AI Report

Team name:

Ais Teh

Members:

Khor Li Heng, 19014885

Hassan Azwaan, 18086157

Lim Chze Yee, 16034357

a) What is the problem the algorithms face?

The problem the snake faces is locating the food without colliding on to the wall or killing itself too quickly on the way to the food. Both algorithms which are Breadth First Search (uninformed search algorithm) and Greedy Best First Search (informed search algorithm) are tailored to tackle the situation of finding the food.

b) How do the Parameters of the problems fit into the algorithm?

· **Overall:**

Problem[“snake_locations”] : initial_state, Problem[“current_direction”] : initial_direction:

- shows position where the snake is located and its direction
- We use the initial_state to expand the first frontier

· **Breadth First Search:**

Problem[“food_location”]: goal_state:

- To identify the goal

· **Greedy Best First Search:**

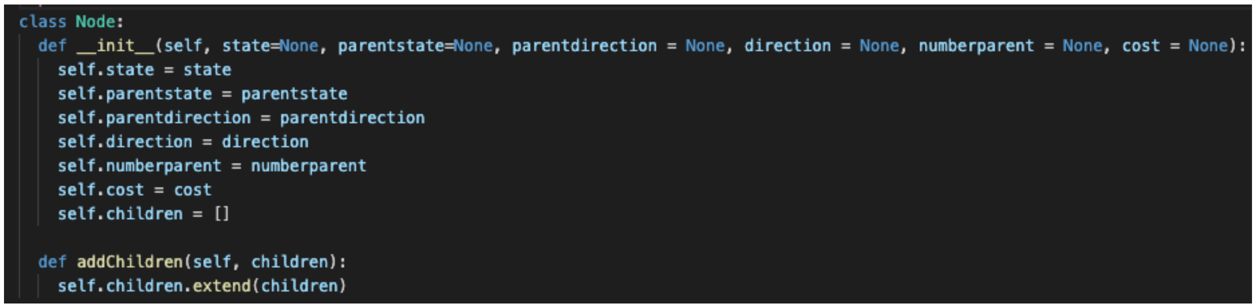
Problem[“food_location”]: goal_state:

- To calculate the cost from nodes to goal and to identify the goal

c) Making the algorithm understand and be able to solve the particular problem

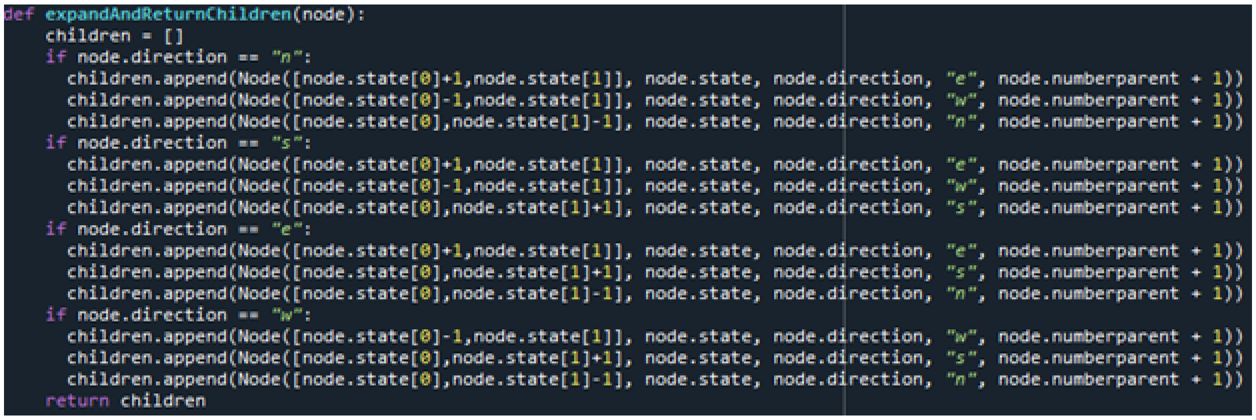
Never just leave the code to explain itself.

