



**TECNOLÓGICO  
DE MONTERREY®**

**Análisis y diseño de algoritmos avanzados**

**AI2 Actividad Integradora 2**

**Profesora:**

Ramona Fuentes Valdez

**Alumno:**

Samir Baidon Pardo A01705403

Angel Francisco Garcia Guzman A01704203

Alejandro Muñoz Shimano A01705550

**Domingo 10 de Noviembre del 2024**

## Problemática

Durante el año 2020 todo el mundo se vio afectado por un evento que nadie esperaba: la pandemia ocasionada por el COVID-19. En todos los países del planeta se tomaron medidas sanitarias para intentar contener la pandemia. Una de estas medidas fue el mandar a toda la población a sus casas, moviendo gran parte de las actividades presenciales a un modelo remoto en el que las empresas proveedoras de servicios de Internet tomaron un papel más que protagonista. Mucha gente se movió a la modalidad de trabajo remoto, o home-office, también la mayoría de instituciones educativas optaron por continuar sus operaciones bajo un modelo a distancia aumentando de gran forma la transmisión de datos en Internet. Continuando muchas de ellas operando de esta manera hasta la fecha. Estamos a cargo de manejar los servicios de Internet de una población pequeña y asegurar Internet estable para todos los habitantes.

## Preguntas Clave

- ¿Podríamos decidir cómo cablear los puntos más importantes de dicha población de tal forma que se utilice la menor cantidad de fibra óptica?
- Para una persona que tiene que ir a visitar todos los puntos de la red, ¿Cuál será la forma óptima de visitar todos los puntos de la red y regresar al punto de origen?
- ¿Podríamos analizar la cantidad máxima de información que puede pasar desde un nodo a otro ?
- ¿Podríamos analizar la factibilidad de conectar a la red un nuevo punto (una nueva localidad) en el mapa ?

## Propuesta de Solución

- Punto 1
  - En el primer punto buscamos las distancias mínimas entre colonias para obtener la forma en la que cableamos las fibras, lo que obtenemos en principio es una matriz cuadrada de  $N \times N$  que representa el grafo con las distancias en kilómetros entre las colonias de la ciudad. Algo de este estilo:  
0 16 45 32  
16 0 18 21  
45 18 0 7  
32 21 7 0

- Para esta primera parte utilizaremos el algoritmo de Floyd-Warshall, esto porque Floyd-Warshall es eficiente para grafos densos y es ideal para encontrar las distancias entre todos los pares de nodos, entonces declaramos el algoritmo dentro de una función para tenerlo accesible.

```
main.cpp

/**
 * Algoritmo de Floyd-Warshall para encontrar distancias mínimas entre todos los pares de nodos.
 *
 * @param N Número de nodos.
 * @param grafo Matriz de adyacencia con las distancias entre nodos.
 */
void floydWarshall(int N, vector<vector<int>>& grafo) {
    for (int k = 0; k < N; ++k) {
        for (int i = 0; i < N; ++i) {
            for (int j = 0; j < N; ++j) {
                if (grafo[i][k] != numeric_limits<int>::max() && grafo[k][j] != numeric_limits<int>::max()) {
                    grafo[i][j] = min(grafo[i][j], grafo[i][k] + grafo[k][j]);
                }
            }
        }
    }
}
```

- Este algoritmo nos dará una complejidad de  $O(n^3)$ , donde  $n$  es el número de nodos. Esto se debe a los tres bucles anidados que tenemos, una vez ya con este código para el algoritmo, nos hace falta otra función para procesar la info e imprimir la forma de cablear las fibras.

```
main.cpp

/**
 * Imprime las distancias mínimas entre colonias calculadas con el algoritmo de Floyd-Warshall.
 *
 * @param N Número de nodos.
 * @param grafo Matriz de distancias entre colonias.
 */
void calcularDistancias(int N, const vector<vector<int>>& grafo) {
    cout << "          Punto 01" << endl;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (i != j) {
                cout << "Colonia " << (i + 1) << " a colonia " << (j + 1) << ": " << grafo[i][j] << endl;
            }
        }
        cout << endl;
    }
}
```

