Sistema de Navegación del Subte de Buenos Aires

Implementación del Algoritmo de Dijkstra para Grafos Pesados

1. Introducción y Problema Real

Contexto del problema

El sistema de transporte público subterráneo de Buenos Aires cuenta con 5 líneas principales (A, B, C, D, E) que conectan 83 estaciones distribuidas por la ciudad. Los usuarios frecuentemente necesitan encontrar la ruta más eficiente entre dos puntos, considerando las distancias reales entre estaciones y las posibilidades de transbordo.

Inspiración: El metro de Delhi como base estructural

Este proyecto se inspira en una implementación previa del sistema de transporte **Metro de Delhi**, el cual fue modelado mediante estructuras de grafos en el repositorio original THE-METRO-APP. En dicho modelo, las estaciones se representan como nodos y las conexiones como aristas ponderadas por distancia. A partir de esa base, se adaptó y rediseñó el sistema para representar el **Subte de Buenos Aires**, conservando el enfoque de grafos, pero reformulando el diseño bajo el paradigma de **Tipos Abstractos de Datos (TDA)**.

La lógica de recorridos, cálculo de rutas mínimas y manejo modular de vértices y aristas se mantuvo como eje central, pero la implementación fue adaptada a las necesidades específicas del sistema argentino, incluyendo líneas, estaciones, transbordos y tarifa fija.

Problemas a resolver

Objetivo: Determinar la ruta más corta (en términos de distancia) entre cualquier par de estaciones del sistema de subte, considerando:

- Todas las estaciones de las 5 líneas principales elegidas
- Distancias reales entre estaciones consecutivas
- Conexiones de transbordo entre líneas
- Optimización del recorrido total

Justificación

Este problema es ideal para grafos pesados porque:

- Cada estación representa un vértice
- Cada conexión entre estaciones es una arista con peso (distancia)
- Se requiere encontrar el camino mínimo entre dos puntos
- El grafo es conectado (todas las estaciones están interconectadas)

2. Modelado del problema

Representación como grafo

- Vértices (V): 83 estaciones del subte
- Aristas (E): conexiones bidireccionales entre estaciones
- Pesos (W): Distancias en kilómetros entre estaciones
- Tipo: Grafo no dirigido, conectado y con pesos positivos

Datos reales utilizados

```
Línea A: 16 estaciones, distancia promedio 0.6 km
Línea B: 17 estaciones, distancia promedio 0.7 km
Línea C: 9 estaciones, distancia promedio 0.5 km
Línea D: 17 estaciones, distancia promedio 0.65 km
Línea E: 17 estaciones, distancia promedio 0.66 km
Transbordos: 6 conexiones principales, distancia 0.1 km
```

Estructura de datos

```
// Grafo implementado con lista de adyacencias
Map<String, List<Arista>> adyacencias;

// Cada arista contiene destino y peso
class Arista {
    String destino;
    double peso; // distancia en km
}
```

3. Algoritmo de Dijkstra - Implementación

Descripción del algoritmo

El algoritmo de Dijkstra encuentra el camino más corto desde un vértice origen hacia todos los demás vértices en un grafo con pesos no negativos.

Pseudocódigo implementado

```
DIJKSTRA(grafo, origen, destino):
1. Inicializar distancias[v] = ∞ para todos los vértices
2. distancias[origen] = 0
3. Crear cola de prioridad Q con todos los vértices
4. MIENTRAS Q no esté vacía:
   a. u = extraer vértice con menor distancia de Q
   b. SI u == destino: TERMINAR
   c. PARA cada vecino v de u:
      - alt = distancias[u] + peso(u,v)
      - SI alt < distancias[v]:
        * distancias[v] = alt
        * anterior[v] = u
5. Reconstruir camino desde destino hasta origen
```

Complejidad temporal

- Con cola de prioridad: O((V + E) log V)
- Para nuestro caso: O((83 + 166) log 83) ≈ O(1.660 operaciones)
- Rendimiento medido: ~2 ms promedio por busqueda

Implementación Clave

```
public List<String> dijkstra(String origen, String destino) {
    Map<String, Double> distancias = new HashMap<>();
    Map<String, String> anteriores = new HashMap<>();
    PriorityQueue<NodoDistancia> cola = new PriorityQueue<>();

    // Inicialización
    for (String estacion : vertices.keySet()) {
        distancias.put(estacion, Double.MAX_VALUE);
    }
    distancias.put(origen, 0.0);
    cola.add(new NodoDistancia(origen, 0.0));

    // Algoritmo principal
    while (!cola.isEmpty()) {
        NodoDistancia actual = cola.poll();
        if (actual.nodo.equals(destino)) break;

        for (Arista conexion : adyacencias.get(actual.nodo)) {
            double nuevaDistancia = distancias.get(actual.nodo) + conexion.getPeso();
            if (nuevaDistancia < distancias.get(conexion.getDestino())) {
                distancias.put(conexion.getDestino(), nuevaDistancia);
                anteriores.put(conexion.getDestino(), actual.nodo);
                cola.add(new NodoDistancia(conexion.getDestino(), nuevaDistancia));
            }
        }
    }

    // Reconstruir ruta
    return reconstruirRuta(anteriores, destino);
}
```

4. Diferencias con el Algoritmo Teórico

Adaptaciones realizadas

a. Optimización de Terminación Temprana

- Teoría: Calcula distancias a todos los vértices
- Implementación: Termina al encontrar el destino
- Beneficio: Reducir el tiempo de ejecución promedio en un 60%

b. Estructura de Datos Específicas

- Teoría: Matriz de adyacencias o lista genérica
- Implementación: HashMap optimizado para búsquedas O(1)
- Ventaja: Acceso rápido a estaciones por nombre

c. Bidireccionalidad Automática

- Implementación: Al agregar una arista A→B, se agrega automáticamente B→A
- Justificación: El subte permite viajar en ambas direcciones
- Simplificación: Reducir código de carga de datos

Algoritmos alternativos investigados

a. Algoritmo A* (A-Star)

Descripción:

Extensión del algoritmo de Dijkstra que incorpora una función heurística para guiar la búsqueda más eficientemente.

