“HELP THE ROBOT” PROGRAM REPORT.

Md Saminur Rahman

GRIFFITH UNIVERSITY S5195943

Contents

1) Problem Overview

The problem- which is known as the “Help the Robot” problem- involves guiding a robot from a starting point to a goal point. It sounds straightforward but given the NxN can become very large means that the possible paths to the goal point keeps increasing. Moreover, the possible movements that can be made by the robot have restrictions and the robot can only make a fixed set of moves. Considering all of these, a good overview of the problem would be that robot needs to go to the goal point from the start point on a given NxN grid using an optimal path.

To further simplify, the ‘’Help the Robot” is a popular search problem in which a robot must navigate from start position to goal position on a grid while avoiding obstacles. The grid is best defined as a square matrix; each cell can be empty (robot can move through it), blocked (cliffs in our case, robot cannot move through it), the start cell and the goal cell.

2) Problem Formulation

The process of deciding what states and goals to consider- given a certain goal, is known as problem formulation. Below is the problem formulation for our “Help the Robot” problem:

- **State space**: All possible positions on the grid will be the state space. Each state is represented as a tuple (i, j), where i and j are the row and column indices of the grid, respectively.

- **Initial state**: The initial state is the starting position (S) of the robot on the grid, represented as a tuple (i, j).

- **Goal state:** The goal state is the target position (G) on the grid, represented as a tuple (i, j).

- **Actions**: The robot can move in 8 possible directions: Left (L), Up Left (UL), Up (U), Up Right (UR), Right (R), Down Right (DR), Down (D), and Down Left (DL). However, the robot cannot move diagonally if there is an obstacle (X) in either of the two adjacent orthogonal directions. Also, the robot cannot move to a position occupied by an obstacle.

- **Transition model**: Given a state's (i, j) and an action a (move direction), the transition model defines the next states (i, j) by updating the row and column indices according to the move direction. The next state must be in a valid position on the grid and not be blocked by an obstacle.

- **Path cost:** The cost of each move depends on the direction. The cost for moving horizontally or vertically (L, U, R, D) is 2, while the cost for moving diagonally (UL, UR, DR, DL) is 1.

3) Proposed Solution

Our proposed solution is implementing a program in python that will start by modeling and formulating the problem. It will then proceed to provide an option of five search algorithms: breadth-first search, uniform cost search, iterative deepening learning, A*search and hill climbing. The use of various algorithms would give us a lot of options to explore optimal solution for a wide variety of inputs. Optimal solution would be the solution would be the one with the lowest path among all the other solutions (Xavier, 2021).

The following list of functions and code (in order) implement the solution. I've included information about the data structures used in each algorithm as well. The algorithms will be discussed in a bit more detail later in the report.

1. Reading input: The code reads the input from an "input.txt" file. It first reads the chosen algorithm, the size of the board (n), and the board itself, represented as a list of strings.
2. Finding start and goal positions: The code iterates through the board to find the start position (S) and the goal position (G).
3. move_validator(): This function checks if a given position (i, j) is within the grid bounds and not blocked by an obstacle (X).
4. get_possible_moves(): Given a position (i, j), this function returns a list of tuples containing the valid next positions and their corresponding move costs. It uses the move_set dictionary to check all 8 possible move directions and their costs.
5. bfs(): This function implements the Breadth-First Search algorithm using a deque (double-ended queue) to maintain a queue of positions to explore, along with their corresponding paths and costs. It then outputs the shortest path, its cost, and the set of explored states.
6. ucs(): This function implements the Uniform Cost Search algorithm using a priority queue (heapq) to maintain the positions to explore, sorted by their path costs. It returns the lowest-cost path, its cost, and the number of explored states.
7. iterative_deepening_search() and depth_limited_search(): These two functions implement the Iterative Deepening Search algorithm. iterative_deepening_search() increases the depth limit in each iteration, while depth_limited_search() performs a Depth-Limited Search for the given depth. They return the path, its cost, and the set of explored states.

8. a_star(): This function implements the A* Search algorithm using a priority queue (heapq) to maintain the positions to explore, sorted by their total estimated costs (path cost + heuristic cost). The heuristic used is the Manhattan distance between the current position and the goal position. It returns the path, its cost, and the number of explored states.
9. hill_climbing(): This function implements the Hill Climbing algorithm, which is a local search algorithm. The function takes two optional arguments: the number of restarts and the probability of making a random move. At each step, the algorithm chooses the move with the lowest heuristic value and may make a random move with a given probability. If the goal is not found after a specified number of restarts, the function returns None for the path and cost.
10. get_move_direction(): This function returns the direction of a move between two adjacent positions on the grid.
11. replace_path_on_board(): This function replaces the path found by an algorithm on the input board with "o" symbols to visualize the solution.
12. write_results(): This function writes the results of an algorithm to an "output.txt" file, including the algorithm name, solution path, path cost, path length, number of explored states, and a visualization of the path on the board.
13. algorithm_mapping: This dictionary maps the algorithm choice to the corresponding algorithm function and name.
14. Algorithms are ran and output is printed: This chunk of code selects the algorithm function based on the input, runs the algorithm, and writes the results to the output file using the write_results() function.

The algorithms used can be broken down into two different categories: uninformed search and informed search. The main difference between them is that informed search has additional information about the goal state whereas uninformed search only knowledge about the goal state comes from the problem statement. In our program only A* and hill climbing are informed search. The additional search information that they use is known as heuristics. In the next section, we'll include experiments where different heuristics are used to see how the results vary.

4) Experiments

The first experiment will be a simple one. We'll use a wide range of input and see what kind of results different algorithms produce. The results will include the following: solution path, path cost, number of explored states, execution time and algorithm specific info like heuristic used.

Case a

5

SRRRR

RXXXX

RXRRR

RRRXR

RRRXG

Algorithm	Solution path	Path cost	Number of explored states	Execution time (in ms)	Heuristic used
Breadth-first search	D-D-R-R-U-R-R-D-D	20	18	0.0	NA
Uniform cost search	D-D-DR-UR-U-R-R-D-D	18	18	0.0	NA
Iterative deepening search	D-D-D-R-R-U-R-R-D-D	20	11	17.004	NA
A* search	D-D-DR-UR-U-R-R-D-D	18	18	0.0	Manhattan distance
Hill Climbing search	No solution				Manhattan distance

Case b

10

SRRRRRRRRR

RRRXXXXXXR

RRRRXXXRRR

RRRRRXXRXX

RXRRRRXRRR

RXXRRRRXXR

RRXXRRRRXR

RRRXXRRRXR

RXXXXXXXXR

RRRRRRRRRG

Algorithm	Solution path	Path cost	Number of explored states	Execution time (in ms)	Heuristic used
Breadth-first search	D-DL-D-D-D-D-D-D-R-R-R-R-R-R-R-R-R	34	64	0.0	NA

Uniform cost search	DR-DL-D-D-D-D-D-D-D-R-R-R-R-R-R-R-R-R	34	64	1.0	NA
Iterative deepening search	NA	NA	NA	Takes a very long time (>2 hours)	NA
A* search	DL-D-D-D-D-D-D-D-D-R-R-R-R-R-R-R-R-R	34	60	0.0	Manhattan distance
Hill Climbing search	DR-DR-L-UR-U-R-R-R-R-R-R-R-D-D-L-L-D-D-R-R-D-D-D-D-D	47	26	1.001	Manhattan distance

Case c

20

RRRRRRRRRSRXRXXXRXXR

RRRXXXXXXRRRRRRRXXRR

RRRRXXXRRRRRRRRRRRRR

RRRRRXXRXXRRRXRRRRXX

RXRRRRXRRRRRXRRRRRXX

RXXRRRRXXRRRRRRRRXX

RRXXRRRRXRRRRRRRRRR

RRRXXRRRXRRRRRRXRRR

RXXXXXXXXRRRRRRRRRR

RRRRRRRRRXRRRRXXRRR

RRRRRRRRRRRRRRRRRRR

RRRXXXXXXRRRRRRXRRR

RRRRXRXRRRRRRRRRXXRR

RRRRRXXRXXRRRRRRRRR

RXRRRRXRRRRRRRRRRRR

RXXRRXRXXRRRRRRRRXRR

RRXXRRRRXRRRRRXXXRRG

RRRXXRRRXRRRRRRXRXRR

RXRRRXXXXRRRRRRXRRRR

RRRRRRRRRXRRRRXRRRRR

Algorithm	Solution path	Path cost	Number of explored states	Execution time (in ms)	Heuristic used
Breadth-first search	DR-D-D-D-DR-DR-D-D-D-DR-D-DR-R-R-DR-DR-D	28	234	1.0	NA
Uniform cost search	DR-DR-DL-DL-DR-DL-DR-DR-DR-DR-DR-DR-DR-UR-DR-D-DR	20	223	1.0	NA
Iterative deepening search	NA	NA	NA	Takes a very long time (>2 hours)	NA
A* search	DR-UR-DR-UR-DR-DR-DR-DL-DR-DR-DR-DL-DR-DL-DR-DL-DR-DL-DR-DL-DR	20	25	0.999	Manhattan distance
Hill Climbing search	DR-DR-DR-L-UR-U-DR-UR-L-DR-DR-DR-R-D-D-U-D-U-DL-L-U-R-UL-UR-UR-D-UL-L-DL-D-D-D-DR-D-R-R-DR-DR-D-D-D-D-D-D	70	42	1.001	Manhattan distance

Changing the heuristic of the unformed search methods

For this section of the experiment, we will be using the following heuristics:

- Manhattan distance
- Euclidean distance
- Chebyshev distance

Since the variable here is the heuristic, we'll use case c as control.

