Facsímil 02

Inteligencia clásica

Capítulo 01

Búsqueda: resolver problemas como espacio de estados

686f6c61 · 2026

Capítulo 01: Búsqueda: resolver problemas como espacio de estados

Entrando en el tema

A mediados de los años cincuenta, Allen Newell, J. C. Shaw y Herbert Simon mostraron que un programa podía demostrar teoremas explorando sistemáticamente el espacio de posibles deducciones. Poco después formalizaron el *General Problem Solver* como intento de solucionador general mediante búsqueda en espacios de estados.[1] No usaba redes neuronales. No usaba aprendizaje. Usaba **búsqueda**.

Siete décadas después, la búsqueda sigue siendo el corazón de sistemas que usas a diario. Un sistema de navegación puede encontrar rutas explorando un espacio donde cada estado es una ubicación y cada acción es una carretera. Un videojuego puede mover personajes con A*, un algoritmo de búsqueda de 1968. Y cuando diseñamos un agente LLM que decide qué herramienta llamar a continuación, podemos modelar esa decisión, de forma implícita y heurística, como un problema de búsqueda.

Este capítulo empieza por el principio: ¿qué es exactamente un problema de búsqueda? ¿Cómo se modela? ¿Y por qué algo tan simple es, al mismo tiempo, tan poderoso y tan peligrosamente costoso?

El vocabulario de la búsqueda

Todo problema de búsqueda empieza con cuatro ideas operativas: estado, acción, meta y coste.[2] Luego, cuando lo escribimos con notación formal, esas ideas se descomponen un poco más para que no quede nada en el aire.

Estado

Una descripción suficiente de la situación actual. «Estoy en Madrid». «El robot está en la coordenada (3,7) mirando al norte». «El puzzle tiene las piezas en esta configuración».

El estado debe contener **todo lo necesario para decidir qué hacer a continuación**. Si falta información, el algoritmo tomará decisiones incorrectas. Piensa en el estado como una fotografía del problema en un instante: debe capturar todo lo relevante y nada de lo irrelevante.[3]

Un error clásico es incluir información redundante en el estado, como «la temperatura ambiente» o «el color del coche», que no afecta a las decisiones pero multiplica el número de estados posibles. Cada bit innecesario en el estado duplica el espacio de búsqueda.

Acción

Una operación que transforma un estado en otro. «Viajar de Madrid a Barcelona» (coste: 620 km). «Avanzar una casilla hacia el norte» (coste: 1 paso). «Mover la pieza A a la posición B» (coste: 1 movimiento).

No todas las acciones están disponibles en todos los estados. Desde Madrid puedes viajar a Barcelona, pero no a Buenos Aires (no hay carretera). Esta restricción, llamada **precondición**, es lo que diferencia un problema bien modelado de uno imposible. Las acciones definen la topología del espacio de búsqueda: qué estados están conectados y a qué precio.[4]

Meta

La condición que queremos alcanzar. Puede ser un estado concreto («estar en Berlín») o una propiedad («tener todas las cajas en la zona de carga, da igual en qué orden»). La meta puede ser única o múltiple; puede ser un estado exacto o una condición abstracta.

Lo crucial es que la meta sea **verificable**. Dado un estado, debes poder responder «sí» o «no» a la pregunta «¿es este el objetivo?» sin ambigüedad. Si no puedes verificarlo, no puedes saber cuándo has terminado.[5]

Coste

El precio de cada acción o del camino completo. Distancia, tiempo, dinero, consumo de combustible, tokens de API: cualquier magnitud que queramos minimizar. El coste es lo que convierte «encuentra un camino cualquiera» en «encuentra el mejor camino».

Sin coste, cualquier camino sirve. Con coste, la búsqueda se convierte en **optimización**: no solo encontrar una solución, sino encontrar la mejor. Esta distinción, satisfacer frente a optimizar, es una de las más importantes de la IA.[6]

Cómo es el espacio de estados

Antes de elegir algoritmo conviene saber con qué tipo de espacio tratamos, porque eso decide qué garantías son posibles. Cuatro ejes lo describen.

Finito o infinito. Un tablero de ajedrez tiene un número enorme pero finito de posiciones. El espacio de las coordenadas de un robot que puede avanzar pasos arbitrarios es infinito. En un espacio infinito, un algoritmo puede no terminar nunca si no se controla la profundidad de la

exploración.

Discreto o continuo. Las casillas de un laberinto son discretas: das un paso o no lo das. La posición de un brazo robótico es continua, con infinitos ángulos intermedios. La búsqueda clásica trabaja sobre espacios discretos. Los continuos se suelen discretizar o se atacan con otras técnicas.

Determinista o no determinista. En un espacio determinista, aplicar una acción a un estado produce siempre el mismo estado siguiente. En uno no determinista, la misma acción puede llevar a varios estados, como al tirar un dado o cuando un enemigo se mueve por su cuenta. Casi toda la búsqueda de este bloque asume transiciones deterministas.

Observable o parcialmente observable. Si conoces el estado completo en todo momento, el espacio es observable. Si solo ves una parte, como la niebla de guerra de un videojuego o un sensor con ruido, es parcialmente observable, y el agente debe razonar sobre los estados que cree probables, no sobre el estado real. Esta distinción reaparece cuando hablemos de agentes.

Este capítulo y los dos siguientes asumen el caso más sencillo: espacios discretos, deterministas y observables. Es la base sobre la que después se relajan las suposiciones.

