Integrated Activity 2

Javier Eric Hernández Garza |A01635390

Gildardo Sanchez-Ante

Campus Guadalajara
06 de September del 2023

Analysis and Design of Advanced Algorithms
(Grupo 603)

Kruskal's Algorithm $O(E * \log E)$

```cpp
void kruskal (vector<Edge> edges, int N) {
    int cost = 0;
    vector<Edge> result;
    vector<int> tree_id(N);
    // initialize tree_id
    for (int i = 0; i < N; i++)
        tree_id[i] = i;
    // sort edges
    sort(edges.begin(), edges.end());
    for (Edge e : edges) {
        // if the edge doesn't create a cycle, add it to the result
        if (tree_id[e.u] != tree_id[e.v]) {
            cost += e.weight;
            result.push_back(e);
            int old_id = tree_id[e.u], new_id = tree_id[e.v];
            // update tree_id
            for (int i = 0; i < N; i++) {
                if (tree_id[i] == old_id)
                    tree_id[i] = new_id;
            }
        }
    }
    cout << "The minimum cost is: " << cost << endl;
    cout << "The minimum spanning tree is: " << endl;
    for (Edge e : result) {
        cout << e.u << " " << e.v << endl;
    }
}
```

We used Kruskal's algorithm to generate a minimum spanning tree to generate the optimal way to wire the connecting neighbourhoods.

The algorithm basically chooses the minimum edge between two nodes, unless that edge generates a cycle which means that it joins two nodes in the same tree.

Nearest Neighbour Algorithm (TSP) $O(E * V)$

We also used the Nearest Neighbour algorithm to find the shortest possible route that visits each neighbourhood exactly once and returns to the neighbourhood of origin. This algorithm starts at a specific node and then it chooses all the shortest route to the next node until it has visited every node, then it returns to the first one.

```cpp
void nearestNeighbor(int **matrix,int N){
    int *visited =new int[N];
    for(int i=0;i<N;i++){
        visited[i]=0;
    }
    int current=0;
    visited[current]=1;

    int *path=new int[N];
    path[0]=current;

    int next=0;
    for(int i=1;i<N;i++){
        double min=numeric_limits<double>::max();
        // for each neighbor of current
        for(int j=0;j<N;j++){
            // if neighbor is not visited and distance is less than min
            if(matrix[current][j]!=0 && visited[j]==0){
                if(matrix[current][j]<min){
                    min=matrix[current][j];
                    next=j;
                }
            }
        }
        path[i]=next; // add next to path
        visited[next]=1;
        current=next; // set current to next
    }
```

Ford-Fulkerson Algorithm $O(|V|\, E^2)$

```cpp
int fordFulkerson(std::vector<std::vector<int>>& graph, int s, int t, int N) {
    std::vector<std::vector<int>> rGraph = graph;
    std::vector<int> parent(N, -1);
     int max_flow = 0;
     int u, v;
     // Augment the flow while there is path from source to sink
    while (BFs(rGraph, s, t, parent, N)) {
        int path_flow = INT_MAX;
        // find minimum residual capacity of the edges along the      You, hace 4 días • Kruska
        for (v = t; v != s; v = parent[v]) {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }
        // update residual capacities of the edges and reverse edges along the path
        for (v = t; v != s; v = parent[v]) {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }
        max_flow += path_flow;
    }
    return max_flow;
}
```

We then used Ford-Fulkerson Algorithm to know the maximum information flow from the initial node to the final node. It finds augmenting paths in residual paths setting along flow by the determined amount.

```cpp
//Ford-Fulkerson
bool BFs(std::vector<std::vector<int>>& rGraph, int s, int t, std::vector<int>& parent, int N) {
    bool visited[N];
    memset(visited, 0, sizeof(visited));//set visited to false
    queue <int> q;
    q.push(s);
   visited[s] = true;
   parent[s] = -1;
   while (!q.empty()) {
       int u = q.front();
       q.pop();
       for (int v = 0; v < N; v++) {
           if (visited[v] == false && rGraph[u][v] > 0) {
               q.push(v);
               parent[v] = u;
               visited[v] = true;
           }
       }
   }
   return (visited[t] == true);
}
```

Voronoi Diagrams $\boldsymbol{O(n\ log\ n)}$

Finally, we used a Voronoi Diagram to give the company a tool to decide to which plant new homes can be connected. This algorithm generates perpendicular bisectors between the points so it generates areas for each point.

```
//CGAL - Voronoi Diagram
Triangulation T;
T.insert(plants.begin(), plants.end());

int x,y;
cin>>x>>y;
Point p(x,y);
Point nearest = T.nearest_vertex(p)->point();
cout<<"The nearest exchange is: "<<nearest<<endl;
```