Estructura de Datos y Algoritmos II

Curso 2024-2025

Práctica 2 – Esquema algorítmico Greedy

Alumnos:

Alejandro Doncel Delgado

Jorge Godoy Beltrán

Índice de contenidos

1. OBJETIVO

1.1 Justificación de la aplicación de algortimos Greedy

En el presente trabajo se aborda la aplicación de algoritmos **Greedy** en la resolución de problemas de optimización. A diferencia de otras estrategias como el Divide y Vencerás, el enfoque **Greedy** se caracteriza por construir la solución paso a paso, eligiendo en cada paso la opción más prometedora o localmente óptima, con la esperanza de llegar a un óptimo global.

Este método es especialmente útil en problemas donde la toma de decisiones en cada instante garantiza un resultado aceptable y en muchos casos óptimo, además de ofrecer implementaciones sencillas y eficientes en términos de tiempo de ejecución.

El propósito de este estudio es analizar en profundidad la metodología **Greedy**, destacando sus fundamentos teóricos y prácticas de implementación, para comprender las condiciones en las que este enfoque resulta adecuado y sus limitaciones. Se utilizarán ejemplos clásicos y se presentarán pseudocódigos que evidencian la lógica de implementación de estos algoritmos. De esta manera, se proporcionará una base teórica sólida para el entendimiento y la aplicación práctica de la estrategia **Greedy** en la resolución de problemas.

1.2 Descripción general del problema

El problema para resolver implica diseñar un servicio web que, dada una red, sea capaz de generar rutas óptimas entre nodos de origen y destino. Los elementos clave del problema son:

Modelado de la Red: La red se representa mediante un grafo en el que los nodos pueden representar puntos de interés, intersecciones o estaciones, y las aristas indican las conexiones entre ellos, cada una con un peso que puede representar la distancia, el tiempo o el coste asociado.

Criterio de Optimización: Se busca minimizar o maximizar un determinado parámetro, comúnmente el coste total, el tiempo de trayecto o la distancia recorrida. La tarea es construir una ruta que cumpla con el criterio de manera óptima.

2. TRABAJO EN EQUIPO

2.1 Organización y roles

Con el fin de llevar a cabo la práctica de manera ordenada y eficiente, se ha optado por establecer un rol de liderazgo y roles específicos para cada integrante del equipo. El propósito es asegurar que cada persona tenga una responsabilidad clara y una contribución definida al proyecto, de la siguiente forma:

- **Alejandro Doncel [AD]**: Asume la coordinación general, planifica la estructura del proyecto en el repositorio, métodos y clases dependientes, documenta las funciones y realiza la depuración de los algoritmos.
- **Jorge Godoy [JG]**: Desarrolla la parte principal del código (algoritmos iterativos y recursivos).

2.2 Tabla de distribución de tareas

Tarea	Responsable(s)	Fecha de Finalización
Configuración del repositorio y estructura base del proyecto	[AD]	27 de marzo 2025
Implementación de algoritmos DijkstraV1 y DijkstraNaive	[JG]	27 de marzo 2025
Implementación de algoritmos con cola de prioridad (DijkstraPQ, BestFirst)	[AD], [JG]	30 de marzo 2025
Desarrollo e integración de heurísticas para A* y DijkstraAStar	[JG]	30 de marzo 2025
Implementación del algoritmo DijkstraBestFirstWPP (max bottleneck)	[AD]	1 de abril 2025
Creación del generador de grafos aleatorios y automatización de pruebas	[AD]	4 de abril 2025
Ejecución de pruebas y recogida de métricas (tiempo y aristas)	[AD], [JG]	5 de abril 2025
Análisis teórico de complejidades y validación experimental	[JG]	5 de abril 2025
Diseño y generación de gráficas con MATLAB (tiempo, aristas, R^2, etc.)	[AD]	9 de abril 2025
Redacción de la memoria técnica y anexos	[AD], [JG]	9 de abril 2025
Revisión final, formateo y entrega	[AD], [JG]	13 de abril 2025

3. ANTECEDENTES

El presente trabajo tiene como objetivo aplicar el método Greedy a la resolución del problema de la generación de rutas óptimas en redes, con aplicaciones concretas en el control del tráfico aéreo, análisis de datos espaciales y optimización de rutas.

Para abordar este problema, se han evaluado diferentes enfoques algorítmicos, haciendo especial énfasis en aquellos que permiten la toma de decisiones rápidas mediante criterios locales. Este análisis comparativo representa una oportunidad para profundizar en conceptos clave de diseño y análisis de algoritmos, evidenciando cómo la elección de una estrategia Greedy – basada en la selección de la mejor opción inmediata – puede traducirse en una mejora significativa del rendimiento computacional, especialmente en entornos donde se requiere respuestas en tiempo real.

A lo largo de la práctica, se han explorado distintos enfoques algorítmicos, evaluando sus alternativas con el propósito de encontrar un equilibrio entre precisión en la solución y eficiencia en términos de tiempo de ejecución. Las principales soluciones consideradas han sido

3.1 Alternativas contempladas y solución final adoptada

3.1.1 Enfoque Greedy Iterativo Básico dijkstraV1 y dijkstraNaive

DijkstraV1

Se implementa de forma iterativa y se basa en una exploración exhaustiva de todos los vértices no procesados para seleccionar, en cada iteración, aquel con la mínima distancia acumulada desde el origen.

Contexto: La idea es sencilla y se fundamenta en tomar decisiones Greedy de forma local: para cada vértice se actualiza la distancia considerando todas las posibles conexiones desde el vértice actual.

Limitación: Esta solución tiene una complejidad de aproximadamente $O(V^2)$ cuando se evalúa cada vértice, lo que la hace adecuada únicamente para grafos pequeños o como método de validación y depuración.

DijkstraNaive

Es una variante similar, en la que se exploran únicamente los vecinos directos del vértice actual en lugar de todos los posibles pares entre vértices.

Limitación: Aunque se reduce ligeramente el número de comparaciones innecesarias, la complejidad sigue siendo $O(V^2)$, limitando su utilidad en redes de gran tamaño.

3.1.2 Enfoque Greedy Mejorado con Cola de Prioridad

DijkstraPQ

Se introduce una cola de prioridad para extraer de forma eficiente el vértice con la menor distancia acumulada.

Contexto: La utilización de una estructura de datos adecuada permite que, en cada iteración, el algoritmo seleccione de manera óptima el vértice "más prometedor" en tiempo O(log V), lo que repercute en una mejora significativa de la eficiencia computacional (complejidad O((E + V) log V)).

Ventaja: Este enfoque es el más adecuado para aplicaciones en tiempo real sobre redes de gran tamaño, ya que equilibra la toma de decisiones Greedy con un manejo eficiente de la información.

3.1.3 Adaptaciones Greedy Basadas en Best-First y A*

DijkstraBestFirst

Emplea una variante del algoritmo Greedy que utiliza una cola de prioridad de elementos triples, donde cada entrada contiene el vértice, su distancia acumulada y el vértice predecesor.

Contexto: Esta versión elimina la necesidad de una estructura separada para el control de vértices visitados, integrando en la cola el criterio de selección y facilitando una actualización dinámica de las distancias.

A* (aStar)

Incorpora heurísticas basadas, por ejemplo, en la distancia euclidiana, para guiar la exploración de la búsqueda Greedy.

Contexto: La función evaluadora (f = g + h) permite que el algoritmo combine la información del camino recorrido (g) y la estimación restante (h). Esto mejora la eficiencia en contextos espaciales, garantizando que la búsqueda se enfoque en caminos con mayor probabilidad de ser óptimos.

DijkstraAStar

Se integra lo mejor de ambos mundos utilizando la estructura de cola de prioridad y adaptando la estrategia de actualización de distancias para trabajar con un valor f (g + h).

Contexto: Esta versión busca optimizar la toma de decisiones al considerar simultáneamente la distancia acumulada y la estimación del costo restante, mostrando así la flexibilidad del enfoque Greedy cuando se complementa con heurísticas.

3.1.4 Problema Específico: dijkstraBestFirstWPP

Se adapta el enfoque Greedy para maximizar el "ancho" del camino (es decir, la capacidad o el mínimo valor de una arista a lo largo del camino, conocido como "bottleneck").

Contexto: Para este problema se utiliza una cola de prioridad invertida, en la que se priorizan los caminos que mantengan el mayor ancho posible.

Ventaja: Este método es ideal cuando se busca maximizar el valor mínimo de la capacidad de un camino, y se demuestra la versatilidad del enfoque Greedy para resolver tanto problemas de minimización (como el camino mínimo) como de maximización.