La explosión combinatoria: por qué la búsqueda es difícil

El espacio de estados crece de forma aterradora. Imagina un problema donde cada estado tiene, de media, 10 acciones posibles (factor de ramificación $b = 10$). A profundidad 1, tienes 10 estados. A profundidad 2, 100. A profundidad 5, 100 000. A profundidad 10, diez mil millones.[7]

La forma compacta de verlo es esta:

$$N(d) = \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$N(d)$	Número total de nodos que aparecen hasta profundidad d, contando la raíz.	Si exploramos hasta profundidad 5, $N(5)$.
b	Factor de ramificación medio: cuántas acciones nuevas aparecen desde cada estado.	$b = 10$.
d	Profundidad máxima explorada: cuántas acciones tiene el camino más largo considerado.	$d = 5$.
i	Nivel concreto del árbol de búsqueda.	$i = 0$ es el estado inicial; $i = 5$, los estados a cinco acciones.

Con números:

$$N(5) = 1 + 10 + 100 + 1\,000 + 10\,000 + 100\,000 = 111\,111$$

Cinco decisiones encadenadas ya generan más de cien mil nodos. Con profundidad 10:

$$N(10) = \frac{10^{11} - 1}{9} = 11\,111\,111\,111$$

Más de once mil millones. Por eso no basta con decir «probemos todas las opciones». La búsqueda clásica consiste, en buena medida, en decidir **qué opciones no merece la pena mirar**.

Esto es la **explosión combinatoria**: el número de estados crece exponencialmente con la profundidad. No es un problema de hardware: aunque tuvieras un ordenador mil millones de veces más rápido, solo podrías explorar unos pocos niveles adicionales. La explosión combinatoria es el obstáculo central de la búsqueda.

Por eso los algoritmos de búsqueda no intentan explorar todo el espacio. Usan la **frontera** para decidir qué explorar a continuación, ignorando la mayor parte del espacio. La diferencia entre un algoritmo que encuentra la solución en segundos y uno que no la encuentra nunca está en cómo gestiona esa frontera.

Árbol de búsqueda vs grafo de estados

Aquí hay una distinción sutil pero crítica. El **grafo de estados** es el mapa real: todas las configuraciones posibles y cómo se conectan. Es la realidad física del problema. El **árbol de búsqueda** es lo que el algoritmo construye mientras explora: un registro de los caminos que ha probado.[8]

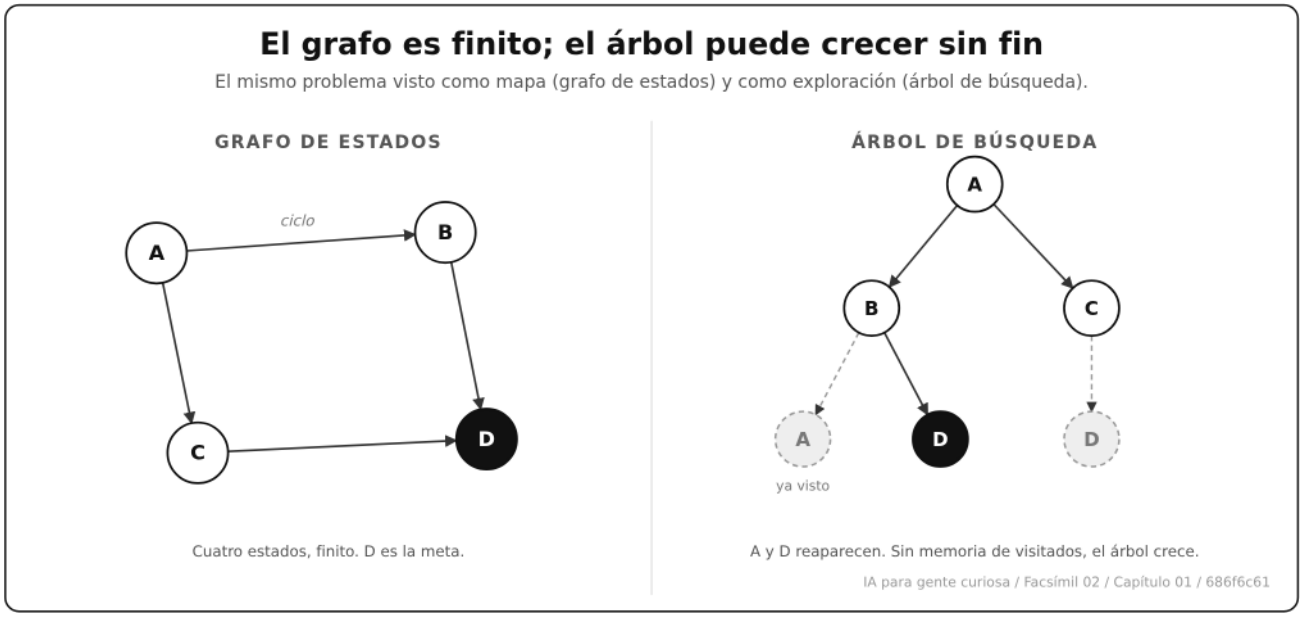
Imagina un laberinto. El grafo de estados es el laberinto completo, donde cada casilla es un estado y cada pasillo una acción. El árbol de búsqueda es el recorrido que haces al explorar: empiezas en la entrada, pruebas un pasillo, vuelves atrás si no lleva a ningún sitio, pruebas otro. El árbol crece a medida que avanzas, pero nunca ves el grafo completo. Si lo vieras, ya habrías resuelto el problema.

¿Por qué importa esta diferencia? Porque el grafo puede tener **ciclos**: puedes ir de A a B y de B a A, dando vueltas. El árbol, si no tienes cuidado, puede crecer infinitamente reflejando esos ciclos una y otra vez. Los buenos algoritmos de búsqueda mantienen un conjunto de estados **visitados** y evitan reexplorarlos. Sin esta precaución, BFS y UCS pueden quedar atrapados en bucles infinitos.[9]

Búsqueda en árbol y búsqueda en grafo

De aquí salen las dos variantes del mismo algoritmo. La **búsqueda en árbol** (*tree search*) no recuerda dónde ha estado: expande la frontera sin comprobar si un estado ya había aparecido. Es más simple y gasta menos memoria, pero en un espacio con ciclos puede reexpandir el mismo estado una y otra vez, e incluso no terminar nunca. La **búsqueda en grafo** (*graph search*) mantiene un conjunto de estados visitados y descarta los que ya expandió. Nunca repite trabajo ni cae en bucles, a cambio de guardar en memoria todos los estados vistos.

