**Contributors**

1. **Sunil Mittal (2021sc04968)**
2. **Kirti Karki (<Kindly fill >)**
3. **Vikram Panwar (2021sc04958)**
4. **Indira Saha (2021sc04956)**

**Problem statement**

There are two agents named R1 and G1. Both are searching for a "heart" as shown in the below configuration as “H” that gives everlasting power. Both agents are trying to reach the heart. In this process many obstacles may be encountered to reach the heart. Help them in finding the best path to reach the heart from any arbitrary start positions. [Dynamically fetch the start position while executing the code]

|  |  |
| --- | --- |
| **Scenario I** | **Scenario II** |
| Chart  Description automatically generated | Chart  Description automatically generated |

For the agent R1 the obstacle is the green room. If R1 enters the green room it incurs a penalty of +10 cost and if it uses the red room it incurs a penalty of -10 points. For the agent G1 the obstacle is the red room. If G1 enters the red room it incurs a penalty of +10 cost and if it uses the green room it incurs a penalty of -10 points. In addition to the given cost, for every transition an agent visits incurs a path cost of 1.

For any arbitrary node “n” the heuristic to reach the Heart h(n) is given by the below: *Manhattan distance + Color Penalty where, Color Penalty = +5 if the node “n” and goal node is in different colored room and Color Penalty = -5 if the node “n” and goal node is in same colored room*

Use the Greedy Best First Search algorithm for both the below configurations and interpret which agent works well in which environment. Justify your interpretation with relevant performance metrics.

**Note:** The agents are not competing with each other. You need to run the simulation for both agents in each of the below scenarios separately & submit the results of 4 runs.

**Solution:**

As , given problem is a classic maze problem where agent is to best possible exit opening. The given problem is has below components

1. Agent 🡪 Task of the agent is to find the best possible path
2. Start Node 🡪 Entry point in the maze
3. End Node 🡪 the point to reach

The task of the agent is to reach from start node to end node using the best path possible. There are several algorithms which can be used to find the path. Some of the algorithms are as below :

**Informed Search algorithms** have information on the goal state which helps in more efficient searching. This information is obtained by a function that estimates how close a state is to the goal state. Example: Greedy Search and Graph Search.

**Uninformed Search algorithms** have no additional information on the goal node other than the one provided in the problem definition. The plans to reach the goal state from the start state differ only by the order and length of actions. Examples: Depth First Search and Breadth-First Search.

For Agents to find the best possible path using minimal time and space complexities , there certain frameworks an agent has to conform , one of the framework is called PEAS.

**The PEAS (Performance measure, Environment, Actuator, Sensor)** model is a framework used to describe the characteristics of an agent in a given environment. It consists of four elements:

**Performance measure:**

This refers to the metric or metrics used to evaluate the agent's performance. In the case of the agents R1 and G1 searching for the "heart", the performance measure could be the length of the path taken to reach the heart or the total cost of the path.

**Environment:**

This refers to the environment in which the agent operates, including the physical layout of the environment and any constraints or limitations. In the case of the agents R1 and G1, the environment consists of rooms with different colors, and the agents incur different penalties for entering certain rooms.

**Actuator:**

This refers to the mechanism by which the agent is able to take actions in the environment. In the case of the agents R1 and G1, the actuator could be the ability to move from one room to another.

**Sensor:**

This refers to the mechanism by which the agent is able to perceive and gather information about the environment. In the case of the agents R1 and G1, the sensor could be the ability to detect the colors of the rooms and the presence of the heart.

Overall, the PEAS model provides a useful framework for understanding the characteristics and capabilities of an agent in a given environment, and can be used to design and evaluate the performance of the agent

As , It is requested to use informed search and use “Greed Best First Search Algorithm” , We are also given the cost and heuristic value for each transition. Below is the flow chart explaining the working :

Calculate the heuristic cost and penalty.

Remove it from the open list and put on visited. Save the index of the Node n which has the smallest cost

If n is end node

Terminate the algorithm and use the pointers of indexes to get optimal path

Detected all successor Nodes of n which are not visited

Calculate cost function of each Node

Yes

No

**Implementation :**

Create the maze structure in python as below :

scenario1 = [

        ['R', 'G', 'G', 'G', 'R', 'G'],

        ['G', 'G', 'G', 'R', 'G', 'G'],

        ['G', 'G', 'R', 'G', 'G', 'G'],

        ['G', 'R', 'G', 'G', 'G', 'R'],

        ['R', 'G', 'G', 'G', 'R', 'G'],

        ['G', 'G', 'G', 'R', 'G', 'G'],

    ]

    scenario2 = [

        ['R', 'G', 'R', 'G', 'R', 'G'],

        ['G', 'R', 'G', 'R', 'G', 'R'],

        ['R', 'G', 'R', 'G', 'R', 'G'],

        ['G', 'R', 'G', 'R', 'G', 'R'],

        ['R', 'G', 'R', 'G', 'R', 'G'],

        ['G', 'R', 'G', 'R', 'G', 'R'],

    ]

