UNIVERSIDAD PANAMERICANA

UNIVERSIDAD PANAMERICANA
Campus México

Subject: Programming and Data Structure (COM104)

Professor: Francisco Javier Aguilar Juárez

Due date: May 24th, 2024

Cicle: 1242

Project name:

Navigation System for the Mexico City Metrobús Using the Dijkstra Algorithm in C++

Team members		
ID	**Name (Last name, First name)**	**Career**
0256930	Sáenz Kammermayr Eduardo Alejandro	LIDCI
0251139	Jiménez Reyes Jorge Iván	LIDCI

Index

Introduction

Graphs are a fundamental data structure in programming and computing, used to represent and manipulate complex relationships between objects. They are composed of nodes (or vertices) connected by edges (or links), allowing a wide variety of problems to be modeled, from social networks and route maps to recommendation systems and network analysis. Nowadays, the use of data structures, especially graphs, is crucial for the functioning of many systems.

The implementation of graphs has allowed the development of multiple new technologies and the improvement of existing ones, allowing us to use weight-based search algorithms to locate the fastest route between a source node and a destination node. In this paper, we will talk about a real-life application for graphs.

Problem Description

Context of the problem

The Mexico City metrobus is a public transportation system that covers numerous routes and stations dispersed throughout the city. With the constant growth in demand for the service, efficiency in route planning becomes a critical need for users who want to optimize their travel times.

Problem to solve

Metrobús users often face difficulties when trying to determine the fastest route between two given points within the transportation network. The lack of a tool that provides real-time information on the best possible route complicates trip planning, especially during peak hours or in situations of temporary changes in service.

Specific Objectives

Application Design

Our objective is to create an application in C++ which, through the use of the object-oriented programming paradigm, models the Metrobús route system as a directed graph. Thus allowing us to use an algorithm that shows us the shortest path between two points.

For this, we will implement Dijkstra's algorithm to find the shortest path between any pair of stations, where the graph weight or "distance" can represent time, cost or combinations of both.

Route Optimization

With the use of our application, we want to allow Metrobús users to efficiently calculate the shortest route based on a specific criterion, in our case, the distance between one station and another.

Scalability and Maintenance

Ensure that the application can be easily updated with changes to routes or the addition of new stations and lines.
Design the application so that it can scale to handle a large number of simultaneous users without degrading performance.

Challenges faced

Graph Representation

Specify how the graph that represents all the Metrobús stations and routes will be built and updated, obtaining the data from a JSON file. For this, we use a list of adjacencies to which we assign the nodes and weights.

Algorithm Efficiency

Ensure that Dijkstra's algorithm is efficiently implemented to handle large volumes of data and frequent queries without significant performance impact. Allowing us to obtain routes between different stations and even between different lines.

User interface

Although the current scope focuses only on code implementation, the future inclusion of an intuitive user interface will be crucial for accessibility by end users. This gives you the potential to increase your user base and ensure that everyone has access to such a useful tool.

Design

Data Structures Used

Adjacency List:

Used to represent the graph of Metrobus stations. Each node (station) has a list of (destination, weight) pairs, where weight represents the travel time between stations.

Map (std::unordered_map):

Map that stores the station ID mapping to your name. This allows station IDs to be translated into user-readable names.

Priority Queue (std::priority_queue):

Used in the implementation of Dijkstra's algorithm to select the next closest node that has not yet been processed.

Vector (std::vector):

- dist: Vector that stores the minimum distance from the source station to all other stations.
- prev: Vector that stores the predecessor of each station on the shortest path.

Dijkstra's algorithm

Dijkstra's algorithm is used to find the shortest route from a source station to all other stations. It works iteratively, selecting the station with the current minimum distance, updating the distances to its neighbors, and using a priority queue to manage the stations to process.

Expected Impact

The successful implementation of this application will significantly improve the travel experience of Metrobús users, allowing them to save time and reduce the uncertainty associated with planning their routes, making this experience more pleasant and accurate. In addition, it will provide a technological base on which continuous improvements and new functionalities can be developed in the future.

