

Actividad 1: Búsqueda de rutas en empresa de paquetería

Realizado por:

- Araceli Ruiz Vallecillo
- Fernando Jesús Fuentes Carrasco
- Javier Jordán Luque
- Teodoro Hidalgo Guerrero

En el marco del estudio de algoritmos de búsqueda, este trabajo se enfoca en analizar y comparar distintas estrategias de búsqueda para resolver un problema práctico de navegación en un entorno simplificado. La situación modela un desafío logístico en el que una furgoneta autónoma debe recorrer una ciudad representada como una matriz rectangular para recoger paquetes distribuidos en distintos puntos.

Se proporciona el código necesario para ejecutar los algoritmos en el notebook *Cuaderno_Actividad_1_Busqueda.ipynb*, de modo que el trabajo se limita a interpretar los resultados obtenidos, identificar patrones y realizar comparaciones fundamentadas en las características teóricas de cada estrategia.

Este enfoque permitirá valorar la efectividad de los algoritmos bajo diferentes configuraciones, destacando sus ventajas y limitaciones.

Caso 1: Comparación entre Búsqueda en Amplitud y Búsqueda en Profundidad

En este primer caso, se analiza el rendimiento de los algoritmos de *Búsqueda en Amplitud* (BFS) y *Búsqueda en Profundidad* (DFS).

Datos obtenidos

Amplitud	Profundidad
Total length of solution: 9	Total length of solution: 15

Total cost of solution: 8.0 Max fringe size: 5 Visited nodes: 23 Iterations: 23	Total cost of solution: 14.0 Max fringe size: 5 Visited nodes: 15 Iterations: 15
--	---

Tabla 1: Resultados de la ejecución del caso 1

Amplitud (BFS)

El algoritmo de *Búsqueda en Amplitud* (BFS) sigue una **estructura de cola FIFO** (First in First Out), donde el primer nodo añadido a la frontera es el primero en ser explorado. El algoritmo es **completo**, ya que encontrará una solución si existe. Además, es **óptimo** sólo cuando todas las acciones tienen el mismo coste, ya que garantiza encontrar la solución con el menor número de pasos o el menor coste acumulado. En este caso concreto, dado que los costos son uniformes, el algoritmo de *Búsqueda en Amplitud* permite encontrar la solución óptima, como se puede observar en la Tabla 1.

BFS almacena todos los nodos de un nivel antes de pasar al siguiente, lo que puede hacerlo ineficiente para problemas con grafos muy grandes o de gran profundidad. La **complejidad de memoria** es $O(b^d)$ (donde b es el factor de ramificación y d es la profundidad máxima del árbol de búsqueda).

Profundidad (DFS)

El algoritmo de *Búsqueda en Profundidad* (DFS) sigue una estructura de pila LIFO (Last In First Out), donde el último nodo añadido a la frontera es el primero en ser explorado. El algoritmo es **completo** siempre y cuando se eliminen los estados duplicados en la misma rama, y la memoria no sea un factor limitante al poder explorar indefinidamente una rama si se trata de un espacio de búsqueda infinito. Aunque, en este caso concreto, el espacio de búsqueda es finito, y, por tanto, si existe una solución, así que la encontrará. En cuanto a la optimalidad, el algoritmo DFS **no es óptimo**, ya que puede encontrar una solución más costosa si esta se encuentra en un camino más profundo. En este caso concreto, como se muestra en la Tabla 1, el algoritmo de *Búsqueda en Profundidad* encuentra una solución, aunque no es la óptima.

DFS necesita almacenar sólo los nodos en el camino actual y los nodos de la frontera (los sucesores del nodo actual). La **complejidad de memoria** es $O(d)$ (donde d es la profundidad máxima del árbol de búsqueda).

Comparación

Por lo que podemos observar de ambos algoritmos, podemos definir que el algoritmo de amplitud obtiene la solución óptima a costa de una mayor cantidad de iteraciones en comparación al algoritmo en profundidad.

Dependiendo de nuestro ámbito, deberíamos escoger BFS si la optimalidad es crítica para nuestra aplicación de planificación de rutas de paquetería, o DFS en caso de que el coste computacional tenga mayor criticalidad, siendo las soluciones subóptimas encontradas aceptables.

