Solving the Maze by Breadth-First Traversal

Linh Bien

Table of content

Introduction

- In this project, we need to solve the problem whether the ball can go from the start position to the destination position in the Maze (wheeled robots move in a hotel).
- The proposed solution is to apply Breadth First Search.
- The solution will be explained and tested in some examples with diagram and programming.

Design

- **Problem:** Find the way to go from the start position to the destination position in a Maze. The ball can go through the empty spaces by rolling right, left, up, down, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Design

- **Solution:**
- We can use Depth First Search or Breadth First Search.
- Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration, while the Depth First Search (DFS) uses stack.
- Since we need to find the optimal answer so Breadth First Search is a better solution.
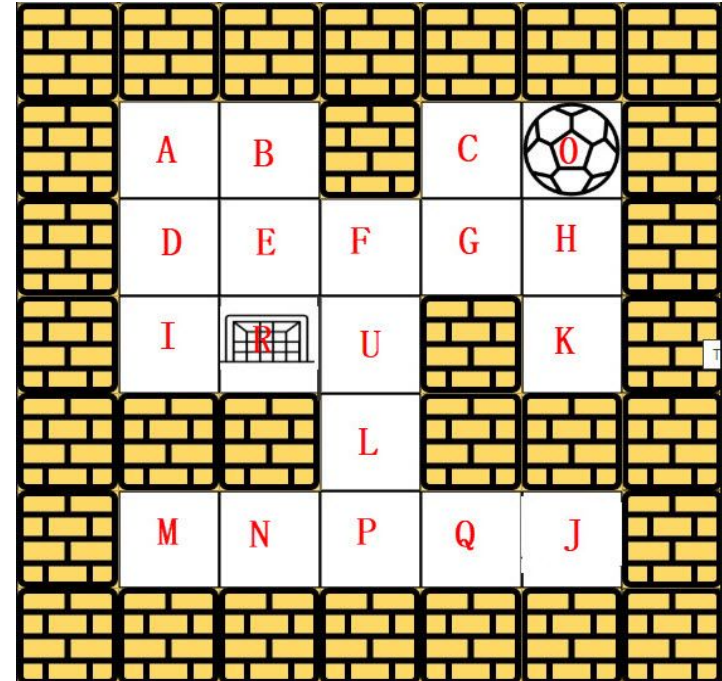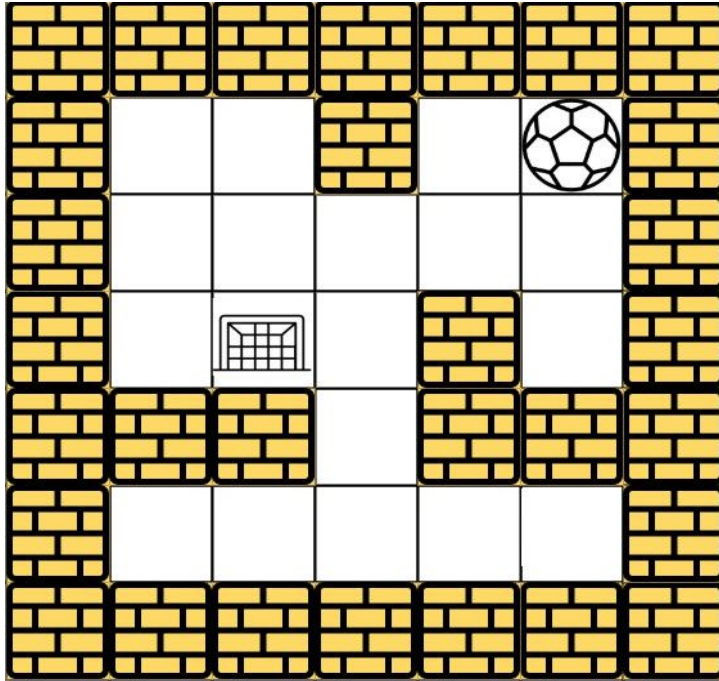
Design

