

Universidad de Costa Rica  
Facultad de Ingeniería  
Escuela de Ingeniería Eléctrica  
IE 0217 - Estructuras Abstractas de Datos y Algoritmos para Ingeniería  
III ciclo 2024

Proyecto  
Sistema de Reserva de Vuelos

Alvarado Taleno Edgar C10351  
Fonseca Zumbado Daniela B93070  
Rojas Álvarez Cristhian C16760  
Narajo Arias Carlos B44870  
Grupo 01

Profesor: Karen Tovar Parra

03 de marzo del 2025

## Resumen

El Sistema de Reserva de Vuelos es una aplicación en C++ y Qt que permite gestionar rutas aéreas mediante un grafo dirigido ponderado. Utiliza el algoritmo de Dijkstra para calcular la ruta más corta entre ciudades, almacenando los datos en archivos CSV para garantizar la persistencia. La interfaz gráfica facilita la interacción de usuarios y administradores, permitiendo la consulta de rutas y la gestión de la red de vuelos. Además, se implementó una visualización gráfica del grafo. Finalmente, se proponen mejoras como optimización del algoritmo, integración con mapas reales y mejoras en la visualización.

## 1. Introducción

El Sistema de Reserva de Vuelos es una aplicación desarrollada en C++ utilizando el framework Qt, diseñada para ofrecer una solución eficiente en la planificación de rutas aéreas. Su objetivo principal es permitir a los usuarios calcular la ruta más corta entre dos ciudades y brindar a los administradores la capacidad de gestionar la red de vuelos de manera intuitiva.

El sistema está basado en la representación de un grafo, donde las ciudades funcionan como vértices y las conexiones entre ellas como aristas con pesos que representan las distancias. Para calcular la ruta óptima entre dos destinos, el programa implementa el algoritmo de Dijkstra, asegurando que los usuarios obtengan siempre la opción más eficiente de viaje.

Además de la funcionalidad de cálculo de rutas, el sistema proporciona una interfaz gráfica interactiva que permite visualizar un mapa con todas las ciudades registradas y sus respectivas conexiones o un submapa con la ruta más corta. Esta funcionalidad no solo ayuda a comprender la estructura de la red de vuelos, sino que también facilita la gestión de rutas por parte de los administradores.

La persistencia de datos se maneja mediante archivos CSV, lo que permite guardar y cargar el estado del grafo dinámicamente. Esto garantiza que las modificaciones realizadas, como agregar o eliminar ciudades y conexiones, sean almacenadas correctamente y estén disponibles en futuras ejecuciones del programa.

Este documento presenta el desarrollo del sistema, detallando su arquitectura, algoritmos utilizados y funcionalidades implementadas, así como los resultados obtenidos y las posibles mejoras futuras.

## 2. Descripción del Proyecto

Este proyecto se basa en la gestión de una red de vuelos representada mediante un grafo, donde las ciudades actúan como *nodos* y las conexiones entre ellas como *aristas ponderadas*, cuyo peso representa la distancia entre los destinos. La aplicación permite a los usuarios calcular rutas óptimas entre ciudades y a los administradores gestionar la red de vuelos de manera dinámica a través de una interfaz gráfica desarrollada en Qt.

- **GitHub:** <https://github.com/EdgarPrueba/Sistema-de-Reserva-de-Vuelos.git>

### 2.1. Componentes Principales

#### 2.1.1. Grafo

El grafo es la estructura de datos fundamental del sistema. Se utiliza para modelar las ciudades y sus conexiones mediante la siguiente implementación:

- Los nodos representan las ciudades.
- Las aristas ponderadas representan las conexiones entre ciudades con un valor que indica la distancia en kilómetros.
- Se implementa usando una estructura de datos basada en un `unordered_map<string, unordered_map<string, double>>`, lo que permite un acceso eficiente a los nodos y sus conexiones.

### 2.1.2. Algoritmo de Dijkstra

Para determinar la ruta más corta entre dos ciudades, se implementa el algoritmo de Dijkstra, el cual funciona de la siguiente manera:

- Se utiliza una cola de prioridad (min-heap) para seleccionar el nodo con la menor distancia acumulada.
- Se exploran las conexiones disponibles y se actualizan las distancias mínimas registradas.
- Una vez finalizada la ejecución, se obtiene la ruta óptima entre los dos puntos seleccionados.

### 2.1.3. Persistencia de Datos

Para garantizar la conservación de la información entre ejecuciones del programa, se emplea la clase `FileManager`, la cual permite leer y escribir en archivos CSV. Los datos almacenados incluyen:

- Lista de ciudades registradas en el sistema.
- Conexiones entre ciudades junto con sus respectivas distancias.

### 2.1.4. Interfaz Gráfica

El sistema cuenta con una interfaz gráfica desarrollada en Qt, la cual permite la interacción con el usuario de manera intuitiva. Sus funcionalidades incluyen:

- Selección de ciudades para calcular la ruta óptima y visualización del subgrafo.
- Visualización del grafo con todas las ciudades y conexiones disponibles.
- Gestión de la red de vuelos mediante opciones para agregar o eliminar ciudades y conexiones.

## 2.2. Funcionalidades del Sistema

El sistema está dividido en dos módulos principales:

### 2.2.1. Módulo de Usuario

Los usuarios pueden:

- Calcular la ruta más corta entre dos ciudades mediante el algoritmo de Dijkstra.
- Visualizar la ruta más corta con las conexiones en un mapa interactivo
- Visualizar el grafo de conexiones en un mapa interactivo.

### 2.2.2. Módulo de Administrador

Los administradores tienen la capacidad de:

- Agregar o eliminar ciudades dentro del sistema.
- Definir nuevas conexiones entre ciudades, especificando la distancia correspondiente.
- Eliminar conexiones existentes entre ciudades.

## 3. Desarrollo

Esta sección describe la implementación del proyecto, detallando las decisiones de diseño y las estructuras empleadas para el sistema de reservas de vuelos, que permite gestionar y consultar rutas entre ciudades de manera eficiente.

### 3.1. Estructuras de Datos

#### 3.1.1. Grafo

El sistema de reservas modela la red de vuelos entre ciudades usando un grafo dirigido ponderado. En este grafo:

- Los nodos representan las ciudades del sistema.
- Las aristas ponderadas indican las rutas de vuelo entre las ciudades, con un peso asociado que representa la distancia en kilómetros.

Este grafo se implementa con un `unordered_map` de C++, lo cual permite un acceso eficiente a las conexiones entre las ciudades. La definición de la estructura de datos del grafo se presenta en el [8.1](#)

#### 3.1.2. Cola de Prioridad

El algoritmo de Dijkstra se basa en la necesidad de seleccionar el nodo con la menor distancia acumulada en cada paso. Para lograr esto de manera eficiente, se utiliza una cola de prioridad implementada con un `priority_queue` de C++, que mantiene los nodos ordenados en función de la distancia mínima.

Esta estructura garantiza que siempre se procesen primero los nodos más cercanos al origen, lo que optimiza la ejecución del algoritmo.

### 3.2. Algoritmos

#### 3.2.1. Algoritmo de Dijkstra

El algoritmo de Dijkstra calcula la ruta más corta entre dos ciudades utilizando la siguiente estrategia:

1. Inicialización: Se establece la distancia de la ciudad de origen a cero y las demás a infinito.
2. Cola de Prioridad: Se utiliza una cola de prioridad para seleccionar el nodo con la menor distancia acumulada.

3. Actualización de distancias: Para cada ciudad seleccionada, se actualizan las distancias a las ciudades vecinas si se encuentra una ruta más corta.
4. Iteración: Este proceso se repite hasta que se procesen todos los nodos o se llegue a la ciudad destino.
5. Reconstrucción de la ruta: Una vez procesado el destino, se reconstruye el camino más corto desde el destino hasta el origen.

Este algoritmo garantiza encontrar el camino más corto en un grafo ponderado, asegurando eficiencia en su ejecución, incluso para grandes conjuntos de ciudades. El código correspondiente al algoritmo de Dijkstra está disponible en el [8.2](#).

### 3.2.2. Persistencia de Datos

El sistema utiliza archivos CSV para almacenar y recuperar la información sobre las ciudades y sus conexiones. La clase `FileManager` maneja la lectura y escritura de estos archivos, permitiendo que la estructura de datos se conserve entre ejecuciones del programa.

Al cargar los datos desde los archivos, se construye el grafo de acuerdo con las ciudades y rutas representadas en el archivo CSV, y cualquier modificación realizada se guarda de vuelta al archivo para mantener la persistencia de la información. El código relacionado con la gestión de archivos CSV está documentado en el [8.3](#).

## 3.3. Interfaz Gráfica

La interfaz gráfica está desarrollada en Qt, proporcionando una experiencia de usuario intuitiva y eficiente. Esta interfaz permite a los usuarios gestionar las ciudades y rutas de vuelo de manera fácil y visual. Se compone de:

- Ventanas: Secciones separadas para el modo de usuario y el modo de administrador.
- Botones: Facilitan la interacción, permitiendo realizar consultas sobre rutas, así como agregar o eliminar ciudades y conexiones.
- Combobox: Un menú desplegable que permite seleccionar las ciudades disponibles.
- Gráficos: Una representación visual del grafo, donde cada ciudad se muestra como un nodo y cada conexión como una arista ponderada.