De la misma manera, este contexto es algo idílico ya que encontramos un escenario en el que el orden de las acciones y la limitación del área hacen posible que se encuentre una solución de forma eficiente. Lo cuál no tiene porque ser correcto al escalar el aplicativo, debido a la posibilidad de encontrar ramificaciones infinitas en un mapa real.

Caso 2: Comparación entre Búsqueda en Amplitud, UCS y A*

En este segundo caso, se analizan y comparan los algoritmos de *Búsqueda en Amplitud* (BFS), *UCS* y *A**.

Datos obtenidos

Amplitud	UCS
Total length of solution: 9 Total cost of solution: 28.0 Max fringe size: 5 Visited nodes: 23 Iterations: 23	Total length of solution: 9 Total cost of solution: 22.0 Max fringe size: 6 Visited nodes: 22 Iterations: 22

A*
Total length of solution: 9 Total cost of solution: 22.0 Max fringe size: 6 Visited nodes: 20 Iterations: 20

Tabla 2: Resultados de la ejecución del caso 2

UCS

El algoritmo de *Búsqueda de Coste Uniforme* (UCS) es un **algoritmo de búsqueda no informada** utilizado para encontrar el camino de coste mínimo en un grafo. Es similar a la *Búsqueda en Amplitud*, pero en lugar de expandir nodos en función de su profundidad en el árbol, UCS los expande en función del coste acumulado desde el nodo inicial. Este algoritmo es **completo** y **óptimo** al tener todos los costes con valores enteros positivos, por lo que encontrará una solución, siempre que la haya. En este caso concreto, como se muestra en la Tabla 2, el algoritmo UCS sí encuentra la solución óptima.

UCS utiliza una cola de prioridad basada en el coste acumulado, lo que asegura que:

- Siempre se expanden los nodos con el coste más bajo primero.
- No se omite ningún nodo alcanzable, ya que todos los costes positivos garantizan que cada expansión progresa hacia una solución (o hacia la exploración completa de las posibilidades).

Por lo tanto, si, UCS garantiza encontrar el camino de coste óptimo si los costes no son negativos. Esto es porque UCS siempre explora los nodos de menor coste acumulado primero, lo que garantiza que el primer nodo objetivo que se encuentre sea el de coste mínimo. La **complejidad de memoria** es $O(b^d)$ (donde b es el número de nodos hijos y d es la profundidad máxima del árbol de búsqueda).

A*

El algoritmo A* sigue una estrategia de **búsqueda informada** que utiliza una heurística para guiar la exploración hacia el objetivo de manera eficiente. A* mantiene una estructura de prioridad, en la que los nodos con menor valor de $f(n) = g(n) + h(n)$ (donde $g(n)$ es el coste acumulado y $h(n)$ es la heurística) son los primeros en ser explorados. El algoritmo es **completo**, ya que, si existe una solución, la encontrará. Además, es **óptimo si la heurística utilizada es admisible**, lo que garantiza que A* encontrará la solución con el menor coste posible. En este caso concreto, el algoritmo de A* permite encontrar la solución óptima, como se puede observar en la Tabla 2.

A* necesita almacenar tanto los nodos de la frontera como los nodos expandidos, lo que implica un **mayor coste de memoria** en comparación con otros algoritmos de búsqueda. La **complejidad de memoria** de A* es $O(b^d)$ (donde b es el factor de ramificación y d es la profundidad máxima del árbol de búsqueda).

Debido a que A* mantiene más nodos en memoria que, por ejemplo, la búsqueda en profundidad, su consumo de memoria puede ser considerablemente mayor, especialmente en problemas con grandes espacios de búsqueda.

Comparación

A modo de comparativa, se puede decir que UCS es un mejor algoritmo que *Búsqueda en Amplitud*, ya que incorpora el concepto de costes en los caminos, priorizando la expansión

de nodos según el coste acumulado. Por su parte, A* incorpora una heurística para guiar la búsqueda, permitiendo así encontrar soluciones más rápidamente al estimar la proximidad al objetivo.

