Estructura de Datos y Algoritmos II

Curso 2024-2025

Práctica 4 – Backtracking y Branch-and-Bound

Alumnos:

Alejandro Doncel Delgado

Jorge Godoy Beltrán

Índice de contenidos

1. OBJETIVO

1.1 Justificación de la aplicación de Programación Dinámica

Las técnicas de **Backtracking** y **Branch-and-Bound** son las estrategias algorítmicas seleccionadas para abordar el problema de asignación de proyectos a directores en EDAPital. Estos métodos son especialmente adecuados para problemas de optimización combinatoria donde se busca la mejor solución entre un vasto conjunto de posibilidades.

A diferencia de una búsqueda exhaustiva que evaluaría todas las asignaciones posibles, lo que resulta inviable factorialmente, Backtracking explora el espacio de soluciones de forma sistemática, construyendo candidatos paso a paso y descartando ramas enteras del

árbol de búsqueda que no cumplen con los criterios o no pueden llevar a una solución óptima (poda). Branch-and-Bound es una mejora de Backtracking para problemas de optimización, que utiliza funciones de cota optimistas y pesimistas, para explorar de manera más eficiente el espacio de soluciones, priorizando aquellas ramas que parecen más prometedoras y podando de forma más agresiva las que no pueden conducir a una mejora de la mejor solución encontrada hasta el momento.

Para el problema de EDAPital, donde se deben asignar **n** proyectos a **n** directores de manera unívoca, buscando minimizar tiempos o maximizar eficacias, estas técnicas permiten:

Exploración Sistemática: Garantizan que todas las posibles asignaciones válidas sean consideradas, ya sea explícita o implícitamente mediante la poda.

Obtención de Soluciones Óptimas: Con una correcta implementación y funciones de cota adecuadas, pueden garantizar la obtención de la asignación óptima.

Manejo de Restricciones: Incorporan de manera natural la restricción de que cada proyecto se asigna a un único director y cada director gestiona un único proyecto.

1.2 Descripción general del problema

El desafío central de esta práctica se sitúa en EDAPital, el parque tecnológico de Almería, que ha recibido la aprobación de n proyectos tecnológicos estratégicos. EDAPital cuenta con n directores de proyectos, cada uno capaz de dirigir cualquiera de los proyectos, aunque con variaciones en tiempo y eficacia.

Para gestionar la asignación, la Dirección Técnica ha elaborado dos tablas:

- **T[i, j]:** Estima el tiempo en semanas que el director de proyectos i tardaría en completar el proyecto j con su equipo.
- **E[i, j]:** Estima la eficacia, en porcentaje, [0.0, 100.0] con la que el director de proyectos i realizaría el proyecto j.

El objetivo es realizar la asignación óptima de proyectos a directores con una asignación 1:1, considerando dos metas independientes:

1 **Minimizar la suma total de tiempos:** Se busca una asignación $(x_1, x_2 \dots x_n)$ es el proyecto asignado al director i, tal que se minimice la expresión $\sum_{i=1}^{n} T[i, x_1]$.

2 **Maximizar la suma total de eficacias**: Se busca una asignación $(x_1, x_2 \dots x_n)$ tal que se maximice la expresión $\sum_{i=1}^{n} E[i, x_1]$.

Las soluciones deben ser permutaciones de los n proyectos, asegurando que cada proyecto se asigna a un director diferente y viceversa.

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 aborda el problema de la asignación óptima de proyectos a directores en EDAPital, un desafío común en la gestión de recursos que requiere encontrar la mejor correspondencia entre un conjunto de tareas y un conjunto de ejecutores según ciertos criterios de optimización. Para resolverlo, se emplean las técnicas algorítmicas de Backtracking y Branch-and-Bound.

Estos métodos se caracterizan por explorar el espacio de soluciones de una manera sistemática y estructurada. A diferencia de los algoritmos voraces, que toman decisiones locales con la esperanza de alcanzar una solución global óptima, o de la fuerza bruta, que evalúa todas las combinaciones posibles, Backtracking y Branch-and-Bound ofrecen un camino para encontrar soluciones óptimas mediante la construcción incremental de candidatos y la poda inteligente de aquellas ramas de exploración que no pueden conducir a la mejor solución.

En el contexto de la asignación de proyectos en EDAPital, donde se busca minimizar tiempos o maximizar eficacias, una mala asignación podría llevar a retrasos significativos, sobrecostes o una calidad de ejecución inferior a la deseada. Las técnicas de Backtracking y Branch-and-Bound permiten modelar las restricciones del y explorar las diferentes posibilidades para identificar la asignación que cumpla con el objetivo óptimo.

La elección de estas técnicas se fundamenta en la necesidad de equilibrar:

- **Optimalidad**: Garantizar que la asignación encontrada sea la mejor posible (la que minimice el tiempo total o maximice la eficacia total), evitando soluciones subóptimas que podrían tener consecuencias negativas.
- **Eficiencia Computacional**: Aunque en el peor de los casos la complejidad puede ser exponencial, las estrategias de poda inherentes a Backtracking y, especialmente, las funciones de acotación de Branch-and-Bound, permiten reducir drásticamente el espacio de búsqueda explorado en comparación con la fuerza bruta, haciendo factible la resolución de instancias de tamaño considerable.

A lo largo de esta práctica, se implementarán y evaluarán variantes de estos esquemas para los dos objetivos propuestos:

1. **Backtracking**: Construirá soluciones paso a paso, retrocediendo cuando una rama no sea viable o no pueda mejorar la solución actual.
2. **Branch-and-Bound**: Utilizará cotas para guiar la búsqueda y podar de forma más efectiva, enfocándose en las asignaciones más prometedoras.

3.1 Alternativas contempladas y solución final adoptada

Para resolver el problema de asignación de proyectos en EDAPital, se ha optado por implementar y analizar dos esquemas algorítmicos principales: **Backtracking** y **Branch-and-Bound.** Ambas técnicas son adecuadas para la naturaleza combinatoria y de optimización del problema, donde se busca una permutación de asignaciones que minimice tiempos o maximice eficacias. A continuación, se describen brevemente los enfoques específicos implementados dentro de cada esquema.

3.1.1 Enfoques con Backtracking

El Backtracking es una técnica de búsqueda sistemática que construye la solución candidata de forma incremental. Si en algún punto se determina que la solución parcial no puede conducir a una solución válida o óptima, el algoritmo "retrocede" para explorar otras alternativas. Para el problema de asignación, esto implica asignar secuencialmente proyectos a directores, verificando en cada paso la viabilidad y utilizando una función de estimación para podar ramas no prometedoras del árbol de búsqueda.

Backtracking para Minimización de Tiempos - backtrackingMin

Este algoritmo busca la asignación de proyectos a directores que resulte en la menor suma total de tiempos estimados.

Funcionamiento: Construye una asignación director por director. En cada nivel k, se prueba a asignarle cada proyecto j no asignado previamente. Se calcula el coste acumulado y se utiliza una función de estimación del coste restante, basada en los mínimos tiempos posibles para los directores restantes, como `estimacion = calcularEstimacion(matriz, true)` que utiliza `rapido[i]`, para podar la búsqueda si el coste actual más la estimación supera el mejor coste encontrado hasta el momento.

Poda: Se descarta una rama si `coste_actual + estimacion_restante >= mejor_coste_conocido`.

