Tecnológico de Monterrey

Act 4.3 - Actividad Integral de Grafos

José Pablo Galindo Hernández

A01276079

Programación de estructuras de datos y algoritmos fundamentales

Noviembre 2023

Indice

Introducción ……pág.3

Problemática…….pág.3

Solución Propuesta ……..pág.3

Código y explicación ……...pág.4, pág.10

Funcionamiento del Código..pág.10, pág.11

Conclusión ……...pág.11

Introducción

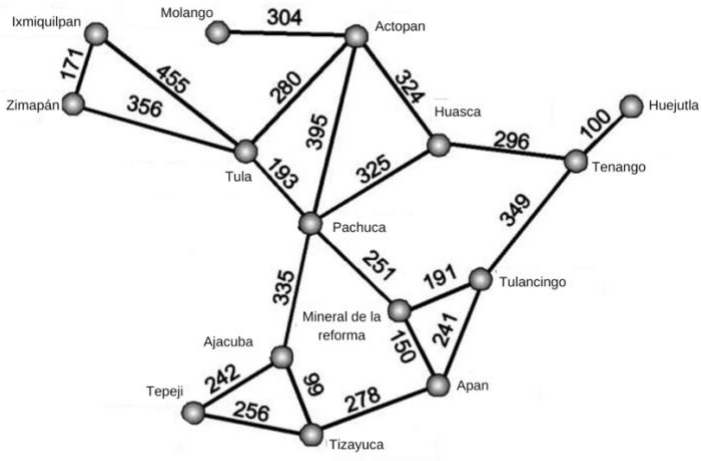
Para la realización de este proyecto hicimos uso de todos los conocimientos adquiridos sobre grafos a lo largo de estas últimas 5 semanas, los que nos permitieron analizar distintas soluciones para poder seleccionar la más eficiente, además hicimos uso de otras herramientas de programación aprendidas en todo el semestre, utilizando distintas librerías y funciones que nos permitirían crear el siguiente programa.

Problemática

Nuestro proyecto fue realizar un sistema que simula estaciones de autobús puestas por todos el estado de Hidalgo, donde nuestro programa permite al usuario identificar la ruta más rapida, estas estaciones están ubicadas en los municipios más importantes del estado, esto puede ayudar a alguien simplificando un poco el trayecto que deben hacer para llegar a un lugar ya que hay ocaciones donde los mapas de los sistemas de transporte pueden ser un poco confusos y con el nuestro se simplifica para saber el número de la estación para ahorrar tiempo al buscar el viaje y también al viajar porque nos muetra la ruta más rapida.

Solución Propuesta

Para resolver este problema que nosotros identificamos decidimos usar el algoritmo de Dijkstra, que nos permite encontrar el camino más corto desde un nodo de inicio hasta uno de fin, que en este caso en particular se trata de estaciones de transporte publico ubicadas por todo el estado, además asignando un peso a los nodos nos permite encontrar la ruta más corta mucho más facilmente. Para este programa utilizamos el lenguaje de programación C++ además de programar orientado a objetos. Para poder programar el sistema de estaciones de transporte público nos basamos en el siguiente nodo que incuye el nombre de las ciudades donde está ubicada cada estación y el peso que tiene cada trayecto.



Código y explicación

Para este código utilizamos las librerías iostream que nos proporcionan las funciones predeterminadas de entrada y salida, también hicimos uso de la librería vector que nos permite crear arreglos dinámicos que pueden cambiar de tamaño en este código, en particular la utilizamos para la lista de adyacencias, la tercer librería y última que utilizamos fue limits, que nos permite poner un valor máximo en el algoritmo de Dijkstra.

```
1  //José Pablo Galindo Hernández A01276079
2  //Roberto Carlos Pedraza Miranda A01277764
3
4  #include <iostream>
5  #include <vector>
6  #include <limits>
7  using namespace std;
8
```

Empezamos la clase CGrafo con las variables que tenemos declaradas como privadas, la primera es de tipo entero y tiene de nombre V que significa la cantidad de vértices que tiene el grafo, seguido de la segunda cosa que está declarada como privada, que es la lista de adyacencias que es un vector de vectores explícitamente contiene pares de enteros, ésta nos sirve para poder almacenar las conexiones que tiene cada nodo con sus respectivos pesos.

```
class CGrafo {
private:
    int V;
    vector<vector<pair<int, int>>> ListaAdyacencia;
```

