Graphs

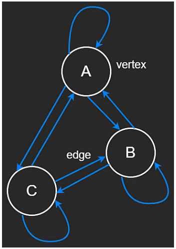
Presented by Yasin Ceran

March 21, 2024

Introduction to Graphs

Graph Terminology:

- *Vertices:* The nodes within a graph.
- *Edges:* The connections between vertices.
- A graph can vary widely in structure, from fully connected to completely disconnected (*null graph*).



Characteristics of Graphs:

- The number of edges E in a graph with V vertices is at most V^2, reflecting the potential for each vertex to connect to every other vertex and itself.
- *Directed Graphs:* Edges have a direction (e.g., trees and linked lists with pointers like *prev*, *next*, *left_child*, *right_child*).
- *Undirected Graphs:* Edges have no direction, implying a bidirectional relationship between vertices.

Formats of Graphs in Interviews

- Graphs, being abstract data structures, can be concretely represented in multiple ways, notably:

Order Property:

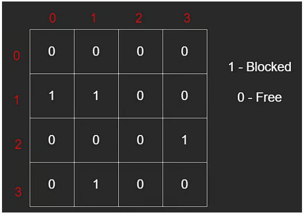
1. Matrix
2. Adjacency Matrix
3. Adjacency List

Matrix Representation:

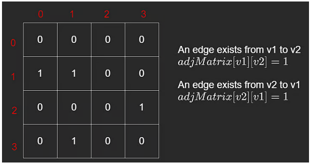
- Utilizes a two-dimensional array to represent graphs.
- Example: Consider a 4x4 grid where each element can be accessed via `grid[row][col]` notation.

```
grid = [[0, 0, 0, 0],
        [1, 1, 0, 0],
        [0, 0, 0, 1],
        [0, 1, 0, 0]]
```

- Vertices can be represented by specific values (e.g., all "0"s) in the matrix.
- Edges are implied by the adjacency of "0"s, allowing traversal up, down, left, and right to connect components.



Adjacency Matrix Representation

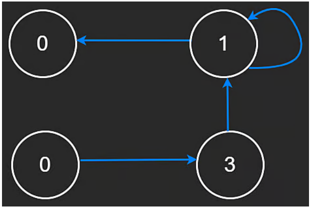


- In this matrix:
  - A value of `0` indicates no edge between two vertices.
  - A value of `1` indicates the presence of an edge.
- **Example:**

```
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.

Applying DFS to Graphs

- **Example Problem:** Count unique paths from top left to bottom right in a grid, moving only on "0"s.
- **Matrix Representation:**

```
matrix = [[0,0,0,0],
          [1,1,0,0],
          [0,0,0,1],
          [0,1,0,0]]
```

- **DFS Approach:**
  - DFS is inherently recursive, perfect for exploring all paths through the grid.

Base case considerations

- Out-of-bounds movements return 0 (invalid path).
- Reaching the bottom right corner signifies a valid path.

  - Incorporate backtracking to undo moves that don't lead to a solution, ensuring each cell is visited only once per path.
- This approach is akin to backtracking, with the goal of counting all valid paths given the movement constraints.

DFS in Graphs: Understanding the Base Cases

- **Base Case 1: Path Does Not Exist**
  - Movement leads out of bounds ($r < 0$, $r \geq \text{rows}$, $c < 0$, $c \geq \text{cols}$).
  - 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.

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:

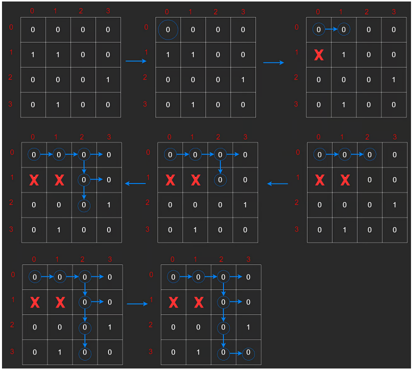
1. Check base cases to determine if the current path is valid or needs backtracking.
2. Add the current coordinate to the visited set to avoid cycles.
3. Recursively call DFS in four directions and manage the count of unique paths.
4. Upon returning from recursive calls, remove the current coordinate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

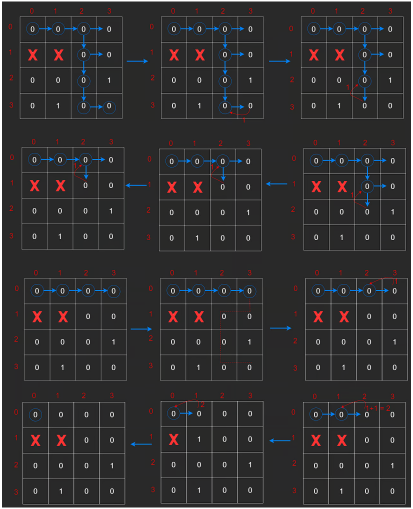
Example Pseudo code