Implementación

Graph.h

```
#ifndef GRAPH_H
#define GRAPH_H

#include <vector>
#include <unordered_map>
#include <string>
using namespace std;
```

```
class Graph {
public:
    Graph(int numVertices); // Constructor que inicializa el número de vértices
    void addEdge(int src, int dest, int weight); // Añadir una arista con peso
    void addTransfer(int src, int dest, int extraTime); // Añadir una transferencia con tiempo
extra
    std::vector<int> dijkstra(int src); // Algoritmo de Dijkstra para encontrar el camino más
corto desde src
    const std::vector<int>& getPrev() const; // Obtener el vector de predecesores
    void printPathWithNames(const std::vector<int>& prev, int dest, const
std::unordered_map<int, std::string>& stationIDToName) const; // Imprimir el camino más
corto con nombres de estaciones
    void printStationInfo(int stationID, const std::unordered_map<int, std::string>&
stationIDToName) const; // Imprimir información de una estación
    void printConnections(int stationID, const std::unordered_map<int, std::string>&
stationIDToName) const; // Imprimir conexiones de una estación

private:
    int numVertices; // Número de vértices en el grafo
    std::vector<std::vector<std::pair<int, int>>> adjList; // Lista de adyacencia
    std::vector<int> prev; // Vector de predecesores
};

#endif // GRAPH_H
```

Graph.cpp

```
#include "Graph.h"
#include <iostream>
#include <vector>
#include <unordered_map>
#include <queue>
#include <utility>
#include <limits>
#include <algorithm>

// Constructor que inicializa la lista de adyacencia con el número de vértices
Graph::Graph(int numVertices) : numVertices(numVertices) {
    adjList.resize(numVertices);
}

// Método para añadir una arista bidireccional al grafo
```

```
void Graph::addEdge(int src, int dest, int weight) {
   adjList[src].emplace_back(dest, weight); // Añade arista de src a dest
   adjList[dest].emplace_back(src, weight); // Añade arista de dest a src para asegurar
bidireccionalidad
}

// Método para añadir una transferencia bidireccional al grafo
void Graph::addTransfer(int src, int dest, int extraTime) {
   adjList[src].emplace_back(dest, extraTime); // Añade transferencia de src a dest
   adjList[dest].emplace_back(src, extraTime); // Añade transferencia de dest a src para
asegurar bidireccionalidad
}

// Método que implementa el algoritmo de Dijkstra para encontrar el camino más corto
desde src
std::vector<int> Graph::dijkstra(int src) {
   std::vector<int> dist(numVertices, std::numeric_limits<int>::max()); // Inicializa las
distancias a infinito
   prev.resize(numVertices, -1); // Inicializa el vector de predecesores
   dist[src] = 0; // La distancia a la fuente es 0

   using pii = std::pair<int, int>;
   std::priority_queue<pii, std::vector<pii>, std::greater<pii>> pq; // Cola de prioridad para
el algoritmo de Dijkstra
   pq.emplace(0, src); // Añade la fuente a la cola de prioridad

   while (!pq.empty()) {
      int u = pq.top().second; // Obtiene el vértice con la menor distancia
      pq.pop(); // Elimina el vértice de la cola de prioridad

      for (const auto& [v, weight] : adjList[u]) { // Recorre los vecinos del vértice u
         int newDist = dist[u] + weight; // Calcula la nueva distancia

         if (newDist < dist[v]) { // Si la nueva distancia es menor que la distancia actual
            dist[v] = newDist; // Actualiza la distancia
            prev[v] = u; // Actualiza el predecesor
            pq.emplace(dist[v], v); // Añade el vecino a la cola de prioridad
         }
      }
   }

   return dist; // Devuelve el vector de distancias
}
```

