**Graph**

**Center for Distributed Computing, Jadavpur University**

**Graph - Definitions**

❖ A graph G is represented by a tuple (V, E) where V is

a set edges of vertices E=(e

1

, V=(v

e

2

, e

3

, e

1

4 , ,...) v

2

, v

3

, v

4

,...) and E is a set of

❖ An So element of E, E ⊆ V X V

say e i

is a pair of vertices (v

m

, v

n

).

❖ If directed e

i = vertex. (v

We m one; ,v

call n ) v

is m such is an the ordered graphs start vertex as pair, Digraphs.

then and the graph is a

v

n

is the end

❖ If said e i

= to (v

be m

, adjacent.

v

n

) is an edge of G, then v

m

and v

n

are

**Center for Distributed Computing, Jadavpur University**

**Graph – Definitions ...**

❖The number of edges incident on a vertex is said to be degree. In a digraph, in-degree and out-degree are differentiated.

❖A graph G’ = (V’, E’) is called a subgraph of

G = (V, E) if V’ ⊆ V and E’ ⊆ E.

❖A path from vertex vertices edge for v

i 1

= v 2

v 1 3

...v to k

k-1 , such v 1 to .

v that n

is a sequence of

(v i , v

i+1

) is an

❖Paths may have cycles and edges may have

associated weights.

**Center for Distributed Computing, Jadavpur University**

**Examples**

v1

v2

v3 v4

v5 v6 v7 v8

v9

**Center for Distributed Computing, Jadavpur University**

v1 40

v2

v4 1

v3

10 -5

6 99

0

v5

**Graph Representation – Adjacency Matrix**

• Let G = (V, E) be a graph of n vertices. Its adjacency matrix is an n\*n matrix M consisting of 0s and 1s.

• M[i][j] = 1 iff (v

i

, v j ) is an edge of G = 0 otherwise

• For undirected graphs, adjacency matrix is necessarily symmetric.

• For a weighted graph, M[i][j] = = infinity, weight((v

otherwise

i

, v j )), if (v

i

, v j ) ∈ E

**Center for Distributed Computing, Jadavpur University**

**Adjacency Matrix**

#define maxnovertices 100

typedef struct {

int novertices;

float M[maxnovertices][maxnovertices]; } graph;

**Center for Distributed Computing, Jadavpur University**

**Graph Representation – Adjacency List**

Here a list is maintained for each vertex.

The list for any vertex contains the vertices adjacent to it.

For weighted graphs, additionally, the weight information is stored.

**Center for Distributed Computing, Jadavpur University**

**Adjacency List**

#define maxnovertices 100

typedef struct node {

int to\_vertex; float weight; struct node \* neighbour;

} nodetype; typedef struct {

int novertices; nodetype \* List[maxnovertices];

} graph;

**Center for Distributed Computing, Jadavpur University**

**Adjacency List ...**

Total number of nodes required for a graph (undirected) of n vertices and e edges is n+2e.

For digraph, the number is n+e.

For sparse graph, this representation is efficient.

**Center for Distributed Computing, Jadavpur University**

**Graph Traversal – Depth-First Search**

rec-dfs (v)

{

visit(v); for (each vertex u adjacent to v) if (u is not yet visited)

rec-dfs(u); }

**Center for Distributed Computing, Jadavpur University**

**Iterative Depth-First Search**

iter\_dfs ( graph v){

int i,u1,u2; stack-of-vertices stak; unsigned char visited[maxnovertices];

for (i=0; i< maxnovertices; i++) visited[i] = false;

s\_create(stak); push(v, stak); do{

u1 = pop(stak); if (!visited[u1]) {visit(u1); visited[u1] = true;} for (each u2 adjacent to u1){ if (!(visited[u2])) push(u2, stak);} }while (! empty(stak)); }

**Center for Distributed Computing, Jadavpur University**

**Graph Traversal – Breadth-First Search**

BFS(graph v){ int i,u1,u2 ; queue-of-vertices q; unsigned char visited[maxnovertices];

for (i=0; i< maxnovertices; i++) visited[i] = false; init-q(q); enqueue(v, q);

do{

u1 = dequeue(q); if (!(visited[u1])) {visit(u1); visited[u1] = true;} for (each u2 adjacent to u1){ if (!(visited[u2])) enqueue(u2, q);} }while (!empty\_q(q)); }

**Center for Distributed Computing, Jadavpur University**

**Spanning Tree**

□ A subgraph G’ of G which contains all the vertices of G and is a tree is called a spanning tree of G. There may be more than one spanning trees of a graph.