La decisión no es estética: la búsqueda en grafo cambia coste de cómputo por coste de memoria. Evita reexpansiones, pero el conjunto de visitados puede crecer hasta ser tan grande como el propio espacio alcanzable. En problemas con muchos caminos que llevan al mismo sitio, como rejillas o mapas de carreteras, la búsqueda en grafo es casi obligatoria. En árboles sin ciclos reales, la búsqueda en árbol basta y ahorra memoria.



Cómo funciona un algoritmo de búsqueda

Todos los algoritmos de búsqueda clásicos comparten la misma estructura.[10] Mantienen una **frontera**: el conjunto de estados que han sido descubiertos pero aún no explorados. El bucle es:

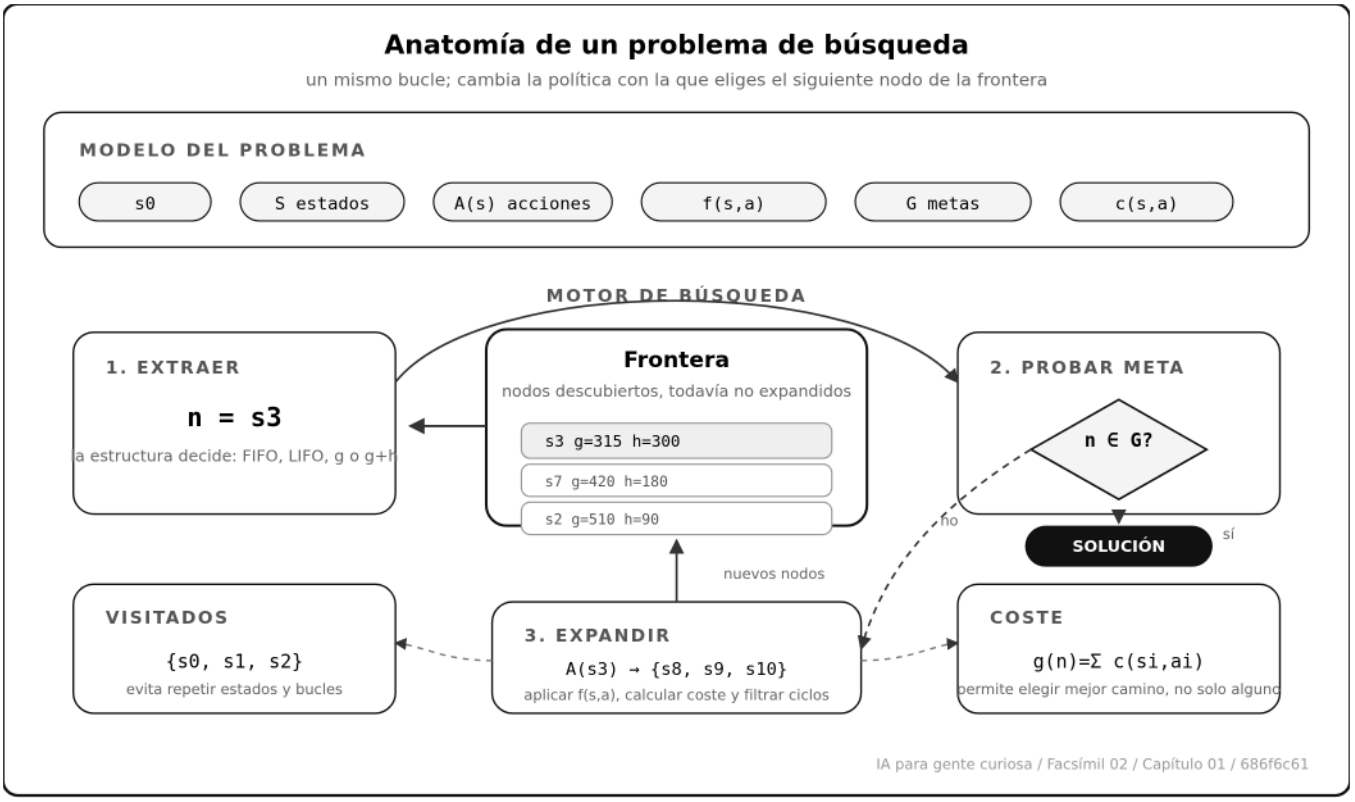
1. **Extraer** un estado de la frontera.
2. **Comprobar** si es la meta. Si lo es, reconstruir el camino y terminar.
3. **Expandir**: generar todos los estados alcanzables desde este estado mediante las acciones disponibles.
4. **Añadir** los nuevos estados a la frontera, siempre que no hayan sido visitados antes.
5. **Repetir** desde el paso 1.

Eso es todo. Cuatro pasos y un bucle. La diferencia entre BFS, DFS, coste uniforme, greedy y A* está exclusivamente en el paso 1: **qué estado extraemos de la frontera**. La estructura de datos que usemos (cola, pila, cola de prioridad) determina completamente el comportamiento del algoritmo.

En pseudocódigo, el esqueleto común de la búsqueda en grafo es este:

```
buscar(problema):
    frontera  ← { nodo(s0, coste=0, padre=None) }
    visitados ← { }
    mientras frontera no esté vacía:
        n ← extraer(frontera)              # aquí decide el algoritmo
        si n.estado ∈ G:
            devolver reconstruir_camino(n)
        si n.estado ∈ visitados:
            continuar
        visitados ← visitados ∪ { n.estado }
        para cada acción a en A(n.estado):
            s' ← f(n.estado, a)
            si s' ∉ visitados:
                g' ← n.coste + c(n.estado, a)
                frontera ← frontera ∪ { nodo(s', g', padre=n) }
    devolver fracaso
```

La única línea que cambia entre algoritmos es `extraer(frontera)`. Si la frontera es una cola FIFO, sale el nodo más antiguo y tenemos BFS. Si es una pila LIFO, sale el más reciente y tenemos DFS. Si es una cola de prioridad ordenada por coste acumulado, sale el más barato y tenemos coste uniforme. Si se ordena por coste más heurística, tenemos A*. El bucle es idéntico, la política de extracción lo es todo. La función `reconstruir_camino` sigue los punteros al padre desde la meta hasta el estado inicial para devolver el plan completo.



