Estructura de Datos y Algoritmos II

Curso 2024-2025

Práctica 3 – Programación Dinámica

Alumnos:

Alejandro Doncel Delgado

Jorge Godoy Beltrán

Índice de contenidos

1. OBJETIVO

1.1 Justificación de la aplicación de Programación Dinámica

La Programación Dinámica (PD) es la estrategia elegida para resolver el problema de diseño de rutas en EDAland debido a su capacidad para combinar eficiencia y precisión. A diferencia de métodos como la fuerza bruta, que evalúan todas las rutas posibles, o los algoritmos voraces, que toman decisiones rápidas pero no siempre óptimas, la PD descompone el problema en etapas más simples.

Si se quiere la ruta más corta que parte de Almería, visita todas las ciudades y regresa. En lugar de recalcular caminos una y otra vez, la PD "recuerda" soluciones parciales. Por ejemplo, si ya calculamos la distancia mínima para visitar un grupo de ciudades desde Almería, reutilizamos ese resultado al planificar rutas más largas. Esto evita redundancias y reduce drásticamente el tiempo de cálculo.

Este enfoque es ideal para **EDAWasteCollection** porque:

Garantiza la mejor solución: A diferencia de métodos aproximados, la PD explora todas las opciones relevantes de manera estructurada.

Maneja redes en crecimiento: Aunque el costo computacional aumenta con más ciudades, técnicas como el almacenamiento en tablas permiten escalar hasta 25-28 nodos, algo imposible para la fuerza bruta.

Prioriza seguridad y economía: Un error en la ruta podría aumentar costos de combustible o riesgos ambientales. La PD minimiza estos riesgos al asegurar la optimalidad.

1.2 Descripción general del problema

El objetivo es diseñar una ruta semanal para camiones de residuos en EDAland que cumpla tres reglas básicas:

Salir y volver a Almería: Todos los recorridos empiezan y terminan en esta ciudad, donde está la planta de reciclaje.

Visitar cada ciudad una sola vez: Ninguna parada puede repetirse u omitirse.

Recorrer la menor distancia posible: Se deben aprovechar al máximo las carreteras disponibles para ahorrar tiempo y combustible.

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 de estructura base del proyecto	[AD]	16 de abril 2025
Implementación de algoritmos TSP Brute Force	[JG]	19 de abril 2025
Implementación de algoritmo TSP Recursivo	[AD], [JG]	22 de abril 2025
Desarrollo e integración de métodos auxiliares	[JG]	23 de abril 2025
Implementación del algoritmo TSP Iterativo y Greedys	[AD]	27 de abril 2025
Creación del generador de grafos aleatorios y automatización de pruebas	[AD]	27 de abril 2025
Ejecución de pruebas y recogida de métricas	[AD], [JG]	29 de abril 2025
Análisis teórico de complejidades y validación experimental	[JG]	29 de abril 2025
Diseño y generación de tablas de comportamiento	[AD]	29 de mayo 2025
Redacción de la memoria técnica y anexos	[AD], [JG]	1 de mayo 2025
Revisión final, formateo y entrega	[AD], [JG]	4 de mayo 2025

3. ANTECEDENTES

El presente trabajo aplica la Programación Dinámica para resolver el problema de diseño de rutas óptimas en redes complejas, como las requeridas por el sistema EDAWasteCollection. A diferencia de enfoques basados en decisiones, la PD prioriza la optimalidad global mediante la descomposición del problema en etapas interconectadas, donde cada paso aprovecha soluciones previas almacenadas en tablas.

En sistemas de transporte críticos, como la recogida de residuos tóxicos, errores en la planificación de rutas pueden derivar en costes elevados o riesgos ambientales. Mientras métodos voraces ofrecen rapidez sacrificando precisión, y la fuerza bruta garantiza optimalidad a costa de recursos inalcanzables, la PD surge como un equilibrio pragmático. Por ejemplo, en redes como la de **EDAland**, con topografía variable y restricciones operativas estrictas, la PD permite modelar no solo distancias, sino también factores como pendientes o capacidad de carga, integrando estas variables en cada etapa del cálculo.

Este análisis comparativo surge de la necesidad de equilibrar dos aspectos críticos en sistemas de transporte:

1. **Precisión**: Garantizar la ruta más corta posible, evitando riesgos económicos o logísticos.
2. **Eficiencia**: Reducir tiempos de cálculo, incluso en redes en expansión (hasta 28 ciudades).

A lo largo de la práctica, se exploraron alternativas como:

- **Fuerza bruta**: Inviable debido a su complejidad factorial
- **Algoritmos voraces**: Rápidos pero subóptimos, al depender de elecciones locales irreversibles
- **Programación Dinámica**: Combina precisión y eficiencia mediante la reutilización sistemática de resultados parciales.

3.1 Alternativas contempladas y solución final adoptada

3.1.1 Enfoque del problema del viajero con caminos simples

SimplePaths

Este algoritmo explora todos los caminos posibles entre un vértice de origen y uno de destino en un grafo, garantizando que cada vértice se visite solo una vez. Su enfoque recursivo permite identificar rutas completas que conectan los puntos de interés sin repeticiones, lo que es útil para problemas que requieren exhaustividad, como encontrar circuitos Hamiltonianos (rutas que pasan por todos los vértices).

Limitaciones:

Complejidad exponencial: En grafos con *n* vértices, el número de caminos posibles crece factorialmente (O(n!)), lo que lo hace inviable para redes grandes (n>15).

Uso intensivo de memoria: Almacena temporalmente todas las rutas exploradas, consumiendo recursos significativos en grafos densos.

TspSimplePaths

Este algoritmo resuelve el problema del viajante de comercio utilizando simplePaths para encontrar la ruta más corta que visita todos los vértices del grafo exactamente una vez y regresa al origen. Es una solución exacta, ideal para redes pequeñas donde la optimalidad es crítica, como en la planificación inicial de rutas para **EDAWasteCollection**.

Limitaciones:

Hereda la complejidad de `simplePaths`: Su rendimiento se degrada rápidamente al aumentar el número de ciudades, volviéndose impracticable para n>12.

Dependencia de la estructura del grafo: En redes con conexiones desiguales o asimétricas, puede generar soluciones subóptimas si no se exploran todas las permutaciones.

3.1.2 Enfoque del problema del viajero con Fuerza Bruta

TspBruteForce

Este algoritmo resuelve el problema del viajante de comercio (TSP) generando todas las permutaciones posibles de las ciudades, evaluando la distancia total de cada una y seleccionando la ruta más corta. Es una implementación clásica de **fuerza bruta**, diseñada para garantizar la solución óptima en redes pequeñas.

Limitaciones

Complejidad factorial: Para *n* ciudades, evalúa (n−1)! permutaciones. Con n=15, hay 87,178, 291, rutas posibles.

Inviabilidad práctica: Su tiempo de ejecución crece exponencialmente, siendo útil solo para n≤12.

Alto consumo de memoria: Almacena todas las permutaciones temporalmente, lo que limita su uso en dispositivos con recursos restringidos.

3.1.3 Enfoque iterativo y recursivo del problema del viajero

TspIterative

Este algoritmo resuelve el problema del viajante de comercio utilizando Programación Dinámica**.** A diferencia de la fuerza bruta, que evalúa todas las permutaciones, este método optimiza el cálculo reutilizando resultados de subproblemas intermedios. Es

especialmente útil para redes con hasta 20-25 ciudades, donde la fuerza bruta es inviable pero aún se requiere una solución exacta.

Funcionamiento clave

Subconjuntos como máscaras binarias: Representa grupos de ciudades visitadas usando bits (ejemplo: 101 = ciudades 0 y 2 visitadas).

Memorización: Almacena distancias mínimas entre subconjuntos de ciudades en una tabla, reutilizando resultados previos.

Reconstrucción de la ruta: Tras calcular todas las combinaciones, ensambla la ruta óptima desde la tabla.

TspRecursive

Este algoritmo resuelve el problema del viajante de comercio utilizando un enfoque de Programación Dinámica, basado en el algoritmo clásico de **Bellman-Held-Karp**. A diferencia de su versión iterativa, esta implementación prioriza la claridad conceptual al dividir el problema en subetapas naturales mediante llamadas recursivas, lo que facilita entender cómo se combinan las soluciones parciales para alcanzar la óptima global.