- Para esta segunda parte del punto 1 tenemos una complejidad de  $O(n^2)$ , ya que recorre toda la matriz de distancias. Este paso es necesario para visualizar las distancias mínimas entre colonias y poder conocer así la mejor forma de cableado.
- Punto 2
  - En el segundo punto buscamos ahora la ruta a seguir por el personal que reparte correspondencia, considerando inicio y fin de la misma colonia, para esto utilizaremos la matriz de  $N \times N$  que representa el grafo con las distancias en kilómetros entre las colonias de la ciudad:
 

0	16	45	32
16	0	18	21
45	18	0	7
32	21	7	0
  - Para este punto utilizaremos una técnica similar a la de TSP (Traveling Salesman Problem) y aunque no es una solución exacta para el TSP, es una buena aproximación que funciona en tiempo razonable para grafos de tamaño moderado, aquí utilizamos una Heurística de vecino más cercano, resultando en un algoritmo codicioso en donde nos vamos siempre por la mejor opción sin considerar consecuencias futuras.

```

main.cpp

/**
 * Punto 2: Realizar un recorrido simple (TSP aproximado).
 *
 * @param grafo Matriz de distancias entre las colonias.
 */
void tspAproximado(const vector<vector<int>>& grafo) {
    cout << "          Punto 02" << endl;

    int N = grafo.size();
    vector<int> recorrido;
    vector<bool> visitado(N, false);
    int costoTotal = 0;

    int actual = 0;
    visitado[actual] = true;
    recorrido.push_back(actual + 1);

    for (int i = 1; i < N; i++) {
        int siguiente = -1;
        int minDistancia = numeric_limits<int>::max();

        for (int j = 0; j < N; j++) {
            if (!visitado[j] && grafo[actual][j] < minDistancia) {
                minDistancia = grafo[actual][j];
                siguiente = j;
            }
        }

        if (siguiente != -1) {
            visitado[siguiente] = true;
            costoTotal += minDistancia;
            recorrido.push_back(siguiente + 1);
            actual = siguiente;
        }
    }

    costoTotal += grafo[actual][0];
    recorrido.push_back(1);

    cout << "El recorrido:" << endl;
    for (size_t i = 0; i < recorrido.size(); i++) {
        cout << recorrido[i];
        if (i < recorrido.size() - 1) cout << " → ";
    }
    cout << endl;
    cout << "El costo: " << costoTotal << endl << endl;
}

```

- Este algoritmo nos dará una complejidad de  $O(n^2)$ , donde  $n$  es el número de nodos nuevamente. Esto se debe a que debe buscar el nodo más cercano en cada iteración, en este caso siendo dos ciclos anidados.

- Punto 3

- En el tercer punto buscamos el valor de flujo máximo de información del nodo inicial al nodo final para esto utilizaremos otra matriz de  $N \times N$  que representa las capacidades máximas de flujo de datos entre colonia  $i$  y colonia  $j$ , será de esta forma:

0 48 12 18

52 0 42 32

18 46 0 56

24 36 52 0

- Para este punto utilizaremos el algoritmo de Edmonds-Karp, el cual es una buena opción para calcular el flujo máximo en grafos medianos, y garantiza una solución en tiempo polinomial. Entonces aplicaremos este algoritmo para calcular el flujo máximo entre dos nodos en un grafo de flujo, usando BFS para encontrar caminos aumentantes.

```

main.cpp

/**
 * Implementación del algoritmo Edmonds-Karp para encontrar el flujo máximo.
 *
 * @param capacidades Matriz de capacidades entre nodos.
 * @param fuente Nodo fuente.
 * @param sumidero Nodo sumidero.
 * @return El flujo máximo entre el nodo fuente y el sumidero.
 */
int edmondsKarp(const vector<vector<int>>& capacidades, int fuente, int sumidero) {
    int N = capacidades.size();
    vector<vector<int>> flujo(N, vector<int>(N, 0));
    int flujoTotal = 0;

    while (true) {
        vector<int> padre(N, -1);
        padre[fuente] = fuente;
        queue<pair<int, int>> q;
        q.push({fuente, numeric_limits<int>::max()});

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

            for (int v = 0; v < N; v++) {
                if (padre[v] == -1 && capacidades[u][v] > flujo[u][v]) {
                    padre[v] = u;
                    int nuevoFlujo = min(flujoActual, capacidades[u][v] - flujo[u][v]);
                    if (v == sumidero) {
                        flujoTotal += nuevoFlujo;
                        int cur = v;

                        while (cur != fuente) {
                            int prev = padre[cur];
                            flujo[prev][cur] += nuevoFlujo;
                            flujo[cur][prev] -= nuevoFlujo;
                            cur = prev;
                        }
                        break;
                    }
                    q.push({v, nuevoFlujo});
                }
            }
        }
        if (padre[sumidero] == -1) break;
    }

    return flujoTotal;
}