Definición formal del problema

Conviene fijar la notación con precisión. Un problema de búsqueda se define formalmente como una tupla:[11]

$$\text{Problema} = (S, A, f, s_0, G, c)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
S	Espacio de estados: conjunto de todas las configuraciones posibles.	En un mapa, todos los cruces y ciudades que podrían visitarse.
$A(s)$	Acciones aplicables desde el estado s.	Desde Zaragoza, tomar la carretera hacia Barcelona o hacia Madrid.
$f(s,a)$	Función de transición: el estado que aparece al aplicar una acción.	$f(\text{Zaragoza}, \text{ir a Barcelona}) = \text{Barcelona}$.
s_0	Estado inicial.	Madrid.
G	Conjunto de estados meta.	$\{\text{Barcelona}\}$.
$c(s,a)$	Coste de aplicar la acción a en el estado s.	Kilómetros, minutos, euros o tokens consumidos.

La fórmula dice algo bastante sencillo: para resolver el problema necesitamos saber dónde podemos estar, qué podemos hacer, qué ocurre al hacerlo, dónde empezamos, qué cuenta como llegar y cuánto cuesta cada paso. Si falta una de esas piezas, la búsqueda queda coja.

Una **solución** es una secuencia de acciones:

$$\pi = (a_1, a_2, \ldots, a_n)$$

La letra π se lee «pi» y aquí representa un **plan**: una lista ordenada de acciones. No es todavía «la mejor» solución; solo es una candidata. Para comprobar si realmente sirve, aplicamos cada acción una detrás de otra:

$$s_i = f(s_{i-1}, a_i), \quad i = 1, 2, \ldots, n$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
π	Plan o secuencia de acciones.	$(\text{Madrid} \to \text{Zaragoza}, \text{Zaragoza} \to \text{Barcelona})$.
a_i	Acción número i del plan.	$a_1 =$ ir de Madrid a Zaragoza.
s_i	Estado alcanzado después de ejecutar i acciones.	$s_1 =$ Zaragoza; $s_2 =$ Barcelona.
f	Función de transición que calcula el siguiente estado.	Aplicar «ir a Zaragoza» desde Madrid produce Zaragoza.
n	Número total de acciones del plan.	$n = 2$.

El plan es una solución si el último estado pertenece al conjunto de metas:

$$s_n \in G$$

En nuestro ejemplo, $s_2 = \text{Barcelona}$ y $G = \{\text{Barcelona}\}$. Por tanto, el plan llega a la meta.

Ahora falta saber cuánto cuesta. El coste total de un plan es la suma de los costes de cada acción:

$$C(\pi) = \sum_{i=1}^{n} c(s_{i-1}, a_i)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$C(\pi)$	Coste total del plan π.	Kilómetros totales recorridos.
$c(s_{i-1}, a_i)$	Coste de ejecutar la acción a_i desde el estado anterior.	$c(\text{Madrid}, \text{ir a Zaragoza}) = 315$ km.
i	Índice de la acción dentro del plan.	$i = 1$ para el primer tramo; $i = 2$ para el segundo.
n	Número de acciones del plan.	$n = 2$.

Con números:

$$C(\pi) = 315 + 300 = 615$$

Si existe otro plan Madrid → Valencia → Barcelona con coste 720, ambos llegan a la meta, pero el primero es mejor. Una solución es **óptima** si minimiza $C(\pi)$ entre todas las soluciones posibles.

Esta formalización permite razonar sobre propiedades como la completitud (¿el algoritmo siempre encuentra solución si existe?), la optimalidad (¿encuentra la mejor?) y la complejidad (¿cuánto tarda y cuánta memoria consume en función del tamaño del problema?).[1][2]

Las cuatro propiedades de un algoritmo de búsqueda

Cada vez que evaluamos un algoritmo de búsqueda nos hacemos las mismas cuatro preguntas. Son el lenguaje con el que el capítulo 2 compara BFS, DFS y coste uniforme, y el capítulo 3, A*.

Completitud. ¿El algoritmo encuentra una solución siempre que exista? Un algoritmo completo no se queda dando vueltas para siempre ni abandona un problema que sí tenía respuesta. BFS es completo. DFS, en un espacio infinito, no lo es.

Optimalidad. ¿La solución que encuentra es la de menor coste? Un algoritmo puede ser completo pero no óptimo: encuentra una solución, no necesariamente la mejor. BFS es óptimo solo si todas las acciones cuestan lo mismo. El coste uniforme lo es siempre.

Complejidad temporal. ¿Cuántos estados expande en el peor caso? Se mide con el factor de ramificación y la profundidad de la solución.

Complejidad espacial. ¿Cuánta memoria necesita a la vez? Suele ser el tamaño máximo de la frontera más los visitados. En la práctica, la memoria es el límite que primero se alcanza.

Para una búsqueda no informada con factor de ramificación b y profundidad de la solución d, las cotas de referencia son:

$$T = O(b^d), \qquad M = O(b^d)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
T	Complejidad temporal: orden del número de estados expandidos en el peor caso.	Con $b = 10$ y $d = 8$, del orden de 10^8.
M	Complejidad espacial: orden de la memoria ocupada por la frontera y los visitados.	También 10^8 si hay que mantener toda la frontera.
b	Factor de ramificación medio.	$b = 10$.
d	Profundidad de la solución más superficial.	$d = 8$.