4. ESTUDIO TEÓRICO Y DE IMPLEMENTACIÓN

4.1 Introducción

Tras haber presentado el contexto y desarrollo del sistema de rutas óptimas mediante algoritmos Greedy, en esta sección se analizarán en detalle cada una de las soluciones implementadas. El objetivo es estudiar su funcionamiento, su eficiencia teórica y práctica, y compararlas entre sí. Para mantener un enfoque claro y estructurado, el estudio de cada algoritmo se organizará del siguiente modo:

- **Estudio de la implementación**: Se describen los aspectos esenciales del funcionamiento de cada algoritmo, haciendo hincapié en las decisiones de diseño y su relación con la estrategia Greedy.
- **Estudio teórico**: Se analiza la complejidad temporal y espacial de cada versión, y se destacan sus ventajas o limitaciones según el tipo de grafo.
- **Estudio experimental**: Se validan las implementaciones mediante pruebas sobre grafos reales y generados, comparando los resultados obtenidos con las predicciones teóricas.
- **Anexo**: Se incluirán pseudocódigos, diagramas y referencias relevantes para complementar el análisis. Se trata de una sección extensa e introductoria de cada uno de los algoritmos, más adelante se compararán unos con otros, agrupándolos en iterativos o recursivo.

4.2 DijkstraV1

Se trata de un algoritmo voraz que resuelve el problema del camino más corto en un grafo dirigido y ponderado desde un vértice origen a un vértice destino. Utiliza una estructura auxiliar para registrar las distancias mínimas conocidas y selecciona en cada iteración el vértice con menor distancia aún no procesado, actualizando las distancias de sus vecinos si se encuentra un camino más corto. El proceso finaliza cuando se alcanza el destino o no quedan vértices alcanzables.

Estudio de implementación - Funcionamiento del algoritmo

Inicialización de estructuras

Se preparan dos estructuras clave: un conjunto de vértices restantes por visitar y una estructura personalizada que almacena las distancias mínimas conocidas hasta cada vértice junto a su predecesor. En esta fase también se inicializan las distancias desde el

vértice de origen a todos los demás, usando `Double.MAX_VALUE` para aquellos vértices no adyacentes directamente al origen.

Selección del vértice de menor distancia

Esta es la parte más relevante del algoritmo en cuanto a su comportamiento voraz. En cada iteración se selecciona, entre los vértices restantes, aquel cuya distancia desde el origen sea mínima. Esta selección condiciona la eficiencia general, ya que se repite tantas veces como vértices haya.

Relajación de aristas

Una vez seleccionado el vértice actual u, se recorren sus vecinos aún no procesados. Si la distancia conocida desde el origen hasta el vecino v puede reducirse pasando por u, se actualiza esta distancia. Esta operación clave permite ir construyendo el camino más corto de manera incremental.

Estudio Teórico - Análisis de la complejidad del algoritmo

Sea *V* el número de vértices y *E* el número de aristas del grafo.

Mejor caso: O(V^2)

El mejor caso ocurre cuando el grafo es poco denso y no se requiere actualizar muchas distancias. Aun así, dado que la estructura `dp.getVerticeMinDistancia()` recorre linealmente el conjunto de vértices restantes para obtener el de menor distancia, cada iteración cuesta **O(V^2)**, y hay hasta V iteraciones

Aunque las actualizaciones dentro del bucle interno se minimicen, el coste sigue dominado por la selección del vértice mínimo, lo que da una complejidad total de **O(V^2)**

Peor Caso: O(V^2)

En el peor caso, el grafo es denso, es decir, E≈V^2, y se realizan muchas actualizaciones de distancia en `dp`. Sin estructuras de datos eficientes como una cola de prioridad, la selección del vértice mínimo en cada iteración sigue siendo O(V), y para cada vértice se puede llegar a evaluar todos los demás como posibles vecinos.

Esto genera un bucle anidado que recorre hasta V vértices por cada uno de los V, resultando nuevamente en una complejidad total de **O(V^2)**

Análisis de eficiencia

Ventajas: implementación clara y sencilla, adecuada para grafos de tamaño pequeño o moderado, por ejemplo, con menos de 1000 vértices. No requiere estructuras complejas como montículos o colas de prioridad, lo que facilita su comprensión y depuración. Además, permite observar el funcionamiento básico del algoritmo de Dijkstra y visualizar las aristas exploradas en cada paso.

Desventajas: a selección lineal del vértice con menor distancia reduce notablemente su eficiencia en grafos grandes, ya que impone una complejidad cuadrática. No está optimizado para grafos densos ni para aquellos con un alto número de aristas. Esta ineficiencia se corrige en versiones mejoradas del algoritmo, que está explicadas más adelante.

4.3 DijkstraNaive

Se trata de una versión mejorada del algoritmo de Dijkstra que sigue un enfoque voraz para encontrar el camino más corto entre dos vértices en un grafo dirigido y ponderado. A diferencia de la versión anterior, esta implementación optimiza el recorrido considerando únicamente los vecinos directos del vértice actual, evitando iteraciones innecesarias sobre todos los vértices restantes.

Se apoya en una estructura auxiliar para mantener las distancias mínimas conocidas y actualiza dichas distancias cuando se detectan caminos más eficientes. El proceso concluye al llegar al destino o al agotar los vértices alcanzables.

Estudio de implementación - Funcionamiento del algoritmo

El algoritmo parte de una validación inicial para asegurar que tanto el vértice de origen como el de destino estén presentes en el grafo. A continuación, inicializa la estructura de distancias **CustomMap** y el conjunto de vértices restantes.

Una de las principales mejoras respecto a versiones anteriores es el uso del método `getNeighbors(u)`, lo que permite iterar únicamente sobre los vecinos directos del vértice actual. Esto evita recorrer todos los vértices en cada iteración y mejora la eficiencia general del algoritmo.

Otro aspecto destacable es el control explícito de los pesos mediante `Double.isNaN(peso)`, lo que garantiza que solo se procesen aristas válidas. Además, se contabilizan las aristas exploradas durante el proceso, permitiendo analizar el rendimiento del algoritmo en distintas ejecuciones. Finalmente, se construye y retorna un objeto Camino, que encapsula tanto la solución como información adicional sobre el número de aristas exploradas.

Estudio Teórico - Análisis de la complejidad del algoritmo

Sea V el número de vértices y E el número de aristas del grafo.

Mejor caso: O(V^2)

El mejor caso se da en un grafo densamente conectado con pesos uniformes. En cada iteración se selecciona el vértice con la menor distancia conocida mediante una búsqueda lineal en el conjunto de vértices restantes, lo que implica un coste de O(V) por iteración. Aunque las actualizaciones de distancias se realizan solo sobre los vecinos directos, el coste total sigue estando dominado por la selección lineal del nodo mínimo.

Peor Caso: O(V^2)

En el peor caso, el algoritmo sigue realizando V iteraciones donde en cada una se busca el vértice de menor distancia en un conjunto de hasta V vértices. A esto se le suma el coste de revisar los vecinos, que en el caso más desfavorable puede acercarse a V por iteración (si el grafo es muy denso). Como no se utiliza una estructura como un heap, el rendimiento se mantiene en **O(V^2)** tanto en el mejor como en el peor caso.

Ambos algoritmos tienen la misma complejidad en su forma básica, pero dijkstraNaive puede ser más eficiente en grafos dispersos porque solo revisa los vecinos directos de cada nodo. En grafos densos, la diferencia de rendimiento es mínima porque la selección del nodo mínimo sigue siendo O(V).

Análisis de Eficiencia

Ventajas: Este algoritmo mejora respecto a versiones anteriores al evitar recorrer todos los vértices en cada iteración. Al usar únicamente los vecinos directos de cada vértice `getNeighbors`(), se reduce notablemente el número de operaciones en grafos dispersos, mejorando su rendimiento práctico sin alterar la complejidad teórica. Además, su implementación es sencilla y clara.

Desventajas: A pesar de su mejora, sigue teniendo una complejidad cuadrática **O(V^2)** debido a la búsqueda lineal del vértice con menor distancia en cada iteración. En grafos grandes o muy densos, esto limita su eficiencia.

4.4 DijkstraPQ

Esta versión optimizada del algoritmo de Dijkstra emplea una cola de prioridad para gestionar los vértices según su distancia mínima conocida al origen. Esta estructura permite seleccionar de forma eficiente el vértice más prometedor en cada iteración, reduciendo el coste de búsqueda con respecto a las versiones anteriores que usaban estructuras menos eficientes como conjuntos o listas. Al mantener un seguimiento de los vértices ya visitados, se evita el reprocesamiento innecesario y se garantiza que cada vértice se procesa una sola vez.

