# Integrative Activity 2

José Carlos Martínez Núñez, Tania Sayuri Guizado Hernández, Gerardo Cortés Leal


November 18, 2022

# Contents

## Problem Description

In teams of 3 people maximum, write in C++ a program that will help a company that wants to venture into Internet services by answering problem situation 2.

The program must:

1. Read an input file from standard input (stdin) containing the information of a graph represented in the form of an adjacency matrix with weighted graphs
    - The weight of each edge is the distance in kilometers from neighborhood to neighborhood, where it is feasible to lay cabling.
    - **The program should display the optimal way of wiring fiber optic cables connecting neighborhoods in such a way that information can be shared between any two neighborhoods.**
2. Because cities are just entering the technological world, it requires someone to visit each neighborhood to drop off physical statements, advertisements, printed notices and notifications. So we want to know **what is the shortest possible route that visits each neighborhood exactly once and returns to the neighborhood of origin?**
    - **The program must display the route to be considered, taking into account that the first neighborhood will be called A, the second B, and so on.**
3. The program must also read another square matrix of $N \times N$ data representing the maximum data transmission capacity between neighborhood `i` and neighborhood `j`. Since we are working with cities with numerous electromagnetic fields, which can generate interference, estimates have already been made and are reflected in this matrix.
    - **The company wants to know the maximum flow of information from the initial node to the final node. This should also be displayed in the standard output.**
4. Given the geographic location of several "hubs" to which new homes can be connected, the company wants to have a way to decide, given a new service contract, **which is the closest hub geographically to that new contract**. There is not necessarily one hub for each neighborhood. It is possible to have neighborhoods without a hub, and neighborhoods with more than one hub.

**Input:** An integer $N$ representing the number of neighborhoods in the city. $N \times N$ square matrix representing the graph with the distances in kilometers between the neighborhoods in the city An $N \times N$ square matrix representing the maximum data flow capacities between neighborhood `i` and neighborhood `j` List of $N$ ordered pairs of the form $(A, B)$ representing the location on a coordinate plane of the hubs

### Example:

```
4
0 16 45 32
16  0 18 21
45 18  0  7
32 21  7  0
0 48  12  18
52  0 42 32
18 46  0 56
24 36 52  0
(200,500)
(300,100)
(450,150)
(520,480)
```

### Output:

1. Way of wiring the fiber neighborhoods (list of arcs of the form $(A, B)$)
2. Route to be followed by mail delivery personnel, considering start and end in the same neighborhood.
3. Maximum information flow value from the initial node to the final node.
4. List of polygons (each element is a list of points of the form $(x, y)$).

## Algorithm and Implementation

#### Kruskal's Algorithm

Kruskal's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). The algorithm is as follows:

1. Sort each edge according to its weight in a non-decreasing order.
2. Choose the thinnest edge. Test to see if a cycle can be built using the currently formed spanning tree. Include this edge if a cycle cannot be established. Throw it away if not.
3. Up till the spanning tree has (V-1) edges, repeat step #2.

This algorithm has a time complexity of $O(E \log E)$ or $O(E \log V)$, where $E$ is the number of edges and $V$ is the number of vertices.

For this algorithm, we created three functions:

- `void kruskal(vector<vector<int>> &graph)`: is the main function that calls the other two functions and returns the minimum spanning tree.
- `int find(vector<int> &parent, int x)`: finds the parent of a node.
- `void Union(vector<int> &parent, int x, int y)`: unites two nodes.

The last two functions represent a DSU (Disjoint Set Union) data structure, which is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.