Define the heuristic function. This function should take in the current cell and the end cell. This function should return the Manhattan distance between the two nodes, plus the color penalty as defined in the problem statement.

def heuristic(pos,matrix):

    #For any arbitrary node “n” the heuristic to reach the Heart h(n) is given by the below:

    #Manhattan distance + Color Penalty

    #where, Color Penalty = +5 if the node “n” and goal node is in different colored room

    #and Color Penalty = -5 if the node “n” and goal node is in same colored room

    # Calculate the Manhattan distance

    manhattan\_distance = abs(pos[0] - end\_position[0]) + abs(pos[1] - end\_position[1])

    # Calculate the color penalty

    color\_penalty = 5 if matrix[pos[0]][pos[1]] != matrix[end\_position[0]][end\_position[1]] else -5

    # Return the total heuristic value

    return manhattan\_distance + color\_penalty

Define the cost function. This function should take in the current node, the neighboring node, and the agent (R1 or G1), and return the cost of transitioning from the current node to the neighboring node.

def cost(current, neighbor, agent):

    if agent == 'R1':

        if neighbor[0] == 'G':

            return 10

        elif neighbor[0] == 'R':

            return -10

    elif agent == 'G1':

        if neighbor[0] == 'R':

            return 10

        elif neighbor[0] == 'G':

            return -10

    return 1

Implement the greedy best first search algorithm as explained in the flow chart above using

Greedy best first search algorithms fall into the category of "best-first search" algorithms, which are algorithms that can use both the knowledge acquired so far while exploring the search space, denoted by g(n)., and a heuristic function, denoted by h(n), which estimates the distance to the goal node, for each node n in the search space (often represented as a graph).

To use the greedy best first search algorithm below parameters are needed :

* Start Node: the start node of the search
* End Node: the end node of the search
* Maze: a dictionary that represents the maze or matrix. The keys are the nodes and the values are lists of the neighbors of each node.
* heuristic: a function that calculates the heuristic cost from a node to the end node.

**Kindly change the name of the function as needed.**

def a\_star\_search(start, end, agent,matrix,total\_iterations):

    # Create a set to store the explored nodes

    visited\_nodes = set()

    #total\_iterations=0

    # Create a dictionary to store the parent of each node, Initial set it to None

    parents = {start: None}

    # Create a dictionary to store the cost of each node, initially cost is 0 for the start node

    costs = {start: 0}

    # Create a heap to store the unexplored nodes

    heap = []

    # Push the start node onto the heap with a cost of 0

    #https://realpython.com/python-heapq-module/

    heap.append((0, start))

    next\_cost=0

    # Loop until the heap is empty

    while heap:

        # Pop the node with the lowest cost from the heap

        #https://docs.python.org/3/library/heapq.html

        total\_iterations=total\_iterations+1

        current = heapq.heappop(heap)[1]

        # If the current node has been explored, continue

        if current in visited\_nodes:

            continue

        # If the current node is the emd, return the path

        #if matrix[current[0]][current[1]] == "H":

        if current==end:

            path = [current]

            while current != start:

                current = parents[current]

                total\_iterations=total\_iterations+1

                if current not in path:

                    path.append(current)

            print("cost: ", next\_cost)

            print (f"Total iterations :{total\_iterations}")

            return path[::-1]

        # Mark the current node as visited

        visited\_nodes.add(current)

        # Expand the current node and look in each possible direction, currently only 4 directions defined

        # Left Right Up and down

        for i, (dx, dy) in enumerate(directions):

            # Calculate the next position

            next\_pos = (current[0] + dx, current[1] + dy)

            total\_iterations=total\_iterations+1

            # Skip the next position if it is outside the grid or is an obstacle

            if (

                next\_pos[0] < 0

                or next\_pos[0] >= len(matrix)

                or next\_pos[1] < 0

                or next\_pos[1] >= len(matrix[0])

                or next\_pos in visited\_nodes

            ):

                #print("Outside the matrix")

                continue

            # Calculate the cost of the next position, every trainsition cost 1 hence add the cost by 1

            next\_cost = costs[current] + 1

            if  matrix[next\_pos[0]][next\_pos[1]] == "G" and agent == "R1":

                next\_cost += 10

            elif matrix[next\_pos[0]][next\_pos[1]] == "R" and agent == "R1":

                next\_cost -=10

            # If the next position is the goal and the agent is G1, add the penalty for the red room

            if matrix[next\_pos[0]][next\_pos[1]] == "R" and agent == "G1":

                next\_cost += 10

            elif matrix[next\_pos[0]][next\_pos[1]] == "G" and agent == "G1":

                next\_cost -=10

            # If the next position has not been explored or the cost of the next position is lower than the current cost, update the cost and parent of the next position

            if next\_pos not in costs or next\_cost < costs[next\_pos]:

                costs[next\_pos] = next\_cost

                priority = next\_cost + heuristic(next\_pos,matrix)

                heapq.heappush(heap, (priority, next\_pos))

                parents[next\_pos] = current

    # Return None if the end position was not reached

    return None

Finally do the simulation and print the simulation result as below :