El algoritmo continúa siendo voraz, avanzando siempre hacia el vértice con la menor distancia estimada. A medida que explora los vecinos del vértice actual, actualiza sus distancias si encuentra un camino más corto.

Estudio de implementación - Funcionamiento del algoritmo

El algoritmo **dijkstraPQ** representa una mejora significativa en comparación con versiones anteriores, especialmente en cuanto a eficiencia. Comienza validando que el grafo contiene tanto el vértice de origen como el de destino. A continuación, inicializa tres estructuras principales: un CustomMap para registrar las distancias mínimas y predecesores, una **PriorityQueue** que actúa como cola de prioridad para seleccionar el siguiente vértice más cercano, y un conjunto de visitados para evitar procesar el mismo nodo más de una vez.

Todos los vértices se inicializan con distancia infinita, salvo el origen que se establece en 0, y se añade a la cola de prioridad. El bucle principal del algoritmo extrae el vértice con menor distancia de la cola y, si no ha sido visitado, evalúa sus vecinos.

Para cada vecino no visitado, se calcula la distancia provisional sumando el peso de la arista al coste acumulado del vértice actual. Si esta nueva distancia es mejor que la registrada, se actualiza en el CustomMap y se reintroduce el vecino en la cola. Además, se registra la arista explorada, y el proceso finaliza al alcanzar el destino o vaciar la cola.

Finalmente, se devuelve un objeto Camino con la información necesaria, incluyendo el número de aristas exploradas.

Estudio Teórico - Análisis de la complejidad del algoritmo

Emplea una cola de prioridad para optimizar la selección del vértice con menor distancia, mejorando significativamente el rendimiento respecto a versiones anteriores. La operación principal del algoritmo es la extracción del vértice con menor distancia, que ahora se realiza en tiempo logarítmico gracias a la cola de prioridad. Además, cada vértice se procesa solo una vez, y cada arista se analiza como mucho una vez para actualizar posibles distancias más cortas.

Mejor caso: (O(E + V log V))

Este escenario se da cuando el grafo es denso y la cola de prioridad mantiene un número reducido de elementos activos. Cada uno de los V vértices es extraído una vez de la cola, lo que requiere O(V log V), y se procesan las E aristas en tiempo lineal, resultando en una complejidad total de O(E + V log V).

Peor Caso (O(E + V log V))

Incluso en el peor caso, la complejidad se mantiene en O(E + V log V), aunque con más operaciones de actualización de distancias. Cada vez que se encuentra un camino más corto, se actualiza la cola de prioridad con un nuevo valor, lo que implica múltiples inserciones en O(log V). No obstante, el uso eficiente de la cola garantiza que el rendimiento siga siendo mucho mejor que en versiones previas del algoritmo con búsqueda lineal.

Análisis de eficiencia

Ventajas: El uso de una cola de prioridad optimiza significativamente la selección del vértice con menor distancia, reduciendo el coste de esta operación de O(V) a O(log V). Esto permite que el algoritmo sea mucho más eficiente, especialmente en grafos grandes y dispersos. Además, solo se procesan las aristas necesarias, lo cual evita exploraciones redundantes y mejora el rendimiento general.

Desventajas: Aunque la complejidad teórica mejora notablemente respecto a versiones anteriores, pueden producirse múltiples inserciones duplicadas para un mismo vértice si se encuentra un mejor camino después de haberlo insertado previamente, lo que genera cierto sobrecoste en operaciones logarítmicas.

4.5 DijkstraBestFirst

Este algoritmo es una versión del clásico de Dijkstra que mezcla una estrategia de búsqueda Greedy con el cálculo de distancias mínimas. Funciona utilizando una cola de prioridad para elegir el vértice más prometedor, teniendo en cuenta la distancia acumulada hasta ese momento y la heurística que lo acerca al destino.

A medida que avanza, va actualizando las distancias de los vértices vecinos y explorando las aristas del grafo, siempre priorizando aquellos vértices que ofrecen el camino más corto hacia el objetivo. Este enfoque resulta muy útil en situaciones donde la heurística puede llevar al algoritmo a su meta de manera más eficiente que el método tradicional de Dijkstra.

Estudio de implementación - Funcionamiento del algoritmo

El algoritmo hace uso de varias estructuras de datos clave:

Inicialización

Primero, el algoritmo verifica que tanto el vértice de origen como el de destino existan en el grafo. Si no es así, se lanza una excepción. Luego, se inicializan las distancias para cada vértice como infinito, excepto para el origen, al que se le asigna una distancia de cero.

Selección del Vértice con la Menor Distancia

En cada iteración, el vértice con la menor distancia acumulada (es decir, el más prometedor para llegar al destino) se extrae de la cola de prioridad. Si ese vértice es el destino, se termina el proceso.

Actualización de Distancias

Para cada vértice vecino del vértice actual, se calcula una nueva distancia acumulada. Si esta nueva distancia es menor que la previamente registrada en el mapa, se actualiza y se agrega el vértice vecino a la cola de prioridad para su posterior exploración.

El algoritmo finaliza cuando se extrae el vértice destino de la cola de prioridad o cuando se han explorado todos los vértices alcanzables. En ambos casos, se genera un objeto Camino que contiene la ruta más corta encontrada y la cantidad de aristas exploradas.

Estudio Teórico - Análisis de la complejidad del algoritmo

Sea V el número de vértices y E el número de aristas del grafo.

Mejor Caso (O(E + V log V))

En el mejor caso, cuando el grafo está bien distribuido y la heurística "Best-First" puede ayudar a reducir el número de nodos explorados, la complejidad del algoritmo se calcula como sigue:

1. **Inserción y extracción en la cola de prioridad (PriorityQueue)**: La inserción y extracción de un vértice en la cola de prioridad se realiza en O(log V).
2. **Extracción de nodos**: Cada vértice se extrae de la cola de prioridad una vez. Esto ocurre en O(log V) tiempo, ya que cada operación de extracción tiene un coste de O(log V) y se realiza V veces.
3. **Relajación de aristas**: La relajación de cada arista, es decir, la actualización de las distancias de los vértices vecinos se realiza una vez por cada arista. Esto ocurre en O(E log V), ya que por cada arista se realiza una posible inserción en la cola de prioridad, lo cual tiene un coste de O(log V).

En total, la complejidad temporal en el mejor caso es: **(O(E + V log V))**

Peor Caso (O(E + V log V))

En el peor caso, cuando la heurística no ayuda a reducir el número de exploraciones, el algoritmo se comporta de manera similar a la versión estándar de Dijkstra con cola de prioridad, lo que lleva a la siguiente complejidad:

1. **Extracción de nodos**: Cada vértice sigue siendo extraído de la cola de prioridad una vez, lo que da una complejidad de O(V log V).
2. **Relajación de aristas**: Cada arista se relaja una vez, lo que da una complejidad de O(E log V).

Por lo tanto, en el peor caso, la complejidad temporal también es **(O(E + V log V)).**

Análisis de eficiencia

Ventajas: El uso de una cola de prioridad mejora significativamente la eficiencia al reducir el coste de seleccionar el vértice con la distancia más corta a O(log V), lo que resulta especialmente útil en grafos grandes y dispersos. Además, la heurística "Best-First" permite explorar los vértices más prometedores primero, reduciendo el número de aristas exploradas y mejorando el rendimiento general del algoritmo.

Desventajas: Si la heurística no es eficaz, el algoritmo se comporta de manera similar a Dijkstra estándar, lo que elimina las ventajas esperadas. Además, pueden ocurrir inserciones duplicadas de vértices en la cola de prioridad si se encuentra un mejor camino, lo que genera un sobrecoste en las operaciones logarítmicas. También, el uso de memoria puede ser elevado, especialmente en grafos muy grandes, debido al almacenamiento de la cola de prioridad y las distancias.

4.6 AStar

El algoritmo A* es una técnica de búsqueda que se utiliza para encontrar el camino más corto entre dos puntos en un grafo. Combina el algoritmo de Dijkstra con una heurística para hacerlo más eficiente. Funciona con una función de coste que se expresa como **f(n) = g(n) + h(n)**, donde g(n) representa el coste acumulado hasta el vértice n y h(n) es una estimación del coste restante hasta el destino. A* se enfoca en explorar los vértices más prometedores, lo que a menudo lo hace más eficaz que Dijkstra.

