Algoritmos Clásicos y Estructuras de Datos

Proyecto Final

Sistema Gestión Rutas Transporte Público

Presentado por:

Claudio Ángel Yciano Rodríguez (1015-9349)

Javier Arturo Abbott Gómez (1016-0015)

Presentado a:

Prof. Freddy Peña

Fecha de entrega:

04 de Diciembre del 2025

I. Introducción

1.1. Propósito del Documento

Este informe técnico describe la arquitectura, la implementación de las estructuras de datos y los algoritmos utilizados en el "Sistema de Gestión de Rutas de Transporte Público", desarrollado como proyecto final de la asignatura. Se detallan las decisiones de diseño tomadas para cumplir con los objetivos de la rúbrica: encontrar la ruta óptima bajo múltiples criterios (tiempo, distancia, costo, transbordos) e incorporar una simulación en tiempo real.

1.2. Descripción General del Proyecto

El sistema modela una red de transporte público como un *Grafo Dirigido Ponderado* para gestionar paradas y rutas. La funcionalidad principal es calcular la ruta más corta/rápida/barata/con menos transbordos entre una parada de origen y un destino. El proyecto utiliza la plataforma *JavaFX* para una interfaz gráfica interactiva, incluyendo la visualización de la red y la gestión de datos persistentes.

II. Marco Teórico y Estructuras de Datos

2.1. Modelo Conceptual: Grafo Dirigido Ponderado

El sistema de transporte se modela mediante la estructura de grafo dirigido:.

- **Nodos (Vértices):** Representados por la clase Parada.
- **Aristas (Arcos):** Representados por la clase Ruta. La dirección indica la posible trayectoria del transporte.
- **Ponderación:** Cada arista (Ruta) posee cuatro atributos de peso independientes: distancia, tiempo (tiempoBase), y costo (costoBase).

2.2. Definición de Estructuras Clave

Clase	Descripción	Atributos Clave del Grafo
Parada	Representa los nodos del grafo.	nombre, tipo (ej. "Metro", "Bus"), rutasDeEntrada (para gestión inversa).
Ruta	Representa las aristas del grafo.	inicio, destino, distancia, tiempo, costo. Incorpora tiempoBase y costoBase para la simulación de eventos, estado (abierto/cerrado) y evento (tráfico, accidente).

Grafo	Estructura principal. Implementa el patrón Singleton.	Map<Parada, List<Ruta>> mapa (Implementación de Listas de Adyacencia).
RutaMasCorta	Objeto resumen de la solución.	rutas, totalTiempo, totalCosto, totalDistancia, totalPeso, transbordos.

Justificación de Lista de Adyacencia: Se utiliza Map<Parada, List<Ruta>> (Lista de Adyacencia) debido a su eficiencia para grafos dispersos o semi-densos (común en redes de transporte) en operaciones clave: iterar sobre las rutas de salida de una parada $(O(E))$ y almacenar el grafo.

III. Algoritmos Implementados

El sistema implementa tres algoritmos de camino más corto, seleccionando dinámicamente el más apropiado según el criterio de optimización (filtro) y la existencia de rutas cerradas.

3.1. Algoritmo de Dijkstra (Dijkstra.java)

- Propósito**:** Encontrar la ruta más corta/rápida (caminos de fuente única) con pesos no negativos.
- Uso en el programa**:** Se utiliza para los filtros tiempo y distancia cuando no hay rutas cerradas o costos negativos simulados (Grafo.obtenerMejorRuta verifica !hayRutasCerradas).
- Detalle de Implementación:
    - Utiliza una PriorityQueue<Nodo> con un comparador por peso (Comparator.comparingDouble(n -> n.peso)).
    - La complejidad es $O(E_{log}V)$
    - La clase interna Nodo encapsula la Parada y el peso acumulado.

3.2. Algoritmo de Bellman-Ford (BellmanFord.java)

- Propósito: Encontrar la ruta más corta/barata de fuente única, permitiendo pesos negativos y detectando ciclos negativos.
- Uso en el programa:
    1. Filtro costo cuando puede haber "costos negativos" (descuentos) o rutas cerradas (simuladas por eventos, aunque una ruta cerrada es un peso infinito, el uso de Bellman-Ford permite manejar cualquier peso resultante de la simulación).
    2. Filtro de transbordo (sección 3.4).
- Detalle de Implementación**:** Ejecuta la relajación (el proceso de revisar una ruta apra ver si ofrece el camino más corto) |V| - 1 veces y una última iteración para detectar ciclos negativos, lanzando un IllegalStateException si se encuentra uno. La complejidad es O(V*E).

