# Mini Project

**Date of Performance :** 21/04/23 **Date of Submission:** 28/04/23

**SAP ID: 60004190057** **Name : Junaid Girkar**

**Div: A** **Batch : A2**

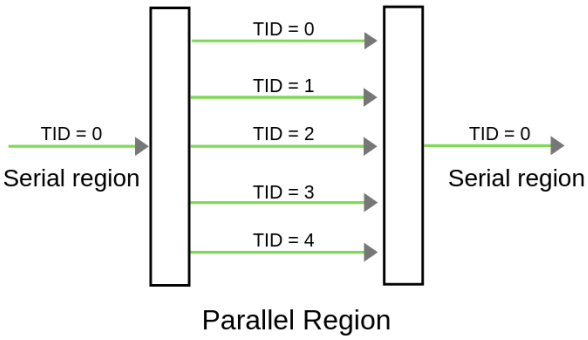
## Aim of Experiment

Implementation of a distributed algorithm to find the Minimum Spanning Tree of an undirected graph.

## Theory / Algorithm / Conceptual Description

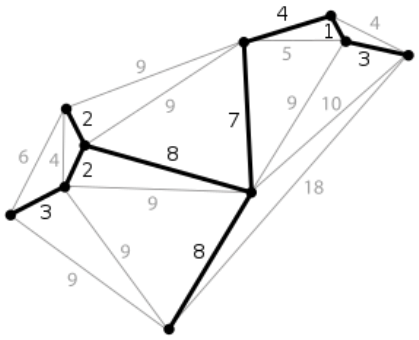
### Parallel Programming

Parallel programming is the use of multiple resources, in this case, processors, to solve a problem. This type of programming takes a problem, breaks it down into a series of smaller steps, delivers instructions, and processors execute the solutions at the same time. It is also a form of programming that offers the same results as concurrent programming but, in less time, and with more efficiency. Many computers, such as laptops and personal desktops, use this programming in their hardware to ensure that tasks are quickly completed in the background.




### Minimum Spanning Tree

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible. More generally, any edge-weighted undirected graph has a minimum spanning forest, which is a union of the minimum spanning trees for its connected components.




Thus, a spanning tree for that graph would be a subset of those paths that have no cycles but still connects every house; there might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost, representing the least expensive path for laying the cable.

**Prim's Algorithm**

Prim's algorithm is a special case of the generic minimum-spanning-tree method. Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph. Prim's algorithm has the property that the edges in the set A always form a single tree. The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. Each step adds to tree A: a light edge that connects A to an isolated vertex - one on which no edge of A is incident. This rule adds only edges that are safe for A; therefore, when the algorithm terminates, the edges in A form a minimum spanning tree. This strategy qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree's weight.

To implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A. In the pseudocode below, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm. During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key attribute. For each vertex $v$, the attribute $v.key$ is the minimum weight of any edge connecting $v$ to a vertex in the tree; by convention, $v.key = \infty$ if there is no such edge. The attribute $v.\pi$ names the parent of $v$ in the tree.

The algorithm implicitly maintains the set A from GENERIC-MST as

$$A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$$

When the algorithm terminates, the min-priority queue Q is empty; the minimum spanning tree A for G is thus,

$$A = \{(v, v.\pi) : v \in V - \{r\}\}$$

*Pseudocode*

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

## Code

A) MST

```
#include<mpi.h>
#include<vector>
#include<iostream>
using namespace std;

int main(int argc, char** argv) {
    int myid, numprocs;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    int n;
    if (myid == 0) {
        //master
        int result = 0;
        n = 9;
        cout << "Number of Nodes 9\n\n";
        int graph[9][9] = {
            {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}
        };

        /*int graph[4][4] = {
            {0,10,6,5},
            {10,0,0,15},
            {6,0,0,4},
            {5,15,4,0}
        };*/
        cout << "Adjacency Matrix\n";
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                cout << graph[i][j] << " ";
            }
            cout << "\n";
        }

        for (int i = 1; i < numprocs; i++) {
            MPI_Send(&n, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
```