- **Rule 1**
  - Visit the adjacent unvisited vertex.
  - Mark it as visited. Display it.
  - Insert it in a queue.
- **Rule 2**
  - If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3**
 - Repeat Rule 1 and Rule 2 until the queue is empty.
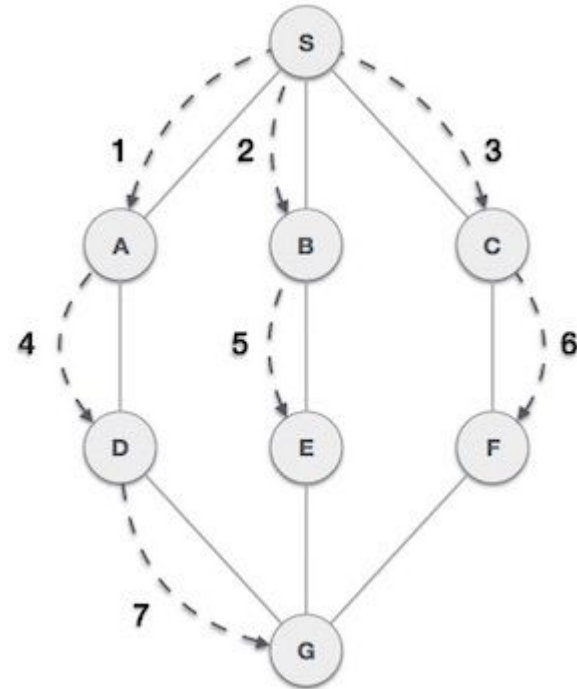
Example

Conduct Depth_First Traversal - Right, Left, Up, Down

Example

Let's view the given search space in a form of a tree:

- Starting position: the root node of the tree
- Right, left, up or down: 4 different routes, 4 branches
- The new node reached from the root traversing over the branch represents the new position occupied by the ball after choosing the corresponding direction of travel
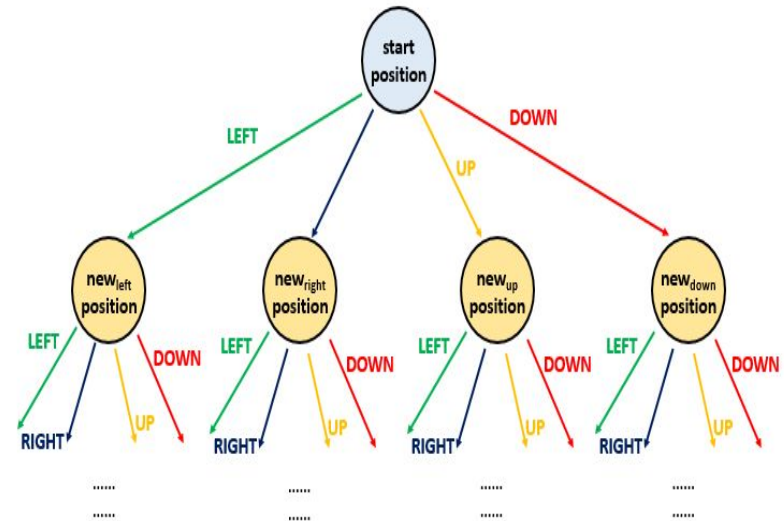
Example

In a tree: 0 is the node, 4 branches: right, left, up, down. Appy BFS:

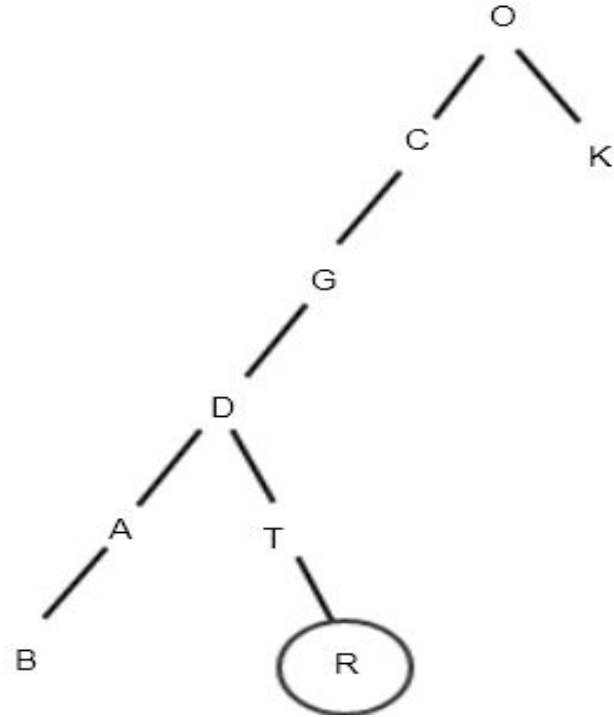From 0 can go to C (or K). K is dead end. Go to C

From C can go to G

From G can go to D. (H is visited)

From D can go to A or T.