### 3.3.1. Visualización del Grafo

El módulo de gráficos de Qt se utiliza para representar visualmente las ciudades y las rutas entre ellas. Cada ciudad se dibuja como un círculo, y las conexiones entre ciudades se muestran como líneas, con etiquetas que indican la distancia entre ellas. La interfaz permite interactuar con el grafo, facilitando la visualización tanto de las rutas específicas como del mapa completo. El código que describe la visualización gráfica está incluido en el [8.4](#).

## 3.4. Flujo de Trabajo

### 3.4.1. Modo Usuario

En el modo de usuario, el flujo de trabajo es el siguiente:

1. El usuario selecciona una ciudad de origen y una ciudad de destino desde la interfaz gráfica.
2. El sistema ejecuta el algoritmo de Dijkstra para encontrar la ruta más corta entre ambas ciudades.
3. La ruta más corta y su distancia total se muestran en un mapa interactivo.
4. El mapa completo de todas las conexiones de vuelos también puede ser visualizado para un análisis general de la red.

### 3.4.2. Modo Administrador

En el modo administrador, las funcionalidades son las siguientes:

1. El administrador puede agregar o eliminar ciudades en el sistema.
2. También tiene la capacidad de definir o eliminar conexiones entre ciudades, especificando las distancias entre ellas.
3. Todos los cambios realizados se guardan automáticamente en los archivos CSV para asegurar que los datos sean persistentes.

## 4. Funcionamiento del Programa

En esta sección se detalla el uso del programa, incluyendo los requisitos de instalación, la guía de uso para usuarios y administradores, así como la visualización del grafo de conexiones.

### 4.1. Requisitos de Uso

Antes de ejecutar el programa, es necesario contar con Qt Creator instalado. Para sistemas Linux, se puede instalar con los siguientes comandos:

Listing 1: Instalación de Qt en Linux

```
1 sudo apt update
2 sudo apt install qtcreator qtbase5-dev
```

Alternativamente, puedes referirte a esta guía para instalar el programa: <https://doc.qt.io/qt-5/qt-online-installation.html>

### 4.2. Guía de Uso

#### 4.2.1. Iniciar el proyecto en Qt Creator

Para ejecutar el programa, primero es necesario abrir el archivo del proyecto en Qt Creator:

1. Abrir Qt Creator.
2. Hacer clic en la opción Open.
3. Buscar el archivo `interfaz.pro` en la siguiente ubicación:

## Listing 2: Abrir el archivo de proyecto

```
1 ../ Sistema-de-Reserva-de-Vuelos/src/gui/interfaz.pro
```

4. Ejecutar el programa presionando el botón de reproducción en la esquina inferior izquierda.

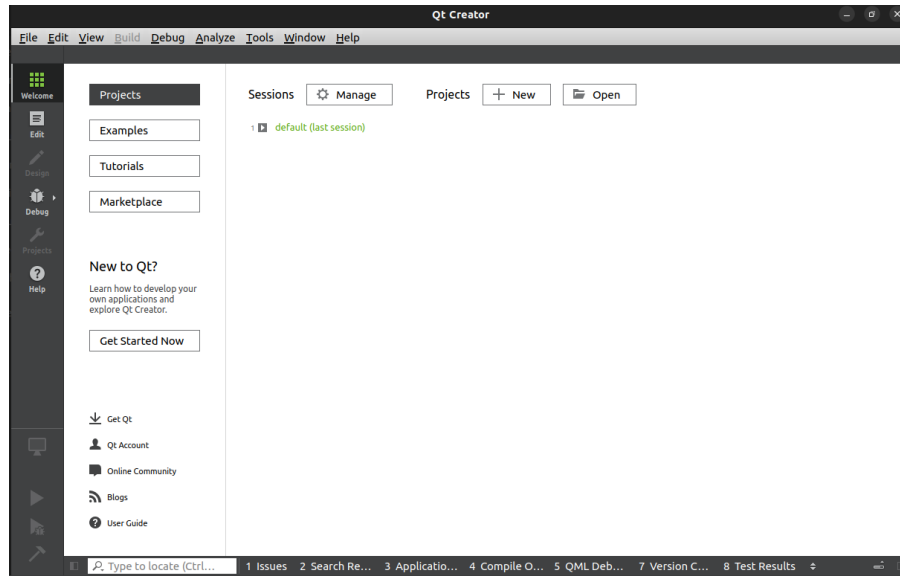


Figura 1: Pantalla inicial de Qt Creator

## 4.3. Acceso de Usuario

Los usuarios pueden calcular la mejor ruta entre dos ciudades y visualizar las conexiones en un mapa interactivo.

### 4.3.1. Reserva de Vuelos

1. Seleccionar la ciudad de origen y la ciudad de destino.
2. El sistema calculará la mejor ruta utilizando el algoritmo de Dijkstra.
3. Se mostrará la información de la ruta, incluyendo:
  - Ciudad de origen.
  - Ciudades intermedias (si las hay).
  - Ciudad de destino.

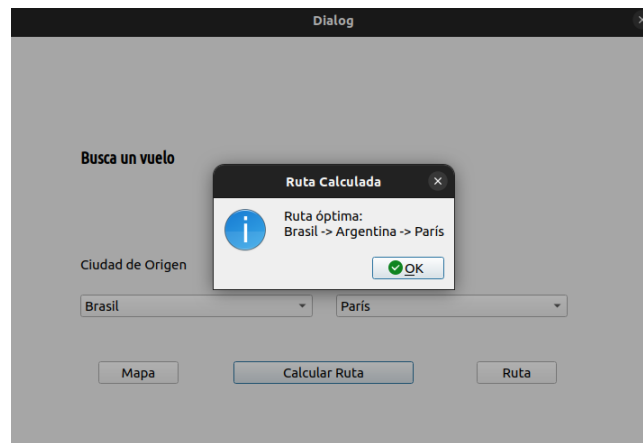


Figura 2: Interfaz de usuario para la reserva de vuelos

#### 4.3.2. Visualización del Grafo

El usuario puede visualizar el grafo de conexiones con las siguientes opciones:

- Mapa General: Muestra todas las ciudades y conexiones disponibles.
- Ruta Calculada: Permite visualizar únicamente la ruta óptima calculada.

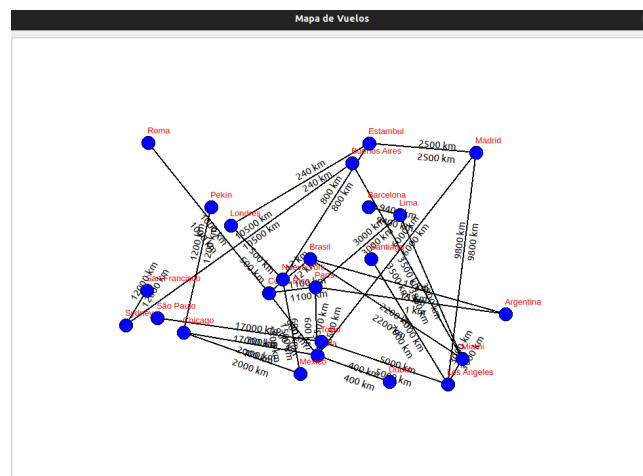


Figura 3: Mapa general de conexiones



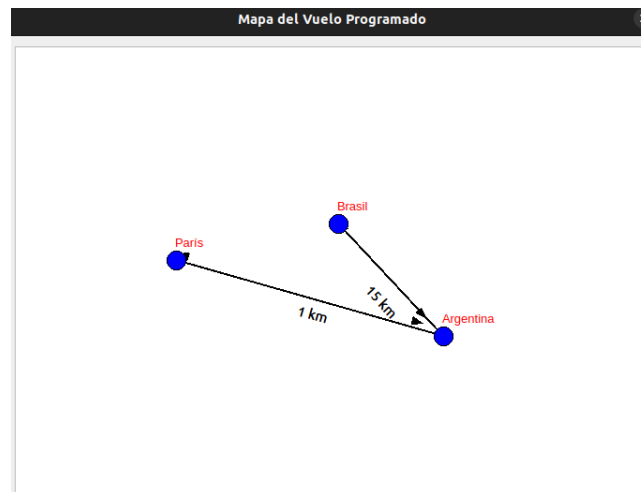


Figura 4: Ruta óptima calculada en el grafo

## 4.4. Acceso de Administrador

El administrador tiene la capacidad de gestionar las ciudades y conexiones del sistema.

### 4.4.1. Funciones del Administrador

- Agregar una ciudad: Se añade una nueva ciudad al sistema.
- Eliminar una ciudad: Se remueve una ciudad junto con todas sus conexiones.
- Establecer una conexión: Se define un nuevo vuelo entre dos ciudades con una distancia específica.
- Eliminar una conexión: Se elimina una ruta de vuelo entre dos ciudades.

The figure shows a "Dialog" window with two main sections: "Gestionar Ciudades" and "Gestionar Conexiones".