Es ideal para redes de tamaño moderado, donde se requiere garantizar la solución exacta sin comprometer la legibilidad del diseño. Por ejemplo, en la red inicial de **EDAland** (21 ciudades), este método permite validar rutas críticas asegurando la distancia mínima, algo imposible con heurísticas voraces.

Funcionamiento clave

Máscaras binarias: Representa subconjuntos de ciudades visitadas como números binarios (ejemplo: 1101 = ciudades 0, 2 y 3 visitadas).

Memorización: Almacena en una tabla el costo mínimo para llegar a una ciudad desde un subconjunto específico, evitando recálculos.

Reconstrucción de la ruta: Usa una tabla auxiliar para registrar el siguiente nodo óptimo, permitiendo reconstruir la ruta completa al final.

3.1.4 Enfoques Greedy del problema del viajero

TspGreedyVertex

Este algoritmo resuelve el TSP mediante una heurística voraz **Greedy**, seleccionando en cada paso la ciudad más cercana a la actual. Es rápido y sencillo, ideal para redes grandes donde métodos exactos (como Programación Dinámica) son inviables. Sin embargo, no garantiza la solución óptima, ya que decisiones locales pueden llevar a rutas globalmente ineficientes. En cada iteración, elige la ciudad no visitada más próxima a la actual.

Limitaciones

Puede quedar atrapado en "mínimos locales". Ejemplo: Si desde Almería se elige Granada, pero esto obliga a un tramo posterior mientras otra ruta inicial más larga evitaría ese costo.

Dependencia del punto de inicio: Rutas pueden variar significativamente según la ciudad inicial.

Falta de garantías: A diferencia de PD, no asegura la mejor solución.

TspGreedyEdges

Este algoritmo resuelve el TSP mediante una heurística voraz basada en aristas, priorizando las conexiones más cortas del grafo para construir progresivamente un **circuito Hamiltoniano**. A diferencia de métodos que seleccionan vértices aquí se ordenan todas las aristas por distancia y se van añadiendo al recorrido, evitando ciclos parciales hasta completar la ruta. Es útil en redes con conexiones heterogéneas, donde ciertos tramos cortos pueden reducir significativamente la distancia total, aunque no garantiza optimalidad.

Limitaciones

Requiere grafos completos: Si hay ciudades desconectadas o distancias infinitas, el algoritmo falla.

Sensibilidad a pesos desiguales: En redes con muchas aristas de similar longitud, puede generar rutas muy distintas ante pequeños cambios en los datos.

No reconsidera decisiones pasadas, ignorando que una arista ligeramente más larga inicialmente podría evitar costos mayores después.

4. ESTUDIO TEÓRICO Y DE IMPLEMENTACIÓN

4.1 Introducción

Tras presentar el contexto y desarrollo del sistema de rutas óptimas mediante **Programación Dinámica (PD)**, en esta sección se analizarán en detalle las soluciones implementadas. El objetivo es estudiar su funcionamiento, eficiencia teórica y práctica, y compararlas con enfoques alternativos, como fuerza bruta o heurísticas Greedy. 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.

Mientras en la práctica anterior se priorizaba la toma de decisiones locales irreversibles, aquí el énfasis está en la combinación óptima de subproblemas interconectados. Este análisis no solo valida la eficacia de la PD en problemas combinatorios complejos, sino que también sienta las bases para entender sus límites en escenarios de escalabilidad extrema

4.2 SimplePaths

El algoritmo simplePaths está diseñado para encontrar todos los caminos simples posibles entre un vértice origen y sus vecinos directos en un grafo, generando ciclos que parten y regresan al origen. Utiliza una estrategia de backtracking recursivo para explorar exhaustivamente cada ruta sin repetir vértices, lo que lo convierte en una herramienta útil para problemas que requieren exhaustividad, como identificar circuitos Hamiltonianos en redes pequeñas.

Estudio de implementación - Funcionamiento del algoritmo

El algoritmo tiene como finalidad encontrar todos los caminos simples que parten de un vértice de origen hacia sus vecinos inmediatos en el grafo. Para ello, se emplea una estrategia recursiva que evita ciclos mediante una estructura de tipo **`HashSet`** para los vértices visitados.

El método principal, simplePaths, recorre cada vecino directo del vértice de origen utilizando el método **`getVecinos`**, e invoca para cada uno el procedimiento auxiliar simplePathsAux, que se encarga de construir los caminos de manera controlada. La clase `Solucion` se utiliza para almacenar temporalmente el camino actual, añadiendo y retirando vértices conforme avanza y retrocede la recursión.

Una característica destacable de la implementación es la forma en que se asegura la validez de cada camino: el conjunto visitados garantiza que no se repitan vértices, manteniendo así la simplicidad del recorrido. Al alcanzar el destino, se calcula el coste total del camino mediante una suma acumulativa de pesos obtenidos con **`getPeso`**, lo cual permite asociar a cada solución su correspondiente medida cuantitativa.

Cada camino válido encontrado se encapsula como una instancia de `Solucion` y se almacena en una lista, lo que permite disponer de todas las rutas posibles desde el vértice de partida con información relevante para su análisis posterior.

Estudio Teórico - Análisis de la complejidad del algoritmo

Sea n el número de vértices del grafo.

Complejidad temporal:

El algoritmo explora todos los caminos simples que se pueden formar desde un vértice de origen hacia sus vecinos, sin repetir vértices en el mismo camino. En el caso más general, esto equivale a generar permutaciones de los vértices, excepto el de inicio, respetando las restricciones impuestas por las conexiones del grafo.

En el peor de los casos, por ejemplo, si el grafo es completo, el número de caminos simples posibles desde un vértice **es del orden de (n−1)**!, ya que el algoritmo puede formar caminos recorriendo todos los vértices sin repetir ninguno. Para cada uno de estos caminos, se realiza un recorrido del grafo y una suma acumulativa de los pesos.

Por tanto, la complejidad temporal total es:

$$T(n) = O(n \cdot (n-1)!) = O(n!)$$

Complejidad espacial:

Respecto al espacio utilizado, se identifican tres estructuras principales:

- **visitados**: un conjunto que almacena los vértices del camino actual. Ocupa espacio O(n).
- **resultado**: una lista temporal con los vértices del camino parcial, también de tamaño O(n).
- **soluciones**: la lista que almacena todos los caminos válidos encontrados. En el peor caso, grafo completo, esta lista puede llegar a contener hasta (n−1)! caminos, cada uno con hasta n vértices.

Por tanto, la complejidad espacial es:

$$S(n) = O(n \cdot (n-1)!) = O(n!)$$

Análisis de eficiencia

El algoritmo simplePaths ofrece la ventaja de generar de forma exhaustiva todos los caminos simples posibles entre un vértice de origen y sus vecinos, lo cual es útil en contextos donde se requiere un análisis completo de rutas o soluciones óptimas basadas en múltiples criterios. Además, su implementación es clara y aprovecha estructuras básicas como conjuntos y listas, facilitando su comprensión y mantenimiento.

Sin embargo, su principal desventaja es la baja escalabilidad: al tener una complejidad factorial en tiempo y espacio, se vuelve ineficiente para grafos de tamaño moderado o grande. Esto limita su aplicabilidad práctica a grafos pequeños o muy específicos, donde el coste computacional sea asumible.

4.3 TspSimplePaths

El algoritmo **tspSimplePaths** aborda la resolución del problema del viajante, en grafos pequeños mediante una estrategia de búsqueda exhaustiva. Su objetivo es encontrar el camino más corto que visita exactamente una vez cada vértice del grafo y regresa al punto de origen. Esta implementación no utiliza técnicas de poda ni estructuras de optimización avanzadas, sino que explora todas las permutaciones posibles de los vértices a través de una búsqueda recursiva.

Ideal para propósitos didácticos o escenarios donde el número de vértices es reducido y se desea obtener la solución óptima exacta.

Estudio de implementación - Funcionamiento del algoritmo

El algoritmo comienza con una llamada a **tspSimplePaths**, que inicializa las estructuras necesarias: una solución parcial `resultado`, una solución óptima `solucion` y un conjunto de vértices visitados. Luego, para cada vecino del vértice de origen, invoca el método auxiliar recursivo `tspSimplePathsAux`.

Dentro del método auxiliar, se construye recursivamente cada camino posible sin repetir vértices, usando `HashSet` para controlar los nodos visitados. Una vez se alcanza el vértice de destino y se ha visitado todo el grafo, comprobado con `resultado.size() == g.size()` , se completa el ciclo añadiendo el vértice de origen al final del camino. A continuación, se calcula la distancia total del recorrido utilizando los pesos entre vértices y se compara con la mejor solución registrada hasta el momento.