From A can go to B

From T can go to R: destination. Return True

Example

Use queue:

Visited: 0 0 Queue:	Visited: 0 1 Queue: 0 1. Add 0 to the queue 2. Mark 0 as visited	Visited: 0 1 Queue: 1. Remove 0 from the queue 2. Print 0	Visited: 0 C K 1 1 1 Queue: C K 1. Add C and K to the queue 2. Mark C and K as visited
Visited: 0 C K 1 1 1 Queue: K 1. Remove C from the queue 2. Print 0 C	Visited: 0 C K G 1 1 1 1 Queue: K G 1. Add G to the queue 2. Mark G as visited	Visited: 0 C K G 1 1 1 1 Queue: G 1. Remove K from the queue 2. Print: 0 C K	Visited: 0 C K G 1 1 1 1 Queue: 1. Remove G from the queue 2. Print 0 C K G

Visited: 0 C K G D
       1 1 1 1 1
Queue: D
1. Add D to the queue
2. Mark D as visited
	Visited: 0 C K G D
	       1 1 1 1 1
	Queue:
	1. Remove D from the queue
	2. Print: 0 C K G D
		Visited : 0 C K G D A T
		       1 1 1 1 1 1 1
		Queue : A T
		1. Add A, T to the queue
		2. Mark A, T as visited
		
		Visited: 0 C K G D A T
		       1 1 1 1 1 1 1
		Queue: T
		1. Remove A from the queue
		2. Print: 0 C K G D A
			Visited: 0 C K G D A T B
			       1 1 1 1 1 1 1 1
			Queue: T B
			1. Add B to the queue
			2. Mark B as visited

Visited: 0 C K G D A T B
       1 1 1 1 1 1 1 1
Queue: B
1. Remove T from the queue
2. Print: 0 C K G D A T
	Visited: 0 C K G D A T B R
	       1 1 1 1 1 1 1 1 1
	Queue: B R
	1. Add R to the queue
	2. Mark R as visited
		Visited : 0 C K G D A T B R
		       1 1 1 1 1 1 1 1 1
		Queue : R
		1. Remove B from the queue
		2. Print 0 C K G D A T B
			Visited : 0 C K G D A T B R
			       1 1 1 1 1 1 1 1 1
			Queue :
			1. Remove R from the queue
			2. Print 0 C K G D A T B R

Implementation

- Implement Breadth First Search

  on Python 3

490. The Maze (Leetcode)

Description:

There is a ball in a `maze` with empty spaces (represented as `0`) and walls (represented as `1`). The ball can go through the empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the `m x n` `maze`, the ball's `start` position and the destination, where start = [start$_{row}$, start$_{col}$] and destination = [destination$_{row}$, destination$_{col}$], return `true` if the ball can stop at the destination, otherwise return `false`.

You may assume that the borders of the maze are all walls
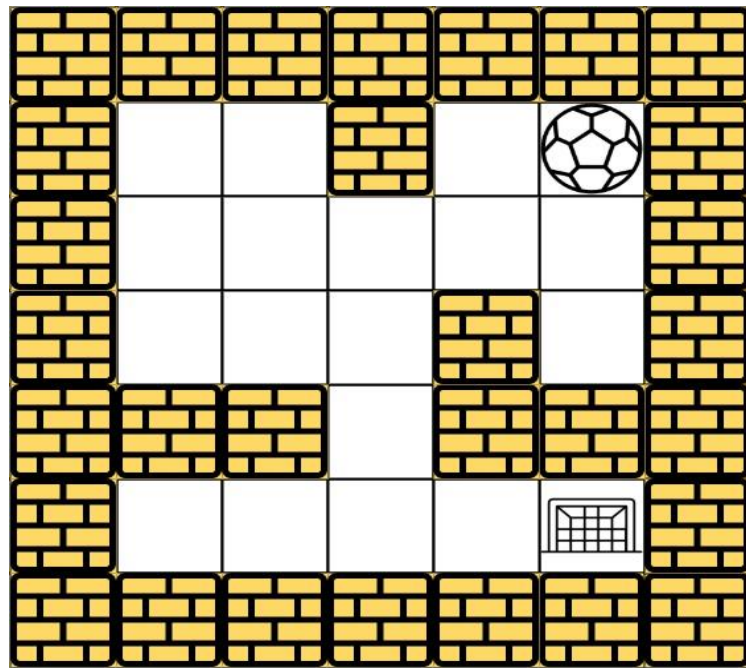
Programming

```python
import collections
from typing import List
class Solution:
    def hasPath(self, maze: List[List[int]], start: List[int], destination
        directions = [(1,0),(-1,0),(0,-1),(0,1)]
        m = len(maze)
        n = len(maze[0])
        q = collections.deque()
        seen = set()
        q.append((start[0], start[1]))
        seen.add((start[0], start[1]))
        while q:
            curr_i, curr_j = q.popleft()
            for d in directions:
                i = curr_i
                j = curr_j
                while 0 <= i < m and 0 <= j < n and maze[i][j] ==0:
                    i += d[0]
                    j += d[1]
                i -= d[0]
                j -= d[1]
                if i == destination[0] and j == destination[1]:
                    return True
                if (i,j) not in seen:
                    q.append((i,j))image.png
                    seen.add((i,j))
        return False
```