3.3. Algoritmo de Floyd-Warshall (FloydWarshall.java)

- Propósito**:** Encontrar la ruta más corta entre todos los pares de nodos.
- Uso en el programa: Se utiliza para el filtro distancia (Grafo.rutaMasCortaFloyd). Su uso para distancia garantiza la pre-computación de todas las rutas, lo que podría usarse para mostrar rutas alternativas de forma eficiente.
- Detalle de Implementación: Utiliza dos matrices NXN: dist[][] (para la distancia) y next[][] (para la reconstrucción del camino). La reconstrucción del camino (next[i][j]) almacena una List<Ruta> para mantener la secuencia completa de aristas. La complejidad es O(V^3).

3.4. Algoritmo Especializado: Mínimo Número de Transbordos

El cálculo de mínimo número de transbordos (calcularMinTransbordos dentro de BellmanFord.java) es una solución que modela el problema como un Grafo de Estados Extendido. Con el objetivo de transformar un problema de ruta que de normal podría hacerse con Dijkstra en uno en donde el peso de la arista representa directamente el número de transbordos.

- Estado**:** En lugar de sólo el nodo P (Parada), el estado es el par (P, L), donde L es la línea de transporte utilizada para llegar a P.
- Peso de la Arista:

  $$Peso = (Cambio\ de\ Linea\ x\ BIG) + Distancia$$

  - Cambio de Línea: Es 1 si la línea de la ruta actual es diferente a la línea L del estado anterior, y 0 si es la misma.
  - Constante BIG: Una constante grande (1,000,000.0) asegura que el algoritmo siempre priorice minimizar el número de cambios sobre la minimización de la distancia (que actúa como desempate).
- Algoritmo de Cálculo: Se utiliza el algoritmo Bellman-Ford sobre este grafo de estados. Esto es apropiado porque el grafo de estados es *acíclico* en términos de transbordos (no puede haber más de V*E transbordos sin repetir una línea), pero tiene una estructura adecuada para Bellman-Ford.

IV. Decisiones de Diseño e Implementación Técnica

4.1. Arquitectura del Sistema

La arquitectura sigue el patrón Modelo-Vista-Controlador (MVC), evidente en:

- Model (models package)**:** Contiene clases como: Grafo, Parada, Ruta, RutaMasCorta, Dijkstra, BellmanFord, FloydWarshall. Contiene toda la lógica de negocio y algoritmos.
- View/Controller: (Controladores y archivos fxml respectivos de cada clase, imperativo por el uso de JavaFX y la dependencia de ObservableList<Parada> en Grafo.java).

4.2. Simulación en Tiempo Real y Criterio de Selección

- Simulación**:** El método Grafo.simularEventosRetorno() implementa el requisito opcional. Asigna aleatoriamente eventos (Accidente grave, Retraso, Lluvia) a las rutas, modificando dinámicamente los pesos (tiempo y costo) y el estado (isEstado).
- Selección Dinámica de Algoritmo: El método Grafo.obtenerMejorRuta() decide el algoritmo a usar:
    - Si costo o tiempo son elegidos, y hay una ruta con estado = false (cerrada, simulación de accidente), se usa Bellman-Ford. Esto es una decisión de diseño precautoria y robusta, asegurando la solidez ante cambios extremos en los pesos. Si no hay rutas cerradas, se usa el más eficiente Dijkstra.
    - Si es distancia, se usa Floyd-Warshall (pre-cálculo de todos los pares).
    - Si es transbordos, se usa la variante especializada de Bellman-Ford.

