## Graph Theory

### Dijkstra’s Algorithm:

priority\_queue< pi, vector<pi> , greater<pi> > pq;

vector<ll> dist(n+1,intmax);

dist[1] = 0;

pq.push(mp(0,1));

parent[1] = -1;

while(!pq.empty()){

ll u = pq.top().ss;

pq.pop();

FOR0(i,v[u].size()){

ll wt = v[u][i].ss;

ll vertex = v[u][i].ff;

if (dist[vertex]>dist[u]+wt){

dist[vertex] = dist[u] + wt;

parent[vertex] = u;

pq.push(mp(dist[vertex],vertex));

}

}

}

### Floyd Warshall(All pair)

for (k = 0; k < V; k++)  
 for (i = 0; i < V; i++)  
 for (j = 0; j < V; j++)  
 if (dist[i][k] + dist[k][j] < dist[i][j])  
 dist[i][j] = dist[i][k] + dist[k][j];

### Bellman-Ford(for negative edges):

void BellmanFord(struct Graph\* graph, LL src)

{

LL V = graph->V;

LL E = graph->E;

LL dist[V];

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

dist[i] = INT\_MAX;

dist[src] = 0;

for (LL i = 1; i <= V-1; i++)

{

for (LL j = 0; j < E; j++)

{

LL u = graph->edge[j].src;

LL v = graph->edge[j].dest;

LL weight = graph->edge[j].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

dist[v] = dist[u] + weight;

}

}//to check for negative weight cycle, repeat above

} // if shorter path is found, cycle exists

### Prim’s Algorithm for MST

void primMST()

{

priority\_queue< pi, vector<pi> , greater<pi> > pq;

LL src = 0;

vector<LL> key(V, INF);

vector<LL> parent(V, -1);

vector<bool> inMST(V, false);

pq.push(make\_pair(0, src));

key[src] = 0;

while (!pq.empty())

{

LL u = pq.top().second;

pq.pop();

inMST[u] = true; // Include vertex in MST

list< pair<LL, LL> >::iterator i;

for (i = adj[u].begin(); i != adj[u].end(); ++i)

{

LL v = (\*i).first;

LL weight = (\*i).second;

if (inMST[v] == false && key[v] > weight)

{

key[v] = weight;

pq.push(make\_pair(key[v], v));

parent[v] = u;

}

}}}

### LCA:

**Pre-processing: O(nlogn) , Query: O(logn)**

vector <int> tree[MAXN];

int depth[MAXN];

int parent[MAXN][level];

// pre-compute depth for each node and their first parent(2^0th parent)

void dfs(int cur, int prev){

depth[cur] = depth[prev] + 1;

parent[cur][0] = prev;

for (int i=0; i<tree[cur].size(); i++) {

if (tree[cur][i] != prev)

dfs(tree[cur][i], cur);

}

}

void precomputeSparseMatrix(int n){

for (int i=1; i<level; i++){

for (int node = 1; node <= n; node++){

if (parent[node][i-1] != -1)

parent[node][i]=parent[parent[node][i-1]][i-1];

} }}

int lca(int u, int v){

if (depth[v] < depth[u]) swap(u, v);

int diff = depth[v] - depth[u];

for (int i=0; i<level; i++)

if ((diff>>i)&1)

v = parent[v][i];

if (u == v) return u;

for (int i=level-1; i>=0; i--)

if (parent[u][i] != parent[v][i]){

u = parent[u][i];

v = parent[v][i];

}

return parent[u][0];

}

### Topological Sort:

void topologicalSortUtil(LL v, bool visited[],

stack<LL> &Stack)

{

visited[v] = true;

list<LL>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

topologicalSortUtil(\*i, visited, Stack);

Stack.push(v);

}

void topologicalSort()

{

stack<LL> Stack;

bool \*visited = new bool[V];

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

visited[i] = false;

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

if (visited[i] == false)

topologicalSortUtil(i, visited, Stack);

while (Stack.empty() == false)

{

cout << Stack.top() << " ";

Stack.pop();