Backtracking para Maximización de Eficacias – backtrackingMax

Este algoritmo tiene como objetivo encontrar la asignación que maximice la suma total de las eficacias estimadas.

Funcionamiento: Similar al de minimización, pero adaptado para maximización. En cada nivel, se calcula la eficacia acumulada. La función de estimación calcula la máxima eficacia posible para las asignaciones restantes, basada en las máximas eficacias posibles para los directores restantes, `como estimacion = calcularEstimacion(matriz, false)` que utiliza `mejor[i]` para decidir si se continúa explorando una rama.

Poda: Se descarta una rama si `eficacia_actual + estimacion_restante <= mejor_eficacia_conocida`.

Limitaciones Generales del Backtracking - sin poda muy efectiva

Complejidad Potencialmente Exponencial: En el peor de los casos, si la poda no es muy efectiva, el algoritmo puede necesitar explorar una gran parte del espacio de soluciones, que es de orden factorial **(O(n!)).** La efectividad de las funciones de estimación es crucial.

Sensibilidad a la Estimación: La calidad de la función de estimación impacta directamente en el rendimiento. Una estimación poco ajustada puede llevar a una poda escasa, mientras que una demasiado compleja podría añadir sobrecarga.

3.1.2 Enfoques con Branch-and-Bound

Branch-and-Bound es una técnica de optimización que mejora el Backtracking mediante un uso más sofisticado de cotas para guiar la búsqueda y podar el espacio de soluciones de manera más eficiente. Mantiene una lista de nodos vivos, soluciones parciales y explora el más prometedor según una función de cota optimista.

Branch-and-Bound para Minimización de Tiempos - branchAndBoundMin

Busca la asignación de proyectos a directores que **minimice la suma total de tiempos**, utilizando una cola de prioridad para gestionar los nodos vivos.

Funcionamiento: Cada nodo en la cola de prioridad representa una asignación parcial. Los nodos se ordenan según una estimación optimista del coste total que se podría alcanzar desde esa asignación parcial, coste actual + estimación del coste restante usando `estimacion[nivel]` basada en `rapido[i]`. Se extrae el nodo más prometedor (menor estimación optimista). Si un nodo representa una solución completa y mejora la mejor solución conocida, se actualiza. Si la estimación optimista de un nodo ya es peor que la mejor solución completa conocida, se descarta.

Cola de Prioridad: Se utiliza una cola de mínimos, `PriorityQueue<Nodo>`, donde la prioridad la da el coste actual más la estimación optimista del coste restante.

Poda: Un nodo X se poda si su `coste_optimista(X) >= coste_mejor_solucion_conocida`.

Branch-and-Bound para Maximización de Eficacias – branchAndBoundMax

Tiene como objetivo encontrar la asignación que **maximice la suma total de eficacias**, también mediante una cola de prioridad

Funcionamiento: Similar al caso de minimización, pero adaptado para maximización. Los nodos en la cola de prioridad se ordenan según una estimación optimista de la eficacia

total, eficacia actual + estimación de la eficacia restante usando `estimacion[nivel]` basada en `mejor[i]` . Se extrae el nodo con la mayor estimación optimista.

Cola de Prioridad: Se utiliza una cola de máximos, implementada a través de la PriorityQueue con un comparador adecuado o negando valores si es necesario, aunque tu código parece usar la misma PriorityQueue y la condición de poda `newEstimacion > sol.getVoA()` maneja la lógica de maximización.

Poda: Un nodo X se poda si su `eficacia_optimista(X) <= eficacia_mejor_solucion_conocida`.

Limitaciones Generales del Branch-and-Bound:

***Complejidad Exponencial en el Peor Caso*:** Aunque suele ser más eficiente, la complejidad en el peor caso sigue siendo exponencial.

***Calidad de las Cotas*:** El rendimiento depende críticamente de la calidad de las funciones de cota optimista, y pesimista, si se usara explícitamente para la poda. Cotas más ajustadas conducen a una mejor poda.

***Uso de Memoria*:** La cola de prioridad puede llegar a almacenar una cantidad considerable de nodos en algunos casos.

4. ESTUDIO TEÓRICO Y DE IMPLEMENTACIÓN

4.1 Introducción

Tras presentar el contexto del problema de asignación de proyectos en EDAPital y la justificación para el uso de las técnicas de **Backtracking** y **Branch-and-Bound,** en esta sección se analizarán en detalle las cuatro soluciones algorítmicas implementadas. El objetivo es estudiar su funcionamiento interno, su eficiencia teórica y, de cara a la sección experimental posterior, su comportamiento práctico.

Para mantener un enfoque claro y estructurado, el estudio de cada uno de los algoritmos (Backtracking para minimización, Backtracking para maximización, Branch-and-Bound

para minimización y Branch-and-Bound para maximización) se organizará del siguiente modo:

- **Estudio de la implementación**: Se describirán los aspectos esenciales del funcionamiento de cada algoritmo, haciendo hincapié en las decisiones de diseño clave, las estructuras de datos utilizadas (como el manejo de soluciones parciales, el array de asignado o la cola de prioridad en Branch-and-Bound) y cómo se aplican las estrategias de poda o acotación específicas para cada caso.
- **Estudio teórico**: Se analizará la complejidad temporal y espacial de cada algoritmo implementado. Se destacarán sus ventajas y limitaciones inherentes en el contexto del problema de asignación, que es un problema de permutaciones.

A diferencia de enfoques que podrían tomar decisiones heurísticas rápidas pero no necesariamente óptimas, o de la fuerza bruta que exploraría todas las n! permutaciones de asignación de forma inviable, en esta práctica el énfasis se pone en la **búsqueda sistemática y la poda inteligente** del espacio de soluciones, característico del Backtracking y en la **exploración guiada por cotas** para alcanzar eficientemente la optimalidad, fundamental en Branch-and-Bound.

Este análisis permitirá comprender la efectividad de estas técnicas para resolver problemas de asignación complejos y sentará las bases para evaluar sus límites en escenarios de escalabilidad

4.2 Algoritmo Backtracking para Minimización de Tiempos, backtrackingMin

El algoritmo `backtrackingMin` está diseñado para resolver el problema de asignación de tareas, específicamente, para encontrar la asignación óptima de *n* proyectos a *n* directores de manera que la suma total de los tiempos de ejecución sea mínima. Utiliza una estrategia de Backtracking, o Vuelta Atrás, para explorar sistemáticamente el espacio de soluciones, construyendo una asignación paso a paso y podando aquellas ramas de búsqueda que no pueden conducir a una solución mejor que la ya encontrada.

Estudio de implementación - Funcionamiento del algoritmo

El propósito fundamental del algoritmo es encontrar una permutación de proyectos, uno para cada director, que minimice el tiempo total acumulado. Para ello, el algoritmo construye una solución candidata, `s`, que es un array donde `s[i]` representa el proyecto asignado al director `i`.

La implementación es iterativa y gestiona el proceso de exploración mediante una variable `nivel`, que indica el director actual, es decir, el índice del array `s`, para el cual se está intentando asignar un proyecto. El proceso principal se desarrolla en un bucle `do-while` que continúa mientras nivel sea válido, es decir, mayor o igual a -1. Dentro del bucle:

1. Se llama a **`generar(matriz, sol)`.** Este método intenta asignar el siguiente proyecto disponible al director del nivel actual. Incrementa `s[nivel]` y actualiza bact y el array asignado.

2. Se comprueba si se ha encontrado una solución completa y válida mediante `solucion()`. Esta función verifica si nivel ha alcanzado al último director y si la asignación actual es válida según `criterio()`, que comprueba que el proyecto s[nivel] esté asignado únicamente a este director en la solución parcial. Si es una solución completa y bact es menor que el VoA de sol, se actualiza sol con la nueva mejor asignación y su coste.

3. Se evalúa la condición de avance y poda: si no se ha llegado al último director, `nivel < s.length - 1`, la asignación actual es válida, criterio(), y el coste actual más la estimación para los niveles restantes es prometedora, `bact + estimacion[nivel] < sol.getVoA()`, entonces se avanza al siguiente nivel, nivel++.

4. Si no se puede avanzar o la rama no es prometedora, se retrocede. El bucle `while (nivel > -1 && !masHermanos())` llama a `retroceder(matriz)`. `masHermanos()` comprueba si hay más proyectos para probar en el nivel actual. retroceder deshace la última asignación, actualiza bact y asignado, y decrementa nivel.

El algoritmo finaliza cuando nivel se vuelve -1, lo que significa que se han explorado todas las ramas relevantes del espacio de búsqueda.

Estudio Teórico - Análisis de la complejidad del algoritmo

Sea *n* el número de directores y también el número de proyectos.

Complejidad temporal:

El algoritmo explora un árbol de permutaciones. En el peor escenario, si la función de estimación no permite una poda significativa, el algoritmo podría acercarse a explorar las n! posibles asignaciones.

Para cada nodo del árbol de búsqueda, las operaciones como generar, criterio, solucion y retroceder realizan una cantidad constante de trabajo o trabajo proporcional a n en el caso de algunas inicializaciones o la función calcularEstimacion.

La función calcularEstimacion(matriz, true) se ejecuta una vez al inicio. Implica iterar para cada director, desde n-2 hacia abajo, y para cada uno, encontrar el mínimo en la fila de la matriz del siguiente director, lo cual es O(n), y luego sumar. Esto resulta en una complejidad de $O(n^2)$ para el pre-cálculo de estimacion.

El bucle principal, en su peor caso, puede ejecutar un número de iteraciones relacionado con el número total de nodos en el árbol de búsqueda de permutaciones, que es del orden

de O(n!). Aunque la poda reduce este número, la cota superior teórica sigue siendo factorial.

Por lo tanto, **la complejidad temporal total en el peor caso es O(n!·n)** si consideramos el trabajo por nodo, o más ajustado O(n!) si las operaciones por nodo son amortizadas a O(1) respecto a la generación de nodos, aunque el pre-cálculo de estimaciones es $O(n^2)$. La complejidad dominante sigue siendo **O(n!).**

Complejidad espacial:

Las principales estructuras de datos utilizadas y su consumo de espacio son:

- **s**: array para la solución actual, de tamaño *n*. Ocupa O(n).
- **asignado**: array para marcar proyectos asignados, de tamaño *n*. Ocupa O(n).
- **estimacion**: array para las estimaciones, de tamaño *n*. Ocupa O(n).
- **sol**: objeto Solucion que almacena la mejor solución, que incluye un array de tamaño *n*. Ocupa O(n).
- **matriz**: la matriz de entrada, de tamaño n×n. Ocupa $O(n^2)$. Si se considera parte de la entrada, el espacio adicional es O(n). Dado que la implementación es iterativa, no hay una pila de recursión explícita que crezca con *n* más allá de las variables locales. Por tanto, la complejidad espacial adicional utilizada por el algoritmo es O(n), si no contamos la matriz de entrada, u O $O(n^2)$ si la incluimos

Análisis de eficiencia

El algoritmo backtrackingMin ofrece un método sistemático para encontrar la solución óptima al problema de asignación minimizando tiempos. Su principal ventaja sobre la fuerza bruta es la capacidad de **poda**. Al utilizar la función estimacion[nivel], que calcula una cota inferior del coste restante, el algoritmo puede descartar ramas enteras del espacio de búsqueda que se sabe que no conducirán a una solución mejor que la ya encontrada. Esto puede resultar en un ahorro computacional significativo, especialmente si las estimaciones son ajustadas. La implementación iterativa evita problemas de desbordamiento de pila que podrían ocurrir con una recursión profunda en lenguajes con limitaciones en el tamaño de la pila.

Sin embargo, la principal desventaja es su **complejidad temporal factorial en el peor de los casos**. Si las estimaciones no son lo suficientemente buenas o la estructura de costes del problema no favorece una poda temprana, el rendimiento puede degradarse rápidamente a medida que *n* aumenta. Esto limita su aplicabilidad práctica a problemas de tamaño moderado. La eficiencia real depende en gran medida de la calidad de la cota inferior utilizada para la poda y de la distribución de los costes en la matriz de entrada.

4.3 Algoritmo Backtracking para Maximización de Eficacias, backtrackingMax

El algoritmo backtrackingMax aborda el problema de asignación de proyectos a directores con el objetivo de encontrar la combinación que **maximice la suma total de las eficacias estimadas**. Al igual que su contraparte para minimización, emplea una estrategia de Backtracking, o Vuelta Atrás, para explorar el espacio de soluciones de forma sistemática, construyendo incrementalmente las asignaciones y aplicando criterios de poda para descartar ramas no prometedoras.

Estudio de implementación - Funcionamiento del algoritmo

El objetivo principal de backtrackingMax es identificar la permutación de asignaciones, un proyecto por director, que resulta en la mayor eficacia total posible. La lógica de construcción de la solución es similar a backtrackingMin, pero adaptada para un problema de maximización. El algoritmo opera de forma iterativa, utilizando variables clave con un significado análogo, pero orientado a la maximización

El flujo del algoritmo sigue el bucle do-while principal:

1. Se invoca **`generar(matriz, sol)`** para asignar el siguiente proyecto disponible al director del `nivel` actual, actualizando `bact` con la eficacia correspondiente y marcando el proyecto en `asignado`.
2. Mediante `solucion()`, se verifica si se ha alcanzado una asignación completa y válida, es decir, nivel es el último director y `criterio()` es verdadero. Si es así, y la `bact`, eficacia actual, es mayor que `sol.getVoA()`, la eficacia de la mejor solución anterior, se actualiza sol.
3. La condición de avance y poda se evalúa: si no se ha completado la asignación, `nivel < s.length - 1`, la asignación parcial es válida, criterio(), y la eficacia actual más la estimación de la eficacia máxima restante es mayor que la mejor eficacia encontrada, `bact + estimacion[nivel] > sol.getVoA()`, entonces se profundiza en la búsqueda, nivel++. Esta condición asegura que solo se exploran ramas que tienen el potencial de superar la mejor solución actual.
4. Si no se cumplen las condiciones para avanzar, se retrocede mediante el bucle `while (nivel > -1 && !masHermanos())`, que llama a `retroceder(matriz)`. Esta función anula la última asignación, ajusta `bact` y asignado, y disminuye nivel para explorar otras alternativas.

El algoritmo concluye cuando `nivel` desciende por debajo de 0, indicando que todas las trayectorias factibles y prometedoras han sido exploradas.

