quwsarohi@gmail.com Section: Graph Algorithms Page : 1

**// Breadth First Search**

**// Complexity: O(|V|+|E|) : for Adjacency Matrix**

/\* Some Important Implements up-left, up, u-right, left, right, down-left, down, down-right

/ int path[8][8] (adjacency matrix path), int r[] = {-1, -1, -1, 0, 0, 1, 1, 1}, c[] = {-1, 0, 1, -1, 1, -1, 0, 1}

bool visited[8][8] (adjacency matrix)

char direction[5] = "NESW";

\*/

// To find the second shortest path the edges and nodes which builds the shortest path(s) needs to be cut off

vector<int>parent, G[MAX]; // This comments are for adjacency matrix

void printPath(int u, int source\_node) { // void pathPrinter(int start\_x, int start\_y, int end\_x, int end\_y)

if(u == source\_node) { // if(end\_y == start\_y && end\_x == start\_x) {

printf("%d", u); // printf("%d", path[end\_x][end\_y]);

return; // return;

}

printPath(parent[u], source\_node); // if(path[end\_x][end\_y] == 0) pathPrinter(start\_x, start\_y, end\_x, end\_y+1);

printf(" %d", u); // do this for path\_[end\_x][end\_y] [0, 1, 2, 3] : [up, right, down, left]

} // printf(“%c ”, direction[path[end\_x][end\_y]])

int BFS(int source\_node, int finish\_node, int vertices) {

vector<int>dist(vertices+5, INF); // Contains the distance from source to end point

queue<int>Q; // Make pair<int, int> if adjacency matrix is used

Q.push(source\_node);

parent.resize(vertices+5, -1); // For path printing

while(!Q.empty()) { // This comments are for adjacency matrix

int u = Q.front(); //int \_x = Q.front().first, \_y = Q.front().second;

Q.pop();

if(u == finish\_node) // Remove this line if shortest path to all nodes are needed

return dist[u]; // if(\_x == end\_x && \_y == end\_y) return;

for(int i = 0; i < G[u].size(); i++) { // for(int i = 0; i < total\_points; i++) {

int v = G[u][i]; // int x = \_x + r[i], y = \_y + c[i];

if(dist[v] == INF) { // if(x >= 0 && y >= 0 && !visited[x][y]) {

dist[v] = dist[u] + 1; // visited[x][y] = 1;

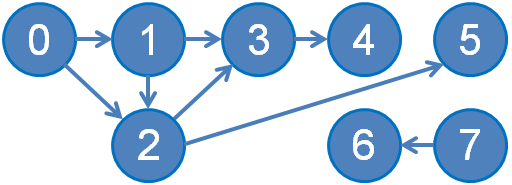
parent[v] = u; // path[x][y] = 1; -------- [0, 1, 2, 3] : [up, right, down, left]

Q.push(v); // Q.push(make\_pair(x, y));

}

}

}

 return -1; //if not found, return -1

}

**// Topological Sort**

**// DFS method**

**// Topological Sorting for a graph is not possible if the graph is not a DAG**

stack<int>topsort; // In this DFS method output is

bool visited[1000]; // 7 6 0 1 2 5 3 4 (remember that there can be >= 1 valid toposort)

// Khans Algorithm output: 0 7 1 2 6 3 5 4 [In-degree wise sorting]

quwsarohi@gmail.com Section: Graph Algorithms Page : 2

void dfs2(int u) {

visited[u] = 1; // Mark the starting node as visited

for(size\_t i = 0; i < G[u].size(); i++) { // For all nodes connected with u (v)

if(visited[G[u][i]] == 0) // if not visited v

dfs2(G[u][i]); // dfs on v

} // Note: paths are saved backwards

topsort.push(u); // push in stack

}

main() { …...

for(int i = 1; i <= E; i++)

if(visited[i] == 0)

dfs2(i);

while(!topsort.empty()) { // Top-sort printing

if(topsort.size() == 1) printf("%d\n", topsort.top()); // If this is the last topological point

else printf("%d ", topsort.top()); // If this is not the last topological point

topsort.pop();

