ACTIVIDAD TEMA 4

JUAN RONDÁN RAMOS – 48854653P

Subgrupo 3.3 Diciembre 2024

INDICE

LISTADO DE EJERCICIOS

Los ejercicios que he resuelto son los siguientes:

- 402. Búsqueda primero en anchura. (Verde). Envío: 503
- 403. Fuera del laberinto. (Azul). Envío: 862
- 407. Brasilia capital de Brasil. (Azul). Envío: 606
- 413. Operación terraformación. (Amarillo). Envío: 1071

Total: 4 ejercicios resueltos

* Además de los 4 ejercicios resueltos, me gustaría utilizar un comodín (de los dos que tengo) en esta práctica, que equivale a la resolución de un ejercicio amarillo extra. De modo que, la resolución de problemas final quedaría de la siguiente forma: 1 Verde, 2 Azules, 2 Amarillos. Resultando un total de 11 puntos.

RESOLUCIÓN DE PROBLEMAS

402. BÚSQUEDA PRIMERO EN ANCHURA. ENVÍO: 503

El código implementa un algoritmo de búsqueda primero en anchura (BPA) sobre un grafo dirigido representado por una matriz de adyacencia.

En la función leerGrafo, inicialmente, tras leer el número de nodos y aristas, se hace una comprobación para verificar que estos no se salen del rango establecido en el enunciado. Posteriormente se inicializa la matriz de adyacencia G con ceros, indicando que no existen conexiones de nodos al principio. Luego, al leer cada arista representada por dos caracteres c1 y c2 (correspondientes a los nodos origen y destino respectivamente), se actualiza la matriz de adyacencia de modo que G[c1 - 'A'][c2 - 'A'] = true, indicando que existe una arista dirigida desde c1 a c2.

Para la función bpa comenzamos definiendo 3 variables, cola, que es un array que servirá para manejar los nodos por explorar, frente, que indica el índice del nodo que será procesado, comenzando por el inicial e incrementándose a medida que se sacan los nodos de la cola, y fin que representará el índice donde se añadirá el próximo nodo, incrementándose cada vez que se añade un nuevo nodo a la cola. En primer lugar, añadimos el nodo inicial pasado como parámetro a la cola y lo marcamos como visitado en el array global visitado. Posteriormente, mientras haya nodos en la cola (while frente < fin), sacaremos el que este al frente y lo imprimiremos. Además, para cada nodo que se saque de la cola, comprobaremos si tiene nodos adyacentes no visitados, los cuales añadiremos a la cola y marcaremos como visitados.

Para la función busquedaPA, comenzamos inicializando todos los nodos del array visitados a false, es decir, a no visitados. A continuación, se recorre cada nodo y, si no ha sido visitado, se llama a la función bpa para iniciar una búsqueda desde ese nodo.