4.3. Recálculo Total y Persistencia de Datos

- Cálculo de Totales: Los algoritmos calculan el camino óptimo basándose en un solo filtro (totalPeso), pero la clase RutaMasCorta acumula todos los totales (totalTiempo, totalCosto, totalDistancia) en la fase de reconstrucción del camino. Esto permite mostrar la información completa (ej. la ruta más corta en distancia, pero con su tiempo y costo asociado).
- Persistencia (Implicada): La clase Grafo incluye un método cargarDesdeDB() que utiliza las clases ParadaDAO y RutaDAO para reconstruir el grafo desde una capa de datos externa, asegurando la persistencia de la red entre sesiones.

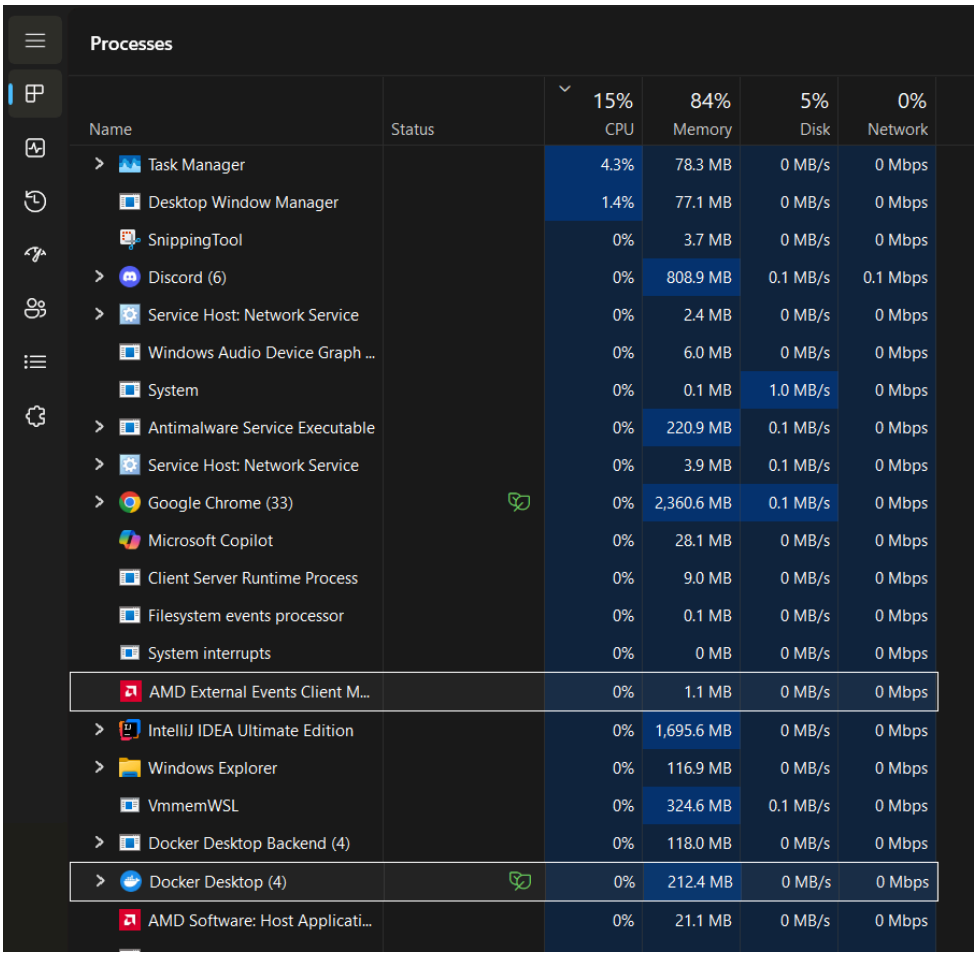
4.4. Algoritmos declarados sin implementar

Aunque los algoritmos de Árbol de Expansión Mínima (MST) (Prim y Kruskal) no forman parte de la funcionalidad de cálculo de ruta *directa* para el usuario final, se implementaron con el objetivo de demostrar la capacidad del sistema para optimizar la estructura de la red de transporte. Esta implementación exploratoria satisface el requisito de la rúbrica de aplicar algoritmos de optimización para la minimización del costo de conexión entre paradas.