Si la nueva ruta es más corta, se actualiza la solución óptima. La implementación garantiza así que, al finalizar, se retorna el camino Hamiltoniano más corto posible que forma un ciclo cerrado.

Una mejora relevante en esta implementación respecto a una enumeración ingenua es la evaluación inmediata del coste acumulado, permitiendo evitar almacenar todas las rutas

exploradas. Las estructuras `Solucion` y `HashSet` juegan un papel central en la gestión eficiente del estado parcial y en la comparación entre soluciones.

Estudio Teórico - Análisis de la complejidad del algoritmo

Sea n el número de vértices del grafo.

Complejidad temporal:

El algoritmo explora todas las posibles permutaciones de los vértices, buscando ciclos Hamiltonianos que comiencen y terminen en el vértice de origen. Aunque solo almacena la mejor solución encontrada, el número de caminos que se generan y evalúan sigue siendo el mismo que en un enfoque exhaustivo.

En concreto, se generan (n−1)! caminos posibles, y para cada uno se calcula el coste total del recorrido, lo que implica una operación lineal adicional en cada evaluación.

Por tanto, la complejidad temporal se expresa como:

$$T(n) = O((n-1)! \cdot n) = O(n!)$$

Complejidad espacial:

A diferencia de otras implementaciones que almacenan todas las soluciones posibles, este algoritmo guarda únicamente el mejor camino encontrado hasta el momento. Aun así, utiliza varias estructuras temporales:

- **visitados**: conjunto que guarda los vértices del camino actual → O(n)
- **resultado**: lista que construye el recorrido temporal, con hasta n+1 vértices → O(n)
- **solucion**: contiene el mejor ciclo hallado → O(n)

Dado que ninguna de estas estructuras crece más allá de un tamaño proporcional a n, la complejidad espacial total es**:**

$$S(n) = O(n)$$

Análisis de Eficiencia

Una de las principales ventajas del algoritmo *tspSimplePaths* es que garantiza la obtención de la solución óptima al problema del viajante, explorando todas las rutas posibles y seleccionando aquella con menor coste. Además, a diferencia de enfoques que almacenan todas las soluciones, este solo conserva la mejor encontrada, lo que reduce significativamente el consumo de memoria.

Sin embargo, su mayor desventaja es su baja escalabilidad: presenta una complejidad temporal factorial (O(n!)), lo que lo vuelve inviable para grafos medianos o grandes. Esta limitación lo restringe principalmente a aplicaciones sobre grafos pequeños o con fines académicos, donde se prioriza la exhaustividad por encima de la eficiencia.

4.4 TspBruteForce

El algoritmo tspBruteForce resuelve el problema del viajante (TSP) explorando de forma exhaustiva todas las posibles rutas que visitan exactamente una vez cada vértice del grafo y retornan al vértice de origen. Utiliza un enfoque iterativo basado en permutaciones lexicográficas y una matriz de adyacencia para calcular eficientemente el coste de cada recorrido. Al evaluar todas las combinaciones posibles, garantiza la obtención de la ruta de menor coste.

Estudio de implementación - Funcionamiento del algoritmo

La implementación comienza construyendo una matriz de adyacencia `double[][] matriz` que contiene los pesos entre pares de vértices. A partir del vértice de origen, se genera un arreglo con los índices del resto de vértices, sobre el cual se aplican permutaciones para representar todas las rutas posibles.

Por cada permutación generada, se calcula el coste total del recorrido con `calcularCosto`, sumando los pesos de cada tramo, incluyendo el regreso al nodo de origen. La ruta con menor coste se almacena como solución óptima.

Este enfoque evita la recursión y se basa completamente en estructuras iterativas, lo que lo hace claro y fácil de seguir. Al finalizar, se traduce la mejor ruta en una lista de vértices y se encapsula en un objeto Solucion.

Métodos auxiliares utilizados

El algoritmo se apoya en varios métodos auxiliares que permiten modularizar las operaciones clave:

`siguientePermutacion(Integer[] secuencia)`: Genera la siguiente permutación lexicográfica del arreglo recibido. Se utiliza para iterar de forma sistemática sobre todas las rutas posibles sin repeticiones.

`obtenerPrimerIndice(Integer[] secuencia)`: Busca el primer índice desde el final donde la secuencia todavía es creciente. Es un paso esencial para identificar si aún hay permutaciones posibles o si se ha alcanzado la última.

`swap(Integer[] secuencia, int i, int j)`: Intercambia dos elementos en la secuencia. Se usa tanto en la generación de la siguiente permutación como en la inversión de sufijos.

`calcularCosto(Integer origen, Integer[] ruta, double[][] matriz)`:Calcula el coste total de una ruta dada, accediendo directamente a la matriz de pesos. Es el núcleo evaluador del algoritmo, pues determina si una ruta mejora la mejor solución actual.

Estudio Teórico - Análisis de la complejidad del algoritmo

Sea n el número de vértices del grafo.

Complejidad temporal

El algoritmo tspBruteForce evalúa de forma exhaustiva todas las posibles rutas que parten de un vértice fijo, el origen, y visitan todos los demás exactamente una vez, regresando finalmente al vértice de partida.

Dado que se mantiene el vértice de origen fijo y se permutan los n−1 vértices restantes, el número total de rutas posibles es:

$$(n-1)!\ caminos\ posibles$$

Para cada una de estas rutas, el método calcularCosto recorre la ruta completa, de longitud n, y suma los costes entre vértices, lo que implica un coste lineal:

$$(Costo\ por\ ruta\colon\ O(n))$$

Por tanto, la complejidad total en tiempo es:

$$T(n)=O((n-1)!\cdot n)=O(n!)$$

El algoritmo tiene una complejidad factorial **O(n!),** lo que lo vuelve ineficiente para grafos grandes**.**

Análisis de eficiencia

El algoritmo tspBruteForce presenta algunas ventajas destacables, especialmente en contextos donde se requiere una solución exacta. Al evaluar todas las rutas posibles, garantiza encontrar siempre la solución óptima al problema del viajante. Además, su implementación es directa y relativamente sencilla, lo que lo convierte en una opción útil para fines educativos o de análisis teórico.

Otra ventaja es que puede aplicarse a grafos sin requerir condiciones especiales, siempre que se adapten adecuadamente los pesos.

Sin embargo, sus desventajas son significativas. La principal es su altísima complejidad temporal, que crece de forma factorial con el número de vértices, lo que lo hace impracticable en grafos medianos o grandes. Esta falta de eficiencia lo convierte en una opción poco escalable para aplicaciones reales. Además, no incorpora ninguna técnica de poda ni heurística que permita descartar rutas no prometedoras, lo que aumenta innecesariamente el número de combinaciones exploradas y agrava su lentitud.

4.5 TspIterative

El algoritmo tspIterative es una implementación del Problema del Viajante de Comercio utilizando programación dinámica con *bitmasking*, también conocido como **algoritmo de**

Held-Karp. Esta técnica permite reducir significativamente la cantidad de rutas a evaluar en comparación con los enfoques puramente fuerza bruta, ya que almacena resultados parciales en una tabla memo que se reutiliza durante la ejecución. El objetivo es encontrar el ciclo hamiltoniano de menor coste que recorre todos los vértices del grafo exactamente una vez, comenzando y terminando en el vértice de origen.

Estudio de implementación - Funcionamiento del algoritmo

El método principal tspIterative comienza inicializando la matriz de distancias del grafo y una tabla `memo` para almacenar subresultados. Cada entrada `memo[i][S]` representa el coste mínimo para llegar al vértice i habiendo visitado el conjunto de vértices representado por el subconjunto S codificado como un entero *bitmask*. A medida que se generan subconjuntos crecientes de vértices mediante la función **`combinations`**, se van actualizando los valores en la tabla `memo`.

Una vez completada la tabla dinámica, se determina el mínimo coste total recorriendo todos los vértices y retornando al origen. Después, se reconstruye el camino óptimo en orden inverso utilizando la información almacenada en la tabla. El camino se convierte finalmente en una instancia de la clase Solucion.

Métodos auxiliares utilizados

`notIn(int elem, int subset)`: Verifica si un vértice no está incluido en el subconjunto codificado como un entero mediante operaciones de bits.