Después pasamos con lo que está declarado como privado dentro de la misma clase, lo primero que ponemos es el constructor de la clase que tiene el mismo nombre, aquí toma como parámetros los vértices para después utilizar la función resize para así cambiar al tamaño de la lista de adyacencia dependiendo del número de vértices. La siguiente función que tiene de nombre Estaciones, primero declara u y v que son los encargados del vértice de inicio y fin, también se declara el peso que tiene esta conexión, usamos la función puch_back para agregar un valor más a la lista de adyacencia que en este caso es el peso, dentro de esta usamos la función make_pair que nos ayuda a crear un par juntando los argumentos de v ó u y peso.

```
14  public:
15      CGrafo(int vertices) : V(vertices) {
16          ListaAdyacencia.resize(V);
17      }
18
19      void Estaciones(int u, int v, int peso) {
20          ListaAdyacencia[u].push_back(make_pair(v, peso));
21          ListaAdyacencia[v].push_back(make_pair(u, peso));
22      }
```

El siguiente paso es la función de Dijkstra donde lo primero es poner que nos devuelva un par de elementos los que representan la distancia mínima desde el nodo de inicio y el siguiente es el camino más corto entre los dos nodos requeridos. Después creamos un vector de tipo entero de nombre distancia donde usando la cantidad de nodos establecemos un límite máximo usando la función numeric_limits que es parte de la librería limits. El siguiente paso es establecer la distancia entre el nodo de inicio, luego crear un vector con el nombre de padre de tipo entrero con el valor de -1 que significa que todavía no está asignado a ningún vértice.

```
24      pair<int, vector<int>> Dijkstra(int NodoInicio, int NodoFin) {
25              vector<int> distancia(V, numeric_limits<int>::max());
26              distancia[NodoInicio] = 0;
27              vector<int> padre(V, -1);
```

El primer bucle for se utiliza porque en el algoritmo de Dijkstra la longuitud del camino más corto no puede ser mayor a la cantidad de nodos menos uno, el siguiente bucle se utiliza para recorrer todos los nodos del grafo, el tercer bucle incluímos auto para poder deducir automáticamente el tipo de dato que tiene la lista adyecente de u, en este caso vecino junto con la funcion first se refieren al primer elemento que en este caso es v y donde vecino.second se refiere al peso. Para cerrar los bucles lo último que hicimos fue con un if verificar si la distancia entre nodos es infinita que si lo es, no existe un camino entre los vértices y por último se encarga de cambiar si se encuentra un camino más corto y después sumarlo.

```
28
29              for (int i = 0; i < V - 1; ++i) {
30                  for (int u = 0; u < V; ++u) {
31                      for (auto vecino : ListaAdyacencia[u]) {
32                          int v = vecino.first;
33                          int peso = vecino.second;
34                          if (distancia[u] != numeric_limits<int>::max() && distancia[u] +
                                peso < distancia[v]) {
35                              distancia[v] = distancia[u] + peso;
36                              padre[v] = u;
37                          }
38                      }
39                  }
40              }
```

Lo último que hay dentro de la clase CGrafo se encarga de reconstruir los pasos que sigue el algoritmo para encontrar el camino más corto entre nodos, lo primero es crear un vector de nombre recorrido que es donde se almacenarán los datos, después se declara que actual es igual al nodo final, el siguiente paso es usar un ciclo while donde la condición es que el vértice actual no debe ser igual al nodo de inicio, si esto se cumple usamos puch_back para poner el nodo correspondiente hasta el último, usamos el segundo puch_back para poner el nodo de inicio hasta el final para sabe de donde empezamos y lo último es hacer un return que regrese al camino más corto recorrido.

```
42              vector<int> recorrido;
43              int actual = NodoFin;
44              while (actual != NodoInicio) {
45                  recorrido.push_back(actual);
46                  actual = padre[actual];
47              }
48              recorrido.push_back(NodoInicio);
49              return make_pair(distancia[NodoFin], recorrido);
50          }
51      };
```

Entrando al main lo primero que hacemos es declarar lo que necesitamos para que el programa funcione.

```
54  int main() {
55      int opcion;
56      int Estacionsalida;
57      int Estacionllegada;
58      int V = 15;
59      CGrafo migrafo(V);
```

Después ingresamos los datos de las estaciones usando el formato de (primer nodo, segundo nodo, peso).

```
61      migrafo.Estaciones(0, 1, 251);
62      migrafo.Estaciones(0, 6, 335);
63      migrafo.Estaciones(0, 9, 325);
64      migrafo.Estaciones(0, 10, 395);
65      migrafo.Estaciones(0, 12, 193);
66      migrafo.Estaciones(1, 2, 191);
67      migrafo.Estaciones(1, 3, 150);
68      migrafo.Estaciones(2, 3, 241);
69      migrafo.Estaciones(2, 7, 349);
70      migrafo.Estaciones(3, 4, 278);
71      migrafo.Estaciones(4, 5, 256);
72      migrafo.Estaciones(4, 6, 99);
73      migrafo.Estaciones(6, 5, 242);
74      migrafo.Estaciones(7, 8, 100);
75      migrafo.Estaciones(7, 9, 296);
76      migrafo.Estaciones(9, 10, 324);
77      migrafo.Estaciones(10, 11, 304);
78      migrafo.Estaciones(10, 12, 280);
79      migrafo.Estaciones(12, 13, 356);
80      migrafo.Estaciones(12, 14, 455);
81      migrafo.Estaciones(13, 14, 171);
```