```
# 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.

Visualization of the Implementation





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

Time Complexity of DFS for Unique Paths

- **Branching Factor:** Each cell in the matrix has up to four directions to move - up, down, left, or right, resulting in a branching factor of 4.
- **Height of Decision Tree:** The depth of recursive exploration is determined by the size of the matrix, denoted as $n \times m$, where n is the number of rows, and m is the number of columns.
- **Worst-Case Scenario:** In the most complex case, every cell might be visited, and from each cell, all four directions might need to be explored.
- **Time Complexity:** The total number of operations is proportional to $4^{n \times m}$, which signifies the exhaustive exploration of all paths.
- **Space Complexity:** Due to the recursive nature of DFS, the space complexity is primarily dictated by the call stack, which grows to $O(n \times m)$ in the worst case.

Breadth-First Search for Shortest Path

Objective:

Use BFS to find the shortest path from the top-left to the bottom-right of a grid.

Efficiency of BFS:

Unlike DFS, BFS efficiently finds the shortest path due to its level-wise exploration. The first discovery of a vertex signifies the shortest distance from the source.

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))
```

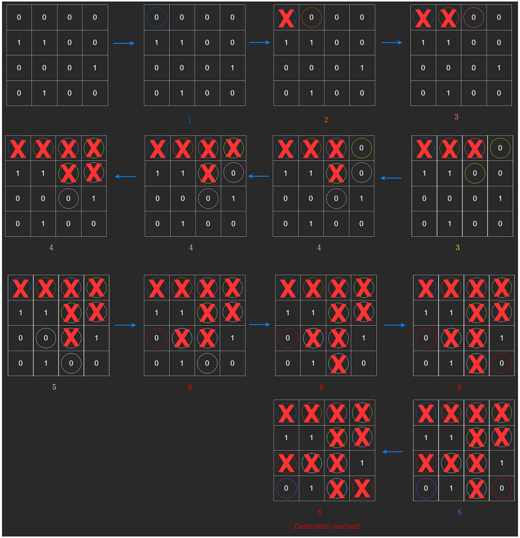
Applying BFS on Graphs for Shortest Path

- **Objective:** Determine the shortest path's length in a grid using BFS.
- **Traversal Process:**
  - Initialize a `length` variable to 0 to track the path length.
  - Use a `while` loop to process each level (vertex) stored in the queue.
  - On dequeuing, we obtain row (`r`) and column (`c`) coordinates, analogous to accessing a node's children in tree BFS.
  - The search continues unless the bottom-right corner ($r = \text{ROWS} - 1$ and $c = \text{COLS} - 1$) is reached.
- **Exploring Neighbors:**
  - Define movement directions with a 2-D array: `neighbors = [[0, 1], [0, -1], [1, 0], [-1, 0]]` for right, left, down, up, respectively.
  - Exclude moves that lead out of bounds, to blocked coordinates, or to already visited vertices.
  - Append viable neighbors to the queue and mark them as visited to prevent reprocessing.
- **Efficiency:**
  - Adding vertices to a visited set upon queueing ensures no duplicates in the queue, enhancing time efficiency.

Example Pseudo Code for BFS on Graphs