Ventajas:

- Más rápido que Dijkstra en grafos grandes
- Útil cuando se conoce la posición geográfica (coordenadas) de las estaciones

Desventajas:

- Requiere una función heurística adecuada (por ejemplo, distancia euclidiana)
- Más complejo de implementar
- En grafos pequeños como este, la mejora es mínima o inexistente

Aplicabilidad:

Recomendado para sistemas de transporte muy grandes (más de 500 estaciones), especialmente con soporte geográfico o mapas reales.

b. Algoritmo Floyd-Warshall

Descripción:

Calcula todos los caminos más cortos entre todos los pares de vértices. Es un algoritmo de programación dinámica.

Ventajas:

- Precalcula todas las rutas posibles
- Las consultas posteriores son instantáneas (O(1))
- Ideal para sistemas con muchas consultas entre pares fijos

Desventajas:

- Alta complejidad temporal: $O(V^3)$ → en nuestro caso, con 83 estaciones: $O(83^3) \approx 571.787$ operaciones
- Alto consumo de memoria ($O(V^2)$)
- Tiempo de inicialización elevado

Comparación de rendimiento:

Inviable para ejecución en tiempo real si se recalcula frecuentemente, pero excelente si el sistema puede precargar los datos y responder consultas rápidamente.

c. Algoritmo de Bellman-Ford

Descripción:

Encuentra los caminos más cortos desde un vértice origen hacia todos los demás. A diferencia de Dijkstra, permite pesos negativos.

Ventajas:

- Útil en redes donde pueden existir costos negativos (por ejemplo, descuentos o penalizaciones)

Desventajas:

- Más lento que Dijkstra en grafos con solo pesos positivos
- No mejora el rendimiento en nuestro caso

Aplicabilidad:

No relevante para nuestro grafo del subte de Buenos Aires, ya que todos los pesos representan distancias o tiempos, los cuales son estrictamente positivos.

5. Implementación bajo TDA

Diseño de interface

```
public interface IGrafo {
    void agregarVertice(Vertice vertice);
    void agregarArista(String origen, String destino, double peso);
    List<String> dijkstra(String origen, String destino);
    double getPesoArista(String origen, String destino);
}

public interface IArista {
    String getDestino();
    double getPeso();
}
```

Ventajas del Enfoque TDA

1. Abstracción : Oculta detalles de implementación
2. Reutilización : Interfaces aplicables a otros problemas de gráficos
3. Mantenibilidad : Fácil cambio de algoritmos
4. Testabilidad : Interfaces claras para pruebas unitarias

6. Resultados y análisis

Casos de prueba ejecutados

```
Test 1: Ruta simple (Peru → Catedral)
- Resultado: 2 estaciones, 0.1 km, <1ms

Test 2: Ruta con transbordo (Lima → 9 de Julio)
- Resultado: 4 estaciones, 1.3 km, 2ms

Test 3: Ruta compleja (San Pedrito → Plaza de los Virreyes)
- Resultado: 15 estaciones, 8.2 km, 3ms

Test 4: Rendimiento (1000 consultas aleatorias)
- Promedio: 2.1ms por consulta
- Éxito: 100% de rutas encontradas
```

Validación del Sistema

Cobertura:

Se logra una cobertura del 100% de estaciones modeladas en el grafo.
Todas las estaciones son accesibles desde cualquier punto conectado.

Precisión:

Las distancias fueron verificadas con datos oficiales y estimaciones razonables.
Las conexiones entre estaciones reflejan el trazado real del Subte de Buenos Aires.

Robustez:

Se implementaron validaciones para:
- Estaciones inexistentes (entrada invalida)
- Consultas con la misma estación de origen y destino
- Casos sin ruta disponible (grafo desconectado, si existiera)

7. Conclusiones

Efectividad del algoritmo de Dijkstra

El algoritmo de Dijkstra resulta ser la mejor opción para este caso de uso:

- Garantía de optimización: Siempre encuentra la ruta más corta disponible
- Eficiencia adecuada: Excelente rendimiento para un grafo con menos de 100 nodos
- Simplicidad: La implementación es clara, modular y mantenible
- Robustez: Resuelve todos los casos reales del sistema de transporte sin errores

Comparación con alternativas

Dijkstra vs A*:
Dijkstra es suficiente para este tipo de grafo. A* solo mejora tiempos en grafos mucho mas grandes con coordenadas geográficas.

Dijkstra vs Floyd-Warshall:
Floyd-Warshall precalcula todas las rutas, pero tiene alto costo en tiempo y memoria.
Dijkstra es mas eficiente para consultas puntuales y dinamicas.

Dijkstra vs Bellman-Ford:
Bellman-Ford permite pesos negativos, lo cual no aplica aqui.
Dijkstra es mas rapido para grafos con pesos positivos, como distancias.

Aplicaciones futuras

El sistema actual puede ampliarse para incluir nuevas funcionalidades:

- Integrar tiempos reales de viaje y horarios programados
- Agregar soporte para frecuencia de trenes o tarifas variables
- Incorporar una interfaz grafica o mapa visual del recorrido
- Extender a otros medios de transporte: trenes, colectivos, bicicletas