DSA Term project Report: Queue (Rat in a Maze)

ITX 2010/CSX 3003

Anurag Karki
Bachelor of Information Technology, Assumption University

Sanjana Subrahmanya
Bachelor of Information Technology, Assumption University

Manal Mahmood
Bachelor of Computer Science, Assumption University

Content

Part One: The Problem

DMOJ Our project problem starts off by introducing us to the scenario being set. In this hypothetical situation we have a pet rat that is placed in a square maze (visualize a matrix = num of columns and rows). Our maze is made up of walls and pathways, and our rat must figure out if he's able to reach the end of the maze and obtain his cheese.

Constraints Our rat begins his exploration always starting at the top left corner and his reward lies at the bottom right. The rat is able to move in the directions left, right, up and down; but he can only pass through 0's, since the 1's work as walls.
DMOJ our final submission verifier has the specifications that our N (size of the maze) must be $1 \leq N \leq 500$; and that our end result be a simple yes or no: can our rat reach his cheese at the end of the maze or not.

The **Input** we were given to work with is one that looks like our first figure. The first number at the top of the input is our N. In our code this will translate into something that is manually chosen by the tester/us. Right below our given N. We come across a matrix/maze that fits the specification of N * N. The matrix input is filled out with 1's and 0's and while the position of the walls / walkable areas can be different, this is what is provided to us. Later on in this report we'll cover sample inputs different to this and how the solution logic can be easily recognized.

```
4
0 0 1 1
0 0 0 1
0 0 1 0
0 1 0 0
```

Figure 1: Sample Input (DMOJ magicsoup, 2019)

Part Two: Concept of Breadth-first search

Overview

The Breadth First Search algorithm is commonly used in the exploration of nodes and edges in a graph, The BFS algorithm is very efficient in *coming up with the shortest path on unweighted graphs*. As stated above when covering the specifications of the problem, our rat has to begin and end its path in the bottom right & top left corner. The breadth first search starts at a specific "node" and explores all of its "neighbor nodes" before making its decision on where to move. This concept can be visualized quite easily by a matrix like the one our question provides. Our rat starts off at the top left, and explores the "nodes" around them looking for a "0" space to move to. This process continues on and on until all nodes are explored and our rat can determine if the end goal is achievable. This is one way we would be able to solve this solution.

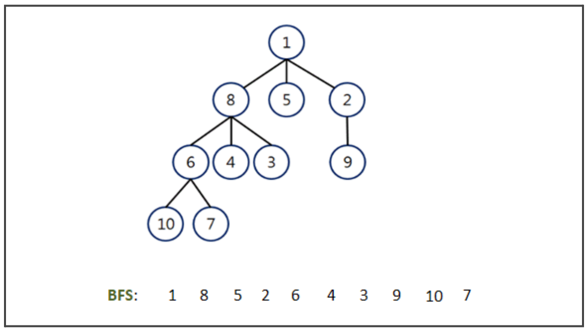
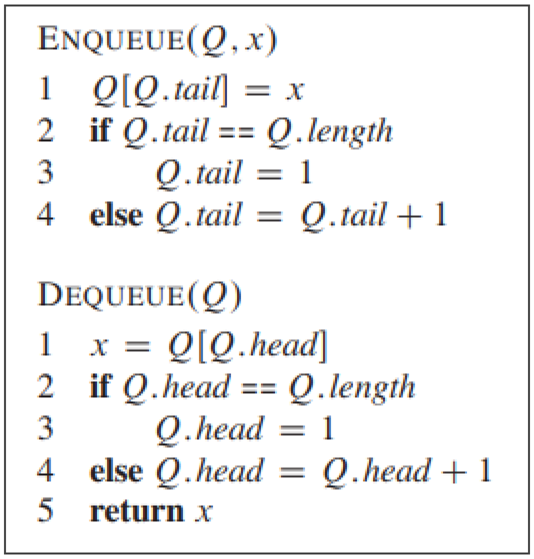


Figure 2: BFS (Anamika Ahmed, 2020)

Part Three: Applied Concept of Queue

Queue is a "first in first out" out structure. What this means is that a queue is open at both ends of the structure, and it works linearly. *"The element which is first pushed into the order, the operation is first performed on that."* Elements enter the array, and exit the array.