maze.py 2   mazebfs.py ✕   groupanagram.py   housepaint.py   wordsearch.py   parindrosub

mazebfs.py > Solution > hasPath

```python
3      class Solution:
4          def hasPath(self, maze: List[List[int]], start: List[int], destination: List[int]) -> bool:
5              directions = [(1,0),(-1,0),(0,-1),(0,1)]
6              m = len(maze)
7              n = len(maze[0])
8              q = collections.deque()
9              seen = set()
10             q.append((start[0], start[1]))
11             seen.add((start[0], start[1]))
12             while q:
13                 curr_i, curr_j = q.popleft()
14                 for d in directions:
15                     i = curr_i
16                     j = curr_j
17                     while 0 <= i < m and 0 <= j < n and maze[i][j] ==0:
18                         i += d[0]
19                         j += d[1]
20                     i -= d[0]
21                     j -= d[1]
22                     if i == destination[0] and j == destination[1]:
23                         return True
24                     if (i,j) not in seen:
25                         q.append((i,j))
26                         seen.add((i,j))
27             return False
28
```

PROBLEMS (12)   DEBUG CONSOLE   OUTPUT   TERMINAL   JUPYTER

```
sers/bienh/OneDrive/Documents/CS summer tremester/Algorithms/mazedfs.py"
Input: [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4], destination = [4,4]
Output:  True
Input: maze = [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4], destination = [3,2]
Output:  False
Input: maze = [[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]], start = [4,3], destination = [0,1]
Output:  False
PS C:\Users\bienh\OneDrive\Documents\CS summer tremester\Algorithms>
```

Ln 25, Col 36   Spaces: 4   UTF-8   CRLF   Python   3.9.12 ('base': conda)

Test

```python
def main():

    test = Solution()

    print("Input: [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4], destination = [4,4]")

    print("Output: ",test.hasPath([[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]],[0,4],[4,4]))


    print("Input: maze = [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4], destination = [3,2]")

    print("Output: ",test.hasPath([[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], [0,4],[3,2]))


    print("Input: maze = [[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]], start = [4,3], destination = [0,1]")

    print("Output: ",test.hasPath([[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]],[4,3],[0,1]))

main()
```

Output

Input: [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4], destination = [4,4]

Output: True

Input: maze = [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4], destination = [3,2]

Output: False

Input: maze = [[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]], start = [4,3], destination = [0,1]

Output: False

Enhancement ideas

- Depth First Search (DFS) can be another solution. Stack will be used instead of queue.

Conclusion

- Although Breadth First Search is less space efficient than Depth First Search, it always returns the optimal answer
- Breadth First Search is applied in finding shortest path and minimum spanning tree for unweighted graph, Peer to Peer Networks (ex: BitTorrent), Crowlers in Search Engines, Social Networking Website, GPS Navigation systems, and so on.

References

Karleigh M, Ken J, Jimin K, Depth-First Search (DFS). Retrieved 07/18/2022 from
https://brilliant.org/wiki/depth-first-search-dfs/#complexity-of-depth-first-search

Tutorialspoint. Data Structure - Depth First Search.
https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm