Facsímil 02

Inteligencia clásica

Capítulo 04

Búsqueda en agentes modernos: del algoritmo a la política

686f6c61 · 2026

Capítulo 04: Búsqueda en agentes modernos: del algoritmo a la política

Entrando en el tema

Has aprendido a formular un problema como estados, acciones, meta y coste. Has visto qué ocurre cuando exploras a ciegas con BFS, DFS y UCS. Después has añadido estimaciones heurísticas con Greedy y A*. Ahora toca cerrar este recorrido con una pregunta muy práctica: ¿por qué importa todo esto si hoy hablamos de LLMs, agentes y *tools*?

Porque muchas decisiones de un agente moderno pueden modelarse como decisiones en un espacio de posibilidades. Puede que no dibuje un árbol de búsqueda en pantalla. Puede que no tenga una cola de prioridad explícita llamada `frontier`. Pero cuando decide si debe leer un archivo, consultar una base de datos, llamar a una API, pedir aclaración o responder, está eligiendo una acción desde un estado, con costes, restricciones y una meta.[1]

El tema central es ese: leer un agente moderno con el lenguaje de la búsqueda clásica, para diseñarlo y auditarlo sin confundir fluidez con buen criterio. Este capítulo cierra los cuatro primeros, que han tratado todos sobre búsqueda, y el siguiente cambia de herramienta hacia las restricciones. Por eso, al final, recogeremos en pocas líneas lo esencial de esos cuatro capítulos: no para terminar, sino para llegar al capítulo de CSP con los cimientos firmes.

El agente racional

La pregunta de fondo no es solo «cómo busca un agente», sino «qué significa que un agente decida bien». Russell y Norvig dan una respuesta precisa: un agente racional selecciona, en cada momento, la acción que maximiza su medida de rendimiento esperada dada la evidencia de que dispone.[2] No se trata de adivinar el futuro ni de no equivocarse nunca, sino de elegir lo mejor posible con lo que se sabe. La herramienta matemática que formaliza esa idea es la teoría de la decisión y, en concreto, el principio de la utilidad esperada:

$$a^* = \arg\max_a \ \mathbb{E}[\, U(\text{resultado}) \mid a, e \,]$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
a^*	Acción que el agente debería elegir.	Leer la consola antes que editar código.
$U(\cdot)$	Utilidad: cómo de bueno es un resultado para el objetivo.	Resolver el ticket bien, barato y sin romper nada.
$\mathbb{E}[\cdot]$	Valor esperado: promedio ponderado por la probabilidad de cada resultado.	Promedia los desenlaces posibles de una acción incierta.
e	Evidencia disponible en ese momento.	Logs leídos, respuestas de herramientas, mensajes previos.

En palabras: de todas las acciones posibles, el agente elige la que, en promedio y dada la evidencia que tiene, conduce a los mejores resultados según su medida de utilidad.

Esto sustituye a la antigua plantilla $F = G + H + R$, que sumaba coste, avance y riesgo a mano. En el marco real, esos tres factores no son sumandos arbitrarios: son componentes de la utilidad U. El coste (tokens, latencia, dinero) entra como utilidad negativa, el avance hacia la meta entra como utilidad positiva, y el riesgo entra a través de los resultados malos que una acción puede provocar y de su probabilidad. La diferencia es importante: en lugar de inventar una fórmula propia, se usa el lenguaje estándar con el que la inteligencia artificial razona sobre decisiones desde hace décadas.

Piensa en un agente de soporte que recibe un ticket. Puede responder con una solución inmediata, pedir más datos al cliente o escalar a un humano. La respuesta inmediata es barata pero arriesgada si no entiende bien el problema; escalar es seguro pero lento y caro. Un agente racional no elige por intuición: estima, para cada acción, la utilidad esperada (probabilidad de resolver bien el ticket, descontando coste y descontando el daño esperado de equivocarse) y se queda con la de mayor valor. Esa es la brújula que recorre todo el capítulo.

Decidir cuando el resultado es incierto: el MDP

La fórmula de la utilidad esperada describe una decisión aislada. Pero un agente real encadena muchas decisiones, y cada acción cambia el mundo de forma a veces incierta: ejecutar un test puede pasar o fallar, llamar a una API puede devolver datos o un error. Cuando hay incertidumbre en los resultados y recompensas que se acumulan a lo largo del tiempo, el marco oficial es el *Proceso de Decisión de Markov* (MDP), una tupla de cinco elementos:

$$\text{MDP} = (S, A, P(s' \mid s, a), R(s, a), \gamma)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
S	Conjunto de estados posibles.	Situaciones del agente: ticket sin diagnosticar, test ya ejecutado, etc.
A	Conjunto de acciones disponibles.	Leer logs, ejecutar test, responder, escalar.
$P(s' \mid s, a)$	Probabilidad de pasar al estado s' tras hacer a en s.	El test falla el 30 % de las veces aunque el arreglo parezca correcto.
$R(s, a)$	Recompensa inmediata de hacer a en s .	Premio por resolver, penalización por gastar tokens.
γ	Factor de descuento entre 0 y 1 que pesa el futuro.	$\gamma = 0{,}9$: el futuro importa, pero menos que el presente.

En palabras: un MDP describe un mundo donde el agente está en un estado, elige una acción, recibe una recompensa y salta a un nuevo estado según una probabilidad, una y otra vez, intentando acumular la mayor recompensa posible.

La regla que dice qué hacer en cada estado se llama *política*, escrita $\pi(s)$: una función que asigna a cada estado la acción a tomar. Para saber cómo de buena es cada situación se usa la *función de valor* $V^*(s)$, que mide la recompensa total esperada si se actúa de forma óptima desde s en adelante. Esa función cumple la *ecuación de Bellman*, una de las piezas centrales de la teoría:[3] [4]

$$V^*(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s' \mid s, a)\, V^*(s') \right]$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$V^*(s)$	Valor óptimo de estar en el estado s.	Cómo de prometedora es la situación actual.
$\max_a$	Se elige la mejor acción posible.	La que maximiza recompensa inmediata más futuro.
$R(s, a)$	Recompensa inmediata.	Lo que se gana ahora mismo.
$\gamma \sum_{s'} P(s' \mid s, a)\, V^*(s')$	Valor futuro descontado y promediado sobre los estados a los que se podría llegar.	El bien que vendrá después, ponderado por su probabilidad.

En palabras: el valor de un estado es la mejor recompensa que puedes conseguir ahora más el valor (descontado) de los estados a los que esa acción te puede llevar, promediado según su probabilidad. La decisión presente y el futuro quedan ligados en una sola ecuación recursiva.

Esto es una presentación introductoria; la resolución de MDP y el aprendizaje de políticas se trata a fondo en el fascículo de aprendizaje por refuerzo.[5] Aquí basta con ver la conexión con la búsqueda clásica de los capítulos anteriores: A* y UCS son, de hecho, el caso particular del MDP en que el mundo es *determinista* (cada acción lleva siempre al mismo estado, así que P vale 1 para un único s') y *conocido* (sabes de antemano el efecto de cada acción). Cuando el resultado deja de ser seguro, la búsqueda de caminos se generaliza al MDP. Imagina un robot de almacén que ordena «avanzar»: el suelo resbala y el 10 % de las veces acaba en la casilla equivocada. Ya no hay un camino fijo que seguir; hay una política que dice, en cada casilla, hacia dónde moverse para maximizar la recompensa esperada a pesar del azar.

Cuando no se ve todo: POMDP y belief state

Hay un detalle que el MDP da por supuesto y que casi nunca se cumple en un agente real: que el agente sabe en qué estado está. Un agente LLM no observa el estado verdadero del mundo. Solo recibe señales parciales: el texto de un log, la respuesta de una herramienta, un mensaje del usuario. El estado real (qué falla exactamente, qué quiere de verdad el cliente) permanece oculto. El marco que añade esta capa es el *POMDP*, el Proceso de Decisión de Markov *parcialmente observable*, que incorpora observaciones y obliga al agente a razonar sobre lo que no ve.[6]

Como el estado real es desconocido, el agente mantiene un *belief state* (estado de creencia): una distribución de probabilidad sobre todos los estados en los que podría estar. En vez de afirmar «el bug está en el módulo de pago», el agente sostiene algo más honesto: «hay un 60 % de probabilidad de que esté en el pago, un 30 % en la red y un 10 % en la base de datos». Cada vez que actúa y observa algo nuevo, recalcula esa creencia con la regla de actualización bayesiana:

$$b'(s') = \eta \, P(o \mid s') \sum_{s} P(s' \mid s, a) \, b(s)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$b(s)$	Creencia actual: probabilidad de estar en el estado s.	60 % pago, 30 % red, 10 % base de datos.
$b'(s')$	Creencia actualizada tras actuar y observar.	Tras leer el log, sube la probabilidad del módulo de red.
o	Observación recibida del entorno.	El mensaje de error concreto que devuelve el sistema.
$P(o \mid s')$	Probabilidad de ver esa observación si el estado fuese s'.	Cómo de típico es ese error en cada hipótesis de fallo.
η	Constante de normalización para que las probabilidades sumen 1.	Reescala el resultado a una distribución válida.

En palabras: la nueva creencia combina lo que el agente ya pensaba, hacia dónde le lleva su acción y cómo de compatible es lo que acaba de observar con cada hipótesis; después se renormaliza para que siga siendo una distribución de probabilidad.

Esto sustituye al antiguo $s_t = (m, o, r, q)$, que fingía que el estado del agente era una fotografía exacta y cerrada de lo que sabe. El estado real de un agente que no lo ve todo no es un dato fijo: es una creencia que se actualiza con cada observación. Y esa lectura cambia la forma de diseñarlo, porque obliga a representar la incertidumbre en vez de esconderla.[7] Un agente de código no «ve» el bug directamente. Mantiene hipótesis sobre dónde está, ejecuta una acción barata para conseguir evidencia (leer el error, correr un test mínimo) y, al observar el resultado, sube la probabilidad de unas hipótesis y baja la de otras. Diagnosticar es, literalmente, actualizar un belief state hasta que una hipótesis domina lo suficiente para actuar sobre ella.

Buscar en el árbol de decisiones: MCTS

Saber que existe una utilidad esperada o una función de valor no dice cómo calcularla cuando el árbol de posibilidades es gigantesco. Los agentes modernos más potentes usan un algoritmo de búsqueda concreto para ello: *Monte Carlo Tree Search* (MCTS), búsqueda en árbol por muestreo aleatorio. Es el algoritmo detrás de AlphaGo y de varios agentes que exploran cadenas de razonamiento antes de comprometerse con una.[8] En lugar de explorar el árbol entero, MCTS construye poco a poco las ramas más prometedoras repitiendo cuatro fases.

