Topic:
Project on Depth First Search
The Maze
(Slides 1-15)

Practical Applications of Algorithms(CS501)

Guided by
Dr.Henry Chang

Table of Contents

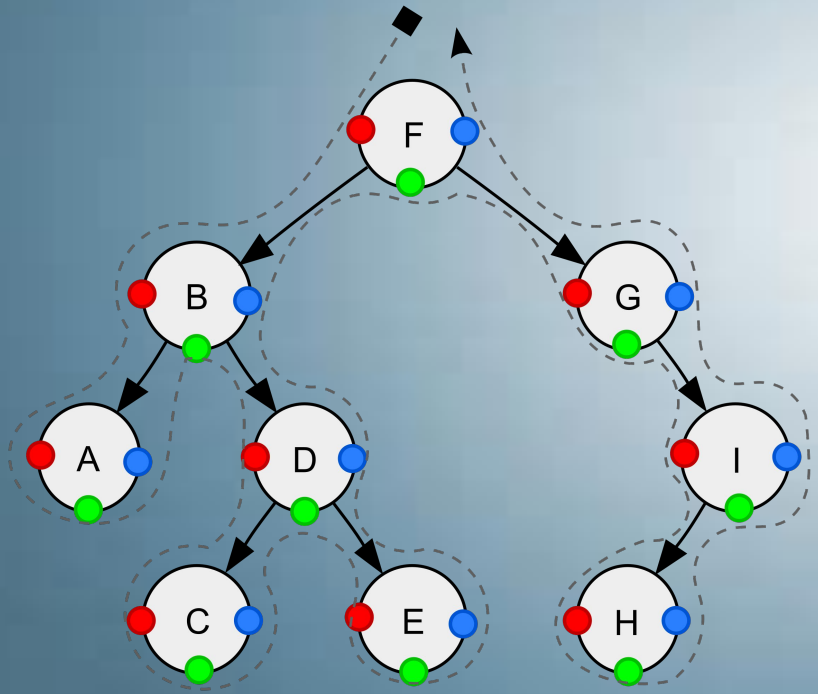
Introduction

- Depth-first traversal (DFS) is a method for exploring a tree or graph.
- In a DFS, you go as deep as possible down one path before backing up and trying a different one. Depth-first search is like walking through a corn maze. You explore one path, hit a dead end, and go back and try a different one

Uses:

- Depth-first search is used in topological sorting, scheduling problems, cycle detection in graphs, and solving puzzles with only one solution, such as a maze or a sudoku puzzle

Design



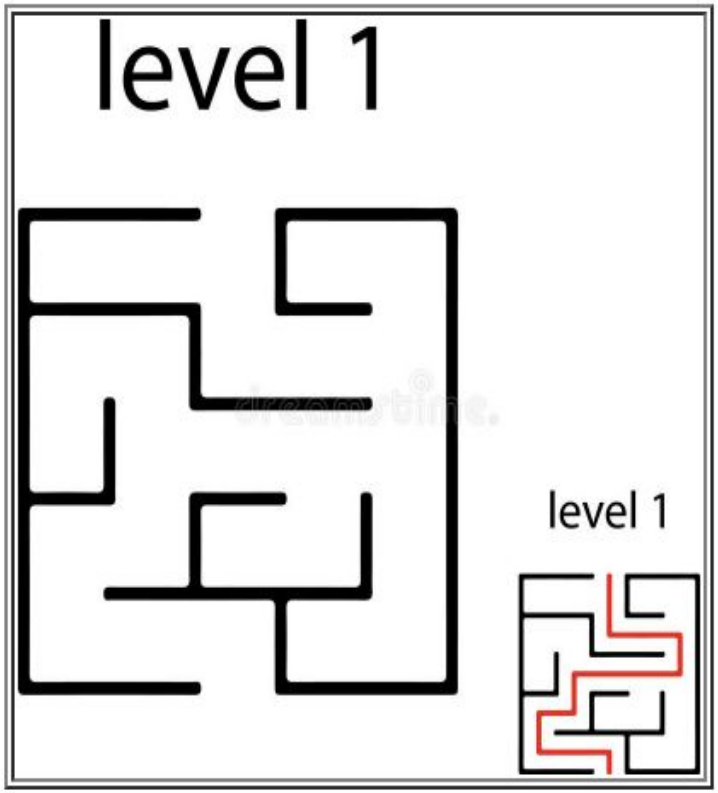
Approach:

- Depth-first search is an algorithm for traversing or searching tree or graph data structures.
- The algorithm starts at the root node and explores as far as possible along each branch before backtracking.
- So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node.
- Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

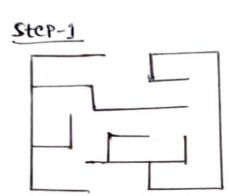
Algorithm:

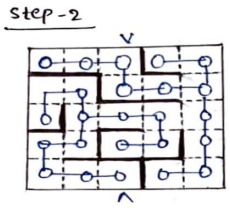
- Create a recursive function that takes the index of the node and a visited array.
- Mark the current node as visited and print the node.
- Traverse all the adjacent and nodes that are unmarked which calls the recursive function with the index of the adjacent node.
- In this traversal algorithm, it will first try a path, then keep going deep until it reaches a dead-end or it finds the final destination which is the solution.
- If it encounters a dead end, we will fall back to the previous point.

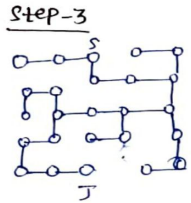
Step 1.1: Tree Approach to Depth-First Traversal to manually solve the problem Maze example

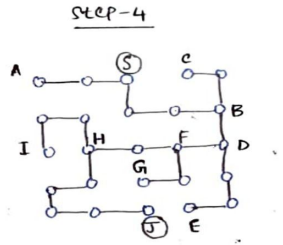


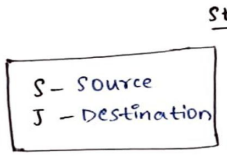
Solution to Clear Route (Street, Highway) Tree

