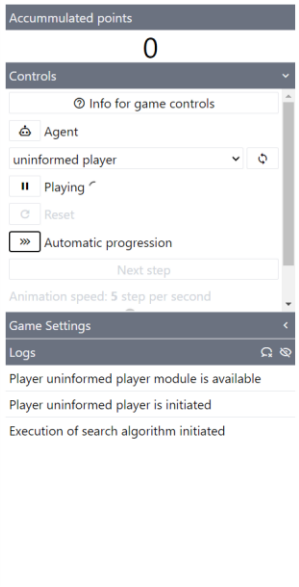
When testing the bfs algorithm there exists certain cases where the snake is unable to reach the food and bites itself when it has a dynamic snake length, during these cases there is a clear pattern that causes this which is when the food location is in the opposite side to where the snake head is facing in this scenario the snake tries to turn back but however it eats itself. This issue occurs due to the incrementing and decrementing of the snake location when generating the solution path as the coordinates are the same as the snake body when the food is in the opposite direction to the snake head. An example of this can be seen in the following image:

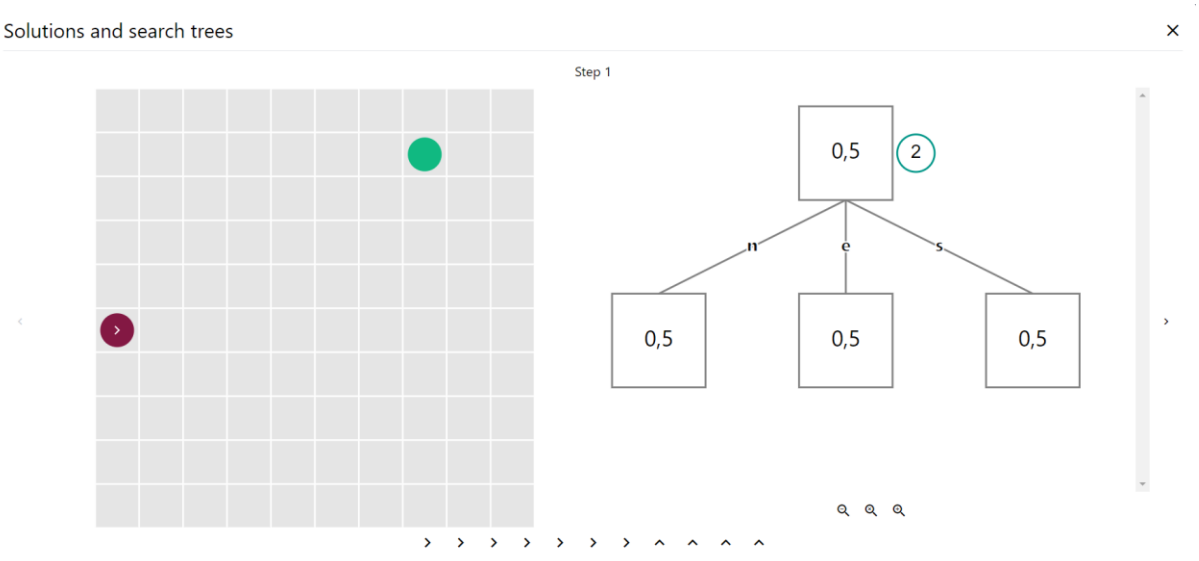


Furthermore, since in bfs all the nodes of the children are expanded until the goal node is generated this means that the search exponentially grows until a goal is found and therefore massively increases the computational cost used when running this algorithm and thus as a

result it impacts the running of the algorithm and the snake does not move. Furthermore, in our current implementation of the code the search true does not accurately place the correct does when updating the search tree, this is because the explored list is exponentially growing and therefore is unable to keep track of the correct element to take as the parent state resulting in an inaccurate search tree.







Thus, in running the algorithm the search tree generation function must be commented in order to run the code efficiently.

In order to improve the algorithm an alternative approach must be taken to implementing the search tree dictionaries such that it only considers the nodes of the solution path rather than all the nodes expanded since a majority of them do not lead to the goal. Furthermore, in order to overcome the error where the snake bites itself a function should be declared to constantly check if the food is directly opposite to the snake head and account for this when incrementing and decrementing the snake's initial location to reach the goal

Greedy best-first search algorithm:

When testing the GBFS algorithm, we can conclude that the snake reaches the goal of all challenges as discussed above. Despite that, when it comes to improving the algorithm so that the snake does not die, here are some key points that we found:

As the snake gets longer, it loops in itself and gets trapped, making it impossible for it to reach the food.

One way to counter this problem would be to generate an algorithm that is able to predict these loops and avoid them. Machine learning could be utilized to achieve this goal.

SECTION IV: CONCLUSION

In conclusion when comparing both the informed greedy best-first search and the uninformed breadth-first search it becomes clear that the greedy best-first search is the better suited algorithm for this task as the breadth-first search has an enormous amount of computational overhead since it expands each node until the goal is found unlike in the informed algorithm where the goal location is known therefore making the greedy best-first search more efficient.