1. We have focused primarily on time complexity in this course, but when choosing data structures, space complexity is often as important of a constraint. Given an adjacency matrix, what is the 'space complexity' in Big-O. That is, given n nodes, how much space (i.e. memory) would I need to represent all of the relationships given. Explain your response.

Answer:

When dealing with an adjacency matrix for a graph with n nodes, the space complexity is O(n^2). This is because the adjacency matrix is represented as a two-dimensional array of size n x n, where each element corresponds to a potential connection between a pair of nodes. If there is an edge between two nodes, the corresponding element in the matrix is set to 1 (or a weight value in the case of a weighted graph); otherwise, it is set to 0.

As a result, the total space required to represent all the relationships in the graph is directly proportional to the square of the number of nodes. Therefore, as the number of nodes increases, the amount of memory needed to store the adjacency matrix grows quadratically. This can become a significant constraint when dealing with large-scale graph data, necessitating careful consideration of memory usage and potential optimizations.

2. Will it ever make sense for ROWS != COLUMNS in an adjacency matrix? That is, if we want to be able to model relationships between every node in a graph, must rows always equal the number of columns in an adjacency matrix? Explain why or why not.

Answer:

In the context of an adjacency matrix representing a graph, it does not make sense for the number of rows to be different from the number of columns. The adjacency matrix is designed to capture the relationships between every pair of nodes in the graph, and each element at position (i, j) represents the existence (or absence) of an edge between node i and node j.

If the number of rows and columns were not equal, it would imply that we are trying to represent relationships between nodes of two different sets, which is not the typical use case for an adjacency matrix. In a standard undirected graph, each node should be able to form an edge with any other node, so the adjacency matrix must be square (i.e., have the same number of rows and columns) to reflect this symmetry.

Even in the case of a directed graph, where edges have a specific direction, the adjacency matrix is still square. The asymmetry in the graph is captured by allowing the matrix to be asymmetric: the presence of an edge from node i to node j does not imply the presence of an edge from node j to node i. However, the number of rows and columns still corresponds to the total number of nodes in the graph.

Therefore, if we want to be able to model relationships between every node in a graph using an adjacency matrix, the number of rows must always equal the number of columns. This ensures that each node in the graph is represented by both a row and a column in the matrix, allowing us to capture all possible edge relationships within the graph.

3. Can you run topological sort on a graph that is undirected?

Answer:

No, you cannot directly run a topological sort on an undirected graph. Topological sort is an algorithm that is specifically designed for directed acyclic graphs (DAGs), which have a clear directionality and no cycles. In a DAG, each edge represents a relationship where one node precedes another node in some order.

An undirected graph, on the other hand, does not have this directionality. Edges in an undirected graph connect pairs of nodes without specifying a direction, meaning that the relationship between two nodes is bidirectional. As a result, there is no inherent "order" or "precedence" between the nodes in an undirected graph, which is essential for a topological sort.

If you have an undirected graph and you want to perform some kind of sorting or ordering based on the relationships between the nodes, you may need to convert the graph to a directed graph first. However, this conversion would require you to make assumptions about the directionality of the edges, which may not always be appropriate or meaningful for your specific application.

In summary, topological sort is not applicable to undirected graphs because they lack the necessary directionality and ordering requirements. If you need to perform some kind of sorting on an undirected graph, you may need to consider alternative algorithms or approaches that are designed to handle undirected graphs.

4. Can you run topological sort on a directed graph that has cycle?

Answer:

No, you cannot run a topological sort on a directed graph that contains a cycle. Topological sort is an algorithm that is designed to order the vertices of a directed acyclic graph (DAG) in a way that for every directed edge uv, vertex u comes before vertex v in the ordering. This ensures that all directed edges point from vertices earlier in the ordering to vertices later in the ordering, reflecting a "precedence" relationship between the vertices.

However, if a directed graph contains a cycle, it is not possible to create such an ordering because there will be at least one vertex that is both preceded and succeeded by another vertex in the cycle. This violates the requirements of a topological sort, which assumes that the graph is acyclic.

Therefore, if you attempt to run a topological sort algorithm on a directed graph with a cycle, the algorithm will either fail to produce a valid ordering or it will detect the presence of a cycle and terminate without providing an output. In either case, you cannot obtain a topological sort for a directed graph that contains a cycle.

5. For question number 4, how would you know you have a cycle in a graph? What algorithm or strategy could you use to detect the cycles? Hint we have already learned about this traversal.

Answer:

To detect cycles in a directed graph, you can use several algorithms and strategies. One of the most common approaches is to use depth-first search (DFS) traversal. DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking. By carefully tracking the visited nodes and their parent nodes during DFS, you can detect cycles in the graph.

Here's a high-level strategy for detecting cycles using DFS:

Start the DFS traversal from any node in the graph.
Mark the current node as visited and push it onto a stack or a recursive call stack.
For each unvisited neighbor of the current node:
a. Recursively perform DFS on the neighbor.
b. If during the recursive DFS call, you encounter a node that is already marked as visited and it is not the parent of the current node, then you have found a cycle.
If no cycles are detected during the recursive DFS calls for all neighbors, backtrack to the previous node and continue the traversal.
Repeat steps 2-4 until all nodes in the graph have been visited.
To keep track of visited nodes and their parents, you can use additional data structures such as a visited array or a set, and a parent array or a map. The parent data structure helps you determine whether the encountered visited node is a part of the current DFS path or not.

There are also other algorithms that can be used to detect cycles in directed graphs, such as breadth-first search (BFS) with color coding, or algorithms specifically designed for cycle detection like Johnson's algorithm or Tarjan's strongly connected components algorithm. However, DFS is a commonly taught and relatively straightforward approach for cycle detection in directed graphs.

Remember that the specific implementation details of the DFS-based cycle detection algorithm can vary depending on the programming language and data structures you are using.