}

### Strongly Connected Components (Kasuraja’s Algo):

void fillOrder(int v, bool visited[], stack<int> &Stack)

{

visited[v] = true;

list<int>::iterator i;

for(i = adj[v].begin(); i != adj[v].end(); ++i)

if(!visited[\*i])

fillOrder(\*i, visited, Stack);

Stack.push(v);

}

void printSCCs()

{

stack<int> Stack;

bool \*visited = new bool[V];

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

visited[i] = false;

// Fill vertices in stack according to their finishing times

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

if(visited[i] == false)

fillOrder(i, visited, Stack);

Graph gr = getTranspose();

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

visited[i] = false;

while (Stack.empty() == false)

{

// Pop a vertex from stack

int v = Stack.top();

Stack.pop();

if (visited[v] == false)

{

gr.DFSUtil(v, visited);

cout << endl;

}

}}

### Articulation points and Bridges:

**v** : vector used to store adjacency list

**visited** : boolean array to keep track of nodes visited

**disc** : int array to store discovered time of vertex

**low** is int array to which stores, for every vertex v, the discovery time of the earliest discovered vertex to which v or any vertices in the subtree rooted at v is having a back edge. initialized by INFINITY.  
**parent** : int array used to store parent of each node.  
**is** : bool array if ith vertex is an articulation point.

**time** : used to keep track of discovered time.  
**ans** : vector of pair<int ,int> used to store bridges.  
 void dfs(ll x, ll time) {  
 visited[x] = true;  
 disc[x] = low[x] = time+1;  
 ll child = 0;  
 fr(i,v[x].size()) {  
 ll a = v[x][i];  
 if(a==parent[x]) continue;   
 if(visited[a]) low[x] = min(low[x] , disc[a] );  
 else {  
 child++;  
 parent[a] = x;  
 dfs(a,time+1);  
 low[x] = min(low[x], low[a]);  
 if(parent[x]==-1 && child>1)  
 is[x] = true,num++;  
 else if(parent[x]!=-1 && low[a]>=disc[x])  
 is[x] = true,num++;  
 if(low[a]>disc[x])  
 ans.pb(mp(x,a));  
 }} }

### 0-1 BSF:

**You have a graph G with V vertices and E edges. The graph is a weighted graph but the weights can only be 0 or 1. Write an efficient code to calculate shortest path from a given source.**

for all v in vertices:  
 dist[v] = inf  
dist[source] = 0;  
deque d  
d.push\_front(source)  
while d.empty() == false:  
 vertex = get front element and pop as in BFS.  
 for all edges e of form (vertex , u):  
 if travelling e relaxes distance to u:  
 relax dist[u]  
 if e.weight = 1:  
 d.push\_back(u)  
 else:  
 d.push\_front(u)

### Euler path/circuit:

Euler path in undirected graph:

Graph is connected and all vertices have even degree except or 2 have odd degrees.

Euler Circuit in undirected graph:

All vertices have even degree and graph is connected.

Euler circuit in directed graph:

All vertices are a part of a single strongly connected component and indegree and outdegree of all vertices is same,

### Hierholzer’s algorithm for directed graph:

void printCircuit(vector< vector<int> > adj)

{

unordered\_map<int,int> edge\_count;

for (int i=0; i<adj.size(); i++)

{

edge\_count[i] = adj[i].size();

}

if (!adj.size())

return;

stack<int> curr\_path;

vector<int> circuit;

curr\_path.push(0);

int curr\_v = 0;

while (!curr\_path.empty())

{

if (edge\_count[curr\_v])

{

curr\_path.push(curr\_v);

int next\_v = adj[curr\_v].back();

edge\_count[curr\_v]--;

adj[curr\_v].pop\_back();

curr\_v = next\_v;

}

else

{

circuit.push\_back(curr\_v);

curr\_v = curr\_path.top();

curr\_path.pop();

}

}

for (int i=circuit.size()-1; i>=0; i--)

{

cout << circuit[i];

if (i)

cout<<" -> ";

}

}

Bipartite graph: Coloring possible with 2 colors.