**Gestionar Ciudades:**

- Lista de Ciudades: A dropdown menu showing "Tokio".
- Ingrese una Ciudad para agregar: A text input field.
- Buttons: "Agregar" and "Eliminar".

**Gestionar Conexiones:**

- Ciudad Origen: A dropdown menu showing "Tokio".
- Ciudad Destino: A dropdown menu showing "Tokio".
- Distancia (km): A text input field.
- Buttons: "Agregar" and "Eliminar".

Figura 5: Interfaz de administrador

### 4.4.2. Agregar una Nueva Ciudad

1. Ingresar el nombre de la ciudad en la caja de texto.
2. Presionar el botón Agregar.
3. Confirmar la acción.

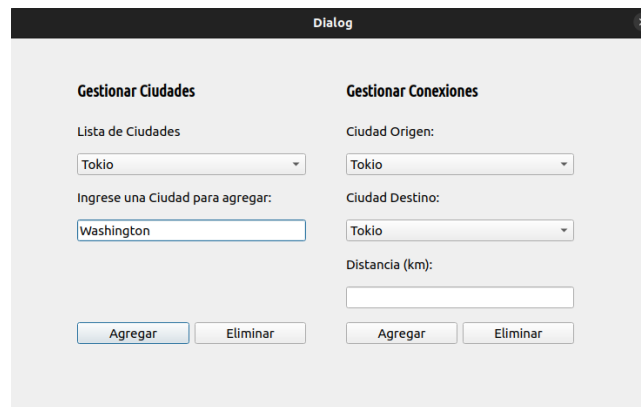


Figura 6: Agregar una nueva ciudad

Una vez agregada, la ciudad aparecerá en la lista de selección.

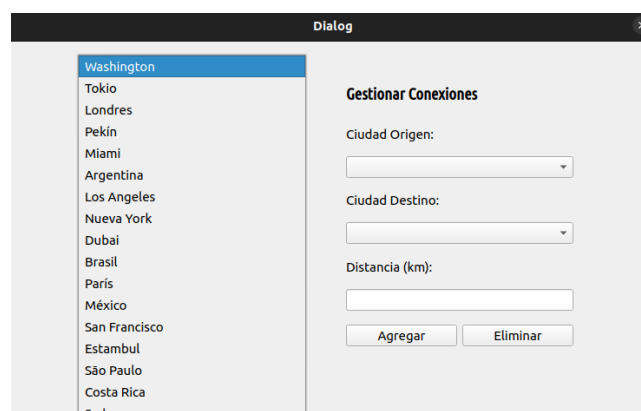


Figura 7: Confirmación de la nueva ciudad agregada

#### 4.4.3. Eliminar una Ciudad

1. Seleccionar la ciudad en el menú desplegable.
2. Presionar el botón Eliminar.
3. Confirmar la eliminación.

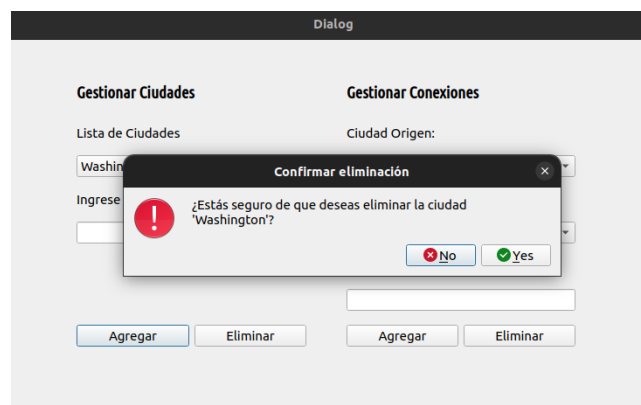
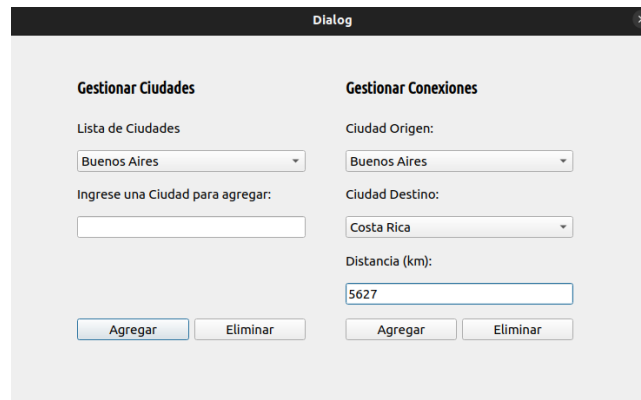


Figura 8: Eliminar una ciudad

#### 4.4.4. Agregar una Conexión

1. Seleccionar la Ciudad Origen y la Ciudad Destino.
2. Ingresar la distancia en la caja de texto.
3. Presionar el botón Agregar.
4. Confirmar la acción.

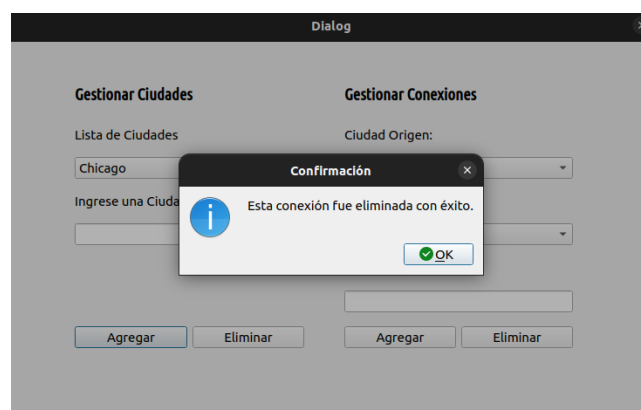


The screenshot shows a 'Dialog' window with two panels: 'Gestionar Ciudades' and 'Gestionar Conexiones'. In the 'Gestionar Ciudades' panel, the 'Lista de Ciudades' dropdown is set to 'Buenos Aires', and there is an empty text input for 'Ingrese una Ciudad para agregar:'. In the 'Gestionar Conexiones' panel, 'Ciudad Origen:' is 'Buenos Aires', 'Ciudad Destino:' is 'Costa Rica', and 'Distancia (km):' is '5627'. Both panels have 'Agregar' and 'Eliminar' buttons at the bottom.

Figura 9: Agregar una conexión entre ciudades

#### 4.4.5. Eliminar una Conexión

1. Seleccionar la Ciudad Origen y la Ciudad Destino.
2. Presionar el botón Eliminar.
3. Confirmar la eliminación.



The screenshot shows the 'Dialog' window with the 'Gestionar Ciudades' panel set to 'Chicago'. A 'Confirmación' dialog box is overlaid in the center, displaying an information icon and the text 'Esta conexión fue eliminada con éxito.' with an 'OK' button. The 'Gestionar Conexiones' panel is partially visible behind the dialog.

Figura 10: Eliminar una conexión existente

## 5. Conclusiones

El proyecto del Sistema de Reserva de Vuelos ha logrado cumplir con sus objetivos principales, proporcionando una solución eficiente y funcional para la gestión y consulta de rutas de vuelo. A continuación, se destacan los principales logros:

## 5.1. Logros del Proyecto

- Implementación exitosa del grafo: Se modeló la red de conexiones entre ciudades utilizando un grafo dirigido ponderado, lo que permitió representar de manera eficiente las rutas de vuelo y sus distancias.
- Algoritmo de Dijkstra: Se implementó correctamente para calcular la ruta más corta entre dos ciudades, garantizando que los usuarios obtengan la opción de viaje más eficiente.
- Persistencia de datos: La clase **FileManager** permitió manejar la lectura y escritura de archivos CSV, asegurando que los datos de ciudades y conexiones se mantengan consistentes entre sesiones.
- Interfaz gráfica intuitiva: La interfaz desarrollada en Qt proporciona una experiencia de usuario amigable, facilitando tanto la consulta de rutas como la gestión de ciudades y conexiones.
- Visualización del grafo: Se implementó una representación gráfica donde las ciudades se muestran como nodos y las conexiones como líneas con flechas y etiquetas de distancia.

Sin embargo, durante el desarrollo se identificaron algunas limitaciones y áreas de mejora, las cuales se detallan en la siguiente sección.

## 6. Mejoras Futuras

A pesar de que el proyecto cumple con sus funcionalidades básicas, existen diversas optimizaciones que podrían mejorar su rendimiento y usabilidad:

### 6.1. Mejora en la señalización de flechas

Actualmente, las flechas que indican la dirección de las conexiones pueden superponerse o no ser visibles en ciertos casos. Para solucionar esto, se propone:

- Implementar un sistema de colisión que evite la superposición de flechas con otros elementos del gráfico.
- Ajustar dinámicamente la posición de las flechas para garantizar que siempre apunten en la dirección correcta y sean claramente visibles.

### 6.2. Ajuste de las proporciones en la visualización del grafo

Las distancias entre ciudades en el grafo no siempre reflejan proporcionalmente los valores reales. Para mejorar esto, se podría:

- Implementar un sistema de posicionamiento basado en las distancias reales entre ciudades.
- Utilizar un algoritmo de layout, como el de fuerza dirigida, para distribuir los nodos de manera más natural y evitar cruces innecesarios de líneas.