En palabras: el tiempo crece de forma exponencial con la profundidad, y la memoria suele crecer igual de rápido, que es la razón de fondo por la que la búsqueda ciega solo sirve para espacios pequeños. La diferencia entre algoritmos está en la constante y, sobre todo, en si la memoria es $O(b^d)$, como en BFS, o solo $O(b \cdot d)$, como en DFS, que guarda únicamente el camino actual. Esa tensión entre tiempo, memoria y garantías es la que ordena todo el resto del bloque.

Búsqueda no informada vs informada

Los algoritmos de búsqueda se dividen en dos familias, y la línea que las separa es la información disponible:

No informados (ciegos). No tienen ninguna pista sobre qué acciones son mejores. Solo conocen el coste de las acciones ya realizadas y la definición de la meta. Son como explorar un laberinto a oscuras, tanteando las paredes. BFS, DFS y coste uniforme pertenecen a esta familia. Su gran ventaja es que no necesitan conocimiento específico del dominio: funcionan para cualquier problema bien definido. Su gran desventaja es la ineficiencia: exploran a ciegas, sin priorizar.[13]

Informados (heurísticos). Disponen de una función heurística $h(n)$ que estima lo lejos que está un estado de la meta. Es una estimación diseñada para ser barata de calcular y suficientemente informativa para ordenar la búsqueda; no sustituye al coste real ni a la prueba de optimalidad. Greedy y A* usan heurísticas para priorizar estados prometedores. Su

gran ventaja es la eficiencia: pueden encontrar soluciones explorando órdenes de magnitud menos estados. Su gran desventaja es que necesitan una buena heurística, y diseñar una heurística admisible, que nunca sobreestime el coste real, no siempre es fácil.[14]

El resto de este bloque de búsqueda se estructura alrededor de esta división: el capítulo 2 explora los algoritmos ciegos; el capítulo 3, los informados; y el capítulo 4 tiende el puente hacia los agentes modernos.

En el día a día

La búsqueda en espacios de estados no es una reliquia académica: aparece en sistemas reales constantemente. La navegación GPS busca una ruta útil entre dos puntos modelando cada cruce como un estado y cada calle como una acción con un coste, normalmente distancia, tiempo o una combinación de ambas. A* y sus variantes ayudan porque una heurística geométrica permite mirar antes las rutas prometedoras, aunque los sistemas reales añaden tráfico, restricciones de giro, cierres, peajes y optimizaciones de ingeniería.[15]

La planificación logística es otro caso cotidiano. Una empresa con 50 paquetes y 5 furgonetas tiene un espacio de estados astronómico, combinatorio y con un factor de ramificación enorme, y aun así los algoritmos de búsqueda heurística encuentran soluciones casi óptimas en segundos. La diferencia entre una ruta óptima y una subóptima son miles de euros al mes en combustible.

Hasta los agentes LLM encajan en este marco. Cuando un agente decide si llamar a `search_database` o a `check_calendar`, podemos leer esa decisión como una evaluación de acciones en un espacio de estados implícito, y el propio modelo actúa como heurística: «dado el contexto actual, ¿qué herramienta parece más prometedora?». El sistema que lo rodea debe añadir costes, permisos, observación y un criterio de parada.

Antes del algoritmo: auditar el modelo

Un error muy común es discutir si usar BFS, A* o una heurística sofisticada antes de haber comprobado que el problema está bien definido. En ingeniería, el primer paso no es elegir algoritmo: es validar el **contrato de búsqueda**.

PREGUNTA	QUÉ COMPRUEBA	FALLO TÍPICO
¿s_0 pertenece a S?	Que el estado inicial existe en el modelo.	Arrancar desde un identificador que no está en el grafo.
¿$G \subseteq S$?	Que todas las metas son estados válidos.	Definir como meta una etiqueta textual que ninguna transición alcanza.
¿Toda acción tiene origen, destino y coste?	Que $f(s, a)$ y $c(s, a)$ están definidos.	Acción sin coste o transición a estado inexistente.
¿Los costes son no negativos?	Que UCS y A* puedan razonar correctamente.	Costes negativos que rompen garantías de optimalidad.
¿Hay ciclos?	Que el algoritmo necesitará `visitados`.	Madrid → Zaragoza → Madrid repitiéndose para siempre.
¿Cuál es b y hasta qué profundidad buscarías?	Estimación de explosión combinatoria.	Descubrir tarde que b^d no cabe en memoria.

Esta auditoría parece humilde, pero es exactamente el tipo de trabajo que evita sistemas frágiles. Si el contrato está mal, el algoritmo puede estar perfectamente implementado y aun así devolver basura.

Un plan candidato también se puede comprobar antes de hablar de optimalidad. Un plan no es más que una secuencia de acciones; decimos que es **válido** (una *solución*, en el vocabulario de Russell y Norvig) cuando cada acción es aplicable en el estado en que se ejecuta, encadena bien sus transiciones y termina en un estado meta:[16]

$$\text{válido}(\pi) \Leftrightarrow s_0 \overset{a_1}{} s_1 \overset{a_2}{} \cdots \overset{a_n}{} s_n \wedge s_n \in G$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
π	Plan candidato: la secuencia $\langle a_1, \ldots, a_n \rangle$.	Madrid → Zaragoza → Barcelona.
$s_{i-1} \overset{a_i}{} s_i$	Cada acción es aplicable y lleva del estado anterior al siguiente.	«ir a Zaragoza» es una transición legal desde Madrid.
$s_n \in G$	El último estado es una meta.	Barcelona pertenece a G.
$\Leftrightarrow$	Válido si, y solo si, se cumplen ambas cosas a la vez.	Cadena legal **y** final en meta.