}……

}

**// Topological Sort**

**// Khans Algorithm [Sorts in-degree wise]**

**// Complexity : O(V+E)**

void khansTopsort() {

int indegree[110]; // This algorithm uses in-degree for every node

memset(indegree, 0, sizeof(indegree));

for(int i = 0; i < Edges; i++) { // Calculating in-degree for every node

for(int j = 0; j < G[u].size(); j++)

v = G[u][j], indegree[v]++;

}

priority\_queue<int, vector<int>, greater<int> >pq; //Normally queue can be used

for(int i = Edges-1; i >= 0; i--) {

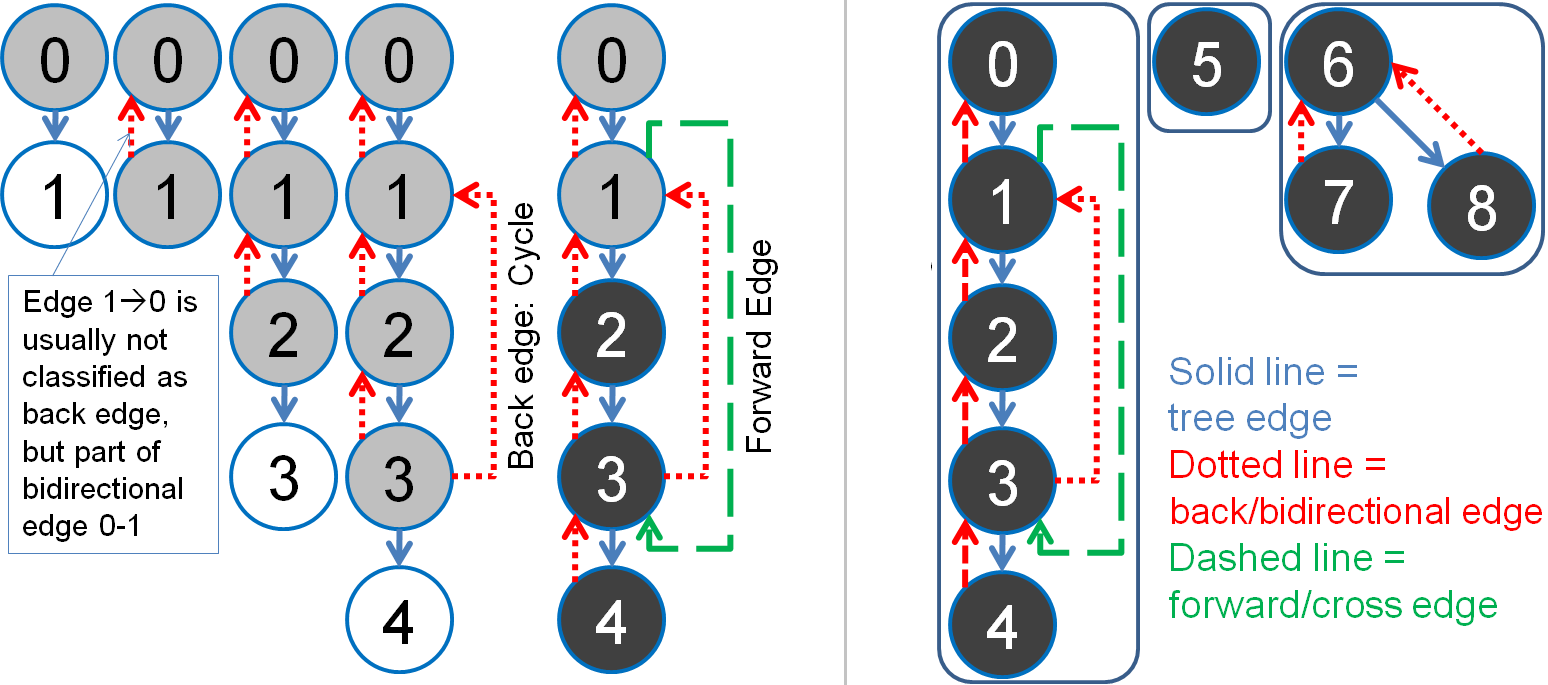
if(indegree[i] == 0)

pq.push(node); // All zero in-degree nodes are inserted in a queue

}

**//int cnt = 0; // To detect if the graph is Acyclic, check below**

vector<int>ans; // This contains the topological answer

 while(!pq.empty()) {

int u = pq.top();

pq.pop();

ans.push\_back(u);

for(int i = 0; i < G[u].size(); i++)

