UNIVERSIDAD PANAMERICANA
Campus México

UNIVERSIDADPANAMERICANA

Materia: Programación y Estructura de Datos (COM104)

Profesor: Francisco Javier Aguilar Juárez

Fecha de entrega: 24 de mayo de 2024

Ciclo: 1242

Nombre del proyecto:

Sistema de Navegación para el Metrobús de la Ciudad de México Utilizando el Algoritmo de Dijkstra en C++

Miembros del Equipo		
ID	**Nombre (ap.Paterno, ap.Materno, nombre)**	**Carrera**
0256930	Sáenz Kammermayr Eduardo Alejandro	LIDCI
0251139	Jiménez Reyes Jorge Iván	LIDCI

Índice

Introducción
…… 2

Descripción del problema
…… 2

- Contexto del problema
…… 2

- Problema a resolver
…….. 2

Objetivos específicos
…… 3

- Desarrollo de la aplicación
…… 3

- Optimización de rutas
….. 3

- Escalabilidad y mantenimiento
…… 3

Desafíos enfrentados
…… 3

- Representación del grafo
…….. 3

- Eficiencia del algoritmo
…….. 4

- Interfaz de usuario
……. 4

Diseño
..... 4
- Estructuras de datos empleadas
..... 4
- Algoritmo de Dijkstra
..... 5

Impacto esperado
..... 5

Implementación
..... 5
- Graph.h
..... 5
- Graph.cpp
..... 6
- Source.cpp
..... 9

Capturas de resultados
13

Guía del usuario
..... 14
- Inicio
..... 14
- Menú principal
..... 14

- Encontrar la ruta más corta …….. 14
- Ver información de la estación …… 14
- Ver conexiones de la estación ……. 14

Introducción

Los grafos son una estructura de datos fundamental en la programación y la informática, utilizados para representar y manipular relaciones complejas entre objetos. Se componen de nodos (o vértices) conectados por aristas (o enlaces), permitiendo modelar una amplia variedad de problemas, desde redes sociales y mapas de rutas hasta sistemas de recomendación y análisis de redes. Hoy en día, la utilización de estructuras de datos, especialmente grafos, es crucial para el funcionamiento de muchos sistemas.

La implementación de grafos ha permitido el desarrollo de múltiples tecnologías nuevas y la mejora de otras existentes permitiéndonos usar algoritmos de búsqueda basados en peso para localizar la ruta más rápida entre un nodo de origen y uno de destino. En este trabajo, hablaremos acerca de una aplicación de la vida real para los grafos.

Descripción del problema

Contexto del problema

El metrobús de la Ciudad de México es un sistema de transporte público que abarca numerosas rutas y estaciones dispersas a lo largo de toda la ciudad. Con el crecimiento constante en la demanda del servicio, la eficiencia en la planificación de rutas se convierte en una necesidad crítica para los usuarios que desean optimizar sus tiempos de viaje.

Problema a resolver

Los usuarios del Metrobús a menudo enfrentan dificultades al tratar de determinar la ruta más rápida entre dos puntos dados dentro de la red de transporte. La falta de una herramienta que ofrezca información en tiempo real sobre la mejor ruta posible complica la planificación del viaje, especialmente durante las horas pico o en situaciones de cambios temporales en el servicio.

Objetivos Específicos

Desarrollo de la aplicación

Nuestro objetivo es crear una aplicación en C++ la cual, mediante la utilización del paradigma de programación orientada a objetos, modele el sistema de rutas del Metrobús como un grafo dirigido. Permitiéndonos así, utilizar un algoritmo que nos muestre el camino más corto entre dos puntos.

Para esto, implementaremos el algoritmo de Dijkstra para encontrar la ruta más corta entre cualquier par de estaciones, donde el peso del grafo o la "distancia" puede representar tiempo, coste o combinaciones de ambos.

Optimización de Rutas

Con el uso de nuestra aplicación, queremos permitirle a los usuarios del Metrobús calcular eficientemente la ruta más corta basada en un criterio específico, en nuestro caso, la distancia que existe entre una estación y otra.

