SCHOOL OF ENGINEERING AND TECHNOLOGY

ARTIFICIAL INTELLIGENCE – ASSIGNMENT 1

ACADEMIC SESSION: MARCH 2021

CSC 3206 ARTIFICIAL INTELLIGENCE

DEADLINE: FRIDAY , 22 MAY 2021

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 but before Sunday (23/05) 11.50pm will be deducted 3 marks from your final mark.
- Work handed after Sunday (23/05) 11.50pm will be regarded as a non-submission and marked zero.

Academic Honesty Acknowledgement

I, Brysen Poi Yong Zhen 17119363,

I, Dylan Ng Jia Jun 18031641,

I, Loh Khay Wen 18017889,

verify that this paper contains entirely my own work. I 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, I have not copied or inadvertently copied ideas, sentences, or paragraphs from another student. I realize the penalties *(refer to page 16, 5.5, Appendix 2, page 44 of the student handbook diploma and undergraduate programme)* for any kind of copying or collaboration on any assignment.

Brysen, Dylan, Loh Khay Wen / 21/5/2021

Contents

Problem Description

The snake game was popular back in the late '90s whereby the players have control over the movement of the snake in the 2D Cartesian plane based on the four directions: North, South, West, or East. The goal of the entire game is to eat the food that is generated randomly in the 2D Cartesian plane while keeping in mind that hitting the wall of the maze and eating oneself is not allowed. Thus, the problem described in the given assignment is to assist the snake to search for the ideal path to get the food, via the means of the creation of an agent through the incorporation of knowledge of artificial intelligence. Thus, to solve the problem, we will create an agent and implement two different types of search algorithms to search for food, mainly using the uninformed search algorithm and the informed search algorithm. The uninformed search algorithm that we will be using is the Breadth-First Search algorithm while the Greedy Best First Search algorithm is used for the informed search algorithm.

Defining the requirements of the assignment

For the scope of this assignment, we have managed to develop the agents for the snake game to be played using the two proposed search algorithms under three conditions:

1. A non-increasing snake length with a fixed length of one upon eating the food that is spawned one at any time, with the achievement of at least 15 points.
2. With a starting length of one, the snake increases its length with food, upon eating the food that is spawned one at any time, with the achievement of at least 10 points.
3. With a starting length of one, the snake increases its length with food, upon eating the food that is generated twice when no food is available on the maze, upon eating them with the achievement of at least 10 points.

Problem Formulation

Before proposing a suitable search algorithm for the snake to achieve its goal in terms of locating the food, there is an additional significant piece of information that has to be understood clearly. The state space defined for the snake game is a 2D Cartesian plane, whereby it has two coordinate axes which will provide us the value for the location of the snake and the food.

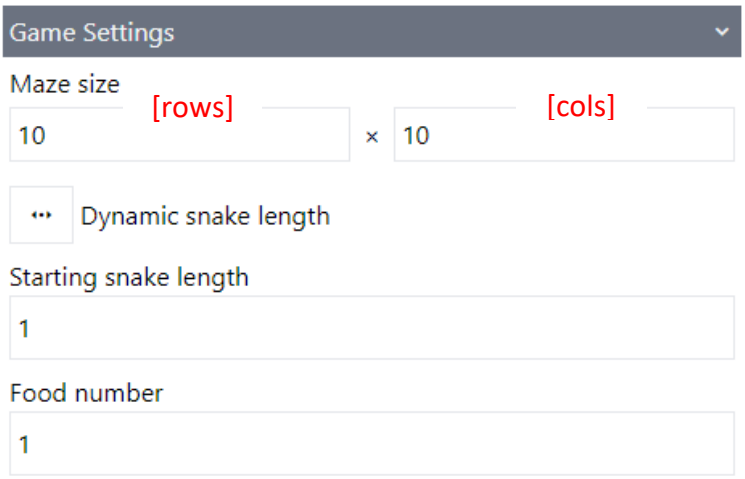


Figure 1 Game Setting also known as State Space

The state space for the snake game is dynamic, as it visualizes the maze according to the user's input values for the number of rows and columns for the maze. Based on the image above, there are two text fields that accepts integer values from the user input, and these will recognize the values and translate the state space according to it. The first text field depicts the rows while the second depicts the columns for the state space. Therefore, based on the above example, there will be 10 rows and 10 columns for the state space of the snake game.