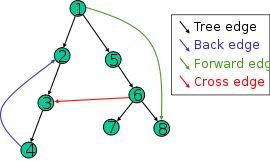
if(--indegree[G[u][i]] == 0)

pq.push(G[u][i]);

//cnt++;

}

quwsarohi@gmail.com Section: Graph Algorithms Page : 3

**//if(cnt != to\_int.size()) //it has no topological order (Acyclic)**

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

if(i == 0) printf(“%d”, ans[i]);

else printf(“ %d”, ans[i]);

}

**// Dijkstra Algorithm (Greedy)**

**// Complexity : O(E logV)**

vector<int>dist, G[MAX], W[MAX];

void printPath(int u) {

if (u == s) { // Extract information from ‘vi p’

printf("%d", s); // Base case, at the source s

return;

}

printPath(p[u]); // Recursive: to make the output format: s -> ... -> t

printf(" %d", u);

}

void dikjstra(int source, int destination, int nodes) {

dist.resize(nodes+1, INF); // dist[v] contains the distance from u to v

dist[source] = 0;

priority\_queue<pair<int, int> > pq; // pq is sorted in ascending(low→hi) order according to weight and edge

pq.push({0, -source});

while(!pq.empty()) {

int u = -pq.top().second;

int wu = -pq.top().first;

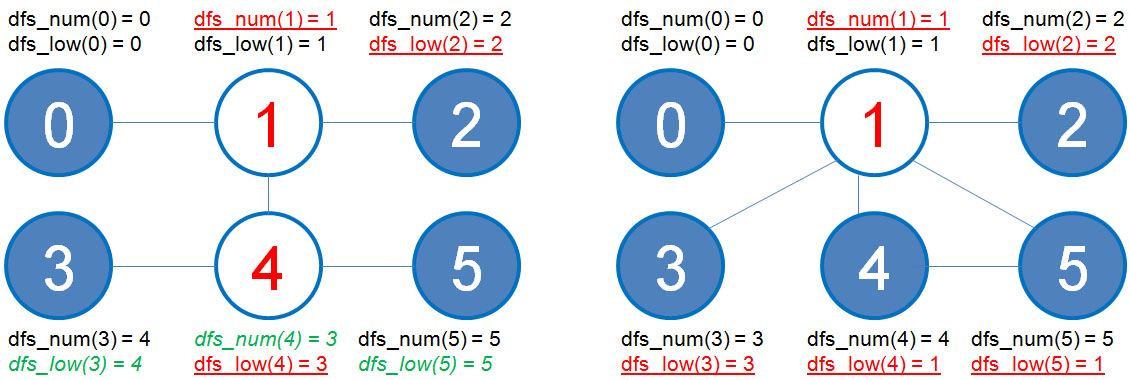
pq.pop();

if(u == destination) return; // If we only need distance of destination, then we may return here

if(wu > dist[u]) continue; // Skipping the longer edges, if we have found shorter edge earlier

for(int i = 0; i < G[u].size(); i++) {

int v = G[u][i];

 int wv = W[u][i];

// Path relaxation

if(wu + wv < dist[v]) {

dist[v] = wu + wv;

pq.push({-dist[v], -v});

}}}}

**// Articulation Point**

**// Complexity O(V+E)**

**// Tarjan, DFS**

Fig: Finding Articulation Point

vector<int>G[101];

int dfs\_num[101], dfs\_low[101], parent[101], isAtriculationPoint[101], dfsCounter, rootChildren, dfsRoot;

// dfs\_num[] : how many times dfs found this node; dfs\_low[] : dfs\_low = the shortest counter on which the node was found

quwsarohi@gmail.com Section: Graph Algorithms Page : 4

void articulationPoint(int u) {

dfs\_low[u] = dfs\_num[u] = ++dfsCounter;

for(int i = 0; i < G[u].size(); i++) {

int v = G[u][i];

if(dfs\_num[v] == 0) {

parent[v] = u;

if(u == dfsRoot) // Special case for root node

rootChildren++; // If root node has child, increment counter

articulationPoint(v);