### 6.3. Optimización del Algoritmo de Dijkstra

Para grafos más grandes, el algoritmo de Dijkstra podría volverse ineficiente. Como alternativa, se recomienda:

- Implementar el algoritmo A\* (A estrella), que es más eficiente en grafos con muchas conexiones.
- Usar estructuras de datos más optimizadas, como un Fibonacci Heap, para mejorar el rendimiento de la cola de prioridad.

### 6.4. Autenticación de Usuarios y Administradores

Para mejorar la seguridad del sistema, se podría integrar un sistema de autenticación que permita a los usuarios y administradores acceder con credenciales únicas.

### 6.5. Integración con Mapas Reales

Como mejora adicional, se podría integrar el sistema con un servicio de mapas, como Google Maps, para mostrar las rutas sobre un mapa real y mejorar la experiencia del usuario.

## 7. Conclusiones finales

El Sistema de Reserva de Vuelos ha demostrado ser un proyecto funcional y eficiente que cumple con sus objetivos principales. Sin embargo, las mejoras propuestas permitirían optimizar su rendimiento, usabilidad y precisión en la visualización de datos.

Estas mejoras no solo beneficiarían a los usuarios finales, proporcionando una mejor experiencia en la consulta de rutas, sino que también facilitarían la gestión de la red de vuelos por parte de los administradores. Con estas optimizaciones, el sistema podría evolucionar a una herramienta más avanzada y robusta para la planificación de vuelos.

## 8. Anexos

En esta sección se presentan los fragmentos de código más relevantes utilizados en la implementación del Sistema de Reserva de Vuelos. Estos códigos incluyen la estructura de datos del grafo, el algoritmo de Dijkstra, la gestión de archivos CSV y la interfaz gráfica desarrollada en Qt.

### 8.1. Definición del Grafo

El grafo se modela utilizando un `unordered_map` de C++, donde cada ciudad se asocia a sus conexiones con otras ciudades y sus respectivas distancias.

Listing 3: Definición de la clase Graph

```
1 class Graph {
2     private:
3         unordered_map<string, unordered_map<string, double>> adjMatrix
4         ;
5     public:
6         void printGraph();
7         bool addVertex(string vertex);
8         bool addEdge(string vertex1, string vertex2, double weight);
9         bool addEdgeDir(string vertex1, string vertex2, double weight)
10        ;
11        bool removeVertex(string vertex);
12        bool removeEdge(string vertex1, string vertex2);
13        bool modifyEdge(string vertex1, string vertex2, double weight)
14        ;
15        vector<string> getNodes() const;
16        const unordered_map<string, unordered_map<string, double>>&
17        getEdges() const;
18        const unordered_map<std::string, double> getEdges(string
19        vertex) const;
20        bool isEdge(string vertex1, string vertex2);
21        std::vector<std::string> dijkstra(const std::string& origen,
22        const std::string& destino);
23    };
```

### 8.2. Implementación del Algoritmo de Dijkstra

El algoritmo de Dijkstra se utiliza para calcular la ruta más corta entre dos ciudades, utilizando una cola de prioridad basada en un min-heap.

Listing 4: Algoritmo de Dijkstra

```
1 pair<double, Graph> dijkstra(const Graph& graph, const string& origin,
2     const string& destination) {
3     const auto& adjMatrix = graph.getEdges();
4     unordered_map<string, double> dist;
5     unordered_map<string, string> parent;
6     for (const auto& cityPair : adjMatrix) {
```

```

7     dist[cityPair.first] = INF;
8 }
9 dist[origin] = 0.0;
10
11 priority_queue<
12     pair<double, string>,
13     vector<pair<double, string>>,
14     greater<pair<double, string>>
15 > pq;
16
17 pq.push({0.0, origin});
18
19 while (!pq.empty()) {
20     double currentDist = pq.top().first;
21     string currentCity = pq.top().second;
22     pq.pop();
23
24     if (currentCity == destination) {
25         break;
26     }
27
28     if (currentDist > dist[currentCity]) {
29         continue;
30     }
31
32     for (const auto& edge : adjMatrix.at(currentCity)) {
33         const string& neighborCity = edge.first;
34         double flightDistance = edge.second;
35
36         if (dist[currentCity] + flightDistance < dist[neighborCity]) {
37             dist[neighborCity] = dist[currentCity] +
38                 flightDistance;
39             parent[neighborCity] = currentCity;
40             pq.push({dist[neighborCity], neighborCity});
41         }
42     }
43
44     return {dist[destination], reconstructGraph(parent, origin,
45         destination)};
46 }

```

### 8.3. Gestión de Datos con CSV

Los datos de las ciudades y conexiones se almacenan en archivos CSV. La clase `FileManager` permite cargar y guardar estos datos de manera eficiente.

Listing 5: Lectura de datos desde CSV

```

1 Graph FileManager::loadGraphFromCSV(const string& filename) {
2     Graph graph;
3     ifstream file(filename);

```

```

4      if (!file.is_open()) {
5          std::cerr << "Error al abrir el archivo CSV." << std::endl;
6          return graph;
7      }
8      string line;
9      getline(file, line);
10
11     unordered_set<string> vertices;
12     while (getline(file, line)) {
13         stringstream ss(line);
14         string source, target;
15         double weight;
16
17         getline(ss, source, ',');
18         getline(ss, target, ',');
19         ss >> weight;
20
21         vertices.insert(source);
22         vertices.insert(target);
23     }
24
25     for (const string& vertex : vertices) {
26         graph.addVertex(vertex);
27     }
28
29     file.clear();
30     file.seekg(0);
31     getline(file, line);
32
33     while (getline(file, line)) {
34         stringstream ss(line);
35         string source, target;
36         double weight;
37
38         getline(ss, source, ',');
39         getline(ss, target, ',');
40         ss >> weight;
41
42         graph.addEdge(source, target, weight);
43     }
44
45     return graph;
46 }

```

## 8.4. Interfaz Gráfica en Qt

La interfaz gráfica fue desarrollada en Qt, proporcionando herramientas intuitivas para gestionar rutas y visualizar el grafo de ciudades y conexiones.

Listing 6: Visualización del grafo en Qt

```

1 void thirdDialog::loadGraph() {
2     Graph graph;

```



```

3   QString archivoCopia = "../gui/ciudades_copy.csv";
4   QString archivoOriginal = "../gui/ciudades.csv";
5
6   if (QFile::exists(archivoCopia)) {
7       graph = FileManager::loadGraphFromCSV(archivoCopia.toStdString
8       ());
9   } else {
10      graph = FileManager::loadGraphFromCSV(archivoOriginal.
11      toStdString());
12  }
13
14  QMap<QString, QPointF> cityPositions;
15  for (const auto& city : graph.getNodes()) {
16      if (!cityPositions.contains(QString::fromStdString(city))) {
17          QPointF pos = randomPosition();
18          cityPositions[QString::fromStdString(city)] = pos;
19      }
20  }
21
22  for (const auto& [city1, edges] : graph.getEdges()) {
23      for (const auto& [city2, weight] : edges) {
24          QPointF pos1 = cityPositions[QString::fromStdString(city1)];
25          QPointF pos2 = cityPositions[QString::fromStdString(city2)];
26
27          QGraphicsLineItem *line = new QGraphicsLineItem(pos1.x() +
28          10, pos1.y() + 10, pos2.x() + 10, pos2.y() + 10);
29          line->setPen(QPen(Qt::black, 2));
30          scene->addItem(line);
31
32          QPointF midPoint = (pos1 + pos2) / 2;
33          QString weightText = QString::number(weight) + "-km";
34          QGraphicsTextItem *weightLabel = new QGraphicsTextItem(
35          weightText);
36
37          double angle = std::atan2(pos2.y() - pos1.y(), pos2.x() -
38          pos1.x()) * 180 / M_PI;
39          if (angle > 90 || angle < -90) {
40              angle += 180;
41          }
42          weightLabel->setRotation(angle);
43
44          double offset = 10;
45          QPointF perpendicularOffset(-offset * sin(angle * M_PI /
46          180), offset * cos(angle * M_PI / 180));
47
48          if (city1 < city2) {
49              weightLabel->setPos(midPoint + perpendicularOffset);
50          } else {
51              weightLabel->setPos(midPoint - perpendicularOffset);
52          }
53      }
54  }

```

```

48         scene->addItem(weightLabel);
49     }
50 }
51
52 for (auto it = cityPositions.begin(); it != cityPositions.end();
53      ++it) {
54     QGraphicsEllipseItem *node = new QGraphicsEllipseItem(it.value
55         ().x(), it.value().y(), 20, 20);
56     node->setBrush(Qt::blue);
57     scene->addItem(node);
58
59     QGraphicsTextItem *label = new QGraphicsTextItem(it.key());
60     label->setFont(QFont("Arial", 10));
61     label->setDefaultTextColor(Qt::red);
62     label->setPos(it.value().x() + 5, it.value().y() - 20);
63     scene->addItem(label);
64 }

```

# Referencias

## Grafo

Para saber cómo implementar la matriz de adyacencia con: `std::unordered_map<int, std::unordered_map<int, int>> adjMatrix;`. Esto permite guardar directamente un string con el nombre de la ciudad origen, y también guardar un string con la ciudad destino y el peso de esta conexión, el cual representa la distancia entre las ciudades.