The implementation of the algorithm is as follows:

```
// Kruskal's algorithm
void kruskal(vector<vector<int>> &graph)
{
    vector<int> parent(graph.size());

    int min = 0;
    int count = 0;
    for (int i = 0; i < graph.size(); i++)
    {
        parent[i] = i;
    }

    while (count < graph.size() - 1)
    {
        int min = INT32_MAX, a = -1, b = -1;
        for (int i = 0; i < graph.size(); i++)
        {
            for (int j = 0; j < graph.size(); j++)
            {
                if (find(parent, i) != find(parent, j) && graph[i][j] < min)
                {
                    min = graph[i][j];
                    a = i;
                    b = j;
                }
```

```
            }
        }
        Union(parent, a, b);

        cout << "(" << a << ", " << b << ")" << endl;

        count++;
    }
}
```

```
// Find function
int find(vector<int> &parent, int x)
{
    if (parent[x] == x)
    {
        return x;
    }
    return find(parent, parent[x]);
}
```

```
// Union function
void Union(vector<int> &parent, int x, int y)
{
    int xset = find(parent, x);
    int yset = find(parent, y);
    parent[xset] = yset;
}
```

### Travelling Salesman Problem

The travelling salesman problem (TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?". It is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science. As the number of cities increases, the number of possible routes increases rapidly. For example, there are more than 17 million possible routes through five cities, more than 6 trillion through six cities, and more than 3.6 sextillion through seven cities. Therefore, the

problem is difficult to solve in a reasonable amount of time for numerous cities. This problem can be solved using a brute force approach, but it is not efficient for large inputs. The brute force approach is to generate all possible routes and then select the shortest one. This approach has a time complexity of $O(n!)$, where $n$ is the number of cities.

The approach we used to solve this problem is the nearest neighbor algorithm. This algorithm is a greedy algorithm that starts with a vertex and repeatedly adds the closest vertex to the current vertex to the path. The algorithm is as follows:

1. Start at the first vertex.
2. Find the closest vertex to the current vertex.
3. When all vertices have been visited, return to the first vertex.
4. Print the path.

Our implementation of the algorithm is as follows:

```
// Travelling Salesman Problem
void travelling_salesman(vector<vector<int>> &graph)
{
    vector<bool> visited(graph.size(), false);
    visited[0] = true;

    int value = 0;
    int current = 0;
    int prev = 0;
    cout << "0 -> ";
    for (int i = 1; i < graph.size(); i++)
    {
        for (int j = 0; j < graph.size(); j++)
        {
            if (prev != j && !visited[j] && (graph[prev][j] < value || value == 0))
            {
                value = graph[prev][j];
                current = j;
            }
        }
        visited[current] = true;

        prev = current;
```

```
        cout << current << " -> ";
        value = 0;
    }
    cout << "0";
    cout << endl;
}
```

The time complexity of this algorithm is $O(n^2)$. It is important to note that this algorithm does not always find the optimal solution. It is a heuristic algorithm that finds a good solution in polynomial time.

### Ford-Fulkerson Algorithm

The Ford-Fulkerson algorithm is an algorithm that tackles the max-flow min-cut problem. The max-flow min-cut theorem states that in a flow network, the maximum possible flow from source to sink is equal to the minimum cut capacity from source to sink. The Ford-Fulkerson algorithm uses this theorem to find the maximum possible flow in a network. The algorithm is as follows:

1. Start with initial flow as 0.
2. While there is an augmenting path between the source and the sink, add this path to the flow.
3. Update the residual graph.
4. Return the flow.

This algorithm has a time complexity of $O(E \cdot F)$, where $E$ is the number of edges and $F$ is the maximum flow.

We implemented this algorithm using two functions:

- `bool bfs(vector<vector<int>> &rGraph, vector<int> &parent)`: is a function that finds an augmenting path in the residual graph.
- `void ford_fulkerson(vector<vector<int>> &graph)`: is the main function that calls the `bfs` function and returns the maximum flow.

The implementation of the algorithm is as follows:

```
// BFS
bool bfs(vector<vector<int>> &rGraph, vector<int> &parent)
{
    int V = rGraph.size();
```

```
    vector<bool> visited(V, false);

    queue<int> q;
    q.push(0);
    visited[0] = true;
    parent[0] = -1;

    while (!q.empty())
    {
        int u = q.front();
        q.pop();

        for (int v = 0; v < V; v++)
        {
            if (visited[v] == false && rGraph[u][v] > 0)
            {
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }

    return (visited[V - 1] == true);
}
```

```
void ford_fulkerson(vector<vector<int>> &graph)
{
    int u, v;

    int V = graph.size();

    vector<vector<int>> rGraph(V, vector<int>(V));

    for (u = 0; u < V; u++)
    {
        for (v = 0; v < V; v++)
```

```
        {
            rGraph[u][v] = graph[u][v];
        }
    }

    vector<int> parent(V);

    int max_flow = 0;

    while (bfs(rGraph, parent))
    {
        int path_flow = INT32_MAX;
        for (v = V - 1; v != 0; v = parent[v])
        {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }

        for (v = V - 1; v != 0; v = parent[v])
        {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }

        max_flow += path_flow;
    }

    cout << max_flow << endl;
}
```

