UNIVERSIDAD PANAMERICANA

Programación y Estructura de Datos

SISTEMA DE NAVEGACIÓN PARA EL METROBÚS DE LA CIUDAD DE MÉXICO

Utilizando el Algoritmo de Dijkstra en C++

Fecha:

24/05/24

Presentado por:

Eduardo Alejandro Sáenz Kammermayr
Jorge Iván Jiménez Reyes

AGENDA

01 Contexto

02 Problema

03 Diseño

04 Desafíos

05 Algoritmo de Dijkstra

06 Estructuras empleadas

07 Implementación

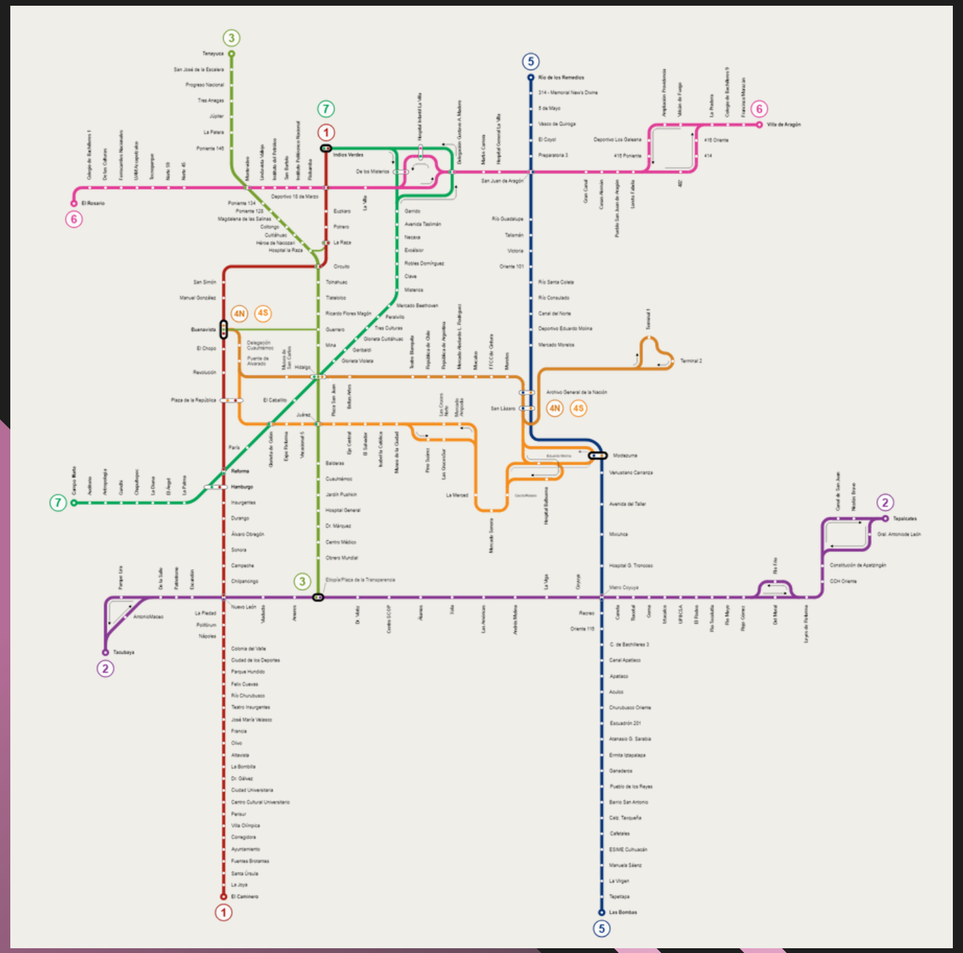
08 Resultados

UNIVERSIDAD PANAMERICANA

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 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.

UNIVERSIDAD PANAMERICANA



Problema

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.

UNIVERSIDAD PANAMERICANA

Diseño

DESARROLLO

Crear una aplicación en C++ que modele el sistema de rutas del Metrobús como un grafo dirigido e implementar el algoritmo de Dijkstra para encontrar la ruta más corta entre cualquier par de estaciones, donde la "distancia" puede representar tiempo, coste o combinaciones de ambos.

OPTIMIZACIÓN

Permitir a los usuarios del Metrobús calcular eficientemente la ruta más corta hacia su destino basada en un criterio específico como el menor número de paradas, el menor tiempo de viaje o la saturación del sistema.

ESCALABILIDAD

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.

UNIVERSIDAD PANAMERICANA

Desafíos

01

REPRESENTACIÓN DEL GRAFO

Precisar cómo se construirá y actualizará el grafo que representa todas las estaciones y rutas disponibles del Metrobús.

