Jay Nash | Derin Gezgin | Russell Kosovsky
Fall 2024 | COM316: Artificial Intelligence | Problem #1: Simple Search

Intro:
One of the first problems tackled in the field of AI was basic search. Finding a path from a start state to a goal state.

Given:

1. A 20x20 2d array with random 0s and 100000s. This is to represent an area of terrain with 400 blocks of space. A 0 means the block is free; a 100000 means it is occupied by an obstacle. This array can also be used to mark blocks that have been visited (looked at during the search).
2. One of the 0s (free blocks) will be designated as the start position and one as the goal position (marked as -1). What we will be doing is finding if there is a path from the start to the finish. If so, what blocks are part of the path?
3. A function (block-status B) that will return -1, 0, or 100000 for block B. Block B will be designated by a list of 2 elements (x y) designating the x and y coordinates of the box with the top left block being (0 0); x increases from left to right, y increases from top to bottom.
4. A function (adjacent B) that returns a list of blocks adjacent (one step vertical or horizontal) to block B.
5. A function (stepo B C) that returns C if a step from B to C is a legal move (neither B or C are obstacles and C is adjacent to B). It returns #f if B->C is not a legal move.
6. A function (stepv B C) that returns C if a step from B to C is a new move (neither B or C are obstacles, C is not marked as visited, and C is adjacent to B). It returns #f if not.

Problem:

- Assuming you know where the start is, using only horizontal and vertical movements, find a path to the goal if one exists.
- Does your algorithm find the path quickly?
- How much storage space does it require?
- Does it find the shortest possible path?

We met on September 4th, 2024, from 4:00 to 6:00 p.m. to work on this assignment and develop a solution. We each had a slightly different approach to implement the solution, so we wanted to add and explain our versions.

Derin Gezgin

In this solution, our common idea was to keep track of the cost of each location/square in the grid always to have the shortest path possible. We decided that we could achieve this using a hash map. To solve the issue of having a hash map in Scheme, we implemented it in a 2D vector. In this case, the indices (0, 1, 2, 3…) are the hash values (the distances), and in every index, there's another vector to which we append the locations with the relevant cost. When we find the goal node, we'll backtrack from the final node to the first node by following the hash list backward. When we're going back, for each node in each depth level, we'll use the *stepo* function to check if moving from the current node to the new node is legal; if so, we'll move to the next depth level and approach our start position. In my implementation, I had two main and two helper functions. There's a main and a helper function for doing the search and saving the nodes into the relevant hash levels, and there's also a main and a helper function for backtracking from the goal node to the starting node. The logic is very similar, so I have separate main and helper searching functions that do DFS and BFS. Both will solve the problem and have pros and cons, which I'll summarize.

The thought process is simple:

- In both searches, we start by having an empty list, which we'll use as a hash map.

- In BFS implementation, we'll add the starting node to the hash map (as the 0th level) and expand our hash map by gradually increasing the depth. For each depth level, we first check if we're on the goal node, and if so, we start backtrack. Otherwise, we loop through the adjacent nodes and check if the move from the current node to the adjacent node is legal. If so, we mark the node as visited and add it to the relevant depth level in the hash list. We continue this process until we are in the goal node or no adjacent nodes from which to move.
- The DFS implementation is very similar, except we call the function recursively inside the function to check whether the move is valid. This will construct a path forward. But the difference is it'll go as deep as possible in the current path before changing to an alternative one. I'm not very sure about the pseudo-code implementation of DFS. I wanted to add it as an alternative.

After we find the goal node, both implementations call the backtrack function:

1. In the *backtrack* function, we'll start by accessing the goal node from the hash and adding it to a new list to construct the new path backward. We'll call the backtracker to handle the rest.
2. In *backtracker*, we first check whether the current_depth is -1. If so, we are in the start node. We will return to the path we found. This is the end of our solution.
3. If we're not, we first access and iterate the possible next nodes in the current depth level.
4. In the iteration, we check whether the move "*current node we're in → new node*" is legal.
5. If so, we append the next node to our path and call the backtracker function again by decreasing the depth level and updating the current path. If not, we will pass to the next iteration.

