ULPGC
Escuela de Ingeniería Informática

ESTRATEGIAS DE BÚSQUEDA

FUNDAMENTOS DE LOS SISTEMAS INTELIGENTES

ALBA RAMOS QUINTANA

2025/2026

ÍNDICE

1. **PARTE 1** - RAMIFICACIÓN Y ACOTACIÓN

1.1. ESTRATEGIA IMPLEMENTADA

La estrategia de Ramificación y Acotación consiste en expandir siempre el nodo con menor coste acumulado, que en este caso, como es un mapa, se refiere a la distancia de una ciudad a otra, garantizando que la primera solución encontrada sea óptima.

Para su implementación:

- Se ha reutilizado la función general *graph_search*, manteniendo intacta su estructura.
- Se ha definido una nueva estructura de datos *PriorityQueue*, que ordena los nodos de menor a mayor según su atributo *path_cost*. Para ello se ha creado la clase *PriorityQueue*, que funciona como una cola de prioridad utilizando el módulo heapq de Python.

Internamente, la estructura almacena los nodos en una lista que se mantiene ordenada automáticamente según el coste del camino recorrido hasta cada nodo. De esta forma, en cada iteración de la búsqueda se extrae siempre el nodo más prometedor desde el punto de vista del coste acumulado.

Cada elemento insertado en la cola se almacena como una tupla de la forma (coste, contador, nodo) , donde:

- **coste** corresponde al valor *path_cost* del nodo, que es el criterio principal de ordenación.
- **contador** es un identificador incremental que se utiliza para desempatar cuando dos nodos tienen el mismo coste, garantizando un comportamiento determinista del algoritmo.
- **nodo** es la instancia de la clase Node que representa el estado del problema.

La clase proporciona los métodos básicos necesarios para integrarse con el código base:

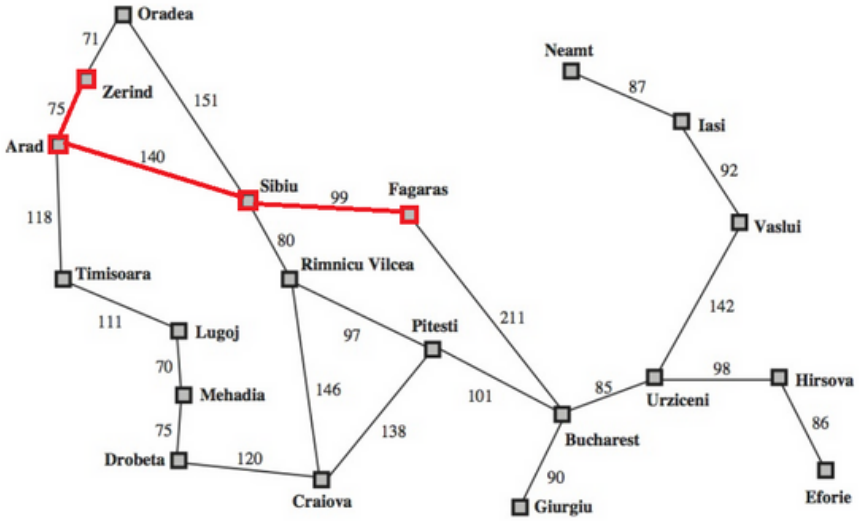
- **append(node):** inserta un nodo en la cola respetando el orden por coste.
- **pop():** extrae el nodo con menor coste acumulado.
- **extend(nodes):** permite insertar múltiples nodos.
- **__len__():** devuelve el número de elementos almacenados en la cola.

Esta cola de prioridad sustituye al FIFOQueue y al Stack utilizados en BFS y DFS.