Este algoritmo utiliza una cola de prioridad para gestionar los vértices que aún no se han procesado y una lista cerrada para aquellos que ya han sido analizados. Además, emplea una heurística basada en la distancia euclidiana para estimar el coste hasta el destino, lo que mejora su rendimiento en grafos donde las distancias espaciales son relevantes.

Estudio de implementación - Funcionamiento del algoritmo

El algoritmo A* comienza asignando a todos los vértices del grafo un coste acumulado y una suma de coste heurístico elevados, inicialmente `Double.MAX_VALUE`, salvo al vértice de origen, que se inicializa con un coste acumulado de cero y su suma de coste heurístico como la distancia estimada hacia el destino, calculada con la heurística de distancia euclidiana. Esta heurística guía la búsqueda, priorizando aquellos vértices más cercanos al objetivo.

En cada iteración, el algoritmo extrae el vértice con la menor suma de coste, coste acumulado más heurística de la cola de prioridad openList. A continuación, evalúa sus vecinos, calculando su coste provisional, y actualizando su coste acumulado y la suma de coste heurístico si se encuentra un camino mejor. Si un vértice es actualizado, se reinsertará en la cola de prioridad para ser explorado más tarde.

Este proceso se repite hasta que se extrae el destino de la cola o se exploran todos los vértices alcanzables. Al final, se devuelve el camino más corto desde el origen hasta el destino, incluyendo las aristas exploradas y la distancia total recorrida.

Estudio Teórico - Análisis de complejidad del algoritmo

El análisis de la complejidad de A* se basa en los mismos principios que otros algoritmos de búsqueda de caminos, como Dijkstra, pero con la inclusión de una función heurística para optimizar la exploración de los vértices. A continuación, se detalla el análisis en los mejores y peores casos.

Mejor Caso (O(V log V))

El mejor caso ocurre cuando la heurística h es perfecta, es decir, cuando predice exactamente el costo real del camino desde un vértice hasta el destino. En este caso, A* sigue la ruta óptima sin explorar caminos innecesarios, lo que reduce significativamente el número de vértices y aristas explorados.

La complejidad del algoritmo, en este escenario, se reduce a O(V log V), donde V es el número de vértices, ya que solo se exploran los vértices relevantes para encontrar el camino más corto. El coste logarítmico proviene de las operaciones de inserción y

extracción en la cola de prioridad, y dado que cada vértice se extrae solo una vez, el número de operaciones es proporcional a O(V log V).

Peor Caso (O(E + V log V))

En el peor caso, cuando la heurística es inexacta o igual a cero (h = 0), A* se comporta igual que el algoritmo de Dijkstra. Esto significa que el algoritmo tiene que explorar todas las aristas del grafo y puede acabar evaluando todos los vértices antes de encontrar el camino más corto. En este caso, el comportamiento del algoritmo es similar al de Dijkstra, con una complejidad de O(E + V log V), donde E es el número de aristas y V es el número de vértices.

Análisis de eficiencia

Ventajas: El algoritmo A* es muy eficiente cuando se usa una heurística adecuada, ya que reduce significativamente el número de vértices y aristas exploradas en comparación con otros algoritmos como Dijkstra. Esto lo hace ideal para aplicaciones donde la heurística puede guiar la búsqueda, garantizando encontrar el camino óptimo de manera rápida, siempre y cuando la heurística sea admisible.

Desventajas: Si la heurística es inadecuada o nula, A* puede comportarse de forma similar a Dijkstra, lo que lleva a un mayor coste computacional al explorar más vértices y aristas. Su complejidad sigue siendo O(E + V log V) en el peor caso, lo que puede ser ineficiente para grafos grandes o densos.

4.7 DijkstraAStar

El algoritmo Dijkstra-A* es una combinación de los algoritmos de Dijkstra y A*, diseñado para mejorar la eficiencia en la búsqueda de caminos en grafos. Mientras que Dijkstra es un algoritmo de búsqueda de caminos más corto que explora exhaustivamente todos los vértices y aristas, A* utiliza una heurística para guiar la búsqueda, favoreciendo los caminos más prometedores.

Estudio de implementación - Funcionamiento del algoritmo

El algoritmo **Dijkstra-A*** sigue una estructura similar al algoritmo de Dijkstra estándar, pero con una mejora significativa al incorporar una heurística para guiar la exploración del grafo. Aquí se describe su funcionamiento detallado, que permite una búsqueda más eficiente al buscar el camino más corto entre dos vértices.

Verificación de nodos:

Se comprueba que los vértices de origen y destino existan en el grafo. Si no, se lanza una excepción.

Inicialización:

Se crea un mapa `dp` para almacenar las distancias y predecesores. Se inicializa una cola de prioridad `pq` con la fórmula f=g+h, donde g es la distancia desde el origen y h es la heurística. La distancia del origen se establece en 0 y se calcula su heurística.

Bucle Principal:

- Se extrae el vértice con el menor valor de f de la cola de prioridad.
- Si el vértice extraído es el destino, se termina el bucle.
- Si no, se actualizan las distancias de sus vecinos y se añaden a la cola de prioridad si el nuevo camino es mejor.

Terminación:
El algoritmo termina cuando se extrae el destino de la cola de prioridad o cuando la cola está vacía. Se devuelve el camino más corto y las aristas exploradas.

Análisis de complejidad

Mejor Caso (O(E log V))

El mejor escenario se presenta cuando la heurística h(n)h(n)h(n) es perfecta, es decir, cuando estima exactamente el costo real restante. En este caso, el algoritmo sigue un camino directo sin explorar nodos innecesarios, funcionando de manera similar a A*.

La complejidad depende de las operaciones de la cola de prioridad:

- **Extracción mínima**: O(1) por cada nodo.
- **Actualización de distancias**: O(log V) por cada arista evaluada.

Si el grafo es denso, el peor caso podría ser **O(V^2)**, pero el comportamiento real depende del número de aristas y vértices.

Peor Caso (O(V^2))

En el peor caso, cuando la heurística no proporciona ninguna ventaja, por ejemplo, si h(n)=0 para todos los nodos, el algoritmo funciona de manera similar a Dijkstra puro. Esto lleva a explorar todos los nodos, sin ningún tipo de reducción en el número de exploraciones.

La complejidad en este caso es **O(V^2)**, que corresponde a la complejidad de Dijkstra cuando se usa una matriz de adyacencia y no hay una heurística útil. Esto ocurre cuando el grafo es denso o la heurística no contribuye al proceso.

Análisis de eficiencia

Ventajas: El algoritmo Dijkstra-A* es muy eficiente cuando se utiliza una heurística efectiva, ya que combina lo mejor de Dijkstra y A*, reduciendo significativamente el número de nodos explorados. Esto se logra gracias a la cola de prioridad, que optimiza la selección de nodos y permite que el algoritmo se enfoque solo en los caminos más prometedores, lo que mejora la velocidad en grafos grandes.

Desventajas: La principal desventaja de Dijkstra-A* radica en que su rendimiento depende en gran medida de la calidad de la heurística utilizada. Si la heurística es inapropiada o se establece como 0, la eficiencia se ve drásticamente reducida, con un rendimiento similar al de Dijkstra estándar, lo que puede resultar ineficiente, especialmente en grafos densos.

4.7 DijkstraBestFirstWPP

El algoritmo **Dijkstra Best-First WPP** es una variante del algoritmo Dijkstra que, en lugar de buscar el camino de menor costo, se enfoca en maximizar el **ancho de banda mínimo** en el camino entre dos vértices. Es decir, su objetivo no es encontrar la ruta más corta, sino aquella en la que el mínimo peso de las aristas a lo largo del camino sea lo más grande posible.

Este enfoque es útil en problemas donde se necesita garantizar un ancho de camino mínimo, como en redes de comunicaciones, donde se prioriza el flujo más amplio posible entre los nodos. Aunque no está orientado a la optimización de costos, permite encontrar caminos con las mejores capacidades de ancho de banda dentro de un grafo.

Estudio de implementación - Funcionamiento del algoritmo

Verificación de Nodos

El algoritmo comienza verificando si los vértices de origen y destino están presentes en el grafo. Si alguno de estos vértices no existe, se lanza una excepción, asegurando que el algoritmo solo procese datos válidos.

