Andrew Taylor en605.645 08/29/24

Self-Check #1

1) Given the microworld that the robot must navigate, and that the robot is the only changing element in that microworld, the robot's position given by (x,y), within the dimensions of the microworld, constitute a state. Accounting for every position the robot can take legally on the board gives us the set of states, S. This can easily be stored as a list.

2) The set of transitions T defines the possible moves the robot can take (up, down, left, right), and the successors function uses T to determine all possible states that can be reached from a given state. We don't need to specify T explicitly if the movements are simple and consistent; instead, they can be embedded directly in the successors function. For the starting state S , the successors function would check each direction to see if it leads to a valid, unblocked position within the grid, and then return the states corresponding to those legal moves. This allows the robot to explore all potential paths toward the goal state G .

3) The graph is generated piecemeal as the algorithm explores, as it is checking a frontier for emptiness and adding states to the explored list one at a time. The "graph" of the successor function is basically a list of valid moves from each possible position in the microworld, which would look something like this:

Legal position -> successors

(0,0) -> (0,1), (1,0)

(0,1) -> (0,0), (0,2)

(0,2) -> (0,1), (1,2), (0,3)

(0,3) -> (0,2), (1,3)

(1,0) -> (0,0), (2,0)

(1,3) -> (0,3), (1,2)

(2,0) -> (1,0), (3,0)

(2,2) -> (2,1), (2,3)

(3,0) -> (2,0), (3,1)

(3,1) -> (3,0), (3,2)

(3,2) -> G (Goal)

4) DFS solution to the State Space Search Problenm

Problem Setup:

- Start Position (S): (0,0)
- Goal Position (G): (3,3)

DFS Algorithm Steps:

```
Place the initial state (0,0)) on the frontier.
Initialize the explored list.
While the frontier is not empty:
    Set current-state to the next state on the frontier.
    Return the path if current-state is the goal (3,3).
    Get the children by exploring the successors of current-state.
    For each child in children:
        o Add child to the frontier if it is not already in the explored list or on the
          frontier.
    Add current-state to the explored list.
Return nil if the goal is not found.
```

DFS Execution:

- Step 1: Start at (0,0).
  - Successors: (0,1), (1,0)
  - Choose (0,1 (DFS explores depth first).
- Step 2: Move to (0,1).
  - Successors: (0,0), (0,2)
  - (0,0) is already visited, so choose (0,2).
- Step 3: Move to (0,2).
  - Successors: (0,1), (0,3), (1,2) (but ((1,2) is blocked).

    - (0,1) is already visited, so choose (0,3).
- Step 4: Move to (0,3).
    - Successors: (0,2), (1,3)
    - (0,2) is already visited, so choose (1,3).
- Step 5: Move to (1,3).
    - Successors: (0,3), (2,3)
    - (0,3) is already visited, so choose (2,3).
- Step 6: Move to (2,3).
    - Successors: (1,3), (3,3)
    - (1,3) is already visited, so choose (3,3).
- Step 7: Move to (3,3).
    - Goal Reached!

Solution Path:

(0,0)→(0,1)→(0,2)→(0,3)→(1,3)→(2,3)→(3,3)

5) DFS is not an optimal algorithm. In this case, it doesn't find the optimal solution. The optimal solution is (0,0)→(1,0)→(2,0)→(2,1)→(2,2)→(2,3)→(3,3) which is one move less than DFS.
6) The "fold" in the DFS path, where it detours through (0,3) and (1,3) before reaching (2,2), is caused by the nature of DFS and the order in which successors are placed on the frontier. DFS explores as deeply as possible along one branch before backtracking, and in this case, it followed the path((0,3) first because that successor was placed on the frontier before the more direct path through (1,2). While changing the order of successor placement (e.g., not prioritizing downward moves) could reduce such detours, this approach is not foolproof and will not guarantee the shortest path for all goal placements. To consistently find the optimal path, an algorithm like Breadth-First Search or A* Search, which systematically explores all paths and guarantees the shortest one, is necessary.

7) To recover the path from the start state S to the goal state G during a search algorithm like DFS it is essential to maintain bookkeeping information that records the predecessor of each visited node. This involves storing for each node explored the node from which it was reached. By keeping track of these relationships in a data structure like a dictionary, you can reconstruct the complete path by backtracking from the goal node to the start node, following the chain of predecessors.

To Save for Path Recovery:

1. Parent Dictionary Mapping:

- Create a dictionary that maps each visited node to its predecessor.
- Example: {current_node: predecessor_node}.

