Question 1

**part (a): BFS algorithm**

The Alice maze will be represented by a 2-dimensional array of nodes, with each node corresponding to a square in the maze.

**Note:** Terminology from here onwards

* “cell”= “node”=“square” = location in Alice Maze.
* Alice Maze = Graph = Maze= Array of Nodes.

Each node has the attributes:

* **loc:** the (*row, column*) coordinates of the location of the node in the maze.
* **moves:** a list of directions *(horizontal\_step, vertical\_step*) the node can move in.
* **my\_d:** the step size from source node to destination in each (row, column) direction.
* **d\_change:** the size to adjust the current step size by.
* **d\_arrivals:** a dictionary of *d \_value: source\_sqaure* pairs keeping track of all step sizes *d* that were used to enter the square/node.
* **is\_start:** a boolean indicating whether a node represents the start square in the maze.
* This will be used to terminate the backtracking when mapping out shortest path from goal.
* **is\_goal:** a boolean indicating whether a node represents the goal in the maze.
* This will be used to terminate the breadth first search if a path to the goal exists.

We then the traverse the maze from a given start square using Breadth First Search as in slide 22 of 27 of the week 8 lectures with some modifications.

* The major modification is that we do not keep track of whether a cell/square/node has been visited.
* Instead, in each node, we keep track of the step sizes that have been used to visit a node as well as the first source node that was moved from to that node with a given step size (see **d\_arrivals** above).
* Doing this ensures we do not traverse the graph in an infinite cycle.
* Additionally, since there are a finite number of step sizes, this guarantees the search terminates after all possible steps sizes from a node are exhausted OR the shortest path is found.

Assume M is our array of nodes representing an Alice Maze and its squares.

Let S=starting node, and d=initial step size.

**aliceBFS**(M, S, d):

1. Initialize an empty Q
2. S.my\_d d **#** set initial step size of S as d.
3. ENQUEUE(Q, S)  **#** push start square into queue
5. **while** Q **not** empty:
7. cell\_parent DEQUEUE(Q)
8. current\_d cell\_parent.my\_d **#** get current step size from source square.
10. **if** cell\_parent **is** goal:
11. **return** cell\_parent
13. **if** current\_d > 0:
15. **for each** move in cell\_parent.moves:
16. cell\_child destination square by from moving cell\_parent with move
17. **If** cell\_child **is** valid and current\_d **not in** cell\_child.d\_arrivals:
18. cell\_child.setMyD(current\_d)
19. cell\_child.d\_arrivals[current\_d] cell\_parent
20. ENQUEUE(cell\_child)
22. **print** “Destination Not Found”
23. **return** None

**part (b): Text Representation of Alice Maze**

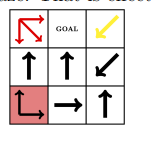
A text representation of a maze needs to store the following information:

1. **Dimensions:** number of rows (row\_num) and number of columns (col\_num) of a maze.
2. **Start:** coordinates of start position in the maze as (row, col).
3. **Goal:** coordinates of goal position in the maze as (row, col).
4. **d:** the initial step size.
5. **Square data:** various attributes of each square.

* A single line of text can represent a square and its data.
* The format of each square presentation can be defined as:
* square\_loc|moves|step change|is\_start|is\_goal
* Information store in each line of text for a square in the maze includes:
* Square Location: coordinates as (row, col).
* Moves - a representation of all directions that can be stepped to from the square.
* Step Change – integer representation of the size to adjust the current step size by.
* is\_start – indication of whether square is start position in maze (T or F)
* Is\_goal – indication of whether the square is goal (T or F)

An example of this representation is of the Alice Maze given in the ps3.pdf handout using lines 1–15 in example\_maze.txt:



1.  row\_num-3



1. col\_num-3
2. start-0,0



1. goal-0,1
2. d-1
3. square\_loc| moves |step change|is\_start|is\_goal
4. 0,0| 1,0#0,1#1,1 |1|F|F