El código del problema es el siguiente:

```
#include <stdlib.h>  // Funcion exit

#include <string.h>  // Funcion memset

#include <iostream>  // Variables cin y cout

using namespace std;


#define MAX_NODOS 26


int nnodos;                 // Numero de nodos del grafo

int naristas;               // Numero de aristas del grafo

bool G[MAX_NODOS][MAX_NODOS]; // Matriz de adyacencia
```

```
bool visitado[MAX_NODOS];     // Marcas de nodos visitados

// Procedimiento para leer un grafo de la entrada
void leeGrafo (void) {
  cin >> nnodos >> naristas; // Lee número de nodos y aristas
  if (nnodos<0 || nnodos>MAX_NODOS) {
    cerr << "Numero de nodos (" << nnodos << ") no valido\n";
    exit(0);
  }
  memset(G, 0, sizeof(G)); //Inicializa matriz de adyacencia con ceros
  char c1, c2;
  for (int i= 0; i<naristas; i++) {
    cin >> c1 >> c2;    //Lee arista como dos caracteres
    G[c1-'A'][c2-'A']= true;    //Actualiza matriz en función de la arista
  }
}

// Procedimiento iterativo de la búsqueda primero en anchura
//   inicio - primer nodo visitado en la bpa
void bpa(int inicio) {
  int cola[MAX_NODOS];  //Maneja nodos por explorar
  int frente = 0;   // Indice del nodo que será procesado
  int fin = 0;      // Indice donde se añadirá el próximo nodo

  cola[fin++] = inicio; // Añade el nodo inicial a la cola
  visitado[inicio] = true;  //Marca como visitado

  while (frente < fin) {    // Mientras haya nodos en la cola
    int v = cola[frente++]; // Sacar el nodo al frente de la cola
    cout << char(v + 'A');  // Imprimir el nodo (convertir índice a letra)
```

```
    // Añadir a la cola los adyacentes no visitados
    for (int w = 0; w < nnodos; w++) {
      if (G[v][w] && !visitado[w]) {
        cola[fin++] = w;    //añade a la cola
        visitado[w] = true; //marca como visitado
      }
    }
  }
}

// Procedimiento principal de la busqueda en anchura
void busquedaPA (void) {
  memset(visitado, 0, sizeof(visitado)); //Marca todos los nodos como no visitados
  for (int v= 0; v<nnodos; v++) //Recorre todos los nodos
    if (!visitado[v]) //Si el nodo no ha sido visitado
      bpa(v);   //Llamamos a bpa para explorar sus ramas
  cout << endl;
}



int main (void)
{
  int ncasos;
  cin >> ncasos;
  for (int i= 0; i<ncasos; i++) {
    leeGrafo();
    busquedaPA();
  }
}
```

403. FUERDA DEL LABERINTO. ENVÍO: 862

El código implementa un algoritmo de búsqueda primero en profundidad (DFS) sobre un grafo no dirigido representado por listas de adyacencia para cada nodo.

En primer lugar, declaramos cuatro variables globales, numNodos que representa el número de nodos del grafo, la lista L, que manejará las listas de adyacencia para cada nodo, un array booleano visitado, para marcar los nodos visitados, y una cola que manejará el recorrido de Pablito.

Para la función leeGrafo, al igual que en el ejercicio anterior, se lee el número de nodos y se verfica que está dentro del rango establecido. Tras ello, limpiamos la lista de adyacencia de cada nodo para evitar datos residuales. Posteriormente, para cada nodo i, leemos sus nodos adyacentes utilizando la variable aux, los almacenamos en su respectiva lista de adyacencia en base 0 (L[i].push_back(aux-1)) y leemos el siguiente carácter con c1. Mientras c1 sea un espacio y aux2 sea menor o igual a 10, seguiremos leyendo nodos adyacentes y agregándolos a la lista de adyacencia del nodo i.

Para la función buscar, el objetivo es realizar la búsqueda en profundidad para explorar el grafo y encontrar un camino desde el nodo a hasta el nodo de salida. En primer lugar, marcamos el nodo inicial a como visitado y lo agregamos a la cola del recorrido de Pablito. Si este nodo inicial a es el nodo de salida, devolvemos true indicando que hay un camino de salida. Si no se cumple esta condición, recorreremos los nodos adyacentes a “a” y, si no han sido visitados, se hace una llamada recursiva a buscar para explorar esos nodos. Si se encuentra un camino de salida devolvemos true, y si no, añadimos el nodo a la cola del recorrido y seguimos explorando. Si finalmente no se encuentra ningún camino, retorna false.

Para la función listar, que se usará en el main cuando hayamos encontrado una salida del laberinto, su funcionamiento es imprimir los nodos de la cola, es decir, los nodos visitados por Pablito para salir del laberinto. Para cada iteración, imprimimos el nodo índice+1 (recordemos el hecho de pasarlos a base cero con -1 en leeGrafo) y lo eliminamos de la cola. Cuando se vacíe la cola y queden impresos todos los nodos, finalizará el bucle y la función.

Para la función vaciarCola, que se empleara en el main cuando no se ha conseguido salir del laberinto, únicamente se encargará de vaciar la cola.