- <https://chatgpt.com/share/67b21547-3ef4-8009-9b74-ec8ca62f73f4>
- [https://cplusplus.com/reference/unordered\\_map/unordered\\_map/](https://cplusplus.com/reference/unordered_map/unordered_map/)

¿Cómo puedo implementar un grafo ponderado usando un mapa desordenado, en forma de matriz de adyacencia?

Puedes implementar un grafo ponderado utilizando un `unordered_map` (mapa desordenado) en C++ para simular una matriz de adyacencia. La clave de este mapa será el nodo de origen y el valor será otro mapa (también `unordered_map`) donde las claves serán los nodos de destino y los valores serán los pesos de las aristas entre esos nodos.

Aquí te dejo un ejemplo de cómo podrías hacerlo:

### Definición de la clase `Graph`:

cpp

Copiar Editar

```
#include <unordered_map>
#include <iostream>
#include <vector>

class Graph {
private:
    // Grafo representado como un mapa desordenado de nodos (clave) y un mapa de destinos y pesos
    std::unordered_map<int, std::unordered_map<int, int>> adjMatrix;
```

En realidad quiero ingresar strings

¡Entiendo! Si deseas trabajar con nodos representados por `strings` en lugar de `int`, simplemente debes modificar el tipo de las claves en el `unordered_map`. A continuación te muestro cómo implementar un grafo ponderado utilizando `std::string` para los nodos.

**Definición de la clase `Graph` con `std::string`:**

cpp

Copiar Editar

```
#include <unordered_map>
#include <iostream>
#include <string>

class Graph {
private:
    // Grafo representado como un mapa desordenado de nodos (clave) y un mapa de destinos y peso
    std::unordered_map<std::string, std::unordered_map<std::string, int>> adjMatrix;
```

También se utilizó para entender cómo funciona `const auto` para declarar un iterador en un loop, basándose en estos ejemplos en Stack Overflow sobre cómo iterar en un `std map`:

- <https://stackoverflow.com/questions/26281979/how-do-you-loop-through-a-stdmap>

## ¿Para qué sirve const auto?

El uso de `const auto` en C++ tiene varios propósitos, dependiendo del contexto en el que se utilice. En general, es una forma de declarar variables sin tener que especificar explícitamente su tipo, mientras que al mismo tiempo aseguramos que el valor de la variable no pueda ser modificado.



### Desglosando `const auto`:

#### 1. Inferencia de tipo con `auto`:

- `auto` permite que el compilador infiera el tipo de la variable de acuerdo con el valor con el que se inicializa. Esto es útil cuando no quieres especificar el tipo de manera explícita, lo que hace que el código sea más conciso y fácil de mantener.

Ejemplo:

cpp

 Copiar  Editar

```
auto x = 42; // El compilador infiere que x es de tipo int
```

#### 2. Uso de `const` con `auto`:

- Cuando se agrega `const` antes de `auto`, se asegura que la variable no pueda ser modificada después de su inicialización. Es útil cuando quieres asegurarte de que el valor de una variable no cambie, especialmente cuando trabajas con tipos complejos o estructuras como contenedores o elementos de un contenedor.

## ¿Por qué usar `const auto`?

1. **Para garantizar que el valor no cambie:** Cuando usas `const auto`, el compilador infiere el tipo, pero al mismo tiempo le aseguras que la variable es constante. Esto es particularmente útil cuando iteras sobre contenedores o estructuras donde no deseas modificar los elementos.

Ejemplo:

```
cpp                                                                    Copiar Editar

std::vector<int> vec = {1, 2, 3};
for (const auto& val : vec) {
    // val es una referencia constante a cada elemento de vec
    std::cout << val << " "; // Solo lectura, no se puede modificar
}
```

2. **Para evitar errores de modificación accidental:** Usar `const auto` previene la modificación accidental de valores cuando se hace iteración sobre colecciones o se manipulan datos que no deberían ser alterados.

Ejemplo:

```
cpp                                                                    Copiar Editar

std::map<std::string, int> ages = {{ "Alice", 30 }, { "Bob", 25 }};
for (const auto& pair : ages) {
    // 'pair' es una referencia constante y no puede modificarse
    std::cout << pair.first << " is " << pair.second << " years old.\n";
}
```

3. **Cuando el tipo es complicado:** A veces, los tipos pueden ser largos o complicados, como las iteraciones sobre contenedores de punteros, estructuras complejas, o iteradores. Usar `auto` junto con `const` facilita la lectura y evita errores en la declaración de tipos complejos.

Ejemplo:

```
cpp                                                                    Copiar Editar

std::unordered_map<std::string, std::vector<int>>> data;
const auto& item = data["key"];
// 'item' es una referencia constante a un std::vector<int> asociado a "key"
```

# Algoritmo Dijkstra

## Verificación de existencia de ciudades en el grafo:

Acá simplemente se revisa si origin y destination se encuentran en la matriz de adyacencia.

```
if (adjMatrix.find(origin) == adjMatrix.end() ||
    adjMatrix.find(destination) == adjMatrix.end()) {
    cout << "Una o ambas ciudades no existen en el grafo.\n";
    return {-1, Graph()}; // Retornar valor inválido y grafo vacío
}
```

## Inicialización de estructuras:

“dist” permite el mapeo de cada ciudad con la distancia mínima desde origin

“parent “ almacena el predecesor de cada ciudad para luego construir la ruta.

“pq” es la cola de prioridad que permite el procesamiento de nodos ordenados con la menor distancia.

```
unordered_map<string, double> dist;
unordered_map<string, string> parent;
priority_queue<
    pair<double, string>,
    vector<pair<double, string>>,
    greater<pair<double, string>>
> pq;
```

## Inicialización de distancias

Dado que no se conoce ninguna distancia mínima a ninguna ciudad, se inicializan todas las ciudades en infinito a excepción del origen, ya que este se coloca en la cola de prioridad.

```
for (const auto& cityPair : adjMatrix) {
    dist[cityPair.first] = INF;
}
dist[origin] = 0.0;
pq.push({0.0, origin});
```

## Algoritmo Dijkstra

La condición del bucle es que la cola mientras no esté vacía se procede a extraer la ciudad con la menor distancia acumulada, se pregunta si ya se está en el destino para terminar, asimismo, si la distancia que se obtuvo está desactualizada se ignora la iteración. Por último, si se logra encontrar una ruta con menor distancia, se actualizan las distancias de las ciudades vecinas.

```
while (!pq.empty()) {
    double currentDist = pq.top().first;
    string currentCity = pq.top().second;
    pq.pop();

    if (currentCity == destination) {
        break;
    }
}
```

```

    if (currentDist > dist[currentCity]) {
        continue;
    }

    for (const auto& edge : adjMatrix.at(currentCity)) {
        const string& neighborCity = edge.first;
        double flightDistance = edge.second;

        if (dist[currentCity] + flightDistance < dist[neighborCity]) {
            dist[neighborCity] = dist[currentCity] + flightDistance;
            parent[neighborCity] = currentCity;
            pq.push({dist[neighborCity], neighborCity});
        }
    }
}

```

### Verificación de existencia de ruta:

Se pregunta si la distancia final sigue siendo infinita, ya que si esto pasa significa que no existe una conexión entre las ciudades.

```

if (dist[destination] == INF) {
    cout << "No existe ruta entre " << origin << " y " << destination << ".\n";
    return {INF, Graph()};
}

```

### Reconstrucción de la ruta más corta:

Desde “destination” a “origin” se reconstruye la ruta usando el mapa “parent” para luego realizar una inversión. Ya luego simplemente se imprime la ruta

```

vector<string> path;
for (string at = destination; /* vacío */; ) {
    path.push_back(at);
    if (at == origin) break;
    at = parent[at];
}
reverse(path.begin(), path.end());

```

### Generación del subgrafo con la ruta óptima:

Simplemente se genera un subgrafo con los nodos y aristas que se ven involucrados en la ruta óptima. Ya luego se retorna este subgrafo y la distancia mínima.

```

Graph subgraph;
for (size_t i = 0; i < path.size() - 1; ++i) {
    const string& currentCity = path[i];
    const string& nextCity = path[i + 1];

    subgraph.addVertex(currentCity);
    subgraph.addVertex(nextCity);
}

```



```
        double weight = adjMatrix.at(currentCity).at(nextCity);
        subgraph.addEdgeDir(currentCity, nextCity, weight);
    }
    return {dist[destination], subgraph};
```

Para empezar en la construcción de este algoritmo se indagó en los dos videos linkeados a continuación. Sin embargo se observa que en el algoritmo presente en el repositorio, obtenido del video de grafo dinámico C++ Algoritmo Dijkstra, este recibe el grafo como una lista de adyacencia, donde cada índice del vector principal representa un nodo y estos tienen consigo una par que maneja al nodo vecino y el peso. Sin embargo, dado que este código manejaba índices enteros, para el proyecto era necesario manejarlos como strings que representan los nombres de los países, por ende se procedió a adaptar el algoritmo para que reciba una matriz de adyacencia y este tipo de cadenas más flexible para el manejo de las ciudades (nodos) con ayuda de GitHub Copilot. El manejo de distancias también se tuvo que adaptar, ya que el código de referencia utilizaba un vector de enteros donde las posiciones son los nodos por índice, entonces dado que las distancias de interés podrían considerar valores flotantes se corrigió este funcionamiento a uno que permita manejar los nodos con un nombre y peso asociado equivalente a la distancia. Luego, es claro que el primer código al no ser tan flexible para agregar o eliminar nodos y aristas, el cual era un requisito para el proyecto se tuvo que modificar la estructura del grafo en tiempo de ejecución para que se pudieran agregar o eliminar ciudades. Ahora bien, dado que el código tomado solo calcula la distancia mínima entre nodos, era necesario realizar una modificación donde se almacenaran los predecesores para luego reconstruir la ruta óptima y mostrarla a través de un subgrafo, entonces dichas modificaciones fueron hechas utilizando GitHub Copilot y ChatGpt.

