**What are graphs?**

An abstract way of representing connectivity using nodes (or vertices) and edges

Graph G=(V,E) , E ⊆ V\*V

We will label the nodes from 1 to n (V)

m edges connect some pairs of nodes (E)

Edges can be either one-directional (directed) or bidirectional

Nodes and edges can have some auxiliary information

**Terms used**

Vertex: node of a graph

Adjacency(u) = {v | (u, v) \in E}

Degree(u) = |Adjacency(u)|

Subgraph: A subset of vertices and edges is a subgraph

Walk: A sequence v\_1, v\_2, \ldots, v\_k such that (v\_i, v\_{i+1}) \in E

Trial: A trial is a walk in which no edge occurs twice

Closed path: A walk where starting and ending vertex are the same

**Storing Graphs**

We need to store both the set of nodes V and the set of edges E

Nodes can be stored in an array. Edges must be stored in some other way.

We want to support the following operations

- Retrieving all edges incident to a particular node

- Testing if given two nodes are directly connected

**Adjacency matrix**

An easy way to store connectivity information

Checking if two nodes are directly connected: O(1) time

Make an n × n matrix A

a[i][j] = 1 if there is an edge from i to j

a[i][j] = 0 otherwise

Uses Θ(n^2 ) memory. So, use when n is less than a few thousands, AND when the graph is dense

**Adjacency List**

Each vertex maintains a list of vertices that are adjacent to it.

lists have variable lengths

We can use: vector< vector<int> >

Space usage: Θ(n + m)

Checking if edge (Vi,Vj) is present in G: O(min(deg(Vi),deg(Vj)))

**Special Graphs**

Implicit graphs

Two squares on an 8x8 chessboard. Determine the shortest sequence of knight moves from one square to the other.

Tree: a connected acyclic graph

The most important type of graph in CS

Alternate definitions (all are equivalent!)

- connected graph with n − 1 edges

- An acyclic graph with n − 1 edges

- There is exactly one path between every pair of nodes

- An acyclic graph but adding any edge results in a cycle

- A connected graph but removing any edge disconnects it

Directed Acyclic Graph (DAG)

Bipartite Graph

Nodes can be separated into two groups S and T such that edges exist between S and T only (no edges within S or within T)

\section{Graph Traversal}

**Graph Traversal**

The most basic graph algorithm that visits nodes of a graph in certain order

Used as a subroutine in many other algorithms

We will cover two algorithms

Depth-First Search (DFS): uses recursion

Breadth-First Search (BFS): uses queue

**Depth First Search**

DFS(v): visits all the nodes reachable from v in depth-first order

- Mark v as visited

- For each edge v → u:

If u is not visited, call DFS(u)

Use non-recursive version if recursion depth is too big (over a few thousands)

Replace recursive calls with a stack

Uses:

Biconnected components

A node in a connected graph is called an articulation point if the deletion of that node disconnects the graph.

A connected graph is called biconnected if it has no articulation points. That is, the deletion of any single node leaves the graph connected.

Complexity

Time: O(|V|+|E|)

Space: O(|V|) [to maintain the vertices visited till now]

**Breadth First Search**

BFS(v): visits all the nodes reachable from v in breadth-first order

- Initialize a queue Q

- Mark v as visited and push it to Q

- While Q is not empty:

Take the front element of Q and call it w

For each edge w → u:

If u is not visited, mark it as visited and push it to Q

Uses:

Finding a Path with Minimum # of edges from starting vertex to any other vertex.

Solve Shortest Path problem in unweighted graphs

Same Time and Space Complexity as DFS.

SPOJ Problem http://www.spoj.pl/problems/PPATH/

**Topological Sort**

Input: a DAG G = V, E

Output: an ordering of nodes such that for each edge u → v, u comes before v

There can be many answers

- Precompute the number of incoming edges deg(v) for each node v

- Put all nodes with zero degree into a queue Q

- Repeat until Q becomes empty:

- Take v from Q

- For each edge v → u

Decrement deg(u) (essentially removing the edge v → u)

If deg u becomes zero, push u to Q

Time complexity: Θ(n + m)

\section{MST}

**Minimum Spanning Tree (MST)**

Given an undirected weighted graph G = V, E

Want to find a subset of E with the minimum total weight that connects all the nodes into a tree

There are two algorithms:

- Kruskal’s algorithm

- Prim’s algorithm

**Kruskal’s Algorithm**

Main idea: the edge e with the smallest weight has to be in the MST

Keep different supernodes, which are “local MST’s” and then join them by adding edges to form the MST for the whole graph

Pseudocode:

Sort the edges in increasing order of weight

Repeat until there is one supernode left:

Take the minimum weight edge e⋆

If e\* connects two different supernodes:

Connect them and merge the supernodes

Otherwise,

ignore e \*

**Prim’s Algorithm**

Reading Homework

\section{Shortest path algos}

**Floyd-Warshall Algorithm**

All pair shortest distance

Runs in Θ(n^3) time

Extremely easy to code

Define f(i, j, k) as the shortest distance from i to j, using 1 ... k as intermediate nodes

- f(i, j, n) is the shortest distance from i to j

- f( i, j, 0) = cost(i, j)

The optimal path for f i, j, k may or may not have k as an intermediate node

- If it does, f (i, j, k) = f (i,k k-1) + f(k, j, k-1)

- Otherwise, f (i, j, k) = f (i, j, k-1)

Therefore, f (i, j, k) is the minimum of the two quantities above

Pseudo code:

Initialize D to the given cost matrix

For k = 1 ... n:

For all i and j:

d\_{ij} = min{d\_{ij} , d\_{ik} + d\_{kj}}

Can also be used to detect negative weight cycles in graph

How?

If d\_{ij}+ d\_{ji}< 0 for some i and j, then the graph has a negative weight cycle

**Dijkstra’s Algorithm**

Used to solve Single source Shortest Path problem in Weighted Graphs

Only for Graphs with positive edge weights.

the algorithm finds the path with lowest cost (i.e. the shortest path) between that source vertex and every other vertex

Greedy strategy

Idea: Find the closest node to s, and then the second closest one, then the third, etc

Pseudo code:

Maintain a set of nodes S, the shortest distances to which are decided

Also maintain a vector d, the shortest distance estimate from s

Initially, S = s , and d\_v = cost(s, v)

Repeat until S = V:

Find v ∉ S with the smallest d\_v , and add it to S

For each edge v → u of cost c:

d\_u = min{ d\_u , d\_v + c}

Time complexity depends on the implementation:

Can be O(n^2 + m) , O(m log n) , O(n log n)

Use priority\_queue<node> for implementing Dijkstra’s

SPOJ Problem

http://www.spoj.pl/problems/CHICAGO

**Bellman-Ford Algorithm**

Single source shortest path for negative weights

Can also be used to detect negative weight cycles

Pseudo code:

Initialize d\_s = 0 and d v = ∞ for all v ≠ s

For k = 1 ... n − 1:

For each edge u → v of cost c:

d\_v = min{ d\_v , d\_u + c}

Runs in Θ(nm) time

Extremely easy to code