Ege Alpay

19551

1) Model the puzzle as a search problem: specify the states, successor state function, initial state, goal test, and step cost.

Initial State: Block has placed randomly on the board, in any orientation.

State: Block on the board in a random position which can be either in vertical or horizontal orientation.

Successor State Function: Block can be move left, right, up, down. So, we have 4 different actions. With the any action, there are 2 possibilities: Block can fall down from the board or block will be on the board either in vertical or horizontal orientation.

Goal Test: After applying action A to the block in state S, we check if the block is at point G in the matrix, in a vertical position. If yes, we have reached the goal. If not, we did not reach the goal.

Step Cost: Each action/move will cost 1.

2) Implement in Python a UCS algorithm to solve the puzzle.

```
def best_first_graph_search(problem, f, x_block_1, y_block_1, x_block_2, y_block_2):
    """Search the nodes with the lowest f scores first.
    You specify the function f(node) that you want to minimize; for example,
    if f is a heuristic estimate to the goal, then we have greedy best
    first search; if f is node.depth then we have breadth-first search.
    There is a subtlety: the line "f = memoize(f, 'f')" means that the f
    values will be cached on the nodes as they are computed. So after doing
    a best first search you can examine the f values of the path returned."""
    f = memoize(f, 'f')
    # Our initial node / state
    initial_node = Node(x_block_1, y_block_1, x_block_2, y_block_2)
    # Priority queue will give our the node which has the lowest f value
    frontier = PriorityQueue('min', f)
    frontier.append(initial_node)
    explored = set()
    # While there is a node in frontier:
    while frontier:
        # Take node from frontier
        node = frontier.pop()
        # Check if this node is the goal node
        if problem.goal_test(node):
            # return the goal node and memory consumption
            return node, resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
        # Add it to explored set
        explored.add((node.x1, node.y1, node.x2, node.y2))
        # Find the successors of the current node
        successors = find_successors(node)
        # For each successor of the current node
        for successor in successors:
            # If the successor has not visited and is not in frontier
            if (successor.x1, successor.y1, successor.x2, successor.y2) not in explored
and successor not in frontier:
                # Add it to frontier
                frontier.append(successor)
            # If it is in frontier
            elif successor in frontier:
                # Compare f values and update if needed
                incumbent = frontier[successor]
                if f(successor) < f(incumbent):
                    del frontier[incumbent]
                    frontier.append(successor)
    # Return None if we can't reach to goal node
    return None, resource.getrusage(resource.RUSAGE_SELF).ru_maxrss


def uniform_cost_search(problem, x_block_1, y_block_1, x_block_2, y_block_2):
    """[Figure 3.14]"""
    return best_first_graph_search(problem, lambda node: node.path_cost, x_block_1,
y_block_1, x_block_2, y_block_2)
```

You can find the solution in ucs.py file.

3) Extend your search model for A* search: Find a heuristic function and prove that it is admissible.

```
# This method represents h(n). Calculates the distance from current position to
goal
def find_distance_to_goal(node):
    x_avg = (node.x1 + node.x2) / 2
    y_avg = (node.y1 + node.y2) / 2

    hypotenuse = math.sqrt(math.pow((x_goal - x_avg), 2) + math.pow((y_goal -
y_avg), 2))

    return hypotenuse
```

Instead of calculating the differences between x and y coordinates of the current node of the block and the goal node, I choose to take the coordinates of the center of the block and calculated hypotenuse distance with the coordinates of the goal node.

We can think about **triangle equality**: the sum of the lengths of any two sides must be greater than or equal to the length of the remaining side. One of the two sides is differences of x, other one is y, and the remaining side is hypotenuse.

That why, this function won't overestimate the distance.