- <https://github.com/EEmanuelLG/GrafoDinamico/blob/master/GrafoDinamico/Grafo.cpp>
- <https://youtu.be/vs14lxuW33I?si=U913nfEpYse8h99h>
- [https://youtu.be/pymKkF7Rk\\_8?si=8re5mSkRI4RIzfoO](https://youtu.be/pymKkF7Rk_8?si=8re5mSkRI4RIzfoO)



```
compara ese código con este : # include <iostream>
# include <vector>
# include <algorithm>
# include <queue>

vector < vector < pair < int , int > > grafo;
vector < int > distancias;

// función crear lista
void crear_lista(int n){
    for (int i = 0; i < n; i++){
        distancias.push_back(1000000000);
        vector < pair < int , int > > nodo = {};
        grafo.push_back(nodo);
    }
}

// agregar vecino
void agregar_vecino(int origen, int destino, float distancia){
    grafo[origen].push_back(make_pair(destino, distancia));
    grafo[destino].push_back(make_pair(origen, distancia));
}

// djikstra
int djikstra(int origen, int destino){
    distancias[origen] = 0;
    priority_queue < pair < int , int > ,
    vector < pair < int , int > > ,
```



dado este proyecto con 30 mínimo de nodos que corresponden a países y donde algunas rutas deben hacer escalas, qué código me recomiendas: Sistema de Reserva de Vuelos

Requerimientos específicos:

1. Gestión de vuelos:

- Crear y eliminar ciudades (nodos del grafo).
- Añadir, modificar y eliminar conexiones (aristas con peso que representan distancia o costo).

2. Rutas óptimas:

- Implementar Dijkstra o A\* para calcular rutas óptimas entre ciudades.

3. Interfaz gráfica:

- Mostrar un mapa interactivo con nodos (ciudades) y aristas (vuelos).
- Permitir seleccionar ciudades para calcular rutas.

4. Persistencia:

- Guardar el grafo y las conexiones en un archivo para su recuperación.

Estructuras de datos obligatorias:

6

- Grafo ponderado dirigido para representar vuelos.
- Cola de prioridad para implementar Dijkstra o A\*.
- Mapas o vectores para manejar pesos y conexiones.

Sugerencia para la interfaz gráfica: Usar Qt para mostrar un mapa interactivo

donde cada ciudad sea un nodo y los vuelos, líneas entre ellos.

Permitir al usuario

Se le pasan ambos códigos para ver ciertas diferencias que presentaban y ver un poco la eficiencia algorítmica, esto permitió tomar en cuenta ciertos aspectos necesarios como los

que se mencionaron anteriormente, manejo de string y flotantes, una mejor estructura dinámica donde se pudieran modificar ciudades agregar-eliminar, o hasta los mismos predecesores que no se tomaban en cuenta para la reconstrucción de la ruta óptima,

nota, el valor de las aristas corresponde a la distancia del vuelo

Esta orden permitió establecer como flotantes los valores de las distancias.



este metodo de clase no existe en graph

graph.getAdjMatrix().

Acá había problemas con lo que es la obtención de las conexiones de los nodos , por lo que se corrigió eso por getEdges.

```
return 0;
}
```

Obtengo esta salida, como hago para ver escalas, que no todos los viajes sean directos

Estaba obteniendo una salida no deseada, por lo que se adecuó al formato final dado: por origen->escala->destino.

```
    }
    }
    return false;
}

// Implementación del método getEdges para obtener las aristas
del grafo
const unordered_map<string, unordered_map<string, double>>&
Graph::getEdges() const {
    return adjMatrix;
}

// Método para obtener aristas de un vértice específico
const unordered_map<string, double>
Graph::getEdges(string vertex) const {
    if (adjMatrix.count(vertex) != 0) { // Si encuentra el elemento en
    el mapa
        return adjMatrix.at(vertex); // Retornar las conexiones del
vértice específico
    }
    return {}; // Si no encuentra el elemento en el mapa, retornar
mapa vacío
}
```

Tu implementación de `Graph` está bien estructurada, pero hay un **error en la firma del método**

`getEdges(string vertex) const`.



```

src/algorithms/Dijkstra.cpp: In function 'double dijkstra(const
Graph&, const string&, const string&)':
src/algorithms/Dijkstra.cpp:29:35: error: 'const class Graph' has
no member named 'getAdjMatrix'; did you mean
'std::unordered_map<std::__cxx11::basic_string<char>,
std::unordered_map<std::__cxx11::basic_string<char>, double> >
Graph::adjMatrix?' (not accessible from this context)
 29 |   const auto& adjMatrix = graph.getAdjMatrix();
    |                               ^~~~~~
In file included from src/algorithms/Dijkstra.cpp:7:
src/algorithms/./../include/graph/Graph.h:12:62: note: declared
private here
 12 |   unordered_map<string, unordered_map<string,
    |   double>> adjMatrix;

```

Estos errores se debían a que no estaba declarado un método `getAdjMatrix` en la clase `graph`, pero sí contaba con uno, el cual era `adjmatrix`, pero era privado, entonces no se podía acceder desde el archivo `Dijkstra`, sin embargo la clase `graph` al tener un método que ya devolvía la matriz de adyacencia se soluciona a través de acceder este, el cual es `getEdges` y así no se violentaba la encapsulación. Otro error que se presentó es con el llamado de `getEdges`, ya que este podía generar conflictos al llamarlo con parámetros, dado que podía devolver las conexiones de un nodo en específico, entonces podría generar confusión en el llamado, ya que al pretender llamar a la matriz de adyacencia se estaba trayendo a los vecinos del parámetro.

## Persistencia

Para poder implementar la lectura de un archivo CSV, se basa en el siguiente enlace:  
[https://medium.com/@ryan\\_forrester /reading-csv-files-in-c-how-to-guide-35030eb378ad](https://medium.com/@ryan_forrester /reading-csv-files-in-c-how-to-guide-35030eb378ad)

En este se muestra como se puede obtener el contenido de un archivo.

Se usa este prompt para poder crear el grafo a partir de los vertices, guardando todos los vértices de primero y luego agregando las aristas.

```

puedes generar que esto primero cree los vértices que es la
primera columna del csv y que luego agregue las aristas:
Graph FileManager::loadGraphFromCSV(const string& filename) {
    Graph graph;
    ifstream file(filename);
    string line;
    getline(file, line); // Ignorar la primera línea

    while (getline(file, line)) {
        stringstream ss(line); // Convertir la línea en un stream
        string source, target; // Variables para almacenar los nodos
        origen y destino
        double weight; // Variable para almacenar el peso de la arista

        getline(ss, source, ','); // Leer el nodo origen
        getline(ss, target, ','); // Leer el nodo destino
        ss >> weight; // Leer el peso de la arista

        graph.addVertex(source); // Agregar el nodo origen al grafo
        graph.addEdge(source, target, weight); // Agregar la arista al
        grafo con su peso

    }
    return graph;
}

```

Para guardar un grafo en formato CSV:

digamos, que tome el csv, lo lea y pase esos datos a otro csv y luego de eso que genere el grafo llamando al Graph.cpp?

```

void FileManager::saveGraphToCSV(const Graph& graph, const std::string& filename) {
    std::ofstream file(filename);

    // Recorrer todas las aristas del grafo y guardarlas en el archivo
    for (const auto& [city1, edges] : graph.getEdges()) {
        for (const auto& [city2, weight] : edges) {
            file << city1 << "," << city2 << "," << weight << "\n";
        }
    }
}

```

## Interfaz gráfica

Pestaña Admin:

Se utilizó IA para una versión preliminar de la función Admin::inicializarCiudades. Específicamente para entender cómo usar strings en QT. Para esto se usa la clase QString, la cual recibe un string del std library y lo convierte a la versión de QT.