1. 0,1||0|F|T
2. 0,2| 1,-1 |-1|F|F
3. 1,0| -1,0 |0|F|F
4. 1,1| -1,0 |0|F|F



1. 1,2| 1,-1 |0|F|F
2. 2,0| -1,0#0,1 |0|T|F
3. 2,1| 1,0 |0|F|F
4. 2,2| -1,0 |0|F|F

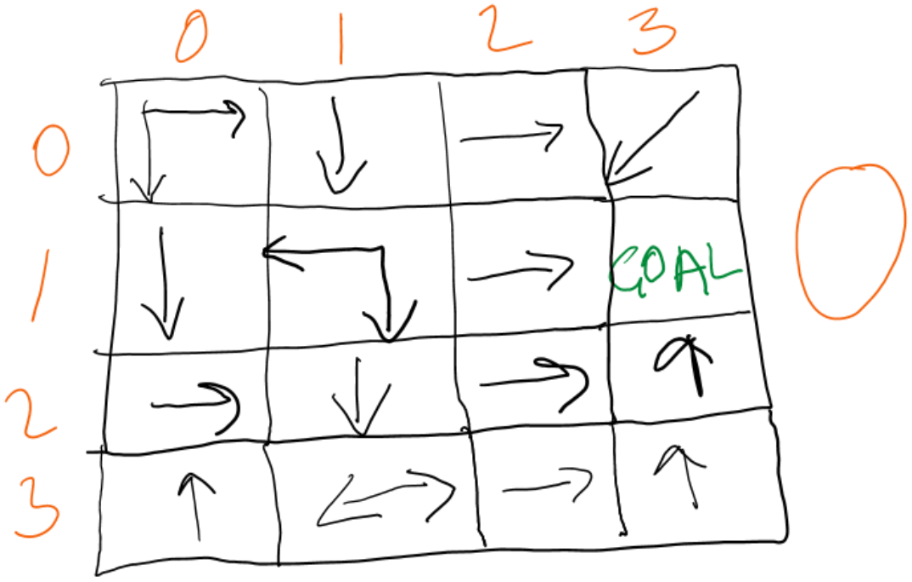
**part (c): Python Implementation**

See Alice.py

**part (d): Tests**

**test1\_in.txt: Algorithmic Correctness**

* This test has been designed to check if the algorithm works as per specifications with a basic maze.
* The 4 x 4 maze has at least one path from start to goal with constant step size of d=1, that is none of the squares is configured to adjust the step size d.
* The **input** of this test is in test1\_in.txt, with the maze represented depicted below.

****



* The **expected output:**

\*\*\*\*\*\*\*\*\*\*\*\*Hello World, I Solve Alice Mazes\*\*\*\*\*\*\*\*\*\*\*\*

Start: (0, 0)

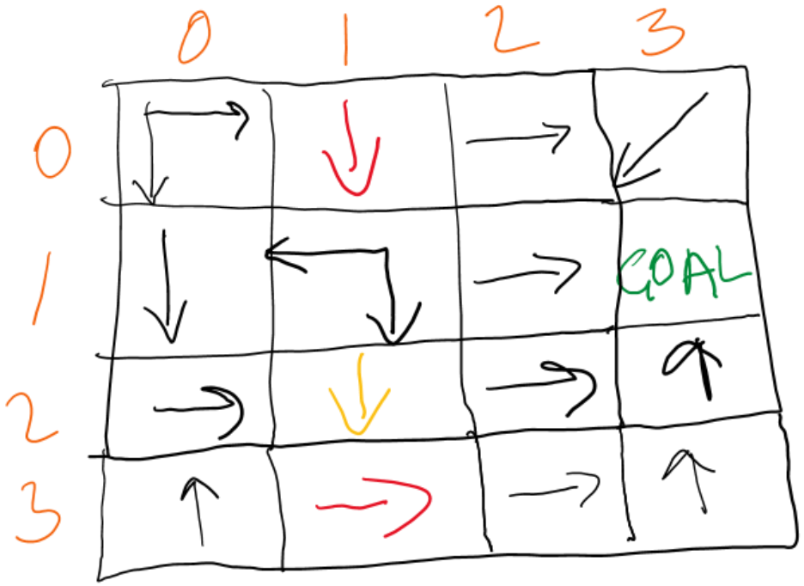
Goal Found: (1, 3)