Estudio Teórico - Análisis de la complejidad del algoritmo

Sea *n* el número de directores y proyectos.

Complejidad temporal:

De forma similar a backtrackingMin, el algoritmo backtrackingMax explora un árbol de permutaciones. En el peor de los casos, la complejidad puede ser del orden de O(n!) si la poda no es efectiva. El pre-cálculo del array estimacion mediante calcularEstimacion(matriz, false), que implica encontrar la máxima eficacia por cada director restante y sumarlas, tiene una complejidad de $O(n^2)$.

Las operaciones dentro del bucle principal, por cada nodo explorado, son generalmente de tiempo constante o proporcionales a *n* en pasos específicos. La complejidad temporal teórica en el peor caso sigue siendo dominada por la naturaleza factorial de la exploración del espacio de soluciones, es decir, O(n!).

Complejidad espacial:

El uso de memoria es análogo al de backtrackingMin:

s: O(n).

asignado, proyectos asignados: O(n).

estimacion, cotas superiores: O(n).

sol, mejor solución: O(n).

matriz, $O(n^2)$. Por lo tanto, la complejidad espacial adicional, sin considerar la matriz de entrada, es O(n). Incluyendo la entrada, sería $O(n^2)$.

Análisis de Eficiencia

El algoritmo backtrackingMax está diseñado para encontrar la asignación óptima que maximiza la eficacia total. Su eficiencia se basa en la estrategia de Backtracking con poda. La principal ventaja es que, a diferencia de métodos heurísticos, garantiza la obtención de la solución óptima global si se le permite completarse. La poda, mediante la comparación de la eficacia actual más la estimación superior del restante, bact + estimacion[nivel], con la mejor solución ya encontrada, sol.getVoA(), permite evitar la exploración de una parte considerable del árbol de búsqueda.

No obstante, sufre de las mismas limitaciones inherentes al Backtracking. La complejidad temporal en el peor caso es factorial, O(n!), lo que lo hace impracticable para valores grandes de *n*. La efectividad de la poda depende crucialmente de cuán ajustada sea la cota superior proporcionada por la función estimacion. Si la estimación es consistentemente mucho mayor que la eficacia real alcanzable, la poda será menos frecuente y el algoritmo explorará más nodos. Su rendimiento práctico estará influenciado por la distribución de los valores de eficacia en la matriz de entrada y la calidad de la función de estimación para la maximización.

4.4 Algoritmo Branch-and-Bound para Minimización de Tiempos, branchAndBoundMin

El algoritmo `branchAndBoundMin` está diseñado para resolver el problema de asignación de proyectos a directores buscando la combinación que minimice la suma total de los tiempos estimados. Esta variante utiliza la técnica de Ramificación y Poda, Branch-and-Bound, que refina la búsqueda de Backtracking mediante el uso de una cola de prioridad para gestionar los nodos vivos, es decir, las soluciones parciales, y expandir siempre el nodo más prometedor según una cota optimista..

Estudio de implementación - Funcionamiento del algoritmo

El objetivo de `branchAndBoundMin` es encontrar la asignación 1:1 de proyectos a directores que resulte en el menor tiempo total acumulado. A diferencia del Backtracking simple que explora el árbol de soluciones en profundidad, Branch-and-Bound explora los nodos más prometedores primero.

Los componentes clave de la implementación son:

- **estimacion**: Un array precalculado, similar al usado en `backtrackingMin`, donde `estimacion[k]` contiene una cota inferior del tiempo mínimo para asignar los proyectos restantes desde el director k+1 en adelante. En tu código, parece que `estimacion[nivel]` del nodo padre se usa para calcular la estimación completa del hijo, lo que es una práctica común.
- **sol**: Objeto Solucion que almacena la mejor asignación completa encontrada y su coste. Se inicializa con una solución heurística o un coste muy alto. Tu código inicializa sol con una asignación diagonal y su coste.
- **LNV**: Una cola de prioridad, `PriorityQueue<Nodo>`, que almacena los nodos vivos. Cada Nodo encapsula una solución parcial, s, su coste actual, bact, el nivel, nivel, y una estimación del coste total, `costeEstimado = bact + cota_inferior_restante`. La cola ordena los nodos para que el que tenga el menor `costeEstimado` sea extraído primero.

El flujo del algoritmo es el siguiente:

1. Se inicializa la mejor solución sol con una primera asignación válida, por ejemplo, asignar el proyecto i al director i, y su coste asociado.
2. Se calcula el array estimacion para las cotas inferiores del coste restante.
3. Se crea un nodo raíz representando una asignación nula o el inicio de la asignación, con `nivel = 0, bact = 0`, y se añade a la cola de prioridad LNV. Su coste estimado será `0 + estimacion[0]`.
4. Este algoritmo de búsqueda encuentra la solución óptima explorando iterativamente una lista de nodos. En cada paso, extrae el nodo con el menor coste

estimado. Si este coste ya supera la mejor solución encontrada, se descarta (poda). Si el nodo es una solución completa y mejor, actualiza la solución óptima. Si no es completo pero es prometedor, genera nodos hijos y los añade a LNV si su estimación de coste es buena.

El algoritmo termina cuando la cola de prioridad LNV está vacía, momento en el cual sol contiene la asignación óptima.

Estudio Teórico - Análisis de la complejidad del algoritmo

Sea *n* el número de directores y proyectos.

Complejidad temporal

En el peor de los casos, Branch-and-Bound puede llegar a explorar una cantidad de nodos comparable a la de Backtracking, lo que llevaría a una complejidad de O(n!). Sin embargo, la eficacia de las cotas y la estrategia de seleccionar el nodo más prometedor suelen reducir significativamente el número de nodos explorados en la práctica en comparación con un Backtracking simple. El pre-cálculo de estimacion es $O(n^2)$.

Cada operación sobre la cola de prioridad, inserción y extracción tiene un coste de O(logM), donde M es el tamaño de la cola. En el peor caso, M podría ser grande, acercándose a O(n!) en escenarios patológicos, aunque en la práctica suele ser mucho menor. El número de nodos generados sigue siendo el factor dominante. Si las cotas son muy buenas, la complejidad puede ser mucho mejor que O(n!), pero no hay una garantía teórica mejor para el caso general.

Complejidad espacial

Las principales estructuras son:

- `s, asignado, estimacion, sol`: Similar a Backtracking, ocupan O(n) o $O(n^2)$, si se incluye la matriz de entrada.
- `LNV`, la cola de prioridad: En el peor de los casos, podría almacenar una gran cantidad de nodos. Si cada nodo almacena una copia de s, y hay M nodos, podría ser M·O(n). Si M es del orden de O(n!), el espacio sería O(n!·n), lo cual es prohibitivo. En la práctica, se espera que M sea manejable. Si los nodos almacenan referencias o se optimiza la gestión de s, el impacto puede variar. La especificación del PDF pide detallar la estructura del nodo.

Análisis de eficiencia

La principal ventaja de branchAndBoundMin sobre Backtracking simple es su **estrategia de exploración más inteligente**. Al expandir siempre el nodo con la cota inferior más prometedora, tiende a encontrar la solución óptima más rápidamente o a podar ramas más grandes del árbol de búsqueda de manera más efectiva.