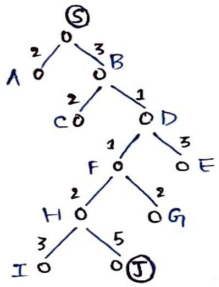










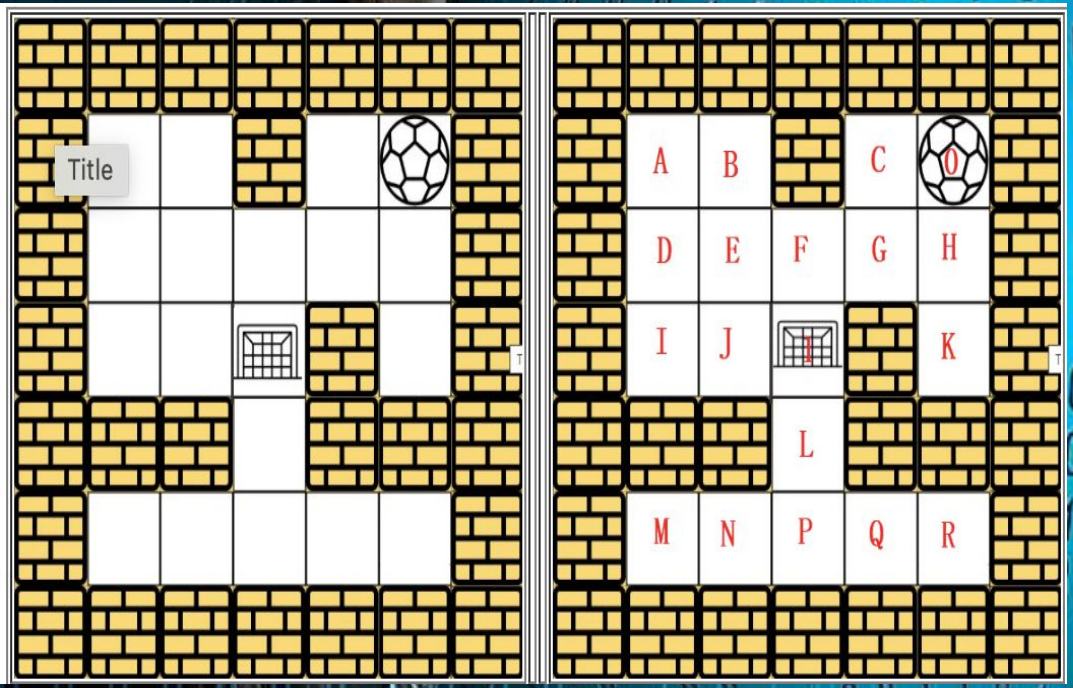


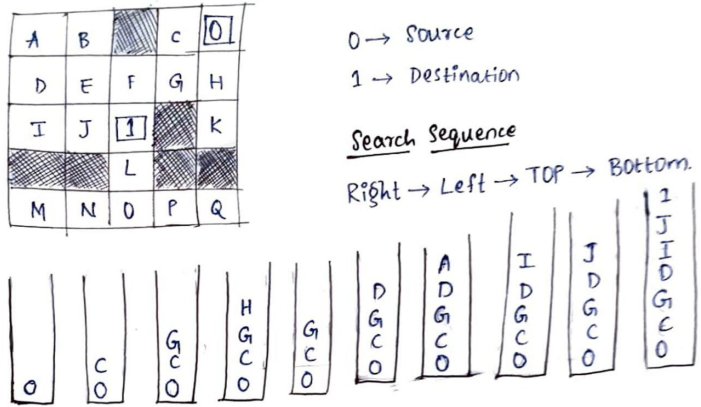
Approach : Depth First Search

- We can view the given search space in the form of a tree.
- The root node of the tree represents the starting position.
- Four different routes are possible from each position i.e. right, left, up or down.
- These four options can be represented by 4 branches of each node in the given tree.
- Thus, 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.

Step 1.2: Matrix Approach to Depth-First Traversal to manually solve the problem Maze example

Unclear Route (Hotel, Hospital) Martrx





Best Approach:

- The best approach is the **Matrix** because the time complexity of this approach is O (V + E) O(V + E) O(V+E) and the space complexity O(bm)
- Where as the time complexity of the tree approach is time complexity of O(m + n) and space complexity is O(n).

Solving Maze Problems:

- For all maze problems, a very simple idea can be used to solve the problem, that is, traverse.

We can start from the starting point.

1. First, determine whether the current point is the existing point.
2. If it is, it means we have found the end; if it is not, then we need to continue the traverse.
3. Then go to its right point. And repeat step 1 .
4. Then go to its below point. And repeat step 1.Then go to its left point. And repeat step 1.
5. Then go to its above point. And repeat step 1.

Applications:

1. Finding all pair shortest path in an undirected graph.
2. Detecting cycle in a graph.
3. Path finding.
4. Topological Sort.
5. Testing if a graph is bipartite.
6. Finding Strongly Connected Component.
7. Solving puzzles with one solution.

Python Implementation

```python
from typing import List
class Solution:
    def hasPath(self, maze: List[List[int]], start: List[int], destination: List[int]) -> bool:
        directions= [(1,0),(-1,0),(0,-1),(0,1)]
        m = len(maze)
        n = len(maze[0])

        stack =[]
        seen = set()
        stack.append((start[0],start[1]))
        seen.add((start[0],start[1]))
        while stack:
            cur_i,cur_j =stack.pop()
            for d in directions:
                ni = cur_i
                nj = cur_j
                while 0 <= ni < m and 0 <= nj < n and maze[ni][nj] == 0:
                    ni += d[0]
                    nj += d[1]

                ni -= d[0]
                nj -= d[1]

                if ni == destination[0] and nj == destination[1]:
                    return True

                if(ni,nj) not in seen:
                    stack.append((ni,nj))
                    seen.add((ni,nj))
        return False


if __name__ =="__main__":
    s = Solution()
    #Assumption1
    maze1 =[[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]]
    start1= [0,4]
    des1 =[4,4]
    print(s.hasPath(maze1,start1,des1))


    #Assumption2
    maze3 =[[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]]
    start3= [4,3]
    des3 =[0,1]
    print(s.hasPath(maze3,start3,des3))

