Meltem Ceylantekin 28089

Ayşenur Güller 27796

1. **Modeling the Color-Maze puzzle as a search problem**

States: The state is represented by a Node structure. The Node has:

- Game board containing elements 1, 0, X, and S.

- 1: Colored Cells. 0: Uncolored Cells. X: Walls. S: Starting point and the current position of the agent.

- f(n), heuristic h(n), move cost g(n) values

- Parent State and parent state movement to react to that state.

Successor State Function: For each possible movement directions (up, down, right, left), the corresponding changes in the game board constitute the successors game board. Each movement continues until a wall is encountered, and '0' cells transform into '1' cells as the agent traverses the path in that direction. The successor state function creates the successors. Node by calculating their h, f, g values and parents using step cost and heuristic function.

Initial State: The initial state is characterized by a board consisting of uncolored cells ‘0’ and walls ‘X’, with a single 'S' indicating the agent's starting position. No colored '1' cells are present in the initial state.

Goal Test: The goal test verifies whether the board contains only colored cells and walls, with no remaining '0' cells in the maze.

Step Cost Function: The step cost function is defined as the Manhattan distance between the agent's current position and the potential next position. The agent moves in the selected direction until encountering a wall or frame, with the cost calculated based on the distance traveled.

2. **Extending search model for Color-Maze to apply A* search**

    a. **Inadmissible heuristic function h_1** Inadmissible heuristic functions overestimate the true cost to the goal from any state, and optimality is not guaranteed. $h(n) > h^*(n)$

    For this problem, we’ve described the inadmissible heuristic function as the number of cells in the entire board (len(board)*len(board[0])). This provides the maximum number of cells to color because it does not account for walls (X) and agent (S) which are defined as obstacles and cannot be colored. For instance, assume there is only one cell to paint near the agent on the board, other cells are walls. The true cost is 1. However inadmissible function is the number of cells on the board overestimates the true cost.

b. Admissible heuristic function h_2 Admissible heuristic functions never overestimate the true cost, and an optimal solution is guaranteed. h(n) <= h*(n)

For this problem, we've described the admissible heuristic function as the number of uncolored cells (num_uncolored). This heuristic never overestimates the true cost of reaching the goal from any state, because the number of uncolored cells represents the minimum job left. Moreover, the heuristic function always returns a positive step cost for all intermediate states. Since every uncolored cell needs to be visited (minimum coloring cost), and our heuristic captures the exact number of uncolored cells, it represents a lower bound on the actual cost of coloring the maze. The movement cost between cells adds to this base cost but cannot decrease it. Therefore, our heuristic never overestimates the final cost, making it admissible.

c. Monotonicity of h_2 The admissible heuristic function h_2 is monotone. Because it satisfies the conditions which are being admissible and ensures the following rule:

For every node n, and its successor n' ; h(n) <= c(n,n') + h(n')

where c(n,n') is the cost from n to n'.
This ensures consistency in the path to goal. Our heuristic counts the number of uncolored cells in the maze. Let's assume all the cells between n and n' are uncolored and reaching from n to n' is required only 1 movement (they are in the same row or column). In this case the heuristic will decrease maximum step cost. So above inequality holds. Consequences of monotonicity implies that f-values along the path increase monotonically or stay the same.

For instance, assume the path between nodes A and B is all colored. Since while moving on this path, for every move we will paint at least one cell, every move provide us to move forward to B. h_2 represents the uncolored cell number for the path to B. The decrease in heuristic function will be the Manhattan distance between A and B. Since Manhattan distance is c(n,n') function, the decrease in heuristic function will be equal to total step cost on this path. For this case, for every move f(n) = g(n) + h(n) ,total cost, either will increase or stay constant. Because g(n) (step cost) will increase but h(n) decrease at most as the step cost. This proves that h_2 is monotone.

```
def A_Star(board,admissibility):
    expanded_node_count=0
    total_distance=0
    closed_list = ClosedList()
    frontier = PriorityQueue()
    initial_node=Node(board,0+heuristic(board,admissibility),heuristic(board,admissibility),0,None,None)
    frontier.push(initial_node)

    while not frontier.is_empty():
        expanded_node=frontier.pop()
        expanded_node_count+=1
        """print("Expanded Node:")
        print_state(expanded_node.state)
        print(expanded_node.f_function)"""
        if(not expanded_node.g_function==None):
            total_distance+=expanded_node.g_function
        if(is_goal(expanded_node.state)):
            #return solution
            total_travel=0
            if(not expanded_node.g_function==None):
                total_travel=expanded_node.g_function

            print("Solution founded")
            print_path_info_reversed(expanded_node)  # Assuming print_path_info_reversed returns a string
            return total_distance,expanded_node_count,total_travel
            break
        else:
            for s in SUCC_H(expanded_node,admissibility):
                if not (closed_list.contains_state(s.state) or frontier.contains_state(s.state)):
                    frontier.push(s)
                elif(frontier.contains_state(s.state)):
                    exists_node=frontier.get_existing_node(s.state)
                    if(exists_node.f_function > s.f_function):
                        frontier.pop(exists_node.state)
                        frontier.push(s)
                elif(closed_list.contains_state(s.state)):
                    exists_node= closed_list.get_existing_node(s.state)
                    if(exists_node.f_function > s.f_function):
                        closed_list.remove_node(exists_node.state)
                        frontier.push(s)

            closed_list.add_node(expanded_node)
        while(frontier.is_empty()):
            print("No solution founded!")
            return total_distance,expanded_node_count,0
```

3.Implementation of A* search algorithm in Phyton for the Color-Maze puzzle

We implement A* algorithm which is admissible and monotone. We use Closed List to store already expanded nodes and won't be revisited. In this way, we ensure that A* never expand same state with higher f value. A* search efficiently finds the shortest path through a map-like graph. It prioritizes exploring paths that seem most promising based on a combination of the distance traveled so far and an

estimate of the distance remaining. It keeps track of explored paths and discards less promising ones, ultimately finding the optimal route.To achieve this, we add 3 if conditions to our algorithm:

if there is no n" in CLOSED or in FRONTIER such that n".state = s then

- This condition checks if a node with the same state as the successor node s already exists in either CLOSED or FRONTIER. If not, it means this successor node hasn't been explored yet, so we can add it to FRONTIER for future exploration.

else if there is some n" in FRONTIER such that n".state = s and f(n") > f(n') then

- This condition applies if a node with the same state as the successor node s already exists in FRONTIER, but the new node n' has a lower f value compared to the existing one (n"). This indicates a more promising path through the successor node s.
- modify n" with f(n") = f(n'), g(n") = g(n'), n".state = s, n".parent = n: The existing node n" in FRONTIER is updated with the new node's (n') lower f and g values. The parent of n" is also changed to n to reflect the more efficient path.

else if there is some n" in CLOSED such that n".state = s and f(n") > f(n') then

- This condition occurs if a node with the same state as the successor node s exists in CLOSED (meaning it was explored earlier), but the new node n' has a lower f value. This suggests that we might have reached this state from a better path through the current node n. Actually our A* is monotone, so we do not need this check.
- **re-open n" in FRONTIER with f(n") = f(n'), g(n") = g(n'), n".state = s

4. Performance Analysis

a. Difficulty levels (easy, normal, difficult)

There are 15 Color-Maze puzzle instances in our code with three difficulty-level

(easy, normal, difficult).

Easy: There are limited number of successors for this level which often leads straightly to the goal. This level is designed with a clear and singular solution path. It requires minimum effort for the decision-making of the agent.

Normal: Complexity is increased for this level by adding multiple successors for some states with the choices of directions. This level can have more than one solution. This multiple path brings up the necessity of choosing the minimum costed path.

Difficult: This level characterized by an increased number of successors and potential paths following multiple solutions. This is the most complex level because it requires extensive decision-making to reach the goal and even backtracking when needed.

b. Results of experiments

	easy 1		easy2		easy3		easy4	NoSol	easy5	
	Inadmissible	Admissible	Inadmissible	Admissible	Inadmissible	Admissible	Inadmissible	Admissible	Inadmissible	Admissible
Total Travel Distance	9	9	27	27	67	47	62	62	42	32
Number Of Expanded Nodes	2	2	3	3	7	5	6	6	8	7
CPU Time (seconds)	0,001688000	0,001856000	0,00291300	0,002758	0,006995	0,004929	0,005315	0,005419	0,007653	0,006359
Memory Usage (bytes)	42057728	41713664	41975808	41811968	42041344	42336256	42156032	41795584	42450944	41861120

	normal1		normal2		normal3	NoSol	normal4		normal5	
	Inadmissible	Admissible	Inadmissible	Admissible	Inadmissible	Admissible	Inadmissible	Admissible	Inadmissible	Admissible
Total Travel Distance	164	74	93	71	43	43	36	648	1012	984
Number Of Expanded Nodes	11	6	10	9	6	6	863	44	58	57
CPU Time (seconds)	0,011426000	0,006256000	0,01024400	0,008664	0,005481	0,005387	0,048847	0,041047	0,056608	0,056623
Memory Usage (bytes)	41877504	41664512	42156032	41369600	42123264	42074112	42205184	42106880	42401792	41893888

	difficult1		difficult2		difficult3		difficult4		difficult5	
	Inadmissible	Admissible	Inadmissible	Admissible	Inadmissible	Admissible	Inadmissible	Admissible	Inadmissible	Admissible
Total Travel Distance	3729	385	9608	1541	23792	1404	85936	24520	609080	195515
Number Of Expanded Nodes	156	24	354	83	794	76	2130	754	10449	3999
CPU Time (seconds)	0,186642000	0,025759000	0,57557400	0,088715	2,143033	0,083497	12,304794	2,035087	277,139253	43,050875
Memory Usage (bytes)	42532864	42156032	42926080	42057728	44285952	42582016	48267264	44793856	56393728	55328768

c. Discussion of the results presented in the table

Admissible Heuristic Function: An admissible heuristic function guarantees that it will never overestimate the cost of reaching the goal state. This ensures that A* search explores the most promising paths first and converges to the optimal solution.

Inadmissible Heuristic Function: An inadmissible heuristic function can overestimate the cost of reaching the goal state. This can lead A* search to explore less promising paths first and potentially get stuck in suboptimal solutions.

Observations from the data:

Traveled Distance: In almost all cases, the admissible heuristic function resulted in a shorter traveled distance compared to the inadmissible function. This indicates that the admissible heuristic guided A* search towards a more direct path to the goal state.

Expanded Nodes: The number of expanded nodes is typically lower for the admissible heuristic function. This suggests that A* search examined fewer dead ends and unnecessary paths when using the admissible heuristic.

CPU Time and Memory Usage: The data suggests that the admissible heuristic function can lead to slight improvements in CPU time and memory usage. This is likely because A* search terminates quicker when it has a more accurate estimation of the remaining cost to the goal.

Overall, using an admissible heuristic function leads to a more efficient A search for solving color maze problems. It reduces the number of explored paths, achieves a shorter solution path, and consumes fewer resources.*

The findings support the notion that admissible heuristic functions significantly enhance the efficiency of A* search for color maze problems. They guide the search towards promising paths, reduce exploration of dead ends, and ultimately lead to shorter solutions with lower resource consumption. However, designing effective admissible heuristics can be challenging, especially for complex problems. Striking a balance between accuracy and computational cost is crucial.