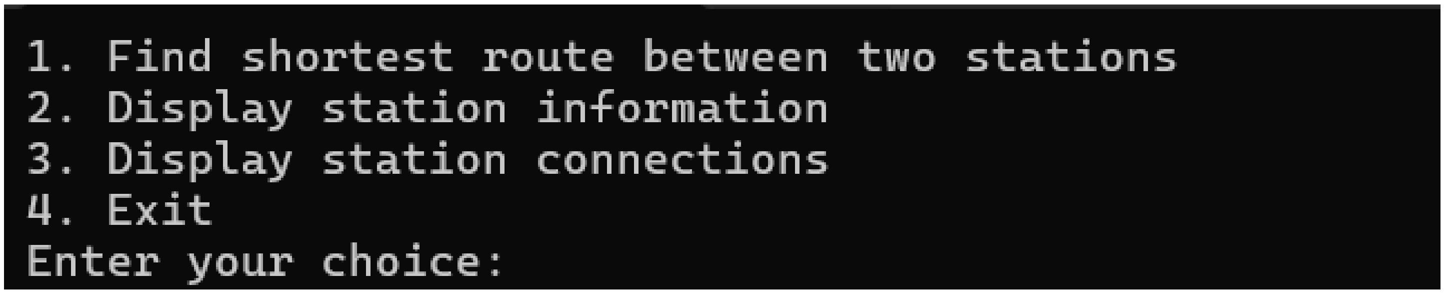
```
// Método para obtener el vector de predecesores
const std::vector<int>& Graph::getPrev() const {
    return prev;
}

// Método para imprimir el camino más corto usando nombres de estaciones
void Graph::printPathWithNames(const std::vector<int>& prev, int dest, const
std::unordered_map<int, std::string>& stationIDToName) const {
    if (dest < 0 || dest >= prev.size() || prev[dest] == -1) { // Verifica si hay un camino válido
        std::cerr << "No route found to the destination." << std::endl;
        return;
    }

    std::vector<int> path; // Vector para almacenar el camino
    for (int at = dest; at != -1; at = prev[at]) { // Recorre el vector de predecesores para
construir el camino
        path.push_back(at);
    }
    std::reverse(path.begin(), path.end()); // Invierte el camino para obtener el orden
correcto

    std::cout << "Shortest route:" << std::endl;
    for (size_t i = 0; i < path.size(); ++i) {
        std::cout << stationIDToName.at(path[i] + 1); // Ajuste para índice basado en 1
        if (i < path.size() - 1) {
            std::cout << " -> ";
        }
    }
    std::cout << std::endl;
}

// Método para imprimir información de una estación
void Graph::printStationInfo(int stationID, const std::unordered_map<int, std::string>&
stationIDToName) const {
    if (stationID < 0 || stationID >= numVertices) {
        std::cerr << "Error: Invalid station ID." << std::endl;
        return;
    }
    std::cout << "Station ID: " << stationID + 1 << std::endl;
    std::cout << "Station Name: " << stationIDToName.at(stationID + 1) << std::endl;
}

// Método para imprimir las conexiones de una estación
```

```
void Graph::printConnections(int stationID, const std::unordered_map<int, std::string>&
stationIDToName) const {
    if (stationID < 0 || stationID >= numVertices) {
        std::cerr << "Error: Invalid station ID." << std::endl;
        return;
    }
    std::cout << "Connections from " << stationIDToName.at(stationID + 1) << ":" <<
std::endl;
    for (const auto& [neighbor, weight] : adjList[stationID]) {
        std::cout << " - To " << stationIDToName.at(neighbor + 1) << " with weight " << weight
<< std::endl;
    }
}
```

