Trabajo práctico obligatorio.
Algoritmos y estructuras de Datos II

Grupo N°10
Integrantes: Lautaro Abate, Gonzalo Altamirano, Dante Fermanelli, Octavio Ficer
Problema real: Google Maps y VideoJuegos con caminata automática.
Algoritmo: A*

Planteamiento

A la hora de utilizar una aplicación de GPS o mapa podremos ver que, al momento de indicarle a dónde queremos ir, se generará una ruta automática en el cual se nos indicará la cantidad de tiempo y kilómetros que nos tomará hasta llegar. El presente trabajo abordara la forma en la que se generarán las rutas en este tipo de aplicaciones. El algoritmo más parecido que soluciona este problema es el algoritmo de A*, usando una estimación (heurística) para saber cuán cerca estamos del destino seleccionado. Aplicaciones como Google Maps utilizan este algoritmo como base, pero utilizan combinaciones de diferentes técnicas ya que A* por sí solo sería demasiado lento a la hora de consultar rutas cortas. Algunas de las Técnicas utilizadas son:

- Preprocesamiento, es decir, precalcula rutas.
- Enrutamiento jerárquico en la cual, divide el mapa en niveles, empezando por autopistas y baja a niveles más detallados (avenidas, calles locales).
- Datos en tiempo real como tráfico en vivo, eventos temporales, obras o cortes,
- Inteligencia artificial, para predecir tiempos de viaje, estimar tráfico futuro.

El preprocesamiento cumple una tarea fundamental ya que guarda rutas previamente calculadas sin que se procesen cada vez que un usuario desea conocer la ruta de hacía un destino.

Objetivo

El objetivo del trabajo es **asemejarnos al algoritmo que utiliza Google Maps** u otro cualquier sistema que requiera el obtener una ruta hacia un destino de la forma más rápida y óptima al mismo tiempo, **utilizando A***, explicando y justificando la elección.

Fundamentación

¿Por qué A*?

Para resolver el problema de encontrar rutas óptimas entre dos puntos en un grafo, como ocurre en aplicaciones de navegación tipo Google Maps o en videojuegos con movimiento automático, seleccionamos el algoritmo A* (A-star) por su eficiencia y precisión.
Decidimos no utilizar Dijkstra porque, aunque este nos garantice la ruta más corta, no lo realizaría de la forma más eficiente ya que este algoritmo recorre todos los caminos posibles, lo cual consume más recursos computacionales y tiempo de procesamiento, a diferencia de A* que busca el camino más prometedor gracias a la función heurística,

evitando recorrer rutas poco relevantes, lo que hace que encuentre un camino rápido en un lapso de tiempo menor a Dijkstra.
En conclusión consideramos que A* es la mejor opción para nuestra problemática, ya que combina exactitud con eficiencia, dos características fundamentales en sistemas de búsqueda de rutas en tiempo real.

Solución

Para llevar a cabo la solución, simulamos destinos (**nodos**), conectados mediante **aristas** (**caminos**), con sus pesos (se pueden interpretar como los **minutos** que se tarda en llegar de un punto a otro).
Lo primero que necesitamos es crear una clase “Nodo”. Este contendrá el valor que se le indique (Nombre del barrio en este caso), una lista de sus vecinos y otra con los pesos. Estos pesos nos indicarán cuánto tiempo se tardará en recorrer el camino hasta el nodo destino.
La clase “Grafo” que va a contener las funciones básicas para agregar nodos y sus conexiones.

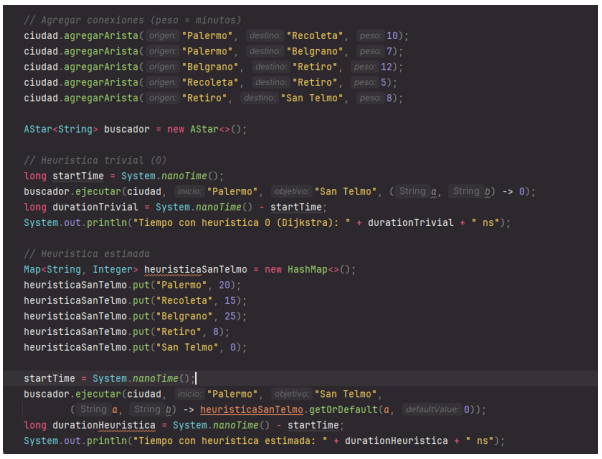
Simulación

Primero, adaptamos el código de A* visto en clase aunque la diferencia principal en el desarrollo del código de A* de nuestro TPO es la presencia del Nodo <T> a diferencia del visto en clase el cual era un nodo que soportaba solamente enteros, que ventajas nos da esto:

- Mayor escalabilidad de proyecto.
- Adaptabilidad.
- Vuelve el código más reusable.

Con esta implementación del <T> obtenemos la capacidad para poder utilizar ciudades, coordenadas, direcciones, entre otras cosas como nodos.