`combínatenos(int r, int n)`: Genera todas las combinaciones posibles de r vértices entre n, representadas como enteros bitmasks. Es fundamental para recorrer todos los subconjuntos de vértices necesarios para la programación dinámica.

`combinations(int set, int at, int r, int n, List<Integer> subsets)`: Método recursivo que construye las combinaciones mencionadas anteriormente. Utiliza operaciones de bit para incluir o excluir vértices y formar subconjuntos de tamaño r.

Estos métodos son esenciales para el funcionamiento del algoritmo, ya que permiten representar de manera compacta los estados del recorrido y explorar eficientemente todas las posibles combinaciones necesarias sin redundancia

Estudio Teórico - Análisis de la complejidad del algoritmo

El algoritmo tspIterative utiliza **programación dinámica con bitmasking**, lo cual mejora la eficiencia respecto a los algoritmos completamente fuerza bruta, aunque sigue siendo exponencial.

Estructura de la Memoria, tabla memo

La tabla memo tiene dimensiones N × 2^N, donde:

- N es el número de vértices del grafo.
- 2^N es el número total de subconjuntos de vértices (es decir, las combinaciones de vértices que podrían estar incluidos en un recorrido).

Para cada conjunto de vértices, se guarda la mejor distancia posible para llegar a un vértice específico, lo que significa que tenemos una entrada en la tabla para cada par.

2. Operaciones por Subconjunto y Vértice

El algoritmo recorre todos los subconjuntos posibles de vértices, exceptuando el vértice de inicio, es decir, hay 2^N subconjuntos. Para cada subconjunto:

- El algoritmo recorre todos los vértices posibles, esto se hace dentro del ciclo para cada end y next, lo que implica un coste adicional de O(N).
- Dentro de cada conjunto, se evalúa la distancia más corta entre los vértices de un conjunto utilizando las entradas de la tabla memo. Esto implica un bucle de búsqueda de O(N) para cada vértice de destino.

3. Cálculo Total de la Complejidad Temporal

Combinando los puntos anteriores:

- Para cada subconjunto de vértices, el algoritmo realiza varias iteraciones sobre los vértices (O(N)), y dentro de esas iteraciones evalúa las distancias entre los vértices dentro del subconjunto (otro coste de O(N)).
- Finalmente, la combinación de estos pasos da lugar a la siguiente complejidad temporal:

$$T(n) = O(n^2 \times 2^n)$$

Esto se desglosa en:

- 2^N: número de subconjuntos posibles de vértices.
- n: número de vértices a considerar por subconjunto.
- n: número de vértices que se deben evaluar en cada paso para calcular la distancia mínima.

Análisis de eficiencia

El algoritmo **tspIterative** presenta una solución eficiente para el problema del TSP en comparación con la fuerza bruta. Aunque sigue siendo exponencial, su complejidad **$O(n^2 * 2^n)$** es más manejable que la **O(n!)** de métodos como la fuerza bruta. Utiliza programación

dinámica con *bitmasking* para almacenar los resultados de subproblemas y evitar la evaluación redundante de rutas, lo que mejora la eficiencia.

A pesar de sus ventajas, como la garantía de encontrar la solución óptima, el algoritmo no es adecuado para grafos grandes debido a su complejidad exponencial. A medida que el número de vértices aumenta, la cantidad de tiempo necesaria para calcular la solución se vuelve prohibitiva. Aunque es útil para grafos de tamaño pequeño a mediano, su aplicabilidad se ve limitada en escenarios más grandes, donde se prefieren métodos aproximados o algoritmos Greedy más escalables.

4.6 TspRecursive

El algoritmo **tspRecursive** resuelve el problema del **Viajante de Comercio** utilizando programación dinámica y memorización para evitar cálculos redundantes. A través de una función recursiva, evalúa todas las posibles rutas que visitan todos los nodos del grafo exactamente una vez, con el objetivo de encontrar la ruta de menor costo. Aunque la solución es óptima, el algoritmo sigue siendo exponencial debido a la cantidad de combinaciones posibles de nodos a evaluar, lo que limita su uso para grafos grandes.

Estudio de implementación - Funcionamiento del algoritmo

El algoritmo **tspRecursive** ofrece una solución exacta al problema del viajante de comercio mediante un enfoque recursivo con programación dinámica. Utiliza una matriz memo para guardar los resultados de subproblemas ya resueltos, evitando así realizar los mismos cálculos múltiples veces. A su vez, mantiene una tabla `prev` que permite reconstruir el camino óptimo una vez se ha calculado el coste mínimo. La clave del algoritmo está en cómo representa el conjunto de nodos visitados mediante una codificación binaria bitmask, lo que permite un seguimiento eficiente del estado de la solución durante la recursión.

El núcleo del algoritmo es la función `tspRecursiveAux`, que para cada nodo calcula recursivamente el coste mínimo de visitar todos los demás vértices restantes y volver al punto de partida. Esta función explora todos los vértices no visitados, actualiza el estado y acumula los costes parciales. Al finalizar, la matriz `memo` contiene los costes mínimos de cada subcamino, y `prev` indica qué nodo visitar a continuación en la ruta óptima. Esta estrategia permite mantener la precisión de una solución exacta reduciendo significativamente el número de caminos evaluados respecto a la fuerza bruta.

Una vez terminada la recursión, se reconstruye el camino óptimo utilizando la tabla `prev`, y se encapsula en una instancia de la clase `Solucion`, junto con su coste total. Aunque más clara y directa que otras implementaciones, `tspRecursive` sigue siendo limitada a grafos pequeños por su naturaleza exponencial, aunque considerablemente más eficiente que una búsqueda completamente exhaustiva.

Estudio Teórico - Análisis de complejidad del algoritmo

El algoritmo **tspRecursive** aplica programación dinámica con memorización y bitmasking para resolver el problema del viajante de comercio. La eficiencia del algoritmo depende de cuántos subproblemas distintos se generan y cuántas llamadas recursivas se realizan para resolverlos, lo que puede variar entre el mejor y el peor de los casos.

Mejor Caso: ($O(n \times 2^n)$)

En el mejor escenario, algunas ramas del árbol de recursión se podan rápidamente gracias a la memorización: cuando el algoritmo detecta que un subproblema ya ha sido resuelto, reutiliza su resultado en lugar de volver a calcularlo. Si las distancias están dispuestas de forma que ciertas rutas son claramente peores que otras, esto puede reducir el número de exploraciones necesarias. En este caso, la complejidad temporal queda limitada por la cantidad total de subproblemas únicos, que es de $O(n \times 2n)$, siendo n el número de vértices.

Aquí, cada combinación de nodo y subconjunto visitado solo se calcula una vez y se guarda en la tabla memo.

Peor Caso ($O(n^2 \times 2^n)$):

En el peor caso, el algoritmo no consigue evitar muchas de las llamadas recursivas porque los caminos alternativos tienen costes muy similares o porque la estructura del grafo no favorece una poda temprana. Esto implica que, además de resolver los $O(n \times 2^n)$ subproblemas únicos, para cada uno de ellos se pueden llegar a realizar hasta $O(n)$ comparaciones al intentar encontrar el siguiente vértice óptimo. Esto eleva la complejidad total a $O(\mathbf{n^2 \times 2^n})$, ya que para cada subproblema se exploran hasta n posibles extensiones del camino.

Análisis de eficiencia

El algoritmo tspRecursive utiliza programación dinámica con memorización para resolver el problema del viajante de comercio, ofreciendo una mejora significativa respecto al enfoque de fuerza bruta. Su principal fortaleza es que evita la repetición de cálculos mediante el almacenamiento de soluciones parciales, lo que permite encontrar el coste mínimo de manera más eficiente. Aunque su complejidad sigue siendo exponencial, el uso de recursión y bitmasking hace que la implementación sea clara y estructurada, lo cual facilita su comprensión y mantenimiento.

Aun así, este enfoque tiene sus limitaciones. A medida que aumenta el número de vértices, el número de subproblemas crece rápidamente, haciendo que el tiempo de ejecución sea inviable para grafos grandes. Por tanto, aunque garantiza la solución óptima, su uso se restringe a instancias pequeñas o medianas del problema, y no es adecuado en contextos donde se requiera escalabilidad o tiempos de respuesta rápidos**.**

4.7 TspGreedyVertex

El algoritmo ***tspGreedyVertex*** es una solución heurística al problema del viajante de comercio, basada en una estrategia voraz o greedy. A diferencia de las soluciones exactas que garantizan el recorrido mínimo evaluando todas las permutaciones posibles, este enfoque busca eficiencia sacrificando la optimalidad. Su funcionamiento se basa en seleccionar, desde el vértice actual, el vecino más cercano que aún no ha sido visitado, hasta recorrer todos los vértices del grafo.