La doble condición importa: no basta con terminar en meta por casualidad (el camino tiene que ser legal paso a paso), ni basta con encadenar acciones legales que nunca lleguen a G. Hacen falta las dos cosas.

Validar es barato; demostrar **optimalidad** es caro. Comprobar que un plan concreto es válido y medir su coste se hace en tiempo lineal, recorriendo la secuencia una vez. Pero saber que es el **mejor** plan exige compararlo contra todos los demás, que es justo el trabajo que hacen algoritmos como UCS o A*. Por eso primero se valida y se mide el coste, y solo después se discute optimalidad:

$$C(\pi) = \sum_{i=1}^{n} c(s_{i-1}, a_i)$$

Un plan π^* es **óptimo** cuando ningún plan válido cuesta menos, es decir, $C(\pi^*) \leq C(\pi)$ para todo π válido. Comparar unos cuantos candidatos te dice cuál es el mejor *entre los que tienes*; demostrar que es el mejor *de todos* requiere explorar el espacio con garantías, no solo enumerar algunos. Por eso el lab de este capítulo evalúa planes candidatos pero no presume de optimalidad global: comparar no es demostrar.

Una práctica útil es mantener en tus sistemas una salida trazable: qué estados se recorrieron, qué acción llevó a cada estado, qué coste se acumuló y por qué el plan termina o no en meta. Eso conecta este capítulo con planificación, agentes y evaluación: no basta con “llegó”; hay que poder explicar cómo.

Por qué debería importarte

La búsqueda es el paradigma más antiguo de la IA y, paradójicamente, uno de los más vigentes. No ha sido sustituido por el *deep learning*: ha sido **aumentado** por él. Los LLMs no reemplazan la búsqueda; la hacen más inteligente, proporcionando heurísticas donde antes había que diseñarlas a mano.

Entender la búsqueda te da un marco mental para razonar sobre problemas secuenciales: aquellos donde la solución no es una respuesta única, sino una secuencia de decisiones. Y te prepara para los capítulos siguientes: sin entender el vocabulario de estados, acciones y fronteras, no puedes entender CSP, planificación ni juegos. Todo el facsímil 2 se construye sobre este capítulo.

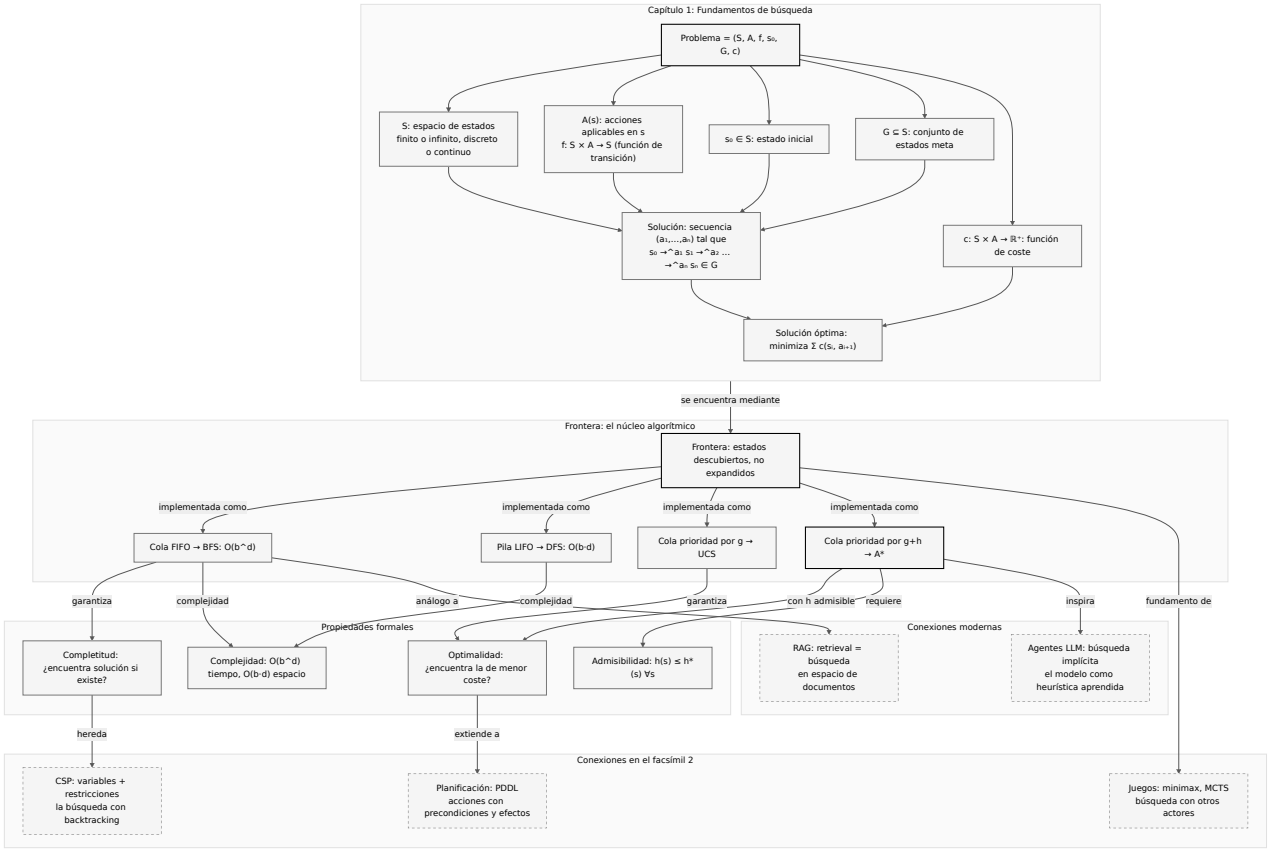
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Confundir el grafo de estados con el árbol de búsqueda	El grafo es el mapa completo; el árbol es lo que el algoritmo explora. Tratar el árbol como si fuera el grafo lleva a reexplorar estados innecesariamente y a no detectar ciclos.	Mantén un conjunto `visited` y nunca reexpandas un estado ya visitado. Es la optimización más rentable en búsqueda.
No definir bien el estado	Si el estado no contiene toda la información necesaria para decidir la siguiente acción, el algoritmo tomará decisiones basadas en información incompleta.	Pregúntate: «con esta información, ¿puedo generar todas las acciones posibles y evaluar si he llegado a la meta?». Si la respuesta es no, tu estado está incompleto.
Subestimar la explosión combinatoria	Un espacio con b=10 y d=10 tiene 10^10 estados. Explorarlos todos es inviable en cualquier hardware. La búsqueda ciega solo funciona para espacios pequeños.	Calcula b y d antes de elegir algoritmo. Si b^d es mayor que unos pocos millones, necesitas una heurística o una poda.
Olvidar el coste	Sin coste, cualquier camino sirve. Con costes variables, el camino más corto en pasos no es necesariamente el más barato. BFS encuentra el primero; UCS encuentra el mejor.	Define el coste de cada acción. Si los costes no son uniformes, usa UCS o A*, no BFS.

