SCHOOL OF SCIENCE AND TECHNOLOGY

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

ACADEMIC SESSION APRIL 2021;

CSC3206: ARTIFICIAL INTELLIGENCE

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

NO.	STUDENT ID	STUDENT NAME
1	17076407	ZAHAEN HILMIE BIN ZAHARUDIN
2	17069469	KOK WUI LEE
3	17068826	GOH WAI HOONG
4	17074683	LEE JYH YONG

INSTRUCTIONS TO CANDIDATES

- This assignment will contribute 15% to your final grade.
- This is a group (maximum 4 person) assignment.

IMPORTANT

The University requires students to adhere to submission deadlines for any form of assessment. Penalties are applied in relation to unauthorized late submission of work. Coursework submitted after the deadline will be subjected to the prevailing academic regulations. Please check your respective programme handbook.

Academic Honesty Acknowledgement

"We .....(Names) verify that this paper contains entirely our own work. We have not consulted with any outside person or materials other than what was specified (an interviewee, for example) in the assignment or the syllabus requirements. Further, we have not copied or inadvertently copied ideas, sentences, or paragraphs from another student. We realize the penalties *(refer to the student handbook and undergraduate programme handbook)* for any kind of copying or collaboration on any assignment."

……..... (Students' Signature / Date)

Introduction

The Snake Game Agent problem is that the Snake and the food is displayed without executing any algorithm for the Snake to move and consume the food present in the maze as shown in Figure 1. In this assignment, we are required to create an algorithm for the Snake to solve this issue using Uninformed Search and Informed Search. Next, the algorithms we will be creating for the Snake is Greedy Best First Search (Informed) and Breadth First Search (Uninformed), these algorithms were chosen by us as the Greedy Best first search was the suitable choice of algorithm as it allows us to make the Snake consume the food closes to it first rather than consuming the food further from it using a heuristic function. Other than that, the Breadth First Search was also chosen due to its simplicity of understanding how the algorithm will be executed by the Snake, this will allow us to help the Snake look for the food in the maze without knowing the food present location in the 10x10 maze.

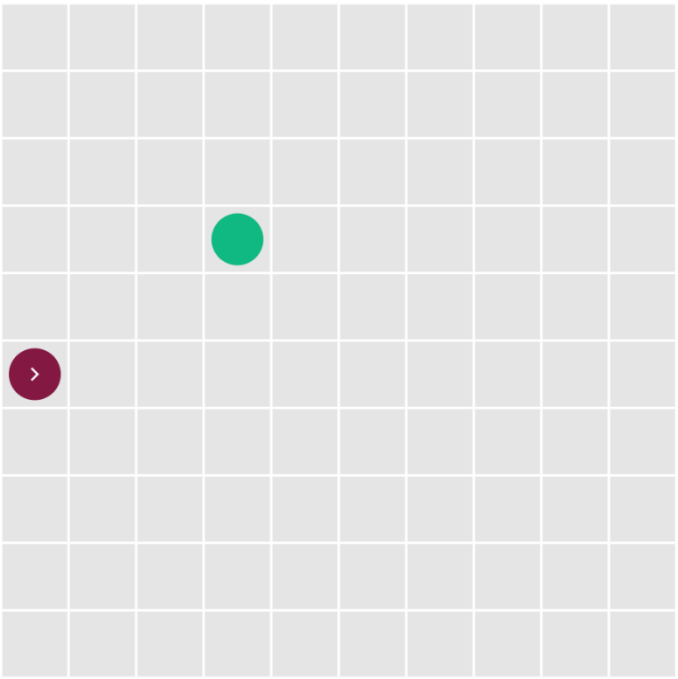

Figure 1

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

Problems Parameters	Breath First Search	Greedy Best First Search
Food location	Used to define the goal state	Used to define the goal state and calculate tile distance of snake with food
Snake location	Used to define the initial state and prevent any collisions that occur between the snake and itself	Used to define the initial state, prevent any collisions that occur between the snake and itself, and calculate tile distance of snake with food
Current Direction	Crucial in generating to generate the final solution and to define a current node's children	Crucial in generating to generate the final solution and to define a current node's children
Max Bounds	Avoid out of bounds	Avoid out of bounds

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

Search algorithms is used in Agent mode of the Snake game to solve the problems which have been stated above by providing the best solution. Search algorithms can help to find the solution more efficiently and quickly compared to human mode. Moreover, with algorithms, the accuracy is higher when compared to human mode as it can provide the best solution (shortest / lowest cost path) in a shorter time. Search algorithms can be classified into uninformed search and informed search algorithms. In our assignment, Breadth First Search (Uninformed Search) and Greedy Best First Search algorithms (Informed Search) are chosen.