//1 : if dfs\_num[u] == dfs\_low[v], then it is a back edge

//2 : if dfs\_num[u] < dfs\_low[v], then u is ancestor of v and there is no back edge

//so, if u is not root node, then we can chose u for Articulation Point

if(dfs\_num[u] <= dfs\_low[v] && u != dfsRoot) //Avoiding root node

isArticulationPoint[u]++;

//if there is any child node of u that is a back edge of a previous node

//then the value of dfs\_low[v] might be less than the present dfs\_low[u]

//we try to save the lowest value possible

dfs\_low[u] = min(dfs\_low[v], dfs\_low[u]);

}

//As nodes are bi-directional, avoiding direct child node

//if it is not direct child node, and visited, then there is a back edge

//so we try to decrease the value of dfs\_low[u] with the dfs\_num[v]

//the dfs\_num[v] is less than dfs\_num[u] (as it it a back edge)

else if(parent[u] != v) dfs\_low[u] = min(dfs\_low[u], dfs\_num[v]);

}

}

int main() { …...

dfsCounter = 0;

memset(dfs\_num, 0, sizeof(dfs\_num));

isArticulationPoint.reset();

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

if(dfs\_num[i] == 0) {

dfsCounter = rootChildren = 0;

dfsRoot = i;

articulationPoint(i); Fig: Finding Bridges

isArticulationPoint[i] = (rootChildren > 1);

} **//Important isAtriculationPoint[i] + 1 = number of nodes that is disconnected if the i’th node is deleted**

}

/\*for(int i = 0; i < 101; i++) //Printing Articulation Points

if(isArticulationPoint[i])

printf("%d ", i);

printf("\n");\*/

printf("%d\n", (int)isArticulationPoint.count());

return 0;

….}

quwsarohi@gmail.com Section: Graph Algorithms Page : 5

**// Finding Bridges (Graph)**

**// Complexity : O(V+E)**

vector<int> G[MAX];

vector<pair<int, int> >ans;

int dfs\_num[MAX], dfs\_low[MAX], parent[MAX], dfsCounter;

void bridge(int u) {

//dfs\_num[u] is the dfs counter of u node

//dfs\_low[u] is the minimum dfs counter of u node (it is minimum if a back-edge exists)

dfs\_num[u] = dfs\_low[u] = ++dfsCounter;

for(int i = 0; i < G[u].size(); i++) {

int v = G[u][i];

if(dfs\_num[v] == 0) {

parent[v] = u;

bridge(v);

//if dfs\_num[u] is lower than dfs\_low[v], then there is no back edge on u node

//so u - v can be a bridge

if(dfs\_num[u] < dfs\_low[v])

ans.push\_back(make\_pair(min(u, v), max(u, v)));

//obtainig lower dfs counter (if found) from child nodes

dfs\_low[u] = min(dfs\_low[u], dfs\_low[v]);

}

//if v is not parent of u then it is a back edge

//also dfs\_num[v] must be less than dfs\_low[u]

//so we update it

else if(parent[u] != v)

dfs\_low[u] = min(dfs\_low[u], dfs\_num[v]);

}

}

int main() { ……..

memset(dfs\_num, 0, sizeof(dfs\_num));

dfsCounter = 0;

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

if(dfs\_num[i] == 0)

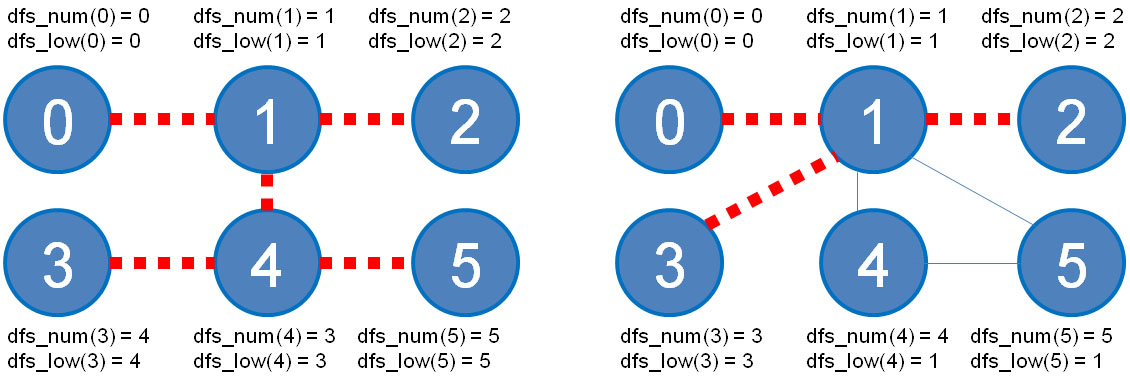
bridge(i);