Cómo encaja todo

Este mapa se lee de izquierda a derecha: primero defines el problema, después eliges cómo gestionar la frontera, y solo entonces aparecen las garantías de completitud, optimalidad y coste. Los capítulos siguientes cambian la política de frontera, añaden heurísticas o convierten estados en asignaciones, planes y decisiones con otros actores.

La conexión con sistemas modernos no es decorativa: un agente con herramientas también necesita estado, acciones, coste, meta y trazabilidad. Cambia la forma de representar el problema, pero no desaparece el patrón.


Capítulo 1: Fundamentos de búsqueda
Problema = (S, A, f, s₀, G, c)
S: espacio de estados finito o infinito, discreto o continuo
A(s): acciones aplicables en s
f: S × A → S (función de transición)
s₀ ∈ S: estado inicial
G ⊆ S: conjunto de estados meta
Solución: secuencia (a₁,...,aₙ) tal que s₀ →^a₁ s₁ →^a₂ ... →^aₙ sₙ ∈ G
c: S × A → ℝ⁺: función de coste
Solución óptima: minimiza Σ c(sᵢ, aᵢ₊₁)
se encuentra mediante
Frontera: el núcleo algorítmico
Frontera: estados descubiertos, no expandidos
implementada como
Cola FIFO → BFS: O(b^d)
implementada como
Pila LIFO → DFS: O(b·d)
implementada como
Cola prioridad por g → UCS
implementada como
Cola prioridad por g+h → A*
garantiza
complejidad
análogo a
complejidad
garantiza
con h admisible
requiere
inspira
fundamento de
Propiedades formales
Completitud: ¿encuentra solución si existe?
Complejidad: O(b^d) tiempo, O(b·d) espacio
Optimalidad: ¿encuentra la de menor coste?
Admisibilidad: h(s) ≤ h*(s) ∀s
Conexiones modernas
RAG: retrieval = búsqueda en espacio de documentos
Agentes LLM: búsqueda implícita el modelo como heurística aprendida
hereda
extiende a
Conexiones en el facsímil 2
CSP: variables + restricciones la búsqueda con backtracking
Planificación: PDDL acciones con precondiciones y efectos
Juegos: minimax, MCTS búsqueda con otros actores

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Estado	Descripción suficiente de la situación actual del problema en un momento dado.
Espacio de estados	Conjunto de todas las configuraciones posibles que puede alcanzar el problema.
Árbol de búsqueda	Estructura que el algoritmo construye al explorar, donde cada nodo es un estado y cada rama una acción.
Frontera	Conjunto de estados descubiertos pero aún no explorados. Su estructura determina el algoritmo.
Factor de ramificación	Número medio de acciones disponibles en cada estado. Determina la explosión combinatoria.
Heurística	Función que estima la distancia de un estado a la meta, guiando la búsqueda informada.
Contrato de búsqueda	Definición verificable de estados, acciones, transición, estado inicial, metas y costes.
Plan candidato	Secuencia de acciones que debe poder ejecutarse desde s_0 y terminar en G para ser solución.
Estado visitado	Estado ya revisado para evitar ciclos y reexpansiones innecesarias.
Búsqueda en grafo	Variante que guarda los estados visitados para no repetirlos, frente a la búsqueda en árbol, que no los recuerda.
Completitud	Propiedad de un algoritmo que encuentra solución siempre que exista.
Optimalidad	Propiedad de un algoritmo que encuentra la solución de menor coste.

Antes de pasar página

- [ ] ¿Puedo definir los cuatro ingredientes de un problema de búsqueda y poner un ejemplo concreto de cada uno? (Si no, vuelve a «El vocabulario de la búsqueda».)

- [ ] ¿Entiendo por qué la explosión combinatoria hace que la búsqueda ciega sea inviable para espacios grandes? (Si no, vuelve a «La explosión combinatoria».)

- [ ] ¿Sé diferenciar el grafo de estados del árbol de búsqueda y explicar por qué importa la diferencia? (Si no, vuelve a «Árbol de búsqueda vs grafo de estados».)

- [ ] ¿Puedo explicar el bucle genérico de un algoritmo de búsqueda y qué línea cambia entre BFS, DFS, UCS y A*? (Si no, vuelve a «Cómo funciona un algoritmo de búsqueda».)
- [ ] ¿Puedo nombrar las cuatro propiedades de un algoritmo de búsqueda y decir qué pregunta responde cada una? (Si no, vuelve a «Las cuatro propiedades de un algoritmo de búsqueda».)
- [ ] ¿Sé la diferencia entre búsqueda en árbol y búsqueda en grafo, y por qué importa `visited`? (Si no, vuelve a «Árbol de búsqueda vs grafo de estados».)
- [ ] ¿Entiendo la diferencia entre búsqueda no informada e informada? (Si no, vuelve a «Búsqueda no informada vs informada».)
- [ ] ¿Puedo auditar si un problema está bien definido antes de elegir algoritmo? (Si no, vuelve a «Antes del algoritmo: auditar el modelo».)
- [ ] ¿Sé distinguir un plan válido de uno inválido en un problema de búsqueda bien definido? (Si no, vuelve a «Definición formal del problema».)