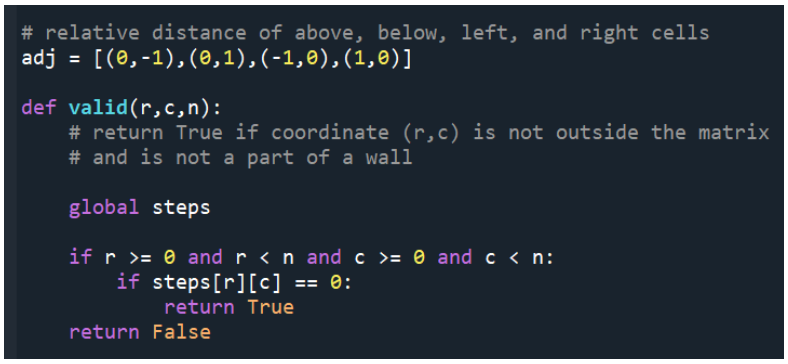


```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3      Q.tail = 1
4  else Q.tail = Q.tail + 1

DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3      Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

Figure 3: Queue Pseudocode (T. Cormen, C. Leiserson, 2009)

The concepts of queues are to error check for underflow and overflow, so it all runs in a single file process (n -1, as element is placed at the head).

Part Four: Code Logic

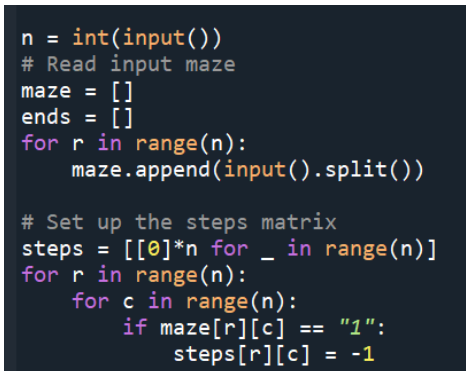


```python
# relative distance of above, below, left, and right cells
adj = [(0,-1),(0,1),(-1,0),(1,0)]

def valid(r,c,n):
    # return True if coordinate (r,c) is not outside the matrix
    # and is not a part of a wall

    global steps

    if r >= 0 and r < n and c >= 0 and c < n:
        if steps[r][c] == 0:
            return True
    return False
```

Figure 4: Rats Movement + bounds

To begin, we start with adjacency. Adj (for short) refers to all the steps that could be valid, (0,-1) is above, (0,1) is one step below, (1,0) is one step left and (-1,0) is one step right. These are all the steps the mouse can do and the goal is to get to the cheese at the end of the maze or rather find a way out of the maze

The next part of the code will check all the steps mentioned before (def valid) and only return true if and only if the coordinate of the following step is not out of bounds. The code will also return False if the mouse comes across a wall, in this case a "1".

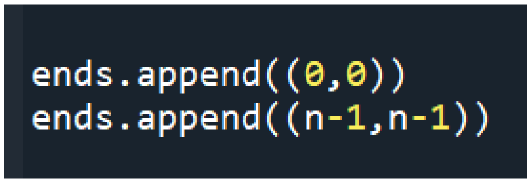


```python
n = int(input())
# Read input maze
maze = []
ends = []
for r in range(n):
    maze.append(input().split())

# Set up the steps matrix
steps = [[0]*n for _ in range(n)]
for r in range(n):
    for c in range(n):
        if maze[r][c] == "1":
            steps[r][c] = -1
```

Figure 5: Input +Movement + bounds

Moving onto our N. The code takes in an integer input "N" from the user first then initializes a matrix of size N x N with the values 0. So, initially the matrix looks like [[0,0,0,],[0,0,0][0,0,0]] if the input is 3

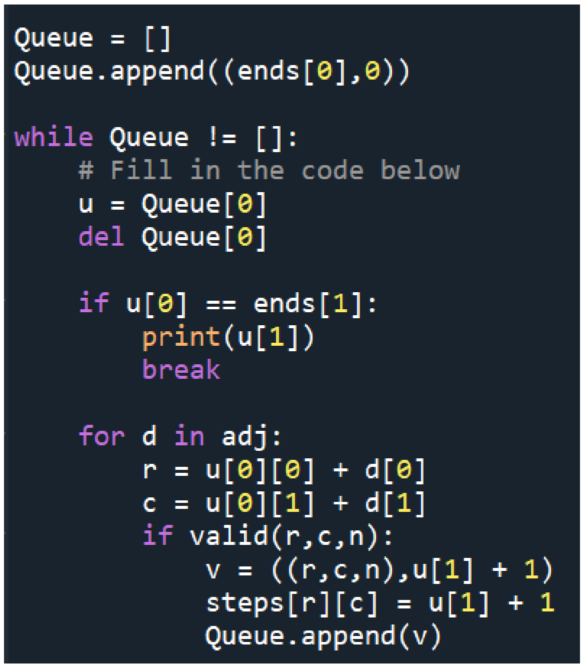
Next the user has to input the matrix by themselves, and to make the input look like a maze and the sample input given, that is, the 1s and 0s that are input by the user are split by a space (" ").



