

# Situación problema 2

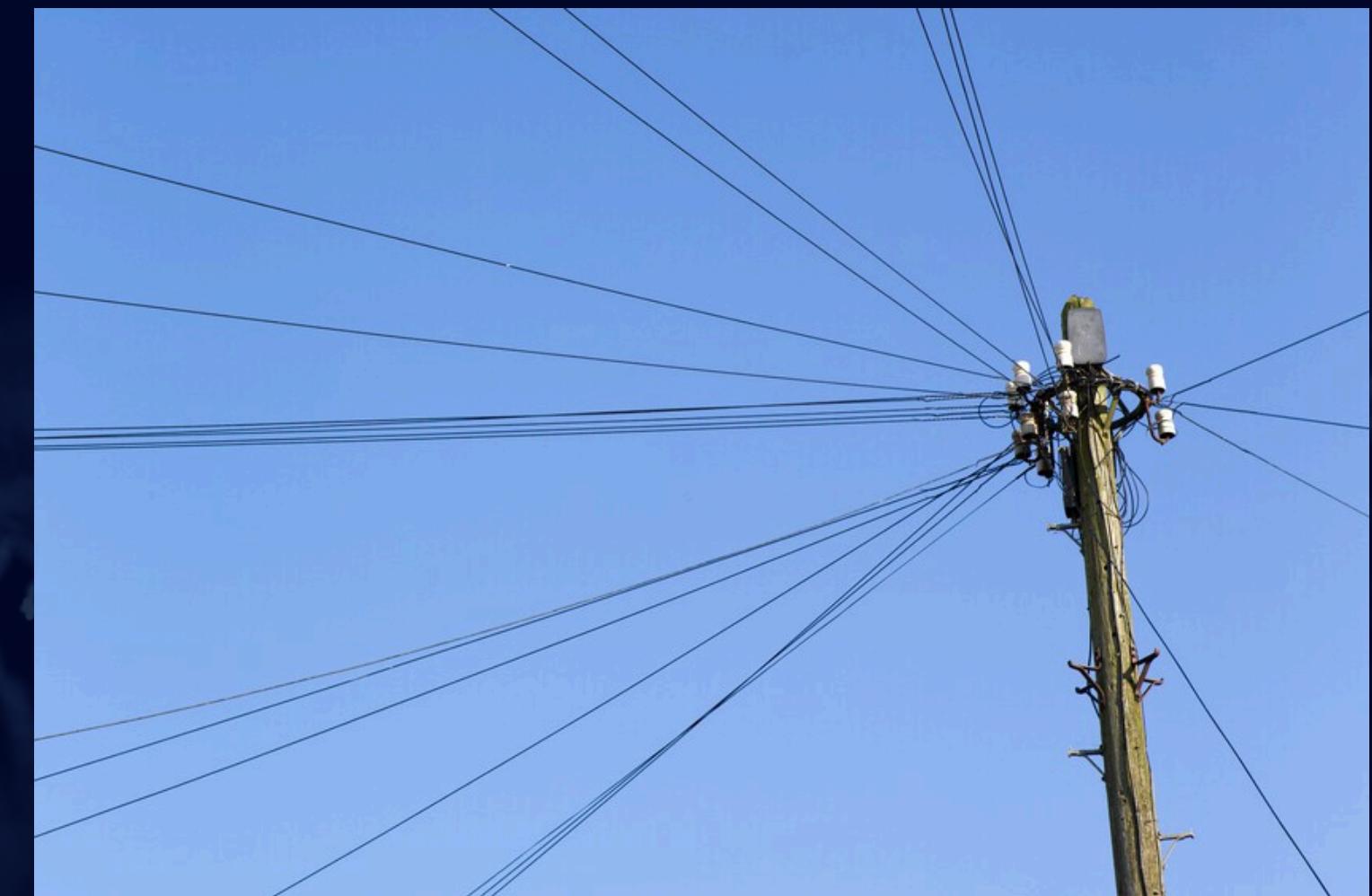
Luisa Fernanda Valdez Guillén - A01711870

Mauricio Olguín Sánchez - A01711522

# Introducción

Esta situación problema aborda cuatro requerimientos prácticos sobre la infraestructura y operación de una ciudad: diseñar el cableado mínimo en fibra óptica entre colonias, calcular la ruta más corta para entregar correspondencia visitando cada colonia una sola vez, estimar el flujo máximo de datos entre el primer y el último nodo dada la capacidad de los enlaces, y determinar qué central existente está geográficamente más cercana a una nueva contratación.