En el caso de que el mapa cambie, UCS siempre proporcionará una **solución correcta y óptima**. Sin embargo, el recorrido o la solución pueden variar dependiendo de la distribución de los obstáculos y los costes de las rutas. Por ejemplo, si se agrega un obstáculo en otro lugar, el camino elegido por UCS cambiará para seguir buscando la ruta con menor coste.

Por otro lado, A* también proporciona la **solución óptima si la heurística es admisible**. Sin embargo, la heurística juega un papel fundamental en cómo se expande la búsqueda. Su papel proviene dada la importancia de que una heurística bien informada proporcionará mayor eficiencia al realizar menos iteraciones y visitar una menor cantidad de nodos.

En el caso de que los costes varíen, si estos permanecen positivos, UCS y A* siempre buscarán el camino de menor coste acumulado. Sin embargo, si los costes se vuelven negativos, funcionarán de manera errónea, ya que se basa en los costes más bajos para expandir un nodo. Una vez visitado un nodo, si se encuentra nuevamente, podría ocurrir que el valor de $g(n)$ disminuya respecto al valor actual, lo que implicaría que el nodo se vuelva a visitar y se generen ciclos infinitos. Además, cuantos más acciones con costes negativos haya, mayor será la posibilidad de encontrar estos ciclos, lo que afecta negativamente a la propiedad de completitud del algoritmo.

En conclusión, BFS explora todos los nodos en orden de su distancia al nodo de inicio, sin tener en cuenta los costes ni usar heurísticas, lo que lo hace **menos eficiente**, especialmente en mapas con muchos obstáculos, ya que explora nodos innecesarios y consume más memoria y tiempo. Por otro lado, UCS es **más eficiente que BFS** porque considera los costes de transición entre nodos y busca el camino de menor coste, pero puede ser menos eficiente en mapas grandes con alto factor de ramificación, ya que explora exhaustivamente hasta encontrar la solución óptima. Por último, A* suele ser **el más eficiente** cuando se utiliza una heurística suficientemente bien informada. Si la heurística es precisa y bien informada, A* puede explorar menos espacio que UCS y BFS, priorizando nodos más prometedores y, por lo tanto, siendo mucho más eficiente en estos casos.

Caso 3: Comparación de Heurísticas en A* y Referencia con UCS

En este tercer caso, se evalúa el desempeño del algoritmo A* utilizando distintas funciones heurísticas, comparando los resultados obtenidos con los del algoritmo UCS, que actúa como referencia sin heurística.

Datos obtenidos

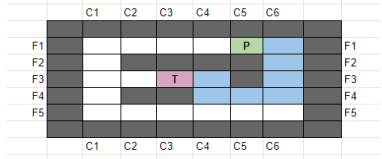
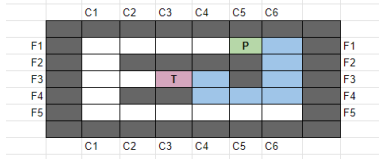
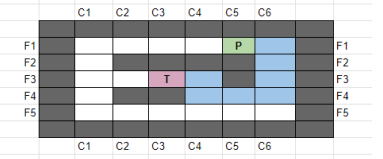
Heurística 1	Heurística 2	Heurística 3
		
Total length of solution: 9 Total cost of solution: 8.0 Max fringe size: 5 Visited nodes: 16 Iterations: 16	Total length of solution: 9 Total cost of solution: 8.0 Max fringe size: 6 Visited nodes: 17 Iterations: 17	Total length of solution: 9 Total cost of solution: 8.0 Max fringe size: 4 Visited nodes: 11 Iterations: 11

Tabla 3: Resultados de la ejecución del caso 3

UCS

Como se ha explicado anteriormente, el algoritmo UCS garantiza encontrar el camino de coste óptimo siempre que todos los costes sean positivos.

A*

En la exploración de este mapa, las tres heurísticas consiguen alcanzar la solución óptima, aunque podemos observar que la tercera heurística tiene un menor coste de exploración asociado, visitando solamente 11 nodos, en comparación con los 16 y 17 nodos de las otras dos.

Para poder afirmar que las tres heurísticas obtendrán la solución óptima al cambiar la especificación del mapa, debemos analizar la admisibilidad de estas.

Admisibilidad

Definición

La admisibilidad es la propiedad de una heurística $h(n)$ tal que, para todos los estados n , se cumple que:

$$h(n) \leq h^*(n)$$

siendo $h^*(n)$ el coste real mínimo para llegar al objetivo desde el nodo n .