- **Overall:**



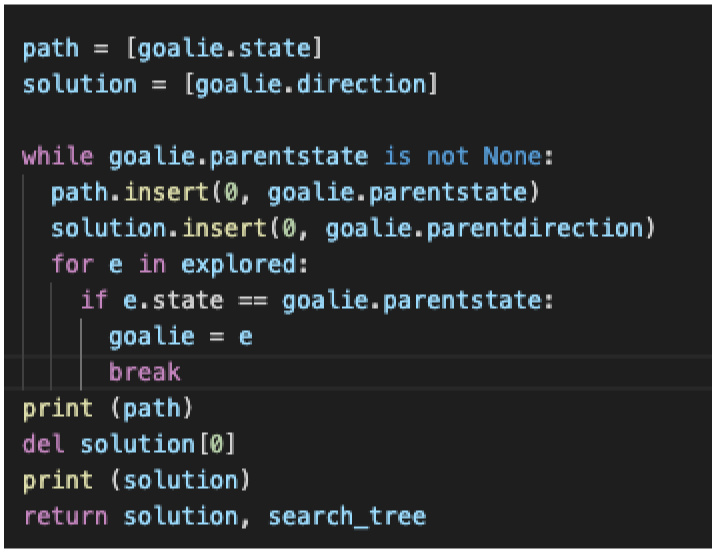
```python
class Node:
  def __init__(self, state=None, parentstate=None, parentdirection = None, direction = None, numberparent = None, cost = None):
    self.state = state
    self.parentstate = parentstate
    self.parentdirection = parentdirection
    self.direction = direction
    self.numberparent = numberparent
    self.cost = cost
    self.children = []

  def addChildren(self, children):
    self.children.extend(children)
```

Figure 1.1 Class Node to store data for each node



```python
def expandAndReturnChildren(node):
    children = []
    if node.direction == "n":
      children.append(Node([node.state[0]+1,node.state[1]], node.state, node.direction, "e", node.numberparent + 1))
      children.append(Node([node.state[0]-1,node.state[1]], node.state, node.direction, "w", node.numberparent + 1))
      children.append(Node([node.state[0],node.state[1]-1], node.state, node.direction, "n", node.numberparent + 1))
    if node.direction == "s":
      children.append(Node([node.state[0]+1,node.state[1]], node.state, node.direction, "e", node.numberparent + 1))
      children.append(Node([node.state[0]-1,node.state[1]], node.state, node.direction, "w", node.numberparent + 1))
      children.append(Node([node.state[0],node.state[1]+1], node.state, node.direction, "s", node.numberparent + 1))
    if node.direction == "e":
      children.append(Node([node.state[0]+1,node.state[1]], node.state, node.direction, "e", node.numberparent + 1))
      children.append(Node([node.state[0],node.state[1]+1], node.state, node.direction, "s", node.numberparent + 1))
      children.append(Node([node.state[0],node.state[1]-1], node.state, node.direction, "n", node.numberparent + 1))
    if node.direction == "w":
      children.append(Node([node.state[0]-1,node.state[1]], node.state, node.direction, "w", node.numberparent + 1))
      children.append(Node([node.state[0],node.state[1]+1], node.state, node.direction, "s", node.numberparent + 1))
      children.append(Node([node.state[0],node.state[1]-1], node.state, node.direction, "n", node.numberparent + 1))
    return children
```

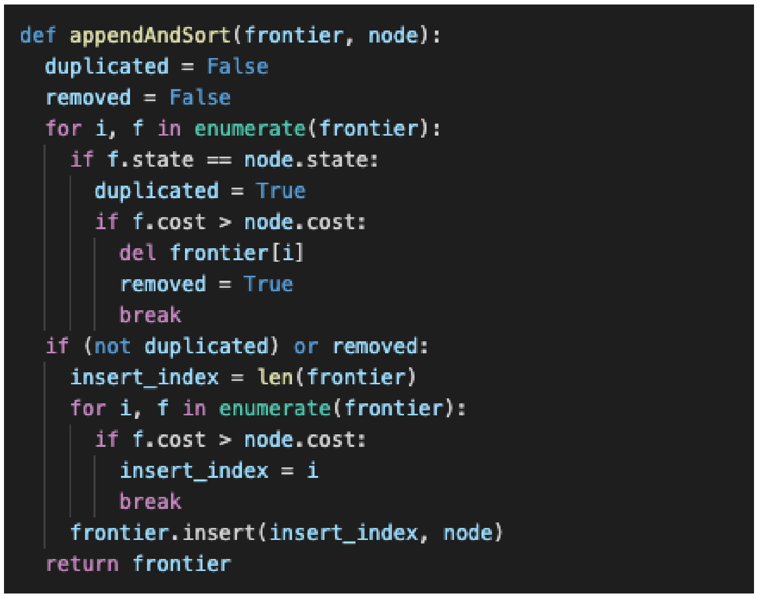
Figure 1.2 The "expandAndReturnChildren" function is to explore, expand and create child node



```
path = [goalie.state]
solution = [goalie.direction]

while goalie.parentstate is not None:
    path.insert(0, goalie.parentstate)
    solution.insert(0, goalie.parentdirection)
    for e in explored:
        if e.state == goalie.parentstate:
            goalie = e
            break
print (path)
del solution[0]
print (solution)
return solution, search_tree
```

