B551 Assignment 0: Searching and Python

Sumanth Gopalkrishna (id- 2000753899)

Part 1: Navigation

In this problem we have a N\*M board with N rows and M columns and one agent (p) and a person (@) and the rest of the places on the board is either an open space (.) or a wall (X). The agent can move one square at a time in any direction given that it is an open space (.) .The objective is to reach the specified goal state with as few moves as possible. (shortest path). The goal state being the agent reaching the person.

States: A state description gives the location of the agent as “p”, the person as “@” and the rest of the squares are either an open space “.” or a wall “X”.

Initial state: Any state with 1 “p” and 1 “@” on different squares with “.” and “X” on the remaining squares on the N\*M board.

Successor function: This function generates the legal states after the agent moves in one of the four directions (left, right, down, up), but the agent can only move into a square which is an open space “.” .

Goal state: This state is when the agent “p” reaches the person “@” on the given “N” \* “M” board.

Path Cost: Each step the agent takes costs 1, so the path cost is the total number of steps/moves the agent takes in the path.

Strategy:

The strategy used here is the Breadth first search (BFS) in which the root node is first expanded followed by its successors, then their successors and so on.

In this case, the cost is uniform for each step, so this is determined to find the shortest path. After I start with an empty fringe, I keep adding the nodes or successor to it from one end and remove them from the other end using FIFO so, the nodes which are visited first are expanded first. It is optimal and the first goal node selected after expanding is the optimal solution.

**Why the original solution failed to work**?

The program fails to find a solution and enters an infinite loop after a while as there is no concept of visited nodes implemented and in that case if a node is already visited it still adds that node to the fringe even though it was previously added and hence it enters an infinite loop of successors being added to the fringe.

So, to fix this I have added a “visited\_node” list which contains all those nodes that have already been visited. After the successor function is called for a particular state, I am checking whether the new successor is already part of the visited node list. If so, I am ignoring it.

Assumptions as given:

Only 1 “p” and 1 “@”

Solution (working of the code):

After the initial board is passed to the search function a “visited\_nodes” list and a “fringe” is also created. The fringe contains all the successor states possible from a given board configuration.

The first state from the fringe is removed and sent to the “moves” function which returns all the possible successor states from that position.

A while loop runs till the fringe is empty. For each of the successors which is returned from the moves function we check whether it is the goal state, if not we find the direction in which the agent has moved from its previous position by calling “direction” function and then append that direction to the “move\_string” variable which contains directions the agent has taken to reach the previous point.

We then add 1 to the path length taken to reach this position. We add all this to the fringe for the move following which we also add this node to the “visited\_nodes” list.

We repeat this process until the agent reaches the goal state in which case we return the path length and the directions the agent took.

In case the agent is unable to reach the goal state “-1” is returned.

**Problems faced**:-

Even after I removed the first element in the queue and figured out the direction in which the agent was moving, I was getting the path length right but not the directions to the goal state.

I took a while to figure out that sometimes the agent goes in the wrong path to meet a dead end and comes back, and these directions should not be appended.

So, I created a new direction variable called “new\_move\_string”\_which adds only that one direction the agent moved to the already existing directions and I kept clearing the “new\_move\_string” before every loop so unnecessary moves of the agent are not added.

**Innvoative** -

I feel the “direction” function is quite innovative as it finds the direction in which the bot has moved based on the rows and columns.

**Other methods** tried:

I tried another method involving bfs without the visited nodes array so less memory is utilized but the solution did not work well.

Part 2: Hide and Seek

In this problem we have a

Input :- N\*M board with N rows and M columns and one agent (p) and a person (@) and the rest of the places on the board is either an open space (.) or a wall (X).

And input- “k” number agents which needs to be placed on the board.

The objective is to reach the specified goal which is placing all the “k” number of “p” agents on the board without any conflicts. The conflicts being, the agents cannot see each other, meaning they cannot be on the same row or column without any “X” or “@” between them.

States: Arrangement of “K” agents on the board with none of them seeing each other meaning they cannot be on the same row or column without any “X” or “@” between them.

Initial state: Any state with 1 “p” and 1 “@” on different squares with “.” and “X” on the remaining squares on the N\*M board.

Successor function: This function generates a board with one more “p” included on the board with no conflicts with the other ‘p’s on the board.

Goal state: ‘k’ number of ‘p’agents placed on the N\*M board without any conflicts.

Path Cost: Not applicable because only the final state counts.

Strategy:

The strategy used here is the Breadth first search (BFS) in which the root node is first expanded followed by its successors, then their successors and so on.

After I start with an empty fringe, I keep adding the successor board configurations to it from one end and remove them from the other end using FIFO so nodes which are visited first are expanded first. In this case no costs are associated with the fringe.

**Why the original solution failed to work**?

The ‘P’ was added to the board irrespective of whether its position conflicts with the other p’s position on the board. It was just adding a ‘p’ if a ‘.’ was found.

So, I implemented a fix where I check if there are any p’s in sight for the particular position (r,c).

In sight meaning they cannot be on the same row or column without any “X” or “@” between them.

Solution:-

The ‘initial board’ with ‘K’ being the number of agents is passed to the ‘Solve’ function.

In the solve function we add the initial board to the fringe and remove it in the FIFO manner. We perform this loop until the fringe is empty.

We call the successor function on the board. We loop through all the positions with ‘.’ on the board from top left to right bottom of the board until we find a position on the board without any conflicts.

For figuring out the conflicts I have used four while loops which checks whether there are any conflicts with any other ‘p’ on the board from left to right of the row and top to bottom of the column.

If any ‘X’ or ‘@’ is present between them, that state is not a conflict.

After figuring out which position to put the ‘p’ in, I call the ‘add\_pichus’ function which adds a ‘p’ in the specific (r,c) location. Then return this board to the ‘solve’ function and call the ‘count\_pichus’ function to check if we have reached the goal state with the number of pichus on the board equal to ‘k’.

If not, we repeat this process by adding this board to the fringe. If no position is found in the entire board to put an extra ‘p’ then “None” is returned.

**Problem faced**:

While checking if the rows and columns had an “X” or a “@” between the ‘p’s, I kept ignoring the case where the ‘X’ can be in this state for example p..pX in which case it is not admissible but only if it is in this state p…Xp it is admissible. So, later I figured it out and implemented the fix.