sort(ans.begin(), ans.end()); //Output

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

printf("%d - %d\n", ans[i].first, ans[i].second);

printf("\n");

return 0;

}

quwsarohi@gmail.com Section: Graph Algorithms Page : 6

**// Floyd Warshal (All Pair Shortest Path)**

**// Complexity : O(V^3) (Use if V <= 400)**

int G[MAX][MAX], parent[MAX][MAX];

void graphINIT() {

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

for(int j = 0; j < MAX; j++)

G[i][j] = INF;

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

G[i][i] = 0;

}

void floydWarshall(int V){

for(int i = 0; i < V; i++) //path printing matrix initialization

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

parent[i][j] = i; //we can go to j from i by only obtaining i (by default)

for(int k = 0; k < V; k++) //Selecting a middle point as k

for(int i = 0; i < V; i++) //Selecting all combination of source (i) and destination (j)

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

if(G[i][k] != INF && G[k][j] != INF) {

//if the graph contains negative edges, then min(INF, INF+ negative edge) = +-INF!

G[i][j] = min(G[i][j], G[i][k]+G[k][j]); //if G[i][i] = negative, then node i is in negative circle

parent[i][j] = parent[k][j]; //if path printing needed

}}

void printPath(int i, int j) {

if(i != j) printPath(i, parent[i][j]);

printf(" %d", j);

}

void minMax(int V) { // Maximum edge weight in minimum distance path

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

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

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

G[i][j] = min(G[i][j], max(G[i][k], G[k][j]));

}

void transitiveClosure(int V) { // Determine if u is connected to v directly or indirectly

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

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

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

G[i][j] |= (G[i][k] & G[k][j]);

**}**

**// Strongly Connected Component (Tarjan)**

**// Complexity : O(V+E)**

vector<int>G[30], SCC;

int dfs\_num[30], dfs\_low[30], dfsCounter, SCC\_no = 1;

quwsarohi@gmail.com Section: Graph Algorithms Page : 7

bitset<30>visited;

void tarjanSSC(int u) {

SCC.push\_back(u); // Generally it is stack data structure, here, it is implemented as vector instead

//visited[u] marks if the node u is usable in a SCC and not used on other SCC

//if visited[u] is false, then it is used in other SCC

visited[u] = 1;

dfs\_num[u] = dfs\_low[u] = ++dfsCounter;

for(int i = 0; i < G[u].size(); i++) { //for all Strongly Connected Component (directed graph), dfs\_low[u] is same

int v = G[u][i];

if(dfs\_num[v] == 0) { //if it is not visited yet, backtrack it

tarjanSSC(v);

}

if(visited[v]) //if node v (visited[v]) is not visited, we can use it to minimize the dfs\_low[u] value from dfs\_low[v]

dfs\_low[u] = min(dfs\_low[u], dfs\_low[v]);

}

if(dfs\_low[u] == dfs\_num[u]) {

bool first = 1; //in a SCC the first node of the SCC, node u is the first node in a SCC if dfs\_low[u] == dfs\_low[v]

printf("SCC %d\n", SCC\_no++);

while(1) { //as we implementing stack like data structure, the nodes from top to u are on the same SCC

int v = SCC.back();

SCC.pop\_back();