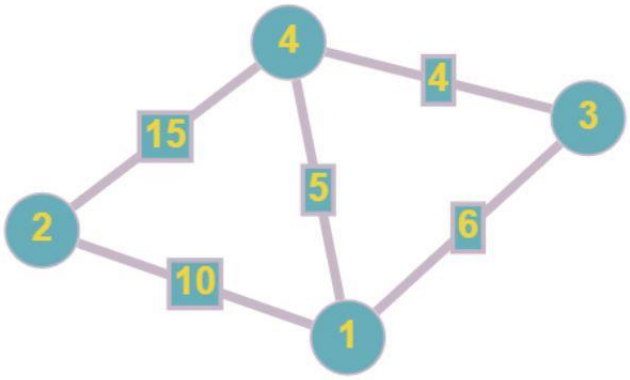
```
            MPI_Send(&graph, n * n, MPI_INT, i, 3, MPI_COMM_WORLD);
        }


        int count = 1;
        int checkVertex = 0;
        int minm = INT_MAX;
        int u, v;
        vector<vector<int> > ans;
        int cost = 0;

        while (count < n) {
            for (int i = 1; i < numprocs; i++) {
                MPI_Send(&checkVertex, 1, MPI_INT, i, 4, MPI_COMM_WORLD);
            }

            int val1, val2, val3;
            for (int i = 1; i < numprocs; i++) {
                MPI_Recv(&val1, 1, MPI_INT, i, 5, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                MPI_Recv(&val2, 1, MPI_INT, i, 6, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                MPI_Recv(&val3, 1, MPI_INT, i, 7, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                if (minm > val1) {
                    minm = val1;
                    u = val2;
                    v = val3;
                }
            }

            count++;
            if (minm == INT_MAX)
                continue;
            cost += minm;
            ans.push_back({ u,v,minm });
            checkVertex = v;
            minm = INT_MAX;

        }
        cout << "\nMinimum Spanning Tree Cost: " << cost;
        cout << "\n\nMinimum Spanning Tree\n";
        for (int i = 0; i < ans.size(); i++)
            cout << "Edge(" << ans[i][0] << "," << ans[i][1] << ") | Weight: "
<< ans[i][2] << endl;

    }
```

```
    else {
        //slave
        MPI_Recv(&n, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        int graph[9][9];
        MPI_Recv(&graph, n * n, MPI_INT, 0, 3, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        int share = (n / (numprocs - 1));
        if (share == 0)
            share = 1;

        int start = (myid - 1) * share;
        int end = start + share - 1;
        if (myid == numprocs - 1)
            end = n - 1;

        vector <pair <int, int > > distance(end - start + 1, { INT_MAX, -1 });
        vector<bool> visited(end - start + 1, 0);
        int checkVertex;

        int mval = INT_MAX;
        int u, v;
        int count = 1;
        while (count < n) {
            MPI_Recv(&checkVertex, 1, MPI_INT, 0, 4, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            if (checkVertex >= start && checkVertex <= end) {
                visited[checkVertex - start] = 1;
            }
            for (int i = 0;i < (end - start + 1);i++) {
                if (visited[i] == 0) {
                    if (graph[checkVertex][start + i] != 0) {
                        int edgeWeight = graph[checkVertex][start + i];
                        if (distance[i].first > edgeWeight) {
                            distance[i].first = edgeWeight;
                            distance[i].second = checkVertex;
                        }
                    }
                    if (mval > distance[i].first) {
                        mval = distance[i].first;
                        u = distance[i].second;
                        v = i + start;
                    }
                }
            }
            MPI_Send(&mval, 1, MPI_INT, 0, 5, MPI_COMM_WORLD);
            MPI_Send(&u, 1, MPI_INT, 0, 6, MPI_COMM_WORLD);
            MPI_Send(&v, 1, MPI_INT, 0, 7, MPI_COMM_WORLD);
```

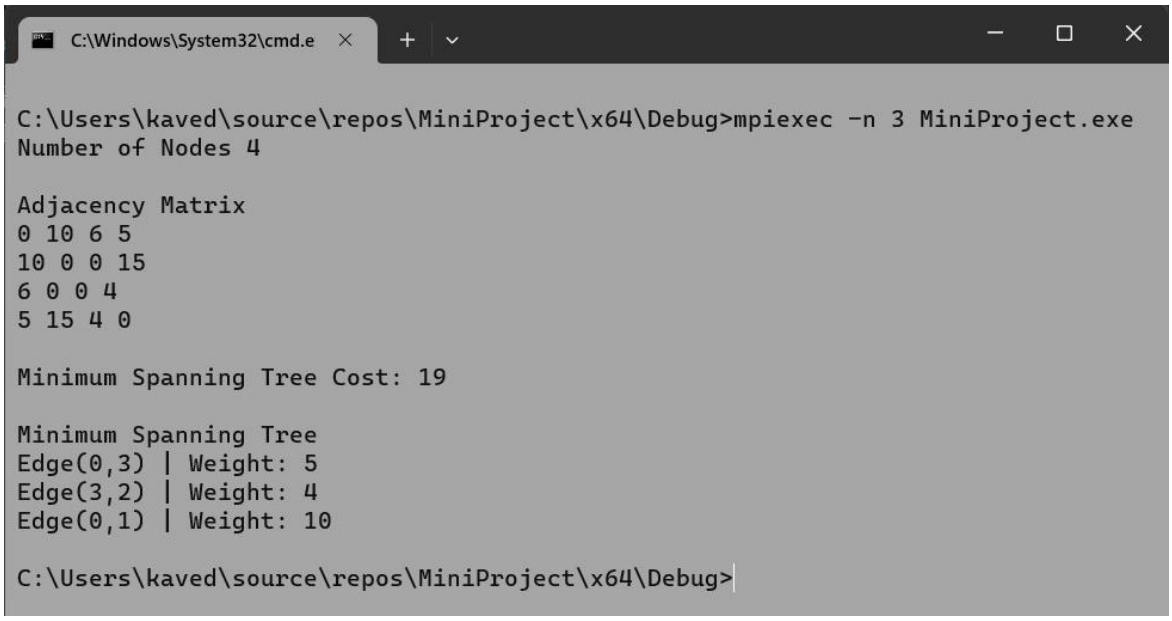
```
            mval = INT_MAX;
            count++;
        }
    }
    MPI_Finalize();
}
```

