CSC263 Winter 2021 Problem Set 3

Morgan Chang (1005127113) and Kapilkumar Ramchandani (1003877879)

Question 1: Alice Mazes

(a) To find a shortest solution of an Alice maze, we will apply BFS algorithm to explore the shortest path from the start location to the goal location. To do so, we will represent any arbitrary Alice maze as a graph with the following data structures:

- Use a directed graph to model an Alice maze, where each vertex represents a cell location in the maze and each edge represents a move between two cell locations.
- Store the vertices in a $width * width$ matrix, where *width* is the number of columns in the maze. Refer to each vertex by its coordinate in the matrix. Each coordinate in the matrix corresponds to its position in the maze.
- Each vertex v (coordinate location in the matrix) stores a dictionary of values/information that need to be used or maintained through BFS.
  - `v[color]`: color of the arrow(s) in the corresponding cell in the maze.
  - `v[directions]`: direction(s) of the arrow(s) in the corresponding cell in the maze.
  - `v[path_length]`: the shortest path length from the start location to v.
  - `v[parents]`: parent(s) of v in BFS tree. Since it is possible to step on the same cell for more than one time to get to the goal, we will keep track of each parent of v at a specific step from the start location to v during BFS by storing it in a dictionary of `{step: parent}`.
  - `v[step_size]`: step size of v. Since it is possible to step on the same cell for more than one time to get to the goal, we will keep track of each step size of v at a specific step from the start location to v during BFS by storing it in a dictionary of `{step: parent}`.
- During BFS, as we get to a vertex v from the start location, at a specific *step*, we can get the edges of v with each of it's directions, stored in `v[directions]`, times it's step size at *step*, stored in `v[step_size][step]`.

Given the graph representation of an Alice maze *maze* and its start location s, solve the shortest solution by applying BFS as follows:

1. Initialize `step = 0`. Set path length of s to 0 (`s[path_length] = 0`) and step size of s at *step* 0 to 1 (`s[step_size][0] = 1`).
2. Since it is possible to step on the same cell more than once, that is, there can be several paths to the same vertex at different steps (but which path can lead to the goal is determined by its edges), we need to apply BFS on each cell from a specific *step*. We can achieve this by storing queues in different layers of *step* and only start exploring the vertices in the next layer when we have finished exploring all vertices in the current layer.

   In this case, we need to keep track of at which *step* we enqueue each vertex. We will use a dictionary of queues as `{step:queue}`, where `step` is the *step* we discover and enqueue a vertex and, `queue` is a queue that stores the vertices enqueued at that step.

   Initialize an empty dictionary of queues `Q={}` and enqueue s in the queue at *step* 0 (`Q[0]`).
3. Enter a loop to explore all vertices in the queue at `Q[step]`. For each dequeued vertex v:
   - Check `v[color]`: update step size of v at *step* (`v[step_size][step]`) if its color is red or yellow; stop running the algorithm if its color is *goal*. Skip to the next loop iteration if the updated step size is less than 1.
   - Get the edges of v by multiplying each direction in `v[directions]` with its current step size. For each edge (v, u), if u is out of range of *maze* or is already discovered by v, i.e. `u[parents]` contains v (here, to prevent from generating a cycle in the BFS tree), skip to the next loop iteration; else, perform the followings:

- Store v to `u[parents][step + 1]`.
- Store `u[step_size][step + 1] = v[step_size][step]` to allow step size updating when u is dequeued later.
- Update shortest path length to u if current `u[path_length]` is greater than $step + 1$.
- Enqueue u to the queue in `Q[step + 1]`.

4. When the queue at `Q[step]` is empty, increse *step* by 1.
5. Repeat 3 and 4 until the goal is reached or Q is empty (there is no solution to the goal).