Las cuatro fases son: *selección* (bajar por el árbol existente eligiendo acciones según un criterio), *expansión* (añadir un nodo nuevo al llegar a una rama poco explorada), *simulación* o *rollout* (jugar al azar o con una política rápida hasta el final para estimar el resultado) y

retropropagación (subir ese resultado por las ramas visitadas, actualizando sus estadísticas). Repetidas miles de veces, estas fases concentran el esfuerzo en las jugadas que de verdad importan.

La pieza clave es cómo se elige qué rama explorar en la fase de selección. MCTS usa la fórmula UCT (el criterio UCB aplicado a árboles), que equilibra explotar lo que ya parece bueno con explorar lo poco visitado:[9]

$$\mathrm{UCT}(s,a) = Q(s,a) + c \quad \frac{\ln N(s)}{N(s,a)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$Q(s,a)$	Valor medio estimado de la acción a en s (lo bueno que ha resultado).	Una jugada que ha ganado el 70 % de las simulaciones.
$N(s)$	Número de veces que se ha visitado el estado s.	Cuántas simulaciones han pasado por esta posición.
$N(s,a)$	Número de veces que se ha probado la acción a en s.	Cuántas veces se exploró esta jugada concreta.
c	Constante de exploración que regula el equilibrio.	Más alta: explora más; más baja: explota más.

En palabras: para cada acción se suma lo bueno que ha resultado hasta ahora (Q) más un premio a las acciones poco probadas (el segundo término crece cuando $N(s,a)$ es pequeño); así el algoritmo no se obsesiona con la primera opción que pareció buena ni malgasta tiempo en ramas ya descartadas.

Esto es búsqueda en árbol, igual que BFS o A* de los capítulos anteriores, pero guiada por muestreo aleatorio y estadística en vez de por una heurística fija. Encaja de lleno en este recorrido por la búsqueda, y reaparece a fondo en el capítulo 11, dedicado a los juegos, donde MCTS y la elección de jugadas entre varios actores se ven en detalle. En un agente LLM, la misma idea aparece cuando explora varias cadenas de razonamiento como ramas de un árbol, estima cuál promete más a base de pequeñas pruebas y solo entonces se compromete con un camino.

Agentes LLM: razonar y actuar

Todo lo anterior se materializa en patrones concretos de ingeniería. El primero y más extendido es *ReAct*, que intercala pensamiento y acción: el modelo escribe un razonamiento breve, decide una acción (llamar a una herramienta), observa el resultado y vuelve a razonar

con esa información nueva.[10] El bucle pensar, actuar, observar, repetir es exactamente el ciclo de un agente que actualiza su creencia tras cada observación, descrito antes con el belief state.

El segundo patrón lleva la búsqueda al razonamiento mismo. *Tree of Thoughts* trata las distintas líneas de razonamiento como ramas de un árbol: el modelo genera varias continuaciones posibles de un razonamiento, evalúa cuáles prometen y explora las mejores, descartando las que llevan a callejones sin salida.[11] Es la misma búsqueda en árbol de la sección anterior, ahora con los «estados» siendo pasos de un razonamiento en vez de posiciones de un juego.

Visto con el lenguaje de los MDP, el LLM cumple dos papeles. Funciona como una *política aprendida*: dado el estado (la conversación y las observaciones), propone la siguiente acción plausible, igual que $\pi(s)$ elige una acción en cada estado. Y a veces funciona como un *modelo de valor heurístico*: estima si una rama de razonamiento promete o no, jugando el papel de $V^*(s)$ sin calcularla de forma exacta. El LLM no es el agente entero: es el corazón que propone y evalúa dentro de un sistema mayor.

Ese sistema alrededor es ingeniería pura, y es lo que separa un buen agente de un prompt con herramientas. Añade el control de coste (presupuesto de tokens, llamadas y tiempo), los permisos (qué acciones requieren aprobación), la captura de observaciones (registrar la fuente y el contenido de cada resultado) y, sobre todo, un criterio de parada (cuándo responder, cuándo pedir ayuda, cuándo escalar). Un agente de código real funciona así: razona sobre la causa probable del fallo, actúa con una acción barata (lee el error, ejecuta un test mínimo), observa lo que devuelve, actualiza su hipótesis y repite, hasta que tiene evidencia suficiente para arreglar y verificar. Razonar, actuar, observar y repetir, dentro de un contrato que pone límites.

Diseñar una política que se pueda auditar

En ingeniería no basta con decir «el agente decide». Una decisión de agente debería poder reconstruirse después: qué sabía, qué acciones podía ejecutar, cuáles estaban bloqueadas, qué coste esperaba pagar, qué riesgo aceptaba y por qué eligió una opción concreta. Si no puedes reconstruir esa historia, tienes una caja negra operativa. Por eso conviene separar lo que se *puede* ejecutar de lo que *conviene* ejecutar, y dejar traza de ambas decisiones.

La primera capa, filtrar las acciones que no son válidas en este momento, tiene nombre propio en el campo: *action masking* (enmascarado de acciones), una técnica estándar en aprendizaje por refuerzo para impedir que el agente siquiera considere acciones imposibles o prohibidas. Cuando esas prohibiciones son restricciones duras que nunca deben violarse (no tocar producción sin evidencia, no consultar datos personales sin permiso), el marco formal es el *constrained MDP* (MDP con restricciones), un MDP donde, además de maximizar la recompensa, hay límites que no se pueden cruzar. La idea de fondo es la misma que verás en el próximo capítulo, sobre restricciones: hay cosas que no se negocian con una puntuación.