**Output**

Graph with 4 nodes:

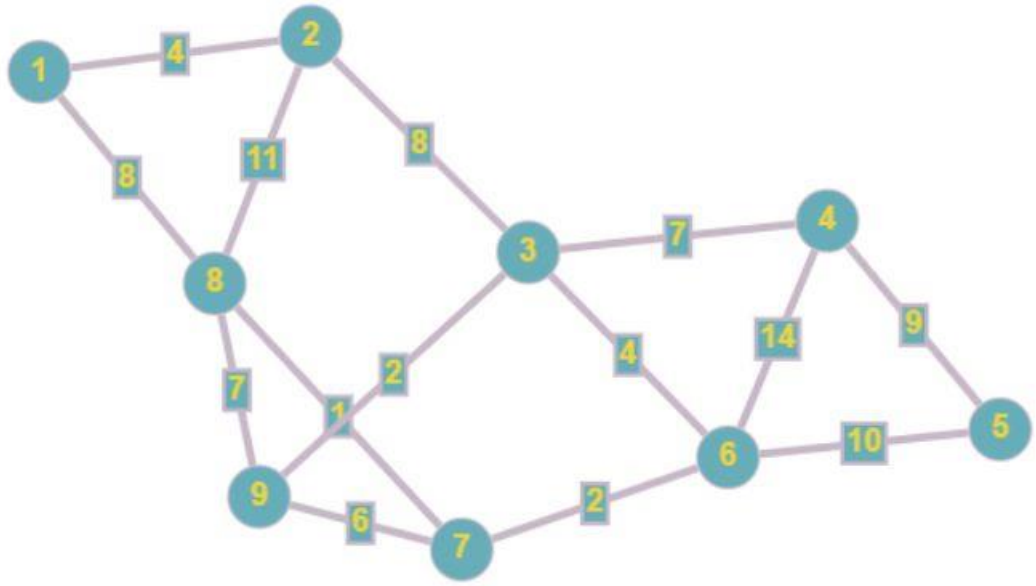



Input Graph




Output MST

Graph with 9 nodes:




Input Graph

```
C:\Windows\System32\cmd.e

C:\Users\kaved\source\repos\MiniProject\x64\Debug>mpiexec -n 4 MiniProject.exe
Number of Nodes 9

Adjacency Matrix
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

Minimum Spanning Tree Cost: 37

Minimum Spanning Tree
Edge(0,1) | Weight: 4
Edge(1,2) | Weight: 8
Edge(2,8) | Weight: 2
Edge(2,5) | Weight: 4
Edge(5,6) | Weight: 2
Edge(6,7) | Weight: 1
Edge(2,3) | Weight: 7
Edge(3,4) | Weight: 9
```

Output MST

**Conclusion**

Minimum Spanning Tree is a graph search problem with a time complexity of $O(n^2)$ with standard computing algorithms such as Prim's algorithm. It evaluates each vertex and computes metrics on each of its neighbors to add an edge to the MST. This step can thus be parallelized for each vertex, and the vertex computation themselves being sequential.

Using MPI, the neighbors can be divided amongst the processes. Each of these nodes performs the computation locally for its set of neighbors and the result is sent back via an aggregator command back to a collecting node. The local minimums are thus combined and the master process calculates the global minimum over all values. This reduces the time complexity to $O(n^2/n\_process)$.