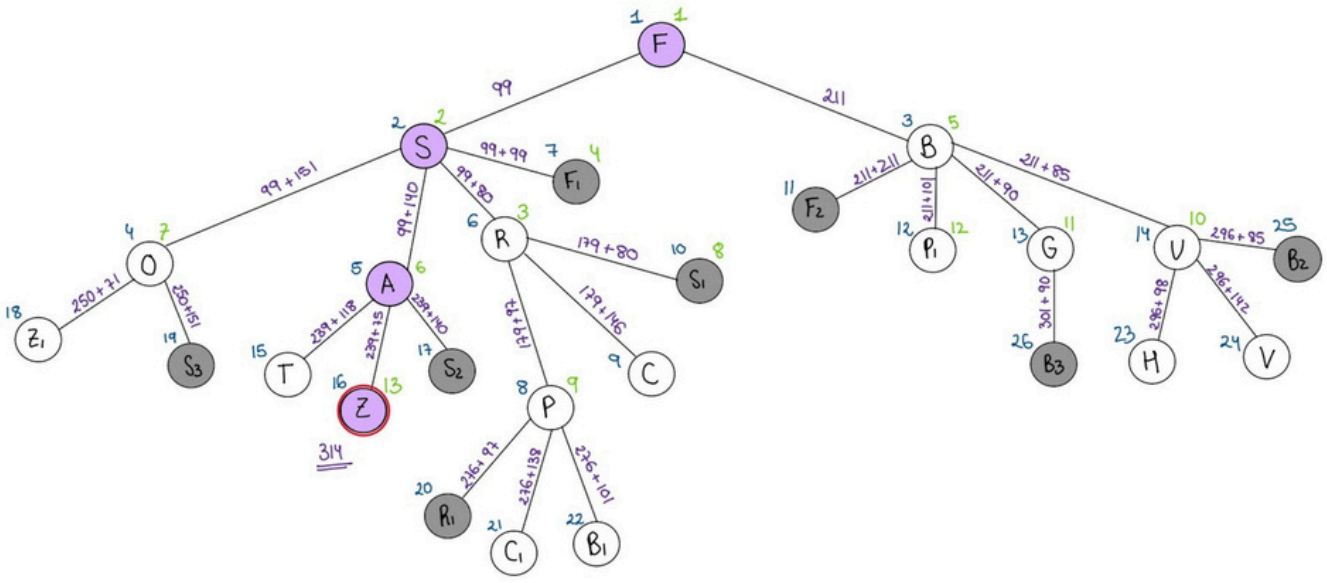
1.2. TRAZA MANUAL DE UNA BÚSQUEDA EN EL GRAFO DE RUMANÍA

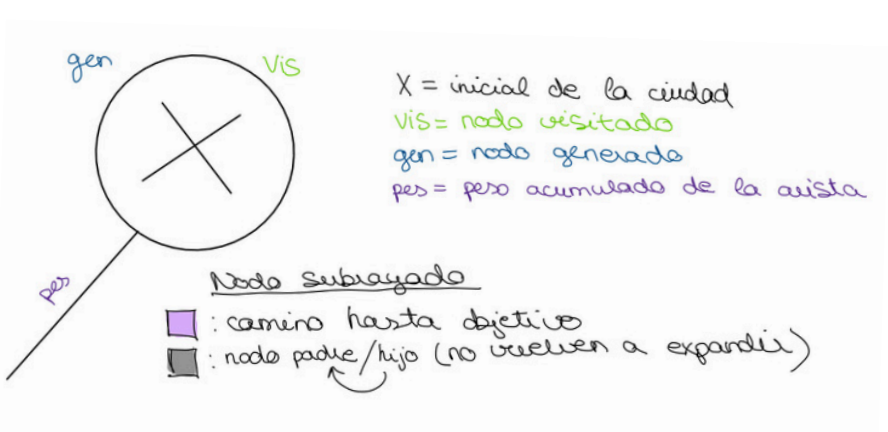
He elegido una ruta diferente que no esté en la tabla, de Fagaras(F) a Zerind(Z) para así verificar el correcto funcionamiento de los algoritmos implementados.
El resultado esperado y **obtenido** es el siguiente:

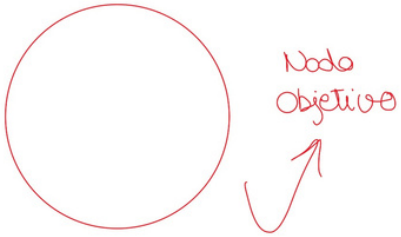
Ruta: [<Node Z>, <Node A>, <Node S>, <Node F>]
Nodos generados: 26
Nodos visitados: 13
Peso: 314



A continuación, se muestra la representación del grafo manual correspondiente, que permite visualizar los nodos y las conexiones de la ruta elegida. Posteriormente, se presenta la traza completa de la búsqueda, la cual permite comprobar paso a paso cómo se han generado y visitado los nodos hasta alcanzar el objetivo.







NODO ACTUAL	LISTA DE NODOS - ABIERTA
	[F]
F	[S/99, B/211]
S	[R/99+80, F1/99+99, B/211, A/99+140, O/99+151]
R	[F1/198, B/211, A/239, O/250, S1/179+80, P/179+97, C/179+146]
F1 (no expande)	[B/211, A/239, O/250, S1/259, P/276, C/325]
B	[A/239, O/250, S1/259, P/276, U/211+85, G/211+90, P1/211+101, C/325, F2/211+211]
A	[O/250, S1/259, P/276, U/296, G/301, P1/312, Z/239+75, C/325, T/239+118, S2/239+140, F2/422]
O	[S1/259, P/276, U/296, G/301, P1/312, Z/314, Z1/250+71, C/325, T/357, S2/379, S3/250+151, F2/422]
S1 (no expande)	[P/276, U/296, G/301, P1/312, Z/314, Z1/321, C/325, T/357, S2/379, S3/401, F2/422]
P	[U/296, G/301, P1/312, Z/314, Z1/321, C/325, T/357, R1/276+97, B1/276+101, S2/379, S3/401, C1/276+138, F2/422]
U	[G/301, P1/312, Z/314, Z1/321, C/325, T/357, R1/373, B1/377, S2/379, B2/296+85, H/296+98, S3/401, C1/414, F2/422, V/296+142]
G	[P1/312, Z/314, Z1/321, C/325, T/357, R1/373, B1/377, S2/379, B2/381, B3/301+90, H/394, S3/401, C1/414, F2/422, V/438]
P1 (no expande)	[Z/314, Z1/321, C/325, T/357, R1/373, B1/377, S2/379, B2/381, B3/391, H/394, S3/401, C1/414, F2/422, V/438]
Z	Objetivo

El desarrollo mostrado encuentra la ruta óptima. Las métricas resultantes coinciden con la ejecución teórica: 13 nodos visitados y 26 nodos generados.

Al utilizar una lista cerrada, antes de expandir un nodo, verifica si ese estado ya ha sido visitado previamente con un coste menor o igual.

- **Caso de padres/ciclos:** Cuando por ejemplo, S genera a F1 (su padre), el algoritmo detecta que F ya fue visitado, por lo que se visita pero no expande de nuevo.

- **Caso específico de P1:** El nodo P se visitó primero a través de la ruta R y se añadió a la lista cerrada. Así que cuando extrae P1, detecta que P ya está en la lista cerrada, por lo lo que se visita pero no se expande.