4) Implement in Python an A* search algorithm to solve the puzzle.

```
def best_first_graph_search(problem, f, x_block_1, y_block_1, x_block_2, y_block_2):
    """Search the nodes with the lowest f scores first.
    You specify the function f(node) that you want to minimize; for example,
    if f is a heuristic estimate to the goal, then we have greedy best
    first search; if f is node.depth then we have breadth-first search.
    There is a subtlety: the line "f = memoize(f, 'f')" means that the f
    values will be cached on the nodes as they are computed. So after doing
    a best first search you can examine the f values of the path returned."""
    f = memoize(f, 'f')
    # Our initial node / state
    initial_node = Node(x_block_1, y_block_1, x_block_2, y_block_2)
    # Find h(n) for initial node and update its value
    initial_node.distance_to_goal = find_distance_to_goal(initial_node)
    # Initialize our Min Heap and add initial node
    frontier = PriorityQueue('min', f)
    frontier.append(initial_node)
    explored = set()
    # While there is a node in frontier:
    while frontier:
        # Take node from frontier
        node = frontier.pop()
        # Check if this node is the goal node
        if problem.goal_test(node):
            # return the goal node and memory consumption
            return node, resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
        # Add it to explored set
        explored.add((node.x1, node.y1, node.x2, node.y2))
        # Find the successors of the current node
        successors = find_successors(node)
        # For each successor of the current node
        for successor in successors:
            # If the successor has not visited and is not in frontier
            if (successor.x1, successor.y1, successor.x2, successor.y2) not in explored
and successor not in frontier:
                # Add it to frontier
                frontier.append(successor)
            # If it is in frontier
            elif successor in frontier:
                # Compare f values and update if needed
                incumbent = frontier[successor]
                if f(successor) < f(incumbent):
                    del frontier[incumbent]
                    frontier.append(successor)
    # Return None if we can't reach to goal node
    return None, resource.getrusage(resource.RUSAGE_SELF).ru_maxrss


def astar_search(problem, x_block_1, y_block_1, x_block_2, y_block_2):
    """A* search is best-first graph search with f(n) = g(n)+h(n).
    You need to specify the h function when you call astar_search, or
    else in your Problem subclass."""
    # h(n) ise find_distance_to_goal() method in this HW
    # f(n) = g(n) + h(n) and we will compare f values in frontier
    return best_first_graph_search(problem, lambda node: node.path_cost +
find_distance_to_goal(node), x_block_1,
                                   y_block_1, x_block_2, y_block_2)
```

You can find the solution in a_star.py file.

5) Compare your UCS and A* codes on some sample puzzle instances. Construct a table that shows, for each instance, time and memory consumption. Discuss the results of these experiments: Are the results surprising or as expected? Please explain.

Matrix	Time	Memory
[["O", "O", "O", "X", "X", "X", "X", "X", "X", "X"], ["O", "O", "O", "O", "O", "O", "X", "X", "X", "X"], ["O", "O", "O", "S", "O", "O", "O", "O", "O", "X"], ["X", "O", "O", "S", "O", "O", "O", "O", "O", "O"], ["X", "X", "X", "X", "X", "O", "O", "G", "O", "O"], ["X", "X", "X", "X", "X", "X", "O", "O", "O", "X"],]	UCS: 0.00066s A*: 0.00035s	UCS: 8151040 bytes A*: 8372224 bytes
[["S", "O", "O"], ["O", "O", "O"], ["O", "O", "G"], ["X", "O", "O"], ["X", "X", "X"],]	UCS: 0.00031s A*: 0.00042s	UCS: 8081408 bytes A*: 8364032 bytes
[['O', 'O', 'O', 'X', 'O', 'X', 'X', 'X'], ['O', 'O', 'O', 'O', 'O', 'O', 'G', 'X'], ['X', 'X', 'O', 'X', 'O', 'O', 'O', 'O'], ['S', 'S', 'O', 'X', 'X', 'X', 'O', 'O']]	UCS: 0.00014s A*: 0.00020s	UCS: 8077312 bytes A*: 8351744 bytes
[['O', 'O', 'O', 'X', 'O', 'X', 'X', 'X'], ['O', 'O', 'O', 'O', 'O', 'O', 'G', 'X'], ['X', 'X', 'O', 'X', 'O', 'O', 'O', 'O'], ['S', 'S', 'O', 'X', 'X', 'X', 'O', 'O']]	UCS: 0.00016s A*: 0.00022s	UCS: 8081408 bytes A*: 8167424 bytes
[['O', 'O', 'O', 'O', 'X', 'X', 'X', 'X', 'X', 'X'], ['O', 'O', 'O', 'O', 'X', 'X', 'X', 'O', 'O', 'O'], ['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'G'], ['O', 'O', 'O', 'O', 'X', 'X', 'X', 'O', 'O', 'O'], ['O', 'O', 'O', 'O', 'X', 'X', 'X', 'X', 'X', 'X'], ['S', 'S', 'O', 'O', 'X', 'X', 'X', 'X', 'X', 'X']]	UCS: 0.00197s A*: 0.00127s	UCS: 8196096 bytes A*: 8204288 bytes

I guess, It won't be true to say anything certain about run time. Sometimes UCS works faster, sometimes vice versa. Heuristic function and the sample matrix have a big impact on run time. If we choose a good heuristic function, then A* will give a better performance in comparison to UCS.

As long as I observed, UCS usually consumes less memory than A*.