UNIVERSIDAD DE GUADALAJARA

CUCEI

ESTRUCTURA DE DATOS

Proyecto Final

Maestra:

Ing. Ana Jazmín Guerrero.

Alumnos:

Román Rodríguez María Fernanda

Vivian Chávez Rubén Emilio

Torres Contreras Jesus Antonio

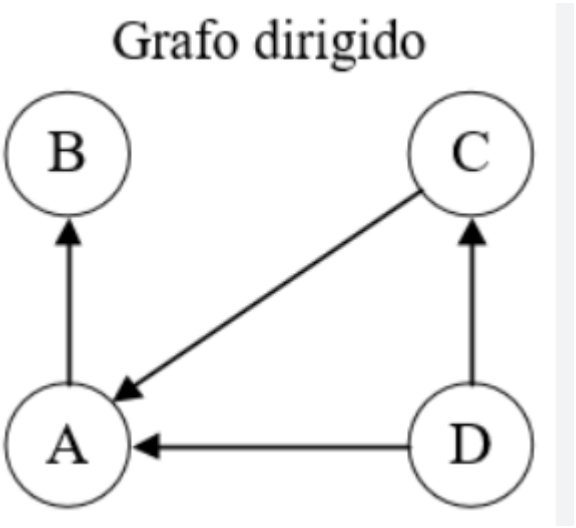
Sección: D02 16/05/2024

Equipo #7

----------Marco teórico----------

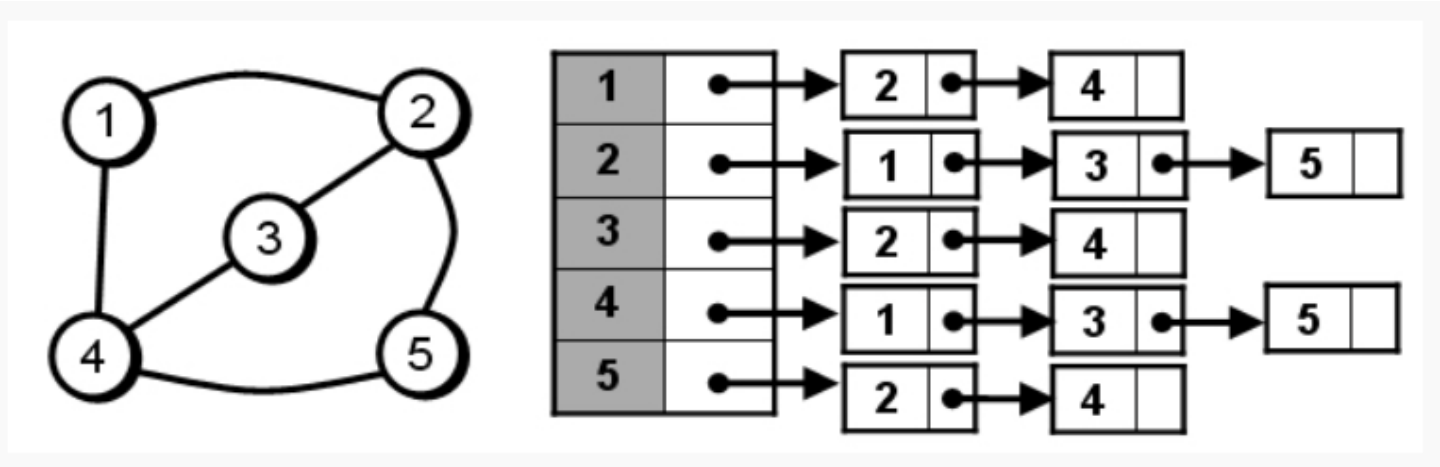
Gráficas dirigidas

Las gráficas dirigidas o digráficas son aquellas cuyas aristas siguen cierta dirección. En este caso, cada arista a = (V1, V2) recibe el nombre de arco y se representa con la notación v1-> V2. Se dice que v1 es el vértice origen o punto inicial y que V2 es el vértice destino o punto terminal del arco a. La figura 8.4 presenta un ejemplo de una gráfica dirigida.



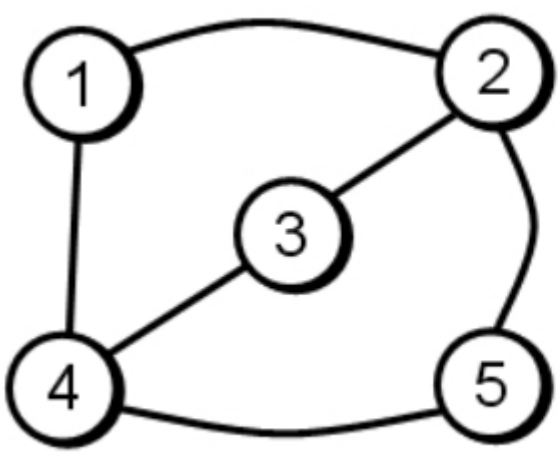
Ejemplo de gráfica dirigida

Las digráficas son estructuras de datos abstractas (como las pilas y colas), por lo tanto, los lenguajes de programación no cuentan con elementos diseñados exclusivamente para su representación y manejo. En consecuencia, se requiere utilizar alguna de las estructuras de datos ya estudiadas para su representación y almacenamiento en memoria. Las más usadas son las listas de adyacencia y las matrices de adyacencia. La lista de adyacencia está formada por una lista de listas. Es decir, cada nodo de la lista representa a un vértice y almacena, además de la información propia del vértice, una lista de vértices adyacentes. La figura 8.5 muestra un ejemplo de una digráfica y su correspondiente lista de adyacencia.



Gráfica dirigida y su representación por medio de una lista de adyacencia

La matriz de adyacencia es una representación de una digráfica mediante una matriz de números enteros. Los elementos en la posición i, j son 1 si hay un arco del vértice i al vértice j, y O en caso contrario. Para una digráfica con N vértices, la matriz será de NxN elementos. En el caso de digráficas etiquetadas, se utiliza una matriz de adyacencia etiquetada, donde en lugar de 1 se asigna la etiqueta o costo del arco. Esta matriz también se conoce como matriz de costos o matriz de distancias. La figura 8.7 muestra un ejemplo de una gráfica dirigida etiquetada junto con su matriz de adyacencia etiquetada correspondiente.



M	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	1
3	0	1	0	1	0
4	1	0	1	0	1
5	0	1	0	1	0

Gráfica dirigida etiquetada y su representación por medio de una matriz de adyacencia etiquetada

----------Descripción----------

Este programa utiliza gráficas dirigidas para simular un sistema ferroviario, permitiendo ingresar información sobre ciudades y costos de pasajes. Su objetivo principal es facilitar el aprendizaje y aplicación de algoritmos clave, como Warshall, Floyd y Dijkstra, para encontrar conexiones, costos mínimos y distancias entre ciudades en una red de transporte. La práctica se centra en comprender y aplicar estos algoritmos en el contexto de redes ferroviarias, utilizando matrices de adyacencia para representar eficientemente la información. La implementación práctica busca que los estudiantes adquieran habilidades en el manejo de algoritmos de grafos y comprendan cómo estas herramientas pueden analizar sistemas complejos como las redes de transporte.

----------Práctica para entregar----------

Muestra el funcionamiento del PROGRAMA 8.2 considerando la representación de un subconjunto de la red ferroviaria de un determinado país. Pág. 418-419. Lo cual permitirá encontrar ciudades comunicadas entre sí, así como los costos mínimos para ir de una ciudad a las otras o entre todas las ciudades. Para ello implementaremos un grafo dirigido el cual contendrá como atributo principal una matriz de adyacencias. Contemple el siguiente menú.

Menú - Sistema Ferroviario
0. Ingresar datos e imprimir matriz de adyacencias
1. Ciudades que están comunicadas entre sí (Warshall)
2. Mínimo costo entre todas las ciudades (Floyd)
3. Mínimo costo entre todas las ciudades y ciudades intermedias (FloydVerInt)
4. Mínimo costo entre una ciudad y las otras (Dijkstra)
5. Salir
Elige tu opción:

----------Código Fuente----------

Grafo.h

```
#ifndef GRAFO_H
#define GRAFO_H
#include <iostream>
#include <locale>
#include <fstream>
#define MAX 10


using namespace std;

template <class T>
class Grafo
{
private:
    T MatAdy[MAX][MAX], MatAux[MAX][MAX];
    int NumVer, Vertices[MAX], DistMin[MAX], CerTran[MAX][MAX], VerInt[MAX][MAX];
public:
    Grafo();
    void Lee();
    void Imprime(int);
    void Guarda(int);
    void Warshall();
    void Floyd();
    void FloydVerInt();
    void Dijkstra();
};


#endif // GRAFO_H
```

Grafo.cpp

```
#include "Grafo.h"

int Minimo(int Val1, int Val2)
{
    int Min= Val1;
    if (Val2 < Min)
    Min= Val2;
    return Min;
}

template <class T>
Grafo<T>::Grafo()
{
    int Ind1, Ind2;
```

```
    for (Ind1= 0; Ind1 < MAX; Ind1++){
        DistMin[Ind1]= 0;
        Vertices[Ind1]= 0;

        for (Ind2= 0; Ind2 < MAX; Ind2++){

            if (Ind1 != Ind2){
                MatAdy[Ind1][Ind2] = 999;
                MatAux[Ind1][Ind2] = 999;
            }

            else{
                MatAdy[Ind1][Ind2] = 0;
                MatAux[Ind1][Ind2] = 0;
            }

            CerTran[Ind1][Ind2]= 0;
            VerInt[Ind1][Ind2]= 0;
        }
    }

    NumVer= 0;
}

template <class T>
void Grafo<T>::Lee()
{
    int NumArcos, Indice, Origen, Destino;
    cout<<"\n\n Ingrese numero de vertices de la grafica dirigida: ";
    cin>>NumVer;
    for (Indice = 0; Indice < NumVer; Indice++){
        cout<<"\n\n Ingrese el nombre del vertice de la grafica dirigida: ";
        cin>>Vertices[Indice];
    }
    cout<<"\n\n Total de aristas de la grafica dirigida: ";
    cin>>NumArcos;
    Indice= 0;
    while (Indice < NumArcos){
        cout<<"\n\n Ingrese vertices origen: ";
        cin>>Origen;
        cout<<"\n\n Ingrese vertices destino: ";
        cin>>Destino;
        cout<<"\n\n Costo de origen a destino: ";
        cin>>MatAdy[Origen - 1][Destino - 1];
        MatAux[Origen - 1][Destino - 1] = MatAdy[Origen - 1][Destino - 1];
        Indice++;
    }
}

template <class T>
void Grafo<T>::Imprime(int Opc)
{
    int Ind1, Ind2;
    switch(Opc)
    {
        case 0: cout<<"\n\n Matriz de Adyacencia o de Costos: \n\n";
            for (Ind1= 0; Ind1 < NumVer; Ind1++)
            {
                cout<<Vertices[Ind1]<<": ";
                for (Ind2= 0; Ind2 < NumVer; Ind2++)
                    cout<<MatAdy[Ind1][Ind2]<<'\t';
                cout<<endl;
            }
            break;
        case 1: cout<<"\n\n Cerradura Transitiva de la Matriz de Adyacencia: "<<endl;
            for (Ind1= 0; Ind1 < NumVer; Ind1++)
            {
                cout<<Vertices[Ind1] <<": ";
                for (Ind2= 0; Ind2 < NumVer; Ind2++)
                    cout<<CerTran[Ind1][Ind2]<<'\t';
                cout<<endl;
            }
            break;
        case 2: cout<<"\n\n Matriz de Distancias Minimas: "<<endl;
            for (Ind1= 0; Ind1 < NumVer; Ind1++)
            {
                cout<<Vertices[Ind1]<<": ";
                for (Ind2= 0; Ind2 < NumVer; Ind2++)
```

```
                    cout<<MatAux[Ind1][Ind2]<<'\t';
                cout<<endl;
            }
            break;
        case 3: cout<<"\n\n Vertices Intermedios para lograr distancias minimas: "<<endl;
            for (Ind1= 0; Ind1 < NumVer; Ind1++)
            {
                for (Ind2= 0; Ind2 < NumVer; Ind2++)
                    cout<<VerInt[Ind1][Ind2]<<'\t';
                cout<<endl;
            }
            break;
        case 4: cout<<"\n\n Distancias minimas a partir del vertice: "<<Vertices[0]<<"\n\n";
            for (Ind1= 0; Ind1 < NumVer; Ind1++)
                cout<<' '<<DistMin[Ind1]<<'\t'<<endl;
            break;
        default: break;
    }
    cout<<endl;
}
template <class T>
void Grafo<T>::Guarda(int Opc)
{
    int Ind1, Ind2;
    switch(Opc){
        case 1:
            {
                ofstream archWarshall;
                archWarshall.open("Warshall.txt", ios::out);
                for (Ind1= 0; Ind1 < NumVer; Ind1++){
                    for (Ind2= 0; Ind2 < NumVer; Ind2++){
                        archWarshall<<CerTran[Ind1][Ind2]<<'\t';
                    }
                    archWarshall<<endl;
                }
                archWarshall.close();
                break;
            }
        case 2:
            {
                ofstream archFloyd;
                archFloyd.open("Floyd.txt", ios::out);
                for (Ind1= 0; Ind1 < NumVer; Ind1++)
                {
                    for (Ind2= 0; Ind2 < NumVer; Ind2++){
                        archFloyd<<MatAux[Ind1][Ind2]<< '\t';
                    }
                    archFloyd<<endl;
                }
                archFloyd.close();
                break;
            }
        case 3:
            {
                ofstream archFloydVerInt;
                archFloydVerInt.open("FloydVerInt.txt", ios::out);
                for (Ind1= 0; Ind1 < NumVer; Ind1++)
                {
                    for (Ind2= 0; Ind2 < NumVer; Ind2++){
                        archFloydVerInt<<VerInt[Ind1][Ind2]<<'\t';
                    }
                    archFloydVerInt<<endl;
                }
                archFloydVerInt.close();
                break;

            }
        case 4:
            {
                ofstream archDijkstra;
                archDijkstra.open("Dijkstra.txt", ios::out);
                for (Ind1= 0; Ind1 < NumVer; Ind1++){
                    archDijkstra<<DistMin[Ind1]<<'\t'<<endl;
                }
                archDijkstra.close();
                break;
            }
        default: break;
    }
```

```
}
template <class T>
void Grafo<T>::Warshall()
{
    int Ind1, Ind2, Ind3;
    for (Ind1= 0; Ind1 < NumVer; Ind1++){
        for (Ind2= 0; Ind2 < NumVer; Ind2++){
            if (MatAdy[Ind1][Ind2] != 999){
                CerTran[Ind1][Ind2]= 1;
            }
        }
    }
    for (Ind3= 0; Ind3 < NumVer; Ind3++){
        for (Ind1= 0; Ind1 < NumVer; Ind1++){
            for (Ind2= 0; Ind2 < NumVer; Ind2++){
                CerTran[Ind1][Ind2] |= CerTran[Ind1][Ind3] && CerTran[Ind3][Ind2];
            }
        }
    }
}
template <class T>
void Grafo<T>::Floyd()
{
    int Ind1, Ind2, Ind3;
    for (Ind3= 0; Ind3 < NumVer; Ind3++){
        for (Ind1= 0; Ind1 < NumVer; Ind1++){
            for (Ind2= 0; Ind2 < NumVer; Ind2++){
                if ( (MatAux[Ind1][Ind3] + MatAux[Ind3][Ind2]) < MatAux[Ind1][Ind2]){
                    MatAux[Ind1][Ind2] = MatAux[Ind1][Ind3] + MatAux[Ind3][Ind2];
                }
            }
        }
    }
}
template <class T>
void Grafo<T>::FloydVerInt()
{
    int Ind1, Ind2, Ind3;
    for (Ind3 = 0; Ind3 < NumVer; Ind3++){
        for (Ind1 = 0; Ind1 < NumVer; Ind1++){
            for (Ind2 = 0; Ind2 < NumVer; Ind2++){
                if ((MatAdy[Ind1][Ind3] + MatAdy[Ind3][Ind2]) < MatAdy[Ind1][Ind2]){
                    MatAdy[Ind1][Ind2] = MatAdy[Ind1][Ind3] + MatAdy[Ind3][Ind2];
                    VerInt[Ind1][Ind2] = Vertices[Ind3];
                }
            }
        }
    }
}
template <class T>
void Grafo<T>::Dijkstra()
{
    int Aux[MAX], VertInc[MAX], Ver1, Ver2, Ind1, Ind2, Menor, Band,
    Origen;
    for (Ind1= 0; Ind1 < NumVer; Ind1++){
        Aux[Ind1]= 0;
        VertInc[Ind1]= 0;
    }
    cout<<"\n\n Ingrese vertice origen: ";
    cin>>Origen;
    Origen -= 1;
    Aux[Origen]= 1;
    for (Ind1= 0; Ind1 < NumVer; Ind1++){
        DistMin[Ind1]= MatAdy[Origen][Ind1];
    }
    for (Ind1= 0; Ind1<NumVer; Ind1++){
        Menor= 999;
        for (Ind2= 1; Ind2 < NumVer; Ind2++){
            if (DistMin[Ind2] < Menor && !Aux[Ind2]){
                Ver1= Ind2;
                Menor= DistMin[Ind2];
            }
        }
        VertInc[Ind1]= Ver1;
        Aux[Ver1]= 1;
        Ver2= 1;
        while (Ver2 < NumVer){
            Band=0;
            Ind2= 1;
```

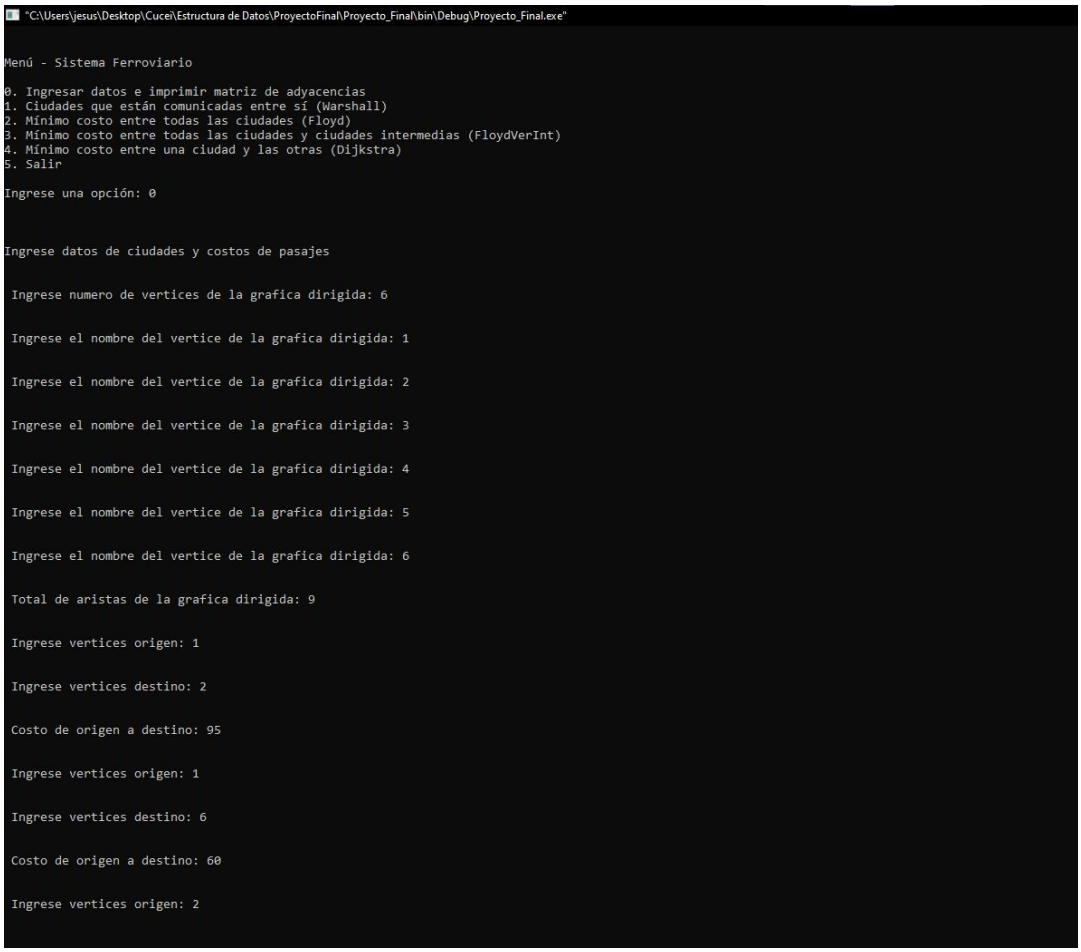
```
            while (Ind2 < NumVer && !Band){
                if (VertInc[Ind2] == Ver2){
                    Band= 1;
                }
                else{
                    Ind2++;
                }
            }
            if (!Band){
                DistMin[Ver2] = Minimo (DistMin[Ver2], DistMin[Ver1] + MatAdy[Ver1][Ver2]);
            }
            Ver2++;
        }
    }
}

template class Grafo<int>;
```

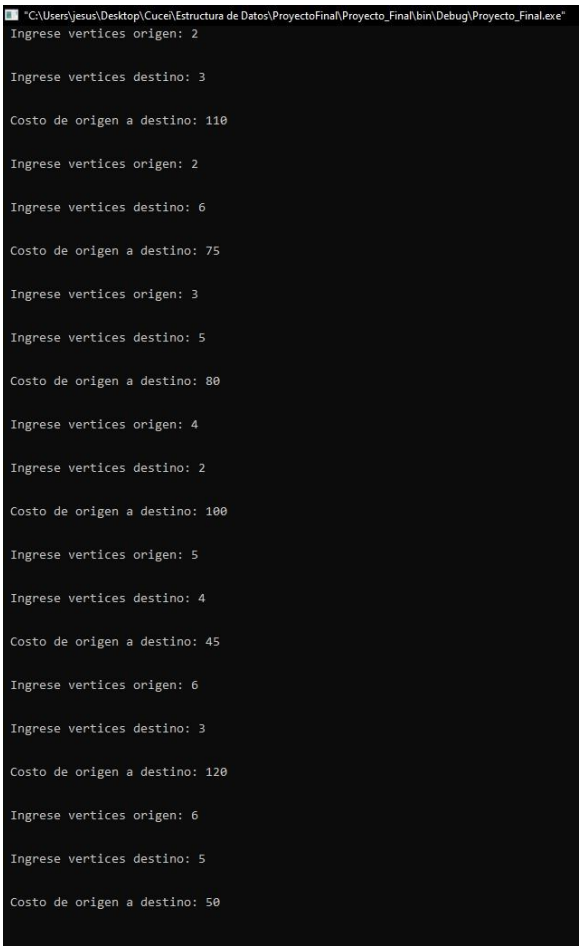
Main.cpp

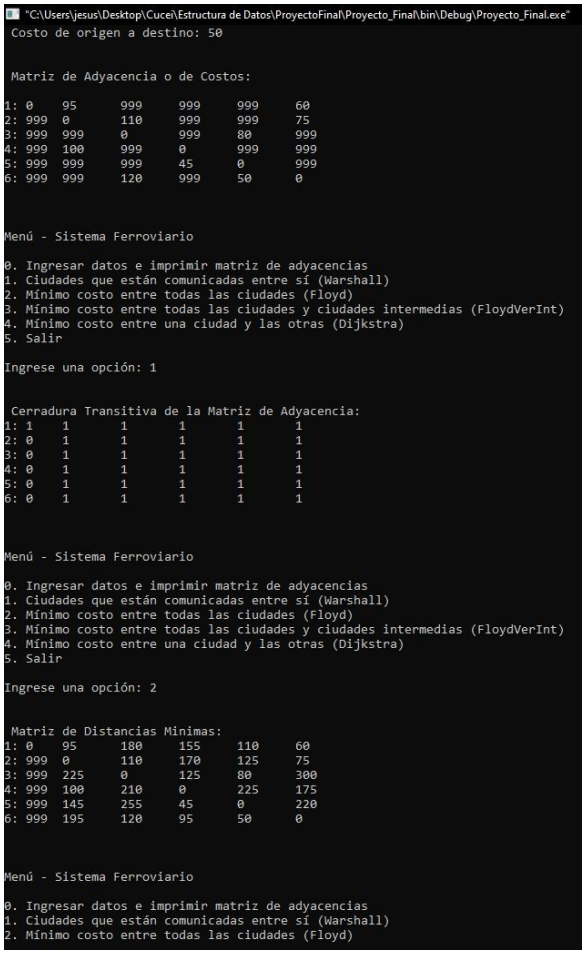
```
/*
Programadores:

Román Rodríguez María Fernanda
Vivian Chávez Rubén Emilio
Torres Contreras Jesus Antonio

Este programa utiliza grÃ¡ficas dirigidas para simular un sistema ferroviario,
permitiendo ingresar informaciÃ³n sobre ciudades y costos de pasajes. Su objetivo
principal es facilitar el aprendizaje y aplicaciÃ³n de algoritmos clave, como Warshall,
Floyd y Dijkstra, para encontrar conexiones, costos mÃ-nimos y distancias entre ciudades
en una red de transporte. La prÃ¡ctica se centra en comprender y aplicar estos algoritmos
en el contexto de redes ferroviarias, utilizando matrices de adyacencia para representar
eficientemente la informaciÃ³n. La implementaciÃ³n prÃ¡ctica busca que los estudiantes
adquieran habilidades en el manejo de algoritmos de grafos y comprendan cÃ³mo estas
herramientas pueden analizar sistemas complejos como las redes de transporte.
*/
#include <iostream>
#include <locale>
#include <fstream>
#include "Grafo.h"
int Menu();

using namespace std;

int Menu()
{
    int Opc;
    do
    {
        cout<<"\n\nMenÃ° - Sistema Ferroviario\n";
        cout<<"\n0. Ingresar datos e imprimir matriz de adyacencias";
        cout<<"\n1. Ciudades que estÃ¡n comunicadas entre sÃ- (Warshall)";
        cout<<"\n2. MÃ-nimo costo entre todas las ciudades (Floyd)";
        cout<<"\n3. MÃ-nimo costo entre todas las ciudades y ciudades intermedias (FloydVerInt)";
        cout<<"\n4. MÃ-nimo costo entre una ciudad y las otras (Dijkstra)";
        cout<<"\n5. Salir";
        cout<<"\n\nIngrese una opciÃ³n: ";
        cin>>Opc;
    } while (Opc < 0 || Opc > 5);
    return Opc;
}

int main()
{
    setlocale(LC_CTYPE, "spanish");
    Grafo<int> graf;
    int Opc;
    do
    {
        Opc= Menu();
        switch (Opc)
        {
        case 0:
            cout<<"\n\n\nIngrese datos de ciudades y costos de pasajes\n";
            graf.Lee();
            graf.Imprime(0);
```