Garantiza la **optimalidad** de la solución. La calidad de la cota inferior, estimacion, es crucial. Cuanto más ajustada sea la cota, es decir, más cercana al coste real sin subestimarlo, más eficiente será la poda y, por tanto, el algoritmo.

Las desventajas incluyen:

- **Sobrecarga de la cola de prioridad**: Las operaciones de inserción y extracción en la cola de prioridad tienen un coste logarítmico, lo que añade una pequeña sobrecarga en comparación con la gestión más directa del Backtracking iterativo.
- **Complejidad exponencial en el peor caso**: Al igual que Backtracking, puede ser lento para instancias grandes de *n*.

La eficiencia del `branchAndBoundMin` depende en gran medida de la calidad de la función de cota inferior y de la estructura específica del problema.

4.5 Algoritmo Branch-and-Bound Maximización de Eficacias, branchAndBoundMax

El algoritmo `branchAndBoundMax` se enfoca en resolver el problema de asignación de proyectos con el objetivo de **maximizar la suma total de las eficacias estimadas**. Similar a su homólogo de minimización, emplea la técnica de Ramificación y Poda, Branch-and-Bound, utilizando una cola de prioridad para dirigir la búsqueda hacia las soluciones parciales más prometedoras, pero adaptada para un contexto de maximización.

Estudio de implementación - Funcionamiento del algoritmo

El propósito de `branchAndBoundMax` es identificar la asignación 1:1 que produce la mayor eficacia acumulada. La estrategia general de Branch-and-Bound se mantiene, pero las cotas y la ordenación en la cola de prioridad se ajustan para la maximización.

Los elementos clave de la implementación son:

- **estimacion**: Array precalculado donde `estimacion[k]` guarda una cota superior de la máxima eficacia obtenible para los directores desde k+1 hasta el final. En tu código, `estimacion[nivel]` del nodo padre se utiliza para calcular la estimación completa del hijo.

- **sol**: Objeto `Solucion` que almacena la mejor asignación completa encontrada y su valor de eficacia. Se inicializa con una solución heurística y su eficacia, o un valor muy bajo. Tu código inicializa sol con una asignación diagonal.

- **LNV**: Una cola de prioridad, `PriorityQueue<Nodo>`. Para maximización, esta cola debe ordenar los nodos de manera que se extraiga primero el que tenga la mayor cota superior estimada, `eficaciaEstimada = bact + cota_superior_restante`. La implementación de PriorityQueue en Java es una cola de mínimos por defecto, por lo que para simular una cola de máximos, se

puede invertir el signo de las prioridades al insertar y extraer, o definir un comparador personalizado para la clase Nodo. Tu código usa directamente la PriorityQueue y la lógica de poda, `newEstimacion > sol.getVoA()`, asegura que se consideren nodos que puedan mejorar la solución máxima.

El algoritmo procede de la siguiente manera:

1. Se inicializa la mejor solución `sol` con una asignación inicial válida y su correspondiente eficacia total.
2. Se calcula el array `estimacion` con las cotas superiores de eficacia para las asignaciones restantes.
3. Se crea un nodo raíz, representando el inicio de la asignación, y se añade a LNV. Su estimación de eficacia será `0 + estimacion[0]`.
4. Este algoritmo busca la solución óptima maximizando la eficacia. Mientras la lista de nodos vivos LNV no esté vacía, extrae el nodo con la mayor eficacia estimada. Si la estimación de eficacia de este nodo es inferior a la mejor solución ya encontrada, se descarta. Si el nodo es una solución completa y su eficacia es superior a la mejor registrada, esta se actualiza. Si el nodo no es completo, genera nodos hijos. Aquellos hijos cuya nueva estimación de eficacia total supere la de la mejor solución actual se añaden a LNV para su posterior exploración.

El algoritmo finaliza cuando LNV se vacía, y sol contiene la asignación de máxima eficacia.

Estudio Teórico - Análisis de la complejidad del algoritmo

Sea *n* el número de directores y proyectos.

Complejidad temporal: Al igual que las otras variantes de Backtracking y Branch-and-Bound para este problema, la complejidad temporal en el peor caso puede alcanzar O(n!). La efectividad de las cotas superiores en la poda es determinante. El pre-cálculo de estimacion para la maximización es $O(n^2)$.Las operaciones en la cola de prioridad, asumiendo M como su tamaño máximo, son O(logM) por operación. El rendimiento depende de cuántos nodos se necesiten explorar y almacenar.

Complejidad espacial: El consumo de espacio es similar al de branchAndBoundMin:

- Estructuras para la solución actual y estimaciones: O(n) o $O(n^2)$.si se incluye la matriz de entrada.
- La cola de prioridad LNV: Su tamaño M en el peor caso puede ser considerable, potencialmente O(n!), haciendo que el espacio sea M·O(n) si cada nodo almacena una copia de la asignación.

Análisis de eficiencia

`branchAndBoundMax` está diseñado para encontrar la solución óptima en problemas de maximización, y su eficiencia radica en la **exploración guiada por cotas superiores**. Al priorizar nodos con mayor potencial de eficacia, puede converger hacia la solución óptima más rápido que un Backtracking simple o podar más ramas ineficaces. Asegura la **optimalidad** de la solución.

Sus desventajas son:

- **Complejidad exponencial en el peor caso**: Si las cotas superiores no son suficientemente ajustadas, el número de nodos a explorar puede ser muy alto.
- **Calidad de la cota superior**: El rendimiento del algoritmo está intrínsecamente ligado a cuán bien la función estimacion puede predecir la máxima eficacia alcanzable desde un nodo parcial. Una cota poco ajustada, es decir, demasiado alta, reducirá la efectividad de la poda.

5. ESTUDIO EXPERIMENTAL

A continuación se realiza el estudio experimental, llevado a cabo para evaluar y comparar el rendimiento de diferentes algoritmos de optimización. El objetivo principal es analizar cómo varía su eficiencia en función del tamaño del problema.

5.1 Metodología de experimentación

Para obtener conclusiones robustas sobre el rendimiento de las implementaciones de los algoritmos, se ha seguido una metodología experimental consistente en la ejecución de una serie de pruebas y el posterior análisis de los tiempos obtenidos.

Algoritmos Evaluados:

El estudio se centra en los siguientes algoritmos, considerando variantes para problemas de minimización ("Min") y maximización ("Max"):

1. **Backtracking (Vuelta Atrás):**
    - BacktrackingMin
    - BacktrackingMax
2. **Branch and Bound (Ramificación y Poda):**
    - BranchAndBoundMin
    - BranchAndBoundMax

Estos algoritmos son métodos exactos diseñados para encontrar soluciones óptimas en problemas de búsqueda y optimización.

Conjunto de Datos:

Se utilizó un conjunto de datos generado aleatoriamente para probar el rendimiento de los algoritmos implementados. Este conjunto de datos se caracteriza por:

- **Tamaños de Problema Crecientes:** Instancias con tamaños definidos de 4, 8, 12, 16, 20, 24 y 28 unidades. Esta variación progresiva en la magnitud del problema permite analizar la escalabilidad de los algoritmos y cómo evoluciona su tiempo de ejecución.
- **Enfoque en el Tiempo de Ejecución:** Para cada algoritmo y tamaño de problema, la métrica principal registrada fue el tiempo de ejecución en milisegundos (ms).