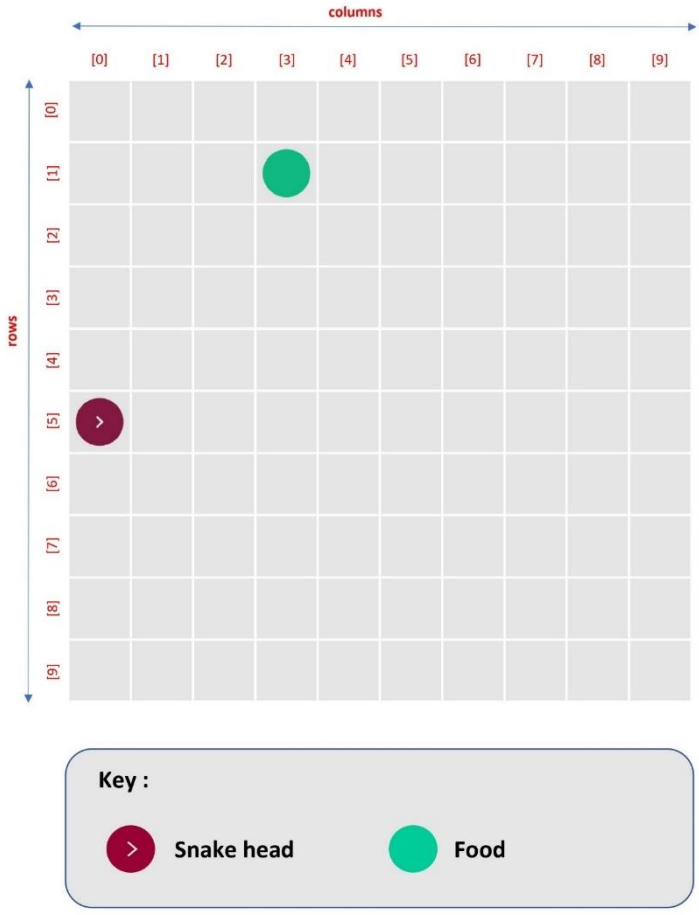


Figure 2 10 X 10 Maze (State Space), Snake's head (Initial State), Food (Goal)

Based on the visualization above, the state space is created for a 10x10 maze. The X-coordinate axis acts as the columns, and its values depict the location of the snake and food in terms of column-wise. The Y-coordinate axis acts as the rows and its values depict the location of the snake and food in terms of row-wise. The range of the values for both axes is from 0 to 9, and using these values together creates coordinates that provide us the exact location of the snake and the food which can be used for our search algorithms.

For the problem to be well-defined, there are five main components used, mainly the **state, actions, transition model, goal test, and path cost.** These five main components that are described below are used to define the snake game's problem in this assignment.

- **Initial state** – The initial state for the snake game can be described as the current location of the snake body in the Cartesian plane before the algorithm is executed. As we know, the length of the snake body can increase whenever it has eaten the food, however, this setting applied on the snake can be modified to either the snake having a dynamic or fixed length. Nevertheless, if dynamic was chosen, the coordinates of the new snake body have to be kept. Therefore, the initial state would be a list containing the location of the snake's body. For example, In[[0,0],[0,1],[0,2]].
  If the length of the snake body is fixed to 1, then the initial state will have only 1 value. For example, In[[1,1]], no matter how much food the snake has eaten.

- **Actions** – The actions that are available for the snake to move, for instance, to the west, east, north, or south. A series of checks will be performed to ensure the snake takes the most appropriate action. If there 4 actions available for a fixed length snake, the applicable actions from the state In[[1,1]] are {Action([0,1], west), Move([2,1],east), Move([[1,0],north), Move([1,2],south)}.

- **Transition model** – Transition will lead to a result where it is supposed to be. In the snake game, we will use a function to return a collection of actions and states that are available, enable us to proceed to the next step with the result we have received. For an instance, Result(In[[1,1]], Move([[1,0],north])) = In[[1,0]]), after the snake has moved towards north direction, its current state has become In[[1,0]].

- **Goal test** – The food represents the goal in this problem. Therefore, the goal test is to determine whether the snake has found the food.

- **Path cost** – In the snake game, each action consumes 1 step of the path cost.

Breadth-First Search (Uninformed Search)

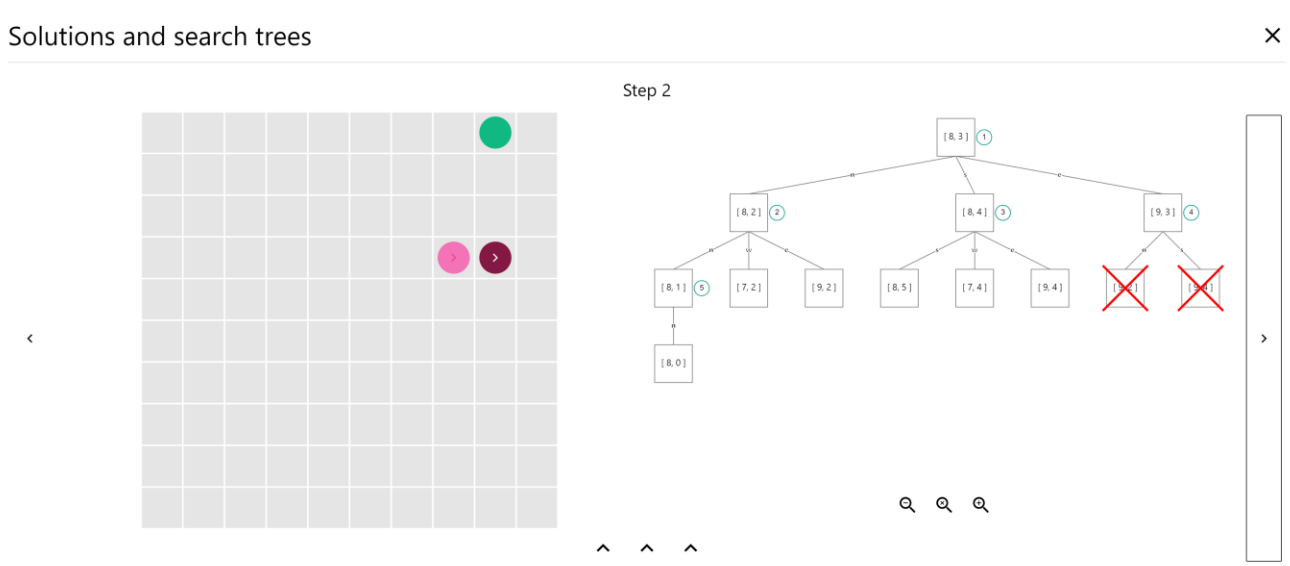


Figure 3 Solution and Search Tree (Breadth-First Search)