```

Test

```python
from typing import List
class Solution:
    def hasPath(self, maze: List[List[int]], start: List[int], destination: List[int]) -> bool:
        directions= [(1,0),(-1,0),(0,-1),(0,1)]
        m = len(maze)
        n = len(maze[0])

        stack =[]
        seen = set()
        stack.append((start[0],start[1]))
        seen.add((start[0],start[1]))
        while stack:
            cur_i,cur_j =stack.pop()
            for d in directions:
                ni = cur_i
                nj = cur_j
                while 0 <= ni < m and 0 <= nj < n and maze[ni][nj] == 0:
                    ni += d[0]
                    nj += d[1]

                ni -= d[0]
                nj -= d[1]

                if ni == destination[0] and nj == destination[1]:
                    return True

                if(ni,nj) not in seen:
                    stack.append((ni,nj))
                    seen.add((ni,nj))
        return False


if __name__ =="__main__":
    s = Solution()
    #Assumption1
    maze1 =[[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]]
    start1= [0,4]
    des1 =[4,4]
    print(s.hasPath(maze1,start1,des1))


    #Assumption2
    maze3 =[[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]]
    start3= [4,3]
    des3 =[0,1]
    print(s.hasPath(maze3,start3,des3))
```

Enhancement Ideas:

- We can improve the performance of DFS by eliminating the cycles.
- The technique, referred to as cycle checking, prevents the generation of duplicate nodes in the DFS search of a graph by comparing each newly generated node to the nodes already on the search path.
- Full cycle checking compares a new node to all nodes on the path
- Parent cycle checking merely compares a new node to the parent of the node being expanded.
- Simple guidelines are presented showing which type of cycle checking should be used on a given problem.

Conclusion:

- Depth-first search is often used as a subroutine in network flow algorithms such as the Ford-Fulkerson algorithm

- DFS is also used as a subroutine in matching algorithms in graph theory such as the Hopcroft–Karp algorithm. Depth-first searches are used in mapping routes, scheduling, and finding spanning trees.

Topic:
Project on Breadth First Search
The Maze
(Slides: 16- 28)

Practical Applications of Algorithms(CS501)

Guided by
Dr.Henry Chang

Table Of Contents

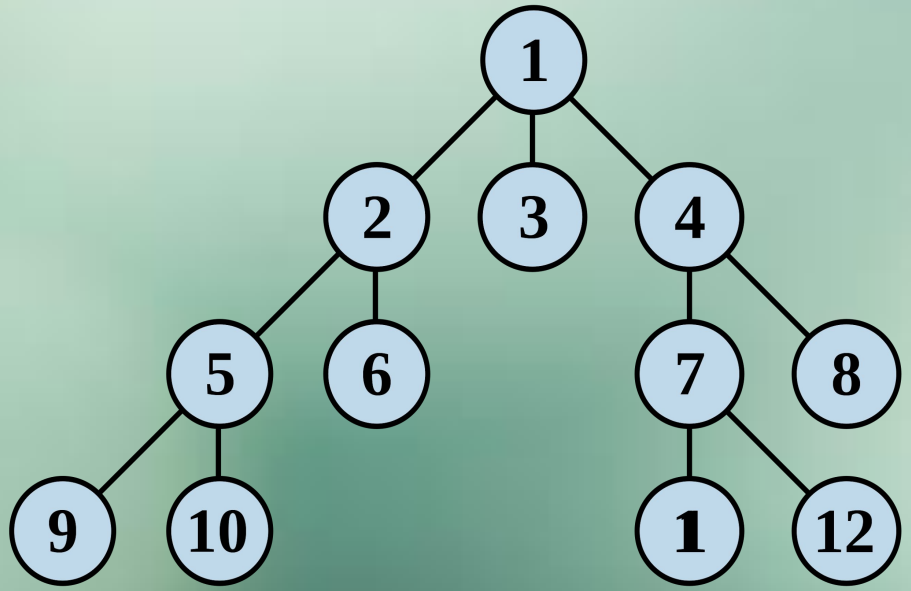
Introduction

- Breadth First Search algorithm traverses a graph in horizontal motion or 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.
- Breadth First Search algorithm traverses a graph in horizontal motion or 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.