The shortest path is (0, 0)->(1, 0)->(2, 0)->(2, 1)->(3, 1)->(3, 2)->(3, 3)->(2, 3)->(1, 3)

The shortest path length is 8

**test2\_in.txt: Step Size Change**

* The point of this test is to check if the implantation still finds the shortest goal when some squares are configured to increase and/or decrease the value of the step size d.
* In this case, it turns out the shortest path includes squares in which the size of d is altered.
* The **input** is in test2\_in.txt that has representation of the maze below.





* The **expected output:**

\*\*\*\*\*\*\*\*\*\*\*\*Hello World, I Solve Alice Mazes\*\*\*\*\*\*\*\*\*\*\*\*

Start: (0, 0)

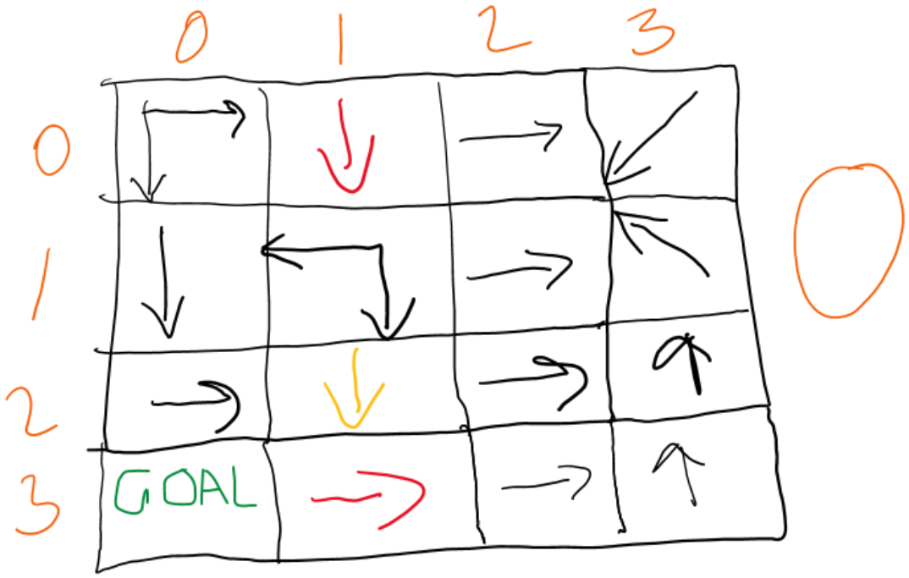
Goal Found: (1, 3)

The shortest path is (0, 0)->(0, 1)->(2, 1)->(3, 1)->(3, 3)->(1, 3)

The shortest path length is 5

**test3\_in.txt: No Path To Goal**

* In this test, we are ensuring the python Alice maze solver reports when there is no path from start to goal when none exists.
* Since test1 and test2 above already confirm the implementation finds the shortest path given changes in step size, we know that ‘no path found’ is reported because there is none.
* The input is in **test3\_in.txt** with pictorial representation of the maze below.

****



* The **expected output:**

\*\*\*\*\*\*\*\*\*\*\*\*Hello World, I Solve Alice Mazes\*\*\*\*\*\*\*\*\*\*\*\*

Start: (0, 0)

Destination is not found

**test4\_in.txt: Multiple Visits To Square**

* Here we check if the program still finds the shortest path where it has to step into at least one square multiple times.
* In those multiple visits to a square, the step size may be adjusted as well.
* The input is in **test4\_in.txt** with pictorial representation of the maze below.