The figure above illustrates the search tree together with its solution for the current state of the snake and the food. Given the snake's head is located at the coordinate of [8,3] and the food (goal) is located at [8,0]. The search tree depicts the sequence of reaching the goal state from the initial state based on the sequence number provided in the search tree. For the above figure, the node expansion is based on the uninformed search algorithm applied, which is the breadth-first search. The implementation of the code is discussed below.

```
search_tree = []
frontier = []
explored = []
solution = []
found_food = False

#Insert first node to frontier and search tree
search_tree.append({"id":1,"state":snake[0],"expansionsequence": -1,"children":[],"actions":[],"removed":False,"parent":None})
frontier.append({'id':1,'state':snake[0],'expansionsequence': -1,'children':[],'actions':[],'removed':False,'parent':None})
```

Figure 4 Code Initialization

Firstly, empty lists for appending nodes are created, namely :

- "search_tree" – to append all of the nodes for the search tree of the problem
- "frontier" – to append nodes that are at frontier level
- "explored" – to append nodes that are already expanded
- "solution" – to append directions for the snake's head to move towards its goal,

while the "found_food" variable is set with a 'False' value which will be used as a flag to identify if the food (goal state) is reached and if the searching operation is to be continued.

Based on the figure above, we append the first node into the "search_tree" and "frontier" lists. This is because the first node represents the current location of the snake's head, which is also defined as the initial state and the only frontier in the search tree.

```
expansion_seq = 1
while not found_food:
    # If all node in frontier have been explored but food still not found, then break out the loop
    if len(frontier) == 0:
        break
    frontier[0]["expansionsequence"] = expansion_seq
    # Get the index of current expanding node in search tree
    index_of_current_state_in_search_tree = next((i for i, state in enumerate(search_tree) if state['id'] == frontier[0].get('id')), None)
    search_tree[index_of_current_state_in_search_tree]["expansionsequence"] = expansion_seq
    #Expand the first frontier
    children ,actions = getChildren(frontier[0],col,row,snake)
```

Figure 5 Code to Expand the First Node in Frontier

The code block above represents the implementation of getting the indexes of the current expanding node in the search tree using a while loop for the condition that the food (goal) is not reached. The “expansionsequence” key for the first node in frontier is assigned with the value of expansion_seq declared outside of the while loop. The indexes values obtained by the means of any matching values of the “id” key for states in search_tree and the first in frontier, are used to reference the index in the search_tree list for the value of “expansionsequence” key to be assigned with a similar value of expansion_seq. Next, the first node in the frontier is passed into getChildren() function for the expansion of the node to obtain its children nodes.

```
# get children and actions function
def getChildren(state_to_expand,col,row,snake_loc):
    children = []
    actions = []
    for direction in ('n','s','w','e'):
        if direction=='n':
            child_state = [state_to_expand.get('state')[0],state_to_expand.get("state")[1]-1]
            # check if the snake is already at the top row and
            if (state_to_expand.get('state')[1]!=0) and not (child_state in [s for s in snake_loc]):
                children.append(child_state)
                actions.append(direction)
```

Figure 6 Get All Available Child and its Action

The getChildren() function is added to expand the node received and return its child node coordinate (or state) values and the directional values in “children” and “actions” lists respectively. It also ensures that the node received from the parameter is not the coordinates of the snake body and its not located at the top row of the maze. This is because if the coordinate of the snake’s head is at the top row, the similar code cannot be applied because it will hit itself towards the wall. Keep in mind that based on the directional values, they are used in later functions to assist the snake to move forward accordingly to the direction.

```
def checkRemoved(child_state,explored,frontier):

    removedBool = False
    if not (child_state in [f.get('state')for f in frontier]) and not (child_state in [e.get('state') for e in explored]):
        removedBool = False
    else:
        removedBool = True

    return removedBool
```

Figure 7 Check Loopy Path or Redundant Node

The figure shown above depicts the checkRemoved() function which returns the truth value of local variable “removedBool”. This function is used to check for any loopy or redundant paths in the search tree and remove them based on the truth value of the “if” condition, such that the child node is not in both frontier and explored list.

