CMPT310: Assignment 2 Analysis
Arda Cifci - 301386128

Proposed turning solution

In the context of the search problems we are solving, my proposed fix to the agent short-coming of not being able to turn is to modify / create several functions such as ‘execute_action’ to calculate and display the agent visually turning and functions ‘path_cost’, ‘h’ to include turning as part of cost. I don’t believe I would have to modify any of the search algorithms themselves (besides cost) as exploring the grid and finding a path solution should not need any new information from turning (besides cost) to decide a path. Based off of assignment 1’s turning implementation, I would implement turning in assignment 2 similarly but instead of turning when the agent finds nearby dirt from precepts I would instead start turning the agent after finding the solution path based on the agent's current location and direction it’s facing.

By implementing turning, the functions ‘Path_cost’ and ‘h’ would have to be modified to accommodate turning in its final path cost result. For the ‘path_cost’ function, since we are only looking at one action (the neighboring room), we can add 0, 0.5, or 1 to the final result cost depending on which direction the agent is currently facing. For example if the agent is facing up and the next action is to move up we don’t have to turn and therefore don't need to include any cost to turn. But if we are facing up but need to go down then we would have to turn twice which would increase the cost by 1. The function ‘h’ would require more work as we are looking at a bigger distance with multiple turning options at each potential path. The only way I could think of adding turning cost to ‘h’ is to perform a dry (purely calculation, no GUI) run from the start node location to the dirt node in question and count the total number of turns it would cost if the agent would choose this path and add it in addition to the euclidean distance number. I believe this would be the most accurate way to measure the correct turning cost and ultimately the correct final cost value which is important in a heuristic. One thing to note is that having each turn cost 0.5 or 1 might be too much and instead having a cost of 0.25 per turn or something similar might be better. We would have to test the heuristic and see.

Effects of turning

Because my proposed solution doesn’t change the way the search algorithms function, the branching factor and number of nodes being added to the frontier at each step should not change very much after implementing turning. Perhaps the search algorithms may take a slightly better or worse path after implementing turning which

may result in at most a few more or less nodes being added to the frontier. I believe from my implementation the chosen path at each search will be noticeably different because of the cost associated with turning. Particularly, I hypothesize that the search algorithm (UCS and A*) will try to avoid all walls as best as it can because bumping into walls will drive the cost up a lot due to the amount of turning it would need to clear them. Additionally, search algorithms like A*, which currently zig-zags a lot to reach dirt that's diagonal to the agent, would prefer to go in straighter lines instead of zig-zags as this would drive costs up a lot as well.

PSEUDO CODE VISUAL PART

Note:

- The method to turn the agent would be similar in style to assignment 1.
- Need to change the function 'execute_action' to get visual turning working.
- There is a field called agent.direction for agents to store their direction and previous_action field saved to env.

```
def execute_action(self, agent, action):
    Let xi and yi = agent location

    If agent is on dirt and action is to suck then:
        dirtCount -= 1
        Agent performance += 100
    Else if agent action is Forward then:
        Set current agent button to white/colored (for solution path)
        Move the agent "forward" depending on previous action
            (if previous action is UP then y=y+1, if LEFT then x = x-1, etc)

        Set agents new button to newly calculated location
        Save agents new location into agent location env variable
        Save agents current direction depending on previous_action
    Else if agent action is UP then:
        Get agents current direction
        Set previous_action to = UP
        Agent performance -= 1
        Turn the agent to face up if agent is not already facing up
            (this is done with agent.direction += Direction.R / Direction.L like
            assignment1)
```

(depending on agents current direction, may have to turn multiple times here)
Force/set next action to be forward then resume regular actions

Else if agent action is DOWN then:
Get agents current direction
Set previous_action to = DOWN
Agent performance -= 1
Turn the agent to face down if agent is not already facing down
(this is done with agent.direction += Direction.R / Direction.L like assignment1)
(depending on agents current direction, may have to turn multiple times here)
Force/set next action to be forward then resume regular actions

Else if agent action is RIGHT then:
Get agents current direction
Set previous_action to = RIGHT
Agent performance -= 1
Turn the agent to face right if agent is not already facing right
(this is done with agent.direction += Direction.R / Direction.L like assignment1)
(depending on agents current direction, may have to turn multiple times here)
Force/set next action to be forward then resume regular actions

Else if agent action is LEFT then:
Get agents current direction
Set previous_action to = LEFT
Agent performance -= 1
Turn the agent to face left if agent is not already facing left
(this is done with agent.direction += Direction.R / Direction.L like assignment1)
(depending on agents current direction, may have to turn multiple times here)
Force/set next action to be forward then resume regular actions

Update number of steps and performance cost label numbers

PSEUDO CODE SEARCH (PATH_COST and H) PART

Note: my implementation does not change the search algorithms, just need to change the functions ‘path_cost’ and ‘h’

Def path_cost(self, c, state1, state2):
    Set x1 and y1 to state1
    Set x2 and y2 to state2
    Set turn_cost to 0

    Calculate number of turns it would take to go from state1 to state2
        (if agent is facing left and we need to go up then turn_cost would be 0.5 (or any other float to optimize the program))
        (because states are neighbors in path_cost, cost would between 0 and 1 (0 = dont turn, 0.5 = 1 turn, 1 = 2 turns))

    Get height z for both states
        (z1 = np.sqrt(x1*x1 + y1*y1, z2 = np.sqrt(x2*x2 + y2*y2)

    Calculate euclidean distance = (x1-x2)**2 + (y1-y2)**2 + (z1-z2)**2

    Return c + turn_cost + e_distance

Def h (self, node):
    Get all dirt locations inside a list
        (this can be a function or you can manually read the map and get all locations in a nested for loop)

    Set variable min_d value to a high value like float/int max

    For (x, y) in dirt locations list:
        x1, y1 = node.state
        x2, y2 = x, y

        Get height z for both states
            (z1 = np.sqrt(x1*x1 + y1*y1, z2 = np.sqrt(x2*x2 + y2*y2)

Get number of turns required from node location to dirt location

(Do a search to this dirt location using BFS/UCS and store the solution path, then similar to my display_solution function, instead of coloring the solution path check how often we would have to turn and return that instead)

(We can make a simple checker by calculating the agent's x,y movement through the path. If the agent is going right then x = x + 1 and the agent should be turning to the right. Every time the agent goes in a different direction (y=y+1), increment the number of turns return variable.)

(we can store the previous movement direction and check if the new direction is a different direction, if not then don't increment.)

(because node doesn't have a initial direction, we can start with no direction and give the return value a default 0.5)

Calculate euclidean distance = (x1-x2)**2 + (y1-y2)**2 + (z1-z2)**2
Check if this distance + turn_cost*0.5 is smaller than the previous dirt locations cost

If yes then store this distance +turn_cost as new min.

Return min

Blocked Dirt Rooms

In our search algorithms we can determine there are dirty room(s) unreachable if the search algorithm returns None as a final result. The search algorithm would only return none if it can't find a solution path. However, if all dirty rooms were cleaned and there were no stuck dirt rooms then the search algorithm would not run again in the first place as another part of the program would stop searching when dirtCount reaches 0. Therefore we can conclude that if the search algorithm returns None then there must be dirty room(s) stuck behind walls somewhere uncleaned.