```
ends.append((0,0))
ends.append((n-1,n-1))
```

Figure 6: Starting - Ending position

The next part of the code reads through each element of the matrix and marks any of the "1" s as (-1). This sets up the steps of the matrix. The starting position of the rat is initialized to be the first element of the matrix that is in position (0,0), and the end of the maze is the last position of the matrix (N-1, N-1) .



```
Queue = []
Queue.append((ends[0],0))

while Queue != []:
    # Fill in the code below
    u = Queue[0]
    del Queue[0]

    if u[0] == ends[1]:
        print(u[1])
        break

    for d in adj:
        r = u[0][0] + d[0]
        c = u[0][1] + d[1]
        if valid(r,c,n):
            v = ((r,c,n),u[1] + 1)
            steps[r][c] = u[1] + 1
            Queue.append(v)
```

Figure 7: Queue

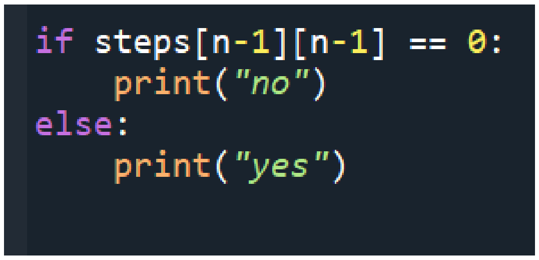
With the queue algorithm, this allows for a breadth first search, which is more optimized than a depth first search in case of the maze problem as it will find the most efficient path from the start to the exit. First, a queue is initialized. The queue has the ends appended and a manual get function is implemented as u gets the first element of the queue and the queue is popped (first element deleted).

This part of the code checks for all the valid moves that the mouse can act on, and the valid moves are added to the queue by adding 1 to the 0 (the mouse can walk through 0s). When the move is considered valid, the row and column coordinate, along with the matrix size is added to a tuple into the queue, which is followed by the step number for the most optimized path.

When the mouse comes across multiple valid moves, ones that it has already crossed, the already explored coordinate is also appended to the queue, the queue keeps on getting appended and popped, and the u keeps getting the first element of the queue.

The mouse can only get to the end of the maze if it is a 0 or the path is not blocked. If the end of the maze, or the last element of the matrix is 1, it would have already been converted to "-1", which with the logic of the code, once the mouse passes through, would not be a valid move and would not be appended in the queue. And the queue would then be popped, causing no more elements in the queue and the program to end. This would cause an output of simply “no”. This also applies to a walled maze with no path to the end.

In the case where there is a path to the end, once the queue is appended and popped, once the mouse gets to the last element of the matrix, which would be a 0, the move would be valid, and when 1 is added, the coordinate is added to the “r” and “c”, or the first two elements of u. This is compared to the last element of ends array which would have the coordinate of the last element of the matrix, and since these would come out as the same tuple, the program breaks.



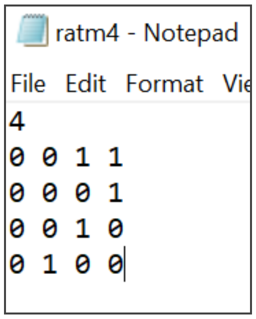
```
if steps[n-1][n-1] == 0:
    print("no")
else:
    print("yes")
```

Figure 8: Path - Yes or No

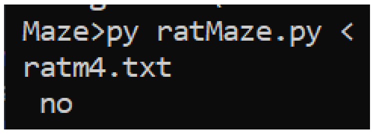
For the outcome, the mouse would not have found the path to the end of the maze, if and only if the end is not explored. Once explored the value of the exit will have been added to become an integer greater than 0. Hence, an unexplored exit will have the value of 0, in which case, if the steps (global variable) has a value of 0 as its last element, the output will be a “no”. Any other value will give an output of “yes” since the path out is found and the mouse does explore the exit.

Part Five: Inputs & Results

Referring to the rats' conditions, we clarified that “0” meant the rat was able to move to that neighbor node position, and “1” meant the rat couldn't pass through. We’ll now take a look at how our code interprets different inputs. Beginning with the standard input we were given. We have an input txt file that looks like the following for N =4. Playing it by eye, and following our “0” pathway, it's obvious the rat cannot make it past this maze to his cheeze. Our code results in:

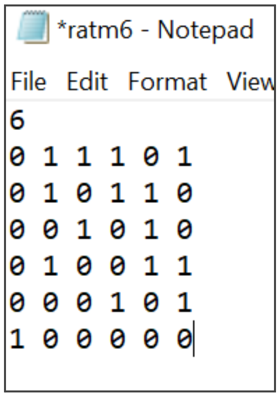