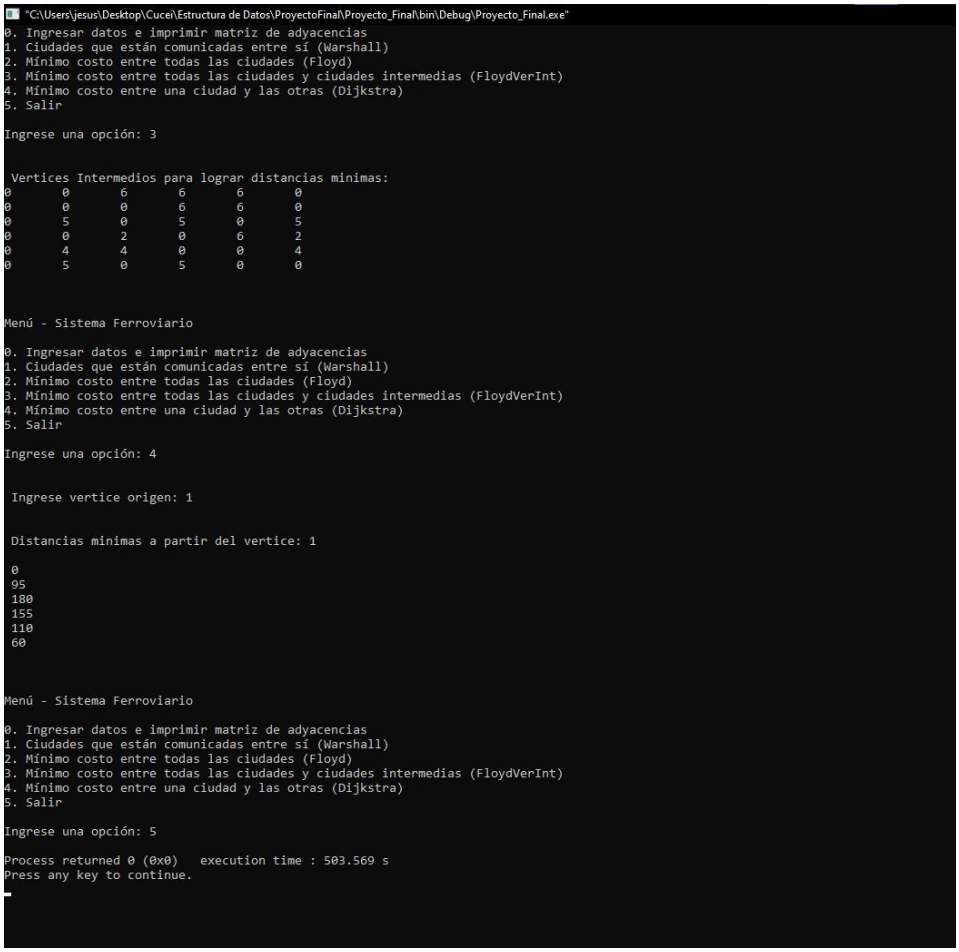
```
            break;
            case 1:
            {
                graf.Warshall();
                graf.Imprime(1);
                graf.Guarda(1);
                break;
            }
            case 2:
            {
                graf.Floyd();
                graf.Imprime(2);
                graf.Guarda(2);
                break;
            }
            case 3:
            {
                graf.FloydVerInt();
                graf.Imprime(3);
                graf.Guarda(3);
                break;
            }
            case 4:
            {
                graf.Dijkstra();
                graf.Imprime(4);
                graf.Guarda(4);
                break;
            }
        }
    } while (Opc < 5 && Opc >= 0);
    return 0;
}
```

----------Pantalla de salida----------




"C:\Users\jesus\Desktop\Cucei\Estructura de Datos\ProyectoFinal\Proyecto_Final\bin\Debug\Proyecto_Final.exe"
Ingrese vertices origen: 2
Ingrese vertices destino: 3
Costo de origen a destino: 110
Ingrese vertices origen: 2
Ingrese vertices destino: 6
Costo de origen a destino: 75
Ingrese vertices origen: 3
Ingrese vertices destino: 5
Costo de origen a destino: 80
Ingrese vertices origen: 4
Ingrese vertices destino: 2
Costo de origen a destino: 100
Ingrese vertices origen: 5
Ingrese vertices destino: 4
Costo de origen a destino: 45
Ingrese vertices origen: 6
Ingrese vertices destino: 3
Costo de origen a destino: 120
Ingrese vertices origen: 6
Ingrese vertices destino: 5
Costo de origen a destino: 50


"C:\Users\jesus\Desktop\Cucei\Estructura de Datos\ProyectoFinal\Proyecto_Final\bin\Debug\Proyecto_Final.exe"
Costo de origen a destino: 50
Matriz de Adyacencia o de Costos:
1: 0 95 999 999 999 60
2: 999 0 110 999 999 75
3: 999 999 0 999 80 999
4: 999 100 999 0 999 999
5: 999 999 999 45 0 999
6: 999 999 120 999 50 0
Menú - Sistema Ferroviario
0. Ingresar datos e imprimir matriz de adyacencias
1. Ciudades que están comunicadas entre sí (Warshall)
2. Mínimo costo entre todas las ciudades (Floyd)
3. Mínimo costo entre todas las ciudades y ciudades intermedias (FloydVerInt)
4. Mínimo costo entre una ciudad y las otras (Dijkstra)
5. Salir
Ingrese una opción: 1
Cerradura Transitiva de la Matriz de Adyacencia:
1: 1 1 1 1 1 1
2: 0 1 1 1 1 1
3: 0 1 1 1 1 1
4: 0 1 1 1 1 1
5: 0 1 1 1 1 1
6: 0 1 1 1 1 1
Menú - Sistema Ferroviario
0. Ingresar datos e imprimir matriz de adyacencias
1. Ciudades que están comunicadas entre sí (Warshall)
2. Mínimo costo entre todas las ciudades (Floyd)
3. Mínimo costo entre todas las ciudades y ciudades intermedias (FloydVerInt)
4. Mínimo costo entre una ciudad y las otras (Dijkstra)
5. Salir
Ingrese una opción: 2
Matriz de Distancias Minimas:
1: 0 95 180 155 110 60
2: 999 0 110 170 125 75
3: 999 225 0 125 80 300
4: 999 100 210 0 225 175
5: 999 145 255 45 0 220
6: 999 195 120 95 50 0
Menú - Sistema Ferroviario
0. Ingresar datos e imprimir matriz de adyacencias
1. Ciudades que están comunicadas entre sí (Warshall)
2. Mínimo costo entre todas las ciudades (Floyd)