```
# 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
```

Visualization of the Implementation of BFS on Graphs



If n represents the number of rows and m represents the number of columns in the grid, then the time complexity of the BFS algorithm in this context is given by: $O(n \times m)$

Breadth-First Search for Shortest Path

Advantage of Adjacency List:

- Offers a compact representation of graphs, especially useful for sparse graphs.
- Directly maps a vertex to its neighbors, facilitating efficient traversal and connectivity checks.

Construction Process:

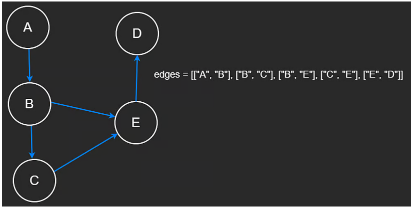
- Given: A list of directed edges representing connections from source vertices to destination vertices.
- Objective: Build an adjacency list to represent the graph.

Implementation:

- Use a hashmap to maintain a mapping of each vertex to its list of neighbors.
- Key: Vertex. Value: List of neighbor vertices.
- Assumes uniqueness of vertex identifiers to effectively use them as keys in the hashmap.

Demonstration of Adjacency List

```python
# GraphNode used for adjacency list
class GraphNode:
    def __init__(self, val):
        self.val = val
        self.neighbors = []

# Or use a HashMap
adjList = { "A": [], "B": [] }

# Given directed edges, build an adjacency list
edges = [["A", "B"], ["B", "C"], ["B", "E"], ["C", "E"], ["E", "D"]]

adjList = {}

for src, dst in edges:
    if src not in adjList:
        adjList[src] = []
    if dst not in adjList:
        adjList[dst] = []
    adjList[src].append(dst)
```

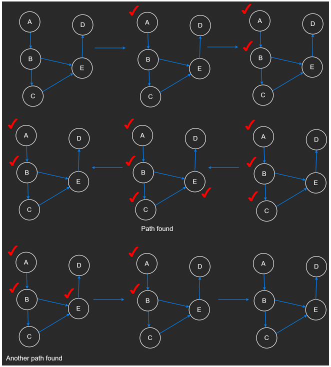


DFS on an Adjacency List

- **Objective:** Count the number of paths from a given source to a destination using DFS on an adjacency list.
- **Adjacency List:** A data structure where each vertex key maps to a list of neighbor vertices, facilitating the representation of graph connections.
- **Traversal Strategy:**
  - Utilize a `HashSet` named `visit` to track visited vertices, ensuring no vertex is explored more than once.
  - Recursively apply DFS starting from the source vertex, exploring all paths through the adjacency list.
  - Upon reaching the target vertex, return 1 to signify a successful path discovery.
  - Implement backtracking by removing nodes from the visited set after exploring paths, enabling the search for new paths.
- **Path Counting:** The recursive nature of DFS, combined with backtracking, allows for an efficient counting of all distinct paths from source to target.
- This method leverages the structure of an adjacency list to explore graph connections deeply, showcasing the flexibility and power of DFS in graph analysis.

DFS on an Adjacency List

```python
# Count paths (backtracking)
def dfs(node, target, adjList, visit):
    if node in visit:
        return 0
    if node == target:
        return 1

    count = 0
    visit.add(node)
    for neighbor in adjList[node]:
        count += dfs(neighbor, target,
        adjList, visit)
    visit.remove(node)

