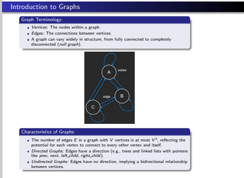


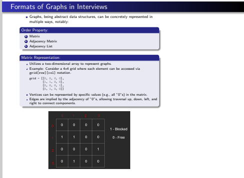
We have actually encountered graphs multiple times so far in the course. A graph is a data structure made up of nodes (which we have seen in form of TreeNodes and ListNodes) and possibly pointers connecting them together.

In graphs, nodes are referred to as vertices and the pointers connecting these nodes are referred to as edges. There are no restrictions in graphs when it comes to where the nodes are placed, and how the edges connect those nodes together.

It is also possible that the nodes are not connected by any edges and this would still be considered a graph - a null graph.

The number of edges, E, given the number of vertices V will always be less than or equal to $E <= V^2$. This is because each node can at most point to itself, and every other node in the graph. If we have a node A, B and C, A can point to itself, B and C. The same goes for B and C, so the rule checks out.

If the pointers connecting the edges together have a direction, we call this a directed graph. If there are edges but no direction, this is referred to as an undirected graph. For example, trees and linked lists are directed graphs because we had pointers like prev, next and left_child, right_child.



A matrix is a two-dimensional array with rows and columns, and a graph can be represented using a matrix. In the code below, each array, separated by commas, represents each row. Here we have four rows and four columns. Everything is indexed by 0 and the idea is that we should be able to access an arbitrary row, column, or a specific element given a specified row or column. Accessing the second row can simply be done by grid[1] and accessing the second column may be done by grid[0][1].

How can this be used to represent a graph? As we mentioned, graphs are abstract and can be defined in many ways. Let's say that all of the 0's in our grid are vertices. To traverse a graph, we are allowed to move up, down, left and right. If we are to connect the 0s together, using our edges, we would end up getting a bunch of connected zeroes, which are connected components, and that denotes a graph. We shall discuss matrix traversal in the next chapter.

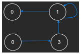
Adjacency Matrix Representation

- In this matrix:
  - A value of 0 indicates no edge between two vertices.
  - A value of 1 indicates the presence of an edge.
- **Example:**

```
adjMatrix = [[0, 0, 0, 0],
             [1, 1, 0, 0],
             [0, 0, 0, 1],
             [0, 1, 0, 0]]
```

- Interpretation:
  - `adjMatrix[1][2] == 0`: No edge between vertex 1 and 2.
  - `adjMatrix[2][3] == 1`: An edge exists between vertex 2 and 3.
- **Space Complexity:** For a graph with V vertices, the space complexity is $O(V^2)$, making it less memory-efficient for sparse graphs.
- Ideal for dense graphs or when constant-time edge lookup is a priority.

Adjacency List: Preferred Graph Representation

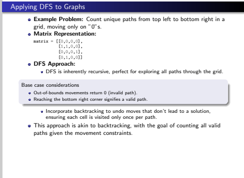


- Represents the graph as an array of lists.
- Each index in the array represents a vertex, and the list at each index contains the neighbors of that vertex.

```
class GraphNode:
    def __init__(self, val):
        self.val = val
        self.neighbors = []
```

- This structure allows easy access to all neighbors of a given vertex, enhancing traversal efficiency.

- **Advantages:**
  - More space-efficient than adjacency matrices, especially for sparse graphs.
  - Directly represents the connections between vertices without redundant information.



In this problem, it is all a matter of choices. You might think of this as similar to backtracking and you would be right. We have mentioned before that DFS is recursive in nature and and we will be using recursion for this. Firstly, we need to think of our base case(s). Well, we know that we can move in all four directions except diagonally. This means that if we go out of bounds, we can return zero.
We know that this will be a brute-force DFS with backtracking since at any point in our path, we might not have a valid way to reach the bottom right, in which case, we will have to backtrack.

DFS in Graphs: Understanding the Base Cases

- **Base Case 1: Path Does Not Exist**
  - Movement leads out of bounds ($r < 0$, $r \geq$ rows, $c < 0$, $c \geq$ cols).
  - Coordinate already visited or is an obstacle ($matrix[r][c] = 1$).
  - **Action:** Return 0 indicating no valid path through this coordinate.
- **Base Case 2: Valid Path Exists**
  - Reached the bottom-right corner without violating any constraints ($matrix[\text{rows} - 1][\text{cols} - 1]$).
  - **Action:** Return 1 to signify a valid path has been found.
- These base cases guide the recursive DFS process, enabling it to backtrack from dead ends and count all unique paths from the top-left to the bottom-right of the matrix.