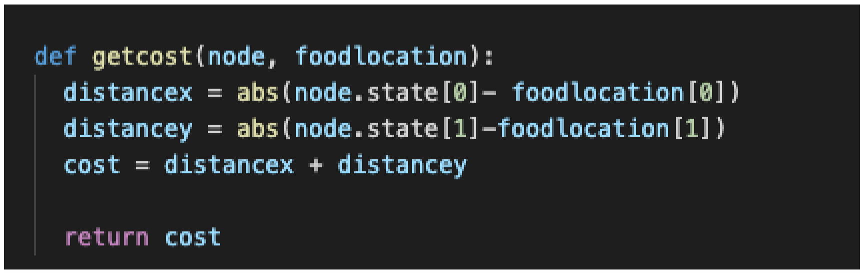
Figure 1.3 collect the path state in path and the movement in solution then return solution

Greedy Best First Search:



```
def appendAndSort(frontier, node):
    duplicated = False
    removed = False
    for i, f in enumerate(frontier):
        if f.state == node.state:
            duplicated = True
            if f.cost > node.cost:
                del frontier[i]
                removed = True
                break
    if (not duplicated) or removed:
        insert_index = len(frontier)
        for i, f in enumerate(frontier):
            if f.cost > node.cost:
                insert_index = i
                break
        frontier.insert(insert_index, node)
    return frontier
```

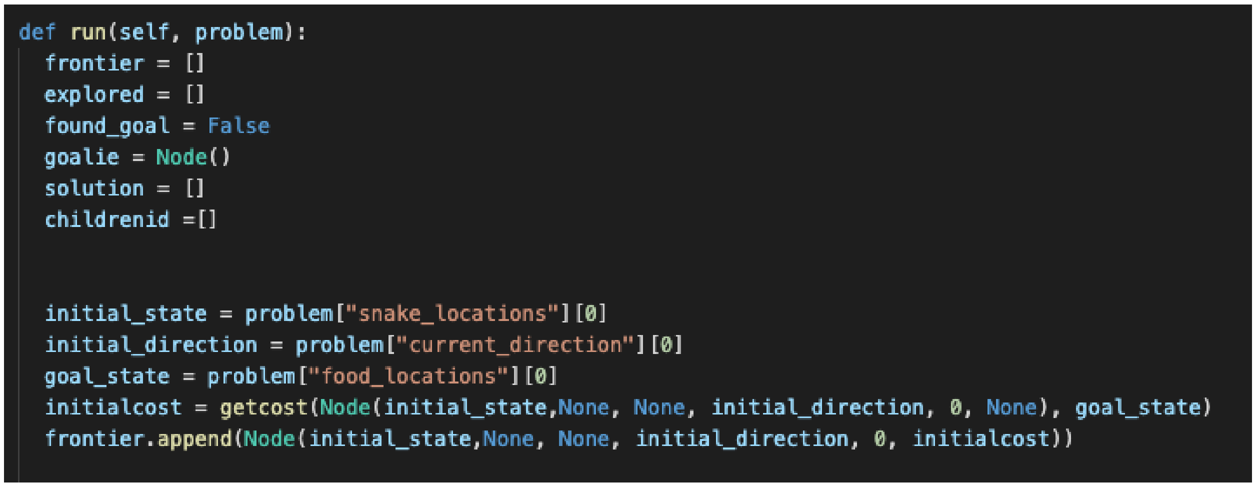
Figure 2.1 Compare and sort the child and frontier if the node has the lowest cost will be in frontier[0]



```
def getcost(node, foodlocation):
    distancex = abs(node.state[0]- foodlocation[0])
    distancey = abs(node.state[1]-foodlocation[1])
    cost = distancex + distancey

    return cost
```

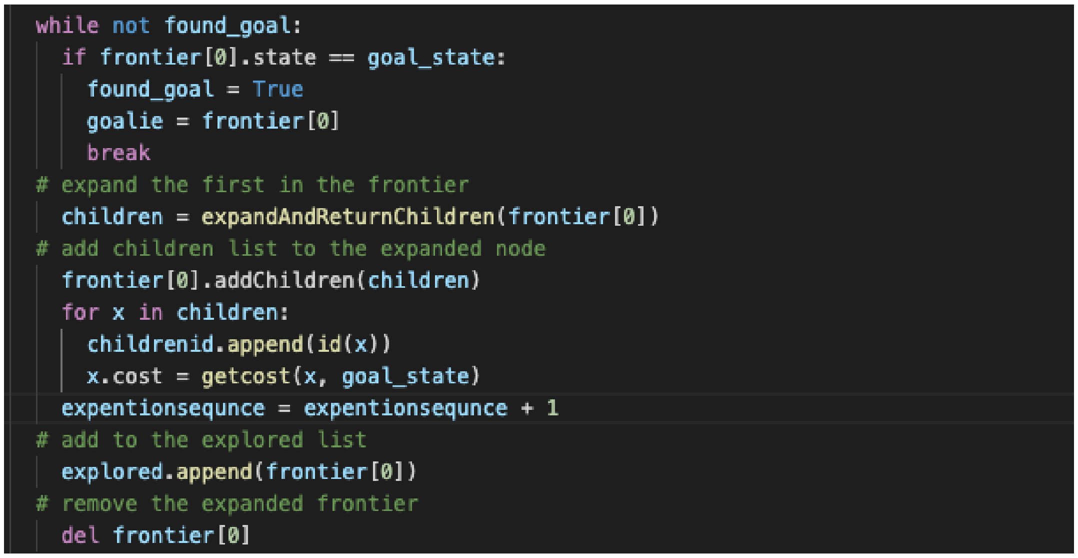
Figure 2.2 To generate cost abs(food location - current location)