Al finalizar, se regresa al vértice de partida para cerrar el ciclo. Este tipo de aproximación es especialmente útil en situaciones donde se requiere una solución rápida y aceptable, como en grafos de gran tamaño o en entornos con recursos limitados.

Estudio de implementación - Funcionamiento del algoritmo

En la implementación del algoritmo `tspGreedyVertex`, se parte de un vértice inicial proporcionado como argumento. A partir de este, se inicializa un conjunto para llevar el control de los vértices visitados, una lista para registrar el recorrido y una variable que acumula el coste total del camino. A medida que se avanza, el algoritmo selecciona iterativamente el vecino más cercano al vértice actual que no haya sido visitado, usando la función auxiliar `vecinoMasCercano`. Esta función itera sobre los vecinos del vértice actual y devuelve aquel con el menor peso de arista, siempre que no haya sido ya visitado.

El proceso se repite hasta que todos los vértices del grafo han sido incluidos en el recorrido. Finalmente, el algoritmo cierra el ciclo conectando el último vértice visitado con el vértice inicial, y suma ese coste al total. La solución resultante se encapsula en un objeto `Solucion`, que contiene tanto la lista de vértices en el orden del recorrido como el peso total del camino.

Esta implementación es simple, eficiente y fácil de entender, lo que la hace ideal para entornos donde la prioridad sea el rendimiento por encima de la optimalidad absoluta.

Análisis de complejidad

El análisis de complejidad temporal del algoritmo `tspGreedyVertex` se basa en cómo se lleva a cabo la búsqueda del vecino más cercano en cada iteración del ciclo principal. El bucle `while` recorre todos los vértices del grafo, ya que en cada iteración se visita un nuevo vértice. El número de iteraciones es igual al número de vértices del grafo, es decir, O(n), donde n es el número de vértices.

Dentro de cada iteración, el algoritmo debe buscar el vecino más cercano que no haya sido visitado. Esto requiere iterar sobre todos los vecinos del vértice actual, lo que tiene un costo de O(n) en el peor de los casos, ya que el vértice actual puede tener hasta n-1 vecinos. Por

lo tanto, para cada uno de los n vértices visitados, el algoritmo realiza O(n) comparaciones para encontrar el vecino más cercano.

Al combinar estas dos operaciones, n iteraciones con n comparaciones por iteración, la complejidad temporal total del algoritmo es **O(n^2),** donde n es el número de vértices del grafo. Esto significa que el algoritmo tiene una complejidad cuadrática, lo que lo hace adecuado para grafos de tamaño pequeño a mediano, pero poco eficiente para grafos grandes.

Análisis de eficiencia

El algoritmo `tspGreedyVertex` es una solución eficiente y fácil de implementar para el problema del vendedor viajero, especialmente cuando se busca una aproximación rápida. Gracias a su enfoque greedy, selecciona siempre el vecino más cercano, lo que le permite reducir significativamente el tiempo de ejecución en comparación con otros algoritmos más complejos, como los basados en programación dinámica o fuerza bruta.

Sin embargo, su principal desventaja radica en que no garantiza la solución óptima. Al seleccionar siempre el vecino más cercano sin considerar las implicaciones a largo plazo de esa elección, puede terminar con una ruta subóptima. Además, aunque su complejidad es cuadrática, puede resultar ineficiente para grafos de gran tamaño.

En resumen, `tspGreedyVertex` es ideal para soluciones rápidas y aproximadas en grafos pequeños, pero para casos donde la exactitud es crucial o el tamaño del grafo es considerable, es recomendable optar por algoritmos más avanzados.

4.7 TspGreedyEdges

El algoritmo `tspGreedyEdges` es un enfoque heurístico para resolver el problema del vendedor viajero utilizando una estrategia greedy basada en la selección de aristas. A diferencia de otros enfoques que seleccionan vértices, este algoritmo se centra en las aristas del grafo. Primero, genera todas las aristas posibles del grafo y las ordena en función de su peso. Luego, construye una solución mediante la selección de aristas que conecten vértices aún no visitados, asegurándose de no formar ciclos o superar el grado máximo permitido para cada vértice. Este algoritmo es útil cuando se desea una solución aproximada rápida, aunque no necesariamente óptima, para grafos de tamaño moderado.

Estudio de implementación - Funcionamiento del algoritmo

El algoritmo tspGreedyEdges hace uso de varias estructuras de datos clave para gestionar las aristas, los vértices y la construcción del recorrido. En primer lugar, se emplea una lista de aristas `(List<Arista>)` para almacenar todas las aristas del grafo, las cuales son luego ordenadas por su peso utilizando el método `sort()` .

Este enfoque asegura que el algoritmo siempre intente elegir la arista más ligera disponible, siguiendo la estrategia greedy. Posteriormente, se utilizan varios mapas: un `Map<Vertice, Integer>` para llevar un registro del grado de cada vértice, es decir, el número de aristas seleccionadas que inciden sobre él, y un `Map<Vertice, Integer>` para asociar cada vértice con un identificador de componente. Este último mapa ayuda a asegurar que las aristas seleccionadas no formen ciclos, garantizando así que el grafo permanezca conectado y acíclico.

Además, se utiliza un `Map<Vertice, List<Vertice>>` llamado **`selectedEdges`**, que almacena las aristas seleccionadas para cada vértice, permitiendo reconstruir el recorrido final una vez que se ha elegido un conjunto válido de aristas. Para la exploración del recorrido, se emplean una lista, `List<Vertice>`, para almacenar el camino final y un conjunto, `Set<Vertice>`, para realizar un seguimiento de los vértices visitados, lo que evita ciclos y asegura que cada vértice se visite solo una vez.

Finalmente, el algoritmo utiliza un bucle principal para construir el recorrido, iterando a través de las aristas seleccionadas hasta que todos los vértices estén visitados y el camino vuelva al vértice inicial.

Análisis de complejidad

El algoritmo `tspGreedyEdges` adopta un enfoque voraz centrado en las aristas del grafo en lugar de en los vértices, priorizando la inclusión de las conexiones más baratas para construir un ciclo hamiltoniano. La idea central consiste en ordenar todas las aristas posibles del grafo según su peso, comenzando con aquellas de menor coste. A partir de ahí, se seleccionan progresivamente las aristas que pueden añadirse al recorrido sin formar ciclos prematuros ni exceder dos conexiones por vértice, garantizando que se respete la estructura de un ciclo cerrado válido.

Este procedimiento recuerda en parte al algoritmo de Kruskal utilizado para obtener árboles de expansión mínima, aunque en este caso se introducen restricciones adicionales propias del TSP. En lugar de construir un árbol, se busca formar un ciclo único que recorra todos los vértices exactamente una vez. Para lograrlo, el algoritmo mantiene estructuras de datos que controlan el grado de cada vértice y los componentes conexos, permitiendo tomar decisiones informadas a medida que se procesan las aristas ordenadas.

Aunque el método no garantiza obtener la solución óptima, suele ofrecer resultados aceptables en muchos escenarios prácticos. Su mayor ventaja frente a algoritmos exactos radica en su eficiencia: el paso más costoso es la ordenación de las aristas, lo que domina la complejidad total con un orden de O(n2log n), dado que en un grafo completamente conectado existen O(n^2) aristas. Esta heurística es especialmente útil cuando se necesita una solución aproximada con buena calidad y bajo coste computacional, lo que la convierte en una alternativa razonable para problemas de tamaño moderado.

Análisis de eficiencia

El algoritmo `tspGreedyEdges` ofrece una buena combinación entre eficiencia y calidad de solución, siendo útil cuando se dispone de poco tiempo de cómputo. Su principal ventaja es que genera recorridos viables sin explorar todas las permutaciones posibles, lo que permite obtener resultados razonables en grafos de tamaño moderado. Utiliza una estrategia de selección de aristas de menor coste, lo que a menudo le permite competir con métodos más complejos, y su implementación clara facilita su comprensión.

Sin embargo, presenta limitaciones: al ser una heurística basada en decisiones locales, no garantiza soluciones óptimas y puede generar recorridos poco eficientes en ciertos casos. Su complejidad es $O(n^2 \log n)$, lo que puede ser un obstáculo en grafos grandes. Además, manejar correctamente restricciones como el grado de los vértices o la conexión de componentes añade dificultad a su implementación.