The base cases
1. A unique path does not exist
Since we are allowed to move in all four directions, it is possible that during our traversal, we end up going out of bounds. This means either our column, c, or our row, r becomes negative, or goes beyond the length of our matrix. It does not matter which of r and c goes out of bounds because we need a valid c AND a valid r to perform our search. We cannot perform a search on matrix[-1][3].
If we have already visited a coordinate, or the current coordinate is 1, then a valid path does not exist through that coordinate. So because a valid path does not exist in all of the aforementioned cases, we will return 0, which denotes absence of a unique path. We shall see this in our code soon.
2. A unique path does exist

If we have not returned 0 from the first case, and we have reached the right-most column and the bottom-most row, it must be the case that we have found a valid path. Remember, our definition of a valid path is if a path exists from matrix[0][0] to matrix[3][3]. We can now return 1 and this will increment our count for the number of unique paths.

2024-03-21

Module-6-Introduction to Algorithms

Matrix DFS

fragile

DFS Implementation for Unique Paths

- To avoid revisiting coordinates, track visited ones in a global `HashSet`.
- Perform DFS recursively in all four directions from any coordinate: $r+1, r-1, c+1, c-1$.
- Increment count based on the return value of DFS calls: 1 signifies a valid path, 0 has no effect.

Key Implementation Steps:

1. Check base cases to determine if the current path is valid or needs backtracking.
2. Add the current coordinate to the visited set to avoid cycles.
3. Recursively call DFS in four directions and manage the count of unique paths.
4. Upon returning from recursive calls, remove the current coordinate from the visited set to allow re-visitation in different paths.

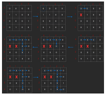
Example Pseudo code

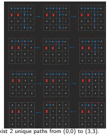
```python
# Matrix (2D Grid)
grid = [[0, 0, 0, 0],
        [1, 1, 0, 0],
        [0, 0, 0, 1],
        [0, 1, 0, 0]]

# Count paths (backtracking)
def dfs(grid, r, c, visit):
    ROWS, COLS = len(grid), len(grid[0])
    if (min(r, c) < 0 or
        r == ROWS or c == COLS or
        (r, c) in visit or grid[r][c] == 1):
        return 0
    if r == ROWS - 1 and c == COLS - 1:
        return 1

    visit.add((r, c))

    count = 0
    count += dfs(grid, r + 1, c, visit)
    count += dfs(grid, r - 1, c, visit)
    count += dfs(grid, r, c + 1, visit)
    count += dfs(grid, r, c - 1, visit)

    visit.remove((r, c))
    return count
```

- This approach allows exploring all potential paths, backtracking as necessary, and accurately counting unique paths from the top-left to the bottom-right of the matrix.

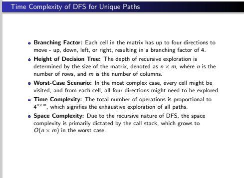
2024-03-21

Visualization of the Implementation

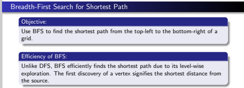




Our function returns 2, denoting there exist 2 unique paths from (0,0) to (3,3).



By now, we understand that our focus is primarily on the worst-case scenario. In such a case, it may be necessary to traverse every single row and column in the matrix. At any given coordinate, movement in all four directions (up, down, left, or right) is possible. Consequently, each coordinate reached by moving in any of these directions can similarly move in four directions. This presents us with four options from each position. Constructing a decision tree to represent this scenario, we observe that each node could have up to four children. The branching factor of the tree is thus 4, and the height of the tree correlates to the size of the matrix, denoted as $n \times m$, where n represents the number of rows and m the number of columns.
Hence, the worst-case time complexity is expressed as $4^{n \times m}$.
Given the recursive nature of the Depth-First Search (DFS) algorithm, the space complexity is primarily influenced by the call stack. Consequently, the space complexity is $O(n \times m)$, accounting for the entire call stack in recursive operations.



Breadth-first search is most commonly used to find the shortest path in a graph. Let's dive into the question straightaway. You will notice that the code for BFS on a graph looks similar to BFS on trees, with some edge cases.

Q: Find the length of the shortest path from top left of the grid to the bottom right.

We can also use DFS to do this but it is more brute-force. BFS is more efficient since the first time a vertex is discovered during the traversal, the distance from our source would give us the shortest path.

Initial Setup

- Define the dimensions of the grid to determine boundaries.
- Utilize a set to track visited vertices, preventing re-visitation.
- Employ a deque (double-ended queue) for managing vertices by levels, initiating with the vertex at $(0,0)$ and marking it as visited.