Inicialización

Se inicializa un mapa `dp` donde se almacenarán los predecesores de cada nodo y el ancho de banda máximo encontrado hasta ese nodo. La cola de prioridad `pq` se utiliza para procesar los nodos en orden descendente de ancho de banda. Se asigna un valor de infinito al ancho de banda del vértice de origen y -infinito a los demás vértices, representando que aún no se ha encontrado un camino para ellos.

Bucle Principal

Dentro del bucle principal, el algoritmo extrae el nodo con el mayor ancho de banda disponible de la cola de prioridad. Si el nodo extraído es igual al destino, se termina el bucle, ya que se ha encontrado el camino, y si el ancho de banda actual es menor que el registrado en `dp` para ese nodo, se ignora, evitando recorridos innecesarios.

Luego, se actualiza el valor de ancho de banda máximo para ese nodo y se exploran sus vecinos. Para cada vecino, se calcula el nuevo ancho de banda como el mínimo entre el valor actual del nodo y el peso de la arista que conecta al vecino. Si este nuevo ancho de banda es mayor que el previamente registrado en `dp`, se actualiza y se agrega el vecino a la cola de prioridad.

Retorno del Resultado

Si no se ha encontrado un camino, el algoritmo retorna null. Si se encuentra un camino, se devuelve el recorrido y el **bottleneck**, que es el ancho de banda mínimo del camino encontrado, así como el número total de aristas exploradas durante la ejecución del algoritmo.

Análisis de complejidad

Mejor Caso (O(E log V))

En el mejor de los casos, cuando se encuentra un camino con un buen ancho de banda de forma rápida, el algoritmo realiza menos exploraciones. La extracción de nodos de la cola de prioridad se realiza en O(1) por cada nodo. La actualización de los anchos de banda para cada vecino ocurre en O(log V), lo que da una complejidad total de O(E log V), donde E es el número de aristas y V es el número de vértices.

Peor Caso (O(E log V))

En el peor de los casos, si el grafo es denso o si los anchos de banda no permiten optimizar la búsqueda rápidamente, el algoritmo necesitará recorrer muchas aristas para encontrar el mejor camino. Esto resulta en una complejidad de O (E log V). En un grafo denso, el número de aristas E puede ser de hasta O(V^2), lo que hace que la complejidad en el peor caso sea **O((V^2 log V).**

Análisis de eficiencia

Ventajas: Encuentra el camino con el mayor ancho de banda disponible, crucial en redes de comunicación, haciendo uso de una cola de prioridad que asegura explorar primero los caminos más eficientes. Por otro lado aprovecha la robustez y eficiencia de Dijkstra para grafos.

Desventajas: En grafos con muchas aristas, la complejidad puede aumentar significativamente, por lo tanto, la eficiencia puede verse afectada por cómo están distribuidos los anchos de banda. Solo maximiza el ancho de banda, sin considerar otros factores como distancia o tiempo.

5. ESTUDIO EXPERIMENTAL

5.1 Metodología de experimentación

Para sacar conclusiones sobre el rendimiento de la implementación de nuestros algoritmos se han realizado una serie de pruebas para después analizar los resultados de tiempos obtenidos. Como conjunto de datos usados podemos diferenciar:

Conjuntos de datos reales

Estos ficheros contienen coordenadas que representan ubicaciones reales o realistas, lo que permite comprobar el comportamiento de los algoritmos en situaciones donde los datos no siguen patrones sintéticos.
Se hace uso de los siguientes archivos

- **graphEDAland.txt** ⇒ Archivo con la red reducida de carreteras de EDAland.
- **graphEDAlandLarge.txt** ⇒ Archivo con la red nacional de carreteras de EDAland, red de carreteras completa.

- **graphEDAlandPositions.txt** ⇒ Archivo con la red reducida de carreteras de EDAland, junto con la posición (x, y) de cada uno de sus vértices, que servirá para su visualización.
- **graphEDAlandLargePositions.txt** ⇒ Archivo con la red nacional de carreteras de EDAland, red de carreteras completa y con la posición (x, y) de cada uno de sus vértices y que servirá para su visualización.

Conjuntos de datos generados

Se ha implementado un generador de grafos aleatorios que nos permitirá probar el rendimiento de los algoritmos implementados para diferentes tamaños de redes.

Para evaluar el desempeño del generador, se han creado grafos con tamaños crecientes, definidos en 5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000, 60000. Esta variación en la magnitud del conjunto permite analizar cómo evoluciona el tiempo de ejecución y verificar experimentalmente si los resultados coinciden con las complejidades teóricas esperadas.

5.2 Toma de medidas y análisis

Medidas

Para el conjunto de datos reales, cada uno de los algoritmos considerados ha sido ejecutado múltiples veces sobre cada conjunto de datos generado.

Por otra parte, para el conjunto generado, el propio generador esta implementado para proporcionar todos los tiempos y aristas exploradas con las medidas para cada algoritmo en cada iteración de tamaño, facilitando así la recogida de los datos.

Este último nos permitirá obtener un análisis más representativo que con los datasets proporcionados.

Procesado de datos

Para el procesado y análisis de los datos se ha utilizado MATLAB como herramienta principal. Concretamente, se ha llevado a cabo:

- **Generación de gráficas** que representan la relación entre el tamaño del grafo (n) y el tiempo medio y aristas exploradas de ejecución registrado.
- **Cálculo de ajustes** mediante modelos de ajuste, acompañados por la determinación del coeficiente R^2. Esto ha permitido evaluar de forma cuantitativa la correspondencia entre los resultados experimentales obtenidos y las curvas teóricas esperadas.

5.3 Resultados de la ejecución e interpretación

La siguiente tabla resume los resultados empíricos obtenidos, contrastados con las expectativas teóricas. Se ha medido el ajuste tanto a un modelo cuadrático $O(n\text{^}2)$ como

a uno subcuadrático $O(nlogn)$, considerando el coeficiente R^2 como indicador de calidad del ajuste.

Algoritmo	Complejidad Teórica	R^2 Ajuste Cuadrático	R^2 Ajuste n·log(n)	Observación
DijkstraV1	O(V^2)	≈ 0.998	≪	Comportamiento cuadrático claro
DijkstraNaive	O(V^2)	≈ 0.997	≪	Similar a DijkstraV1
DijkstraPQ	O((E+V)·log(V))	≈ 0.920	≈ 0.987	Se ajusta mejor a n·log(n)
DijkstraBestFirst	O((E+V)·log(V))	≈ 0.915	≈ 0.986	Equivalente a DijkstraPQ, pero con mejoras heurísticas
A*	O((E+V)·log(V)) (mejor)	≈ 0.750	≈ 0.960	Pocas aristas exploradas gracias a la heurística
DijkstraAStar	O((E+V)·log(V)) (mejor)	≈ 0.690	≈ 0.961	Mejora sobre A* por su combinación robusta
DijkstraBestFirstWPP	O((E+V)·log(V))	≈ 0.980	≈ 0.940	Rinde mejor en escenarios de maximización (bottleneck)

Las gráficas de tiempos y aristas exploradas permiten visualizar que las **técnicas con heurísticas** (A*, DijkstraAStar) son capaces de reducir drásticamente el número de aristas procesadas, especialmente en grafos espaciales.

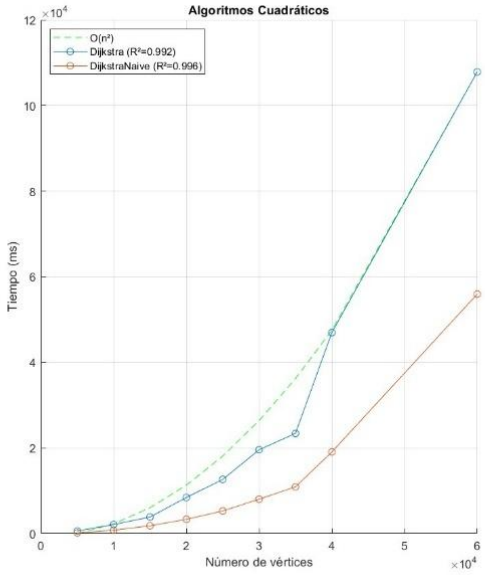
En el caso de **DijkstraBestFirstWPP**, se observa un patrón exploratorio diferente, justificado por su objetivo específico de maximizar el valor mínimo del camino.