La segunda capa, ordenar las acciones permitidas, es una utilidad esperada simplificada. En la práctica se puntúa cada acción combinando su coste, su avance esperado hacia la meta y su riesgo, y se elige la mejor. Eso no es una fórmula propia: es una forma concreta y manejable de la utilidad esperada de la primera sección, con la utilidad expresada como un valor a minimizar (coste y riesgo suman, el avance resta). Conviene mantener separadas las dos capas y dejar registro de todo:

CAPA	PREGUNTA	EJEMPLO
Elegibilidad (*action masking*)	¿Se puede ejecutar esta acción ahora?	No editar producción sin evidencia; no consultar datos personales sin permiso.
Ranking (utilidad simplificada)	Entre las acciones permitidas, ¿cuál conviene primero?	Leer consola cuesta poco y reduce incertidumbre: buena primera acción.
Parada	¿Cuándo dejo de buscar?	Responder si hay evidencia suficiente; pedir aprobación si la acción siguiente es destructiva.
Trazabilidad	¿Qué debería quedar registrado?	Estado inicial, acciones candidatas, puntuación, bloqueos, observaciones y decisión.

Esta separación evita dos errores frecuentes. El primero es usar una puntuación blanda para permitir acciones que deberían estar prohibidas: si una acción requiere aprobación, no debería «ganar» por tener buena puntuación, sino quedar bloqueada hasta tener autorización. El segundo es esconder las razones de la política dentro de un prompt: puedes usar un LLM como parte del sistema, pero el contrato operativo (presupuestos, permisos, formato, evidencias mínimas y criterios de parada) debe existir fuera del texto improvisado. Esta trazabilidad también sirve para depurar: si el agente gastó diez llamadas antes de leer el error principal, el problema no era «el modelo», era la política; si editó código antes de mirar una prueba mínima, el problema no era «la IA», era que faltaba una precondición de evidencia.

Un ejemplo trazado del ranking

Imagina la incidencia «la página de checkout falla». El agente tiene cuatro acciones candidatas. Primero se aplica la elegibilidad (el *action masking*) y después se ordenan las elegibles por una puntuación que combina coste, avance estimado y riesgo (menor es mejor). Esa puntuación es una utilidad negativa simplificada: una forma concreta de la utilidad esperada de la primera sección, no una fórmula propia del capítulo. Llamamos a sus tres componentes coste, avance y riesgo:

ACCIÓN	ELEGIBLE	COSTE	AVANCE	RIESGO	PUNTUACIÓN
Leer la consola del navegador	sí	1	2	0	**3**
Ejecutar el test del checkout	sí	3	2	0	5
Preguntar al usuario	sí	2	4	0	6
Editar el código del pago	no (bloqueada)	2	1	10	no se rankea

Fíjate en la trampa. «Editar el código del pago» tiene el menor componente de avance pendiente (parece la que más acerca a la meta), y un agente puramente codicioso la elegiría. Pero su riesgo es enorme, porque tocaría producción sin evidencia, así que la capa de elegibilidad la bloquea antes de rankear nada. Entre las acciones permitidas gana «leer la consola» con puntuación 3: es barata y reduce mucho la incertidumbre. La elegibilidad va siempre antes que el ranking, y combinar coste, avance y riesgo evita que lo aparentemente directo se imponga a lo prudente. Es exactamente la clase de decisión que un agente bien diseñado debe poder justificar.

En el día a día

Imagina un agente de código que recibe: “la página falla al cargar”. Puede hacer muchas cosas. Leer el error de consola. Abrir el navegador. Buscar en el repositorio. Ejecutar tests. Mirar el último commit. Preguntar al usuario. Todas son acciones válidas, pero no todas son igual de buenas.

Una política Greedy elegiría lo que parece más directo: “abrir el navegador”. Puede funcionar. Pero si no mira los logs, quizá pierda diez minutos en la pantalla equivocada. Una política tipo A* ponderaría el coste y la información esperada: “leer el error de consola cuesta poco y reduce mucho la incertidumbre; hagamos eso primero”.

En producción, esta diferencia se traduce en dinero y fiabilidad. Un agente que llama a diez herramientas para resolver una pregunta sencilla no solo es más lento: también introduce más puntos de fallo. Un agente que responde demasiado pronto parece rápido, pero puede inventar. El buen diseño está en equilibrar información, coste y riesgo.[1][2]

Por qué debería importarte

La búsqueda clásica te da un vocabulario para diseñar agentes modernos sin confundir una salida fluida con una decisión controlada. Si defines mal el estado, el agente se pierde. Si defines mal las acciones, no puede llegar a la solución. Si ignoras el coste, se vuelve caro. Si

ignoras el riesgo, se vuelve peligroso. Si la heurística es mala, parece inteligente mientras da vueltas.

Esto es especialmente importante porque los LLMs son convincentes. Pueden explicar una decisión con mucha seguridad aunque la decisión sea mala. La estructura de búsqueda te obliga a preguntar: ¿qué estado tenía?, ¿qué acciones consideró?, ¿qué coste pagó?, ¿qué evidencia observó?, ¿por qué eligió esa acción y no otra?