### Ford-Fulkerson (Edmond Karp) max flow Algorithm:

O(EV^3)

bool bfs(int rGraph[V][V], int s, int t, int parent[])

{

bool visited[V];

memset(visited, 0, sizeof(visited));

queue <int> q;

q.push(s);

visited[s] = true;

parent[s] = -1;

while (!q.empty())

{

int u = q.front();

q.pop();

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

{

if (visited[v]==false && rGraph[u][v] > 0)

{

q.push(v);

parent[v] = u;

visited[v] = true;

}

}

}

return (visited[t] == true);

}

int fordFulkerson(int graph[V][V], int s, int t)

{

int u, v;

int rGraph[V][V];

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

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

rGraph[u][v] = graph[u][v];

int parent[V];

int max\_flow = 0;

while (bfs(rGraph, s, t, parent))

{

int path\_flow = INT\_MAX;

for (v=t; v!=s; v=parent[v])

{

u = parent[v];

path\_flow = min(path\_flow, rGraph[u][v]);

}

for (v=t; v != s; v=parent[v])

{

u = parent[v];

rGraph[u][v] -= path\_flow;

rGraph[v][u] += path\_flow;

}

max\_flow += path\_flow;

}

return max\_flow;

}

### Dinic’s Algorithm: **O(VE^2)**

const int MAXN = ...;   
const int INF = 1000000000;   
   
int n, c[MAXN][MAXN], f[MAXN][MAXN], s, t, d[MAXN], ptr[MAXN], q[MAXN];  
   
bool bfs() {  
 int qh=0, qt=0;  
 q[qt++] = s;  
 memset (d, -1, n \* sizeof d[0]);  
 d[s] = 0;  
 while (qh < qt) {  
 int v = q[qh++];  
 for (int to=0; to<n; ++to)  
 if (d[to] == -1 && f[v][to] < c[v][to]){  
 q[qt++] = to;  
 d[to] = d[v] + 1;  
 }}  
 return d[t] != -1;  
}  
 int dfs (int v, int flow) {  
 if (!flow) return 0;  
 if (v == t) return flow;  
 for (int & to=ptr[v]; to<n; ++to) {  
 if (d[to] != d[v] + 1) continue;  
 int pushed = dfs (to, min (flow, c[v][to] - f[v][to]));  
 if (pushed) {  
 f[v][to] += pushed;  
 f[to][v] -= pushed;  
 return pushed;  
 }  
 }  
 return 0;  
}   
int dinic()

{  
 int flow = 0;  
 for (;;) {  
 if (!bfs()) break;  
 memset (ptr, 0, n \* sizeof ptr[0]);  
 while (int pushed = dfs (s, INF))  
 flow += pushed;  
 }  
 return flow;  
}

### Maximum Bipartite Matching:

**O(M\*N\*N)**

bool bpm(bool bpGraph[M][N], int u, bool seen[], int matchR[])

{

// Try every job one by one

for (int v = 0; v < N; v++)

{

// If applicant u is interested in job v and v is

// not visited

if (bpGraph[u][v] && !seen[v])

{

seen[v] = true; // Mark v as visited

// If job 'v' is not assigned to an applicant OR

// previously assigned applicant for job v (which is matchR[v])

// has an alternate job available.

// Since v is marked as visited in the above line, matchR[v]

// in the following recursive call will not get job 'v' again

if (matchR[v] < 0 || bpm(bpGraph, matchR[v], seen, matchR))

{

matchR[v] = u;

return true;

}

}

}

return false;

}

int maxBPM(bool bpGraph[M][N])

{

// The value of matchR[i] is the applicant number

// assigned to job i

int matchR[N];

memset(matchR, -1, sizeof(matchR));

int result = 0; // Count of jobs assigned to applicants

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

{

// Mark all jobs as not seen for next applicant.

bool seen[N];

memset(seen, 0, sizeof(seen));

// Find if the applicant 'u' can get a job

if (bpm(bpGraph, u, seen, matchR))

result++;

}

return result;

}