.

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 puntos 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

- Pruebas con grafos reales proporcionados (**graphTSP01.txt hasta graphTSP11.txt**), con número de nodos creciente (**10 a 24**) y conexión completa.

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 un rango de tamaños de nodos y densidades específicas. 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.

- Pruebas con grafos generados aleatoriamente, variando tanto el tamaño del grafo **(8, 12, 16, 20 nodos)** como su densidad **(25%, 50%, 75% y 100%).**

Los algoritmos evaluados han sido:

- Exactos:
    - tspBruteForce (fuerza bruta)
    - tspRecursive (programación dinámica recursiva con memoización)
    - tspIterative (programación dinámica iterativa - Bellman-Held-Karp)
- Heurísticos:
    - tspGreedyVertex (vecino más cercano)
    - tspGreedyEdges (selección por aristas más cortas)

Cada experimento se repitió varias veces, midiendo:

- Tiempo de ejecución en milisegundos.
- Distancia total de la ruta hallada.
- Existencia o no de ciclo hamiltoniano en grafos de baja densidad.

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

Las siguientes tablas resumen los resultados empíricos obtenidos, donde podemos ver la evolución del desempeño de los algoritmos conforme el conjunto de nodos es mayor en tamaño y/o densidad.

5.3.1. Pruebas con archivos proporcionados

Archivo	Nodos	Brute Force (ms)	Recursivo (ms)	Iterativo (ms)	Greedy Vért. (ms)	Greedy Aristas (ms)	Distancia Óptima
graphTSP01.txt	10	29	0	4	0	2	1069
graphTSP02.txt	15	N/A	18	30	0	0	1281
graphTSP03.txt	16	N/A	39	28	0	0	1378
graphTSP04.txt	17	N/A	79	83	0	0	1442
graphTSP05.txt	18	N/A	199	191	8	0	1529
graphTSP06.txt	19	N/A	710	469	0	0	1696
graphTSP07.txt	20	N/A	1641	1680	0	0	1764
graphTSP08.txt	21	N/A	4200	3359	0	0	1810
graphTSP09.txt	22	N/A	10741	8455	3	1	1981
graphTSP10.txt	23	N/A	33804	22956	0	0	2032
graphTSP11.txt	24	N/A	78797	49648	0	1	2163

El algoritmo de fuerza bruta demuestra su viabilidad únicamente en instancias de tamaño reducido (n = 10), con un tiempo de ejecución de 29 ms, y deja de ser aplicable (“N/A”) a partir de 15 nodos debido a la explosión combinatoria inherente al crecimiento factorial.

El método recursivo exhibe también un comportamiento exponencial, incrementándose desde 18 ms en 15 nodos hasta aproximadamente 79 ms en 17 nodos y alcanzando valores cercanos a los 78 s cuando n = 24. **Su versión iterativa** mejora notablemente estos resultados, reduciendo tiempos de forma significativa (por ejemplo, de 0 ms frente a 4 ms en 10 nodos y de 49 648 ms frente a 78 797 ms en 24 nodos).

Las heurísticas voraces basadas en selección de vértice y de arista mantienen tiempos de respuesta constantes (0–8 ms) incluso para 24 nodos, si bien esta eficiencia temporal se logra a costa de renunciar a la garantía de optimalidad de la solución, como sugiere la diferencia frente a la distancia óptima reportada.

5.3.2. Pruebas con grafos generados aleatoriamente

Nodos	Densidad (%)	Greedy Vert. (ms)	Greedy Vert. Dist.	Greedy Edge (ms)	Greedy Edge Dist.	Iterativo (ms)	Iterativo Dist.	¿Hay Ciclo TSP?
8	25	N/A	N/A	N/A	N/A	4	441	Sí (Iterativo)

8	50	N/A	N/A	N/A	N/A	3	311	Sí (Iterativo)
8	75	N/A	N/A	N/A	N/A	2	244	Sí (Iterativo)
8	100	0	240	1	233	2	193	Sí
12	25	N/A	N/A	N/A	N/A	6	433	Sí (Iterativo)
12	50	N/A	N/A	N/A	N/A	5	359	Sí (Iterativo)
12	75	N/A	N/A	N/A	N/A	4	219	Sí (Iterativo)
12	100	1	224	1	171	4	164	Sí
16	25	N/A	N/A	N/A	N/A	53	531	Sí (Iterativo)
16	50	N/A	N/A	N/A	N/A	33	336	Sí (Iterativo)
16	75	N/A	N/A	N/A	N/A	27	239	Sí (Iterativo)
16	100	2	258	2	229	34	208	Sí
20	25	N/A	N/A	N/A	N/A	1680	553	Sí (Iterativo)
20	50	N/A	N/A	N/A	N/A	1596	344	Sí (Iterativo)
20	75	3	348	N/A	N/A	1606	243	Sí
20	100	2	342	2	334	1478	225	Sí

El algoritmo iterativo consigue hallar un ciclo hamiltoniano en todos los casos (“Sí”), sin importar la densidad (25 %–100 %) ni el número de nodos (8–20). No obstante, su tiempo de ejecución crece de manera aproximada lineal con el tamaño de la instancia, pasando de 4 ms en grafos de 8 nodos al 25 % de densidad hasta 1 478 ms en grafos completos de 20 nodos.

Asimismo, a mayor densidad se observa una ligera reducción en los tiempos y una mejora en la calidad de la ruta (distancias menores). Por su parte, **las heurísticas greedy** sólo generan solución cuando la densidad alta (>75 %), con tiempos mínimos (0–3 ms) y distancias algo superiores a las obtenidas por él, evidenciando su dependencia de la conectividad completa del grafo.

5.4 Análisis de resultados

Dividiremos este análisis en tres partes fundamentales observando el comportamiento temporal, las capacidades de las soluciones heurísticas y las curvas de crecimiento calculadas

1. Comportamiento temporal

El algoritmo fuerza bruta solo fue viable para grafos de 10 nodos, debido a su complejidad $O(n!)$

Los algoritmos recursivo e iterativo ejecutaron correctamente hasta 24 nodos, aunque el tiempo se vuelve prohibitivo a partir de 20 nodos (más de 30 segundos).

Las soluciones heurísticas (Greedy) mostraron tiempos constantes y muy bajos incluso para 20 nodos, lo que las hace idóneas para grafos grandes.

2. Calidad de las soluciones heurísticas

Greedy por aristas suele encontrar recorridos de mejor calidad que Greedy por vértices, especialmente cuando la densidad es alta (≥ 75%).

En grafos no completos, solo el algoritmo iterativo garantiza la búsqueda de circuitos Hamiltonianos.

Para instancias grandes (n > 20), la diferencia de distancia respecto a la óptima es moderada, validando el uso de heurísticas en casos prácticos.

3. Curvas de crecimiento

Los resultados confirman las complejidades teóricas:

- BruteForce: $O(n!)$
- Recursive/Iterative: $O(n^2 * 2n)$
- Greedy: $O(n^2)$

Ajustes mediante regresión polinómica y logarítmica muestran coeficientes $R^2 > 0.95$ para el ajuste esperado.

Como conclusión, los resultados confirman que los algoritmos exactos sólo resultan prácticos en grafos de pequeño tamaño, dado su elevado coste computacional, mientras que las heurísticas, y en particular el enfoque GreedyEdges, ofrecen soluciones en tiempos muy reducidos y con una calidad adecuadamente próxima al óptimo.

Por ello, en entornos reales donde el tiempo de respuesta es crítico, resulta altamente recomendable optar por estas heurísticas, reservando el cálculo exacto exclusivamente para redes de tamaño limitado en las que sea factible obtener la solución óptima.

6. ANEXOS

6.1 Pseudocódigo de los algoritmos

A continuación, se presentan los pseudocódigos correspondientes a los algoritmos implementados para la resolución del problema del viajante (TSP) y la generación de caminos simples en grafos. Estos algoritmos representan distintas estrategias de solución, desde aproximaciones completas mediante búsqueda exhaustiva y programación dinámica, hasta métodos heurísticos más eficientes como algoritmos voraces.

Esta recopilación refleja el progreso en complejidad y optimización de las distintas técnicas aplicadas en la búsqueda de soluciones viables para problemas NP-difíciles como el TSP.