Se implementaron soluciones algorítmicas clásicas, cada una enfocada a resolver de forma directa y eficiente el subproblema correspondiente, aprovechando que la entrada viene en formato de matrices y coordenadas 2D.

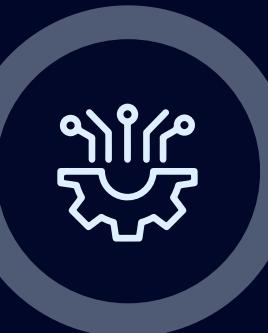
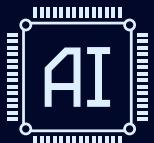


# Algoritmos implementados



## Minimum Spanning Tree – MST

Para conectar todas las colonias minimizando la longitud total de fibra se utilizó el algoritmo de MST. MST construye un árbol que une todos los nodos añadiendo iterativamente la arista de menor peso que conecta un nuevo nodo al árbol actual. Dado que la entrada es una matriz de distancias, la implementación basada en arrays es directa y clara y obtiene el Árbol de Expansión Mínima que reduce el costo total del tendido de fibra.



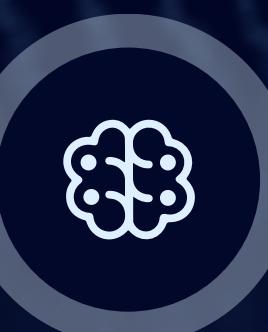
## Fuerza bruta para TSP (Traveling Salesperson Problem)

Para hallar la ruta que visita cada colonia exactamente una vez y retorna al origen se evaluaron todas las permutaciones posibles . Esta aproximación por fuerza bruta garantiza encontrar la solución óptima al TSP cuando “n” es pequeña, ya que calcula el coste de cada ciclo y selecciona el mínimo.



## Máximo flujo (Ford–Fulkerson)

Para calcular el flujo máximo entre el nodo inicial y el final se empleó Ford–Fulkerson, manteniendo una matriz residual y buscando caminos aumentantes mediante DFS. Cada camino encontrado permite incrementar el flujo y actualizar la red residual hasta que no queden caminos..



## KD-Tree (Búsqueda de vecino más cercano en 2D)

Para identificar la central existente más cercana a una nueva contratación se construyó un KD-Tree en 2 dimensiones. El KD-Tree divide el espacio por ejes alternos y permite podar regiones que no pueden contener un vecino más cercano, acelerando la búsqueda respecto a una comprobación exhaustiva. Es una estructura adecuada cuando se hacen consultas espaciales sobre coordenadas (x,y) y amortiza bien el coste cuando hay múltiples centrales.

# MST

```
class MST {
public:
    static void calculateMST(vector<vector<int>>& graph, int n) {
        vector<bool> inMST(n, false);
        vector<int> key(n, INT_MAX);
        vector<int> parent(n, -1);
        key[0] = 0;
        for (int count = 0; count < n - 1; count++) {
            int minKey = INT_MAX, minIndex = -1;
            for (int v = 0; v < n; v++) {
                if (!inMST[v] && key[v] < minKey) {
                    minKey = key[v];
                    minIndex = v;
                }
            }
            int u = minIndex;
            inMST[u] = true;
            for (int v = 0; v < n; v++) {
                if (graph[u][v] && !inMST[v] && graph[u][v] < key[v]) {
                    parent[v] = u;
                    key[v] = graph[u][v];
                }
            }
        }
        cout << "1. Forma de cablear las colonias con fibra:\n";
        for (int i = 1; i < n; i++) {
            char from = 'A' + parent[i];
            char to = 'A' + i;
            cout << "(" << from << ", " << to << ")\n";
        }
        cout << endl;
    }
};
```

```
class TSP {
public:
    static void calculateTSP(vector<vector<int>>& graph, int n) {
        vector<int> cities;
        for (int i = 1; i < n; i++) {
            cities.push_back(i);
        }
        int minCost = INT_MAX;
        vector<int> bestRoute;
        do {
            int currentCost = 0;
            int current = 0;
            bool validRoute = true;
            if (graph[current][cities[0]] == 0) {
                validRoute = false;
            } else {
                currentCost += graph[current][cities[0]];
            }
            for (int i = 0; i < cities.size() - 1 && validRoute; i++) {
                if (graph[cities[i]][cities[i + 1]] == 0) {
                    validRoute = false;
                } else {
                    currentCost += graph[cities[i]][cities[i + 1]];
                }
            }
            if (validRoute && graph[cities[cities.size() - 1]][0] != 0) {
                currentCost += graph[cities[cities.size() - 1]][0];

                if (currentCost < minCost) {
                    minCost = currentCost;
                    bestRoute = cities;
                }
            }
        }

        } while (next_permutation(cities.begin(), cities.end()));
        cout << "2. Ruta a seguir por el personal:\n";
        cout << "A";
        for (int city : bestRoute) {
            cout << " → " << (char)('A' + city);
        }
        cout << " → A\n\n";
    }
};
```