2. **PARTE 2** - RAMIFICACIÓN Y ACOTACIÓN CON SUBESTIMACIÓN

2.1. ESTRATEGIA IMPLEMENTADA

En esta parte se ha implementado Ramificación y Acotación con Subestimación, donde aparte de sumarse el coste acumulado por cada ruta, se le suma una heurística, que es una estimación del coste restante hasta el objetivo. **[f(n) = g(n) + h(n)]** , donde:

- g(n) es el coste acumulado (*path_cost*)
- h(n) es la heurística de distancia euclídea hasta el objetivo

El código base ya proporcionaba:

- El cálculo de la heurística mediante *GPSProblem.h(node)*
- Las coordenadas de las ciudades en el grafo de Rumanía

Por tanto, la implementación se ha limitado a crear una nueva cola de prioridad:

- *PriorityQueueH*, que ordena los nodos según el valor f(n)

La estructura se basa, al igual que en el caso anterior, implementado mediante el módulo heapq de Python. Internamente, cada nodo se almacena como una tupla de la forma: (f(n),contador,nodo), donde:

- **f(n)** se calcula en el momento de insertar el nodo en la cola, combinando el coste real acumulado y la heurística.
- **contador** se utiliza nuevamente para evitar problemas de desempate cuando dos nodos tienen el mismo valor de prioridad, garantizando un orden consistente de extracción.

La clase recibe como parámetro el objeto *problem*, lo que permite acceder directamente a la función heurística h(n) definida en el código base, sin necesidad de modificar la clase Node ni el resto de la estructura del programa.

También tiene nuevamente sus métodos definidos **(append, pop, extend y __len__)**

Esta estrategia reduce significativamente el número de nodos explorados manteniendo la optimalidad de la solución.

2.2. CASO EN EL QUE LA HEURÍSTICA QUE SOBREESTIMA NO GARANTIZA EL CAMINO ÓPTIMO

Para que una heurística garantice la obtención del camino óptimo debe cumplir la condición de consistencia, según la cual, para todo nodo n y todo nodo hijo n' alcanzado mediante una acción a, el valor heurístico debe satisfacer: $h(n) \leq c(n,a,n') + h(n')$

Esta propiedad se cumple, por ejemplo, cuando se utiliza la distancia euclídea, ya que respeta la desigualdad triangular y no sobreestima el coste real hasta el objetivo.

Al modificar la heurística multiplicando la distancia euclídea por un factor mayor que uno en este caso, por **4**, la función heurística pasa a sobreestimar el coste real. Como consecuencia, deja de ser admisible y consistente.

Ya no garantiza la optimalidad de la solución, obteniendo en algunos casos rutas con un coste total mayor, aunque con un menor número de nodos visitados.

En comparación con la tabla original, se observa claramente que en la columna correspondiente a Ramificación y Acotación con heurística se producen diferencias tanto en el peso acumulado como en la ruta obtenida, lo que confirma que el algoritmo no siempre encuentra el camino óptimo.

ID	Origen	Destino	Ramificación y Acotación con sobreestimación
1	Arad	Bucharest	**Generados**: 10 **Visitados**: 4 **Costo total**: 450 **Ruta**: [<Node B>, <Node F>, <Node S>, <Node A>] **Tiempo de ejecución**: 59.8 µs
2	Oradea	Eforie	**Generados**: 18 **Visitados**: 7 **Costo total**: 730 **Ruta**: [<Node E>, <Node H>, <Node U>, <Node B>, <Node F>, <Node S>, <Node O>] **Tiempo de ejecución**: 56.8 µs
3	Giurgiu	Zerind	**Generados**: 15 **Visitados**: 6 **Costo total**: 615 **Ruta**: [<Node Z>, <Node A>, <Node S>, <Node F>, <Node B>, <Node G>] **Tiempo de ejecución**: 48.7 µs
4	Neamt	Dobreta	**Generados**: 19 **Visitados**: 9 **Costo total**: 765 **Ruta**: <Node D>, <Node C>, <Node P>, <Node B>, <Node U>, <Node V>, <Node I>, <Node N>] **Tiempo de ejecución**: 63.5 µs
5	Mehadia	Fagaras	**Generados**: 20 **Visitados**: 12 **Costo total**: 520 **Ruta**: [<Node F>, <Node S>, <Node R>, <Node C>, <Node D>, <Node M>] **Tiempo de ejecución**: 53.7 µs

3. **PARTE 3** - CONTABILIDAD DE NODOS Y TIEMPO DE EJECUCIÓN

➢NODOS GENERADOS

El número de nodos generados se define como el total de nodos creados durante la ejecución del algoritmo de búsqueda, incluyendo el nodo raíz.

