**Assignment - 4**

**1. Write a program to perform Prims algorithm.**

Prim's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a connected, undirected graph. The minimum spanning tree of a graph is a subset of its edges that forms a tree and includes all the vertices of the graph with the minimum possible total edge weight.

Here's a brief overview of how Prim's algorithm works:

1. \*\*Initialization\*\*: Start with an arbitrary vertex as the initial tree. This vertex can be chosen randomly or based on some criteria.

2. \*\*Grow the Tree\*\*: Repeat the following steps until all vertices are included in the tree:

- Find the minimum-weight edge that connects a vertex in the tree to a vertex not yet in the tree.

- Add the selected edge and its connecting vertex to the tree.

3. \*\*Termination\*\*: Stop when all vertices are included in the tree, resulting in a minimum spanning tree.

Prim's algorithm is efficient and typically runs in \(O(V^2)\) time, where \(V\) is the number of vertices in the graph. With the help of priority queues or Fibonacci heaps, its time complexity can be improved to \(O(E + V \log V)\), where \(E\) is the number of edges in the graph.

Overall, Prim's algorithm provides an efficient way to find the minimum spanning tree of a graph, making it useful in various applications such as network design, clustering, and optimization problems.

# include <stdio.h>

# define Vertex 5

# define Max 1000

int minKey(int [Vertex],bool [Vertex]);

void primMST(int [Vertex][Vertex]);

void update\_key\_parent(int,int [Vertex][Vertex],int [Vertex],int [Vertex],bool [Vertex]);

void printMST(int [Vertex],int [Vertex][Vertex]);

int main()

{

    int graph[Vertex][Vertex] = { { 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;

}

int minKey(int key[Vertex],bool mst[Vertex])

{

    int min = Max;

    int minIndex=-1;

    for(int v=0; v<Vertex; v++)

    {

        if(mst[v]==false && min>key[v])

        {

            min = key[v];

            minIndex = v;

        }

    }

    return minIndex;

}

void update\_key\_parent(int min,int graph[Vertex][Vertex],int key[Vertex],int parent[Vertex],bool mst[Vertex])

{

    for(int v=0; v<Vertex; v++)

    {

        if(graph[min][v]!=0 && mst[v]==false && graph[min][v]<key[v])

        {

            key[v] = graph[min][v];

            parent[v] = min;

        }

    }

}

void printMST(int parent[Vertex],int graph[Vertex][Vertex])

{

    int least\_distance = 0;

    printf("Edge  : Weight\n");

    for(int v=1; v<Vertex; v++)

    {

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

        least\_distance+=graph[v][parent[v]];

    }

    printf("\n\nMinimun Distance : %d",least\_distance);

}

void primMST(int graph[Vertex][Vertex])

{

    int key[Vertex];

    int parent[Vertex];

    bool mst[Vertex];

    for(int v=0; v<Vertex; v++)

    {

        key[v] = Max;

        parent[v] = false;

    }

    key[0] = 0;

    parent[0] = -1;

    for(int v=0; v<Vertex-1; v++)

    {

        int min = minKey(key,mst);

        mst[min] = true;

        update\_key\_parent(min,graph,key,parent,mst);

    }

    printMST(parent,graph);

}

Output :-

PS D:\DAA> gcc Prim.cpp

PS D:\DAA> ./a.exe

Edge : Weight

0 - 1 : 2

1 - 2 : 3

0 - 3 : 6

1 - 4 : 5

Minimun Distance : 16

PS D:\DAA>

**2. Write a program to perform Krushkal algorithm.**

Kruskal's algorithm is another method used to find the minimum spanning tree (MST) of a connected, undirected graph. Like Prim's algorithm, the MST is a subset of the graph's edges that connects all the vertices with the minimum possible total edge weight.

Here's a simplified explanation of Kruskal's algorithm:

1. \*\*Initialization\*\*: Initially, each vertex of the graph is considered as a separate component.

2. \*\*Sort Edges\*\*: Sort all the edges of the graph in non-decreasing order of their weights.

3. \*\*Merge Components\*\*: Iterate through the sorted edges, and for each edge:

- If adding the edge to the MST does not create a cycle (i.e., the vertices it connects are not already in the same component), add it to the MST and merge the components containing its vertices.