Figure 8 Update Search Tree and Frontier

This portion of the code is significant as it involves the information of the child nodes to be appended to the relevant specified lists, “frontier” and “search_tree”. Given if the return value is False from the checkRemoved() function, the information of each child nodes under the current parent node is appended into “frontier” and “search_tree” list with an incremented value of the length of the “search_tree” by 1 as the id. Additional information such as the child’s id and its actions are appended into the current parent node in the “search_tree” list.

```
    explored.append(frontier[0])
    # delete expanded state
    del frontier[0]
    expansion_seq += 1

if search_tree[-1].get('state') == [food[0][0],food[0][1]]:
    goal = search_tree[-1]
    goal_path = [goal]
    while not goal_path[-1].get("parent") == None:
        for s in search_tree:
            if s.get("id")==goal_path[-1].get("parent"):
                goal_path.append(s)
                break

    for g in range(1,len(goal_path)):
        index = goal_path[g].get('children').index(goal_path[g-1].get('id'))
        solution.append(goal_path[g].get('actions')[index])

    solution = list(reversed(solution))
```

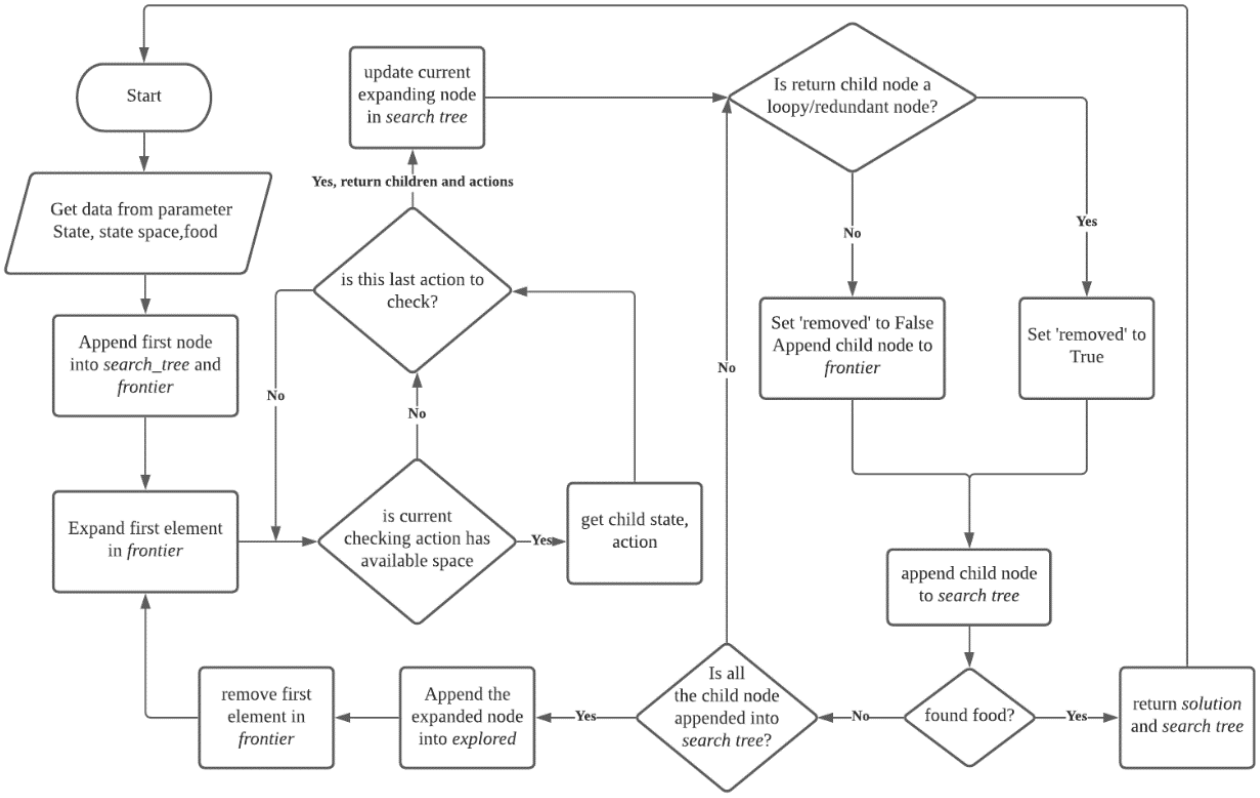
Figure 9 Perform Backtracking to get Solution

Since the information of the expanded node has been added into relevant lists, we append the expanded node into the explored list, subsequently delete it from the frontier list, while incrementing the value of “expansion_seq” by 1 to indicate the expansion sequence of the next node to be expanded.

The code block below it represents the goal test to determine if the last node in the search tree is the food (goal). If the goal is found, we put the node in a list and perform a backtrack by comparing “id” key of parent node with “parent” key of goal node to get the parent node’s information of the goal, until it reached the first expanded node.

Then we start iterating on the second index onwards of the “goal_path” list to get the index of the “actions” key based on the parent node id value as the index for the child nodes before appending it into the “solution” list. The “solution” list is reversed back because initially, the code performs backtracking.

Breadth-First Search (Uninformed Search) Flowchart



Flowchart 1 Breath-First Search (Uninformed Search)

The flowchart above shows a clearer flow and idea of how our Breath-First Search Algorithm works.

1. Starting from the beginning, the algorithm will get the essential information that is required to run the algorithm including, state space (maze size), snake location (state, direction), and food location (goal).

2. Next, it will append the information of the snake's head into the search tree and frontier as a first node to start the node expansion.

3. Moving onwards, the algorithm will start to expand the first element in the frontier which we appended in the previous step.