□ In case of a weighted graph one can assign a cost to a spanning tree, defined as the sum of the weights of its edges.

□ An important problem is to find out the minimum cost spanning tree (MCST) of a graph which may not be unique.

**Center for Distributed Computing, Jadavpur University**

Find the Minimum Cost Spanning Tree

30

12

v1

v4

25 35

v5 v6

3

v2

8 16

20

4

v3

9

**Center for Distributed Computing, Jadavpur University**

**Kruskal’s Algorithm**

MCST-K(graph G){ //G=(V,E) tree T;

init\_t(T); sort E in non-decreasing order according to the weight of the edges;

while(T contains less than (n-1) edges and E is not empty) {

e = edge(u,v) ∈ E having least weight; delete e from E; if (inclusion of e in T does not form a cycle) add e to T; }

if (T contains less than (n-1) edges) error (“no spanning tree exists”) else output T as the MCST; }

Complexity O(|E|log |E|)

**Center for Distributed Computing, Jadavpur University**

**Prim’s Algorithm**

TV = { v

1

}; for (T=Φ; T contains fewer than n-1 edge; add (u,v) to T) {

Let (u,v) be a least-cost edge such that u ∈ TV and !(v ∈ TV); if (there is no such edge) break; add v to TV; } if (T contains fewer than n-1 edges)

print (“no spanning tree exists”);

**Center for Distributed Computing, Jadavpur University**

**Dijkstra’s shortest path algorithm (Greedy)**

**SP-D(G) //Find the shortest paths from v1 to all the vertices {**

**l**

**1**

**\*=0; for (i =2; i <= n; i++} l**

**i**

**=C**

**1i**

**; do{**

**l**

**k**

**= min {l**

**i**

**}; // among all temporary labels l**

**k**

**\* = l**

**k**

**; Let v**

**k**

**be the node which just received a permanent label; for all adjacent vertices of v**

**k**

**having temporary labels do l**

**i**

**= min(l**

**i**

**, l**

**k**

**\*+ C**

**ki**

**);} while there are vertices without permanent labels; }**

**Time Complexity=O(n2)**

**Center for Distributed Computing, Jadavpur University**

1 v2 v3

15 3

10

4

v1 20

v7

v4

4

12

v6

v5 5 9

3

2

**Center for Distributed Computing, Jadavpur University**

**All pair Shortest Paths**

APSP (graph G){ float A[n][n]; for (i = 0; i< n; i++)

for ( j =0; j < n; j++) A[i][j]:=M[i][j]; for ( k = 0; k < n; k++) for (i = 0; i< n, i++)

for ( j =0; j < n; j++) A[i][j] = min(A[i][j], A[i][k] + A[k][j]); }

Time Complexity=O(n3)

**Center for Distributed Computing, Jadavpur University**

**Transitive Closure Matrix**

APSP (graph G){ boolean T[n][n]; for (i = 0; i< n; i++)

for ( j =0; j < n; j++) if (M[i,j] <> ∝) T[i][j] =true else T[i][j] = false; for ( k = 0; k < n; k++) for (i = 0; i< n, i++)

for ( j =0; j < n; j++) T[i][j] = T[i][j] OR ( T[i][k] AND T[k][j]) }

Transitive Closure Matrix – A matrix of boolean values where a true at (i,j)

represents the existence of a path between nodes i and j

**Center for Distributed Computing, Jadavpur University**

**Digraph with cycles and negative weight**

v2 18

v4

-2

6

4

5 v1

8

v3

**Center for Distributed Computing, Jadavpur University**

10

v1 v2 v3 v4

V1 0 ∝ 5 ∝

V2 -2 0 6 18

V3 ∝ 8 0 10

V4 ∝ ∝ 4 0