Caminos simples

Algorithm 1 Encontrar todos los caminos simples desde un origen (Función Principal)

```
1: procedure SimplePaths(Grafo g, Vértice origen)
2:     resultado ← nueva Solución vacía                ▷ Camino parcial actual
3:     soluciones ← nueva ListaEnlazada de Solución    ▷ Lista de caminos completos encontrados
4:     visitados ← nuevo ConjuntoHash de Vértice       ▷ Nodos visitados en el camino actual
5:     for all Vértice destino en g.getVecinos(origen) do SimplePathsAux(origen, destino, g, resultado, visitados, soluciones)
6:     end for
7:     return soluciones
8: end procedure
```

TSP por caminos simples

Algorithm 3 TSP usando SimplePaths (Función Principal)

```
1: procedure TspSimplePaths(Grafo g, Vértice origen)
2:     resultado ← nueva Solución vacía                ▷ Camino parcial actual
3:     solucion_mejor ← nueva Solución con distancia ∞ ▷ Mejor solución encontrada
4:     visitados ← nuevo ConjuntoHash de Vértice       ▷ Nodos visitados en el camino actual
5:     for all Vértice destino en g.getVecinos(origen) do ▷ Itera sobre los posibles 'segundos' vértices TspSimplePathsAux(origen, destino, g, resultado, visitados, solucion_mejor)
6:     end for
7:     return solucion_mejor
8: end procedure
```

TSP fuerza bruta

Algorithm 5 TSP por Fuerza Bruta (Generando Permutaciones)

```
1: procedure TspBruteForce(Grafo g, Vértice origen)
2:     mejorSolucion ← nulo
3:     matriz ← g.mapaMatriz()          ▷ Obtener matriz de adyacencia/pesos
4:     nodo_inicio ← g.getVertexIndex(origen)      ▷ Índice del vértice origen
5:     n ← g.size()                                  ▷ Número de vértices
6:     permutacion ← nuevo array de n − 1 enteros
7:     j ← 0
8:     for i desde 0 hasta n − 1 do
9:         if i no es igual a nodo_inicio then
10:            permutacion[j] ← i
11:            j ← j + 1
12:        end if
13:    end for
14:    mejorRutaIndices ← nulo
15:    mejorCosto ← ∞
16:    repeat
17:        costoActual ← CalcularCosto(nodo_inicio, permutacion,
   matriz)
18:        if costoActual < mejorCosto then
19:            mejorCosto ← costoActual
20:            mejorRutaIndices ← clonar(permutacion)
21:        end if
22:    until no SiguientePermutacion(permutacion)
23:    rutaVertices ← g.obtenerListaVertices(mejorRutaIndices,
   nodo_inicio)          ▷ Convierte índices a vértices, añadiendo inicio
24:    rutaVertices.add(origen)   ▷ Añadir el origen al final para cerrar ciclo
25:    mejorSolucion ← nueva Solución(rutaVertices, mejorCosto)
26:    return mejorSolucion
27: end procedure
```

TSP recursivo

Algorithm 10 TSP Recursivo con Programación Dinámica (Held-Karp - Función Principal)

```
1: procedure TspRecursive(Grafo g, Vértice start)
2:     nodo_inicio ← g.getVertexIndex(start)
3:     distancia ← g.mapaMatriz()
4:     N ← distancia.length
5:     memo ← nueva Matriz N × 2^N de Doubles (inicializada a nulo)
6:     prev ← nueva Matriz N × 2^N de Enteros (inicializada a nulo)  ▷ Para reconstrucción
7:     estado_inicial ← 1 ≪ nodo_inicio  ▷ Máscara de bits con solo el nodo inicio visitado
8:     costoMinimo ← TspRecursiveAux(nodo_inicio, estado_inicial, memo, prev, distancia, nodo_inicio)
9:     tourIndices ← nueva Lista de Enteros
10:    indiceActual ← nodo_inicio
11:    estadoActual ← estado_inicial
12:    loop
13:        tourIndices.add(indiceActual)
14:        siguienteIndice ← prev[indiceActual][estadoActual]
15:        if siguienteIndice = nulo then
16:            break  ▷ Fin de la reconstrucción
17:        end if
18:        estadoSiguiente ← estadoActual ∨ (1 ≪ siguienteIndice)  ▷ Añadir nodo al estado
19:        estadoActual ← estadoSiguiente
20:        indiceActual ← siguienteIndice
21:    end loop
22:    tourIndices.add(nodo_inicio)  ▷ Cerrar el ciclo
23:    tourVertices ← g.obtenerListaVertices(tourIndices)
24:    solucion ← nueva Solución(tourVertices, costoMinimo)
25:    return solucion
26: end procedure
```

TSP iterativo

Algorithm 1 TSP Iterativo con PD (Held-Karp) (Ajustado)

procedure TspIterative(Grafo g, Vértice $startVertex$)
  $inicio \leftarrow g$.getVertexIndex($startVertex$)
  $distancia \leftarrow g$.mapaMatriz()
  $N \leftarrow distancia$.length
  $memo \leftarrow$ nueva Matriz $N \times 2^N$ de Doubles (inicializada a ∞)
  ▷ Inicialización: Costos para caminos de longitud 2
  for fin desde 0 hasta $N-1$ **do**
    if $fin \neq inicio$ **then**
      $estado \leftarrow (1 \ll inicio) \vee (1 \ll fin)$
      $memo[fin][estado] \leftarrow distancia[inicio][fin]$
    end if
  end for
  ▷ Iterar sobre tamaños de subconjuntos $r = 3..N$
  for r desde 3 hasta N **do**
    for all $subset$ en Combinations(r, N) **do**
      if NotIn($inicio$, $subset$) **then continue**
      end if
      for $next$ desde 0 hasta $N-1$ **do**
        if $next = inicio$ **o** NotIn($next$, $subset$) **then continue**
        end if
        $subsetPrev \leftarrow subset \oplus (1 \ll next)$ ▷ Subset sin 'next'
        $minDist \leftarrow \infty$
        ▷ Buscar el nodo 'prev' que minimiza el costo para llegar a 'next'
        for $prev$ desde 0 hasta $N-1$ **do**
          if $prev = inicio$ **o** $prev = next$ **o** NotIn($prev$, $subset$) **then**
            continue
          end if
          if $memo[prev][subsetPrev] < \infty$ **then** ▷ Si hay camino válido hasta 'prev'
            $newDist \leftarrow memo[prev][subsetPrev] + distancia[prev][next]$
            if $newDist < minDist$ **then**
              $minDist \leftarrow newDist$
            end if
          end if
        end for
        if $minDist < \infty$ **then**
          $memo[next][subset] \leftarrow minDist$
        end if
      end for
    end for
  end for
  $END\_STATE \leftarrow (1 \ll N) - 1$
  $minTourCost \leftarrow \infty$
  ▷ Calcular costo final volviendo al inicio
  for i desde 0 hasta $N-1$ **do**
    if $i \neq inicio$ **then**
      if $memo[i][END\_STATE] < \infty$ **then**
        $cost \leftarrow memo[i][END\_STATE] + distancia[i][inicio]$
        if $cost < minTourCost$ **then**
          $minTourCost \leftarrow cost$
        end if
      end if
    end if
  end for
  ▷ Reconstrucción del camino
  $tourIndices \leftarrow$ nueva Lista de Enteros
  $lastIdx \leftarrow inicio$
  $state \leftarrow END\_STATE$
  $tourIndices$.add($inicio$)
  for i desde $N-1$ hasta 1 decrementando **do**
    $bestIdx \leftarrow -1$
    $minVal \leftarrow \infty$
    for j desde 0 hasta $N-1$ **do**
      if $j = inicio$ **o** NotIn(j, $state$) **then continue**
      end if