Finalmente, el main lee el número de casos de prueba y, para cada uno, inicializa los datos del grafo, busca un camino desde el nodo 1 hasta el nodo de salida usando la función buscar(0). Si encuentra el camino, se imprime el recorrido con listar. Si no lo hace, se vacía la cola con vaciarCola y se imprime “infinito”.

El código del ejercicio es el siguiente:

```
#include <stdlib.h>  // exit

#include <string.h>  // memset

#include <iostream>  //  cin y cout

#include <queue>    // cola queue
```

```
#include <list>     // lista adyacencia
using namespace std;

#define MAX_NODOS 20000

int numnodos;           // Numero nodos grafo
list<int> L[MAX_NODOS]; // Listas de adyacencia
bool visitado[MAX_NODOS];     // Marcas visitado
queue<int> cola; // Cola recorrido Pablito



void leeGrafo (void) {
   // Comprobar numnodos dentro de rango
  cin >> numnodos;
  if (numnodos<0 || numnodos>MAX_NODOS) {
   cerr << "Numero de nodos (" << numnodos << ") no valido\n";
   exit(0);
  }

  // Inicialización listas de adyacencia
  for (int i = 0; i < numnodos; i++) {
   L[i].clear();
  }



  char c1;
  int aux=0;
   for (int i= 0; i<numnodos; i++) {
       cin >> aux;
       int aux2=0; //Contador noods adyacentes para el nodo actual
       L[i].push_back(aux-1); // Agrregamos nodo -1 a su lista
```

```
      c1 = cin.get();  // lee siguiente carácter
      aux2++;
      //Ampliar lista adyacencia con nodos vecinos a i
      while(c1==' '&& aux2<=10){
         cin >> aux;
         L[i].push_back(aux-1);
         c1 = cin.get();
         aux2++;
      }
   }
}

// buscar camino desde el nodo 'a' al nodo de salida
bool buscar(int a){
   visitado[a]=true;   //marcamos nodo actual 'a' como visitado
   cola.push(a);       //añadimos 'a' al recorrido de pablito
   if(a==numnodos-1){
      return true;    //Si el nodo inicial es final, hay salida
   }

   // Recorremos nodos adyacentes a nodo actual a
   for (list<int>::iterator it=L[a].begin(); it != L[a].end(); ++it){
      if(!visitado[*it]){     // Si el nodo adyacente no ha sido visitado...
         if(buscar(*it)){    // llamada recursiva a buscar() para explorar nodo adyacente
            return true;  //hay camino a la salida
         }
         cola.push(a);   //si no se encuentra el camino, se añade el nodo "a" a la cola
      }
   }
   return false;  // no hay camino a la salida
}
```

```
void listar(){
   cout<<cola.size()<<endl;    // tamaño cola
   while(!cola.empty()){       // mientras cola no vacia
      cout<<cola.front()+1<<endl; // imprime elemento cola
      cola.pop();              // se elimina
   }
}

void vaciarCola(){
   while(!cola.empty()){
      cola.pop();
   }
}

int main (void) {
  int numcasos;
  cin >> numcasos;
  for (int i= 0; i<numcasos; i++) {
    cout << "Caso " << i+1 << endl;
    memset(visitado, 0, sizeof(visitado));  //Reinicio array
    leeGrafo();
    if(!buscar(0)){            //trata de buscar camino desde el nodo inicial Nodo 1 índice 0 hasta el
nodo de salida nodo N indice N-1
       cout<<"INFINITO"<<endl;        // no salida -> infinito y se vacía la cola
       vaciarCola();
    }else{
       listar();                  // si lo hay, se lista
    }
  }
}
```

407. BRASILIA CAPITAL DE BRASIL. ENVÍO: 606

El código implementa el Algoritmo de Floyd-Warshall para calcular los caminos mínimos entre todos los pares de ciudades en un grafo no dirigido. Este algoritmo es ideal para problemas donde necesitamos obtener las distancias mínimas entre todos los nodos, dado que recorre todas las combinaciones posibles de caminos. Además, analiza las excentricidades de las ciudades para identificar la central.