Primera Heurística

Nuestra primera heurística hace referencia a la **distancia Manhattan**, que nos calcula la cantidad de movimientos necesarios (no diagonales) para alcanzar el objetivo, que en nuestro caso coincide con la cantidad mínima de celdas visitadas en el caso de tener un mapa sin obstáculos entre la celda inicial y objetivo. La heurística nunca sobreestima el coste real del camino, cumpliendo así con la condición de admisibilidad, asegurando que siempre estará por debajo o igualará el coste óptimo cuando los movimientos tienen un coste uniforme.

$$h1(n) \leq h^*(n)$$

Segunda Heurística

Nuestra segunda heurística hace referencia a la **máxima diferencia absoluta** encontrada de forma horizontal o vertical. Podemos ver cómo se cumple con esta heurística que para todo nodo n :

$$h2(n) \leq h1(n)$$

Esta relación indica que para cualquier nodo n , $h2(n)$ nunca será mayor que $h1(n)$. Dado que $h1(n)$ es admisible, es decir siempre cumple que $h(n) \leq h^*(n)$, podemos decir que $h2(n) \leq h1(n) \leq h^*(n)$, por lo tanto, $h2(n)$ también cumple la condición de admisibilidad.

Tercera Heurística

Esta última heurística es una derivada de la **distancia Manhattan multiplicada por un factor de 2**. En este caso, el coste de todas las acciones es unitario, por lo que a través de un contraejemplo podremos verificar que no es admisible debido a que la heurística sobreestima el coste real:

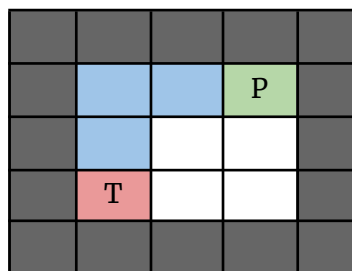


Figura 1: Mapa de ejemplo para criterio de admisibilidad

En este mapa, podemos ver como el coste mínimo es 4, pero para ese nodo T , $h3(T) = 8$, por lo que $h^*(T) < h3(T)$ y por ende, no cumple el criterio de admisibilidad.

Para ver visualmente como esto explica que pueda no encontrar soluciones óptimas proponemos el siguiente mapa con distinto coste asociado (Arriba: 0.5, Abajo: 3, Derecha: 3, Izquierda: 1.2):

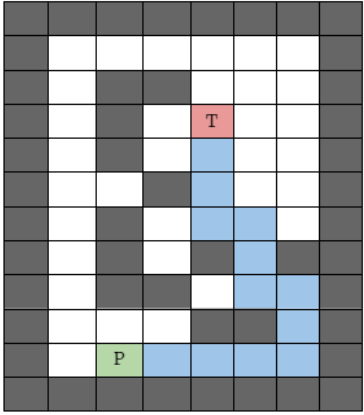
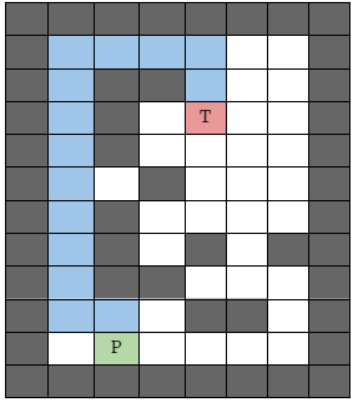
Heurística 1	Heurística 3
	
Coste total: 31.79	Coste total: 34.6

Tabla 4: Resultados de la ejecución de la heurística 1 y la heurística 3 para un mapa en el que la heurística 3 no encuentra la solución óptima.

En cuanto a la eficiencia, el algoritmo A* no es igualmente eficiente en todos los casos. Su rendimiento depende en gran medida de la heurística que se utilice y la estructura del mapa entre otros factores. Cuando la heurística es precisa y bien informada, A* es generalmente muy eficiente, pero cuando la heurística es poco informada o el problema complejo (muchos obstáculos o un alto factor de ramificación), la eficiencia de A* puede disminuir y se puede acercar a la de otros algoritmos como BFS o UCS.