En la implementación, el contador *generated* se inicializa a cero y se incrementa en dos situaciones:

- Al insertar el nodo inicial en la frontera, ya que el nodo raíz también se considera generado.
- Cada vez que se expanden los sucesores de un nodo mediante el método *expand*, sumando el número total de nodos hijos creados en dicha expansión.

```
generated += len(succ)
```

➢NODOS VISITADOS

El número de nodos visitados se define como el número total de nodos a los que se les ha comprobado si son objetivo.

En el código, el contador *visited* se incrementa justo después de extraer un nodo de la frontera (*opened.pop()*), y antes de realizar la comprobación del objetivo. Esto garantiza que cada nodo visitado contabiliza exactamente una vez, independientemente de si se expande o no posteriormente.

➢TIEMPO DE EJECUCIÓN

El tiempo de ejecución se ha medido utilizando *time.perf_counter()*:

- Se registra el instante inicial al comenzar la búsqueda.
- Se calcula la diferencia al encontrar la solución o finalizar la búsqueda.

El tiempo se presenta en microsegundos (µs) para facilitar la comparación entre algoritmos.

3.1. CAMBIOS REALIZADOS RESPECTO AL CÓDIGO BASE

Se ha sustituido el nombre de la estructura *fringe* por *opened*, ya que resulta más intuitivo para representar el conjunto de nodos pendientes de explorar.

La estructura *closed*, que originalmente era un diccionario, se ha implementado como una lista. Esta elección simplifica la gestión de los estados visitados y personalmente, me resulta más sencillo para trabajarlo.

La función *graph_search* ahora devuelve, además del nodo solución, toda la información relevante para el análisis experimental (nodos generados, visitados, coste total y tiempo de ejecución)

4. VISUALIZACIÓN DEL GRAFO, HEURÍSTICAS Y EJECUCIÓN DE PRUEBAS

Se han desarrollado clases auxiliares para facilitar la comprensión del problema:

➢GRAPH.PY

Representación visual del grafo de Rumanía usando *networkx* y *matplotlib*.

El grafo se construye a partir de la estructura definida en el código base y se representa empleando las coordenadas reales de cada ciudad, lo que permite obtener una visualización fiel al mapa original del problema, donde se muestran los nodos aristas y pesos

➢GRAPHH.PY

Este fichero amplía la visualización anterior incorporando el valor de la heurística asociada a cada nodo.

Para ello, se define una ruta concreta especificando un nodo origen y un nodo destino. A partir de esta información, se calcula la heurística de cada ciudad como la distancia euclídea hasta el nodo objetivo, mostrando dicho valor junto a cada nodo del grafo.

➢TABLE.PY Y RUN.PY

El fichero *run.py* se utiliza para la ejecución de una **única** ruta concreta entre un origen y un destino determinados, permitiendo comprobar de forma directa el funcionamiento de todos los algoritmos implementados sobre un caso específico.

Sin embargo, con el objetivo de completar la tabla comparativa solicitada en la práctica de una manera más cómoda y sistemática, se ha desarrollado el fichero *table.py*. Este script automatiza la ejecución de todos los algoritmos de búsqueda sobre las rutas origen–destino que solicitan, evitando tener que modificar manualmente los valores de origen y destino en cada ejecución.

Recopilan la siguiente información (ruta solución encontrada, nº nodos generados, visitados, coste total de la solución y tiempo de ejecución).