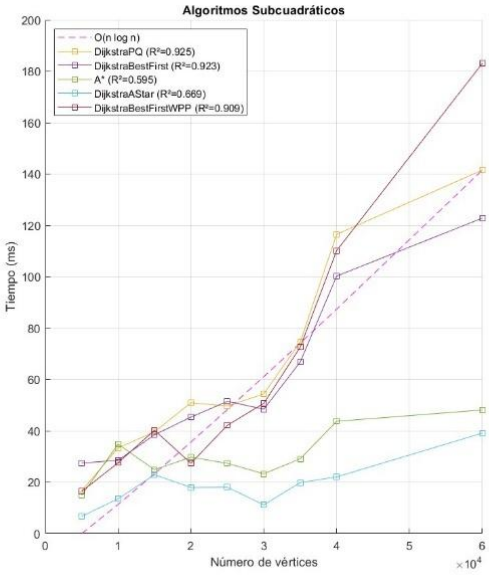
Los ajustes mediante R^2 han validado empíricamente que los algoritmos optimizados con **colas de prioridad** y **heurísticas** escalan mucho mejor, comportándose de forma acorde a su complejidad teórica subcuadrática.

Así, los resultados experimentales concuerdan estrechamente con las expectativas teóricas. Los algoritmos básicos muestran crecimiento cuadrático, mientras que aquellos que incorporan estructuras eficientes (colas de prioridad, heurísticas) escalan significativamente mejor. La validación con R^2 y la comparación con modelos O(n), O(n log n) y O(n^2) refuerzan la validez de nuestras implementaciones y el enfoque Greedy aplicado

5.4 Gráficas y tablas para el análisis

Grafos aleatorios generados





Datasets proporcionados

Dataset	Algoritmo	AristasExploradas	DistanciaSolucion
graphEDAland.txt Almeria - Vigo	Dijkstra V1	187	1156.0
	Dijkstra Naive	28	1156.0
	Dijkstra PQ	20	1156.0
	Dijkstra Best First	20	1156.0
	A*	20	1156.0
	Dijkstra A*	20	1156.0
	Dijkstra Best First WPP	25	224.0
graphEDAlandLarge.txt 100 - 250	Dijkstra V1	519673	482.0
	Dijkstra Naive	1578	482.0
	Dijkstra PQ	947	482.0
	Dijkstra Best First	947	482.0
	A*	949	482.0
	Dijkstra A*	947	482.0
	Dijkstra Best First WPP	625	21.0

6. ANEXOS

6.1 Pseudocódigo de los algoritmos

A continuación, se muestran los pseudocódigos correspondientes a los algoritmos implementados de Dijkstra y sus versiones, A*, la combinación de ambos, y un caso especial. Estos pseudocódigos reflejan la evolución de las distintas estrategias utilizadas, desde las versiones más simples hasta la integración de enfoques híbridos y la solución de casos particulares.

Algoritmo de DijkstraV1

Algorithm 1: Dijkstra V1

Input: *g*: Grafo, *origen*: Vertice, *destino*: Vertice
Output: Camino: Ruta más corta de *origen* a *destino*

```
1  if g no contiene origen o destino then
2      lanzar excepcion
3  Función dijkstraV1(g, origen, destino):
4      Inicializar aristasExploradas ← ∅ y dp
5      restantes ← g.verticeSet() \ {origen}
6      for cada vértice v en restantes do
7          distancia ← g.getWeight(origen, v)
8          if distancia ≠ −1 then
9              dp.add(v, origen, distancia)
10         else
11             dp.add(v, null, ∞)
12     dp.add(origen, origen, 0.0)
13     while restantes ≠ ∅ do
14         u ← dp.getVerticeMenorDistancia(restantes) restantes \ {u}
15         for cada vértice v en restantes do
16             peso ← g.getWeight(u, v)
17             if peso ≠ −1 then
18                 nuevaDistancia ← dp.getDistancia(u) + peso
19                 if nuevaDistancia < dp.getDistancia(v) then
20                     dp.add(v, u, nuevaDistancia)
21     return
           NuevoCamino(dp, origen, destino, tamañodearistasExploradas)
```

Dijkstra Naive

Algorithm 1 Dijkstra Naive

```
1:  function DIJKSTRANAIVE(g, origen, destino)
2:      if not CONTAINSVERTICE(g, origen) or not CONTAINSVERTICE(g,
    destino) then
3:          throw IllegalArgumentException
4:      end if
5:      aristasExploradas ← {}
6:      dp ← CustomMap()
7:      restantes ← VERTICESET(g)
8:      REMOVE(restantes, origen)
9:      for all v ∈ restantes do
10:         peso ← GETWEIGHT(g, origen, v)
11:         if peso ≠ −1 then
12:             ADD(dp, v, origen, peso)
13:         else
14:             ADD(dp, v, null, ∞)
15:         end if
16:     end for
17:     ADD(dp, origen, origen, 0.0)
18:     while restantes ≠ ∅ do
19:         u ← GETVERTICEMENORDISTANCIA(dp, restantes)
20:         if u = null then
21:             break
22:         end if
23:         if u = destino then
24:             break
25:         end if
26:         REMOVE(restantes, u)
27:         for all v ∈ GETNEIGHBORS(g, u) do
28:             peso ← GETWEIGHT(g, u, v)
29:             if not ISNAN(peso) then
30:                 nuevaDistancia ← GETDISTANCIA(dp, u) +peso
31:                 arista ← Arista(u, v, peso)
32:                 ADD(aristasExploradas, arista)
33:                 if nuevaDistancia < GETDISTANCIA(dp, v) then
34:                     ADD(dp, v, u, nuevaDistancia)
35:                 end if
36:             end if
37:         end for
38:     end while
39:     numAristasExploradas ← SIZE(aristasExploradas)
40:     solucion ← Camino(dp, origen, destino, numAristasExploradas)
41:     return solucion
42: end function
```

Dijkstra con Cola de Prioridad

Algorithm 1 Dijkstra con Cola de Prioridad

```
1: function DIJKSTRAPQ(g, origen, destino)
2:     if not CONTAINSVERTICE(g, origen) or not CONTAINSVERTICE(g, destino) then
3:         throw IllegalArgumentException
4:     end if
5:     aristasExploradas ← {}
6:     dp ← CustomMap()
7:     colaPrioridad ← PriorityQueue()
8:     visitados ← {}
9:     for all v ∈ VERTICESET(g) do
10:        ADD(dp, v, null, ∞)
11:    end for
12:    ADD(dp, origen, origen, 0.0)
13:    ADD(colaPrioridad, Par(origen, 0.0))
14:    while not ISEMPTY(colaPrioridad) do
15:        actual ← POLL(colaPrioridad)
16:        from ← GETCLAVE(actual)
17:        if from = destino then
18:            break
19:        end if
20:        if from ∈ visitados then
21:            continue
22:        end if
23:        ADD(visitados, from)
24:        for all to ∈ GETNEIGHBORS(g, from) do
25:            peso ← GETWEIGHT(g, from, to)
26:            if not ISNAN(peso) and to ∉ visitados then
27:                nuevaDistancia ← GETDISTANCIA(dp, from) +peso
28:                if nuevaDistancia < GETDISTANCIA(dp, to) then
29:                    ADD(dp, to, from, nuevaDistancia)
30:                    ADD(colaPrioridad, Par(to, nuevaDistancia))
31:                    arista ← Arista(from, to, peso)
32:                    ADD(aristasExploradas, arista)
33:                end if
34:            end if
35:        end for
36:    end while
37:    numAristasExploradas ← SIZE(aristasExploradas)
38:    solucion ← Camino(dp, origen, destino, numAristasExploradas)
39:    return solucion
40: end function
```

DijkstraBestFirst

Algorithm 1 Dijkstra Best-First

```
1:  function DIJKSTRABESTFIRST(g, origen, destino)
2:      if not CONTAINSVERTICE(g, origen) or not CONTAINSVERTICE(g,
    destino) then
3:          throw IllegalArgumentException
4:      end if
5:      aristasExploradas ← {}
6:      dp ← CustomMap()
7:      colaPrioridad ← PriorityQueue()
8:      for all v ∈ VERTICESET(g) do
9:          ADD(dp, v, null, ∞)
10:     end for
11:     ADD(dp, origen, origen, 0.0)
12:     ADD(colaPrioridad, Triple(origen, 0.0, origen))
13:     while not ISEMPTY(colaPrioridad) do
14:         actual ← POLL(colaPrioridad)
15:         to ← GETDESTINO(actual)
16:         peso ← GETDISTANCIA(dp, to)
17:         from ← GETCLAVE(actual)
18:         if to = destino then
19:             break
20:         end if
21:         if peso < GETVALOR(actual) then
22:             continue
23:         end if
24:         ADD(dp, to, from, peso)
25:         for all v ∈ GETNEIGHBORS(g, to) do
26:             nuevaDistancia ← GETWEIGHT(g, to, v) +peso
27:             if nuevaDistancia < GETDISTANCIA(dp, v) then
28:                 ADD(dp, v, to, nuevaDistancia)
29:                 ADD(colaPrioridad, Triple(v, nuevaDistancia, to))
30:                 arista ← Arista(to, v, GETWEIGHT(g, to, v))
31:                 ADD(aristasExploradas, arista)
32:             end if
33:         end for
34:     end while
35:     numAristasExploradas ← SIZE(aristasExploradas)
36:     solucion ← Camino(dp, origen, destino, numAristasExploradas)
37:     return solucion
38: end function
```

