Graphs

Content

1. Representations of graphs

There are two standard ways to represent a graph G = (V, E) as a collection of adjacency lists or as an adjacency matrix. Either way applies to both directed and undirected graphs. Since the adjacency-list representation provides a compact way to represent sparse graphs—those for which |E| is much less than square of |V| — it is usually the method of choice. We may prefer an adjacency-matrix representation, however, when the graph is dense—|E| is close to square of |V| — or when we need to be able to tell quickly if there is an edge connecting two given vertices.

2. Adjacency-list representation

a. Introduction

The adjacency-list representation of a graph G = (V, E) consists of an array Adj of |V| lists, one for each vertex in V . For each u belonging to V , the adjacency list Adj[u] contains all the vertices such that there is an edge (u, v) belonging to E. That is, Adj[u] consists of all the vertices adjacent to u in G. Since the adjacency lists represent the edges of a graph, in pseudocode we will treat the array Adj as an attribute of the graph, just as we treat the edge set E.

b. Approach

If G is a directed graph, the sum of the lengths of all the adjacency lists is |E|, since an edge of the form (u, v) is represented by having appeared in Adj[u]. If G is an undirected graph, the sum of the lengths of all the adjacency lists is 2 |E|, since if (u, v) is an undirected edge, then u appears in v's adjacency list and vice versa. For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is O(V + E). We can readily adapt adjacency lists to represent weighted graphs, that is, graphs for which each edge has an associated weight, typically given by a weight function w : E --> R. For example, let G = (V, E) be a weighted graph with weight function w. We simply store the weight w(u, v) of the edge (u, v) belonging to E with vertex in u’s adjacency list.

c. Disadvantage
A potential disadvantage of the adjacency-list representation is that it provides no quicker way to determine whether a given edge (u, v) is present in the graph than to search for in the adjacency list Adj[u].

3. <u>**Adjacency-matrix representation**</u>

a. Introduction
An adjacency-matrix representation of the graph remedies the above mentioned disadvantage of the adjacency list, but at the cost of using asymptotically more memory.

b. Approach
For the adjacency-matrix representation of a graph G = (V, E), we assume that the vertices are numbered 1, 2, ….., |V| in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a |V| X |V| matrix A = (aij) such that
aij = 1 if (i, j) belongs to E
aij = 0 otherwise
Like the adjacency-list representation of a graph, an adjacency matrix can represent a weighted graph. For example, if G =(V, E) is a weighted graph with edge weight function w, we can simply store the weight w(u, v) of the edge (u, v) belonging to E as the entry in row u and column of the adjacency matrix. If an edge does not exist, we can store a NIL value as its corresponding matrix entry, though for many problems it is convenient to use a value such as 0 or infinity.

c. Disadvantage
In case of sparse graphs, huge memory is wasted for storing the graphs as 2D adjacency matrix.

4. <u>**Breadth-first search**</u>

a. Introduction
Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Given a graph G = (V, E) and a distinguished source vertex s, breadth-first search systematically explores the edges of G to “discover” every vertex that is reachable from s.
Breadth-first search for a graph is similar to Breadth-First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

b. Steps
   i. Create a visited array of bool type, initially having all values as false.

ii. Create a queue that stores the pending Nodes to be visited.
iii. Use a while loop until the queue becomes until.
iv. In each loop, remove out the front element of the queue and print the vertex of the front.
v. If the front's vertex is visited, then continue.
vi. Else, if it is not visited, then iterate over its neighbours and push them into the queue and mark the neighbours as visited.

c. Implementation
Visit the link
https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Graphs/Codes/breadth_first_search.cpp

d. Time Complexity
O(V+E) where V is a number of vertices in the graph and E is a number of edges in the graph.

e. Space Complexity
O(V), since an extra visited array is needed of size V.

5. Depth-first search

1. Introduction
Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) 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.
Depth-first search for a graph is similar to Depth-first traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, a node may be visited twice. To avoid processing a node more than once, use a boolean visited array.

2. Steps
    a. Create a recursive function that takes the index of the node and a visited array.
    b. Mark the current node as visited and print the node.
    c. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.
3. Implementation
Visit the link-
https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Graphs/Codes/depth_first_search.cpp

4. Time Complexity

O(V+E) where V is a number of vertices in the graph and E is a number of edges in the graph.

5. Space Complexity
   O(V), since an extra visited array is needed of size V.