Ultimately, the main goal is for the snake to find the food on its own through the search algorithm, but in the program that we created, both search algorithms has managed to meet additional requirements which are:

i) Body of the snake expands each time it eats a food
ii) Snake can achieve a score higher than 15
iii) Snake can operate in an environment with two or more food / goals
iv) Snake length starts at 1

<u>Solution 1 (Breadth First Search Algorithm)</u>

To give some context, Breadth First Search (BFS) is a search algorithm that in a tree environment, expands breath-first meaning that every expandable leaf node in each level will need to be expanded first before we go down the tree to expand its children regardless of cost.

Translating that into code, few components will need to be defined: -

1) Frontier list
   This will be a list of leaf nodes that are to be expanded. We will need this to know which nodes prioritize expansion.

2) Explored list
   List of explored nodes with their info and state being kept. We will need this to avoid any loopy path it might come by meaning it will not go back up the tree.

3) Search tree
   A list of every node that is generated including frontiers, explored, and even rejected nodes. Search tree will be needed by the program to generate the search tree in the game menu. Additionally, the search script will utilize this to generate a proper path to the solution

4) List of directions that is passed to the snake for it to follow. E.g. [“n”,”n”,”s”,”w”]. This will be generated by the end of the search script.

The snake has given us the initial position of the snake and the position of where the food is currently. Both will be used as the initial state and goal state respectively.

Step 1: Defining initial node and initializing frontier list

```
# Initialize the initial node
initialState = problem['snake_locations'][0]
foodLoc = problem["food_locations"][0]
initialNode = {
    "id": 1,
    "state": "{},{}".format(initialState[0], initialState[1]),
    "expansionsequence": -1,
    "children": [],
    "actions": [],
    "removed": False,
    "parent": None
}
search_tree.append(initialNode)

## Frontier consists of [{Node information}, Distance of food from Node, The snake's current body position]
frontier.insert(0, [initialNode, problem['current_direction'], problem['snake_locations']])
currentNodeID = 1
nodeExpanded = 1
```

Figure 2: Code implementation of step 1

Our first step is to define the starting point and our end point of our search algorithm. From the above image, initialState is defined through the current location of the head of the snake and the goal state or foodLoc in this case takes the first food location that it is given. This enables the search algorithm to function if multiple foods/goals exist.

Using initialState, we can create initialNode that is a dictionary of information regarding the Node following the program's format. It should be noted that all nodes will follow the same format.

The Initial Node is manually given id 1 since it's the root node and inserted into the frontier list for processing. What we insert in the frontier is a list of the node information, the node's current direction, and the position of every snake parts including the head. We will assign id's and expansion sequence through the use of currentNodeID and nodeExpanded.

Step 2: Node expansion, find and return children

```
# Keeps looping until goal is found
while not game_ended:

    snakeLoc = frontier[0][2]
    currentDirection = frontier[0][1]

    # Expand from current position
    children, updatedParentNode, currentNodeID = expandNode(self.maxBound, frontier[0][0], currentNodeID,
                                                            nodeExpanded, currentDirection, snakeLoc)
```

Figure 3: Calling node expansion function

We past six parameters to function expandNode() which are the max boundary of the game area, node information that includes its state, counter for id assignment, expansion counter for

expansion sequence assignment, its current direction, and the location of the current snake parts that are in play all listed respectively with the image above.

The function will begin finding and expand its children. The current node state for expansion is also the location of the snake's head. Finding the children will add 1 or subtract 1 from the node state array as it can only move one tile at a time depending on the direction. For our program all but the opposite direction will be considered as children as move backwards would guarantee collision unless the snake has a length of 1. Therefore, each node will expand 3 children one representing a move of a direction that is not the opposite of current direction. For example, current direction is south, the 3 children would be nodes moving east, south, and west.