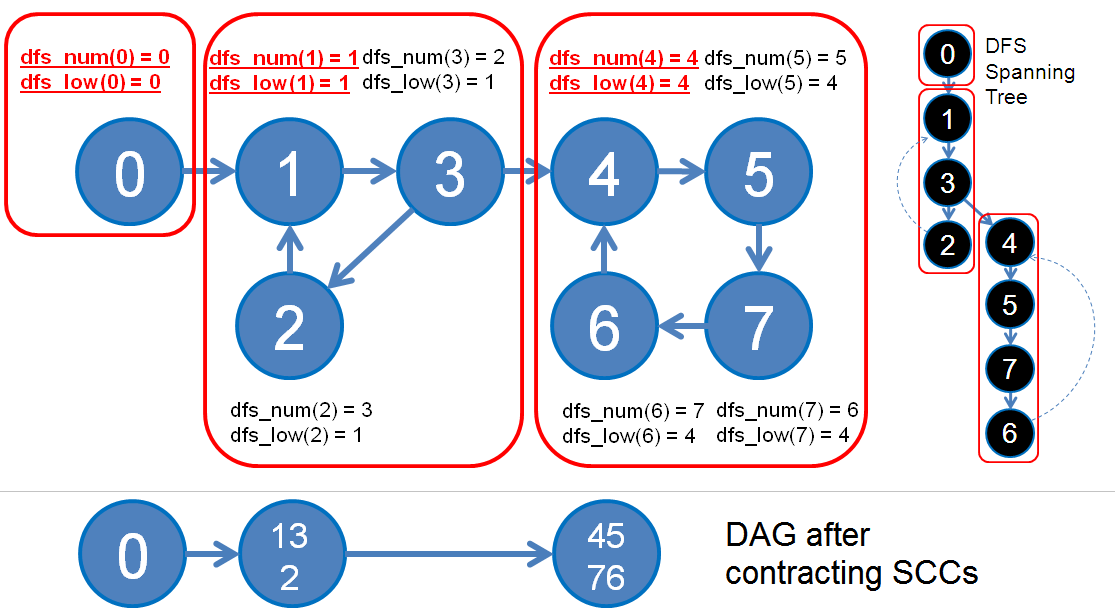
visited[v] = 0; //node v is used, so marking it as false, so that the ancestor nodes

printf("%d\n", v); //doesn't use this node to update it's value

if(u == v)

break;

}

 printf("\n");

}}

int main() { ………...

memset(dfs\_num, 0, sizeof(dfs\_num));

dfsCounter = 0;

visited.reset();

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

if(dfs\_num[i] == 0)

tarjanSSC(i);

**}**

**// Bipartite Graph**

bool bipatite(int n) {

queue<int>q; Fig: Strongly Connected Component

q.push(n), visited[n] = 1;

while(!q.empty()) {

int u = q.front();

for(unsigned int i = 0; i < mat[u].size(); i++) {

if(visited[mat[u][i]] == -1) {

if(visited[u] == 1) visited[mat[u][i]] = 2;

quwsarohi@gmail.com Section: Graph Algorithms Page : 8

else visited[mat[u][i]] = 1;

q.push(mat[u][i]);

}

if(visited[u] == visited[mat[u][i]])

return false;

}

q.pop();

}

return true;

}

**// Bellman Ford’s Algorithm for Single Source Shortest Path (Negative Cycle)**

**// Complexity : O(VE)**

vector<int>G[MAX], W[MAX];

int V, E, dist[MAX];

void bellmanFord() {

for(int i = 0; i <= V; i++) //set to ( -INF ) if **max distance** is needed

dist[i] = INF;

for(int i = 0; i < V-1; i++) //relax all edges V-1 times

for(int u = 0; u < V; u++) //all the nodes

for(int j = 0; j < (int)G[u].size(); j++) {

int v = G[u][i];

int w = W[u][i];

// Relax edges

if(dist[u] != INF) //if there is a negative weight, then INF + negative weight < INF and INF becomes +-INF

dist[v] = min(dist[v], dist[u]+w); //set to max if **max distance** needed

}}

bool hasNegativeCycle() {

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

for(int i = 0; i < G[u].size(); i++) {

int v = G[u][i];

int w = W[u][i];

//if bellmanFord is run for **max distance**, then this code will return **true for positive cycle** by adding this line