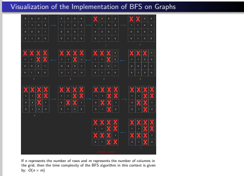
> This approach ensures systematic exploration, with each level in the queue representing a step further from the source, efficiently leading to the identification of the shortest path.

```python
# Matrix (2D Grid)
grid = [[0, 0, 0, 0],
        [1, 1, 0, 0],
        [0, 0, 0, 1],
        [0, 1, 0, 0]]
def bfs(grid):
    ROWS, COLS = len(grid), len(grid[0])
    visit = set()
    queue = deque()
    queue.append((0, 0))
    visit.add((0, 0))
```

Similar to the previous chapter, we take the dimensions of our row and columns, which tells us where our bounds are. We will use a set to keep track of visited vertices. We will use a deque (deck) to keep track of all visited vertices at each level and determine what level we are at currently. We can initialize our deque to the first vertex, (0, 0) and mark it as visited. This is our starting point.

Example Pseudo Code for BFS on Graphs

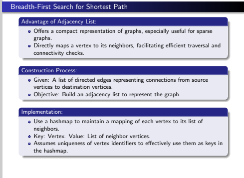
```python
# Shortest path from top left to bottom right
def bfs(grid):
    ROWS, COLS = len(grid), len(grid[0])
    visit = set()
    queue = deque()
    queue.append((0, 0))
    visit.add((0, 0))

    length = 0
    while queue:
        for i in range(len(queue)):
            r, c = queue.popleft()
            if r == ROWS - 1 and c == COLS - 1:
                return length

            neighbors = [[0, 1], [0, -1], [1, 0], [-1, 0]]
            for dr, dc in neighbors:
                if (min(r + dr, c + dc) < 0 or
                    r + dr == ROWS or c + dc == COLS or
                    (r + dr, c + dc) in visit or grid[r + dr][c + dc] == 1):
                    continue
                queue.append((r + dr, c + dc))
                visit.add((r + dr, c + dc))
        length += 1
```



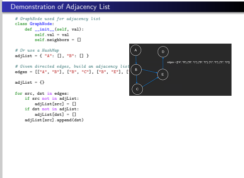
Given the efficient implementation of Breadth-First Search (BFS) on a grid, where each coordinate is visited at most once, we can analyze the time complexity as follows:
By ensuring that no coordinate is visited more than once, the worst-case scenario involves visiting each coordinate in the grid exactly once. If n represents the number of rows and m represents the number of columns in the grid, then the total number of operations required by the BFS algorithm is proportional to the size of the grid, or the product of the number of rows and columns.
Therefore, the time complexity of the BFS algorithm in this context is given by:

$$O(n \times m)$$

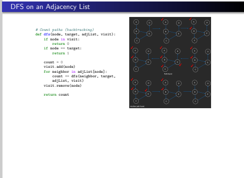
This represents the upper bound on the number of steps required to explore the entire grid using BFS, accounting for the exhaustive traversal of all accessible coordinates.



An adjacency list is probably the "nicest" format out of the three we have covered. Here, we are given a list of directed edges and we have to connect the source to the destinations. In other words, we have to build our adjacency list given an array of edges.



The code demonstrates how we can build an adjacency list. We can use a hashmap where the key is a vertex and it maps to a list of its neighbors, which are also vertices. A hash map works here because we are assuming that all of the values keys are unique.



DFS on an adjacency list

Let's say that we wanted to count the number of paths that lead from a source to destination.

In the code below, we have an adjacency list, a source, and a target. Similar to matrix traversal, we will make use of a hashset called visit to keep track of the vertices that we have already visited.

We will then recursively run DFS on our list until we reach the target node, after which we will return 1. Once we have found a path, we will backtrack by removing nodes from our list and return the count. In the image, the above algorithm is demonstrated. The red check marks indicate that a node has been visited and is in the set.

DFS on an Adjacency List

```python
# Shortest path from node to target
def bfs(node, target, adjList):
    length = 0
    visit = set()
    visit.add(node)
    queue = deque()
    queue.append(node)

    while queue:
        for i in range(len(queue)):
            curr = queue.popleft()
            if curr == target:
                return length

            for neighbor in adjList[curr]:
                if neighbor not in visit:
                    visit.add(neighbor)
                    queue.append(neighbor)
        length += 1
    return length
```

DFS on an adjacency list

Let's say that we wanted to count the number of paths that lead from a source to destination.

In the code below, we have an adjacency list, a source, and a target. Similar to matrix traversal, we will make use of a hashset called visit to keep track of the vertices that we have already visited.

We will then recursively run DFS on our list until we reach the target node, after which we will return 1. Once we have found a path, we will backtrack by removing nodes from our list and return the count. In the image, the above algorithm is demonstrated. The red check marks indicate that a node has been visited and is in the set.