```

- Este algoritmo nos da una complejidad de  $O(v * e^2)$ , en donde  $v$  es el número de nodos y  $e$  es el número de aristas, esto porque el algoritmo

realiza un BFS en cada iteración, y el número total de iteraciones depende de las aristas del grafo, lo que da como resultado una complejidad cuadrática en las aristas y lineal en los nodos

- Punto 4

- En el cuarto y último punto buscamos la distancia más corta entre la ubicación de la nueva central con respecto a la más cercana, por ende utilizamos una lista de N pares anterior con una coordenada final que representa la ubicación en un plano de la nueva central:

(200,500)

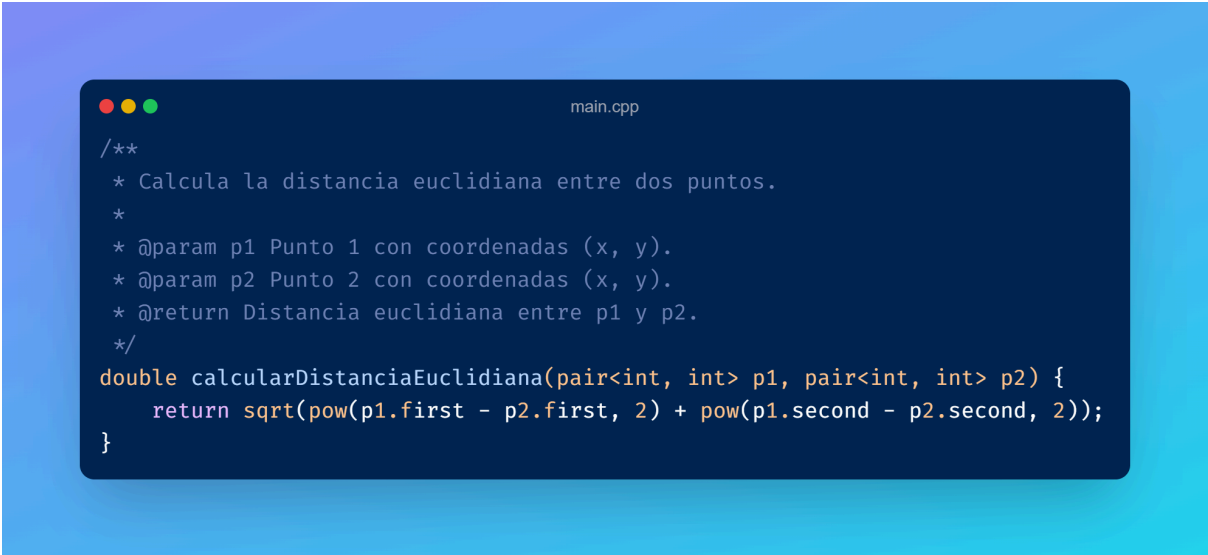
(300,100)

(450,150)

(520,480)

(325,200)

- Para esta primera sección de la última parte utilizaremos la distancia euclidiana, la cual es esencial para calcular la cercanía entre puntos en el espacio, en este caso, entre estaciones. Primero entonces crearemos una función para utilizarla a todo momento



```
main.cpp

/**
 * Calcula la distancia euclidiana entre dos puntos.
 *
 * @param p1 Punto 1 con coordenadas (x, y).
 * @param p2 Punto 2 con coordenadas (x, y).
 * @return Distancia euclidiana entre p1 y p2.
 */
double calcularDistanciaEuclidiana(pair<int, int> p1, pair<int, int> p2) {
    return sqrt(pow(p1.first - p2.first, 2) + pow(p1.second - p2.second, 2));
}
```