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Llamar “agente” a cualquier prompt con herramientas	Tener tools no basta. Un agente necesita estado, acciones, criterio de decisión y actualización tras observar resultados.	Dibuja s_t, $A(s_t)$, a_t y s_{t+1}. Si no puedes, todavía no hay diseño de agente.
Hacer tool selection puramente codiciosa	La herramienta obvia puede ser cara, arriesgada o prematura.	Mete el coste y el riesgo dentro de la utilidad esperada, no solo el avance hacia la meta.
No modelar el coste en tokens y latencia	Un agente puede ser correcto y aun así inviable si tarda demasiado o cuesta demasiado.	Define presupuesto antes de ejecutar: tokens máximos, llamadas máximas y tiempo máximo.
Confundir observación con verdad	Una tool puede devolver datos incompletos, antiguos o mal interpretados.	Guarda la fuente de cada observación y verifica las decisiones importantes.
Recapitular sin comprobar	Leer “ya lo entiendo” no equivale a poder reconstruirlo.	Usa las preguntas de revisión activa; si fallas una, vuelve al capítulo concreto.

Antes de seguir: la búsqueda en limpio

El próximo capítulo cambia de herramienta: pasamos de buscar caminos a satisfacer restricciones (CSP). Antes de ese salto conviene tener firmes los cuatro capítulos de búsqueda. Esto no es un resumen para leer rápido, sino una revisión activa: si una idea no te sale, vuelve al capítulo indicado y sigue.

1. Búsqueda como espacio de estados

El concepto. Un problema de búsqueda se define con estado inicial, acciones, función de transición, prueba de meta y coste. Si una de esas piezas está borrosa, el algoritmo no tiene dónde agarrarse.[13]

Para recordar. El estado no es “todo lo que existe”: es solo la información necesaria para decidir la siguiente acción. Meter información irrelevante infla el espacio de búsqueda.

Ejemplo fresco. En un laberinto, el estado puede ser la coordenada (x, y). No necesitas guardar el color de la pared ni la hora del día. En un agente de código, el estado puede incluir error, archivos leídos y tests ejecutados; no necesita recordar cada token de una conversación si ya tiene un resumen fiable.

Vuelve al capítulo 1 si: no puedes definir estado, acción, meta y coste para un problema cotidiano.

2. BFS, DFS, UCS e IDS

El concepto. La frontera determina el algoritmo. Cola FIFO produce BFS. Pila LIFO produce DFS. Cola de prioridad por $g(n)$ produce UCS. DFS con límites crecientes produce IDS.[14]

Para recordar. BFS es completo y óptimo con costes uniformes, pero consume memoria $O(b^d)$. DFS consume mucha menos memoria, $O(b \cdot m)$, pero no garantiza optimalidad. UCS generaliza BFS cuando las acciones cuestan distinto. Y cuando se conoce la meta y se puede buscar hacia atrás, la búsqueda bidireccional baja el coste de $O(b^d)$ a $O(b^{d/2})$.

Ejemplo fresco. Si todas las calles pesan igual, BFS encuentra el camino con menos cruces. Si unas calles son autopistas y otras caminos lentos, necesitas UCS. Si el espacio es enorme y solo quieres no quedarte sin memoria, IDS es el puente.

Vuelve al capítulo 2 si: no puedes explicar por qué cambiar cola por pila cambia completamente el comportamiento.

3. Greedy, A* y heurísticas

El concepto. Greedy usa $f(n) = h(n)$. A* usa $f(n) = g(n) + h(n)$. La heurística $h(n)$ es una estimación de lo que falta, y su calidad decide cuántos estados se exploran.[15]

Para recordar. Greedy es rápido porque ignora el coste acumulado. A* es más disciplinado porque suma lo que ya pagaste y lo que estimas que falta. Si la memoria aprieta, IDA* da la optimalidad de A* con la memoria de DFS; si prima la velocidad, Weighted A* la cambia por una solución aproximada. Con h admisible, A* conserva garantías de optimalidad.[16]

Ejemplo fresco. En una rejilla, Manhattan puede decirte cuántos pasos mínimos faltan si solo te mueves en horizontal y vertical. Esa estimación no resuelve el problema, pero evita explorar media ciudad.

Vuelve al capítulo 3 si: no puedes explicar admisibilidad, consistencia y dominancia de heurísticas.

---

4. Agentes modernos

El concepto. Un agente moderno selecciona acciones en un estado parcialmente observado, ejecuta herramientas, recibe observaciones y actualiza su estado. La búsqueda puede ser implícita, pero sigue estando ahí.

Para recordar. "LLM + tools" no es automáticamente un buen agente. Hace falta una política: cuándo usar herramientas, cuáles priorizar, cuándo parar, cuándo pedir aclaración y cuándo responder.