- <https://chatgpt.com/share/67b21547-3ef4-8009-9b74-ec8ca62f73f4>
- <https://doc.qt.io/qt-6/qstring.html>



Cómo hago que esto retorne ciudades?

```
void Admin::obtenerCiudadesDesdeGrafo() {
    QStringList ciudades;

    // Obtener la lista de ciudades desde el grafo
    for (const auto& element : grafoCiudades.getEdges()) {
        ciudades.append(QString::fromStdString(element.first));
    }

    // Llamar a la función original que actualiza el ComboBox
    actualizarCiudades(ciudades);
}
```

Para que tu función `obtenerCiudadesDesdeGrafo` retorne las ciudades, en lugar de solo actualizar el `ComboBox` mediante `actualizarCiudades(ciudades)`, necesitas modificar la función para que **devuelva** un valor de tipo `QStringList`, que es una lista de cadenas en Qt. Además, en el código actual, estás utilizando el método `append` de `QStringList` para agregar las ciudades, lo cual está bien, pero la función debe retornar la lista en lugar de simplemente pasarla a otra función.

Aquí te dejo cómo modificar la función para que retorne un `QStringList` con las ciudades:

También para comprender cómo convertir de `QString` a `std::double`, para lo cual se define primero una variable `QString`, por ejemplo una `strVar`, y luego se utiliza el comando `strVar.toDouble(double var)`.



- <https://chatgpt.com/share/67b21547-3ef4-8009-9b74-ec8ca62f73f4>

Y convertir de QString a double?

Sí, puedes convertir un `QString` a `double` en C++ usando el método `toDouble()`, que está integrado en la clase `QString`.

### Ejemplo de conversión de `QString` a `double`:

cpp

 Copiar  Editar

```
#include <QString>
#include <iostream>

int main() {
    QString qstr = "123.456";

    // Convertir a double
    bool ok;
    double value = qstr.toDouble(&ok);

    // Verificar si la conversión fue exitosa
    if (ok) {
        std::cout << "El número es: " << value << std::endl;
    } else {
        std::cout << "Conversión fallida." << std::endl;
    }

    return 0;
}
```



Aquí te dejo cómo modificar la función para que retorne un `QStringList` con las ciudades:

### Función modificada para retornar un `QStringList`:

cpp

Copiar Editar

```
QStringList Admin::obtenerCiudadesDesdeGrafo() {
    QStringList ciudades;

    // Obtener la lista de ciudades desde el grafo
    for (const auto& element : grafoCiudades.getEdges()) {
        ciudades.append(QString::fromStdString(element.first)); // Asumiendo que 'first' es
    }

    return ciudades; // Retornar la lista de ciudades
}
```

### Explicación de los cambios:

1. **Tipo de retorno:** Cambié la firma de la función para que retorne un `QStringList` en lugar de `void`.
2. **Llamada a `append`:** Sigues utilizando `append` para agregar los nombres de las ciudades (convertidos de `std::string` a `QString`), lo cual está correcto.
3. **Retorno de la lista:** En lugar de llamar a `actualizarCiudades(ciudades)`, ahora la función retorna la lista `ciudades`, que puede ser utilizada más tarde.

Se utilizó además para resolver un error al imprimir un mensaje en el `QMessageBox`. El problema es que, de la manera que estaba, se estaba intentando concatenar tipos de datos diferentes, siendo uno `const char*` y el otro `QString`. ChatGPT sugirió en su lugar utilizar esto para convertir el tipo de dato: `.arg(qstrciudadOrigen)`.

- <https://chatgpt.com/share/67b21547-3ef4-8009-9b74-ec8ca62f73f4>
- <https://doc.qt.io/qt-6/qmessagebox.html>





Esta línea está dando error, qué hice mal?

```
connectionConfirmationBox.setText("¿Estás seguro que deseas  
crear esta conexión?" + "\nCiudad Origen: " + qstrciudadOrigen +  
"\nCiudad Destino: " + qstrciudadDestino);
```

Método:

```
void Admin::on_pushButton_clicked()
{
    // Guardar la selección del administrador en variables
    QString qstrciudadOrigen = ui->comboBoxCiudadOrigen-
>currentText();
    QString qstrciudadDestino = ui->comboBoxCiudadDestino-
>currentText();
    QString qstrDistancia = ui->lineEditDistancia->text();

    // Convertir de QString a string convencional
    std::string strCiudadOrigen = qstrciudadOrigen.toStdString();
    std::string strCiudadDestino = qstrciudadDestino.toStdString();
    double distancia = qstrDistancia.toDouble();

    QMessageBox connectionConfirmationBox;
    connectionConfirmationBox.setIcon(QMessageBox::Warning);
    connectionConfirmationBox.setWindowTitle("Confirmar  
conexión");
    connectionConfirmationBox.setText("¿Estás seguro que deseas  
crear esta conexión?" + "\nCiudad Origen: " + qstrciudadOrigen +
```

```

"\nCiuudad Destino: " + qstrciudadDestino);

connectionConfirmationBox.setStandardButtons(QMessageBox::Yes
| QMessageBox::No);

connectionConfirmationBox.setDefaultButton(QMessageBox::No);

// Verificar la respuesta del usuario
if (connectionConfirmationBox.exec() == QMessageBox::Yes) {
    // Añadir conexión al grafo
    grafoCiudades.addEdge(strCiudadOrigen, strCiudadDestino,
distancia);
}

```

El error en esta línea:

cpp

Copiar Editar

```

connectionConfirmationBox.setText("¿Estás seguro que deseas crear esta conexión?" + "\nCiuudad

```

se debe a que estás intentando concatenar un `const char*` (`"\nCiuudad Origen: "`) con un `QString` (`qstrciudadOrigen`), lo cual no está permitido de esa manera en Qt.

## Solución:

Debes asegurarte de que toda la concatenación se haga con `QString`. Usa `QStringLiteral()` o el operador `+` con `QString` explícito:



## ✓ Forma correcta:

cpp

Copiar Editar

```

connectionConfirmationBox.setText(
    QString("¿Estás seguro que deseas crear esta conexión?\nCiuudad Origen: %1\nCiuudad Destino
    .arg(qstrciudadOrigen)
    .arg(qstrciudadDestino)
);

```

## Explicación:

- `QString("texto %1 %2").arg(valor1).arg(valor2);`
  - `%1` será reemplazado por `qstrciudadOrigen`
  - `%2` será reemplazado por `qstrciudadDestino`

Además, para agregar un pop up de confirmación del cambio. Para esto se crea otro QMessageBox que imprima la información que uno desea. Sin embargo, no solo existe el tipo information, también está el critical, lo cual fue aprovechado para agregar varias validaciones (que las entradas no fueran la misma, que el campo de distancia no esté vacío, que la conexión no exista ya).

- <https://chatgpt.com/share/67b21547-3ef4-8009-9b74-ec8ca62f73f4>
- <https://doc.qt.io/qt-6/qmessagebox.html>



Cómo reemplazar el mensaje del cuadro después de la confirmación?

Para reemplazar el mensaje en el QMessageBox después de la confirmación del usuario, puedes hacer lo siguiente:

1. Cerrar el cuadro de diálogo actual (el de confirmación).
2. Abrir un nuevo QMessageBox con el mensaje actualizado.

```
// Si el usuario confirma, añadir la conexión y mostrar otro mensaje
if (connectionConfirmationBox.exec() == QMessageBox::Yes) {
    grafoCiudades.addEdge(strCiudadOrigen, strCiudadDestino, distancia);

    // Mostrar nuevo cuadro de diálogo con mensaje de éxito
    QMessageBox::information(this, "Conexión Creada",
        QString("La conexión entre %1 y %2 ha sido creada con éxito.")
            .arg(qstrciudadOrigen)
            .arg(qstrciudadDestino)
    );
}
}
```

Implementación del método Dijkstra en la interfaz.

Con ayuda de la inteligencia artificial se logró conectar la interfaz para que usara el método Dijkstra construido previamente.

prompt

ya tengo cierta parte de la interfaz ahora quiero probar si puedo usar un metodo dijkstra, que esta compuesto por varios archivos, un main, un node un graph y el archivo del metodo y los headers, son varios archivos y no se como hacer para que la interfaz use estos archivos para acceder al metodo

respuesta

```
#ifndef INTERFAZ_H
#define INTERFAZ_H

#include <QMainWindow>
#include "Graph.h" // ♦ Incluir la clase del grafo
#include "Dijkstra.h" // ♦ Incluir el método de Dijkstra

namespace Ui {
class Interfaz;
}

class Interfaz : public QMainWindow {
    Q_OBJECT

public:
    explicit Interfaz(QWidget *parent = nullptr);
    ~Interfaz();

private slots:
    void calcularRuta();

private:
    Ui::Interfaz *ui;
    Graph grafo; // ♦ Instancia del grafo
};

#endif // INTERFAZ_H
```

