**ASSIGNMENT 2**

**TREASURE HUNT USING UCS AND A\* SEARCH**

**Contents:**

* **Description**
* **Implementation**
  + **UCS**
  + **A\* Search**
* **Instructions**
* **Libraries**
* **Collaboration Details**

**By:**

**Muhid Qaiser 22i-0472**

**Ahmed Zubair 22i-0525**

**Description**

The problem described is a pathfinding task in a dynamic environment represented by an 8x8 grid, with the objective of navigating from a starting point to a treasure located in the southeast corner. The grid contains various elements including safe grids, obstacles, the starting point, the treasure location, and endpoints which act as barriers. Additionally, certain grids are inhabited by wild animals, introducing a probabilistic element where encountering an animal may result in more or less cost being added towards total cost.

The adventurer must use either the Uniform Cost Search (UCS) algorithm or the A\* Search algorithm to find the optimal path with the least cost to the treasure, considering the challenges posed by obstacles, animals, and endpoints. The cost of movement varies depending on the type of forest(grid) encountered, with additional costs incurred for obstacles, animal encounters, and navigating around endpoints.

Each cell represents a specific part of the forest:

0: Safeguard, random cost between 1-3

-1: Obstacle, random cost between 2-4

2: Start

3: Goal

5: Walls, cost 1 for each step

A: Animal, if Killed: 0.8% chance, random cost 2-4, if Survived: 0.2% chance, random cost 1-3

**Implementation:**

**UCS**

I have implemented the UCS (Uniform Cost Search) algorithm for navigating a forest using two different methods, both of which yield correct results. In both approaches, I start by converting the coordinates of the starting position into a 'cell' object, which simplifies storing information about each cell and enhances code readability. This cell object, along with its cost (initially set to 0 since the adventurer starts at this position), is then added to a priority queue as a tuple. Additionally, the coordinates of the starting cell are stored in a set called "explored" to keep track of explored cells.

A while loop is initiated, running until the priority queue is empty or the current cell's coordinates match those of the goal state. Within the loop, the tuple with the minimum cost is dequeued from the priority queue, leveraging the heapq library and operator overloading for comparing cell costs. Heapq library is used to order the tuples in the priority queue correctly via their cost. If 2 tuples have same cost, then the \_\_lt\_\_() operator overloader function is used to correctly order them. If the current cell represents a wall, the algorithm adjusts the current node and increments the cost accordingly. The adjustment is done by moving the current node, 2 steps back and adding 1 cost for each step. If backtracking 2 steps is not possible but 1 step is, then the current node is backtracked to 1 step back with addition of 1 cost to the current cost. The explored set is updated with the current cell to avoid redundant checking of already explored cells.

The algorithm then identifies all valid neighboring cells using a function called get\_neighbors\_nodes() and calculates their costs. If a neighbor hasn't been explored previously, a tuple containing the new cost (created by sum of current cell’s cost and cost of moving towards that neighbor) along with the neighbor is enqueued into the priority queue, ensuring that it's ordered by its new cost. The parent variable of the neighbor is set to the current cell to facilitate backtracking toward the starting cell for pathfinding.

Once the current cell's coordinates match those of the goal, an optimal path is constructed by backtracking toward the starting nodes. The first method utilizes dequeuing from a priority queue via heapq library to explore the minimum-cost next step, offering a highly optimized approach.

The second method employs a dictionary to track the costs of all visited cells, using this information to determine the new cost for neighbors. It revolves around maintaining the cost of visited states at any given time, providing an alternative strategy for exploration.

**A\* Search**

The A\* Search implementation is mostly similar to the UCS search except for when new cost is being calculated. To recap,

I have implemented the A\* search algorithm for navigating a forest using two different methods, both of which yield correct results. In both approaches, I start by converting the coordinates of the starting position into a 'cell' object, which simplifies storing information about each cell and enhances code readability. This cell object, along with its cost (initially set to 0 since the adventurer starts at this position), is then added to a priority queue as a tuple. Additionally, the coordinates of the starting cell are stored in a set called "explored" to keep track of explored cells.

A while loop is initiated, running until the priority queue is empty or the current cell's coordinates match those of the goal state. Within the loop, the tuple with the minimum cost is dequeued from the priority queue, leveraging the heapq library and operator overloading for comparing cell costs. Heapq library is used to order the tuples in the priority queue correctly via their cost. If 2 tuples have same cost, then the \_\_lt\_\_() operator overloader function is used to correctly order them. If the current cell represents a wall, the algorithm adjusts the current node and increments the cost accordingly. The adjustment is done by moving the current node, 2 steps back and adding 1 cost for each step. If backtracking 2 steps is not possible but 1 step is, then the current node is backtracked to 1 step back with addition of 1 cost to the current cost. The explored set is updated with the current cell to avoid redundant checking of already explored cells.

The algorithm then identifies all valid neighboring cells using a function called get\_neighbors\_nodes() and calculates their costs. If a neighbor hasn't been explored previously, a tuple containing the new cost along with the neighbor is enqueued into the priority queue, ensuring that it's ordered by its new cost.

The new cost is calculated by the sum of the current node’s cost and the cost of moving towards that neighbor.

Then comes heuristic cost. I calculate the heuristic cost by calculating the distance between the current node from the goal node using the Manhattan distance formula.

Now I have used 2 methods to add the heuristic cost into the new cost.

In the first method, I add the heuristic cost of the neighbor and subtract the heuristic cost of the current cell. This makes it so actual cost is summed from starting cell to current cell and the heuristic cost of the current cell is also added into the new cost while heuristic cost of the current cell is subtracted to provide the correct new cost. This is efficient as I don’t need to maintain another dictionary containing details of minimum costs of all visited cells.

In the second method, I use a traditional method of maintaining a dictionary containing details of minimum costs of all visited cells to keep track of cost of a cell from the start to that cell and then adding the heuristic cost afterwards, separated from the dictionary, for the tuple that is to be enqueued into the priority queue.

After the new\_cost has been found, if the neighbor has not been explored, it is enqueued into the priority queue with new\_cost aiding in ordering it in the priority queue according to it’s priority. And the parent of the neighbor is set to the current cell to ease backtracking for construction of path.

Once the current cell's coordinates match those of the goal, an optimal path is constructed by backtracking toward the starting nodes. The first method utilizes dequeuing from a priority queue via heapq library to explore the minimum-cost next step, offering a highly optimized approach.

The second method employs a dictionary to track the costs of all visited cells, using this information to determine the new cost for neighbors. It revolves around maintaining the cost of visited states at any given time, providing an alternative strategy for exploration.

**Instructions**

Run the Method1 of MainMethod (BackTracking) of UCS to get the path for UCS algorithm. The parameters of the forest class can be changed to change the rate of spawn of each hurdle such as obstacle, animal, wall etc; the new rates must be floats. The size of the forest can also be changed to any other size using the parameter ‘size’; it must be integer.

The same goes for A\*, run the Method1 of MainMethod (BackTracking) of A\*to get path for the A\* algorithm.

The forest will be random for each run as it is randomized. The adventurer will traverse each forest dynamically, handling each obstacle with correct logic.

**Libraries**

Random library is used to get random cost for each hurdle/danger and to spawn the hurdles\dangers in random cells.

Heapq library is used to aid in priority queue.

**Collaboration Detail**

Muhid Qaiser and Ahmed Zubair both worked on designing the randomness of the grid

Ahmed Zubair worked on UCS algorithm

Muhid Qaiser worked on A\* algorithm