V. Conclusión

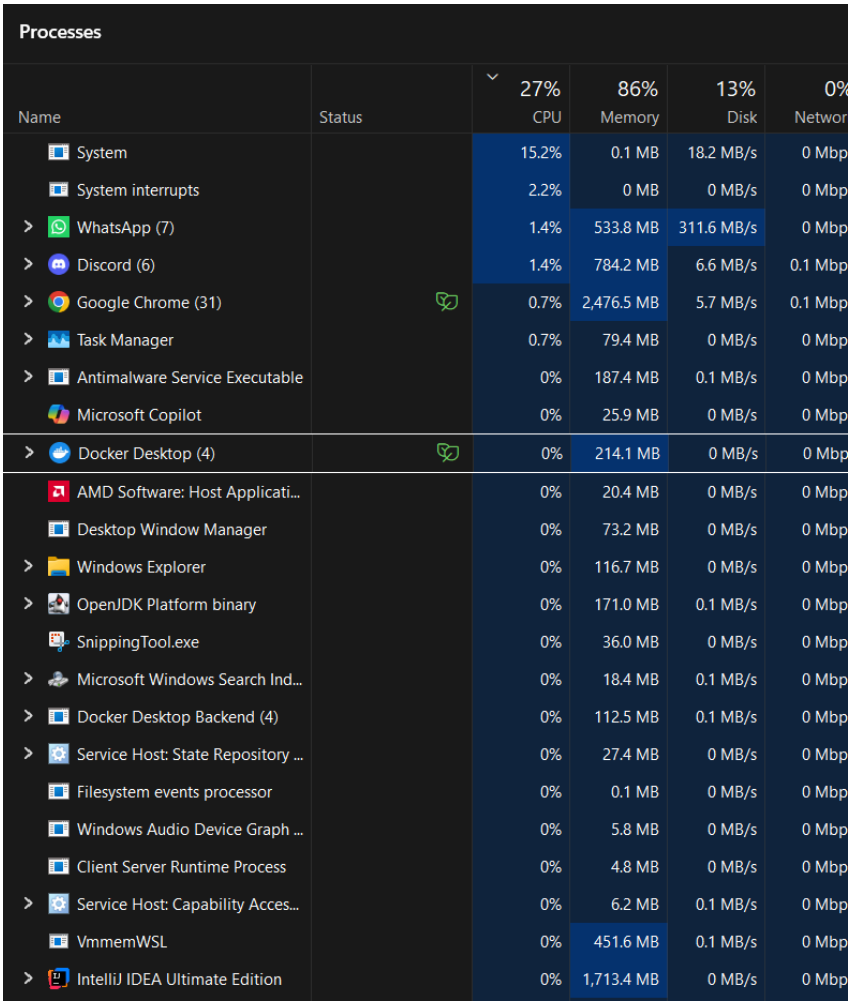
El proyecto implementa con éxito la gestión de rutas de transporte público basada en la teoría de grafos. Se cumplieron los requisitos de implementar los algoritmos Dijkstra, Bellman-Ford, y Floyd-Warshall, y se desarrolló una solución avanzada para el requisito de mínimos transbordos mediante un Grafo de Estados Extendido. La capacidad de simular eventos y recalcular dinámicamente las rutas dota al programa con buena aplicabilidad práctica y robustez algorítmica.