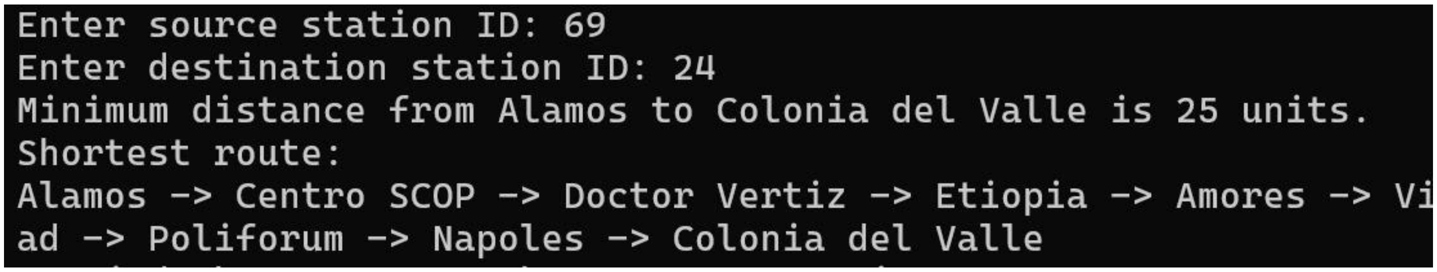
Source.cpp

```
#include "Graph.h"
#include <iostream>
#include <fstream>
#include <nlohmann/json.hpp>
#include <unordered_map>
#include <vector>
#include <string>
#include <algorithm>
#include <limits>

using namespace std;

// Mapa para almacenar el mapeo de ID a nombre de estación
std::unordered_map<int, std::string> stationIDToName;

// Función para cargar el grafo y los nombres de estaciones desde un archivo JSON
void loadGraphFromJSON(Graph& g, const std::string& filename) {
    std::ifstream inFile(filename); // Abre el archivo JSON
    if (!inFile) {
        std::cerr << "Error opening file: " << filename << std::endl;
        return;
    }

    nlohmann::json j;
    try {
        inFile >> j; // Lee el archivo JSON
    }
    catch (const nlohmann::json::parse_error& e) {
        std::cerr << "JSON parse error: " << e.what() << std::endl;
        return;
    }

    if (!j.contains("conexiones") || !j["conexiones"].is_array()) { // Verifica la existencia y el
formato del campo 'conexiones'
        std::cerr << "Invalid JSON structure: 'conexiones' not found or is not an array" <<
std::endl;
        return;
    }

    // Carga las conexiones
    for (const auto& connection : j["conexiones"]) {
        int src = connection["origen"];
        int dest = connection["destino"];
        int time = connection["tiempo"];
        g.addEdge(src - 1, dest - 1, time); // Ajusta para índice basado en 0
    }

    if (!j.contains("transferencias") || !j["transferencias"].is_array()) { // Verifica la existencia y
el formato del campo 'transferencias'
        std::cerr << "Invalid JSON structure: 'transferencias' not found or is not an array" <<
std::endl;
        return;
    }

    // Carga las transferencias
    for (const auto& transfer : j["transferencias"]) {
        int src = transfer["origen"];
        int dest = transfer["destino"];
        int extraTime = transfer["tiempo_extra"];
        g.addTransfer(src - 1, dest - 1, extraTime); // Ajusta para índice basado en 0
    }

    if (!j.contains("estaciones") || !j["estaciones"].is_array()) { // Verifica la existencia y el
formato del campo 'estaciones'
        std::cerr << "Invalid JSON structure: 'estaciones' not found or is not an array" <<
std::endl;
        return;
    }

    // Carga las estaciones
    for (const auto& station : j["estaciones"]) {
        int id = station["id"];
        std::string name = station["nombre"];
        stationIDToName[id] = name; // Almacena el mapeo de ID a nombre
    }
}

// Función para mostrar el menú de opciones
void showMenu() {
    std::cout << "1. Find shortest route between two stations" << std::endl;
    std::cout << "2. Display station information" << std::endl;