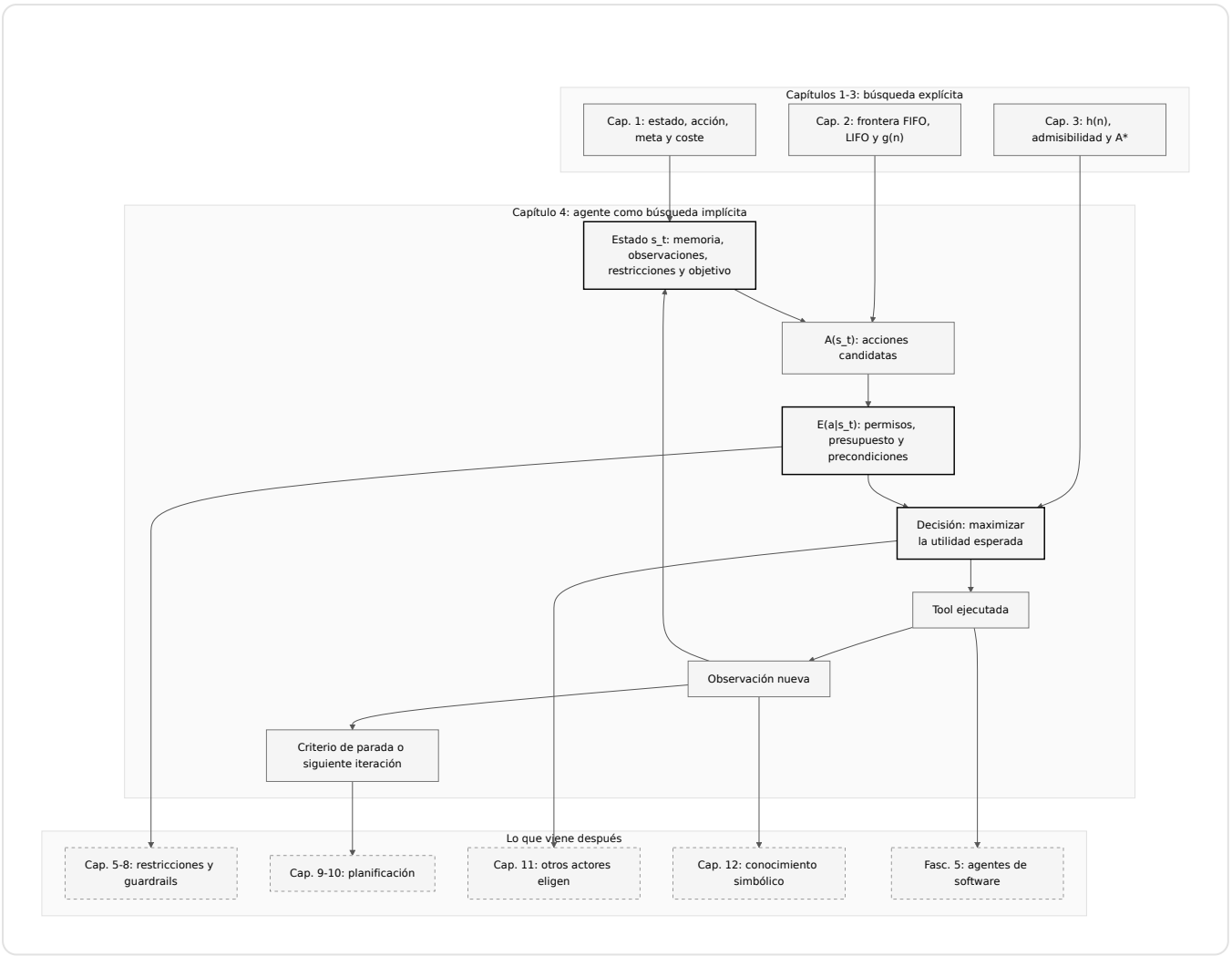
Ejemplo fresco. Ante un fallo de build, un agente prudente no edita a ciegas. Primero lee el error, identifica el archivo probable, ejecuta el test mínimo, modifica, y vuelve a verificar. Eso es búsqueda guiada por evidencia.

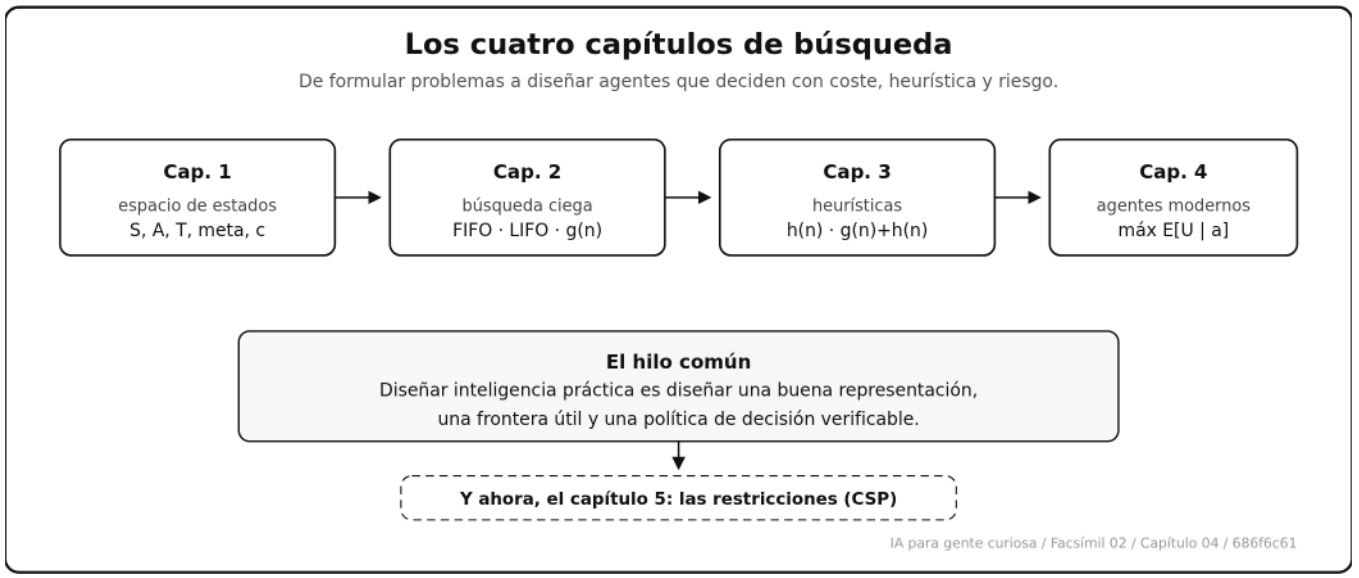
Vuelve a este capítulo si: no puedes mapear un agente a s_t, $A(s_t)$, a_t, observación y s_{t+1}.