**Kindly change the name of the function as needed.**

input\_str = input("Please Enter the start position : ")

try:

    list =(str.split(input\_str, ","))

    start\_position = tuple(map(int, list))

    if(start\_position[0]>6  or start\_position[0]<0 or start\_position[1]>6 or start\_position[1]< 0):

        raise Exception("Invalid Input")

except:

        print ("Invalid Input")

else:

    total\_iterations=0

    end\_position = (3, 2)

    path\_\_For\_R1\_Scenario\_1 = a\_star\_search(start\_position, end\_position, "R1",matrix1,total\_iterations)

    print("Time complexity for A\* Algorithm is O(b power d) . which is inline with the given iterations as above")

    print("Space complexity for A\* Algorithm is O(b power d) . which is inline with the given iterations as above")

    print(f"Path for R1 (Scenario 1): {path\_\_For\_R1\_Scenario\_1}")

    end\_position = (2, 2)

    path\_\_For\_R1\_Scenario\_2 = a\_star\_search(start\_position, end\_position, "R1",matrix2,total\_iterations)

    print("Time complexity for A\* Algorithm is O(b power d) . which is inline with the given iterations as above")

    print("Space complexity for A\* Algorithm is O(b power d) . which is inline with the given iterations as above")

    print(f"Path for R1 (Scenario 2): {path\_\_For\_R1\_Scenario\_2}")

    end\_position = (3, 2)

    path\_\_For\_G1\_Scenario\_1 = a\_star\_search(start\_position, end\_position, "G1",matrix1,total\_iterations)

    print("Time complexity for A\* Algorithm is O(b power d) . which is inline with the given iterations as above")

    print("Space complexity for A\* Algorithm is O(b power d) . which is inline with the given iterations as above")

    print(f"Path for G1 (Scenario 1): {path\_\_For\_G1\_Scenario\_1}")

    end\_position = (2, 2)

    path\_\_For\_G1\_Scenario\_2 = a\_star\_search(start\_position, end\_position, "G1",matrix2,total\_iterations)

    print("Time complexity for A\* Algorithm is O(b power d) . which is inline with the given iterations as above")

    print("Space complexity for A\* Algorithm is O(b power d) . which is inline with the given iterations as above")

    print(f"Path for G1 (Scenario 2): {path\_\_For\_G1\_Scenario\_2}")

**Below are the simulation results :**

**Kindly change the capture.**

**Text

Description automatically generated**

In this example, the path cost is calculated by adding the cost of transitioning between nodes (1 point for each transition) and adding a penalty of 10 points if the agent encounters an obstacle (either a green room for R1 or a red room for G1). The heuristic function (**h(n)**) and the Manhattan distance are used in this example, as they are only used to determine the best path and not to calculate the cost of the path.

**Space and Time Complexity of the algorithm**

To calculate the space complexity of the greedy best first search algorithm, we can measure the maximum size of the open list, which contains the nodes that have been visited but not yet expanded. This is done by keeping track of the length of the open list at each step of the search and taking the maximum value.

To calculate the time complexity of the greedy best first search algorithm, we can measure the number of nodes that are expanded. This can be done by keeping a counter that is incremented every time a node is expanded.

To calculate the time and space complexity for the informed search, we can simply run the search multiple times with different input sizes and measure the time and space complexity using the above methods.

**Conclusion :**

This implementation of the greedy best first search function takes in the start and end positions, as well as the agent name ('R1' or 'G1') as input and returns the optimal path from the start to the end as output.

The **space complexity** of this implementation is O(b^d),

where b is the branching factor of the search space (i.e. the average number of children per node) and d is the depth of the search space.

The space complexity is determined by the size of the heap, which stores the unvisited nodes, and the size of the dictionaries, which store the costs and parents of each node.

The **time complexity** of this implementation is also O(b^d), where b is the branching factor of the search space and d is the depth of the optimal solution (i.e. the shortest path from the start to the goal). The time complexity is determined by the number of nodes that are expanded and the time required to expand each node.

To interpret the results, one can compare the length of the paths returned by the search function for each agent and the cost of each path.

For example, if the path for R1 is shorter and has a lower cost than the path for G1, this suggests that R1 performs better in this environment. On the other hand, if the path for G1 is shorter and has a lower cost than the path for R1, this suggests that G1 performs better in this environment.

one can also compare the space and time complexity of the search function for each agent to see which one requires less memory and time to find the optimal path. A lower space and time complexity generally indicates a more efficient search algorithm.