VI. Anexos

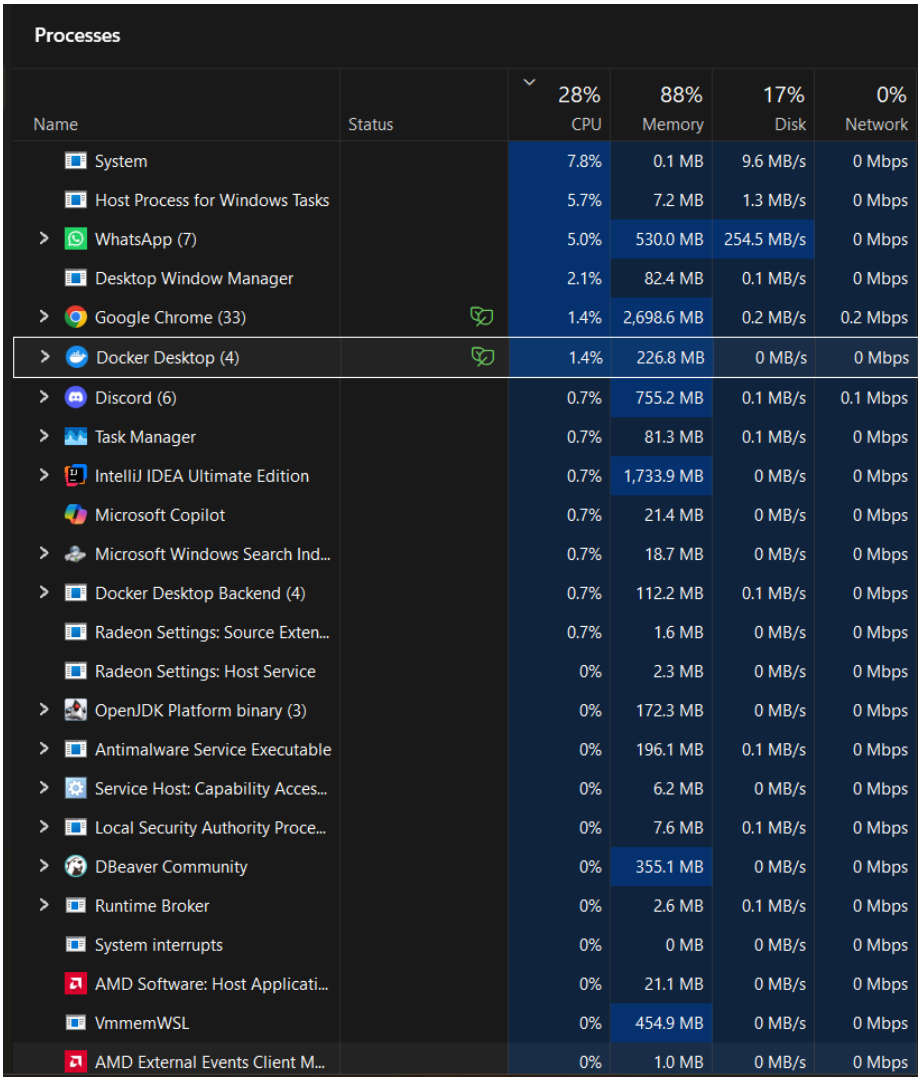


Estado previo a ejecución

.