In general, the advantage of this solution is

- We'll append all the squares in the grid to the hash list maximum for once; in the worst case, we'll use space as big as the grid itself.
- As we will visit every node a maximum of once (while searching), if we assume we have an NxN grid, the time complexity would be $O(N^2)$ in the worst case. Or the number of nodes.
- We'll find the shortest path possible as we backtrack using the cost of the nodes.

As a side note, BFS would perform better than DFS in most cases, except DFS would be better when we have a memory constraint (or an extremely large grid) as it removes the explored paths from memory. DFS doesn't guarantee that we'll find the shortest path; on the other hand, BFS does.

My pseudo-code implementation for the search function that does BFS:

```
def search(grid, start):
   visited_nodes = [[start]] # Starting with creating the 2-D vector
   current_depth = 0
   goal = None
   while not goal: # While the endpoint it not found
       if not visited_nodes[current_depth]: break # If there's no nodes to explore, break
       goal = increase_depth(visited_nodes, current_depth) # Go one layer deeper
       if goal: return backtrack(visited_nodes, goal) # If it's found, return the path
       current_depth += 1
def increase_depth(visited_nodes, current_depth):
   if len(visited_nodes) <= current_depth+1:visited_nodes.append([]) # If it's a new layer
   for starting_node in visited_nodes[current_depth]: # For each node in the depth level
       if block-status(starting_node) == -1:return starting_node # Return node if it's goal
       for adjacent_node in adjacent(starting_node): # For each neighbor..
           if stepv(starting_node, adjacent_node): # Check if it's a new step
               adjacent_node.set(visited) # If it's set as visited and add it to hash
               visited_nodes[current_dept+1].append(adjacent_node)
```

My pseudo-code implementation for the search function that does DFS *(It's similar to BFS)**:***

```
def search(grid, start):
   visited_nodes = [[start]
   searcher(grid, start, 1, visited_nodes)

def searcher(grid, current_node, depth, visited_nodes):
   if len(visited_nodes) < depth + 1: visited_nodes.append([])
   adj_list = adjacent(current_node)
   if not adj_list: return None
   for adjacent_node in adj_list:
       if block-status(adjacent_node) == -1:
           visited_nodes[depth].append(adjacent_node)
           return backtrack(visited_nodes, depth)
       elif stepv(current_node, adjacent_node):
           adjacent_node.set(visited)
           visited_nodes[depth].append(adjacent_node)
           result = searcher(grid, # Calling the function recursively gives us the DFS
                             adjacent_node,
                             depth + 1,
                             visited_nodes)
           if result is not None: return result
```

My pseudo-code implementation for the backtracking from the goal node to start. This works both for the BFS and DFS implementations.

```
def backtrack(visited_nodes, final_depth):
   goal_node = visited_nodes[final_depth][0]
   new_path = [goal_node] # Creating the list to have the path with the goal node
   return backtracker(visited_nodes,
                      goal_node,
                      final_depth-1,
                      new_path)

def backtracker(visited_nodes,
                current_node,
                current_depth,
                current_path):
   if current_depth == -1:return current_path # If current_depth is -1 this means we're done
   possible_next_vals = visited_nodes[current_depth] # All possible next values
   for next in possible_next_vals:
       if stepo(current_node, next): # For each value, we check if it's a valid move
           current_path[current_depth] = next
           return backtracker(visited_nodes, # If so we call the function again.
                              next,
                              current_depth-1,
                              current_path)
```

Jay Nash

```
function search(start):
    nodeCosts = ((start))
    currentCost = 0
    do:
        if nodeCosts[currentCost] is empty:
            return "Failed to Find Path"
        else:
            nodeCosts.extend(1)
            nodeCosts.append(())
            for node in nodeCosts[currentCost]:
                if node is goal:
                    return find_path(nodeCosts, cost.index, node)
                else:
                    for neighbor in adjacent node:
                        nodeCosts[cost.index + 1].append(neighbor)
        currentCost = currentCost + 1

function find_path(nodeCosts, goalCost, goal):
    path = (goal)
    for i=goalCost-1, i > 0, i--:
        for node in nodeCosts[i]:
            if stepv path.peak node:
                path.push(node)
                break
    return path.push(nodeCosts[0][0])
```