La elección de datos generados permite un control preciso sobre las características de las instancias de prueba y facilita un análisis comparativo homogéneo del comportamiento de los algoritmos.

Mediciones Realizadas:

En cada experimento, se midió principalmente:

- **Tiempo de ejecución en milisegundos (ms):** Tiempo total que tarda cada algoritmo en encontrar la solución para una instancia de un tamaño determinado.

Dado que los algoritmos evaluados (Backtracking y Branch and Bound) son métodos exactos, se asume que, el análisis se centra en la eficiencia temporal.

5.2 Toma de medidas y análisis

Medidas:

Los datos de rendimiento, consistentes en el algoritmo ejecutado, el tamaño de la instancia y el tiempo de ejecución resultante en milisegundos, fueron extraídos en formato csv por un método implementado. Cada combinación de algoritmo y tamaño de entrada fue ejecutada, y su tiempo de ejecución correspondiente fue registrado para el análisis.

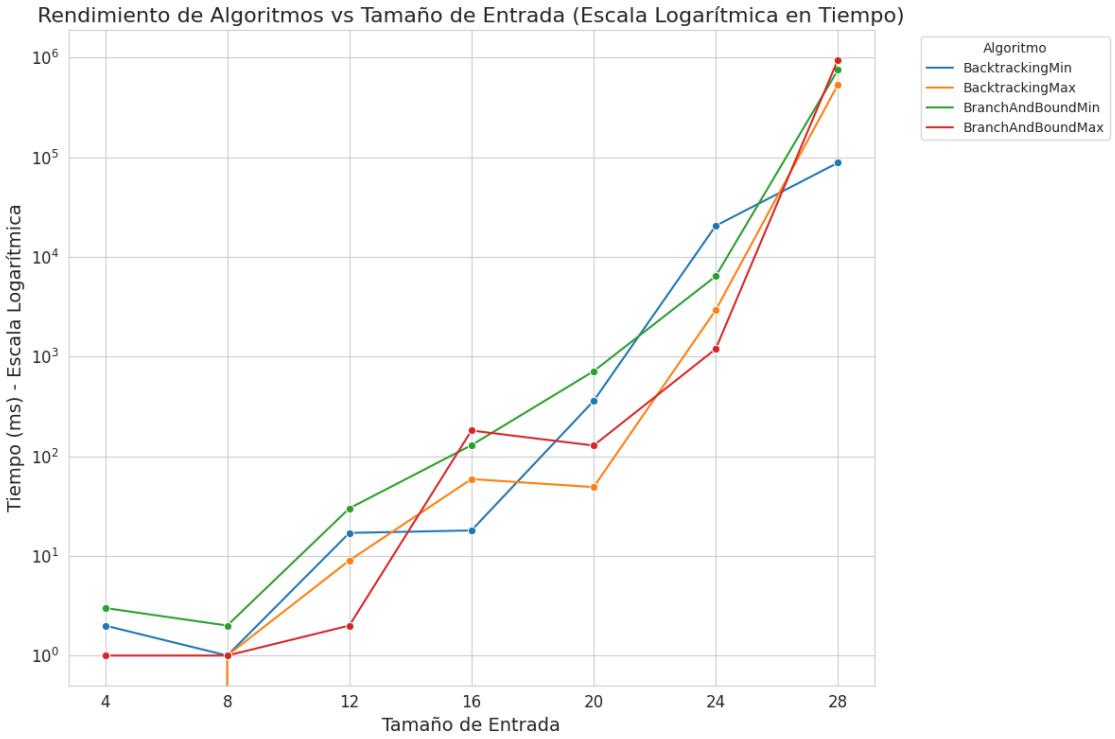
Procesado de Datos:

Para el procesado y análisis de los datos recolectados, se han utilizado la biblioteca **Python** con pandas para la manipulación de datos, y matplotlib. Este enfoque permite:

- **Generación de Gráficas Comparativas:** Se han creado diversas gráficas que representan la relación entre el tamaño del problema (eje X) y el tiempo medio de ejecución (eje Y). Estas visualizaciones son cruciales para identificar tendencias, comparar el rendimiento relativo de los algoritmos y observar su escalabilidad. Se incluyen gráficas con escalas lineales y logarítmicas para apreciar mejor las diferencias, especialmente cuando los tiempos crecen exponencialmente.
- **Análisis Visual de Tendencias:** Las gráficas permiten una interpretación visual directa de cómo el costo computacional de cada algoritmo se ve afectado por el incremento en el tamaño de la entrada.

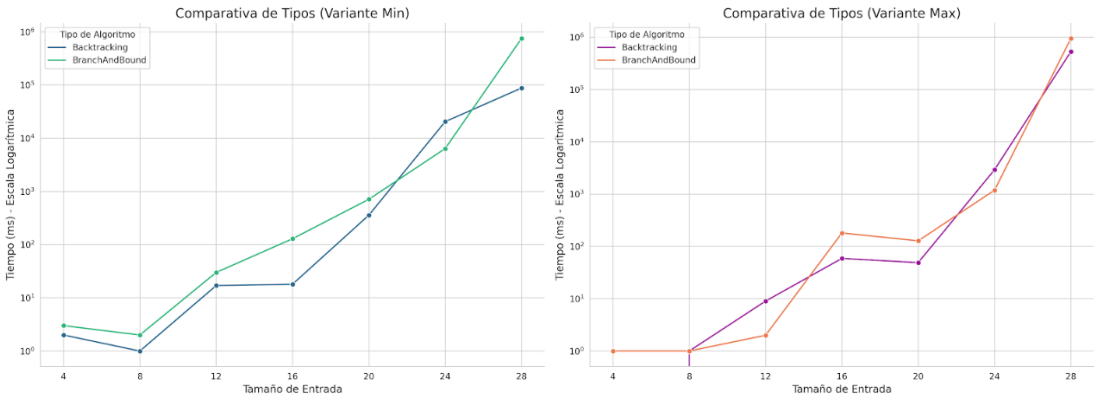
5.3 Resultados de la ejecución e interpretación

Los tiempos de ejecución demuestran un **crecimiento exponencial** del coste para todos los algoritmos al aumentar el tamaño del problema. Esto es especialmente evidente a partir de 16-20 unidades, donde la escala logarítmica se vuelve esencial para la visualización. Para Tamaño = 28, los tiempos son prohibitivos, confirmando la explosión combinatoria.



Comparativas Clave:

- **Análisis por Variante (Min/Max):**
    - Las variantes de **minimización** (BacktrackingMin, BranchAndBoundMin) muestran un crecimiento exponencial claro, como se observa en la Gráfica



- Similarmente, las variantes de **maximización** (BacktrackingMax, BranchAndBoundMax) también crecen exponencialmente
- Generalmente, las variantes Max tienden a ser más rápidas que sus contrapartes Min, aunque con variaciones en tamaños grandes.

- **Backtracking vs. Branch & Bound**
  - Para **minimización**, BacktrackingMin suele superar a BranchAndBoundMin conforme crece el tamaño.
  - Para **maximización**, BacktrackingMax es, en general, más eficiente que BranchAndBoundMax.