4. \*\*Termination\*\*: Stop when the MST contains \(V - 1\) edges, where \(V\) is the number of vertices in the graph.

Kruskal's algorithm typically runs in \(O(E \log E)\) time, where \(E\) is the number of edges in the graph. The sorting step dominates the time complexity.

Overall, Kruskal's algorithm provides an alternative method to find the minimum spanning tree of a graph. It is particularly useful when the graph is sparse or when the number of edges is significantly smaller than the number of vertices. This algorithm finds applications in various fields such as network design, circuit layout, and logistics optimization.

# include <stdio.h>

# define E 5

void make\_Set(int [],int []);

int find\_Parent(int [],int);

void sort\_Edges(int [E][3]);

void union\_Set(int,int,int [],int []);

void Krushkal(int [E][3]);

int main()

{

    int edges[5][3] = { { 0, 1, 10 },

                       { 0, 2, 6 },

                       { 0, 3, 5 },

                       { 1, 3, 15 },

                       { 2, 3, 4 } };

    Krushkal(edges);

    return 0;

}

void make\_Set(int parent[],int rank[])

{

    for(int e=0; e<E; e++)

    {

        parent[e] = e;

        rank[e] = 0;

    }

}

int find\_Parent(int parent[],int v)

{

    if(parent[v]==v)

    {

        return v;

    }

    else

    {

        return find\_Parent(parent,parent[v]);

    }

}

void sort\_Edges(int graph[E][3])

{

    int exchg=0;

    for(int i=0; i<E; i++)

    {

        exchg=0;

        for(int j=0; j<E-i; j++)

        {

            if(graph[j][2]>graph[j+1][2])

            {

                int t=graph[j][2];

                graph[j][2]=graph[j+1][2];

                graph[j+1][2]=t;

                t=graph[j][1];

                graph[j][1]=graph[j+1][1];

                graph[j+1][1]=t;

                t=graph[j][0];

                graph[j][0]=graph[j+1][0];

                graph[j+1][0]=t;

                exchg++;

            }

        }

        if(exchg==0)

        {

            break;

        }

    }

}

void union\_Set(int v1,int v2,int parent[],int rank[])

{

    v1 = find\_Parent(parent,v1);

    v2 = find\_Parent(parent,v2);

    if(rank[v1]>rank[v2])

    {

        parent[v2] = v1;

    }

    else if(rank[v2]>rank[v1])

    {

        parent[v1] = v2;

    }

    else

    {

        parent[v1] = v2;

        rank[v2]++;

    }

}

void Krushkal(int edges[E][3])

{

    int parent[E];

    int rank[E];

    sort\_Edges(edges);

    make\_Set(parent,rank);

    int min\_cost=0;

    printf("Edges in graph\nv1 - v2 = E\n");

    for(int e=0; e<E; e++)

    {

        int v1 = find\_Parent(parent,edges[e][0]);

        int v2 = find\_Parent(parent,edges[e][1]);

        if(v1!=v2)

        {

            union\_Set(v1,v2,parent,rank);

            min\_cost+=edges[e][2];

            printf("\n%d - %d = %d",edges[e][0],edges[e][1],edges[e][2]);

        }

    }

    printf("\n\nMinimun cost of Spanning Tree : %d",min\_cost);

}

Output :-

PS D:\DAA> gcc Krushkal.cpp

PS D:\DAA> ./a.exe

Edges in graph

v1 - v2 = E

2 - 3 = 4

0 - 3 = 5

0 - 1 = 10

Minimun cost of Spanning Tree : 19

PS D:\DAA>

**3. Write a program to perform Depth First Search algorithm.**

Depth-First Search (DFS) is a fundamental graph traversal algorithm used to explore and navigate through the vertices of a graph. It traverses a graph in a depthward motion and aims to visit as far as possible along each branch before backtracking. DFS can be implemented using either iterative or recursive approaches.

Here's a simplified explanation of how DFS works:

1. \*\*Start at a Vertex\*\*: Choose a starting vertex as the initial point of exploration.

2. \*\*Explore Adjacent Vertices\*\*: Visit an adjacent vertex from the current vertex and continue exploring further along the chosen path.

3. \*\*Backtrack if Necessary\*\*: If there are no unvisited vertices from the current vertex, backtrack to the most recent vertex with unexplored neighbors.