Cómo encaja todo

Con este capítulo se cierran los cuatro primeros, todos sobre búsqueda, y el capítulo 5 cambia de pregunta. Los capítulos 1, 2 y 3 nos dieron el lenguaje clásico (estado, frontera, coste real y heurística) y aquí lo hemos usado para leer un agente moderno sin caer en la fantasía de que todo ocurre dentro del modelo. El capítulo 5 reutiliza ese mismo lenguaje, pero la pregunta deja de ser «qué camino» y pasa a ser «qué asignación cumple todas las reglas».

La decisión nueva es separar propuesta, elegibilidad, puntuación, ejecución y observación. Esa separación volverá en restricciones, planificación, agentes de software y operación.


Capítulos 1-3: búsqueda explícita
Cap. 1: estado, acción, meta y coste
Cap. 2: frontera FIFO, LIFO y g(n)
Cap. 3: h(n), admisibilidad y A*
Capítulo 4: agente como búsqueda implícita
Estado s_t: memoria, observaciones, restricciones y objetivo
A(s_t): acciones candidatas
E(a|s_t): permisos, presupuesto y precondiciones
Decisión: maximizar la utilidad esperada
Tool ejecutada
Observación nueva
Criterio de parada o siguiente iteración
Lo que viene después
Cap. 5-8: restricciones y guardrails
Cap. 9-10: planificación
Cap. 11: otros actores eligen
Cap. 12: conocimiento simbólico
Fasc. 5: agentes de software


Los cuatro capítulos de búsqueda
De formular problemas a diseñar agentes que deciden con coste, heurística y riesgo.
Cap. 1
espacio de estados
S, A, T, meta, c
Cap. 2
búsqueda ciega
FIFO · LIFO · g(n)
Cap. 3
heurísticas
h(n) · g(n)+h(n)
Cap. 4
agentes modernos
máx E[U | a]
El hilo común
Diseñar inteligencia práctica es diseñar una buena representación,
una frontera útil y una política de decisión verificable.
Y ahora, el capítulo 5: las restricciones (CSP)
IA para gente curiosa / Facsímil 02 / Capítulo 04 / 686f6c61

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Agente de búsqueda	Sistema que elige acciones para avanzar desde un estado hacia una meta.
Agente racional	Sistema que elige la acción que maximiza su medida de rendimiento esperada dada la evidencia.
Utilidad esperada	Valor medio de los resultados de una acción, ponderado por la probabilidad de cada uno.
MDP	Proceso de Decisión de Markov: marco (S, A, P, R, γ) para decidir bajo incertidumbre con recompensas en el tiempo.
Política	Regla $\pi(s)$ que asigna a cada estado la acción a tomar.
Función de valor	Recompensa total esperada de actuar de forma óptima desde un estado en adelante.
Ecuación de Bellman	Relación recursiva que liga el valor de un estado con el valor de los estados siguientes.
POMDP	MDP parcialmente observable: el agente no ve el estado real, solo observaciones.
Belief state	Distribución de probabilidad sobre los estados posibles, actualizada con cada observación.
MCTS	Monte Carlo Tree Search: búsqueda en árbol por muestreo con selección, expansión, simulación y retropropagación.
ReAct	Patrón de agente LLM que intercala razonamiento y acciones sobre herramientas.
Tool selection	Elección de la siguiente herramienta o acción que conviene ejecutar.
Coste operacional	Tokens, latencia, dinero, riesgo y complejidad acumulados al actuar.
Heurística aprendida	Estimación producida por un modelo o regla para priorizar acciones.
Política de decisión	Regla que transforma estado y acciones disponibles en una acción concreta.
Criterio de parada	Condición que decide si el agente responde, pide aprobación, sigue buscando o escala.
Trazabilidad de decisión	Registro de estado, acciones candidatas, bloqueos, puntuaciones y observaciones.

Antes de pasar página

- [ ] ¿Puedo escribir la fórmula de la utilidad esperada y explicar qué es un agente racional? (Si no, vuelve a «El agente racional».)
- [ ] ¿Sé qué cinco elementos forman un MDP y qué dice la ecuación de Bellman? (Si no, vuelve a «Decidir cuando el resultado es incierto: el MDP».)
- [ ] ¿Entiendo por qué un agente LLM trabaja con un belief state y no con un estado exacto? (Si no, vuelve a «Cuando no se ve todo: POMDP y belief state».)
- [ ] ¿Puedo nombrar las cuatro fases de MCTS y para qué sirve la fórmula UCT? (Si no, vuelve a «Buscar en el árbol de decisiones: MCTS».)
- [ ] ¿Sé qué hacen ReAct y Tree of Thoughts y cómo el LLM actúa como política y como valor? (Si no, vuelve a «Agentes LLM: razonar y actuar».)
- [ ] ¿Puedo separar acciones bloqueadas de acciones elegibles antes de rankear? (Si no, vuelve a «Diseñar una política que se pueda auditar».)
- [ ] ¿Puedo recapitular BFS, DFS, UCS, Greedy y A* sin mirar? (Si no, vuelve a «Antes de seguir: la búsqueda en limpio».)
- [ ] ¿Entiendo por qué el siguiente capítulo, CSP, sigue siendo búsqueda pero con restricciones? (Si no, vuelve a «Cómo encaja todo».)