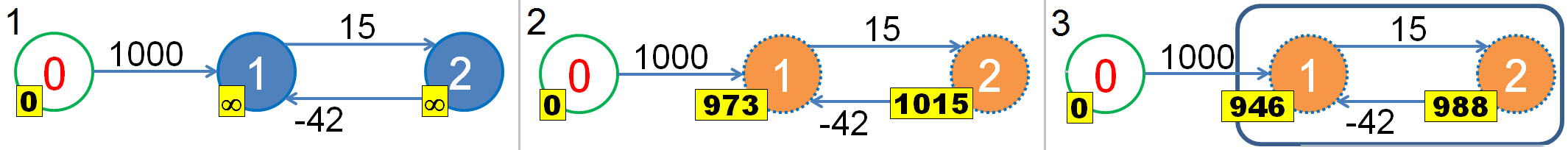
//if(dist[v] < dist[u] + w)

if(dist[v] > dist[u] + w)

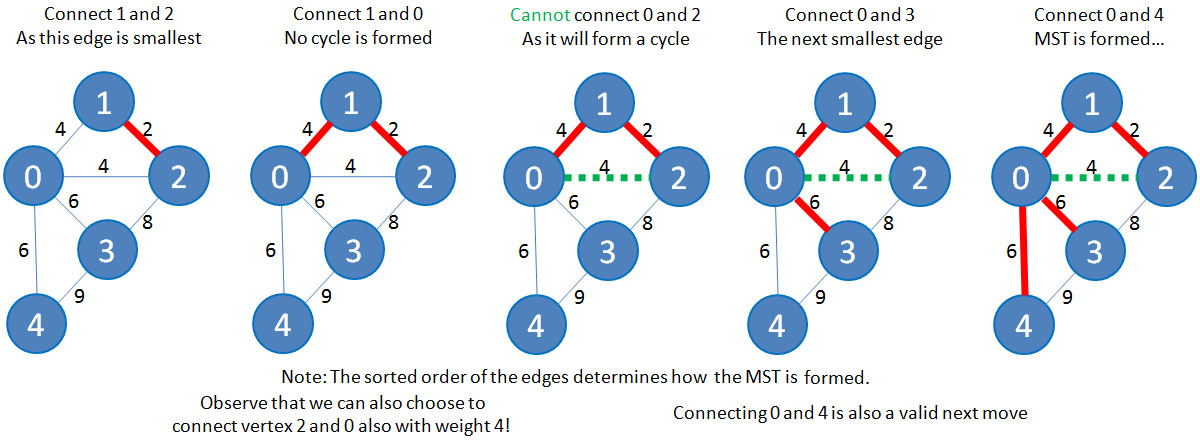
return 1;

}

return 0; }

Fig : Single Source Shortest Path (Negative Cycle)

quwsarohi@gmail.com Section: Graph Algorithms Page : 9

**// Minimum Spanning Tree (Kruskal)**

**//** NOTE: IMPLEMENT UNION DISJOINT FUNCTIONS FIRST

vector<pair<int, pair<int, int> > > Edge;

int main() {

int V, E, u, v, w;

scanf("%d %d", &V, &E);

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

scanf("%d %d %d", &u, &v, &w);

Edge.push\_back(make\_pair(w, make\_pair(u, v))); Fig: Animation of Kruskal’s Algorithm

}

sort(Edge.begin(), Edge.end()); // Sort according to weight min to max

int mst\_cost = 0, selected\_edge = 0;

unionInit(V); // My union disjoint set initialization

for(int i = 0; i < E && selected\_edge < V; i++) {

u = Edge[i].second.first;

v = Edge[i].second.second;

w = Edge[i].first;

if(!isSameSet(u, v)) {

selected\_edge++;

mst\_cost += w;

makeUnion(u, v);

}}

printf("MST in Kruskal : %d\n", mst\_cost);

Edge.clear();

return 0; }

**// Minimum Spanning Tree (Prim’s)**

**// Complexity : O(E logV)**

vector<int> G[MAX], W[MAX];

