****

**MOI UNIVERSITY**

**SCHOOL OF BIOLOGICAL AND PHYSICAL SCIENCES**

**COMPUTER SCIENCE**

COM 315: ALGORITHMS

COM/37/15: KIPKEMEI ISAAC KISIERO

TAKE AWAY CAT

LECTURER: MR. BRIAN MWOTIE

DATE: 07/11/2016

SIGN:………………………..

1. **Kruskal’s Algorithm**

This is a minimum spanning algorithm that finds an edge of the least possible weight that connects any two trees in the forest. It is a greedy algorithm in the graph theory because it connects minimum spanning algorithm in graph theory because it connects minimum spanning trees for a connected graph adding increasing cost arcs at each step.The implementation in C++ is as follows:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct Edge{

int src, dest, weight;

};

struct Graph{

int V, E;

struct Edge\* edge;

};

struct Graph\* createGraph(int V, int E){

struct Graph\* graph = (struct Graph\*) malloc( sizeof(struct Graph) );

graph->V = V;

graph->E = E;

graph->edge = (struct Edge\*) malloc( graph->E \* sizeof( struct Edge ) );

return graph;

}

struct subset{

int parent;

int rank;

};

int find(struct subset subsets[], int i){

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

void Union(struct subset subsets[], int x, int y){

int xroot = find(subsets, x);

int yroot = find(subsets, y);

if (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

else if (subsets[xroot].rank > subsets[yroot].rank)

subsets[yroot].parent = xroot;

else {

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

int myComp(const void\* a, const void\* b)

{

struct Edge\* a1 = (struct Edge\*)a;

struct Edge\* b1 = (struct Edge\*)b;

return a1->weight > b1->weight;

}

void KruskalMST(struct Graph\* graph){

int V = graph->V;

struct Edge result[V];

int e = 0;

int i = 0;

qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

struct subset \*subsets =

(struct subset\*) malloc( V \* sizeof(struct subset) );

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

subsets[v].parent = v;

subsets[v].rank = 0;

}

while (e < V - 1){

struct Edge next\_edge = graph->edge[i++];

int x = find(subsets, next\_edge.src);

int y = find(subsets, next\_edge.dest);

if (x != y) {

result[e++] = next\_edge;

Union(subsets, x, y);

}

}

printf("Following are the edges in the constructed MST\n");

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

printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);

return;

}

int main(){

int V = 4;

int E = 5;

struct Graph\* graph = createGraph(V, E);

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = 10;

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 6;

graph->edge[2].src = 0;

graph->edge[2].dest = 3;

graph->edge[2].weight = 5;

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 15;

graph->edge[4].src = 2;

graph->edge[4].dest = 3;

graph->edge[4].weight = 4;

KruskalMST(graph);

return 0;

}

1. **Prim's Minimum Spanning Tree (MST) algorithm.**

This is a gready algorithm that starts with an empty spaning tree. The idea is to maaintain two spanning trees with two sets of vertices not yet included. The first set contains the vertices already included in minimum spanning tree, the other set contains the vertices not yet included. At every step, it considers all the edges tha connect the two sets and pick the minimum weight edges. After picking the edge, it moves to the other endpoints of the set containing the minimum spanning tree.

#include <stdio.h>

#include <limits.h>

#define V 5

int minKey(int key[], bool mstSet[]){

int min = INT\_MAX, min\_index;

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

if (mstSet[v] == false && key[v] < min)

min = key[v], min\_index = v;

return min\_index;

}

int printMST(int parent[], int n, int graph[V][V]){

printf("Edge Weight\n");

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

printf("%d - %d %d \n", parent[i], i, graph[i][parent[i]]);

}

void primMST(int graph[V][V]){

int parent[V];

int key[V];

bool mstSet[V];

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

key[i] = INT\_MAX, mstSet[i] = false;

key[0] = 0;

parent[0] = -1;

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

{

int u = minKey(key, mstSet);

// Add the picked vertex to the MST Set

mstSet[u] = true;

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

if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}

printMST(parent, V, graph);

}

int main(){

int graph[V][V] = {{0, 2, 0, 6, 0},

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

{0, 5, 7, 9, 0},

};

primMST(graph);

return 0;

}

1. **Dijkstra's single source shortest path algorithm.**

This is an algorithm for finding the shortest path between a nd b. it picks the unvisited vertex with the shortest distance, calcutes the distance to each unvisited neighbor and updates the neighbours distance if smaller.

#include <stdio.h>

#include <limits.h>

#define V 9

int minDistance(int dist[], bool sptSet[])

{

int min = INT\_MAX, min\_index;

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

if (sptSet[v] == false && dist[v] <= min)

min = dist[v], min\_index = v;

return min\_index;

}

int printSolution(int dist[], int n)

{

printf("Vertex Distance from Source\n");

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

printf("%d \t\t %d\n", i, dist[i]);

}

void dijkstra(int graph[V][V], int src)

{

int dist[V];

bool sptSet[V];

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

dist[i] = INT\_MAX, sptSet[i] = false;

dist[src] = 0;

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

{

int u = minDistance(dist, sptSet);

sptSet[u] = true;

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

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX

&& dist[u]+graph[u][v] < dist[v])

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

}

printSolution(dist, V);

}

int main(){

int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},

{4, 0, 8, 0, 0, 0, 0, 11, 0},

{0, 8, 0, 7, 0, 4, 0, 0, 2},

{0, 0, 7, 0, 9, 14, 0, 0, 0},

{0, 0, 0, 9, 0, 10, 0, 0, 0},

{0, 0, 4, 14, 10, 0, 2, 0, 0},

{0, 0, 0, 0, 0, 2, 0, 1, 6},

{8, 11, 0, 0, 0, 0, 1, 0, 7},

{0, 0, 2, 0, 0, 0, 6, 7, 0}

};

dijkstra(graph, 0);

return 0;

}