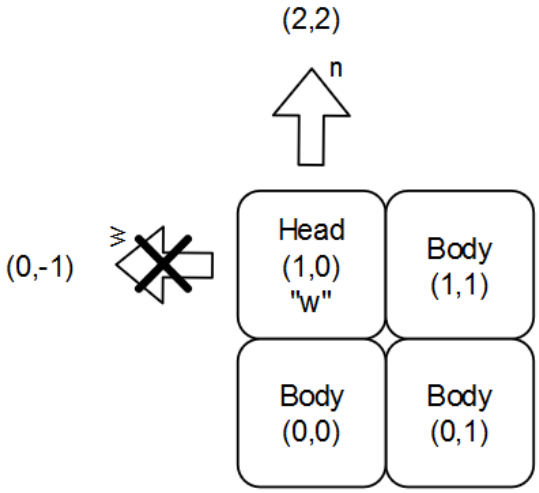

Figure 3.1: Snake example

To meet the additional requirements, some checks would need to be placed to ensure that the children would not lead to a collision. Consider the above image an example of a snake trying to make its next move. In this scenario, the snake that is facing west ("w") can go all possible directions except for east as explained previously since it would go backwards however realistically it can only go on without any issue of collision or out of bounds. The expandNode() function has its own checks ensures that the children would meet these requirements:-

1) New state will not go out of bounds
2) New state does not collide with existing body locations

Failure to pass any of these checks will result in them being removed early on as a candidate to be a frontier node. We will not delete rejected nodes however since we need it for the search tree, so we just define {rejected = true} in the node information to omit it later on.

After checks has been done, we need to update the snake's positions to simulate itself moving then store it in the children so that it can be used to check for body collisions if the child were

to be expanded next. The way we update the body is simply insert new state as the head and cut off the last position which is its tail.

```
# Update snake Loc
currSnakePos = list(snakePos)
currSnakePos.pop()
currSnakePos.insert(0, state)
```

Figure 3.2: Updating snake position list

Function then returns set of children nodes, updated information for the expanded node that includes id of children at its available moves, and lastly an updated counter for the id.

Step 3: Update frontier and explored, goal test.

```
explored.append(frontier[0])

# Update parent in search tree
for x in search_tree:
    if x['id'] == updatedParentNode['id']:
        x['expansionsequence'] = updatedParentNode['expansionsequence']
        x['actions'] = updatedParentNode['actions']
        x['children'] = updatedParentNode['children']
        break

# Confirmed that we expanded a node
nodeExpanded += 1

del frontier[0]


for child in children:

    # No loopy
    if child[0]["state"] in [e[0]["state"] for e in explored]:
        child[0]["removed"] = True

    # No frontier repeats
    elif child[0]["state"] in [f[0]["state"] for f in frontier]:
        child[0]["removed"] = True

    # Add children regardless to search tree for documentation
    search_tree.append(child[0])

    if child[0]["state"] == "{},{}".format(foodLoc[0],foodLoc[1]):
        game_ended = True
        solution = compileSolution(updatedParentNode, search_tree, child[1])
```

Figure 4: Code implementation of step 3

The code snippet above is taken right after function expandNode() has been called. At this point we have the children from the expansion. The most top frontier (frontier[0]) will first be added to the explored list since it has just been "expanded" through the function just now. The for loop after is just to update the expanded node's information in the search tree after its expansion by adding its expansion sequence, children id's and move details. Expansion sequence counter is incremented by one for use of future expansions.