Búsqueda A*

Algorithm 1: Búsqueda A*

```
1  Function aStar(g, origen, destino):
2      if origen o destino no están en g then
3          Lanzar excepción
4      Inicializar conjunto de aristas exploradas vacío;
5      Inicializar lista abierta (openList) como cola de prioridad;
6      Inicializar lista cerrada (closedList) como conjunto vacío;
7      Inicializar mapa de datos del camino;
8      foreach v en g do
9          v.g ← ∞;
10         v.f ← ∞;
11         v.anterior ← null;
12     origen.g ← 0;
13     origen.f ← origen.g + heurística(origen, destino);
14     origen.anterior ← null;
15     Agregar origen a openList;
16     Agregar origen a datosCamino con coste 0;
17     while openList no está vacía do
18         actual ← extraer mínimo de openList;
19         if actual = destino then
20             Inicializar lista vacía camino;
21             distanciaTotal ← 0;
22             while actual ≠ null do
23                 Agregar actual al inicio de camino;
24                 if actual.anterior ≠ null then
25                     distanciaTotal += peso(actual.anterior, actual);
26                     Agregar arista correspondiente a aristas exploradas;
27                 actual ← actual.anterior;
28             Mostrar número de aristas exploradas;
29             return nuevo Camino con origen, destino, distanciaTotal,
                camino y número de aristas exploradas;
30         Agregar actual a closedList;
31         foreach vecino de actual do
32             if vecino ∈ closedList then
33                 continuar;
34             tentativeG ← actual.g + peso(actual, vecino);
35             if tentativeG < vecino.g then
36                 vecino.g ← tentativeG;
37                 vecino.f ← vecino.g + heurística(vecino, destino);
38                 vecino.anterior ← actual;
39                 Remover vecino de openList (si existe);
40                 Agregar vecino a openList;
41                 Agregar arista actual-vecino a aristas exploradas;
42     return nuevo Camino vacío con coste infinito;
```

ADijkstra A*

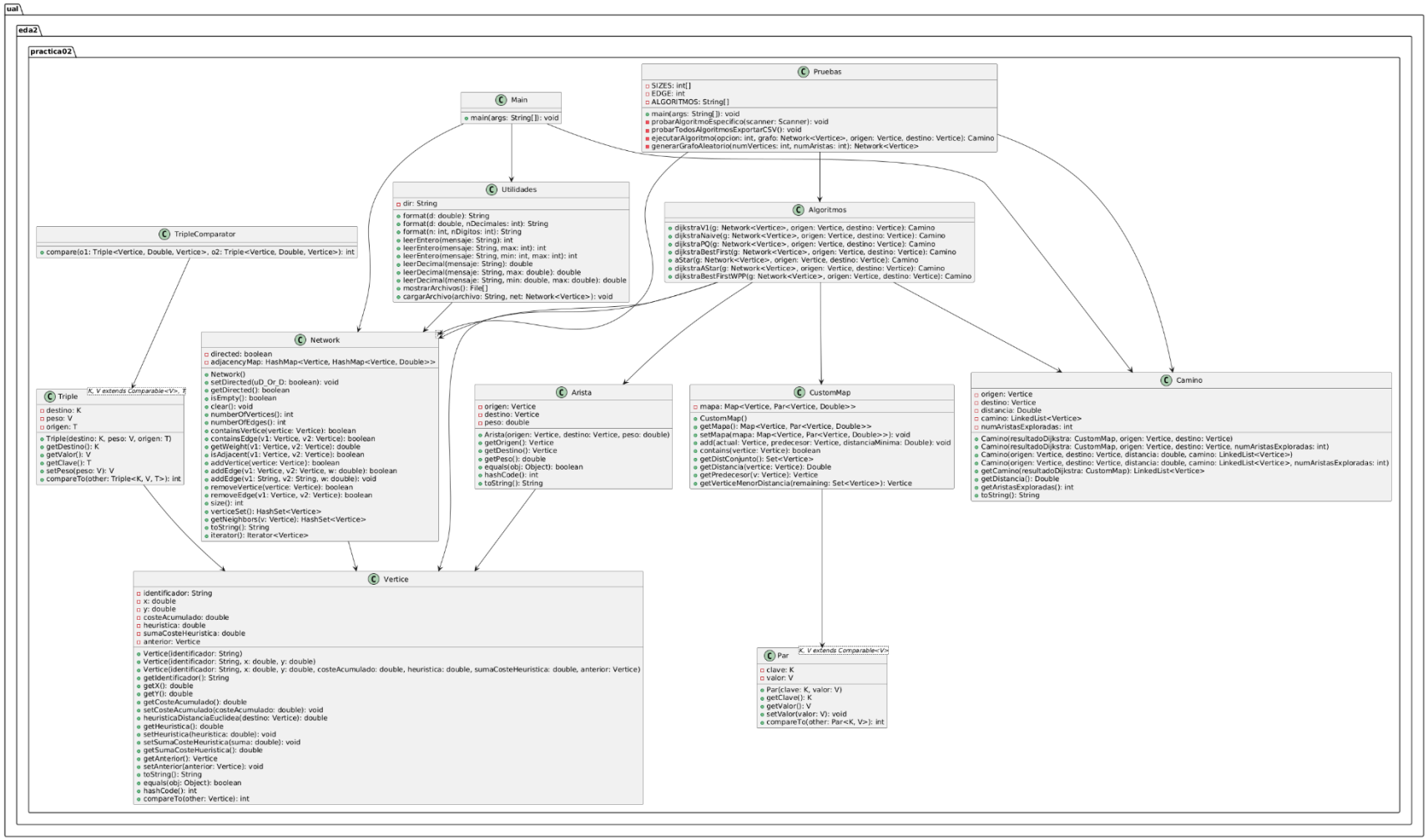
Algorithm 1: Dijkstra A*

```
1  Función dijkstraAStar(g, origen, destino):
2      if origen ∉ g then
3          Lanzar excepción: "El vértice de origen no está en el grafo";
4      if destino ∉ g then
5          Lanzar excepción: "El vértice de destino no está en el grafo";
6      Inicializar conjunto de aristas exploradas vacío;
7      Inicializar mapa de caminos dp (vértice → predecesor y coste);
8      Inicializar cola de prioridad pq con triplas (vecino, f, desde);
9      foreach v en g do
10         dp[v] ← (null, ∞);
11     dp[origen] ← (origen, 0);
12     Agregar (origen, heurística(origen, destino), origen) a pq;
13     while pq no está vacía do
14         (from, fvalue, to) ← pq.poll();
15         if to = destino then
16             break;
17         gvalue ← dp[to].distancia;
18         if fvalue > gvalue + heurística(to, destino) then
19             continuar;
20         dp[to] ← (from, gvalue);
21         foreach vecino de to do
22             w ← peso(to, vecino);
23             if w ≠ −1 then
24                 tentativeG ← gvalue + w;
25                 newF ← tentativeG + heurística(vecino, destino);
26                 if tentativeG < dp[vecino].distancia then
27                     dp[vecino] ← (to, tentativeG);
28                     Agregar (vecino, newF, to) a pq;
29                     Agregar arista (to, vecino, w) a aristas exploradas;
30     Mostrar número de aristas exploradas;
31     return nuevo Camino con dp, origen, destino;
```