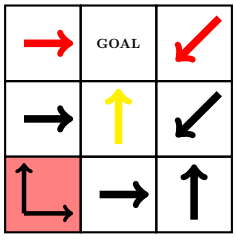
(b)
- The location of a cell in the maze should be written in the format $(row\#, column\#)$. Both row number and column number start from 0.
- The first line states the *width* (number of columns) of the maze.
- The second line specifies the start location in the format "s=$(row\#, column\#)$" (without the quotations marks and any spaces).
- The third line specifies the goal location in the format "g=$(row\#, column\#)$".
- Starting on the forth line, each line represents a row of the maze with *width* number of cells. Each cell, except the goal location, is written in the format "*color*:(*direction*)", where *color* is color of the arrow(s) and *direction* is the direction(s) of the arrow(s) in the cell.
  Each cell representation is separated by a single space.
  - *color* should be one of b, r, or y, which represents black, red, or yellow, respectively.
  - There are 8 possible directions and each should be written as one of the following abbr.

abbr.	description
l	left
r	right
u	up
d	down
ur	up right
ul	up left
dr	down right
dl	down left

  - If there is more than one arrow in a cell, separate each direction with a single comma. The order does not matter.
  - If the cell is the goal location, simply write "*goal*" (without the quotation marks).
- An example representation of the small maze in the handout is as below:
  3
  s=(2,0)
  g=(0,1)
  r:(r,dr,d) goal y:(dl)
  b:(u) b:(u) b:(dl)
  b:(u,r) b:(r) b:(u)

(d) **Test Case 1:** `$ python3 Alice.py test1.in.txt`

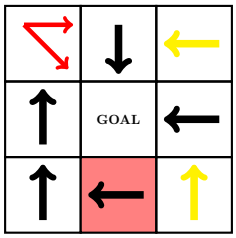
Input: All the paths from the start location either go out of range, go into a cycle, or run out of step size (reach a vertex at some specific *step* where $d < 1$) before reaching the goal in this input maze.



Want to test if the algorithm will stop running properly if it finds that there is no solution to the goal, specifically, if the algorithm will stop building the path in the BFS tree if that path enters a cycle, go out of range, or reach a vertex v that has no out-neighbors at some specific *step* (`v[step_size][step] = 0`). The expected output would be "The maze has no solution." The actual output is also "The maze has no solution."

Test Case 2: `$ python3 Alice.py test2.in.txt`

Input: There are two shortest solutions to the input maze.



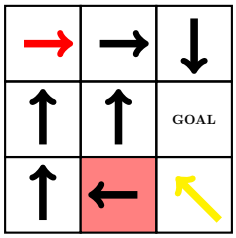
Want to test if the program can return any one of the shortest paths to some specific vertex at some specific *step* when there are more than one shortest paths to that vertex at that *step*. In the input maze, there are two shortest paths to the goal location at *step* 6. The expected output is 6 steps, with either one of the paths:

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

The actual output is 6 steps with a path as same as the first one above.

Test Case 3: `$ python3 Alice.py test3.in.txt`

Input: The shortest path will step on cell (0, 2) twice to get to the goal.



Want to test if the algorithm can build the BFS tree properly when it has to step on some specific vertex more than one time (but each time at a different *step*, that is, from a different in-neighbor) along the way to the goal. Specifically, want to check that a specific vertex is able to maintain which in-neighbor steps on it at which *step* and on it's out-neighbors after that *step*, without them being reset/restored when it is stepped on more than once. The expected output is 9 steps with the path $(2,1) -> (2,0) -> (1,0) -> (0,0) -> (0,2) -> (2,2) -> (1,1) -> (0,1) -> (0,2) -> (1,2)$ on this input maze. The actual output is as same as the expected one.

Test Case 4: `$ python3 Alice.py test4.in.txt`

Input: A maze that is as same as `example_maze.txt` in the handout.

There are two paths with different path lengths to the goal in the example maze. One takes 5 steps with the path $(2,0) -> (1,0) -> (0,0) -> (0,2) -> (1,1) -> (0,1)$ and the other takes 6 steps with the path $(2,0) -> (1,0) -> (0,0) -> (2,2) -> (0,2) -> (1,1) -> (0,1)$. Want to test if the algorithm can correctly return the shortest path if there are more than one paths to the goal. The expected output is the one with 5 steps. The actual output is the same.