Creación mapa con Heurística 0 (Dijkstra) y Heurística estimada



```
// Agregar conexiones (peso = minutos)
ciudad.agregarArista( origen: "Palermo",  destino: "Recoleta",  peso: 10);
ciudad.agregarArista( origen: "Palermo",  destino: "Belgrano",  peso: 7);
ciudad.agregarArista( origen: "Belgrano", destino: "Retiro",    peso: 12);
ciudad.agregarArista( origen: "Recoleta", destino: "Retiro",    peso: 5);
ciudad.agregarArista( origen: "Retiro",   destino: "San Telmo", peso: 8);

AStar<String> buscador = new AStar<>();

// Heuristica trivial (0)
long startTime = System.nanoTime();
buscador.ejecutar(ciudad,  inicio: "Palermo",  objetivo: "San Telmo", ( String a,  String b) -> 0);
long durationTrivial = System.nanoTime() - startTime;
System.out.println("Tiempo con heuristica 0 (Dijkstra): " + durationTrivial + " ns");

// Heurística estimada
Map<String, Integer> heuristicaSanTelmo = new HashMap<>();
heuristicaSanTelmo.put("Palermo", 20);
heuristicaSanTelmo.put("Recoleta", 15);
heuristicaSanTelmo.put("Belgrano", 25);
heuristicaSanTelmo.put("Retiro", 8);
heuristicaSanTelmo.put("San Telmo", 0);

startTime = System.nanoTime();
buscador.ejecutar(ciudad,  inicio: "Palermo",  objetivo: "San Telmo",
        ( String a,  String b) -> heuristicaSanTelmo.getOrDefault(a,  defaultValue: 0));
long durationHeuristica = System.nanoTime() - startTime;
System.out.println("Tiempo con heuristica estimada: " + durationHeuristica + " ns");
```

Mapa de Adyacencia tipo matriz:

	Palermo	Recoleta	Belgrano	Retiro	San Telmo
Palermo	X	**10**	**7**	X	X
Recoleta	X	X	X	**5**	X
Belgrano	X	X	X	**12**	X
Retiro	X	X	X	X	**8**
San Telmo	X	X	X	X	X

Camino óptimo: Palermo > Recoleta (10) > Retiro (5) > San Telmo (8)
Costo total: 23

Justificación A* sobre Dijkstra

```
Camino encontrado: [Palermo, Recoleta, Retiro, San Telmo]
Costo total: 23
Tiempo con heuristica 0 (Dijkstra): 1982000 ns
Camino encontrado: [Palermo, Recoleta, Retiro, San Telmo]
Costo total: 23
Tiempo con heurística estimada: 868400 ns
```

Si bien el camino óptimo es el mismo, con heurística cero A* actúa como Dijkstra, pero al usar una heurística que estime la distancia al destino, A* reduce la exploración innecesaria y mejora el tiempo de respuesta, siendo más eficiente.

Comparativa videojuegos

Esto también aplica para los videojuegos de estrategia o RPG (como en StarCraft 2, Red Dead Redemption, etc.), que utilizan A* (o variantes) ya que es la manera más rápida gracias a la posibilidad de elegir una heurística. Estos algoritmos/métodos se llaman “Pathfinding”, y es un estándar en el desarrollo de éste.

Bibliografía usada:
https://canbayar91.medium.com/game-mechanics-2-path-finding-ab4f55c1d580
- Can Bayar “Path Finding Algorithms“

Diferencias con otros algoritmos

Algoritmo	**Tipo / Uso principal**	**Busca camino más corto**	**Complejidad típica**	**Característica clave**
A*	Búsqueda de camino óptimo con heurística	Sí	Depende de la heurística, generalmente O(E) a O(E log V)	Usa heurística para guiar la búsqueda y acelerar el proceso
Dijkstra	Encontrar camino más corto desde un nodo origen	Sí	$O(V^2)$ o O(E log V) con cola de prioridad	Explora nodos desde el origen actualizando distancias mínimas
Floyd-Warshall	Encontrar caminos más cortos entre todos los pares	Sí	$O(V^3)$	Programación dinámica para calcular distancias entre todos los pares
Kruskal	Encontrar árbol de expansión mínima (MST) Encontrar árbol de expansión mínima (MST)	No	O(E log E)	Ordena aristas por peso y agrega sin formar ciclos (union-find)
Prim	Encontrar árbol de expansión mínima (MST) Encontrar árbol de expansión mínima (MST)	No	O(E log V)	Construye MST agregando aristas de menor peso conectadas al árbol actual

Conclusión

Al analizar en profundidad el algoritmo A* para encontrar rutas óptimas en situaciones como “Google Maps” o en videojuegos (Pathfinding, en “tiempo real”), podemos concluir que A* ayuda bastante para encontrar el camino más corto de la forma más eficiente en comparación con otros algoritmos gracias a las heurísticas que permiten priorizar caminos, aunque también, este algoritmo se apoya con otras alternativas.
Simular un mapa nos permitió poner en práctica el algoritmo y confirmar los resultados esperados. Podemos decir que A* es ideal para resolver problemas a la hora de encontrar un camino óptimo donde se necesita un tiempo de respuesta rápido sin utilizar demasiados recursos computacionales.