- Esta función nos dará una complejidad de  $O(1)$ , ya que ese sólo una fórmula que se ejecuta una sola vez
- Para esta segunda parte del punto 4 tenemos ahora que hacer una búsqueda lineal para por cada coordenada de estación y así sacar cual es la estación más cercana



```

main.cpp

/**
 * Encuentra la central más cercana a una nueva estación.
 *
 * @param posiciones Vector de posiciones de las estaciones.
 * @param nuevaEstacion Coordenadas de la nueva estación.
 */
void calcularMinDistancias(const vector<pair<int, int>>& posiciones, const pair<int, int>& nuevaEstacion) {
    int N = posiciones.size();
    double distanciaMinima = numeric_limits<double>::max();
    int indiceMasCercano = -1;

    for (int i = 0; i < N; i++) {
        double distancia = calcularDistanciaEuclidiana(posiciones[i], nuevaEstacion);
        if (distancia < distanciaMinima) {
            distanciaMinima = distancia;
            indiceMasCercano = i;
        }
    }

    cout << "          Punto 04" << endl;
    cout << "La central más cercana a [" << nuevaEstacion.first << ", " << nuevaEstacion.second << "] es ["
        << posiciones[indiceMasCercano].first << ", " << posiciones[indiceMasCercano].second << "] con una distancia de "
        << distanciaMinima << "." << endl;
}

```

- Para este algoritmo de búsqueda lineal se tiene una complejidad de  $O(n)$  ya que se calcula la distancia Euclidiana por cada coordenada de estación.
- Para todo el código general tenemos una complejidad de  $O(n^3)$  ya que el paso de Floyd-Warshall y el Edmonds-Karp son los que dominan el tiempo de ejecución en función del número de nodos, siendo este el término más costoso.

## Resultados

- Una vez ya creado el código, se utilizaron los archivos de:
  - Equipo\_02\_Entrada\_1.txt
  - Equipo\_02\_Entrada\_2.txt
  - Equipo\_02\_Entrada\_3.txt
- En los resultado podemos apreciar:
  - 1. Forma de cablear las colonias con fibra
  - 2. Ruta a seguir por el personal que reparte correspondencia, considerando inicio y fin en la misma colonia.
  - 3. Valor de flujo máximo de información del nodo inicial al nodo final.
  - 4. Distancia más corta entre dos puntos: el de la ubicación de la nueva central con respecto al más cercano.

- Utilizando de ejemplo el archivo *Equipo\_02\_Entrada\_1.txt*, el cual contiene esta información:

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)  
 (325,200)

- Obtenemos este resultado:

#### *Punto 01*

*Colonia 1 a colonia 2: 16*  
*Colonia 1 a colonia 3: 34*  
*Colonia 1 a colonia 4: 32*

*Colonia 2 a colonia 1: 16*  
*Colonia 2 a colonia 3: 18*  
*Colonia 2 a colonia 4: 21*

*Colonia 3 a colonia 1: 34*  
*Colonia 3 a colonia 2: 18*  
*Colonia 3 a colonia 4: 7*

*Colonia 4 a colonia 1: 32*  
*Colonia 4 a colonia 2: 21*  
*Colonia 4 a colonia 3: 7*

#### *Punto 02*

*El recorrido:*

*1 -> 2 -> 3 -> 4 -> 1*

*El costo: 73*

*Punto 03*

*El flujo máximo: 78*

*Punto 04*

*La central más cercana a [325, 200] es [300, 100] con una distancia de 103.078.*

## **Conclusiones**

Después de realizar esta situación problema, hemos apreciado el poder de los grafos y cómo estos pueden ayudar en no solo problemas de programación, sino en cuestiones del día a día. Los grafos nos sirven de mapas para planear a futuro y entender las consecuencias de nuestras decisiones, comprendiendo cómo es que los elementos se interconectan.

Además de esto pudimos apreciar la importancia de tener algoritmos eficientes para poder tener respuestas claras y rápidas a problemas reales, a comparación de códigos anteriores donde lo primordial era sacar algo, ahora realmente metimos mucho más enfoque en tener algoritmos eficientes y prácticos para al final tener un código más profesional.