Sorting

Inserting Sort

Graph

## Terminology

**Sparse**: not so many edges; **Dense**: many

**Complete Graph**: Simple graph with N vertices and nC2 edges

**In/Out Degree of a Vertex**: # of in/out edges from a vertex

**(Simple) Path**: Sequence of Vertices adjacent to each other;

**(Simple) Cycle**: Path that starts and ends with the same vertex.

**Simple** means: with no repeated vertex except start/end

**Path Length/Cost**:

undirected : # of edges in the path

directed: sum of edge weight in the path

**Component**: A group of vertices that can visit each other via some path

**Connected Graph**: graph with 1 component

**Sub Graph**: subset of vertices of the original graph

**DAG**: Directed Acyclic Graph

**Tree**: Connected graph, one **unique path** between any pair of vertices.

**Bipartite Graph**: if we can partition the vertices into two sets so that there is no edge between members of the same set

Representation of a graph:

Adjacency Matrix/ Adjacency List/ Edge List

Graph

**BFS Pseudo Code:**

BFS(u)

for all v in V

visited[v] = 0

p[v] = -1

Q = {S} // Q is a queue

visited[s] = 1

while Q is not empty

u = Q.dequeue()

for all v adjacent to u

**DFS Pseudo Code:** DFSrec(u)

visited[v] = 1;

for all v adjacent to u

if(visited[v]==0)

p[v] = u;

DFSrec(v);

// In the main method

for all v in V

visited[v] = 0;

p[v] = -1;

DFSrec(s)

**Path reconstruction:**

*void backtrack(u)*

if(u==-1) stop;

backtrack(p[u]);

output u;

*// in main method:*

Output "Path: ";

backtrack(t);

**TopoSort**

*topoVisit(u)*

forall v adjacent to u:

if(visited[v]==0)

topoVisit[v];

append u to the back of toposort

*// in main method:*

for all v in V

visited[v] = 0;

clear toposort;

for all s in V:

if(visited[s]==0)

topoVisit(s)

reverse toposort and output it

**Union-Find Disjoint Sets**

public class UnionFind {

private Vector<Integer> pset;

public UnionFind() {

initSet(0);

}

public UnionFind(int \_size) {

initSet(\_size);

}

public void initSet(int \_size) {

pset = new Vector<Integer>(\_size);

for (int i = 0; i < \_size; i++)

pset.add(i);

}

public int findSet(int i) {

if (pset.get(i) == i) return i;

else {

pset.set(i, findSet(pset.get(i)));

return pset.get(i);

}

}

public void unionSet(int i, int j) {

pset.set(findSet(i), findSet(j));

}

public boolean isSameSet(int i, int j) {

return findSet(i) == findSet(j);

}

}

**MST (Minimum Spanning Tree)**

**Time Complexity: O(E \* log E)**

***Kruskal's***

// preprocessing

sort E edges by increasing weight

// main algorithm

UnionFind UF = new UnionFind(V);

int mst\_cost = 0;

for (int i = 0; i < E; i++) {

iii e = EdgeList.get(i);

int u = e.second(), v = e.third(), w = e.first();

if (!UF.isSameSet(u, v)) {

mst\_cost += w;

UF.unionSet(u, v); // link these two vertices

}

}

***Prim's***

// preprocessing, *pq is a priority queue*

taken.addAll(Collections.nCopies(V, false));

taken.set(0, true);

for (int j = 0; j < AdjList.get(0).size(); j++) {

ii v = AdjList.get(0).get(j);

pq.offer(new ii(v.second(), v.first()));

}

// main algorithm

while (!pq.isEmpty()) {

if (!taken.get(pq.peek().second())) {

taken.set(pq.peek().second(), true);

mst\_cost += pq.peek().first();

int u = pq.peek().second();

pq.poll();

for (int j = 0; j < AdjList.get(u).size(); j++) {

ii v = AdjList.get(u).get(j);

if (!taken.get(v.first())) {

pq.offer(new ii(v.second(), v.first()));

}

}

}

else {

pq.poll();

}

}

***SSSP (Single Source Shortest Path)***

***Init\_SSSP(s):***

for each v in V

D[v] = INF

p[v] = -1

D[s] = 0;

***relax(u,v,w\_u\_v):***

if(D[v] > D[u] + w\_u\_v)

D[v] = D[u] + w\_u\_v;

p[v] = u;

**BellmanFord's:**

**Time Complexity: O(V\*E) or say, O(V^3)**

for (int i = 0; i < V - 1; i++)

for (int u = 0; u < V; u++)

for (int j = 0; j < AdjList.get(u).size(); j++) {

ii v = AdjList.get(u).get(j);

relax(u, v.first(), v.second());

}

*// bonus: negative cycle test*

boolean negative\_cycle\_exist = false;

for (int u = 0; u < V; u++)

for (int j = 0; j < AdjList.get(u).size(); j++) {

ii v = AdjList.get(u).get(j);

if (D.get(v.first()) > D.get(u) + v.second())

negative\_cycle\_exist = true;

}

Some Special Cases:

**1. the graph has no negative weight cycle:**

**Dijkstra's - Pseudo Code**

**Time Complexity**: O(V log V + E log V) = O((V + E) log V)

Init\_SSSP(s)

PQ.enqueue((D[s],s))

While PQ is not empty

(d,u) = PQ.dequeue();

if(d == D[u])

for each vertex v adjacent to u

if(D[v] > D[u] + weight(u,v))

D[v] = D[u] + weight(u,v);

PQ.enqueue((D[v],v));

**2. All edges have weight 1**

*Modified-BFS:*

for all v in V:

D[v] = INF;

p[v] = -1;

Q = {s}

D[s] = 0;

while Q is not empty:

u = Q.dequeue();

for all v adjacent to u

if(D[v] = INF)

D[v] = D[u] + 1;

p[v] = u;

Q.enqueue(v)

**3. The weighted graph is a Tree**

Either BFS/DFS can do

**4. The weighted graph is DAG:**

topoSort; then relax according to this order

***SSLP - Single Source Longest Path***

***stretch(u,v,w\_u\_v):***

if D[v] < D[u] + w\_u\_v

D[v] = D[u] + w\_u\_v;

p[v] = u;

*LIS: Longest Increasing Subsequence:*

***LISDP***

if(i == N-1) return 1;

if(memo[i]!=-1) return memo[i];

int ans = 0;

for(int j = i + 1, j<N; j++)

if(x[i] < x[j])

ans = max(ans, LISDP(j)+1);

return (memo[i] = ans);

***APSP - All Pairs Shortest Paths***

on weighted non-negative-cycle graph

***Floyd Warshall's:***

for(k = 0; k < V; k++)

for(i = 0; i < V; i++)

for(j = 0; j < V; j++)

//D[i][j] = min(D[i][j],D[i][k] + D[k][j]);

if (D[i][k] \* D[k][j] > D[i][j]) {

D[i][j] = D[i][k] \* D[k][j];

p[i][j] = p[k][j]

}

**backTrack(u,v)**

if(p[u][v]!=u) backTrack(u,p[u][v]);

output p[u][v]