# TSP

# Máximo flujo

```
class MaxFlow {
private:
    static int dfsFind(int u, int t, vector<vector<int>>& res, vector<char>& visited, int f) {
        if (u == t) return f;
        visited[u] = 1;
        int n = (int)res.size();
        for (int v = 0; v < n; ++v) {
            if (!visited[v] && res[u][v] > 0) {
                int can = min(f, res[u][v]);
                int pushed = dfsFind(v, t, res, visited, can);

                if (pushed > 0) {
                    res[u][v] -= pushed;
                    res[v][u] += pushed;
                    return pushed;
                }
            }
        }
        return 0;
    }
public:
    static long long calculateMaxFlow(const vector<vector<int>>& capacity, int n) {
        if (n == 0) return 0;
        vector<vector<int>> residual = capacity;
        int s = 0, t = n - 1;
        long long maxflow = 0;
        while (true) {
            vector<char> visited(n, 0);
            int pushed = dfsFind(s, t, residual, visited, INT_MAX);
            if (pushed == 0) break;
            maxflow += pushed;
        }
        cout << "3. Flujo máximo desde nodo inicial (A) al nodo final (" << char('A' + n - 1) << "):\n";
        cout << maxflow << "\n\n";
        return maxflow;
    }
};
```

# KD-Tree

```
class Nearest {
public:
    static vector<double> calculateNearestAndPrint(const vector<vector<double>>& centers, const vector<double>& query) {
        vector<double> res = { -1.0, 0.0 };
        int m = (int)centers.size();
        if (m == 0) {
            cout << "4. No hay centrales registradas.\n";
            return res;
        }
        vector<Point> pts;
        pts.reserve(m);
        for (int i = 0; i < m; ++i) {
            pts.push_back({centers[i][0], centers[i][1], i});
        }
        KDNode* root = buildKD(pts, 0, m, 0);
        double bestDist2 = INFINITY;
        int bestIdx = -1;
        Point q{query[0], query[1], -1};
        nearestRec(root, q, bestDist2, bestIdx);
        double bestDist = (bestIdx == -1) ? 0.0 : sqrt(bestDist2);
        freeTree(root);
        cout << "4. Distancia entre la nueva central y la central existente mas cercana (km):\n";
        cout.setf(std::ios::fixed); cout.precision(6);
        cout << bestDist << "\n";
        return {(double)bestIdx, bestDist};
    }
}
```

# KD-Tree

```
private:  
    struct Point {  
        double x, y;  
        int idx;  
    };  
    struct KDNode {  
        Point p;  
        KDNode* left;  
        KDNode* right;  
        int axis; // Eje de división (0 para x, 1 para y)  
        KDNode(const Point& pt): p(pt), left(nullptr), right(nullptr), axis(0) {}  
    };
```

```
static bool comparePoints(const Point& a, const Point& b, int axis) {  
    return (axis == 0) ? (a.x < b.x) : (a.y < b.y);  
}  
static void merge(vector<Point>& pts, int l, int m, int r, int axis) {  
    int n1 = m - l + 1;  
    int n2 = r - m;  
    vector<Point> L(n1), R(n2);  
    for (int i = 0; i < n1; i++)  
        L[i] = pts[l + i];  
    for (int j = 0; j < n2; j++)  
        R[j] = pts[m + 1 + j];  
    int i = 0, j = 0, k = l;  
    while (i < n1 && j < n2) {  
        if (comparePoints(L[i], R[j], axis)) {  
            pts[k] = L[i];  
            i++;  
        } else {  
            pts[k] = R[j];  
            j++;  
        }  
        k++;  
    }  
    while (i < n1) {  
        pts[k] = L[i];  
        i++;  
        k++;  
    }  
    while (j < n2) {  
        pts[k] = R[j];  
        j++;  
        k++;  
    }  
}  
static void mergeSort(vector<Point>& pts, int l, int r, int axis) {  
    if (l >= r) return;  
    int m = l + (r - l) / 2;  
    mergeSort(pts, l, m, axis);  
    mergeSort(pts, m + 1, r, axis);  
    merge(pts, l, m, r, axis);  
}
```