$COST = abs(Xf - Xs) + abs(Yf - Ys)$



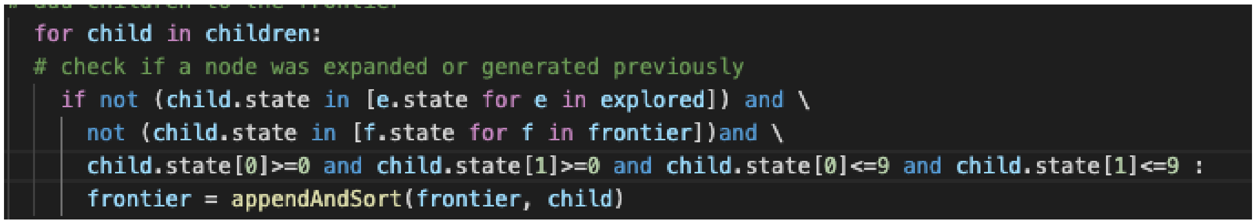
```python
def run(self, problem):
    frontier = []
    explored = []
    found_goal = False
    goalie = Node()
    solution = []
    childrenid =[]


    initial_state = problem["snake_locations"][0]
    initial_direction = problem["current_direction"][0]
    goal_state = problem["food_locations"][0]
    initialcost = getcost(Node(initial_state,None, None, initial_direction, 0, None), goal_state)
    frontier.append(Node(initial_state,None, None, initial_direction, 0, initialcost))
```

Figure 2.3 Gather the information like cost, snake locationand snake direction create the first Node and append into frontier[0]



```python
    while not found_goal:
        if frontier[0].state == goal_state:
            found_goal = True
            goalie = frontier[0]
            break
    # expand the first in the frontier
        children = expandAndReturnChildren(frontier[0])
    # add children list to the expanded node
        frontier[0].addChildren(children)
        for x in children:
            childrenid.append(id(x))
            x.cost = getcost(x, goal_state)
        expentionsequnce = expentionsequnce + 1
    # add to the explored list
        explored.append(frontier[0])
    # remove the expanded frontier
        del frontier[0]
```

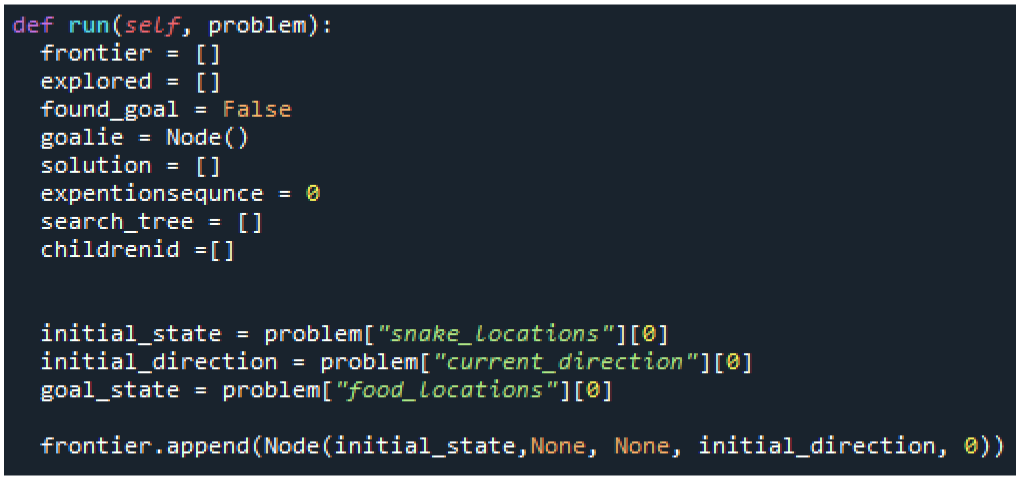
Figure 2.4 Goal test before expanding the frontier and in the for loop get cost for each child. Add the expanded frontier[0] into explored list and delete it from frontier list



```
for child in children:
# 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])and \
    child.state[0]>=0 and child.state[1]>=0 and child.state[0]<=9 and child.state[1]<=9 :
    frontier = appendAndSort(frontier, child)
```