4. \*\*Repeat\*\*: Repeat steps 2 and 3 until all vertices in the graph are visited.

DFS can be used to solve various graph-related problems, such as finding connected components, detecting cycles, determining reachability between vertices, and generating topological orderings. It's important to note that DFS does not necessarily visit vertices in any particular order, and the order of traversal depends on the structure of the graph and the specific implementation of the algorithm.

DFS has a time complexity of \(O(V + E)\), where \(V\) is the number of vertices and \(E\) is the number of edges in the graph. However, this complexity can vary based on the data structure used to represent the graph and the specific implementation details.

In summary, DFS is a versatile algorithm that is widely used in graph theory and computer science due to its simplicity and efficiency in traversing and analyzing graphs.

#include <stdio.h>

#define V 5

int stack[V];

int visited[V];

int top = -1;

int adjacency[V][V] =

{{0,1,0,1,0},

{1,0,1,1,1},

{0,1,0,0,1},

{1,1,0,0,1},

{0,1,1,1,0}};

void dfs(int);

int main()

{

    for(int v=0; v<V; v++)

    {

        visited[v] = 0;

    }

    dfs(0);

    return 0;

}

void dfs(int vertex)

{

    visited[vertex] = 1;

    printf("Vertex : %d\n",vertex);

    for(int v=0; v<V; v++)

    {

        if(adjacency[vertex][v] && !visited[v])

        {

            stack[++top] = v;

            visited[v] = 1;

        }

    }

    if(top!=-1)

    {

        dfs(stack[top--]);

    }

}

Output :-

PS D:\DAA> gcc DFS.cpp

PS D:\DAA> ./a.exe

Vertex : 0

Vertex : 3

Vertex : 4

Vertex : 2

Vertex : 1

PS D:\DAA>

**4. Write a program to perform Breadth First Search algorithm.**

**Breadth-First Search (BFS) is a graph traversal algorithm used to explore a graph systematically. It traverses a graph by visiting all the vertices at the current depth level before moving on to the vertices at the next depth level.**

**Here's a simplified overview of BFS:**

**1. \*\*Start at a Vertex\*\*: Begin traversal from a selected starting vertex.**

**2. \*\*Explore Neighbors\*\*: Visit all adjacent unvisited vertices from the current vertex and mark them as visited.**

**3. \*\*Queue\*\*: Enqueue the visited vertices to a queue to keep track of the order in which they were discovered.**

**4. \*\*Dequeue and Repeat\*\*: Dequeue a vertex from the queue and repeat steps 2 and 3 for its unvisited neighbors.**

**5. \*\*Repeat\*\*: Continue dequeuing vertices and exploring their neighbors until the queue is empty.**

**BFS explores vertices in increasing order of their distance from the starting vertex, making it useful for finding shortest paths in unweighted graphs and solving problems like finding connected components and detecting bipartite graphs.**

**The time complexity of BFS is \(O(V + E)\), where \(V\) is the number of vertices and \(E\) is the number of edges in the graph. This complexity can vary depending on the data structure used to represent the graph and the specific implementation.**

**In summary, BFS is a versatile algorithm for traversing graphs and is widely used in various applications such as shortest path finding, network routing, and web crawling.**

#include <stdio.h>

#define V 5

int visited[V];

int queue[V];

int rear=-1;

int front=-1;

int adjacency[V][V] =

{{0,1,0,1,0},

{1,0,1,1,1},

{0,1,0,0,1},

{1,1,0,0,1},

{0,1,1,1,0}};

void bfs(int);

int main()

{

    for(int v=0; v<V; v++)

    {

        visited[v] = 0;

    }

    bfs(0);

    return 0;

}

void bfs(int vertex)

{

    visited[vertex] = 1;

    printf("Vertex : %d\n",vertex);

    for(int v=0; v<V; v++)

    {

        if(adjacency[vertex][v] && !visited[v])

        {

            queue[++rear] = v;

            visited[v] = 1;

        }

    }

    if(front!=rear)

    {

        bfs(queue[++front]);

    }

}

Output :-

PS D:\DAA> gcc BFS.cpp

PS D:\DAA> ./a.exe

Vertex : 0

Vertex : 1

Vertex : 3

Vertex : 2

Vertex : 4

PS D:\DAA>