# KD-Tree

```
static KDNode* buildKD(vector<Point>& pts, int l, int r, int depth) {
    if (l >= r) return nullptr;

    int axis = depth % 2;
    int m = (l + r) / 2;

    mergeSort(pts, l, r - 1, axis);
    KDNode* node = new KDNode(pts[m]);

    node->axis = axis;
    node->left = buildKD(pts, l, m, depth + 1);
    node->right = buildKD(pts, m + 1, r, depth + 1);
    return node;
}

static void nearestRec(KDNode* node, const Point& q, double &bestDist2, int &bestIdx) {
    if (!node) return;
    double dx = node->p.x - q.x;
    double dy = node->p.y - q.y;
    double d2 = dx*dx + dy*dy;
    if (d2 < bestDist2) {
        bestDist2 = d2;
        bestIdx = node->p.idx;
    }

    int axis = node->axis;
    double diff = (axis == 0) ? (q.x - node->p.x) : (q.y - node->p.y);
    KDNode* first = diff < 0 ? node->left : node->right;
    KDNode* second = diff < 0 ? node->right : node->left;
    if (first) nearestRec(first, q, bestDist2, bestIdx);
    if (second) {
        double diff2 = diff * diff;
        if (diff2 < bestDist2) {
            nearestRec(second, q, bestDist2, bestIdx);
        }
    }
}
```

```
static void freeTree(KDNode* node) {
    if (!node) return;
    if (node->left) freeTree(node->left);
    if (node->right) freeTree(node->right);
    delete node;
}
```

# Casos de prueba

```
[ % ./main.x < input01.txt
```

```
6toSemestre/
```

1. Forma de cablear las colonias con fibra:

(A,B)

(B,C)

(C,D)

2. Ruta a seguir por el personal:

A → B → C → D → A

3. Flujo máximo desde nodo inicial (A) al nodo final (D):

78

4. Distancia entre la nueva central y la central existente mas cercana (km):

103.077641

# Casos de prueba

```
[ % ./main.x < input02.txt                                     6toSemestre/
1. Forma de cablear las colonias con fibra:
(A,B)
(A,C)
(A,D)

2. Ruta a seguir por el personal:
A → B → D → C → A

3. Flujo máximo desde nodo inicial (A) al nodo final (D):
30

4. Distancia entre la nueva central y la central existente mas cercana (km):
5.385165
```

# Casos de prueba

```
[ % ./main.x < input03.txt          5.0s  6toSemestre/
1. Forma de cablear las colonias con fibra:
(C,B)
(E,C)
(A,D)
(D,E)

2. Ruta a seguir por el personal:
A → D → B → C → E → A

3. Flujo máximo desde nodo inicial (A) al nodo final (E):
79

4. Distancia entre la nueva central y la central existente mas cercana (km):
78.746428
```

# Conclusiones

Cada algoritmo fue seleccionado para resolver de forma directa el requisito que representa: MST minimiza el costo total del tendido de fibra; la fuerza bruta para TSP garantiza la ruta óptima para instancias pequeñas; Ford–Fulkerson calcula el flujo máximo a partir de la matriz de capacidades; y KD-Tree identifica eficientemente la central más cercana en el plano. La combinación de estas técnicas entrega una solución integral y verificable para la entrada en formato de matrices y coordenadas.

Para conjuntos de datos mayores conviene contemplar variantes más escalables, pero para tamaños pequeños/medianos las implementaciones actuales priorizan claridad y corrección, cumpliendo los objetivos del problema.



THANK  
YOU