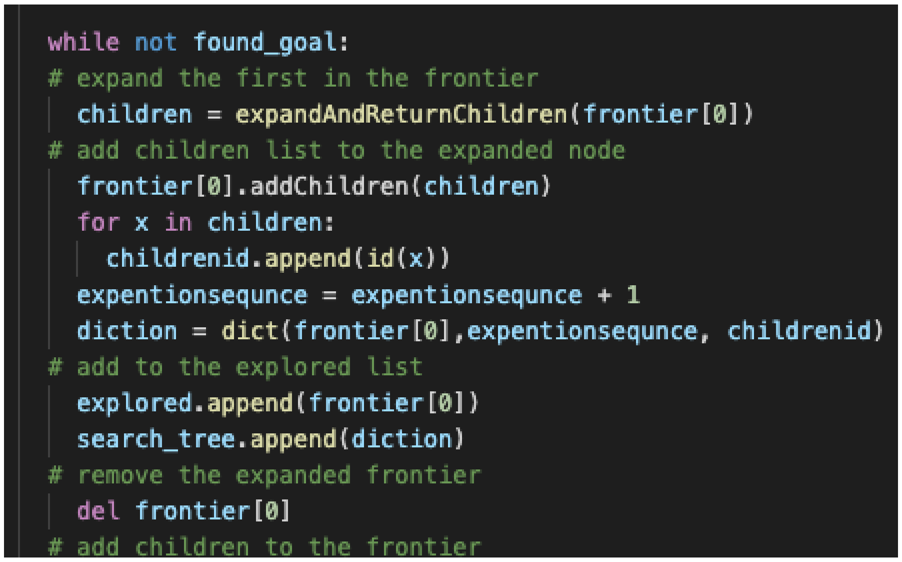
Figure 2.5 In the while loop nest a for loop to loop the children to ensure the repeated child and child which is not in maze 10*10 will not be append in frontier

Breadth First Search



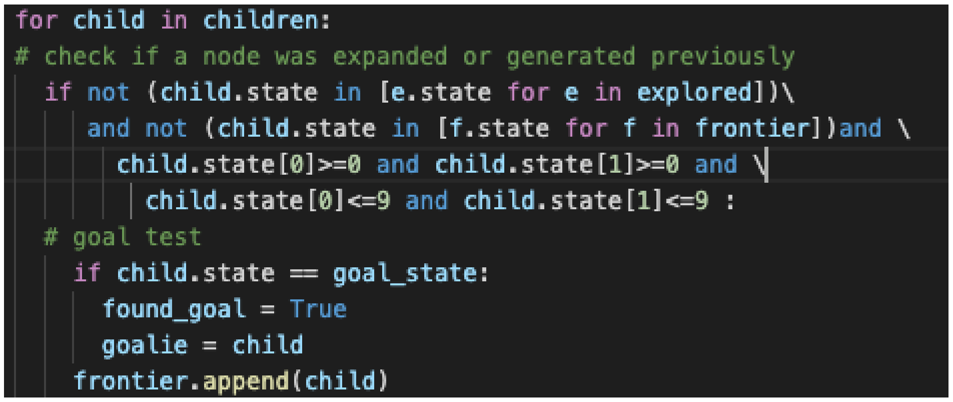
```
def run(self, problem):
  frontier = []
  explored = []
  found_goal = False
  goalie = Node()
  solution = []
  expentionsequnce = 0
  search_tree = []
  childrenid =[]


  initial_state = problem["snake_locations"][0]
  initial_direction = problem["current_direction"][0]
  goal_state = problem["food_locations"][0]

  frontier.append(Node(initial_state,None, None, initial_direction, 0))
```

Figure 3.1 Gather the information like snake location and snake direction create the first Node and append into frontier[0]



```
while not found_goal:
# expand the first in the frontier
  children = expandAndReturnChildren(frontier[0])
# add children list to the expanded node
  frontier[0].addChildren(children)
  for x in children:
    childrenid.append(id(x))
  expentionsequnce = expentionsequnce + 1
  diction = dict(frontier[0],expentionsequnce, childrenid)
# add to the explored list
  explored.append(frontier[0])
  search_tree.append(diction)
# remove the expanded frontier
  del frontier[0]
# add children to the frontier
```