02

INTERFAZ DE USUARIO

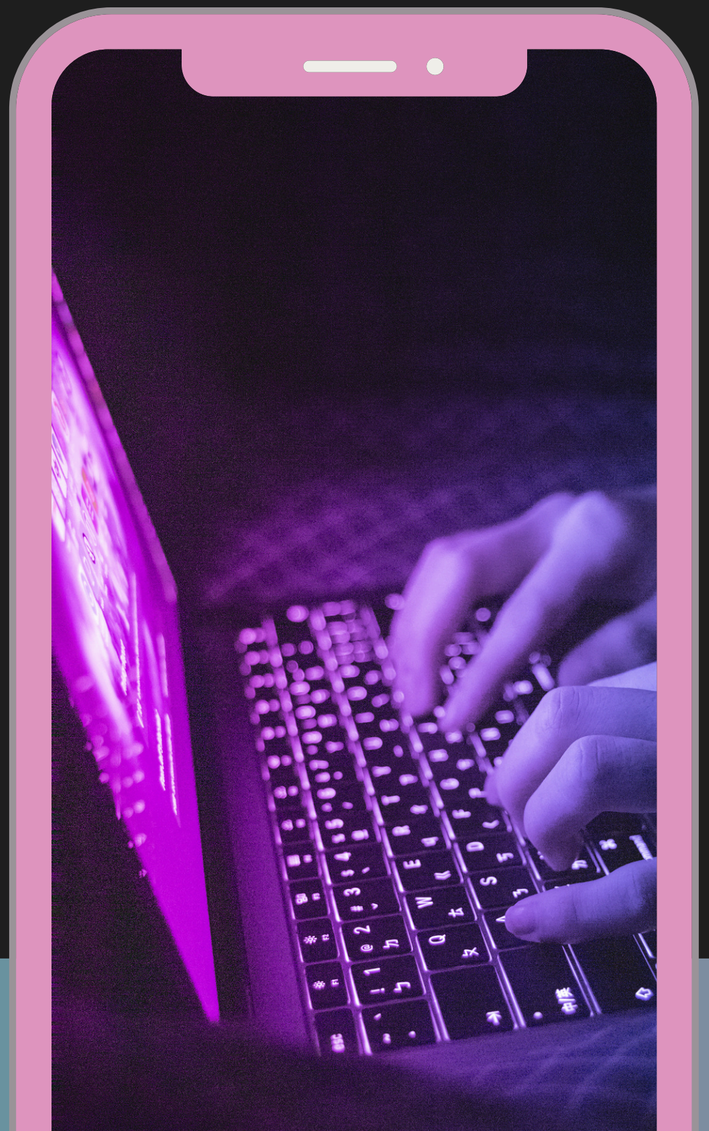
Aunque el alcance actual se centra en la implementación del código, la futura inclusión de una interfaz de usuario intuitiva será crucial para la accesibilidad.

03

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.

UNIVERSIDAD PANAMERICANA



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.

• USOS

Sirve para encontrar la ruta más corta entre los nodos de un grafo. Específicamente, busca el camino más corto desde un nodo a todos los otros nodos del grafo.

• VENTAJAS

Puede aplicarse tanto a grafos dirigidos como no dirigidos.
Tiene una complejidad de tiempo eficiente, especialmente en grafos con pesos no negativos.

UNIVERSIDAD PANAMERICANA

Estructuras de datos empleadas

GRAFO

El uso de grafos es primordial para la correcta implementación de nuestra aplicación, pues nos facilita la busqueda de la ruta mas optima en el menor tiempo.

COLA

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

MAPA

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

VECTOR

Un vector que almacena la distancia mínima desde la estación de origen a todas las demás estaciones.

Otro vector que almacena el predecesor de cada estación en el camino más corto.

UNIVERSIDAD PANAMERICANA

Implementación

UNIVERSIDAD PANAMERICANA

Grafo

Representado mediante una lista de adyacencia (adjList), que es un vector de vectores de pares (std::vector<std::vector<std::pair<int, int>>>)

Lista de Adyacencias

```
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
```

Métodos relevantes

```
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>&
    void printStationInfo(int stationID, const std::unordered_map<int, std::string>& stationIDToName) const; //
    void printConnections(int stationID, const std::unordered_map<int, std::string>& stationIDToName) const; //
```

UNIVERSIDAD PANAMERICANA

Cola de Prioridad (Priority Queue)

Dentro del Método de Dijkstra

```cpp
std::vector<int> Graph::dijkstra(int src) {
    std::vector<int> dist(numVertices, std::numeric_limits<int>::max()); // Inicializa las distancias a in
    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 algorit
    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
}
```

UNIVERSIDAD PANAMERICANA

Declaración del Mapa e Impresión de Resultados

```
std::unordered_map<int, std::string> stationIDToName;
```

```
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;
}
```

UNIVERSIDAD PANAMERICANA

Uso del Mapa en Funciones

```
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;
}

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;
    }
}
```

UNIVERSIDAD PANAMERICANA

Vectores (Distancias y Predecesores en Dijkstra)

```
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
}
```

Almacenamiento Temporal de Caminos

```
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;
}
```

MUCHAS GRACIAS