"C:\Users\jesus\Desktop\Cucei\Estructura de Datos\ProyectoFinal\Proyecto_Final\bin\Debug\Proyecto_Final.exe"
0. Ingresar datos e imprimir matriz de adyacencias
1. Ciudades que están comunicadas entre sí (Warshall)
2. Mínimo costo entre todas las ciudades (Floyd)
3. Mínimo costo entre todas las ciudades y ciudades intermedias (FloydVerInt)
4. Mínimo costo entre una ciudad y las otras (Dijkstra)
5. Salir
Ingrese una opción: 3
Vertices Intermedios para lograr distancias minimas:
0 0 6 6 6 0
0 0 0 6 6 0
0 5 0 5 0 5
0 0 2 0 6 2
0 4 4 0 0 4
0 5 0 5 0 0
Menú - Sistema Ferroviario
0. Ingresar datos e imprimir matriz de adyacencias
1. Ciudades que están comunicadas entre sí (Warshall)
2. Mínimo costo entre todas las ciudades (Floyd)
3. Mínimo costo entre todas las ciudades y ciudades intermedias (FloydVerInt)
4. Mínimo costo entre una ciudad y las otras (Dijkstra)
5. Salir
Ingrese una opción: 4
Ingrese vertice origen: 1
Distancias minimas a partir del vertice: 1
0
95
180
155
110
60
Menú - Sistema Ferroviario
0. Ingresar datos e imprimir matriz de adyacencias
1. Ciudades que están comunicadas entre sí (Warshall)
2. Mínimo costo entre todas las ciudades (Floyd)
3. Mínimo costo entre todas las ciudades y ciudades intermedias (FloydVerInt)
4. Mínimo costo entre una ciudad y las otras (Dijkstra)
5. Salir
Ingrese una opción: 5
Process returned 0 (0x0) execution time : 503.569 s
Press any key to continue.

Dijkstra.txt - Notepad

File Edit Format View Help

```
0
95
180
155
110
60
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

Floyd.txt - Notepad

File Edit Format View Help

```
0       95      180     155     110     60
999     0       110     170     125     75
999     225     0       125     80      300
999     100     210     0       225     175
999     145     255     45      0       220
999     195     120     95      50      0
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

FloydVerInt.txt - Notepad

File Edit Format View Help

```
0       0       6       6       6       0
0       0       0       6       6       0
0       5       0       5       0       5
0       0       2       0       6       2
0       4       4       0       0       4
0       5       0       5       0       0
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

Warshall.txt - Notepad

File Edit Format View Help

```
1       1       1       1       1       1
0       1       1       1       1       1
0       1       1       1       1       1
0       1       1       1       1       1
0       1       1       1       1       1
0       1       1       1       1       1
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

----------Conclusiones----------

Román Rodríguez María Fernanda

Las gráficas dirigidas son como mapas donde las conexiones tienen una dirección. En nuestro programa, usamos una plantilla llamada DiGrafica.h para manejar esta estructura, almacenando las conexiones en una matriz. Usamos métodos como Marshall para encontrar ciudades alcanzables, Floyd para encontrar el camino más económico, FloydVerInt para identificar las ciudades en esos caminos, y Dijkstra para calcular el costo mínimo entre una ciudad y todas las demás.

Vivian Chávez Rubén Emilio

La creación de este programa nos ayudo a reforzar de forma significativa el uso de algoritmos fundamentales de grafos, el uso y la aplicación de Warshall, Floyd y Dijkstra como si fueran una red de trenes, nos dio un punto de vista más práctico del uso de estos para resolver problemas complejos cotidianos. Con dicho conocimiento nos da la oportunidad de analizar y optimizar de forma eficiente sistemas de transporte, conexiones y costos entre ciudades.

Torres Contreras Jesus Antonio

A lo largo del desarrollo de este trabajo nos dimos cuenta de lo importante que es comprender este tipo de algoritmos, como es que gracias a estos logramos obtener una red de grafos conectados entre si que nos indican tanto los precios, como los caminos mas cortos, incluso el precio mas barato para llegar a cierto camino. Con esto nos damos cuenta de las posibles aplicaciones que este tipo de programas pueden tener a futuro para conectar lugares y también vemos la importancia de los grafos, ya que como se sabe, esto es algo que se usa en el día a día.