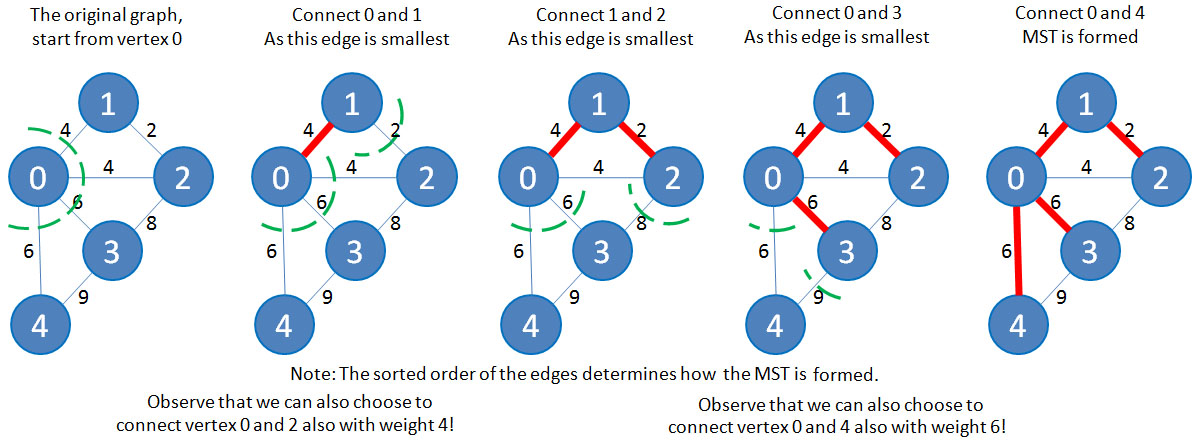
priority\_queue<pair<int, int> >pq;

bitset<MAX>taken; //priority queue returns the minimum node first, if tie, then the first node

void process(int u) {

taken[u] = 1; //mark this node as taken

for(int i = 0; i < G[u].size(); i++) { //get all the edges of this node on a priority queue

 int v = G[u][i];

int w = W[u][i];

if(!taken[v])

pq.push(make\_pair(-w, -v));

} }

int main() {

int V, E, u, v, w;

scanf("%d %d", &V, &E);

while(E--) {

quwsarohi@gmail.com Section: Graph Algorithms Page : 10

scanf("%d %d %d", &u, &v, &w);

G[u].push\_back(v);

W[u].push\_back(w);

G[v].push\_back(u);

W[v].push\_back(w);

}

taken.reset(); //Main Prim's MST code

process(0); //taking 0 node as default

int mst\_cost = 0;

while(!pq.empty()) {

w = -pq.top().first;

v = -pq.top().second;

pq.pop();

if(!taken[v]) { //if the node is not taken, then use this node

mst\_cost += w; //as it contains the minimum edge

process(v);

} }

printf("Prim's MST cost : %d\n", mst\_cost);

return 0; }

**// Dikjstra State-Space Graph (UVa)**

int dikjstra(int start\_node, int end\_node, int gas\_capacity) {

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

for(int j = 0; j <= 100; j++)

cost[i][j] = INF;

priority\_queue<pair<int, pair<int, int> > >pq; //total\_cost, gas, node

cost[start\_node][0] = 0; //node, gas

pq.push(make\_pair(0, make\_pair(0, -start\_node))); //at starting city, cost and gas is zero

while(!pq.empty()) {

int cost\_u = -pq.top().first;

int gas = -pq.top().second.first;

int u = -pq.top().second.second;

pq.pop();

if(u == end\_node)

return cost\_u;

if(cost[u][gas] < cost\_u)

continue;

if(gas < gas\_capacity) { //taking 1 gallon of gas if possible

int new\_cost = cost\_u + price[u];

if(new\_cost < cost[u][gas+1]) {

cost[u][gas+1] = new\_cost;

pq.push(make\_pair(-new\_cost, make\_pair(-(gas+1), -u)));

}}

for(int i = 0; i < G[u].size(); i++) {

int v = G[u][i], w = W[u][i];

if(w <= gas) {

int gas\_left = gas - w;

if(cost\_u < cost[v][gas\_left]) {

cost[v][gas\_left] = cost\_u;

pq.push(make\_pair(-cost\_u, make\_pair(-gas\_left, -v)));

} } } } return -1; }