## Modificar Interfaz.cpp para calcular la ruta

cpp

Copiar Editar

```
#include "Interfaz.h"
#include "ui_Interfaz.h"
#include <QMessageBox>

Interfaz::Interfaz(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::Interfaz)
{
    ui->setupUi(this);

    // * Agregar algunas ciudades y conexiones de prueba
    grafo.addNode("San Jose");
    grafo.addNode("Alajuela");
    grafo.addNode("Cartago");
    grafo.addEdge("San Jose", "Alajuela", 10);
    grafo.addEdge("San Jose", "Cartago", 15);
    grafo.addEdge("Alajuela", "Cartago", 5);
}

Interfaz::~Interfaz() {
    delete ui;
}
```

```

void Interfaz::calcularRuta() {
    QString origen = ui->comboBoxOrigen->currentText();
    QString destino = ui->comboBoxDestino->currentText();

    if (origen == destino) {
        QMessageBox::warning(this, "Error", "El origen y destino no pueden ser iguales.");
        return;
    }

    // * Llamar al método de Dijkstra
    std::vector<std::string> ruta = grafo.dijkstra(origen.toStdString(), destino.toStdString());

    if (ruta.empty()) {
        QMessageBox::warning(this, "Error", "No se encontró una ruta.");
        return;
    }



    // * Mostrar la ruta en un mensaje
    QString mensaje = "Ruta óptima:\n";
    for (const auto &ciudad : ruta) {
        mensaje += QString::fromStdString(ciudad) + " -> ";
    }
    mensaje.chop(4); // Eliminar la última flecha

    QMessageBox::information(this, "Ruta Calculada", mensaje);
}

```

Si usas **QMake** ( **.pro** ), abre el archivo **.pro** y agrega:

bash

 Copiar  Editar

```

SOURCES += \
    src/Graph.cpp \
    src/Node.cpp \
    src/Dijkstra.cpp \
    src/Interfaz.cpp \
    src/secondDialog.cpp \
    src/Admin.cpp

HEADERS += \
    src/Graph.h \
    src/Node.h \
    src/Dijkstra.h \
    src/Interfaz.h \
    src/secondDialog.h \
    src/Admin.h

```

Si usas **CMake**, abre **CMakeLists.txt** y agrega:

Se intentó usar los archivos del grafo y el método Dijkstra que estaban ubicados en directorios diferentes dentro del repositorio

prompt

dice que hay un error en esto include:

```
#include <QDialog>
#include "../include/graph/Graph.h" // Incluir la clase del grafo
#include "../include/algorithms/Dijkstra.h" //Incluir el método de
Dijkstra
```

respuesta

En lugar de escribir rutas complicadas en los `#include`, dile a Qt dónde encontrar los archivos de encabezado modificando el `.pro`:

✦ **Modifica tu `.pro` y agrega esta línea:**

pro

Copiar Editar

```
INCLUDEPATH += $$PWD/include/graph $$PWD/include/algorithms
```

Esto le dice a Qt Creator que busque archivos de encabezado en esas carpetas.

#### ♦ **Solución 2: Usar `#include` sin rutas relativas**

Después de agregar `INCLUDEPATH` en el `.pro`, cambia los `#include` en tu código:

**Antes (incorrecto):**

cpp

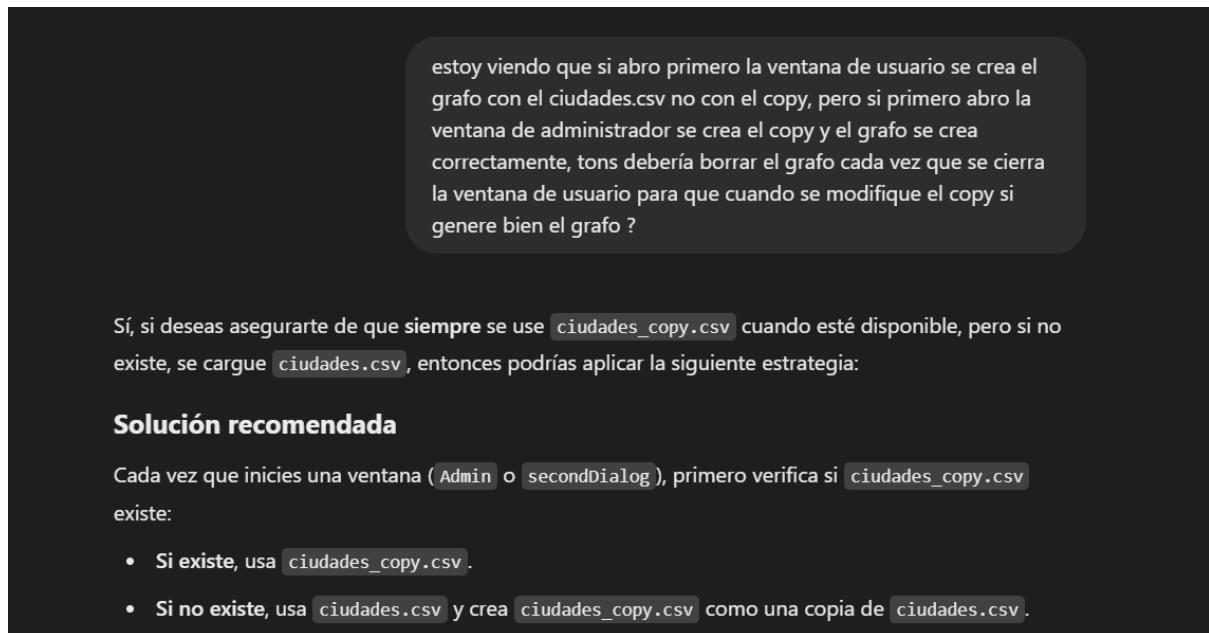
Copiar Editar

```
#include "../include/graph/Graph.h"
#include "../include/algorithms/Dijkstra.h"
```

**Después (correcto):**



Para el manejo de modificaciones en el grafo desde la ventana administrador y que se muestren en la ventana usuario, se usó este prompt:



Al final lo que se decidió realizar fue un manejo de tres archivos, uno base ciudades.csv, ciudades\_copy.csv y subgrafo\_viaje.csv, estos últimos dos se borran automáticamente si se cierra de forma correcta el programa. La lógica de cómo funciona la creación de la lista desplegada en usuario actualizada es la siguiente:

Si no existe ciudades\_copy.csv, eso quiere decir que que no se ha modificado el grafo en la ventana de administrador, y se crea el grafo en la ventana de usuario con ciudades.csv, en dado caso que sí exista es porque se modificó en administrador el grafo y se usa ese CSV. Cada vez que se cierra la ventana de usuario se borra el grafo de esa ventana y el de administrador se mantiene esto para permitir que se carguen las nuevas conexiones que sean agregadas en administrador y se vuelve a crear el grafo de la ventana de usuario si se abre.

Con el subgrafo que contiene la ruta más corta, sucede de forma similar, se crea a partir de la existencia de subgrafo\_viaje.csv, y se crea un grafo “temporal”, este archivo se borra al cerrar el programa.

Finalmente una vez obtenido el subgrafo, se optó por colocar flechas que mostraran la dirección del grafo.

prompt





por favor edita este codigo para el sub grafo que se muestra tenga direccion:

```
#include "fourddialog.h"
#include "ui_fourddialog.h"
#include "FileManager.h"
#include <QGraphicsPixmapItem>
#include <QGraphicsEllipseItem>
#include <QGraphicsLineItem>
#include <QGraphicsTextItem>
#include <QVBoxLayout>
#include <QRandomGenerator>
#include <QDebug>
#include <QFile>
#include <QFont>
#include <QMessageBox>

fourdDialog::fourdDialog(QWidget *parent)
    : QDialog(parent)
    , ui(new Ui::fourdDialog) {
    ui->setupUi(this);
    setWindowTitle("Mapa del Vuelo Programado");
    resize(700, 500);
```



Respuesta

```

#ifndef FOURDDIALOG_H
#define FOURDDIALOG_H

#include <QDialog>
#include <QGraphicsScene>
#include <QGraphicsView>
#include "Graph.h"

namespace Ui {
class fourdDialog;
}

class fourdDialog : public QDialog {
    Q_OBJECT

public:
    explicit fourdDialog(QWidget *parent = nullptr);
    ~fourdDialog();

private:
    Ui::fourdDialog *ui;
    QGraphicsScene *scene;
    QGraphicsView *view;

    void loadGraph();
    QPointF randomPosition();
    void agregarFlecha(QPointF start, QPointF end); // * Declaración correcta
};

```

Pregunta lo que quieras

En la imagen anterior se muestra el código correspondiente al header de la ventana usada para mostrar el subgrafo de la ruta más corta

cpp

Copiar Editar

```
void fourdDialog::agregarFlecha(QPointF start, QPointF end) {
    QLineF line(start, end);
    double angle = std::atan2(-line.dy(), line.dx());

    // * Definir la forma de la flecha
    QPolygonF arrowHead;
    arrowHead << QPointF(0, 0)
                << QPointF(-10, -5)
                << QPointF(-10, 5);

    QTransform transform;
    transform.translate(end.x(), end.y());
    transform.rotateRadians(-angle);

    QGraphicsPolygonItem *arrow = new QGraphicsPolygonItem(transform.map(arrowHead));
    arrow->setBrush(Qt::black);
    scene->addItem(arrow);
}
```

Mientras que aquí se muestra el método editado por la IA correspondiente al archivo principal de la ventana que muestra el subgrafo dirigido.