TSP voraz

Algorithm 15 TSP Voraz por Vértice (Vecino Más Cercano)

```
1:  procedure TspGreedyVertex(Grafo g, Vértice startVertex)
2:      if startVertex = nulo o no g.contieneVertice(startVertex) then
3:          return nulo
4:      end if
5:      visitados ← nuevo ConjuntoHash de Vértice
6:      path ← nueva ListaEnlazada de Vértice
7:      totalWeight ← 0.0
8:      actual ← startVertex
9:      visitados.add(actual)
10:     path.add(actual)
11:     while visitados.size() < g.size() do
12:         siguiente ← VecinoMasCercano(g, visitados, actual)
13:         if siguiente = nulo then        ▷ No debería pasar en grafo completo
14:             Lanzar Excepción("No se encontró vecino no visitado")
15:         end if
16:         path.add(siguiente)
17:         w ← g.getPeso(actual, siguiente)
18:         totalWeight ← totalWeight + w
19:         visitados.add(siguiente)
20:         actual ← siguiente
21:     end while
22:     path.add(startVertex)                                  ▷ Cerrar el ciclo
23:     w ← g.getPeso(actual, startVertex)
24:     if w = nulo then
25:         Lanzar Excepción("No existe conexión de vuelta al origen")
26:     end if
27:     totalWeight ← totalWeight + w
28:     return nueva Solución(path, totalWeight)
29: end procedure
```

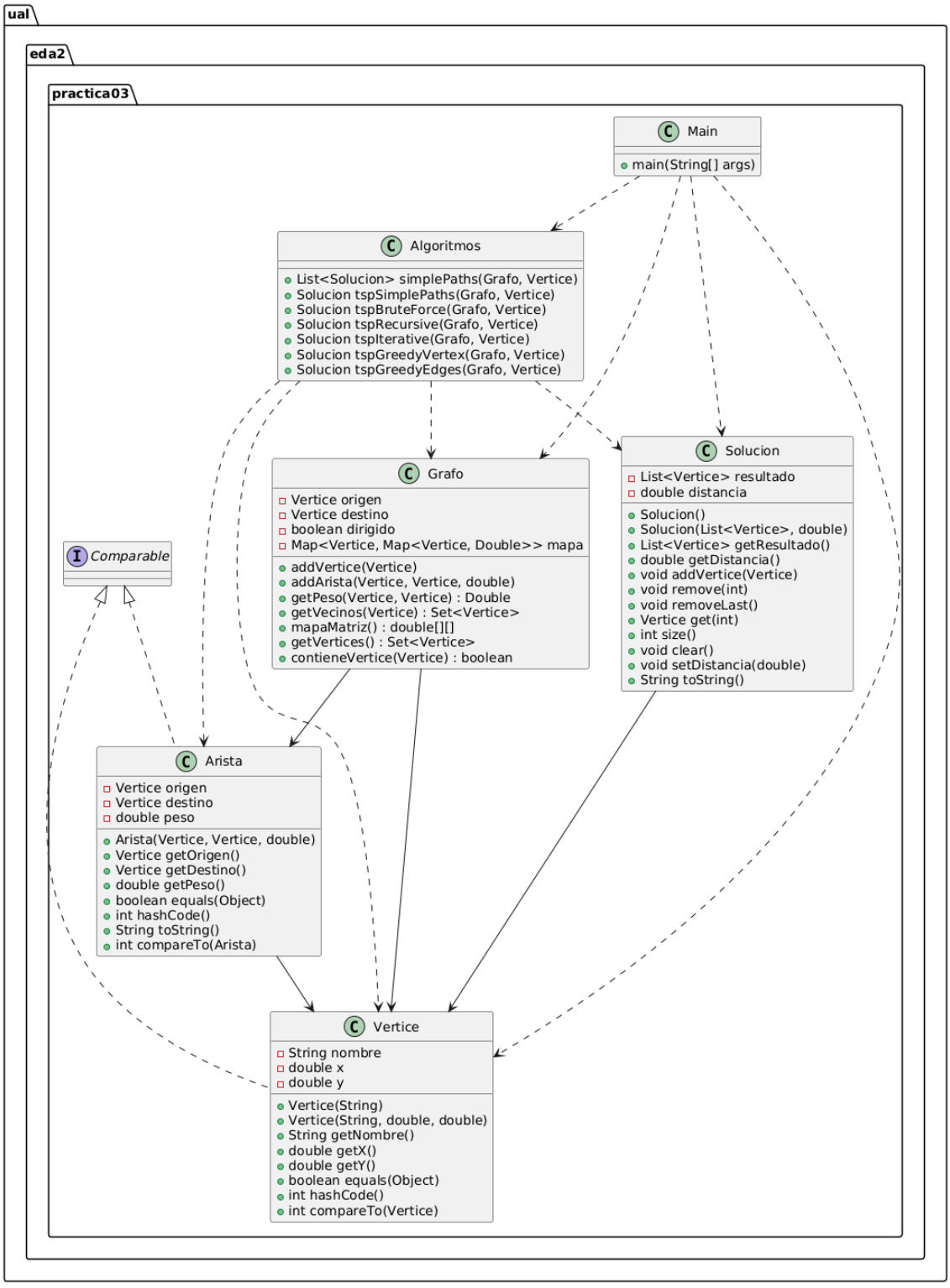
TSP voraz por aristas

Algorithm 2 TSP Voraz por Aristas (Enlace Más Barato) (Ajustado)

Require: Grafo g completo y conexo, Vértice $startVertex$ válido.
Ensure: Una Solución TSP aproximada.

```
1:  procedure TspGreedyEdges(Grafo g, Vértice startVertex)
2:      if startVertex = nulo o no g.contieneVertice(startVertex) then return nulo
3:      end if
4:      N ← g.size()
5:      aristas ← nueva Lista de Aristas                          ▷ Paso 2: Obtener y ordenar aristas
6:      for all Vértice v en g.getVertices() do
7:          for all Vértice u en g.getVecinos(v) do
8:              if v.compareTo(u) < 0 then
9:                  aristas.add(nueva Arista(v, u, g.getPeso(v, u)))
10:             end if
11:         end for
12:     end for
13:     Ordenar aristas por peso ascendente
                                  ▷ Paso 3: Inicializar grado, componentes y grafo parcial
14:     grado ← nuevo Mapa Vértice → Entero (init 0)
15:     componente ← nuevo Mapa Vértice → Entero (init único)
16:     compId ← 0
17:     for all Vértice v en g.getVertices() do
18:         componente.put(v, compId); grado.put(v, 0); compId ← compId + 1
19:     end for
20:     adj ← nuevo Mapa Vértice → Lista de Vértice               ▷ Adyacencia grafo parcial
21:     numAristas ← 0
                                  ▷ Paso 4: Seleccionar aristas válidas
22:     for all Arista e en aristas do
23:         u ← e.getOrigen(); v ← e.getDestino()
24:         if grado.get(u) ≥ 2 o grado.get(v) ≥ 2 then continue
25:         end if                                                ▷ Check grado
26:         esUltima ← (numAristas = N − 1)
27:         if componente.get(u) = componente.get(v) y no esUltima then continue
28:         end if                                                ▷ Check ciclo
                                  ▷ Añadir arista al grafo parcial
29:         Añadir v a la lista de u en adj
30:         Añadir u a la lista de v en adj
31:         numAristas ← numAristas + 1
32:         grado.put(u, grado.get(u) + 1)
33:         grado.put(v, grado.get(v) + 1)
                                  ▷ Unir componentes si son diferentes
34:         compU ← componente.get(u); compV ← componente.get(v)
35:         if compU ≠ compV then
36:             for all Vértice w en componente.keySet() do
37:                 if componente.get(w) = compV then
38:                     componente.put(w, compU)
39:                 end if
40:             end for
41:         end if
42:         if numAristas = N then break
43:         end if                                                ▷ Ciclo completo
44:     end for
45:     if numAristas ≠ N then Lanzar Excepción("No se formó ciclo")
46:     end if
                                  ▷ Paso 6: Reconstruir camino desde startVertex
47:     path ← nueva Lista; visited ← nuevo Conjunto
48:     totalW ← 0.0; curr ← startVertex
49:     path.add(curr); visited.add(curr)
50:     while path.size() ¡ N do
51:         nextFound ← falso
52:         for all Vértice neigh en adj.get(curr) do
53:             if no visited.contains(neigh) then
54:                 path.add(neigh)
55:                 totalW ← totalW + g.getPeso(curr, neigh)
56:                 visited.add(neigh)
57:                 curr ← neigh
```

6.2 Diagrama de clases



7. BIBLIOGRAFÍA Y REFERENCIAS

Algorithms for the travelling salesman problem. (s. f.). Recuperado 4 de mayo de 2025, de https://blog.routific.com/blog/travelling-salesman-problem

González, A. G. (2023, enero 15). Simulando el problema del viajante en Java. Panama Hitek. https://panamahitek.com/simulando-el-problema-del-viajante-en-java/

Travelling salesman problem using dynamic programming. (2013, noviembre 3). GeeksforGeeks. https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/