    return count
```



Time Complexity of DFS on an Adjacency List

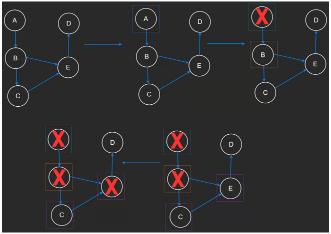
- **Worst-Case Scenario:**
  - Consider a graph where each node is connected to every other node, adhering to the rule $E \leq V^2$, where E is the number of edges, and V is the number of vertices.
  - Assuming each vertex has N edges.
- **Decision Tree Analysis:**
  - A decision tree can model the potential exploration paths from each vertex.
  - With the tree height being V, representing the maximum depth of DFS traversal.
- **Exponential Time Complexity:**
  - In such a dense graph, the DFS algorithm may perform N^V work, reflecting an exponential time complexity.
  - This complexity arises from exploring all possible paths emanating from each vertex, compounded by the graph's dense nature.
- This analysis illustrates the potential computational demands of DFS in the worst-case scenario, echoing discussions from the matrix chapter about the exponential nature of certain graph traversals.

BFS on an Adjacency List for Shortest Path

- **Objective:** Utilize BFS to determine the shortest path from a source node to a target node in a graph represented as an adjacency list.
- **Shortest Path Definition:** The path that connects the source to the destination with the fewest vertices visited.
- **Traversal Method:**
  - Similar to matrix BFS, the approach involves incrementally exploring the graph level by level.
  - Unlike matrix traversal, adjacency list BFS does not encounter boundary edge cases, simplifying the traversal logic.
  - A queue is employed to manage the exploration of vertices, ensuring that vertices are visited in the order they are discovered.
  - The path length is incremented with each level traversed, ensuring that when the target vertex is reached, the shortest path length is recorded.
- **Key Aspects:**
  - The BFS algorithm ensures that the first time the target node is reached, it is via the shortest path due to the algorithm's level-wise exploration.
  - This method effectively utilizes the adjacency list's structure to efficiently explore the graph and find the shortest path.

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
```



Time Complexity of BFS on an Adjacency List

- **Understanding Graph Edges:** The number of edges (E) in a graph is upper bounded by V^2, where V is the number of vertices. This bound assumes a fully connected graph, including self-loops and connections between all pairs of vertices.
- **Adjusting for Real-World Graphs:**
  - Real-world graphs often do not contain self-loops and may not fully connect all vertices, reducing the upper bound on the number of edges.
  - Consequently, the maximal number of edges is not a realistic estimate for many graphs.
- **Time Complexity of BFS:**
  - Considering the practical aspects of graph structures, the time complexity of BFS can be expressed as $O(V + E)$.
  - This complexity accounts for the need to explore all vertices (V) and traverse all edges (E) in the worst-case scenario.
- **Implications:**
  - This formulation highlights the efficiency of BFS in exploring graphs by ensuring that each vertex and edge is considered exactly once in the traversal process.
  - It underscores the importance of both vertices and edges in determining the computational demand of BFS on an adjacency list.

Introduction to Graphs

- Graphs consist of vertices (nodes) and edges (connections between nodes).
- Types of graphs:
  - Directed vs. Undirected
  - Weighted vs. Unweighted
- Graphs can be represented in various forms: adjacency matrices, adjacency lists, edge lists.
- Essential for modeling complex relationships and networks in computer science, engineering, and beyond.

DFS in Matrix

- Depth-First Search (DFS) explores as far as possible along each branch before backtracking.
- In matrices, DFS can navigate through connected components, avoiding revisits to already explored cells.
- Used for problems like finding connected components, solving mazes, and other pathfinding algorithms.
- Time complexity is generally $O(V + E)$ for graphs but tailored to $O(n \times m)$ for matrices, where n and m are dimensions.

BFS in Matrix

- Breadth-First Search (BFS) explores all neighbors of a vertex before going deeper.
- Ideal for finding the shortest path or the minimum number of steps in a grid or matrix representation.
- Utilizes a queue to keep track of the vertices to visit next.
- Time complexity for BFS is also tailored to matrices as $O(n \times m)$.

Adjacency List Representation

- An efficient way to represent graphs, particularly sparse ones.
- Each vertex stores a list of its adjacent vertices, reducing space complexity compared to adjacency matrices.
- Facilitates faster lookups to find all neighbors of a vertex.
- DFS and BFS on adjacency lists have a time complexity of $O(V + E)$, reflecting the need to explore every vertex and edge.