    std::cout << "3. Display station connections" << std::endl;
    std::cout << "4. Exit" << std::endl;
    std::cout << "Enter your choice: ";
}

// Función principal
int main() {
    int numStations = 272; // Ajusta según el número real de estaciones
    Graph metrobus(numStations);

    // Carga el grafo y los nombres de las estaciones desde el archivo JSON
automáticamente
    loadGraphFromJSON(metrobus, "metrobus_data_corrected.json");

    bool exit = false;
    while (!exit) {
        showMenu();
        int choice;
        std::cin >> choice;

        switch (choice) {
        case 1: {
            int sourceID, destinationID;
            std::cout << "Enter source station ID: ";
            std::cin >> sourceID;
            std::cout << "Enter destination station ID: ";
            std::cin >> destinationID;

            if (sourceID <= 0 || destinationID <= 0 || sourceID > numStations || destinationID >
numStations) {
                std::cerr << "Error: Invalid station IDs. Please try again." << std::endl;
                continue;
            }
            std::vector<int> minDistances = metrobus.dijkstra(sourceID - 1); // Ejecuta Dijkstra
desde la estación de origen
            std::vector<int> prev = metrobus.getPrev(); // Obtiene el vector de predecesores

            if (minDistances[destinationID - 1] == std::numeric_limits<int>::max()) {
                std::cout << "No route found from " << stationIDToName[sourceID] << " to " <<
stationIDToName[destinationID] << "." << std::endl;
            }
            else {
                std::cout << "Minimum distance from " << stationIDToName[sourceID] << " to "
<< stationIDToName[destinationID] << " is "
                    << minDistances[destinationID - 1] << " units." << std::endl;
                metrobus.printPathWithNames(prev, destinationID - 1, stationIDToName); //
Imprime la ruta más corta con nombres de estaciones
            }
            break;
        }
        case 2: {
            int stationID;
            std::cout << "Enter station ID: ";
            std::cin >> stationID;

            metrobus.printStationInfo(stationID - 1, stationIDToName); // Imprime información
de la estación
            break;
        }
        case 3: {
            int stationID;
            std::cout << "Enter station ID: ";
            std::cin >> stationID;

            metrobus.printConnections(stationID - 1, stationIDToName); // Imprime conexiones
de la estación
            break;
        }
        case 4:
            exit = true;
            std::cout << "Exiting program." << std::endl;
            break;
        default:
            std::cerr << "Invalid choice. Please try again." << std::endl;
            break;
        }
    }
    return 0;
}
```