Ajuste a Modelos Teóricos (R^2):

Los resultados experimentales son consistentes con las complejidades teóricas exponenciales esperadas para algoritmos exactos. El coeficiente de determinación (R^2) permitiría cuantificar qué tan bien se ajustan los datos a un modelo teórico específico. El cálculo de R^2 para modelos específicos en la tendencia observada sugiere una fuerte correlación con curvas de crecimiento no polinomiales.

Implicaciones:

La eficacia de las estrategias de poda y exploración es sensible a la estructura del problema y al tamaño de la instancia. Los algoritmos exactos son viables para tamaños pequeños, pero su utilidad práctica disminuye drásticamente para instancias grandes debido al alto coste computacional.

5.4 Conclusiones del estudio experimental

El estudio experimental realizado sobre los algoritmos Backtracking (Min y Max) y Branch and Bound (Min y Max) ha permitido extraer las siguientes conclusiones clave respecto a su rendimiento en función del tamaño del problema:

1. **Rendimiento en Tamaños Pequeños:** Para instancias de problema pequeñas (Tamaño <= 12), todos los algoritmos evaluados son eficientes, con tiempos de ejecución generalmente inferiores a unas pocas decenas de milisegundos, haciéndolos viables para este rango.
2. **Impacto del Crecimiento del Problema:** Conforme aumenta el tamaño de la entrada, especialmente a partir de Tamaño = 16 y de forma más acusada para Tamaño >= 20, el tiempo de ejecución de todos los algoritmos crece de manera exponencial. Esto confirma la naturaleza NP-difícil de los problemas subyacentes que estos algoritmos suelen abordar.
3. **Comparativa entre Backtracking y Branch and Bound:**
   - Para problemas de **minimizacion**, BacktrackingMin demostró ser, en general, más rápido que BranchAndBoundMin, especialmente en tamaños de problema medianos y grandes.

- Para problemas de **maximización**, BacktrackingMax tendió a superar a BranchAndBoundMax en eficiencia para la mayoría de los tamaños, aunque con algunas fluctuaciones en los tamaños más grandes donde BranchAndBoundMax mostró competitividad o incluso superioridad en casos puntuales.

4. **Influencia de la Variante Min/Max:** Las variantes destinadas a la maximización (BacktrackingMax, BranchAndBoundMax) mostraron a menudo tiempos de ejecución inferiores a sus contrapartes de minimización. Esto sugiere que, para el conjunto de datos y la estructura del problema evaluados, las condiciones de poda o los criterios de búsqueda pueden ser más efectivos o alcanzarse antes en los escenarios de maximización.
5. **Limitaciones Prácticas:** Para los tamaños de problema más grandes evaluados (Tamaño = 24 y Tamaño = 28), los tiempos de ejecución se vuelven significativos (desde varios segundos hasta más de quince minutos), lo que puede limitar la aplicabilidad de estos algoritmos exactos en entornos donde el tiempo de respuesta es crítico, a menos que se disponga de optimizaciones muy específicas o un poder computacional considerable.

6. ANEXOS

6.1 Pseudocódigo de los algoritmos

A continuación, se presentan los pseudocódigos correspondientes a los algoritmos implementados para la resolución del problema de asignación de proyectos a directores en EDAPital. Estos algoritmos representan las distintas estrategias de solución desarrolladas, empleando las técnicas de **Backtracking**, o Vuelta Atrás, y **Branch-and-Bound**, o Ramificación y Poda.

Esta recopilación detalla el funcionamiento esquemático de cada uno de los cuatro enfoques, dos para minimización de tiempos y dos para maximización de eficacias, reflejando cómo estas técnicas abordan la búsqueda de soluciones óptimas en problemas de optimización combinatoria como el de asignación.

BacktrackingMin

Algorithm 1 Backtracking para Minimización (Muy Resumido)

Entrada: Matriz de tiempos T de $n \times n$
Salida: Solución óptima sol_{opt}

```
1: nivel ← 0
2: sol_parcial ← array de n elementos inicializado a −1
3: coste_actual ← 0
4: sol_opt ← nueva Solucion()
5: sol_opt.valor ← ∞
6: proyectos_asignados ← array de n elementos inicializado a 0
7: estim_coste_restante ← CalcularEstimacion(T, true)
8: do
9:     Generar(T, sol_opt)                          ▷ Llama a tu función 'generar'
10:    if SolucionEncontrada() y coste_actual < sol_opt.valor then
11:        sol_opt.valor ← coste_actual
12:        sol_opt.asignacion ← copia de sol_parcial
13:    if nivel < longitud(sol_parcial) − 1 y CriterioCumplido() y
       coste_actual + estim_coste_restante[nivel] < sol_opt.valor then
14:        nivel ← nivel + 1
15:    else
16:        while nivel > −1 y no MasHermanosDisponibles() do
17:            Retroceder(T)                        ▷ Llama a tu función 'retroceder'
18: while nivel > −1
19: return sol_opt
```

BacktrackingMax

Algorithm 1 Backtracking para Maximización

Entrada: Matriz de eficacias E de $n \times n$
Salida: Solución óptima sol_{opt}

```
1: nivel ← 0
2: sol_parcial ← array de n elementos inicializado a −1
3: eficacia_actual ← 0
4: sol_opt ← nueva Solucion()
5: sol_opt.valor ← −∞
6: proyectos_asignados ← array de n elementos inicializado a 0
7: estim_eficacia_restante ← CalcularEstimacion(E, false)
8: do
9:     Generar(E, sol_opt)                          ▷ Llama a tu función 'generar'
10:    if SolucionEncontrada() y eficacia_actual > sol_opt.valor then
11:        sol_opt.valor ← eficacia_actual
12:        sol_opt.asignacion ← copia de sol_parcial
13:    if nivel < longitud(sol_parcial) − 1 y CriterioCumplido() y
   eficacia_actual + estim_eficacia_restante[nivel] > sol_opt.valor then
14:        nivel ← nivel + 1
15:    else
16:        while nivel > −1 y no MasHermanosDisponibles() do
17:            Retroceder(E)                        ▷ Llama a tu función 'retroceder'
18: while nivel > −1
19: return sol_opt
```

BranchAndBoundMin

Algorithm 1 Branch and Bound para Minimización

Entrada: Matriz de tiempos T de $n \times n$
Salida: Solución óptima sol_{opt}

```
1: n ← longitud de T
2: nivel_actual ← 0
3: sol_parcial_inicial ← array de n elementos
4: coste_inicial ← 0
5: for i ← 0 to n − 1 do
6:     sol_parcial_inicial[i] ← i
7:     coste_inicial ← coste_inicial + T[i][i]
8: sol_opt ← nueva Solucion()
9: sol_opt.valor ← coste_inicial
10: sol_opt.asignacion ← sol_parcial_inicial
11: coste_actual_nodo ← 0
12: estim_coste_restante ← CalcularEstimacion(T, true)
13: LNV ← nueva ColaDePrioridad de Nodos              ▷ Cola de mínimos
14: LNV.añadir(nuevo Nodo(nivel_actual, array vacío o inicial de n, coste_actual_nodo
15: while LNV no esté vacía do
16:     nodo_actual ← LNV.extraerMin()
17:     nivel_nodo ← nodo_actual.nivel
18:     sol_parcial_nodo ← nodo_actual.sol_parcial
19:     coste_nodo ← nodo_actual.coste
20:     if nivel_nodo = n then
21:         if coste_nodo < sol_opt.valor then
22:             sol_opt.valor ← coste_nodo
23:             sol_opt.asignacion ← sol_parcial_nodo
24:     else
25:         copia_sol_parcial ← clonar(sol_parcial_nodo)
26:         for i ← nivel_nodo to n − 1 do            ▷ Generar hijos/permutaciones
27:             Intercambiar(copia_sol_parcial, i, nivel_nodo)   nuevo_coste
   ← coste_nodo + T[nivel_nodo][copia_sol_parcial[nivel_nodo]]
28:             nueva_estimacion_total ← nuevo_coste +
   estim_coste_restante[nivel_nodo]   ▷ O estim_coste_restante[nivel_nodo +
   1]siasíecalcula   if nueva_estimacion_total ¡ sol_opt.valor then
30:                 LNV.añadir(nuevo Nodo(nivel_nodo +
   1, copia de copia_sol_parcial, nuevo_coste, nueva_estimacion_total))
32:
33:             Intercambiar(copia_sol_parcial, i, nivel_nodo)   ▷ Restaurar para
   siguiente iteración si es necesario
34: return sol_opt
```