Uses:

- The Breadth First Search algorithm is a common way to solve node-based path executions.
- It is an algorithm for searching a tree data structure for a node that

Design



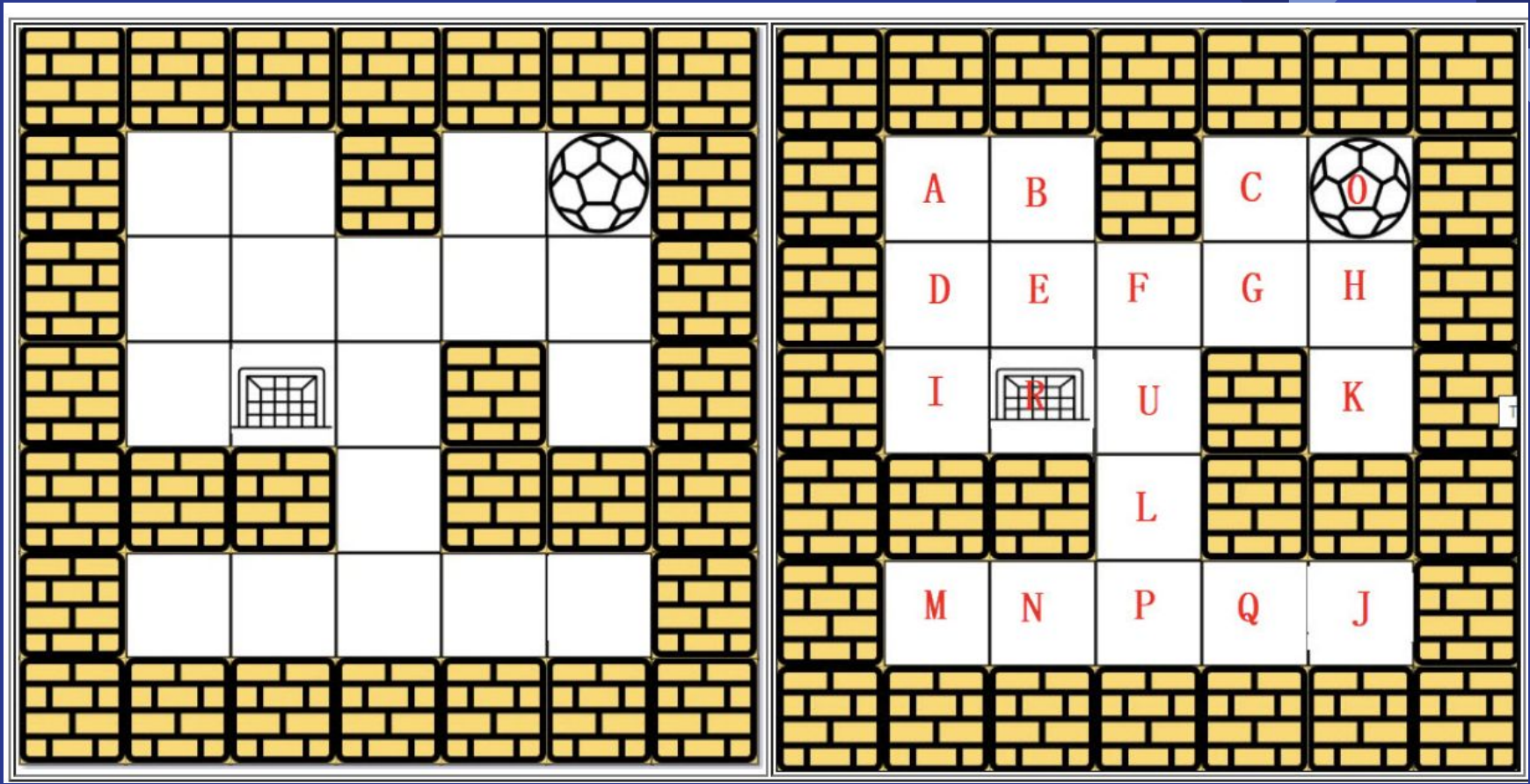
Approach:

- Breadth-first search (BFS) is a method for exploring a tree or graph.
- In a BFS, you first explore all the nodes one step away, then all the nodes two steps away, etc.
- Breadth-first search is like throwing a stone in the center of a pond.
- The nodes you explore "ripple out" from the starting point.