For A* search,

Heuristic	Solution Path	Path cost	Number of explored states	Execution time (in ms)
Manhattan distance	DR-UR-DR-UR-DR-DR-DR-DL-DR-DR-DR-DL-DR-DL-DR-DL-DR-DL-DR	20	25	0.999
Euclidean distance	DR-DL-DR-DL-DR-DR-DR-DL-DR-DR-D-DR-DR-UR-DR-DR-D	20	51	0.0
Chebyshev distance	DR-DR-DL-DL-DR-DL-DR-DR-DR-DR-DR-DR-DR-UR-DR-D-DR	20	116	1.0

For Hill Climbing search,

Heuristic	Solution Path	Path cost	Number of explored states	Execution time (in ms)
Manhattan distance	DR-DR-DR-L-UR-U-DR-UR-L-DR-DR-DR-R-D-D-U-D-U-DL-L-U-R-UL-UR-UR-D-UL-L-DL-D-D-D-DR-D-R-R-DR-DR-D-D-D-D-D-D	70	42	1.001
Euclidean distance	DR-DR-DR-L-DL-L-D-DR-DR-DR-DR-DL-DR-DR-DR-DR-UR-DR-DR-D	24	21	0.00
Chebyshev distance	DR-L-DR-DR-DL-DR-DR-DR-DR-L-DL-L-R-UL-UL-UL-D-DR-R-UR-D-D-DR-DL-DR-UR-DL-R-DR-DR-R-UL-L-U-L-UL-UR-R-R-DR-L-UR-U-DR-DR-D-DL-DR-DL-D-UR-D	72	47	1.001

Using a test case where goal cannot be reached.

Case d

10

SRRRRRRRRR

RRRXXXXXXR

RRRRXXXRRR

RRRRRXXRXX

RXRRRRXRRR

RXXRRRRXXR

RRXXRRRRXR

RRRXXRRRXR

RXXXXXXXXX

XXXXXGXXXX

Algorithm	Solution path	Path cost	Number of explored states	Execution time (in ms)	Heuristic used
Breadth-first search	No path found	NA	NA		NA
Uniform cost search	No path found	NA	NA		NA
Iterative deepening search	No path found	NA	NA		NA
A* search	No path found	NA	NA		Manhattan distance

Hill Climbing search	No path found	NA	NA		Manhattan distance

5) Result Analysis

From the first experiment, the one thing that's been consistent is the poor performance of the IDS. IDS is supposed to be the best of both BFS and UCS- so why does that happen? This is due to the structure of the search space. What those results in is the search tree being quite deep and unbalanced, which means that IDS will keep increasing the depth limit and iterating over the same nodes multiple times before it reaches the goal. This essentially means that the algorithm is exploring a large part of the search space, but most of it is not part of the solution path.

On the other hand, BFS and UCS can efficiently find the shortest path in this case because they explore the search space layer by layer, without having to revisit previously explored nodes. This was unexpected as I had expected IDS to perform better. BFS and UCS both performed well; however, the memory cost was very high for them for the larger boards.

The performance of uniformed search was way better overall, especially the A* search (hill climbing got stuck in local minimum- however it worked after adjustment was made to the parameters).

Hill climbing had a higher path cost when compared to the other search algorithms- this is because they can get stuck in local optimum with quite ease; local optimum is a state where all neighboring states have a worse heuristic value, but it is not the global optimum (i.e., the actual goal state). There is no way of escaping local optima, which can lead to not optimal solutions.

We played around with the heuristics as well and observed that Manhattan is the best one for A* search whereas Euclidean is the best one for Hill Climbing. Finally, we tested our algorithms with an impossible to solve input- as expected all the outputs presented with no possible solution. This verifies that all algorithms are working properly.

6) Conclusion

Considering the results from all the experiments it is safe to say that A* is the best search algorithm for most cases. It combines the power of BFS with the efficiency of heuristic functions. In terms of Big O notation, the time complexity of A* is O(b^d), where b is the branching factor of the search tree (i.e., the average number of children per node) and d is the depth of the optimal solution. However, this is the worst-case time complexity, which assumes that the heuristic function provides no useful information.

When using an admissible heuristic function (one that never overestimates the cost to reach the goal), the A* algorithm is optimally efficient among all search algorithms that expand nodes in best-first order. In other words, no other algorithm that uses the same heuristic information will expand to fewer nodes than A*. This efficiency is achieved because A* expands nodes based on the sum of the path cost from the start state to the current state (g(n)) and the estimated cost to reach the goal state from the current state (h(n)). This combined cost (f(n) = g(n) + h(n)) helps the algorithm to focus on the most promising paths, reducing the number of nodes that need to be explored.

Now if we take Breadth-First Search (BFS) into consideration, it has a time complexity of O(b^d), and Uniform Cost Search (UCS) has a time complexity of O(b^C*/ε), where C* is the cost of the optimal solution and ε is the minimum cost of any action. Both algorithms can be less efficient than A* when an appropriate heuristic is used, as they do not incorporate heuristic information to guide the search. In fact heuristic can even be tailored for specific problems which makes A* even more attractive

In conclusion, the A* algorithm can be more efficient than other search algorithms because it combines both the actual cost of reaching a state and an estimated cost to reach the goal state. By using a suitable heuristic, A* can significantly reduce the number of nodes that need to be explored, leading to better performance in terms of time complexity.