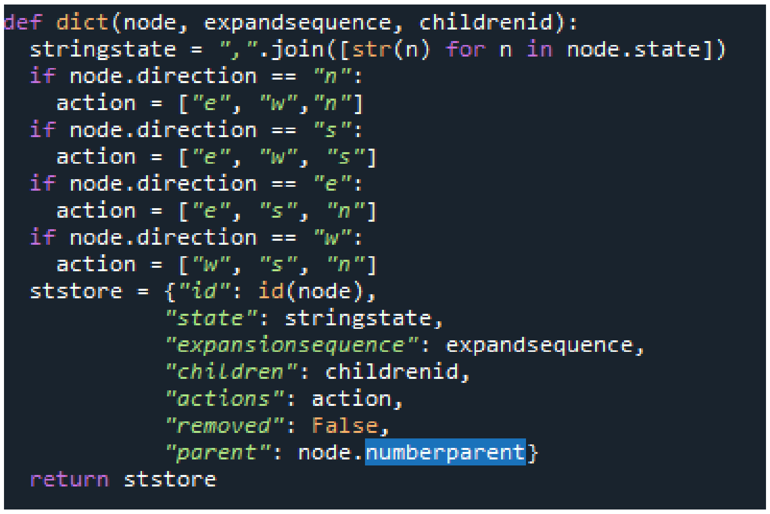
Figure 3.2 Expands the first frontier and saves the data in the Node class. After expanding add the Node into the expanded list and delete the Node in frontier list.



```
for child in children:
# 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])and \
       child.state[0]>=0 and child.state[1]>=0 and \
         child.state[0]<=9 and child.state[1]<=9 :
  # goal test
    if child.state == goal_state:
      found_goal = True
      goalie = child
    frontier.append(child)
```

Figure 3.3 Loop the children list and prevent it from entering frontier list if is explored, repeated and not exceed the 10*10 maze. Finally check the child if it is the goal node.

Breadth First Search Search Tree



```
def dict(node, expandsequence, childrenid):
  stringstate = ",".join([str(n) for n in node.state])
  if node.direction == "n":
    action = ["e", "w","n"]
  if node.direction == "s":
    action = ["e", "w", "s"]
  if node.direction == "e":
    action = ["e", "s", "n"]
  if node.direction == "w":
    action = ["w", "s", "n"]
  ststore = {"id": id(node),
             "state": stringstate,
             "expansionsequence": expandsequence,
             "children": childrenid,
             "actions": action,
             "removed": False,
             "parent": node.numberparent}
  return ststore
```

Figure 4.1 create search tree dictionary and store in ststore

d) Did we achieve the aim of solving the problem?

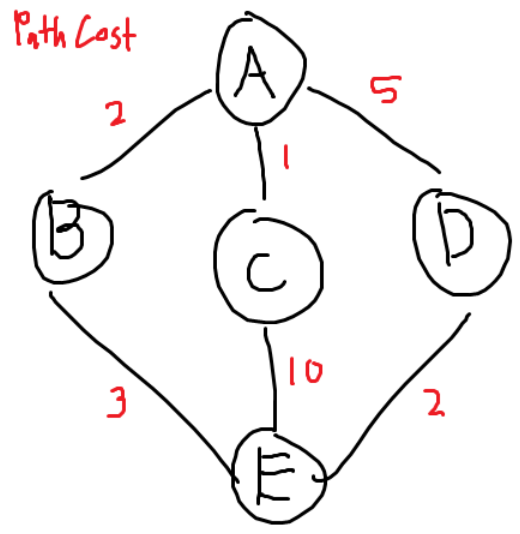
Yes, both uninformed and informed search can indefinitely reach 15 points and above. However, increasing the snake length with food would cause it to fail sometimes. This is because there was no algorithm to avoid collision with itself.

e) How is the Performance?

With the snake length not increasing, the performance for both of the uninformed and informed algorithms are great. If the snake were to increase in length after eating, the performance is heavily dependent on where the food is located. If it is in a non-favourable situation such as the food being located at where the path leads the snake to collide with its body, it will collide. So, it is satisfactory.

- **BFS and GBFS comparisons:**

In theory, BFS should have a slower execution and food search compared to GBFS. Even though it is only concerned about getting to the target without any constraints, it has to search each node until it finds one that contains the food. GBFS is supposedly more cost efficient compared to BFS, because the algorithm itself calculates the total distance between an area to another. However cases could lead it to longer paths that may potentially also cause it to loop as well. Let’s take the figure below as an example.

 Path Cost A 2 5 1 B C D 10 3 2 E Figure 4.1	The goal is to reach E with the least amount of cost needed to move. GBFS will instantly choose C due to its 1 cost, unknowingly taking the longest route to E. While A to B to E is the actual shortest route. While for BFS it will choose the first route. Which, coincidentally, is the shortest path.

However, this will not apply to Snake because each food is spawned randomly after one is eaten.