En primer lugar, se declaran las variables globales: numeroCiudades, que almacena el número de nodos, un vector, ciudades, que contiene los nombres de las ciudades, una matriz C que almacena las distancias directas entre ciudades, una matriz D será usada por el algoritmo para calcular las distancias mínimas entre pares de nodos y excentridadCentro que guarda la excentridad de la ciudad dentral.

Para la función leeGrafo, se inicializa la matriz de distancias C con ceros y se limpia el vector ciudades, evitando datos residuales. A continuación, leemos las ciudades del caso actual, redimensionamos el vector con el tamaño adecuado y, con ayuda de la variable ciudad, vamos leyendo ciudades de la entrada y las vamos almacenando en el vector ciudades. Luego, modificamos la matriz C, inicialmente poniendo todas las celdas en infinito representando que no hay caminos directos entre nodos. Finalmente, iremos creando estos caminos en el último bucle while, leyendo dos nodos con su respectiva distancia, actualizando este valor en ambas direcciones al tratarse de un grafo no dirigido. Los pares de nodos que no estén unidos, quedarán con el infinito establecido la inicialización. Al encontrar una línea 0 0 0, tal y como indica el problema, se finalizará la lectura de caminos entre nodos.

Para la función AlgoritmoFloyd, en primer lugar, se copian los datos de la matriz C a otra matriz D, que se utilizará para almacenar las distancias mínimas entre cualquier par de nodos. Los tres bucles anidados, recorren los nodos k, i, y j del grafo para calcular dichas distancias mínimas. El primer bucle itera sobre cada nodo k, que es considerado como un nodo intermedio entre i y j. El segundo recorre todos los nodos i y el tercero los nodos j (siendo i y j los dos nodos a calcular la distancia mínima). En cada iteración, se verifica si el camino de i a j pasando por k es más corto que el camino directo entre ellos, y, si es así, se actualiza la matriz D[i][j]. La simetría de las distancias se asegura automáticamente debido a que el grafo es no dirigido, lo que implica que D[i][j] es igual a D[j][i]. De este modo, la matriz D contendrá la distancia más corta entre cualquier par de nodos, mientras que la C contendrá la distancia directa para nodos que estén conectados.

Para la función centro, encontramos tres variables locales, capital, que almacena el índice de la ciudad central, longitudMax que almacenará la excentricidad de cada ciudad y un array de enteros excentricidades que almacena la excentricidad de cada ciudad. Seguidamente, para cada ciudad, se recorren todas las distancias a todas las ciudades y se guara la mayor de ellas (excentricidad) en el array excentricidades. Posteriormente, se selecciona la ciudad con una excentricidad menor, guardando su índice en la variable capital. En caso de que varias ciudades tengan la misma excentricidad, se entra en un proceso de desempate lexicográfico. Para ello se utiliza la variable c, que almacena el nombre de la ciudad con la excentricidad más baja encontrada hasta el momento. Si una ciudad con la misma excentricidad tiene un nombre lexicográficamente menor que el nombre de la ciudad previamente almacenada en c, se actualiza la variable capital con el índice de esta ciudad y se actualiza c con el nuevo nombre. Así, se garantiza que, en caso de empate en excentricidad, se elija la ciudad alfabéticamente primera.

El código del programa es el siguiente:

```
#include <iostream>

#include <vector>

#include <string.h>

using namespace std;


#define MAX_NODOS 200

#define INFINITO 1000000



int numeroCiudades; //num ciudades caso actual

vector<string> ciudades;    //nombres ciudades

int C[MAX_NODOS][MAX_NODOS];    //matriz con distancias directas entre ciudades

int D[MAX_NODOS][MAX_NODOS];    //Matriz que usa algoritmo de Floyd para calcular las distancias mínimas entre todas las parejas de nodos

int A;  // A y B ciudades conectadas

int B;

int L;  // Distancia entre ciudades

int excentricidadCentro;    //Excentricidad ciudad central


void leeGrafo (void) {


        memset(C, 0, sizeof(C));    //Inicializa matriz con 0

        ciudades.clear();          // Limpia vector -> vacio


        cin >> numeroCiudades;  //lee ciudades en caso actual

        ciudades = vector<string>(numeroCiudades);  //redimensiona vector con tamaño necesario

        string ciudad;

        getline(cin, ciudad, '\n');


   // Bucle for, para leer todas las ciudades incluidas en el grafo y almacenarlas en ciudades
```

```
        for (int i = 0; i < numeroCiudades; i++) {
                getline(cin, ciudad, '\n');
                ciudades[i] = ciudad;
        }

   // Inizialización de la matriz
        for (int i = 0; i < numeroCiudades; i++){
                for (int j = 0; j< numeroCiudades; j++){
                        C[i][j] = INFINITO;
      }
   }

   //Lee dos ciudades y distancia
cin >> A >> B >> L;
while (!(A == 0 && B == 0 && L == 0)) {
      //Se actualizan ambas direcciones -> no dirigido
   C[A][B] = L;
   C[B][A] = L;
   cin >> A >> B >> L;
}

}


void AlgoritmoFloyd() {

        memset (D, 0, sizeof(D));

   //Copia las distancias directas de la matriz C a la D
        for (int i = 0; i < numeroCiudades; i++) {
                for (int j = 0; j < numeroCiudades; j++) {
```

```
                D[i][j] = C[i][j];
            }
            // Dos nodos iguales la distancia es 0
            D[i][i] = 0;
        }

    //Aplicacion algoritmo calculando distancias más cortas entre ciudades
        for (int k = 0; k < numeroCiudades; k++) {
            for (int i = 0; i < numeroCiudades; i++) {
                for (int j = 0; j < numeroCiudades; j++) {
                    if (D[i][j] > D[i][k] + D[k][j]) {
                        D[i][j] = D[i][k] + D[k][j];
                        }
                    }
                }
            }
}

int Centro(){

        int capital;
        int longitudMax;
        int excentricidades [numeroCiudades];

        // Recorremos todas las ciudades para calcular excentridad
        for (int i = 0; i < numeroCiudades; i++) {
            longitudMax = 0;
            for (int j = 0; j < numeroCiudades; j++) {
                if (D[i][j] >= longitudMax)
                longitudMax = D[i][j];
            }
```

```
                // Guardo excentridad de cada ciudad
                excentricidades[i] = longitudMax;
        }



        capital = 0;
   excentricidadCentro = INFINITO;
        string c;
        //Recorremos todas las ciudades para calcular la de menor excentridad
        for (int i = 0; i < numeroCiudades; i++) {
       //Si excentridad es menor a la que tengo, actualizo
     if (excentricidades[i] < excentricidadCentro) {
                        excentricidadCentro = excentricidades[i];
                        c = ciudades[i];
                        capital = i;
                }
                //Si execentridades iguales, elegimos por nombre lexicográfico menor
                else if (excentricidades[i] == excentricidadCentro) {
                        if (ciudades[i] < c) {
                                excentricidadCentro = excentricidades[i];
                                c = ciudades[i];
                                capital = i;
                        }
                }
        }
        return capital;
}



int main() {
```

```
	int ncasos;
	cin >> ncasos;

	for (int i = 0; i < ncasos; i++) {
		leeGrafo();
		AlgoritmoFloyd();
		cout << ciudades[Centro()] << endl;
	   cout << excentricidadCentro << endl;
	}
}
```

413. OPERACIÓN TERRAFORMACIÓN. ENVÍO: 1071

El código implementa el Algoritmo de Prim para calcular el árbol de expansión mínima en un grafo no dirigido, con el objetivo de determinar la cantidad mínima de cable necesario para conectar todas las colonias marcianas. Este enfoque es adecuado para problemas que requieren minimizar el coste de conectar nodos en un grafo.

En primer lugar, se declaran las variables globales: C, que representa la matriz de adyacencia, dos arrays X e Y, que guardan las coordenadas de las colonias, dos enteros x e y, para manejar la lectura de estas coordenadas y un array denominado visitado para marcar las colonias visitadas durante el cálculo del árbol de expansión mínima.

Para la función leeGrafo, que recibe como parámetro el número de colonias, en primer lugar, se ingresan las coordenadas de cada una de ellas, verificando que se encuentran dentro del rango establecido. Si es así, las guardamos en sus respectivos arrays X e Y. A continuación, inicializamos la matriz de adyacencia con ceros y vamos modificando estos valores con la distancia entre colonias correspondientes. Para ello, verificamos que i y j no sean iguales, lo que supondría que nos referimos a la misma colonia y la distancia se quedaría en cero, e inmediatamente después aplicamos la fórmula proporcionada por el problema con su adecuado truncamiento para obtener la distancia entre dos colonias.

Para la función AlgoritmoPrim comenzamos declarando dos variables locales: un array menorCoste, que almacena los costes mínimos para conectar cada nodo con el árbol de expansión en construcción y un entero cableMinimo, que es una variable acumulativa que guarda el coste total de cable necesario para conectar todas las colonias. A continuación, inicializamos el todo el array visitados a false para marcar que ninguna colonia ha sido visitada al principio, el array menorCoste de modo que contenga la distancia mínima desde el nodo inicial 0 al resto de nodos y marcamos el nodo inicial 0 como visitado. Posteriormente, con la ayuda de las variables min, que almacenará el coste mínimo para un nodo, y k que guardará el índice del nodo que se está añadiendo al árbol mínimo de expansión en cada iteración, buscamos el nodo no visitado con el menor coste. Al encontrarlo, se actualiza la variable min con ese coste y k con el índice del nodo, para poder marcarlo como visitado y añadir el coste a la variable acumulativa cableMinimo. Finalmente, se actualizan los costes mínimos del array menorCoste desde el nodo actual, para poder seguir buscando el nodo con menor coste desde el nuevo nodo añadido.

En el main, para cada caso de prueba, hacemos una pequeña comprobación para verificar que el número de colonias no supera el máximo de nodos establecido en el problema y se comienzan a tratar los casos introducidos.

El código del problema es:

```
#include <stdlib.h>        // exit

#include <string.h>        // memset

#include <iostream>      //  cin y cout

#include <math.h>       // sqrt

using namespace std;
```

```
#define MAX_NODOS 2000
#define INFINITO 1000000

int C[MAX_NODOS][MAX_NODOS];    //matriz de adyacencia
int X[MAX_NODOS];               // almacenamiento coordenadas colonias
int Y[MAX_NODOS];
int x, y;                       //lectura de coordenadas
bool visitado[MAX_NODOS];       // array nodos visitados

void leeGrafo(int numeroColonias) {
   // Lectura coordendas
   for (int i = 0; i < numeroColonias; i++) {
      cin >> x >> y;
      if ((x < -1000) || (x > 1000) || (y < -1000) || (y > 1000)) {
         cerr << "Coordenadas fuera del rango permitido\n";  // Fuera de rango
         exit(0);
      }
      //Si coordenadas correctas, guardamos en sus respectivos arrays
      X[i] = x;
      Y[i] = y;
   }

   // Inicializa matriz costes
   memset(C, 0, sizeof(C));
   for (int i = 0; i < numeroColonias; i++) {
      for (int j = 0; j < numeroColonias; j++) {
         if (i == j) {
            C[i][j] = 0;  // Mismo nodo distancia 0
         } else {
            //Calculo distancia
```

```
            int dx = X[i] - X[j];
            int dy = Y[i] - Y[j];
            // Se trunca pasandolo a int
            C[i][j] = C[j][i] = (int)sqrt(dx * dx + dy * dy);
        }
    }
  }
}

void AlgoritmoPrim(int numeroColonias) {
  int menorCoste[numeroColonias];
  int cableMinimo = 0;

  memset(visitado, false, sizeof(visitado));  //Inicializa todos las colonias array a false

  // Inicializa costes mínimos desde nodo inicial 0 al resto de nodos
  for (int i = 1; i < numeroColonias; i++) {
    menorCoste[i] = C[0][i];
  }

  visitado[0] = true; //nodo inical visitado

  // Busca nodo no visitado con el menor costo
  for (int i = 1; i < numeroColonias; i++) {
    int min = INFINITO;  //inicializamos con infinito para asignarle en cada iteracion el coste minimo desde el nodo actual a otro nodo no visitado
    int k = 0;  //Indice nodo con el costo mínimo

    // Encuentra nodo no visitado con menor coste
    for (int j = 1; j < numeroColonias; j++) {
      if (!visitado[j] && menorCoste[j] < min) {  //Si la colonia j no ha sido visitada...
```

```
            min = menorCoste[j];    //Actualizo costo mínimo
            k = j;      //Guardo el índice
          }
        }

        visitado[k] = true; //Marco como visitado
        cableMinimo += min; //cableMinimo = cableMinimo + min

        // Actualiza los costes mínimos desde el nodo actual al resto de nodos
        for (int h = 1; h < numeroColonias; h++) {
          if (!visitado[h] && C[k][h] < menorCoste[h]) {
            menorCoste[h] = C[k][h];    //Actualizo con costo mas bajo
          }
        }
    }
    cout << cableMinimo << endl;
}

int main() {
    int ncasos;
    cin >> ncasos;

    for (int i = 0; i < ncasos; i++) {
        int numColonias;
        cin >> numColonias;  // Lee el número de colonias
        if (numColonias <= 0 || numColonias > MAX_NODOS) {
            cerr << "Numero de nodos (" << numColonias << ") no valido\n";
            exit(0);
        }
        leeGrafo(numColonias);
        AlgoritmoPrim(numColonias);
```

```
    }


    return 0;
}
```

CONCLUSIONES

Personalmente, pienso que esta actividad es un excelente método para evaluar el tema de grafos. Además de combinar teoría y prácticas para la resolución de ejercicios propuestos, se le da la libertad al alumno para indagar e investigar otros métodos para ampliar sus conocimientos e incluso para escribir un código con mejor rendimiento. La realización de esta tarea me ha supuesto un gran reto, sobre todo al principio, por el manejo de esta nueva estructura de datos y de todos los diferentes algoritmos que lleva detrás. Esto ha llevado un tiempo considerable de unas 4-5 horas únicamente a estudiar los algoritmos. Para los ejercicios resueltos, he seleccionado aquellos que tenían un enunciado sencillo de entender y en cuya resolución estaba claro el algoritmo a usar; en esta parte habrán sido en torno a 20 horas de trabajo. Por otro lado, en cuanto a ayudas externas para la realización del trabajo, se encuentra youtube, con la visualización de varios vídeos para el entendimiento de los algoritmos con ejemplo gráficos y el uso de Chatgpt. Este uso, ha sido en secciones concretas del código, con el fin de entender errores, valorando y estudiando las recomendaciones sugeridas y adaptando sus comentarios a lo visto en clase. Entre las cosas más relevantes:

En el ejercicio 407, en la función centro, intenté hacer la comparación léxicográfica para ciudades con excentridades iguales utilizando strcmp, dado que tengo entendido que sirve para hacer este tipo de comparaciones, sin embargo, tras diversos intentos y varios errores, consulté a Chatgpt, quien me sugirió usar directamente elementos comparadores (< y >), lo cual fue realmente útil pues no sabía que se podía comparar cadenas de esa forma en c++.

En el ejercicio 413, en la función algoritmoPrim, en un inicio traté de realizar la comparación con menorCoste[j] y menorCoste[k] para encontrar el nodo con el menor coste. Sin embargo, al hacerlo, aparecían errores, y cuando conseguí que desaparecieran la comparación no se realizaba correctamente. Fue entonces cuando recurrí a Chatgpt, quien sugirió la variable min para almacenar el valor mínimo de coste en cada paso, lo que solucionó el problema y mejoro la claridad y legibilidad del código.

Es por ello que quiero recalcar, que, sí, se ha usado Chatgpt, pero su uso ha sido moderado y todas y cada una de sus respuestas han sido revisadas por mí y adaptadas al contenido visto en clase. Es por ello que, Chatgpt es una herramienta muy útil, pero hay que saber utilizarla razonadamente.