En resumen

IDEA FUERZA	DETALLE
Todo problema de búsqueda se define con estados, acciones, meta y coste.	Si no puedes definir estos cuatro elementos, no tienes un problema de búsqueda. Si tu estado contiene información irrelevante, tu espacio de búsqueda explota innecesariamente.
La explosión combinatoria es el obstáculo central.	El espacio crece exponencialmente con la profundidad. Más hardware ayuda poco si el modelo explora ramas inútiles. La defensa es modelar mejor, podar, usar heurísticas y medir costes.
La frontera es el corazón del algoritmo.	Qué estado extraes de la frontera determina si estás haciendo BFS, DFS, UCS o A*. Todo lo demás es el mismo bucle.
El contrato del problema va antes del algoritmo.	Si estados, acciones, transición, meta o coste están mal definidos, ningún algoritmo arregla el modelo.
La búsqueda no ha muerto: ha evolucionado.	Los LLMs no reemplazan la búsqueda; pueden aportar heurísticas aprendidas dentro de sistemas con estado, acciones, validación y trazas. El patrón viene de la IA clásica.

Para saber más

Hart, P. E., Nilsson, N. J. y Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100-107. https://doi.org/10.1109/TSSC.1968.300136

Luger, G. F. (2008). *Artificial intelligence: structures and strategies for complex problem solving* (6.ª ed.). Pearson.

Newell, A., Shaw, J. C. y Simon, H. A. (1959). Report on a general problem-solving program. En *Proceedings of the International Conference on Information Processing* (pp. 256-264). UNESCO.

Nilsson, N. J. (1998). *Artificial intelligence: a new synthesis*. Morgan Kaufmann.

Pearl, J. (1984). *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley.

Poole, D., Mackworth, A. y Goebel, R. (1998). *Computational intelligence: a logical approach*. Oxford University Press.

Rich, E., Knight, K. y Nair, S. B. (2009). *Artificial intelligence* (3.ª ed.). McGraw-Hill.

Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. https://aima.cs.berkeley.edu/

Notas

1. Newell, A., Shaw, J. C. y Simon, H. A. (1959). Report on a general problem-solving program. En *Proceedings of the International Conference on Information Processing* (pp. 256-264). UNESCO. ↩

2. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. Los capítulos 3 y 4 abordan la resolución de problemas mediante búsqueda, estableciendo el marco conceptual de estados, acciones, metas y costes que sigue siendo la base de la IA moderna. ↩

3. Nilsson, N. J. (1998). *Artificial intelligence: a new synthesis*. Morgan Kaufmann. El capítulo 7 presenta los fundamentos de la búsqueda en espacios de estados, incluyendo la distinción entre búsqueda ciega y búsqueda heurística. ↩

4. Rich, E., Knight, K. y Nair, S. B. (2009). *Artificial intelligence* (3.ª ed.). McGraw-Hill. El capítulo 2 describe cómo modelar problemas como espacios de estados, enfatizando la importancia de definir correctamente las precondiciones de cada acción. ↩

5. Poole, D., Mackworth, A. y Goebel, R. (1998). *Computational intelligence: a logical approach*. Oxford University Press. La sección 3.3 analiza cómo definir metas verificables y su papel en la terminación correcta de los algoritmos de búsqueda. ↩

6. Luger, G. F. (2008). *Artificial intelligence: structures and strategies for complex problem solving* (6.ª ed.). Pearson. El capítulo 3 desarrolla la distinción entre búsqueda de cualquier solución y búsqueda de la solución óptima, y cómo el coste transforma el problema. ↩

7. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. La sección 3.3 introduce el concepto de complejidad de la búsqueda y analiza cómo el factor de ramificación y la profundidad determinan la viabilidad de los algoritmos. ↩

8. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. La sección 3.3 detalla la diferencia entre el grafo de estados y el árbol de búsqueda. ↩

9. Poole, D., Mackworth, A. y Goebel, R. (1998). *Computational intelligence: a logical approach*. Oxford University Press. El capítulo 3 aborda la detección de ciclos como una optimización crítica para la viabilidad práctica de los algoritmos de búsqueda. ↩

10. Nilsson, N. J. (1998). *Artificial intelligence: a new synthesis*. Morgan Kaufmann. El capítulo 7 presenta un marco unificado donde la única diferencia entre algoritmos es la política de extracción de la frontera. ↩

11. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. La sección 3.1 introduce la definición formal del problema de búsqueda, incluyendo la notación de espacios de estados, función de transición, y función de coste. ↩

12. Pearl, J. (1984). *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley. El capítulo 2 formaliza las propiedades de los algoritmos de búsqueda en términos de completitud, admisibilidad y optimalidad. ↩

13. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. La sección 3.4 clasifica los algoritmos de búsqueda no informada y analiza sus propiedades de completitud, optimalidad y complejidad. ↩

14. Pearl, J. (1984). *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley. Pearl estableció las bases teóricas de la búsqueda heurística, incluyendo las propiedades de admisibilidad y consistencia que garantizan la optimalidad de A*, y el concepto de poder heurístico que permite comparar la eficiencia de distintas heurísticas. ↩

15. Hart, P. E., Nilsson, N. J. y Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100-107. https://doi.org/10.1109/TSSC.1968.300136. ↩

16. Russell, S. y Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4.ª ed.). Pearson. Una solución de un problema de búsqueda se define como una secuencia de acciones que lleva del estado inicial a un estado objetivo. ↩