- **Search Tree comparisons:**

Search Tree for GBFS

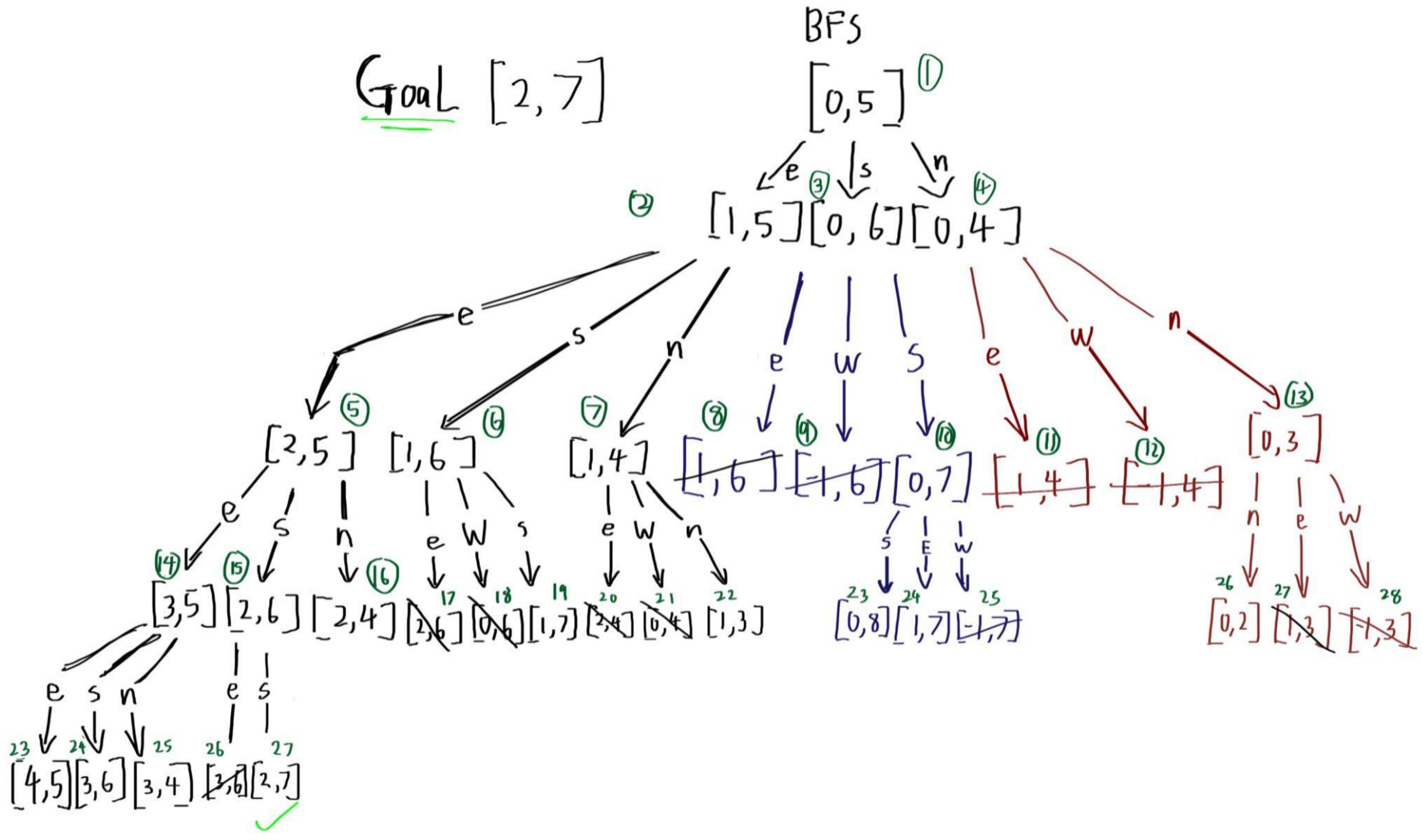
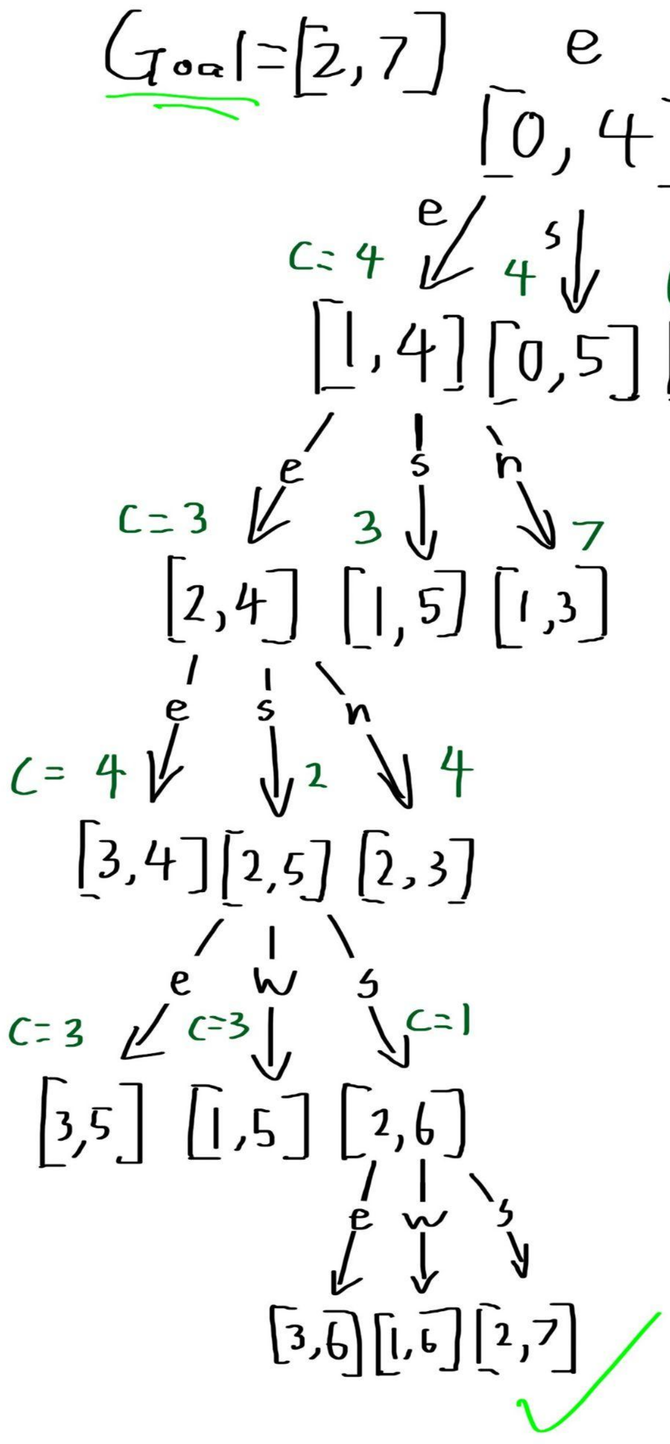