En resumen

IDEA FUERZA	DETALLE
Los agentes modernos no eliminan la búsqueda: la esconden.	Cada decisión de tool es una elección entre acciones candidatas.
El estado decide lo que el agente puede razonar.	Si el estado omite información relevante, la política tomará malas decisiones.
La utilidad esperada es la brújula del agente.	Maximizar $\mathbb{E}[U]$, con coste, avance y riesgo dentro de la utilidad, da agentes menos impulsivos que un Greedy puro.
La elegibilidad va antes que el ranking.	Una acción prohibida o sin permisos no debería ganar por tener buen score.
La búsqueda queda recogida; el capítulo 5 cambia a las restricciones.	Estados, frontera, coste, heurística y política quedan listos para CSP (cap. 5), planificación y juegos.

Para saber más

Bellman, R. (1957). *Dynamic programming*. Princeton University Press.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. y Colton, S. (2012). A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1), 1-43. https://doi.org/10.1109/TCIAIG.2012.2186810

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

Kocsis, L. y Szepesvári, C. (2006). Bandit based Monte-Carlo planning. En *Machine Learning: ECML 2006* (pp. 282-293). Springer. https://doi.org/10.1007/11871842_29

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

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.

Puterman, M. L. (1994). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons. https://doi.org/10.1002/9780470316887

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/

Sutton, R. S. y Barto, A. G. (2018). *Reinforcement learning: an introduction* (2.ª ed.). MIT Press. https://incompleteideas.net/book/the-book-2nd.html

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y. y Narasimhan, K. (2023). *Tree of thoughts: deliberate problem solving with large language models*. https://arxiv.org/abs/2305.10601

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. y Cao, Y. (2023). *ReAct: synergizing reasoning and acting in language models*. International Conference on Learning Representations. https://arxiv.org/abs/2210.03629

Notas

1. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. El capítulo 2 define los agentes racionales como sistemas que seleccionan acciones para maximizar una medida de rendimiento, una idea que conecta directamente con la búsqueda como selección de acciones. ↩

2. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. El capítulo 2 define al agente racional como el que elige la acción que maximiza el valor esperado de su medida de rendimiento, dada la secuencia de percepciones hasta ese momento. Es la definición de referencia del campo. ↩

3. Bellman, R. (1957). *Dynamic programming*. Princeton University Press. Bellman introdujo la programación dinámica y la ecuación recursiva que lleva su nombre, base de casi toda la teoría de decisión secuencial. ↩

4. Puterman, M. L. (1994). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons. Tratado de referencia sobre la formulación y resolución de los MDP. ↩

5. Sutton, R. S. y Barto, A. G. (2018). *Reinforcement learning: an introduction* (2.ª ed.). MIT Press. Texto canónico del aprendizaje por refuerzo, donde los MDP son el marco formal de partida. ↩

6. Puterman, M. L. (1994). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons. La observabilidad parcial extiende el MDP estándar añadiendo un espacio de observaciones y la actualización de creencias. ↩

7. Sutton, R. S. y Barto, A. G. (2018). *Reinforcement learning: an introduction* (2.ª ed.). MIT Press. El capítulo sobre estados y aproximación discute cómo un agente con percepción limitada debe trabajar con representaciones que resumen su historia de observaciones. ↩

8. Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. y Colton, S. (2012). A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1), 1-43. https://doi.org/10.1109/TCIAIG.2012.2186810. Revisión de referencia sobre MCTS, su variante UCT y aplicaciones, incluido el juego de Go que llevó a AlphaGo. ↩

9. Kocsis, L. y Szepesvári, C. (2006). Bandit based Monte-Carlo planning. En *Machine Learning: ECML 2006* (pp. 282-293). Springer. https://doi.org/10.1007/11871842_29. Artículo que introduce UCT, la aplicación del algoritmo UCB de bandidos a la búsqueda en árbol. ↩

10. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. y Cao, Y. (2023). ReAct: synergizing reasoning and acting in language models. En *International Conference on Learning Representations*. https://arxiv.org/abs/2210.03629. Propone alternar trazas de razonamiento y acciones sobre herramientas, lo que mejora la fiabilidad frente a razonar o actuar por separado. ↩

1. Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y. y Narasimhan, K. (2023). Tree of thoughts: deliberate problem solving with large language models. https://arxiv.org/abs/2305.10601. Generaliza el razonamiento en cadena a una búsqueda en árbol sobre pasos intermedios, con evaluación y vuelta atrás. ↩

2. Luger, G. F. (2008). *Artificial intelligence: structures and strategies for complex problem solving* (6.ª ed.). Pearson. El capítulo 3 insiste en que la eficiencia de una búsqueda depende tanto de la representación del problema como de la estrategia de control. ↩

3. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. ↩

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

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

6. 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. ↩