El siguiente paso es crear el menú para que el usuario decida que es lo que quiere hacer. Nuestro menú cuenta con tres opciones, la primera es ingresar el nodo de inicio y el nodo de fin que corresponden a cada estación para saber el recorrido más corto, la siguiente opción es la de ver el número de la estación que corresponde a cada municipio y por último la tercer opción es finalizar el programa.

```
83      do {
84          cout << "Seleccione una opción: \n";
85          cout << "1.- Buscar recorrido más corto \n";
86          cout << "2.- Ver Estaciones \n";
87          cout << "3.- Salir\n";
88          cout << "Ingrese su opción: ";
89          cin >> opcion;
90
```

Lo primero que hace el caso uno es pedir al usuario la estación de salida, lo siguiente es preguntar la estación de destino, después de eso se llama al método Djikstra pasando como argumentos los datos que ingreso el usuario, el resultado.first incluye los datos de la distancia mínima, mientras que el resultado.second contiene el recorrido que realizó.

```
91          switch (opcion) {
92              case 1: {
93                  cout << "Ingrese la estación de salida: ";
94                  cin >> Estacionsalida;
95                  cout << "Ingrese la estación de llegada: ";
96                  cin >> Estacionllegada;
97
98                  pair<int, vector<int>> resultado = migrafo.Dijkstra(Estacionsalida,
                        Estacionllegada);
99                  int distancia = resultado.first;
100                 vector<int> recorrido = resultado.second;
101
102                 if (distancia != -1) {
103                     cout << "\nLa distancia mínima entre las estaciones es: " << distancia
                            << endl;
104                     cout << "El recorrido es: ";
105                 for (int estacion : recorrido) {
106                     cout << estacion << " ";
107
108                 }
109                     cout << endl;
110
111                 } else{
112                     cout << "\nNo hay ruta entre las estaciones" << endl;
113
114                 }
115                 break;
116
117         }
```

El segundo caso imprime el número de estación y el nombre del municipio.

```
118             case 2: {
119                 cout << "0.- Pachuca de Soto\n";
120                 cout << "1.- Mineral de la Reforma\n";
121                 cout << "2.- Tulancingo\n";
122                 cout << "3.- Apan\n";
123                 cout << "4.- Tizayuca\n";
124                 cout << "5.- Tepeji del río\n";
125                 cout << "6.- Ajacuba\n";
126                 cout << "7.- Tenango de doria\n";
127                 cout << "8.- Huejutla de reyes\n";
128                 cout << "9.- Huasca de ocampo\n";
129                 cout << "10.- Actopan\n";
130                 cout << "11.- Molango de Escamilla\n";
131                 cout << "12.- Tula de Allende\n";
132                 cout << "13.- Zimapán\n";
133                 cout << "14.- Ixmiquilpan\n";
134                 break;
135             }
```

Por último si no se ingresa una opción del menú válida muestra el mensaje de “Opción inválida. Intente de nuevo” y se repite el menú, por último al seleccionar tres se cierra el programa.

```
            case 3: {
                cout << "Saliendo del programa\n";
                break;
            }
            default: {
                cout << "Opción inválida. Intente de nuevo." << endl;
            }
        }
    } while (opcion != 3);

    return 0;
}
```

Funcionamiento del Código

```
Seleccione una opción:
1.- Buscar recorrido más corto
2.- Ver Estaciones
3.- Salir
Ingrese su opción:
```

```
Ingrese su opción: 1
Ingrese la estación de salida: 0
Ingrese la estación de llegada: 8

La distancia mínima entre las estaciones es: 721
El recorrido es: 8 7 9 0
```

```
Ingrese su opción: 2
0.- Pachuca de Soto
1.- Mineral de la Reforma
2.- Tulancingo
3.- Apan
4.- Tizayuca
5.- Tepeji del río
6.- Ajacuba
7.- Tenango de doria
8.- Huejutla de reyes
9.- Huasca de ocampo
10.- Actopan
11.- Molango de Escamilla
12.- Tula de Allende
13.- Zimapán
14.- Ixmiquilpan
```

```
Seleccione una opción:
1.- Buscar recorrido más corto
2.- Ver Estaciones
3.- Salir
Ingrese su opción: 3
Saliendo del programa
```

Conclusión

Después de estar practicando el uso de grafos considero que es una herramienta muy útil, sobretodo para aplicaciones actuales donde se busca optimizar el tiempo lo más que se pueda y al estar haciendo este proyecto me dí cuenta de la infinidad de aplicaciones que tiene esta herramienta en distintos rubros de la vida cotidiana, incluyendo servicios de entrega de paquetes, de agua y hasta en aplicaciones de entrega de comida.