5. TABLA COMPARATIVA Y COMPARACIÓN DE RESULTADOS/ALGORITMOS

ID	Origen	Destino	Amplitud (BFS)	Profundidad (DFS)	Ramificación y Acotación (B&B)	Ramificación y Acotación con Subestimación
1	**Arad**	**Bucharest**	**Generados**: 21 **Visitados**: 16 **Costo total**: 450 **Ruta**: [<Node B>, <Node F>, <Node S>, <Node A>] **Tiempo de ejecución**: 92.7 µs	**Generados**: 18 **Visitados**: 10 **Costo total**: 733 **Ruta**: [<Node B>, <Node P>, <Node C>, <Node D>, <Node M>, <Node L>, <Node T>, <Node A>] **Tiempo de ejecución**: 142.0 µs	**Generados**: 31 **Visitados**: 24 **Costo total**: 418 **Ruta**: [<Node B>, <Node P>, <Node R>, <Node S>, <Node A>] **Tiempo de ejecución**: 111.7 µs	**Generados**: 16 **Visitados**: 6 **Costo total**: 418 **Ruta**: [<Node B>, <Node P>, <Node R>, <Node S>, <Node A>] **Tiempo de ejecución**: 98.9 µs
2	**Oradea**	**Eforie**	**Generados**: 45 **Visitados**: 43 **Costo total**: 730 **Ruta**: [<Node E>, <Node H>, <Node U>, <Node B>, <Node F>, <Node S>, <Node O>] **Tiempo de ejecución**: 103.0 µs	**Generados**: 41 **Visitados**: 31 **Costo total**: 698 **Ruta**: [<Node E>, <Node H>, <Node U>, <Node B>, <Node P>, <Node R>, <Node S>, <Node O>] **Tiempo de ejecución**: 120.6 µs	**Generados**: 43 **Visitados**: 40 **Costo total**: 698 **Ruta**: [<Node E>, <Node H>, <Node U>, <Node B>, <Node P>, <Node R>, <Node S>, <Node O>] **Tiempo de ejecución**: 155.6 µs	**Generados**: 32 **Visitados**: 15 **Costo total**: 698 **Ruta**: [<Node E>, <Node H>, <Node U>, <Node B>, <Node P>, <Node R>, <Node S>, <Node O>] **Tiempo de ejecución**: 139.3 µs
3	**Giurgiu**	**Zerind**	**Generados**: 41 **Visitados**: 34 **Costo total**: 615 **Ruta**: [<Node Z>, <Node A>, <Node S>, <Node F>, <Node B>, <Node G>] **Tiempo de ejecución**: 153.3 µs	**Generados**: 32 **Visitados**: 21 **Costo total**: 1284 **Ruta**: [<Node Z>, <Node A>, <Node T>, <Node L>, <Node M>, <Node D>, <Node C>, <Node P>, <Node R>, <Node S>, <Node F>, <Node B>, <Node G>] **Tiempo de ejecución**: 93.9 µs	**Generados**: 41 **Visitados**: 35 **Costo total**: 583 **Ruta**: [<Node Z>, <Node A>, <Node S>, <Node R>, <Node P>, <Node B>, <Node G>] **Tiempo de ejecución**: 222.3 µs	**Generados**: 26 **Visitados**: 12 **Costo total**: 583 **Ruta**: [<Node Z>, <Node A>, <Node S>, <Node R>, <Node P>, <Node B>, <Node G>] **Tiempo de ejecución**: 119.0 µs
4	**Neamt**	**Dobreta**	**Generados**: 32 **Visitados**: 26 **Costo total**: 765 **Ruta**: [<Node D>, <Node C>, <Node P>, <Node B>, <Node U>, <Node V>, <Node I>, <Node N>] **Tiempo de ejecución**: 140.3 µs	**Generados**: 31 **Visitados**: 19 **Costo total**: 1151 **Ruta**: [<Node D>, <Node C>, <Node P>, <Node R>, <Node S>, <Node F>, <Node B>, <Node U>, <Node V>, <Node I>, <Node N>] **Tiempo de ejecución**: 89.8 µs	**Generados**: 32 **Visitados**: 26 **Costo total**: 765 **Ruta**: [<Node D>, <Node C>, <Node P>, <Node B>, <Node U>, <Node V>, <Node I>, <Node N>] **Tiempo de ejecución**: 108.4 µs	**Generados**: 23 **Visitados**: 12 **Costo total**: 765 **Ruta**: [<Node D>, <Node C>, <Node P>, <Node B>, <Node U>, <Node V>, <Node I>, <Node N>] **Tiempo de ejecución**: 103.0 µs
5	**Mehadia**	**Fagaras**	**Generados**: 31 **Visitados**: 23 **Costo total**: 520 **Ruta**: [<Node F>, <Node S>, <Node R>, <Node C>, <Node D>, <Node M>] **Tiempo de ejecución**: 252.4 µs	**Generados**: 29 **Visitados**: 18 **Costo total**: 928 **Ruta**: [<Node F>, <Node B>, <Node P>, <Node R>, <Node S>, <Node A>, <Node T>, <Node L>, <Node M>] **Tiempo de ejecución**: 89.5 µs	**Generados**: 36 **Visitados**: 27 **Costo total**: 520 **Ruta**: [<Node F>, <Node S>, <Node R>, <Node C>, <Node D>, <Node M>] **Tiempo de ejecución**: 82.6 µs	**Generados**: 25 **Visitados**: 16 **Costo total**: 520 **Ruta**: [<Node F>, <Node S>, <Node R>, <Node C>, <Node D>, <Node M>] **Tiempo de ejecución**: 144.4 µs