### Convex Hull

The convex hull of a set of points is the smallest convex polygon that contains all the points of it. The convex hull problem is the problem of finding the convex hull of a finite set of points in the plane. It is one of the most fundamental problems in computational geometry. The problem is NP-hard, and there are many algorithms that can be used to solve it. The algorithm we used to solve this problem is the

Graham scan algorithm. The algorithm is as follows:

1. Find the point with the lowest y-coordinate. If there is a tie, choose the point with the lowest x-coordinate.
2. Sort the points by polar angle in counterclockwise order around the lowest point.
3. Push the first three points onto a stack.
4. For each remaining point p, do the following:
    1. While the angle formed by the points on the top of the stack, the point next to the top of the stack, and p makes a non-left turn, pop the top point off the stack.
    2. Push p onto the stack.
5. The points on the stack define the convex hull.

This algorithm is a method of finding the convex hull of a finite set of points in the plane with time complexity $O(n \log n)$.

This implementation is the same one as in the Activity 4.2.

```
enum Orientation
{
    COLINEAR,
    CLOCKWISE,
    COUNTERCLOCKWISE
};

struct Point
{
    int x;
    int y;
};

Orientation getOrientation(Point p1, Point p2, Point p3)
{
    int val = (p2.y - p1.y) * (p3.x - p2.x) - (p2.x - p1.x) * (p3.y - p2.y);

    if (val == 0)
        return COLINEAR;
    return (val > 0) ? CLOCKWISE : COUNTERCLOCKWISE;
}
```

```
int distance(Point p1, Point p2)
{
    return (p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y);
}

Point initialPoint;
void convex_hull(vector<Point> &points)
{

    int minY = points[0].y, min = 0;

    for (int i = 1; i < points.size(); i++)
    {
        int y = points[i].y;

        if ((y < minY) || (minY == y && points[i].x < points[min].x))
        {
            minY = points[i].y;
            min = i;
        }
    }

    Point temp = points[0];
    points[0] = points[min];
    points[min] = temp;

    initialPoint = points[0];

    sort(points.begin() + 1, points.end(), [](Point p1, Point p2)
         {
             Orientation orientation = getOrientation(initialPoint, p1, p2);

             if (orientation == COLINEAR)
                 return distance(initialPoint, p1) < distance(initialPoint, p2);
             return orientation == COUNTERCLOCKWISE; });

    vector<Point> hull;
```

```
    hull.push_back(points[0]);
    hull.push_back(points[1]);
    hull.push_back(points[2]);

    for (int i = 3; i < points.size(); i++)
    {
        while (getOrientation(hull[hull.size() - 2], hull[hull.size() - 1], points[i]) !=
        ↪ COUNTERCLOCKWISE)
        {
            hull.pop_back();
        }
        hull.push_back(points[i]);
    }

    for (int i = 0; i < hull.size(); i++)
    {
        cout << "(" << hull[i].x << ", " << hull[i].y << ")" << endl;
    }
}
```

## Test Case

For testing our program we generated two random complete graphs with 100 vertices and 1000 edges. The first graph was used for Kruskal's algorithm and The Travelling Salesman Algorithm. The second one for Ford-Fulkerson's algorithm. After that, we generated 100 random points, and we calculated the convex hull of those points. The complete test case is in the `test.txt` file.