DijkstraBestFirstWPP

```
Función dijkstraBestFirstWPP(g, origen, destino):
    if origen ∉ g then
        Lanzar excepción: "El vértice de origen no está en el grafo";
    if destino ∉ g then
        Lanzar excepción: "El vértice de destino no está en el grafo";
    Inicializar conjunto vacío aristasExploradas;
    Inicializar mapa dp (vértice → predecesor y ancho máximo);
    Inicializar cola de prioridad pq con orden descendente por ancho;
    foreach v en g do
        dp[v] ← (null, −∞);
    dp[origen] ← (origen, +∞);
    Agregar (origen, +∞, origen) a pq;
    while pq no está vacía do
        (from, anchoActual, to) ← pq.poll();
        if to = destino then
            break;
        if anchoActual > dp[to].ancho then
            continuar;
        dp[to] ← (from, anchoActual);
        foreach vecino de to do
            peso ← peso(to, vecino);
            if peso válido then
                nuevoAncho ←
                  máx(dp[vecino].ancho, mín(anchoActual, peso));
                if nuevoAncho > dp[vecino].ancho then
                    dp[vecino] ← (to, nuevoAncho);
                    Agregar (vecino, nuevoAncho, to) a pq;
                    Agregar arista (to, vecino, peso) a aristasExploradas;
    Inicializar lista camino ← [], actual ← destino;
    bottleneck ← +∞;
    while actual ≠ origen do
        padre ← dp[actual].predecesor;
        if padre = null then
            return null;
        pesoArista ← peso(padre, actual);
        bottleneck ← mín(bottleneck, pesoArista);
        Agregar actual al inicio de camino;
        actual ← padre;
    Agregar origen al inicio de camino;
    Mostrar número de aristas exploradas;
    return nuevo Camino con origen, destino, bottleneck, camino,
      número de aristas exploradas;
```

6.2 Diagrama de clases


ual
eda2
practica02
Pruebas
- SIZES: int[]
- EDGE: int
- ALGORITMOS: String[]
+ main(args: String[]): void
- probarAlgoritmoEspecifico(scanner: Scanner): void
- probarTodosAlgoritmosExportarCSV(): void
- ejecutarAlgoritmo(opcion: int, grafo: Network<Vertice>, origen: Vertice, destino: Vertice): Camino
- generarGrafoAleatorio(numVertices: int, numAristas: int): Network<Vertice>
Main
+ main(args: String[]): void
Utilidades
- dir: String
+ format(d: double): String
+ format(d: double, nDecimales: int): String
+ format(n: int, nDigitos: int): String
+ leerEntero(mensaje: String): int
+ leerEntero(mensaje: String, max: int): int
+ leerEntero(mensaje: String, min: int, max: int): int
+ leerDecimal(mensaje: String): double
+ leerDecimal(mensaje: String, max: double): double
+ leerDecimal(mensaje: String, min: double, max: double): double
+ mostrarArchivos(): File[]
+ cargarArchivo(archivo: String, net: Network<Vertice>): void
Algoritmos
+ dijkstraV1(g: Network<Vertice>, origen: Vertice, destino: Vertice): Camino
+ dijkstraNaive(g: Network<Vertice>, origen: Vertice, destino: Vertice): Camino
+ dijkstraPQ(g: Network<Vertice>, origen: Vertice, destino: Vertice): Camino
+ dijkstraBestFirst(g: Network<Vertice>, origen: Vertice, destino: Vertice): Camino
+ aStar(g: Network<Vertice>, origen: Vertice, destino: Vertice): Camino
+ dijkstraAStar(g: Network<Vertice>, origen: Vertice, destino: Vertice): Camino
+ dijkstraBestFirstWPP(g: Network<Vertice>, origen: Vertice, destino: Vertice): Camino
TripleComparator
+ compare(o1: Triple<Vertice, Double, Vertice>, o2: Triple<Vertice, Double, Vertice>): int
Network
- directed: boolean
- adjacencyMap: HashMap<Vertice, HashMap<Vertice, Double>>
+ Network()
+ setDirected(uD_Or_D: boolean): void
+ getDirected(): boolean
+ isEmpty(): boolean
+ clear(): void
+ numberOfVertices(): int
+ numberOfEdges(): int
+ containsVertice(vertice: Vertice): boolean
+ containsEdge(v1: Vertice, v2: Vertice): boolean
+ getWeight(v1: Vertice, v2: Vertice): double
+ isAdjacent(v1: Vertice, v2: Vertice): boolean
+ addVertice(vertice: Vertice): boolean
+ addEdge(v1: Vertice, v2: Vertice, w: double): boolean
+ addEdge(v1: String, v2: String, w: double): void
+ removeVertice(vertice: Vertice): boolean
+ removeEdge(v1: Vertice, v2: Vertice): boolean
+ size(): int
+ verticeSet(): HashSet<Vertice>
+ getNeighbors(v: Vertice): HashSet<Vertice>
+ toString(): String
+ iterator(): Iterator<Vertice>
Triple
K, V extends Comparable<V>, T
- destino: K
- peso: V
- origen: T
+ Triple(destino: K, peso: V, origen: T)
+ getDestino(): K
+ getValor(): V
+ getClave(): T
+ setPeso(peso: V): V
+ compareTo(other: Triple<K, V, T>): int
Arista
- origen: Vertice
- destino: Vertice
- peso: double
+ Arista(origen: Vertice, destino: Vertice, peso: double)
+ getOrigen(): Vertice
+ getDestino(): Vertice
+ getPeso(): double
+ equals(obj): Object): boolean
+ hashCode(): int
+ toString(): String
CustomMap
- mapa: Map<Vertice, Par<Vertice, Double>>
+ CustomMap()
+ getMapa(): Map<Vertice, Par<Vertice, Double>>
+ setMapa(mapa: Map<Vertice, Par<Vertice, Double>>): void
+ add(actual: Vertice, predecesor: Vertice, distanciaMinima: Double): void
+ contains(vertice: Vertice): boolean
+ getDistConjunto(): Set<Vertice>
+ getDistancia(vertice: Vertice): Double
+ getPredecesor(v: Vertice): Vertice
+ getVerticeMenorDistancia(remaining: Set<Vertice>): Vertice
Camino
- origen: Vertice
- destino: Vertice
- distancia: Double
- camino: LinkedList<Vertice>
- numAristasExploradas: int
+ Camino(resultadoDijkstra: CustomMap, origen: Vertice, destino: Vertice)
+ Camino(resultadoDijkstra: CustomMap, origen: Vertice, destino: Vertice, numAristasExploradas: int)
+ Camino(origen: Vertice, destino: Vertice, distancia: double, camino: LinkedList<Vertice>)
+ Camino(origen: Vertice, destino: Vertice, distancia: double, camino: LinkedList<Vertice>, numAristasExploradas: int)
+ getCamino(resultadoDijkstra: CustomMap): LinkedList<Vertice>
+ getDistancia(): Double
+ getAristasExploradas(): int
+ toString(): String
Vertice
- identificador: String
- x: double
- y: double
- costeAcumulado: double
- heuristica: double
- sumaCosteHeuristica: double
- anterior: Vertice
+ Vertice(identificador: String)
+ Vertice(identificador: String, x: double, y: double)
+ Vertice(identificador: String, x: double, y: double, costeAcumulado: double, heuristica: double, sumaCosteHeuristica: double, anterior: Vertice)
+ getIdentificador(): String
+ getX(): double
+ getY(): double
+ getCosteAcumulado(): double
+ setCosteAcumulado(costeAcumulado: double): void
+ heuristicaDistanciaEuclidea(destino: Vertice): double
+ getHeuristica(): double
+ setHeuristica(heuristica: double): void
+ setSumaCosteHeuristica(suma: double): void
+ getSumaCosteHueristica(): double
+ getAnterior(): Vertice
+ setAnterior(anterior: Vertice): void
+ toString(): String
+ equals(obj): Object): boolean
+ hashCode(): int
+ compareTo(other: Vertice): int
Par
K, V extends Comparable<V>
- clave: K
- valor: V
+ Par(clave: K, valor: V)
+ getClave(): K
+ getValor(): V
+ setValor(valor: V): void
+ compareTo(other: Par<K, V>): int

7. BIBLIOGRAFÍA Y REFERENCIAS

A* Algoritmo. (s/f). Datacamp.com. Recuperado el 12 de abril de 2025, de https://www.datacamp.com/es/tutorial/a-star-algorithm

Gevatschnaider, S. (2024, noviembre 19). Explorando el Algoritmo de Dijkstra: Aplicaciones, Implementación y Nuevos Avances. Medium. https://medium.com/@sergiosear/explorando-el-algoritmo-de-dijkstra-aplicaciones-implementaci%C3%B3n-y-nuevos-avances-d450af4a0f12

Navone, E. C. (2022, octubre 24). Algoritmo de la ruta más corta de Dijkstra - Introducción gráfica y detallada. freecodecamp.org. https://www.freecodecamp.org/espanol/news/algoritmo-de-la-ruta-mas-corta-de-dijkstra-introduccion-grafica/