```
4
0 0 1 1
0 0 0 1
0 0 1 0
0 1 0 0
```



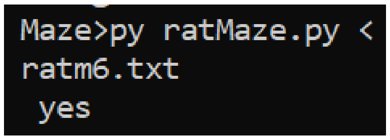
```
Maze>py ratMaze.py <
ratm4.txt
 no
```

With our handmade sample input of N = 6. We constructed the maze so that we start and end with a “0”, and there is a particular pathway of free “0’s” that

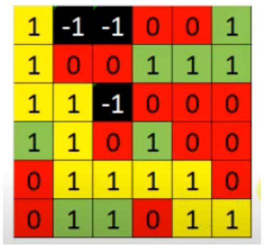


```
6
0 1 1 1 0 1
0 1 0 1 1 0
0 0 1 0 1 0
0 1 0 0 1 1
0 0 0 1 0 1
1 0 0 0 0 0
```

leads to the bottom right. Our code results in:



```
Maze>py ratMaze.py <
ratm6.txt
 yes
```



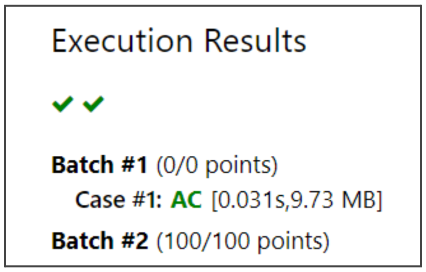
1	-1	-1	0	0	1
1	0	0	1	1	1
1	1	-1	0	0	0
1	1	0	1	0	0
0	1	1	1	1	0
0	1	1	0	1	1

Figure 9: Code Comparison (Rehan Chaudhry 2019)

Referencing the append portion of the code; it can be visualized like this where when the next in line is evaluated and if it is a wall, it is a "-1".

Part Six: Inputs & Results

When placing our code up for evaluation on the DMOJ website, our execution concludes as the following:



Execution Results

✓ ✓

Batch #1 (0/0 points)
  Case #1: AC [0.031s,9.73 MB]
Batch #2 (100/100 points)

All our test cases have been accepted after applying our queue concept to this problem. Our maximum single-case runtime: 0.129s which is an adequate result in the case of our rat deducing if they can acquire their cheese.

References:

[1] magicsoup. "Rat in a Maze - DMOJ: Modern Online Judge." DMOJ, Magicsoup, 2019, https://dmoj.ca/problem/ratmaze.

[2] Fiset, William. "Breadth First Search Algorithm | Shortest Path | Graph Theory." 2 Apr. 2018, https://www.youtube.com/watch?v=oDqjPvD54Ss.

[3] Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, The MIT Press, 3rd Edition.

[4] Queue – Data Structure and Algorithm Tutorials- GeeksforGeeks. (20 Sep, 2022), https://www.geeksforgeeks.org/introduction-to-queue-data-structure-and-algorithm-tutorials/