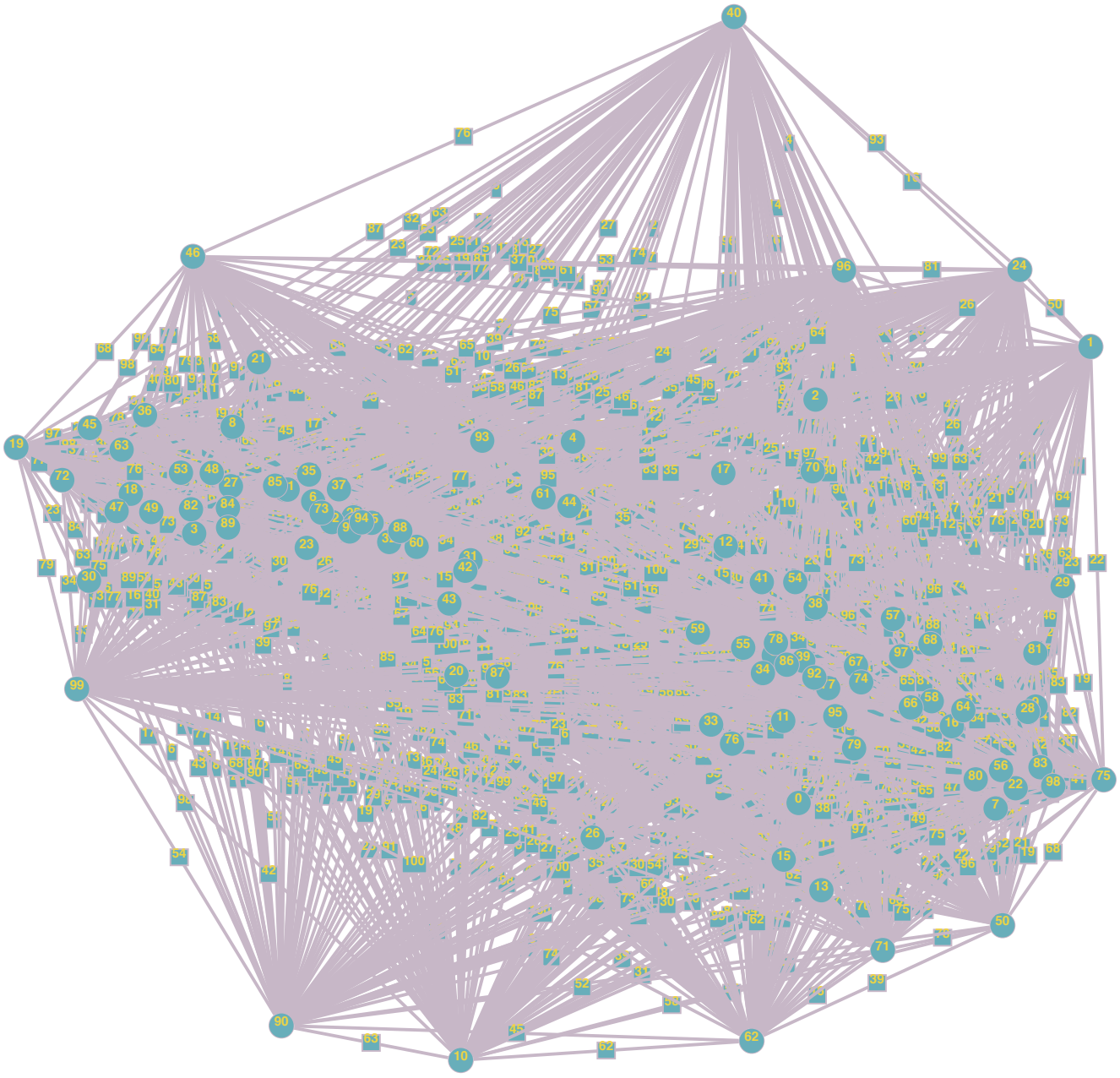

**Figure 1:** First Graph

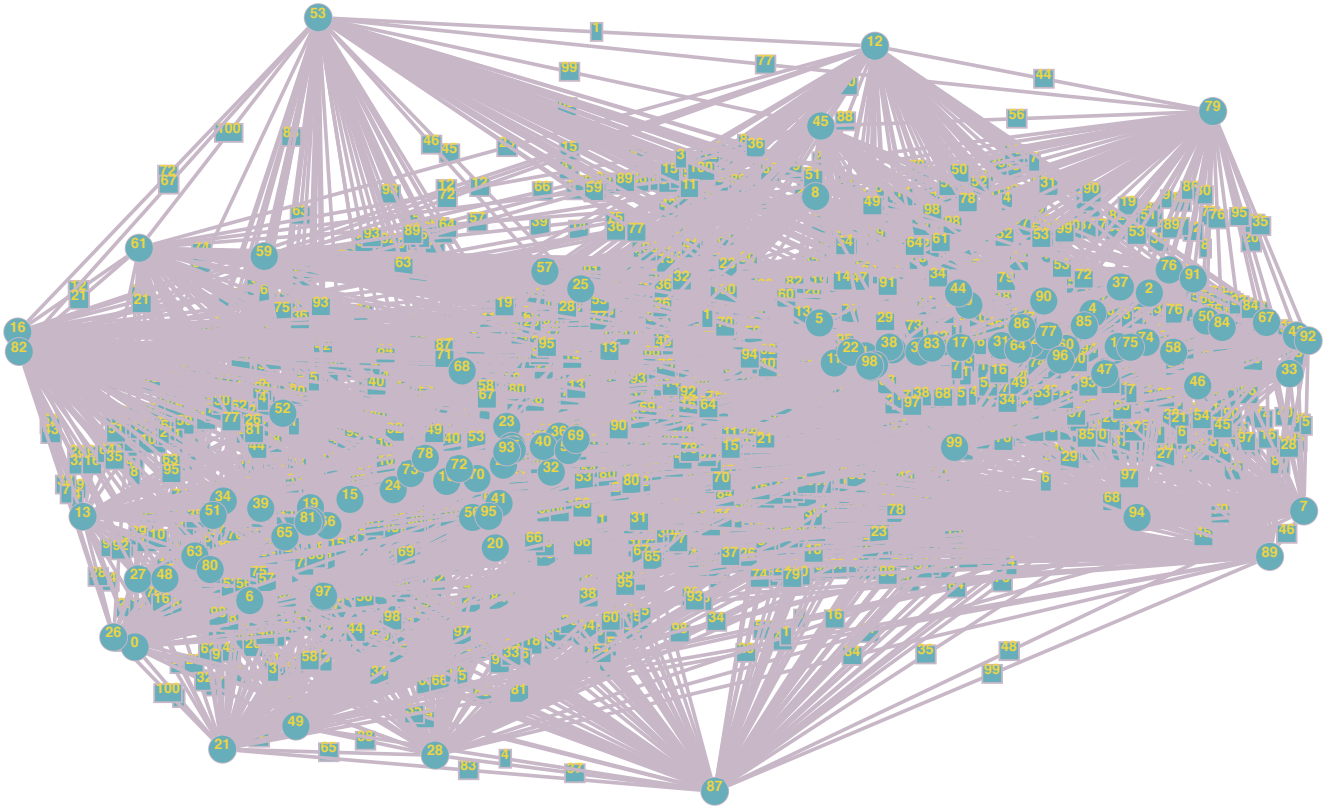

**Figure 2:** Second Graph

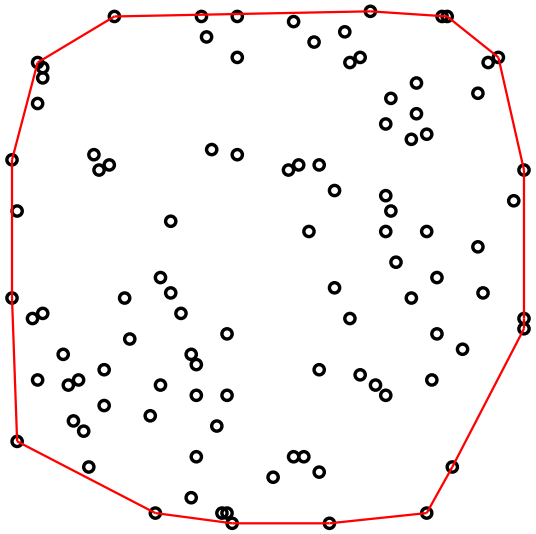

**Figure 3:** Convex Hull

## Results

The results of the test case are in the `results.txt` file.