Explanation:
The general idea is to expand our search breadth-wise, tracking each square we search inside of a vector of vectors, with the primary nodeCost vector tracking the total “move cost” (total number of moves to that square) of the nodes inside the subvector. Once we find the goal, we stop searching and begin building a path to the goal. In order to build the path, we start with the “move cost” of the goal and a list of nodes (our path) only containing the goal at first. Then, we search the nodeCost vector for the vector of nodes with a cost one less than the goal. Once we find a node with the correct cost that satisfies the stepv function, we add it to the front of the path list, and then search again for a node with one less cost than the node we just added. Eventually we will reach 0 cost, and then we know to add the goal node to the head of the path and return the path. This is guaranteed to find the shortest possible path to the goal, and has a total max storage use of O(n) where n is the total number of nodes on the grid. The time complexity in the worst case is O(n) as well but if the goal is not exactly on the other end of the grid it will take less time.

Example:
Lets say that we are searching this grid for the path to the goal, the top left square is (0,0)

Start	Obst.	Obst.	
	Obst.	Goal	

We initialize our vector of vectors as ((start)). Since we have not found the goal, we next expand the vector by 1, and add the neighbors of start to the next index:

(((0,1)), ((0,0),(0,2)))

We have still not found the goal, so we expand the vector by 1 and add the neighbors of the previously added nodes to the next index:

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

We continue the previous step 3 more times and our vector is:

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

We have now found the goal (2,2) and its cost is 5, which means the it is in the 6th vector in the vector of vectors.

Now we start the path finding, first we initialize the path as ((2,2)), because we know that is the final step. To find the rest of the path we search the 5th vector in the vector of vectors for a node that is a legal move from (2,2), which returns (2,3).

Note that there may be multiple legal moves from the goal to other squares in our nodeCost vector, we just take the first one as it does not make a difference.

Now that our path is ((2,3),(2,2)) we search the 4th vector in the vector of vectors for a node that is a legal move from (2,3), which returns (1,3). This makes our path ((1,3),(2,3),(2,2)).

We continue doing this for all the cost vectors, excluding the 0th vector, resulting in a path of ((0,2),(0,3),(1,3),(2,3),(2,2)). We then add the start node to the start of this list to complete our path:

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

We can see that our total path takes 6 steps (including the start node), this is the shortest possible path for this grid, the next shortest path takes 8 steps (including the start node).

Once we have our path, we can simply pop the first node, move the robot to it, and do that until we reach the goal. We can ignore the starting node in the list as the robot starts there, but it is included in order for our path to be complete.

Russell Kosovsky

1) initialization:
   - create queue that will store nodes to be explored, starting with the start node and its path
   - create 2D array to track visited nodes

2) BFS:
   - while the queue is not empty:
   - dequeue the first node and the path to it
   - if this node is the goal, return the path
   - otherwise, mark it as visited and enqueue all valid neighboring blocks (using stepo and stepv)
   - return: if no path is found… return message that no path exists

---

```
search(start, goal):
   queue =  empty queue
   visited =  2d array for visited nodes

   enqueue(queue, (start, [start]))  # enqueue start block with path [start]
   mark start as visited

   while queue not empty:
      currentNode, currentPath = dequeue(queue)

      if currentNode == goal:
         return currentPath  # return the path to the goal

      for each neighbor in getAdjacent(currentNode):
         if stepO(currentNode, neighbor) and stepV(neighbor) is not false:
            mark neighbor as visited
            newPath = currentPath.append(neighbor)  # append neighbor to path
            enqueue(queue, (neighbor, newPath)) #enqueue neighbor with new path

   return “no path”
```