Algorithm:

- Breadth-first search is an algorithm for searching a tree data structure for a node that satisfies a given property.
- It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.
- Traverse all the adjacent and nodes that are unmarked which calls the recursive function with the index of the adjacent node.
- If it encounters a dead end, we will fall back to the previous point.

Manual process to demonstrate concepts using Breadth-First Traversal to solve this problem



Solution to demonstrate concepts using Breadth-First Traversal

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

Working of BFS Algorithm

- A graph traversal is a unique process that requires the algorithm to visit, check, and/or update every single un-visited node in a tree-like structure.
- BFS algorithm works on a similar principle.
- The algorithm is useful for analyzing the nodes in a graph and constructing the shortest path of traversing through these.
- The algorithm traverses the graph in the smallest number of iterations and the shortest possible time.
- BFS selects a single node (initial or source point) in a graph and then visits all the nodes adjacent to the selected node. BFS accesses these nodes one by one.
- The visited and marked data is placed in a queue by BFS. A queue works on a first in first out basis. Hence, the element placed in the graph first is deleted first and printed as a result.
- The BFS algorithm can never get caught in an infinite loop.
- Due to high precision and robust implementation, BFS is used in multiple real-life solutions like P2P networks, Web Crawlers, and Network Broadcasting.

Python Implementation

BFS MAZE.py > ...

```python
from collections import deque
m = 4
n = 3

def Maze(matrix):
    q = deque()
    q.append((0, 0))
    count = 0

    while (len(q) > 0):
        p = q.popleft()

        if (p[0] == n - 1 and p[1] == m - 1):
            count += 1

        if (p[0] + 1 < n and
            matrix[p[0] + 1][p[1]] == 1):
            q.append((p[0] + 1, p[1]))

        if (p[1] + 1 < m and
            matrix[p[0]][p[1] + 1] == 1):
            q.append((p[0], p[1] + 1))
    return count

def main():
    matrix = [ [ 1, 0, 0, 1 ],
               [ 1, 1, 1, 1 ],
               [ 1, 0, 1, 1 ] ]

    print(Maze(matrix))

if __name__ == "__main__" :
    main()
```

Test:

```python
from collections import deque
m = 4
n = 3

def Maze(matrix):
    q = deque()
    q.append((0, 0))
    count = 0

    while (len(q) > 0):
        p = q.popleft()

        if (p[0] == n - 1 and p[1] == m - 1):
            count += 1

        if (p[0] + 1 < n and
            matrix[p[0] + 1][p[1]] == 1):
            q.append((p[0] + 1, p[1]))

        if (p[1] + 1 < m and
            matrix[p[0]][p[1] + 1] == 1):
            q.append((p[0], p[1] + 1))
    return count

def main():
    matrix = [ [ 1, 0, 0, 1 ],
               [ 1, 1, 1, 1 ],
               [ 1, 0, 1, 1 ] ]

    print(Maze(matrix))

if __name__ == "__main__" :
    main()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Python Debug Console - CS501

```
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
Karthiks-MBP:CS501 karthikchatla$  /usr/bin/env /usr/bin/python3 /Users/karthikchatla/.vscode/extensions/ms-python.python-2022.4.1/pythonFiles/lib/python/debugpy/launcher 56576 -- "/Users
sktop/CS501/BFS MAZE.py"
2
Karthiks-MBP:CS501 karthikchatla$
```

Enhancement Ideas

- store all data in an array and make sure to iterate over the array in a standard 'step' size.
- This will make it easier for the CPU to predict memory access.
- Linked lists tend to scatter objects around memory, making it harder for cpus to prefetch data before needing it.

Conclusion

- Breadth Search Algorithm comes with some great advantages to recommend it.
- One of the many applications of the BFS algorithm is to calculate the shortest path.
- It is also used in networking to find neighbouring nodes and can be found in social networking sites, network broadcasting, and garbage collection

Bibliography:

- Google
- Leetcode
- Articles like https://link.springer.com/article/10.1007/BF01389000
- https://bytefish.medium.com/use-depth-first-search-algorithm-to-solve-a-maze-ae47758d48e7
- studylib