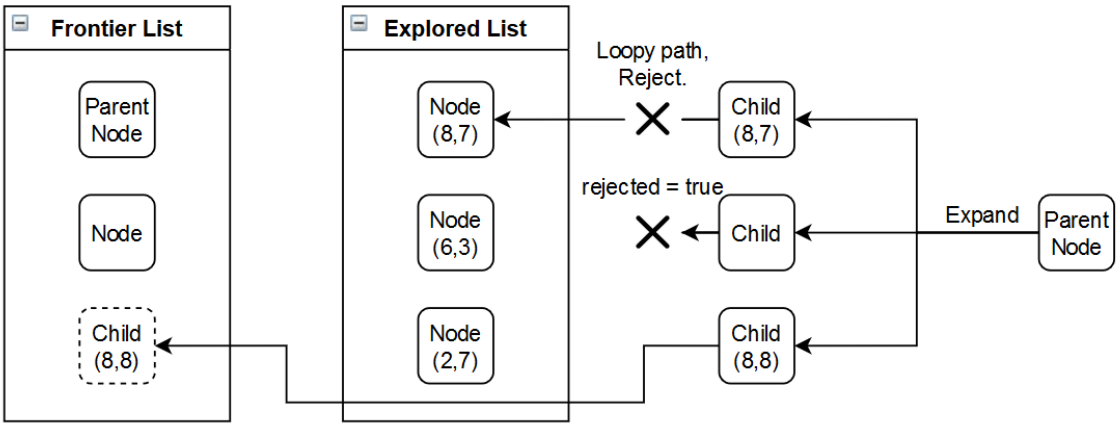

Figure 4.1: Frontier list filter (Node (8,8) is not in frontier list so it passes)

The final for loop is to determine which node is to go into the frontier list as shown on the diagram above. There are two conditions if any of them met, the children will not be entered in frontier and would not be considered for future expansions. The conditions are that: -

1) Their "remove" status is set to false.
2) No node in explored has the same state value as the child.
3) No node in frontier has the same state value as the child.

Function expandNode() might reject a child node early on already by setting "remove" to true. Eligible children are then added at the back of the frontier list sitting at the bottom priority. BFS expansion pattern is parents first then children since nodes in the horizontal line is expanded first before going down, which make this algorithm suitable to be called BFS.

Whilst each node is being checked for frontier eligibility, the goal check is also being run. In BFS, the search stops when the goal node is found. Program will then start looping back from the beginning until a goal is found. In the code, if the child's state is the same as the foodLoc variable or goal state defined in step 1, goal found flag is set to be true and the program will end the search loop to prepare for the final step where it will calculate the solution.

```
# Program ends here if it gets stuck and won't return solution
if len(frontier) < 1:
    print("No solution can be found! Stopping search...")
    game_ended = True
```

Figure 4.2: Alternate goal test where the program exits if the search gets stuck

In the case where the frontier empties out, it means that the algorithm cannot find any possible moves anymore therefore making it stuck. Program will end abruptly while returning no solutions. (Message is printed in anaconda console)

Step 4: Generate solution

```
def compileSolution(endNode, searchTree, currentDirection):
    opposites = {'n': 's', 'e': 'w', 's': 'n', 'w': 'e'}
    solution = [currentDirection]
    parentID = endNode["parent"]
    while parentID is not None:
        directionFacing = "nswe".replace(endNode["actions"][0], "").replace(endNode["actions"][1], "").replace(
            endNode["actions"][2], "")  # ['n','s','w']
        directionFacing = opposites[directionFacing]
        solution.insert(0, directionFacing)
        for x in searchTree:
            if x['id'] == parentID:
                endNode = x
                parentID = x['parent']
                break
    return solution
```

Figure 4.3: Code implementation of step 4

To generate the solution, we separated this section into a separate function called compileSolution(). Its parameters are the end node, the search tree which should be generated from the algorithm and the end node's current direction. The function will start at the end node, and from there utilize "parent" data that is stored in the nodes from the search tree to then go up until it is at the top. That is why the while loop stops if parentID is none meaning it has reached the initial node which does not have any parent nodes.

There is one more issue to address, however. Since there is no information of the current direction inside the nodes itself, we needed to play around with each node's "actions" variable. For example, let us say a node has "[n, s, w]" as its available actions, in this scenario it would be facing opposite of east (west) as east is not listen in the list and going behind is not an available action in the search algorithm. The end results are a list of "nswe" characters which will be used for the solution.

Solution 2 (Greedy Best First Search Algorithm)

A quick summary on how the Best First Search operates is that unlike Breath First Search, it does not blindly expand nodes. Informed search typically will need additional information before executing the search and in the case of Best First Search we will need heuristic information for it to properly run. Our heuristic function will be measured by the distance of tiles from the head to the food.

This script will be built similarly with the Breath First Search Algorithm in solution one with components such as frontier list and search tree etc. It has slight modifications however to accommodate Greedy Best First Search which is that the explored list is removed. During testing we found out that loopy paths would most likely not occur in the program, which would make it unnecessary. Since the steps of how the algorithm operates are like solution one, the explanation will only highlight the major changes to remove redundancy.

Step 1: Defining initial node and initializing frontier list

```
def run(self, problem):
    game_ended = False
    search_tree = []
    frontier = []
    solution = []

    # Initialize the initial node
    initialState = problem['snake_locations'][0]
    foodLoc = find_closest_food(problem["food_locations"], initialState)
    initialNode = {
        "id": 1,
        "state": "{},{}".format(initialState[0], initialState[1]),
        "expansionsequence": -1,
        "children": [],
        "actions": [],
        "removed": False,
        "parent": None
    }
    search_tree.append(initialNode)

    ## Frontier consists of [{Node information}, Distance of food from Node, The snake's current body position]
    frontier.insert(0, [initialNode, calcFoodDistance(foodLoc, initialState), problem['current_direction'],
                        problem['snake_locations']])
    currentNodeID = 1
    nodeExpanded = 1
```

Figure 5: Code implementation of step 1

Initialization of variables are still the same here compared to solution one with few key differences. First is that no explored list is present. Second, the goal state is set so that between all available goals (or food), the program will choose the closest one to the initial state to start searching for thorough the find_closest_food() function. Lastly, is that we are introducing a new variable into the frontier which is the distance of the initial state towards the goal state. This will act as a measurement for our heuristic function to determine which node is to be expanded first.

```python
def calcFoodDistance(foodPos, currentPos):
    # Based of number of tiles
    pcol = currentPos[0]
    prow = currentPos[1]
    distance = abs(foodPos[0] - pcol) + abs(foodPos[1] - prow)
    return distance


def find_closest_food(foodlist, currentPos):
    sortedFoodList = sorted(foodlist, key=lambda x: calcFoodDistance(x, currentPos))
    return sortedFoodList[0]
```

Figure 5.1: Distance calculation related functions

The way we calculate the distance is to get the difference of column position and row position and add them together between the two nodes. Then we just return the distance back. Function find_closest_food() calls the distance finder function and sorts which food state is closer and passes the closest one back. The reasoning on sorting food locations, is that it would better improve the optimality of the search by removing and unnecessary movements like shown in the image below.

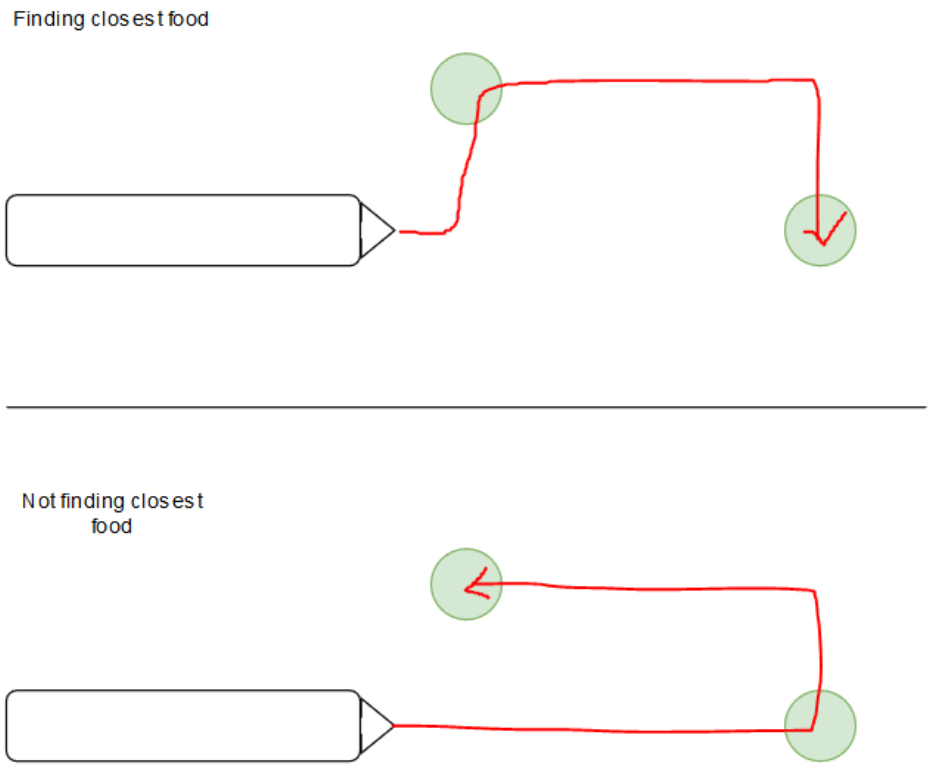

Figure 5.2: Finding closest food vs not finding closest food

The snake below would potentially need to make a U-turn in finding the food meaning more processing.

Step 2: Node expansion, find and return children

```
# Keeps looping until goal is found
while not game_ended:

    snakeLoc = frontier[0][3]
    currentDirection = frontier[0][2]

    # Expand from current position
    children, updatedParentNode, currentNodeID = expandNode(self.maxBound, frontier[0][0], currentNodeID,
                                                            nodeExpanded, currentDirection, foodLoc, snakeLoc)
```

Figure 6: Calling expandNode() with new parameter

Only notable change in this part is that our frontier is now bigger with the introduction of food distance (frontier[0][2] would access the first frontier's food distance) and an additional parameter "foodLoc" that stores the current location of the food as now it is needed for the calculation of the distance rather than just being a goal state. Expansion and finding of children function are then like solution one including the checks for collision and out of bounds that would make node having "removed = false"

Step 3: Update frontier and goal test.

```
del frontier[0]

# Do not consider "removed" nodes in frontier
for child in children:
    search_tree.append(child[0])
    if not child[0]["removed"]:
        frontier.append(child)

# Prioritize low cost paths
frontier.sort(key=lambda x: x[1])
```

Figure 6.1: Frontier check insert and sort

This is the biggest change that was made. To distinct itself with Breath First Search, the frontier list would be important since it determines which nodes will expand on the current loop cycle. Frontier list still acts the same way where the first in the list will be expanded and removed from the list. In the code above, after we deleted the expanded frontier, we do the same check to see that if the child is eligible to be inserted into the frontier by checking the "removed" value to see if its false however we exclude using an explored list here. It should also be noted that we do not require to check against frontier duplicates as it allows the program to jump back up the search tree to find a better alternative route. Before going back, we sort the frontier in an order of whichever has the shortest or least amount of distance first.

Doing this will allow the program to prioritize the better paths that leads the snake closer to the food making it more efficient than solution 1.

```
    # If food location is 0, it means that the current node is goal node therefore program is terminated.
    # However when size of frontier is at 0, the snake is stuck with no possible moves left
    # and program is terminated without a solution being reached
    if len(frontier) < 1:
        print("No solution can be found! Stopping search...")
        game_ended = True
    elif frontier[0][1] == 0:
        game_ended = True
        solution = compileSolution(updatedParentNode, search_tree, frontier[0][2])

return solution, search_tree
```

Figure 6.2: Goal test and alternate no solution goal test

For the goal test we check if the next frontier node is the goal node like solution 1. However over here we check by simply using an if statement to see if that frontier node's distance value is 0 because that means that the frontier node is directly on top of the goal node. The alternative goal test is similar from solution 1 that exists to make the program exists if it gets stuck with no more frontiers left to expand. (Message is printed in anaconda console)

Step 4: Generate solution

Operates the same way as step 4 in solution 1 above.

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

Results for both Uninformed and Informed versions of the problem is achieved successfully with the snake being able to locate its food and continuously search for the randomly generated food with the added bonus for the informed version being able to determine which food is the closest and opt for the closer food as opposed to the further one when 2 foods are generated at once.

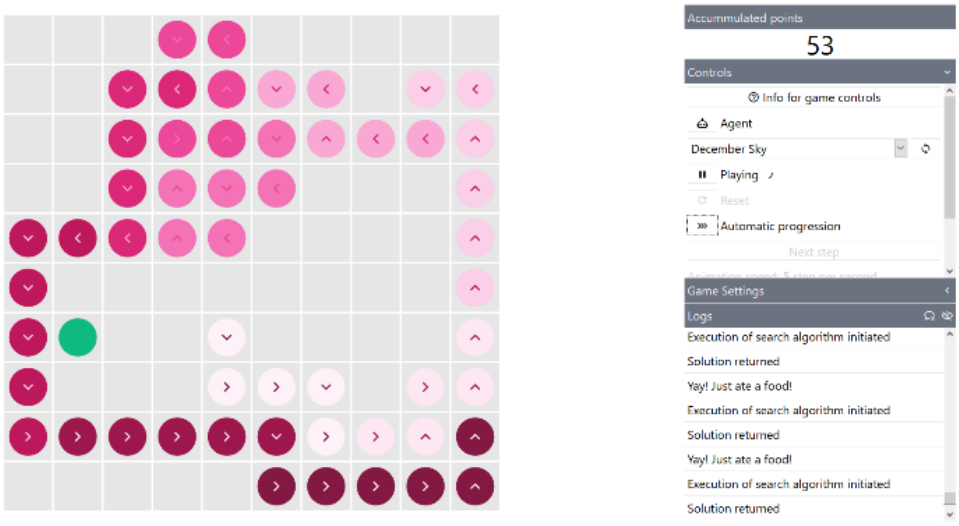

Figure 7: Informed search reaching recorded high score of 53

The average amount of food that the uninformed version of the game can consume varies but is usually at the range of 20 and above on some occasions reaching 40+. The informed would achieve similar point results but with the food location implementation it averages a bit higher than BFS which is around the high 30s and above 40s range. It even managed to reach a recorded high score of 53 shown in the image above. However, since both search algorithms are not complete, getting all the points would be near impossible as both versions would continue to run before it corners itself and stays stuck since the program exits while returning an incomplete solution before being able to escape being trapped by the snake's own length in time.

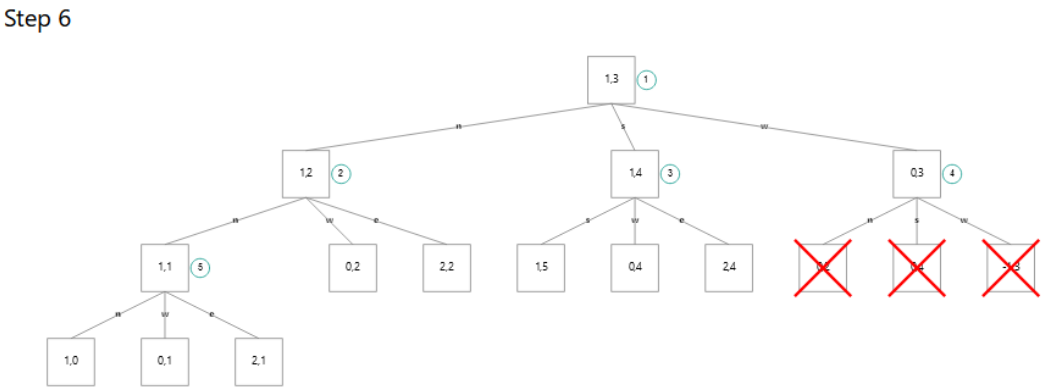

Figure 8: Breath First Search tree in step 6

Using Uninformed Search, the flow of the game can be choppy at times and a little jittery in terms of movement. This is because the BFS search takes quiet a lot of time sometimes in most of the searches as it blindly searches to find the end goal. If the food is already nearby,

the program has no issue in searching with figure 8 above being the average size of the search tree for those steps. However, in a situation where the snake gets longer and more restricted and the food is far from the head, the choppiness will be much more noticeable with even the program freezing for a moment in some instances.

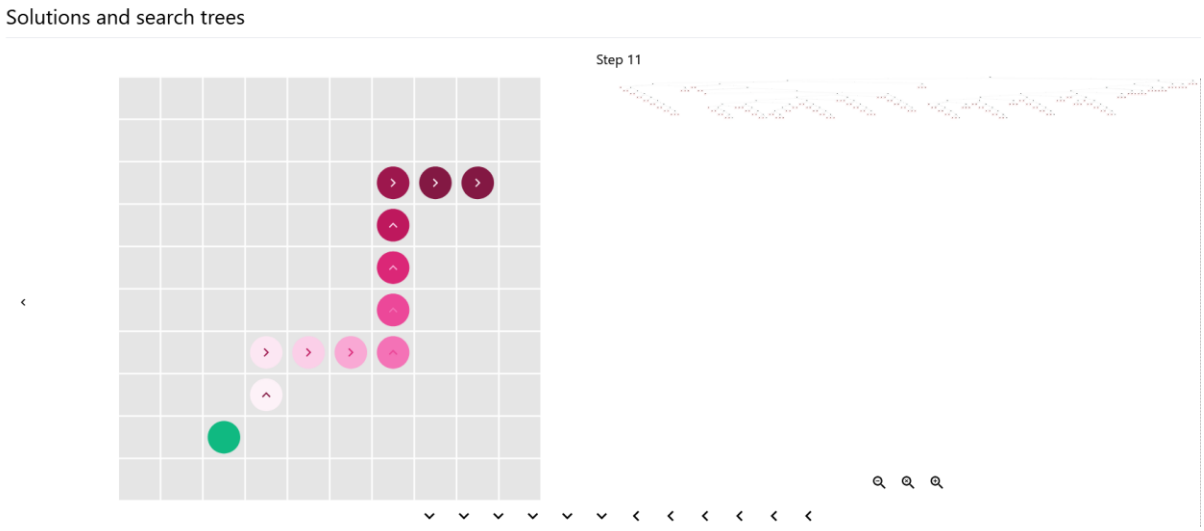

Figure 8: Breath First Search tree in step 11 + snake position

Figure 8 proves the point that was made that the search algorithm is slow at times. At step 11, the algorithm must search almost every possible node which cause the generation of a search tree that huge. It has many unnecessary expansions due to the inefficiency of BFS in this case. In some of the test runs, the algorithm took close to a minute to generate the solution for the next step.

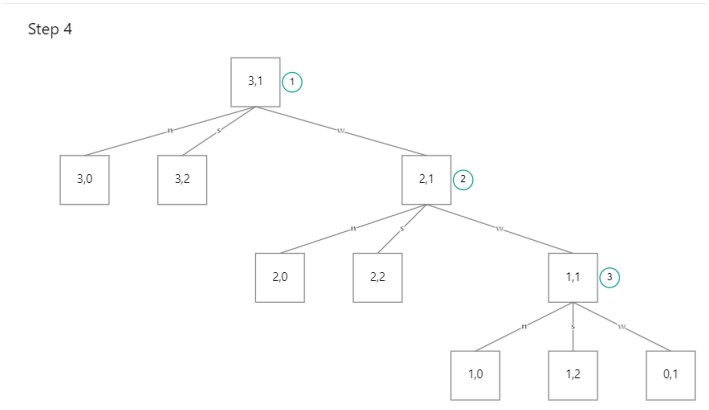

Figure 9: Greedy Best First Search tree in step 4

When it comes to using Informed Search for the game, the flow of the game is extremely smooth as the snake is able to locate the food easily because by using Greedy Best First Search Algorithm, the solution can be found extremely fast when compared to using the

Breadth First Search algorithm, therefore, the snake is able to locate the food as quickly as possible and determine the shortest path it has to take in order to reach the food. There still some jitteriness occurring in some occasions however, but it is less frequent and usually happens when the snake is reaching almost half the size of the map which restricts its movement quiet a lot which is understandable. Furthermore, because the snake will only use the shortest path required to reach the food and not move in a random direction, it has a lesser risk of touching its own body and thus ending the game early due to reducing unnecessary turns. Search trees in informed search are also relatively smaller than BFS's trees as it expands nodes less frequently.

evaluate the result

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

Comparison between Informed Search and Uninformed Search

Basic of comparison	Greedy Best First Search	Breadth First Search
Basic knowledge	Additionally, with goal knowledge, it has the heuristic knowledge which was the distance of snake and food in this case which allowed it to make better decision making in terms of selecting which node to expand first.	Searches blindly with only the knowledge of the goal present.
Optimality	Optimal as the algorithm runs smoothly whilst also being able to determine the shortest path needed to reach the food.	Not optimal. BFS searches nodes one by one until goal is found
Time Complexity	The time taken for GBFS to run depends on the heuristic function that it is given. However overall, it is better than BFS but not the best compared to other Best First Searches.	The time complexity of a breadth-first search algorithm takes linear time, or O(n), where n is the number of nodes in the tree. This algorithm works faster if the goal is close to the source.
Completeness	Incomplete. Although faster and more optimal than Greedy Best First Search, it's still an incomplete search algorithm since the snake can still get stuck on a solution	Incomplete. The snake can still get stuck on the solution

Conclusion

GBFS do really perform better than BFS, however both searches are incomplete. GBFS searches quicker than BFS but does not necessarily guarantee a complete solution return each time the same way BFS does. A*, a complete best first search could do the job but would require more memory and time to process. Every search has its pros and cons and choosing one is in the matter of what resource is available and constraints