BranchAndBoundMax

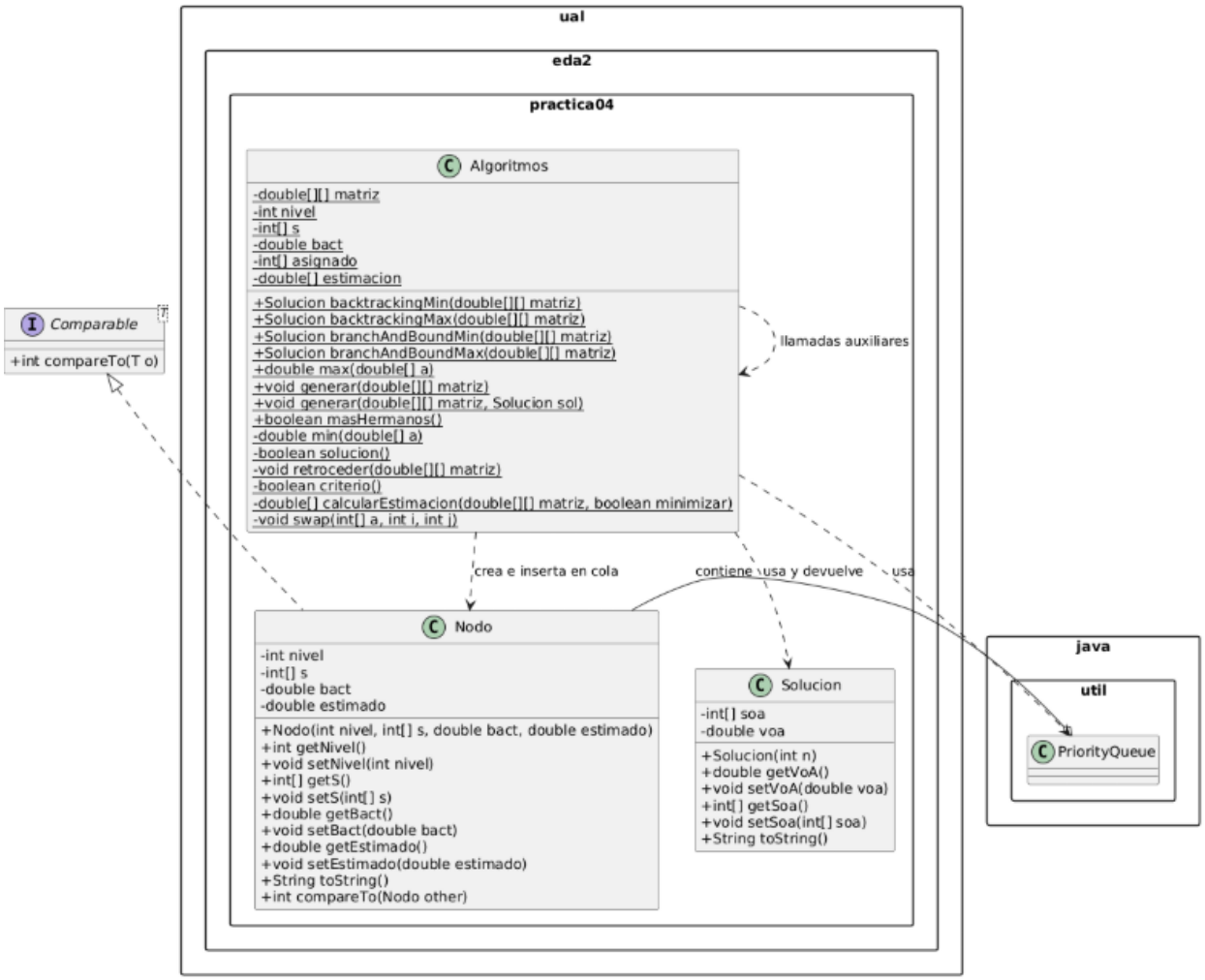
Algorithm 1 Branch and Bound para Maximización (Muy Resumido)

Entrada: Matriz de eficacias E de $n \times n$
Salida: Solución óptima sol_{opt}

```
n ← longitud de E
nivel_actual ← 0
sol_parcial_inicial ← array de n elementos
eficacia_inicial ← 0
for i ← 0 to n − 1 do
    sol_parcial_inicial[i] ← i
    eficacia_inicial ← eficacia_inicial + E[i][i]
sol_opt ← nueva Solucion()
sol_opt.valor ← eficacia_inicial
sol_opt.asignacion ← sol_parcial_inicial
eficacia_actual_nodo ← 0
estim_eficacia_restante ← CalcularEstimacion(E, false)
LNV ← nueva ColaDePrioridad de Nodos                ▷ Cola de máximos
LNV.añadir(nuevo Nodo(nivel_actual, array vacío o inicial de n, eficacia_actual_nodo
while LNV no esté vacía do
    nodo_actual ← LNV.extraerMax()  ▷ O extraerMin y negar prioridades
    nivel_nodo ← nodo_actual.nivel
    sol_parcial_nodo ← nodo_actual.sol_parcial
    eficacia_nodo ← nodo_actual.eficacia
    if nivel_nodo = n then
        if eficacia_nodo > sol_opt.valor then
            sol_opt.valor ← eficacia_nodo
            sol_opt.asignacion ← sol_parcial_nodo
    else
        copia_sol_parcial ← clonar(sol_parcial_nodo)
        for i ← nivel_nodo to n − 1 do          ▷ Generar hijos/permutaciones
            Intercambiar(copia_sol_parcial, i, nivel_nodo)          nueva_eficacia
← eficacia_nodo + E[nivel_nodo][copia_sol_parcial[nivel_nodo]]
            nueva_estimacion_total ← nueva_eficacia +
estim_eficacia_restante[nivel_nodo]                                  ▷
O estim_eficacia_restante[nivel_nodo + 1]          if nueva_estimacion_total
¿ sol_opt.valor then
            LNV.añadir(nuevo Nodo(nivel_nodo                          +
1, copia de copia_sol_parcial, nueva_eficacia, nueva_estimacion_total))

            Intercambiar(copia_sol_parcial, i, nivel_nodo)          ▷ Restaurar
return sol_opt
```

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/