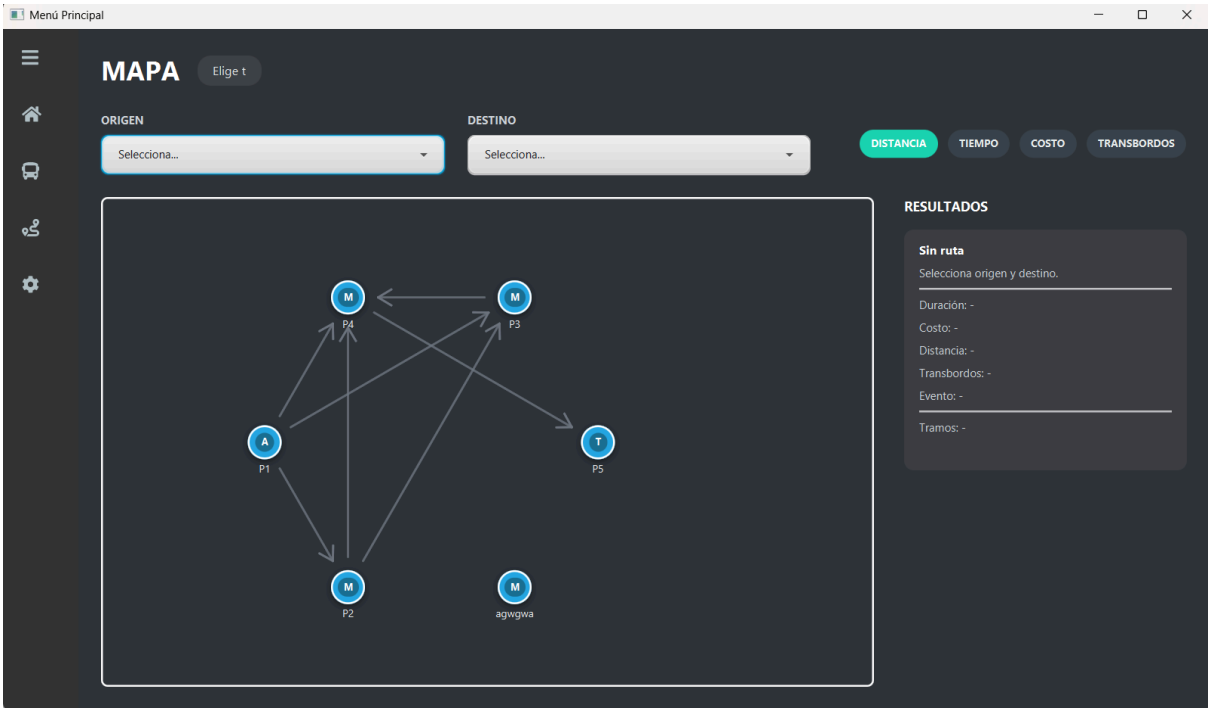


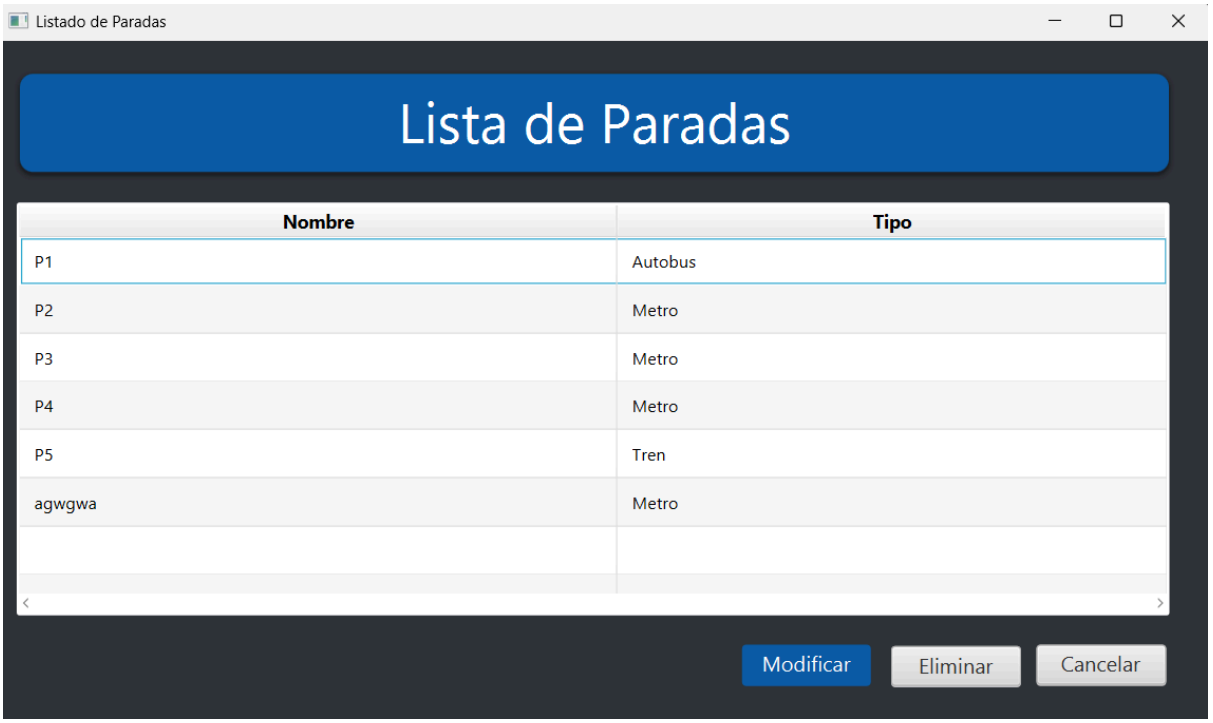
Estado en ejecución: Nótese que esto varía debido a los múltiples procesos del equipo (estado de ejecución sin realizar procesos en el programa).



Estado post-ejecución del programa y realización de procesos (registrar, modificar, listar, etc…) . No varía tanto en comparación a la ejecución, se piensa que sigue debiéndose a la inestable gestión de recursos de Windows junto a las demás pestañas y procesos en segundo plano.

Menú principal





Ejemplo de listado

Ejemplo de registro (Dinámico según las paradas que se agreguen).

.