4. When the algorithm is expanding the first element (node) in frontier, the algorithm will check 4 directions which are east, west, north, and south from the state of the expanding node whether it is available.

5. After checking all the directions, the current expanding node will be updated by appending the available child state and its action into the "children" key and the "actions" key in the dictionary.

6. Then, all the available children return in the previous step will be checked, one by one, whether it is a loopy or redundant node. If yes, the “removed” key will be set to TRUE when appending the child node into the search tree only. If no, the “removed” key will be set to FALSE and it will be appended into the search tree and frontier for the purpose to expand in the future.

7. Lastly, the algorithm will check whether the child node added into the search tree in the previous is equal to the food location. If no food was found, append the expanding node (parent node) into the explored list and delete it from the frontier.

8. If yes, food has been found, return the search tree and solution.

In this algorithm, there is a large chunk of while-loop code repeating on it which is from **Step 3 to Step 7**. This while-loop will only be terminated when the food has been found in **Step 8**.

What made uninformed search into an informed search?

The major difference between Breadth-First Search Algorithm (uninformed) and Greedy Best First Search Algorithm (informed) is that Greedy Best First Search introduces a function that assists the snake to reduce the amount of node expansion while finding for the food. In Breadth-First Search Algorithm, all the available nodes will be expanded vertically, level by level in the search tree until the food is found. However, a function to calculate the distance between the food and snake's head is implemented in Greedy Best First Search whereby it will only expand the node that is closest to the food. Both the algorithms will return the same number of steps to move, but Breadth-First Search Algorithm will take the higher number of expansions during runtime. As a result, informed search has better performance and higher efficiency compared to uninformed search.

Comparison between Breadth-First Search and Greedy Best First Search

The figures below show the search tree and solution return by both algorithms. As we can see that both algorithms return the same number of steps which is 5 steps but in a different direction. Greedy Best First Search moves in the manner of diagonal (zig-zag), however, Breadth-First Search moves in the manner of vertically then horizontally.

The biggest difference between both algorithms as we discussed in the previous section is the number of expansions. In Greedy Best First Search, 5 nodes to be expanded in order to find the food. However, in Breadth-First Search, 23 nodes need to be expanded to find the food. Thus, in terms of performance, Greedy Best First Search will take a shorter time and consume less memory in the search tree to find the food compared to Breadth-First Search.

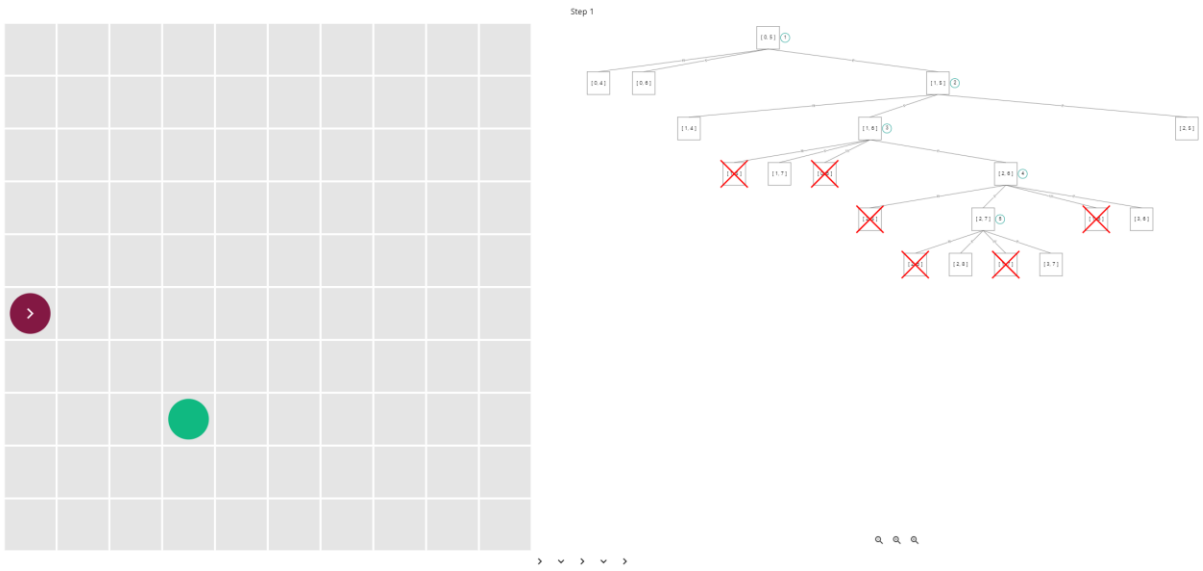


Figure 10 Number of Steps and Expansions in Greedy Best First Search (Informed Search)

Figure 11 Number of Steps and Expansions in Greedy Best First Search (Informed Search)

Greedy Best First Search (Informed Search)

When the length of the snake is less than 18, we will be using the straight-line distance between snakehead and food by using the formula given below:

$$d = \sqrt{((x\_2-x\_1)^2+(y\_2-y\_1)^2)}$$

Hence, the movement of the snake going for the food diagonally (zig-zag) can be seen as in the figure below.

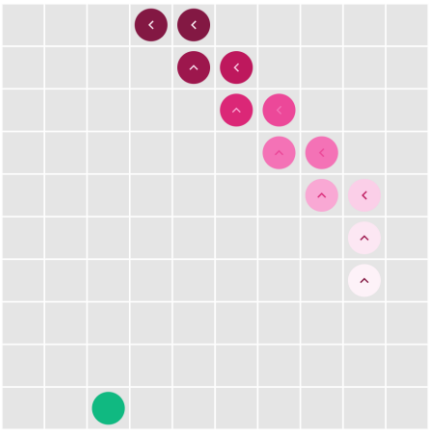


Figure 12 Zig-Zag Movement using Straight-Line Distance

When the length of the snake is more than 18, we will not use straight line distance. This is because when the snake is long, it will occupy more space in the maze. Therefore, running diagonally (zig-zag) using straight line distance will make the body of the snake loose-fitting instead of staying close-packed which will increase the chances for the snake to trap itself. Therefore, when the length of the snake is more than 18, the snake will first move through the rows (vertically) then only columns (horizontally) or vice versa, based on the direction of the snake by using the subtraction to calculate the row distance and column distance.

Two figures below show the differences between the two types of calculation used for the snake to move.

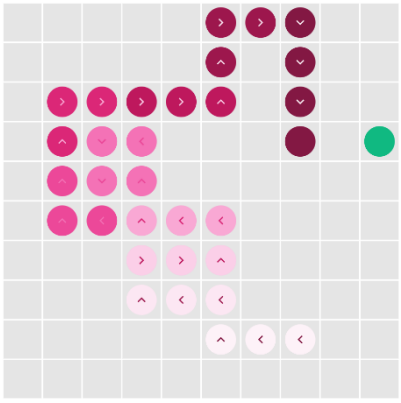


Figure 13 Closed-Packed w Activating Row Distance and Column Distance

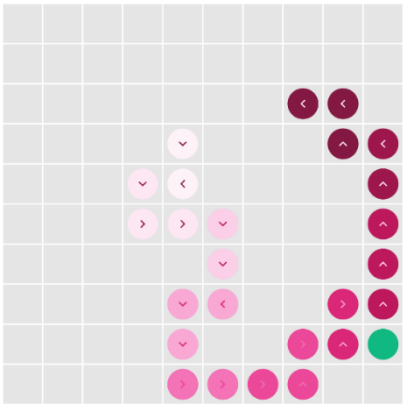


Figure 14 Loose-Fitting w/o Activating Row Distance and Column Distance

The figure on the left uses the straight-line distance calculation, which shows a scattered pattern while the figure on the right uses the row distance and column distance calculation, only applied after the length of the snake is more than 18, shows a much close-packed pattern.

Solution for two food generated

In order to fulfil one of the challenges regarding the spawning of two food as stated previously in the documentation, we have included the solutions as shown below.

```
food = problem.get('food_locations')
```

Figure 15 Getting food locations into a list

Before the searching algorithm is executed, the location of the food has to be known. Therefore, a list containing the coordinates of all the food in the maze are added into the "food" variable.

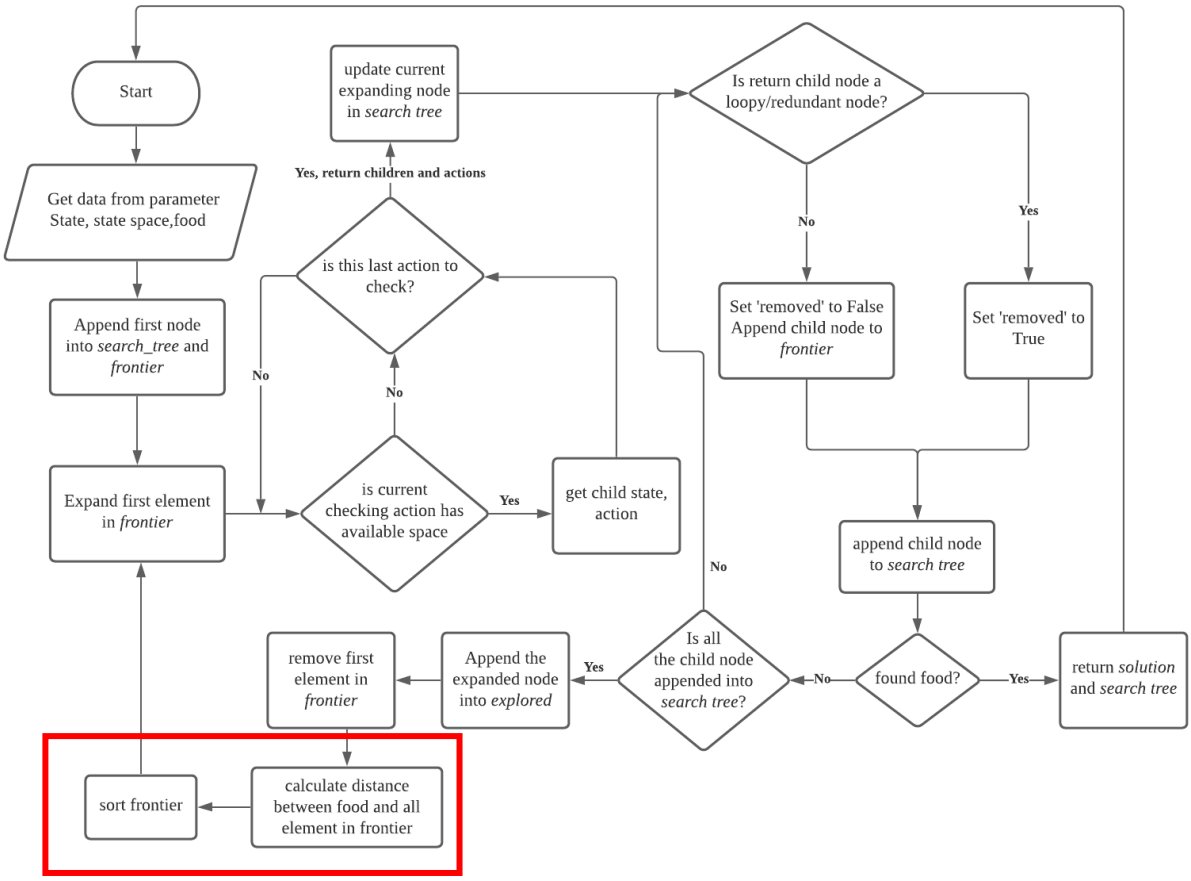
```
# Calculate which food is the nearest
for index in range(len(food)):
    distance = funcDist(food[index],snake[0])
    food[index].append(distance)

# Sort food nearest to snake head (for more than 1 food)
food = sorted(food,key=lambda f:f[-1])
```

Figure 16 Sorting the nearest food to snake head according to the calculated distance

In order to maximise the functionality of the code in Figure () for the stated challenge, we have included the block of code as shown in the above Figure(). Since the main objective of the assignment is to obtain the ideal and shortest path cost for the snake to get to the food, this block of code is significant to achieve this challenge when there is two food. First, the code calculates the distance of every food items in the "food" list using the funcDist() function and append the result accordingly to their index. Lastly, we sort the elements in the "food"

list in an ascending order based on their distance values. This will be used in the search function to help the snake to get to the nearest food available in the maze.

Greedy Best First Search (Informed Search) Flowchart



Flowchart 2 Greedy Best First Search (Informed Search)

The flowchart shown above is almost similar to Flowchart 1 shown previously, just that it will take another two more steps which are to calculate the distance between the food and all nodes in the frontier and sort it in ascending order. By doing this, we can make sure that the node that is closer to the food will be expanded first.

Problem Faced for Uninformed Search Algorithm and Informed Search Algorithm

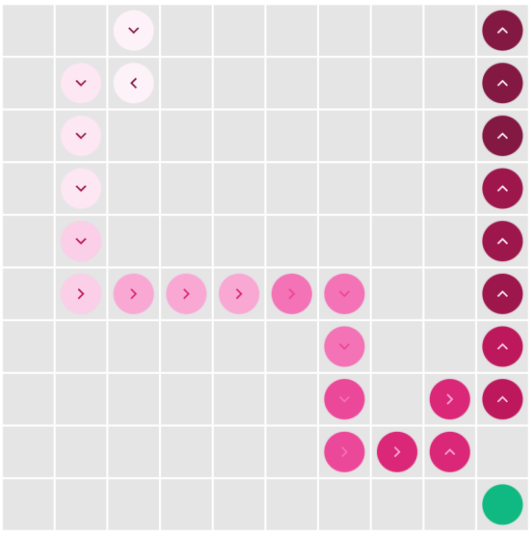


Figure 17 Example of Closed Area and Freeze Situation

Based on the visualization above, the snake could not find the food's location because the snake's body has blocked the path for the snake to search for the food and formed a closed area. A formation of a closed area will trap the snake inside the closed area, but the food is located outside the closed area. The agent will freeze here and would not move at all, causing the game to technically pause without being paused by the user. It will then show "IndexError: list index out of range" under "Logs" in the frontend. This is because the frontier currently is empty due to all nodes are expanded and the food remains not found.

First proposed solutions:

Our first approach was whenever the food cannot be found because of the block by the snake's body, it will run the code as shown in the figure below. This code will make the snake keep moving upwards or towards the 'North' ('denoted as N') direction until it can either find the food or it can no longer move upwards. It will then move towards the left and so on until the food can be found by the algorithm causing it to move towards the food or make the game over by either bite itself or crashing the wall. But this approach was not the best solution because it was not an intelligent approach to do so as it will move towards the direction that hardcoded by us which is to the north, west, south then east which will eventually trap itself very soon.

```
def checkWhatStepSnakeCanMove(snake_loc,col,row,current_direction):

    new_move = []

    if snake_loc[0][1]!=0:
        if not ([snake_loc[0][0],snake_loc[0][1]-1] in [s for s in snake_loc]):
            new_move.append(snake_loc[0][0])
            new_move.append(snake_loc[0][1]-1)
            new_move.append('n')
            return new_move
```

```
if current_direction == 'n':
    new_move.append(snake_loc[0][0])
    new_move.append(snake_loc[0][1]-1)
    new_move.append('n')
    return new_move
```

Figure 18 Hardcoded to Force Snake Move to Any Direction

Final proposed solution:

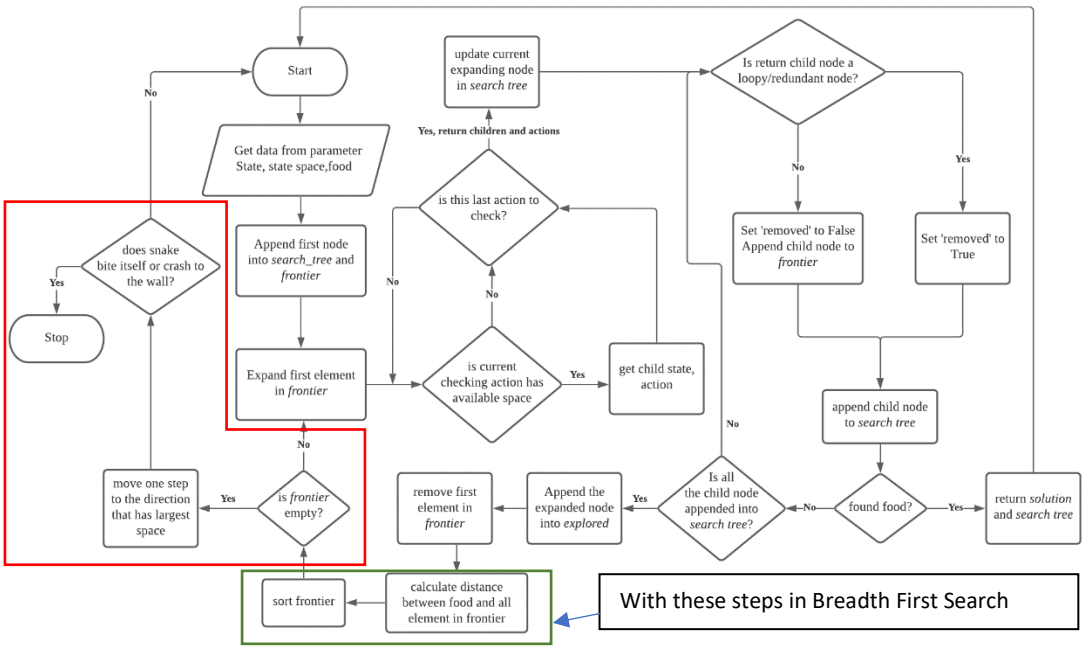
Similarly, when the frontier is empty due to the food is not being found, it will run the code as shown in the figure below. This code will let the snake move towards the direction that has the largest space for it to move, **this will prolong the life of the snake which makes it move more steps, to such an extent that it has a higher possibility to free itself from the trap or closed area.**

```
def findGreatestSpaceToMove(snake_loc,col,row):
    new_move = []
    for direction in ('n','s','w','e'):
        count = 0
        increment = 1
        if direction == 'n':
            while True:
                if not ([snake_loc[0][0],snake_loc[0][1]-increment] in [s for s in snake_loc]):
                    if not (snake_loc[0][1]-increment) == -1:
                        count+=1
                        increment+=1
                    else:
                        break
                else:
                    break
            new_move.append([snake_loc[0][0],snake_loc[0][1]-1,count,'n'])
```

```
else:
    new_state_and_action = findGreatestSpaceToMove(snake,col,row)
    solution.append(new_state_and_action[-1])
    del search_tree[1:]
    search_tree[0].get('children').clear()
    search_tree[0].get('children').append(2)
    del search_tree[0].get('actions')[:]
    search_tree[0].get('actions').append(new_state_and_action[-1])
    search_tree.append({'id':2,'state':[new_state_and_action[0],new_state_and_action[1]],'expansionsequence':-1,'children':[],'actions':[],'removed':False,'parent':1})
```

Figure 19 Function to ensure Snake to Make to Largest Space

Flowchart for the solution



Flowchart 3 Solution to Solve Freeze Situation in Closed Area

The flowchart shown above is almost similar to Flowchart 2 shown previously. However, the only difference is that the algorithm will check whether the frontier is empty before expanding the first element (first node) in the frontier. If not empty, then the algorithm will continue to expand the node, if it is empty, the algorithm will force the snake to move one step to the direction that has the largest moving space. If the snake bites itself or crashes to the wall, it will end the game that is supposed to be instead of freezing the game and throw an error.

Average Score of Breadth-First Search with Dynamic Snake Length

Round / Variant	1	2	3	4	5	6	7	8	9	10	Average Score
Freeze in Closed area	15	24	21	17	23	15	34	13	34	26	22.2
First Solution	33	30	25	27	26	29	32	27	29	36	29.4
Final Solution	34	34	43	22	25	30	30	38	36	39	33.1

Table 1 Average Score Comparison between Different Proposed Solution (Breadth-First Search)

Average Score of Greedy Best First Search with Dynamic Snake Length

Round / Variant	1	2	3	4	5	6	7	8	9	10	Average Score
Freeze in Closed area	23	24	13	16	26	27	27	25	37	15	23.3
First Solution	25	28	27	18	23	28	35	21	15	31	25.1
Final Solution	30	17	37	33	37	31	24	40	28	43	32.0

Table 2 Average Score Comparison between Different Proposed Solution (Greedy Best First Search)