* The **expected output:**

\*\*\*\*\*\*\*\*\*\*\*\*Hello World, I Solve Alice Mazes\*\*\*\*\*\*\*\*\*\*\*\*

Start: (3, 3)

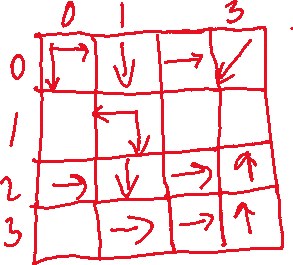
Goal Found: (2, 2)

The shortest path is (3, 3)->(2, 3)->(2, 1)->(2, 3)->(2, 0)->(0, 0)->(0, 2)->(1, 2)->(1, 1)->(2, 1)->(2, 2)

The shortest path length is 10

**test5\_in.txt: Random Start And Goal**

* This test demonstrates the start and goal squares can be chosen arbitrarily, but our Alice maze solver still finds the shortest path.
* In previous tests the start was (0,0) and goal was mostly (1,3).
* For this maze start is (1,2) and goal (3,0).
* The test i**nput** for this test is in test5\_in.txt represented below.



* The **expected output:**

\*\*\*\*\*\*\*\*\*\*\*\*Hello World, I Solve Alice Mazes\*\*\*\*\*\*\*\*\*\*\*\*

Start: (1, 2)

Goal Found: (3, 0)

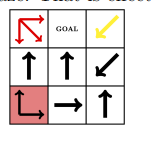
The shortest path is (1, 2)->(1, 1)->(1, 0)->(3, 0)

The shortest path length is 3

**example\_maze.txt: Difference Maze Size**

* This test illustrates that our solver can works on any specified size.
* In this example, the maze is of 3 x 3 but in previous tests we had 4 x 4 mazes.
* This maze also has a looping path: (2, 0)->(2, 1)->(2, 2)->(1, 2)->(2, 1)->(2, 2) ->(1, 2)->(2, 1)->(2, 2)….
* The **input** is example\_maze.txt as with image representation below.







* The **expected output:**

\*\*\*\*\*\*\*\*\*\*\*\*Hello World, I Solve Alice Mazes\*\*\*\*\*\*\*\*\*\*\*\*

Start: (2, 0)

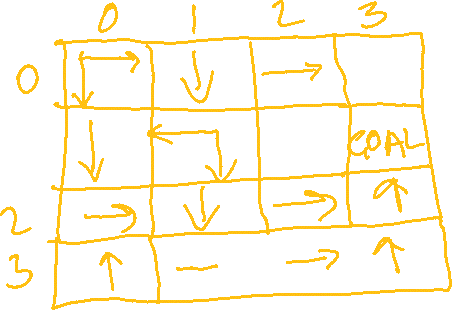
Goal Found: (0, 1)

The shortest path is (2, 0)->(1, 0)->(0, 0)->(0, 2)->(1, 1)->(0, 1)

The shortest path length is 5

**test6\_in.txt: Looping Path – Solution Found**

* Here we demonstrate the solver still finds the shortest path to the goal even if there is a looping path: (0,0) 🡪 (0,1) 🡪 (0,3) 🡪 (1,2) 🡪 (0,1) 🡪 (0,3) 🡪 (1,2) ….
* The **input** file for this test is test6\_in.txt



* The **expected output:**

\*\*\*\*\*\*\*\*\*\*\*\*Hello World, I Solve Alice Mazes\*\*\*\*\*\*\*\*\*\*\*\*

Start: (0, 0)

Goal Found: (1, 3)

The shortest path is (0, 0)->(0, 1)->(2, 1)->(3, 1)->(3, 3)->(1, 3)

The shortest path length is 5

**test7\_in.txt: Looping Path – No Solution**

* Here we demonstrate the solver still terminates if there is no path to the goal even if there is a looping path: (0,0) 🡪 (0,1) 🡪 (0,3) 🡪 (1,2) 🡪 (0,1) 🡪 (0,3) 🡪 (1,2) ….
* The **input** file for this is test7\_in.txt



\*\*\*\*\*\*\*\*\*\*\*\*Hello World, I Solve Alice Mazes\*\*\*\*\*\*\*\*\*\*\*\*

Start: (0, 0)

Destination is not found