La búsqueda en amplitud (***BFS***) siempre encuentra una solución, pero al no tener en cuenta el coste de los caminos no garantiza siempre que sea la óptima. Esto se observa en los resultados, donde los costes obtenidos suelen ser mayores y el número de nodos generados y visitados es elevado.

La búsqueda en profundidad (***DFS***) es la que peores resultados ofrece en cuanto a calidad de la solución. Aunque en algunos casos genera menos nodos, los caminos encontrados tienen costes muy altos, ya que el algoritmo profundiza sin considerar ni el coste acumulado ni la cercanía al destino.

El algoritmo de Ramificación y Acotación (***B&B***) utiliza el coste acumulado para guiar la búsqueda, lo que le permite garantizar el camino óptimo. En la tabla se aprecia que obtiene siempre los menores costes posibles, aunque necesita explorar un mayor número de nodos.

Por último, la Ramificación y Acotación con heurística emplea una estimación basada en la distancia euclídea, lo que reduce significativamente el número de nodos generados y visitados. En la mayoría de los casos mantiene el coste óptimo, aunque su comportamiento depende de la calidad de la heurística utilizada.

6. CONCLUSIONES

Se ha podido comprobar la superioridad de la estrategias de búsqueda informada frente a las de búsqueda no informada (BFS, DFS, B&B). La implementación de Ramificación y Acotación ha demostrado ser infalible para encontrar la solución óptima, guiándose por el coste acumulado, aunque esto implique generar un mayor número de nodos en comparación con búsquedas más directas.

Sin embargo, el punto más destacado ha sido el uso de **Ramificación y Acotación con Subestimación**. Al incorporar la distancia euclídea como heurística, se ha logrado reducir drásticamente el espacio de búsqueda (menos nodos generados y visitados) manteniendo la optimalidad de la ruta.

Además, con la heurística no admisible (**sobreestimación**) permitió verificar la teoría: al romper la consistencia, el algoritmo gana velocidad pero pierde la garantía de encontrar el camino más corto.

Finalmente, la automatización de pruebas mediante *table.py* y la visualización gráfica han sido fundamentales para analizar estas diferencias de rendimiento de forma clara y objetiva.

7. BIBLIOGRAFÍA

heapq
itertools
networkx
networkx2
Búsqueda Informada Búsqueda no Informada

FUNDAMENTOS DE LOS SISTEMAS INTELIGENTES

2025/2026