Escalabilidad y Mantenimiento

Garantizar que la aplicación pueda actualizarse fácilmente con cambios en las rutas o la adición de nuevas estaciones y líneas.
Diseñar la aplicación de manera que pueda escalar para manejar un gran número de usuarios simultáneos sin degradar el rendimiento.

Desafíos enfrentados

Representación del Grafo

Precisar cómo se construirá y actualizará el grafo que representa todas las estaciones y rutas del Metrobús, obteniendo los datos de un archivo JSON. Para esto, utilizamos una lista de adyacencias a la cual asignamos los nodos y los pesos.

Eficiencia del Algoritmo

Asegurar que el algoritmo de Dijkstra se implemente de manera eficiente para manejar grandes volúmenes de datos y consultas frecuentes sin un impacto significativo en el rendimiento. Permitiéndonos obtener rutas entre distintas estaciones e incluso entre distintas líneas.

Interfaz de Usuario

Aunque el alcance actual se centra solamente en la implementación del código, la futura inclusión de una interfaz de usuario intuitiva será crucial para la accesibilidad por parte de los usuarios finales. Esto le da el potencial de incrementar su base de usuarios y que todos tengan acceso a tan útil herramienta.

Diseño

Estructuras de Datos Empleadas

Lista de Adyacencia:

Utilizada para representar el grafo de las estaciones del Metrobus. Cada nodo (estación) tiene una lista de pares (destino, peso), donde peso representa el tiempo de viaje entre las estaciones.

Mapa (std::unordered_map):

Mapa que almacena el mapeo de ID de estación a su respectivo nombre. Esto permite traducir los IDs de las estaciones a nombres legibles para el usuario.

Cola de Prioridad (std::priority_queue):

Utilizada en la implementación del algoritmo de Dijkstra para seleccionar el siguiente nodo más cercano que aún no ha sido procesado.

Vector (std::vector):

- dist: Vector que almacena la distancia mínima desde la estación de origen a todas las demás estaciones.
- prev: Vector que almacena el predecesor de cada estación en el camino más corto.

Algoritmo de Dijkstra

El algoritmo de Dijkstra es empleado para encontrar la ruta más corta desde una estación de origen a todas las demás estaciones. Funciona iterativamente, seleccionando la estación con la distancia mínima actual, actualizando las distancias a sus vecinos y utilizando una cola de prioridad para gestionar las estaciones a procesar.

Impacto Esperado

La implementación exitosa de esta aplicación mejorará significativamente la experiencia de viaje de los usuarios del Metrobús, permitiéndoles ahorrar tiempo y reducir la incertidumbre asociada a la planificación de sus rutas, haciendo esta experiencia algo más agradable y precisa. Además, proporcionará una base tecnológica sobre la cual se podrán desarrollar mejoras continuas y nuevas funcionalidades a futuro.

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;
}
```

Capturas de Resultados

```
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
```

Guía del Usuario

Inicio

Al ejecutar el programa, se cargará automáticamente el archivo “metrobus_data_corrected.json” el cual contiene todos los datos de las estaciones y rutas del Metrobús.

Menú Principal

Se le pedirá que seleccione una opción del menú:

1. Find shortest route: Para encontrar la ruta más corta entre dos estaciones.
2. View station information: Para ver información de una estación específica.
3. View station connections: Para ver las conexiones de una estación específica.
4. Exit: Para salir del programa.

Encontrar la Ruta Más Corta

1. Ingrese el ID de la estación de origen.
2. Ingrese el ID de la estación de destino.

El programa mostrará la distancia mínima y la ruta detallada con los nombres de las estaciones necesarias de cruzar para llegar al destino deseado.

Ver Información de la Estación

Ingresa el ID de la estación para ver su nombre. Esto puede ampliarse para mostrar detalles más específicos como la línea, estaciones colindantes e incluso tiempos estimados de llegada para los próximos metrobuses.

Ver Conexiones de la Estación

Ingresa el ID de la estación para ver las conexiones disponibles y el tiempo de viaje entre las estaciones conectadas.