2. Updating During Search:

- Whenever a node is expanded and its successors are generated, record each successor's predecessor as the current node.
- Ensure that each node is only assigned a predecessor once to avoid incorrect paths.

3. Path Reconstruction:

- Once the goal node G is reached, initialize a path list with G .
- Iteratively follow the predecessor links:
  - Set the current node to its predecessor.
  - Prepend the current node to the path list.
  - Continue until the start node S is reached.
- The resulting list represents the sequence of moves from S to G .

For Example:

- Start at S , set its predecessor to `None`.

- Visit successors, update their predecessors.
- Continue until G is found.
- Start from G , follow predecessors back to S .
- Reverse the collected nodes to get the path from S to G .

By recording the predecessor of each visited node during the search process, we equip the algorithm with the necessary information to reconstruct the path taken from the start to the goal. This additional bookkeeping is a necessary aspect of search algorithms when path retrieval is required and is required for uses that need knowledge of the specific steps taken to reach a solution.

Here's a walk through of the process of solving the problem using DFS and then a record of the path from the start S to the goal G. I will show the frontier, the explored list, the states of the path, and display thefinal path.

Initial Setup:

- Start Position (S): (0,0)
- Goal Position (G): (3,3)
- Grid:

```
  0 1 2 3
0 S o o o
1 o x x o
2 o o o o
3 o o x G
```

DFS Execution with Bookkeeping:

1. Initialization:
   - Frontier: [(0,0)]

- Explored: []
- Parent Dictionary: {(0,0): None}

2. Step 1:
   - Current Node: (0,0)
   - Successors: (0,1), (1,0)
   - Frontier: [(0,1)]
   - Explored: [(0,0)]
   - Parent Dictionary: {(0,0): None, (0,1): (0,0)}

3. Step 2:
   - Current Node: (0,1)
   - Successors: (0,0) (already explored), (0,2)
   - Frontier: [(1,0), (0,2)]
   - Explored: [(0,0), (0,1)]
   - Parent Dictionary: {(0,0): None, (0,1): (0,0), (0,2): (0,1)}

4. Step 3:
   - Current Node: (0,2)
   - Successors: (0,1) (already explored), (0,3)
   - Frontier: [(1,0), (0,3)]
   - Explored: [(0,0), (0,1), (0,2)]
   - Parent Dictionary: {(0,0): None, (0,1): (0,0), (0,2): (0,1), (0,3): (0,2)}

5. Step 4:
   - Current Node: (0,3)

- Successors: (0,2) (already explored), (1,3)
- Frontier: [(1,0), (1,3)]
- Explored: [(0,0), (0,1), (0,2), (0,3)]
- Parent Dictionary: {(0,0): None, (0,1): (0,0), (0,2): (0,1), (0,3): (0,2), (1,3): (0,3)}

6. Step 5:
   - Current Node: (1,3)
   - Successors: (0,3) (already explored), (2,3)
   - Frontier: [(1,0), (2,3)]
   - Explored: [(0,0), (0,1), (0,2), (0,3), (1,3)]
   - Parent Dictionary: {(0,0): None, (0,1): (0,0), (0,2): (0,1), (0,3): (0,2), (1,3): (0,3), (2,3): (1,3)}

7. Step 6:
   - Current Node: (2,3)
   - Successors: (1,3) (already explored), (3,3)
   - Frontier: [(1,0), (3,3)]
   - Explored: [(0,0), (0,1), (0,2), (0,3), (1,3), (2,3)]
   - Parent Dictionary: {(0,0): None, (0,1): (0,0), (0,2): (0,1), (0,3): (0,2), (1,3): (0,3), (2,3): (1,3), (3,3): (2,3)}

8. Step 7 (Goal Reached):
   - Current Node: (3,3)
   - Successors: Goal Reached
   - Frontier: [(1,0)]
   - Explored: [(0,0), (0,1), (0,2), (0,3), (1,3), (2,3), (3,3)]

  - Parent Dictionary: {(0,0): None, (0,1): (0,0), (0,2): (0,1), (0,3): (0,2), (1,3): (0,3), (2,3): (1,3), (3,3): (2,3)}

Path Reconstruction:

Starting from the goal, trace back through the parent dictionary:

(3,3) -> (2,3) -> (1,3) -> (0,3) -> (0,2) -> (0,1) -> (0,0)

Reverse the order to get the path from S to G:

Grid with Path Marked:

```
  0 1 2 3

0 * * * *

1       *

2       *

3       G
```

This shows the path the algorithm followed from S to G (with bookkeeping to reconstruct the path).