Figure 4.2 GBFS Search Tree

Search Tree of BFS



this is not the search tree for a breadth first search

Figure 4.3 BFS Search Tree

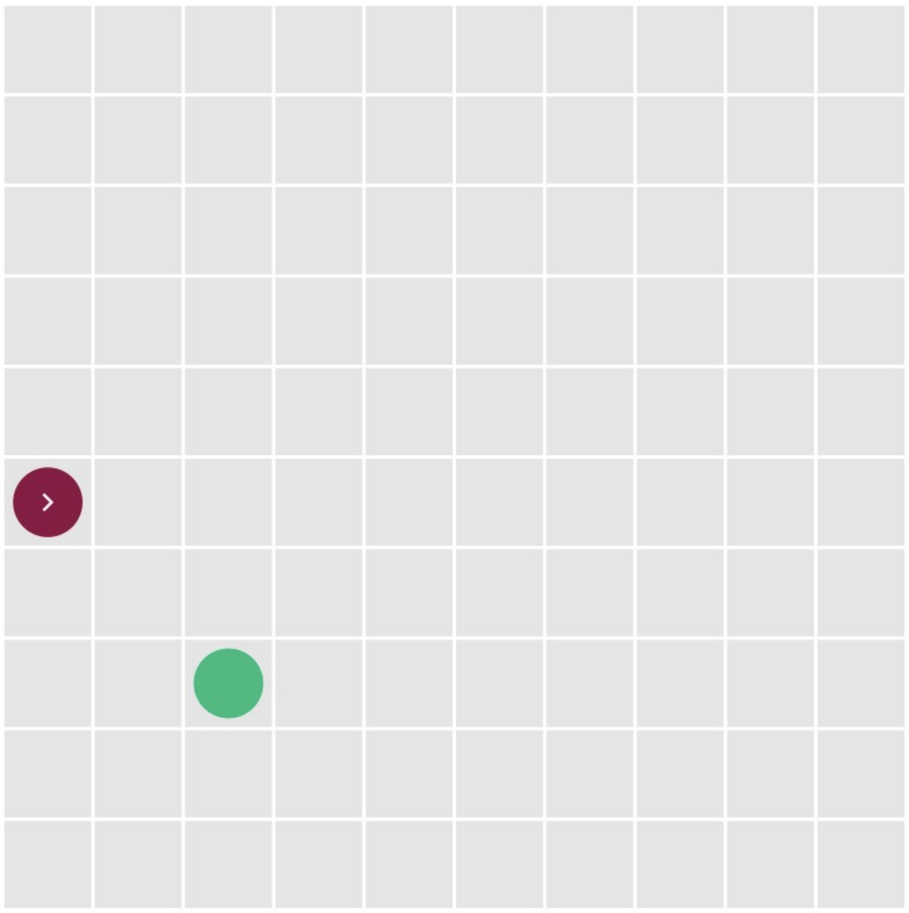


Figure 4.4

The figure above shows the location of the snake [0,5] and food [2,7]. With the Search Trees listed out we can see that in this specific situation, GBFS is superior to BFS due to it being more efficient in finding the food’s location. BFS has to scan a large portion of the map before finding a suitable route.