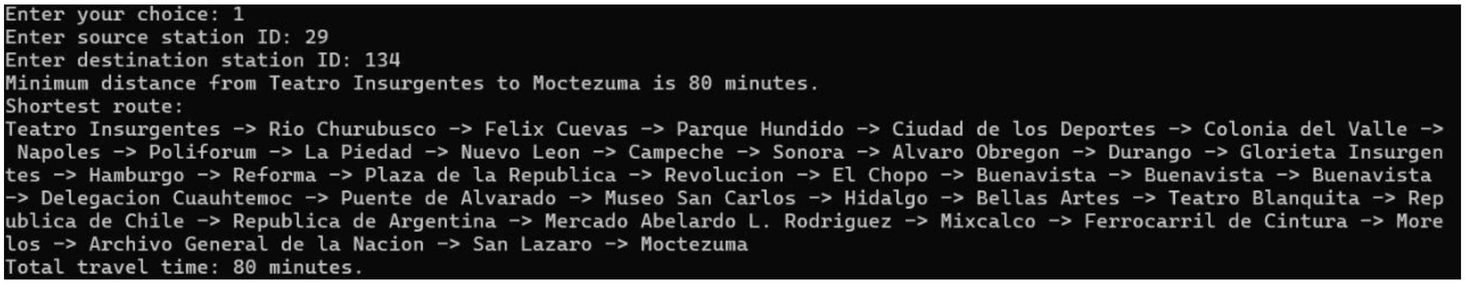
Result Screenshots



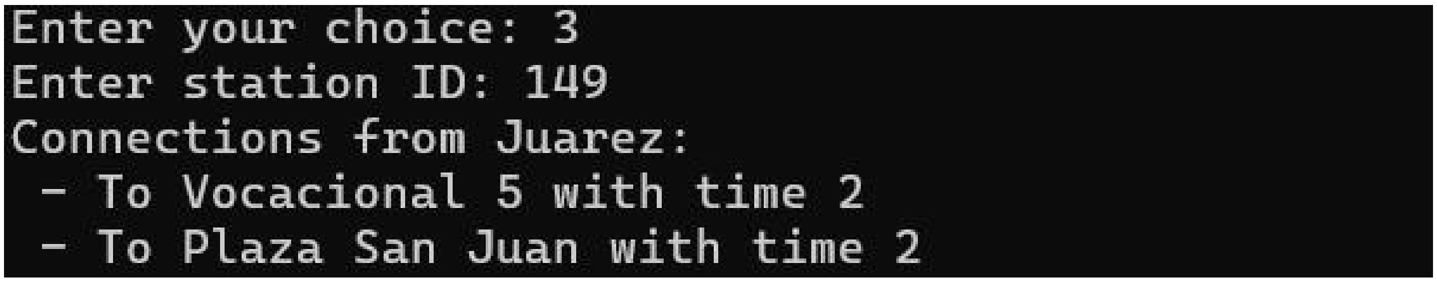
```
1. Find shortest route between two stations
2. Display station information
3. Display station connections
4. Exit
Enter your choice:
```



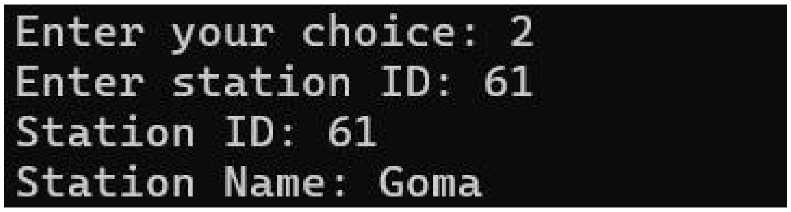
```
Enter source station ID: 69
Enter destination station ID: 24
Minimum distance from Alamos to Colonia del Valle is 25 units.
Shortest route:
Alamos -> Centro SCOP -> Doctor Vertiz -> Etiopia -> Amores -> Vi
ad -> Poliforum -> Napoles -> Colonia del Valle
```



```
Enter your choice: 1
Enter source station ID: 29
Enter destination station ID: 134
Minimum distance from Teatro Insurgentes to Moctezuma is 80 minutes.
Shortest route:
Teatro Insurgentes -> Rio Churubusco -> Felix Cuevas -> Parque Hundido -> Ciudad de los Deportes -> Colonia del Valle ->
 Napoles -> Poliforum -> La Piedad -> Nuevo Leon -> Campeche -> Sonora -> Alvaro Obregon -> Durango -> Glorieta Insurgen
tes -> Hamburgo -> Reforma -> Plaza de la Republica -> Revolucion -> El Chopo -> Buenavista -> Buenavista -> Buenavista
-> Delegacion Cuauhtemoc -> Puente de Alvarado -> Museo San Carlos -> Hidalgo -> Bellas Artes -> Teatro Blanquita -> Rep
ublica de Chile -> Republica de Argentina -> Mercado Abelardo L. Rodriguez -> Mixcalco -> Ferrocarril de Cintura -> More
los -> Archivo General de la Nacion -> San Lazaro -> Moctezuma
Total travel time: 80 minutes.
```



```
Enter your choice: 3
Enter station ID: 149
Connections from Juarez:
 - To Vocacional 5 with time 2
 - To Plaza San Juan with time 2
```



```
Enter your choice: 2
Enter station ID: 61
Station ID: 61
Station Name: Goma
```

User Guide

Start

When you run the program, the “metrobus_data_corrected.json” file will be automatically loaded, which contains all the data of the Metrobús stations and routes.

Main Menu

You will be asked to select an option from the menu:

1. Find shortest route: To find the shortest route between two stations.
2. View station information: To view information for a specific station.
3. View station connections: To view the connections of a specific station.
4. Exit: To exit the program.

Find Shortest Route

1. Enter the source station ID.
2. Enter the destination station ID.

The program will show the minimum distance and the detailed route with the names of the stations necessary to cross to reach the desired destination.

View Station Information

Enter the station ID to see its name. This can be expanded to show more specific details such as the line, neighboring stations and even estimated arrival times for upcoming metrobuses.

See Station Connections

Enter the station ID to see available connections and travel time between connected stations.