Facsímil 02

Inteligencia clásica

Capítulo 07

Propagación, backtracking y heurísticas en CSP

686f6c61 · 2026

Capítulo 07: Propagación, backtracking y heurísticas en CSP

Entrando en el tema

En el capítulo anterior construimos un CSP pequeño: tres cursos, dos horas, dos salas y varias reglas. Sin restricciones había $4^3 = 64$ horarios candidatos. Con reglas, solo quedaban cuatro soluciones válidas.

La pregunta ahora es: ¿tenemos que probar los 64 horarios para descubrirlo? No. Esa es la belleza de los CSP. Antes de buscar, podemos **podar**. Podemos mirar las reglas y eliminar valores imposibles. Después, cuando toque probar, podemos hacerlo con cabeza: elegir primero la variable más estrecha, detectar contradicciones temprano y volver atrás sin dramatismo.

Este capítulo explica tres ideas que convierten un CSP ingenuo en un método práctico: propagación, *backtracking* y heurísticas.

No queremos probarlo todo

Imagina que estás rellenando un sudoku. No pruebas números al azar hasta completar la cuadrícula. Primero miras filas, columnas y bloques. Si en una casilla solo puede ir un 7, lo escribes. Si al poner un 7 otra casilla pierde esa opción, actualizas. Solo cuando ya no puedes deducir más, pruebas.

Eso es exactamente lo que hacen los CSP bien resueltos: deducir antes de explorar. Dechter lo resume como procesamiento de restricciones: reducir dominios, detectar inconsistencias y combinar inferencia local con búsqueda.[1]

ESTRATEGIA	QUÉ HACE	IMAGEN COTIDIANA
Propagación	Elimina valores imposibles antes de probar.	“Python solo puede ir a las 10:00; borra las 9:00”.
Backtracking	Prueba una opción y vuelve atrás si contradice algo.	“Si esta sala causa conflicto, deshaz y prueba otra”.
Heurísticas	Decide qué variable y qué valor probar primero.	“Empieza por quien tiene menos disponibilidad”.

La clave es no confundir inteligencia con fuerza bruta. A veces el algoritmo parece listo porque en realidad evita mirar tonterías.

Propagación: borrar antes de buscar

Propagar significa usar restricciones para reducir dominios. Si una regla dice que Python solo puede ir a las 10:00, no tiene sentido conservar valores a las 9:00 en el dominio de Python.

Podemos escribirlo así:

$$D_i' = \{v \in D_i \mid C_k(X_i = v) = \text{verdadero}\}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
D_i	Dominio original de la variable X_i.	Python: $\{(9, A), (9, B), (10, A), (10, B)\}$.
D_i'	Dominio reducido después de aplicar una restricción.	Python: $\{(10, A), (10, B)\}$.
v	Valor candidato dentro del dominio.	$(9, A)$.
$C_k(X_i = v)$	Restricción evaluada al asignar v a X_i.	“La hora de Python es 10”.

Con nuestro ejemplo:

VARIABLE	DOMINIO INICIAL	REGLA APLICADA	DOMINIO TRAS PROPAGAR
IA	$(9, A), (9, B), (10, A), (10, B)$	Ninguna unaria	$(9, A), (9, B), (10, A), (10, B)$
Python	$(9, A), (9, B), (10, A), (10, B)$	Python a las 10	$(10, A), (10, B)$
Datos	$(9, A), (9, B), (10, A), (10, B)$	Datos en sala B	$(9, B), (10, B)$

Solo con dos reglas unarias hemos pasado de $4 \times 4 \times 4 = 64$ combinaciones a $4 \times 2 \times 2 = 16$. Todavía no hemos buscado. Solo hemos borrado valores imposibles.

Consistencia de arco: cada valor necesita apoyo

La propagación más interesante aparece cuando una restricción conecta dos variables. Mackworth formuló la consistencia de arco como una manera de limpiar dominios mirando si cada valor tiene “apoyo” en la variable vecina.[2]

Un arco (X_i, X_j) es consistente si:

$$\forall x \in D_i,\ \exists y \in D_j :\ C_{ij}(x, y) = \text{verdadero}$$

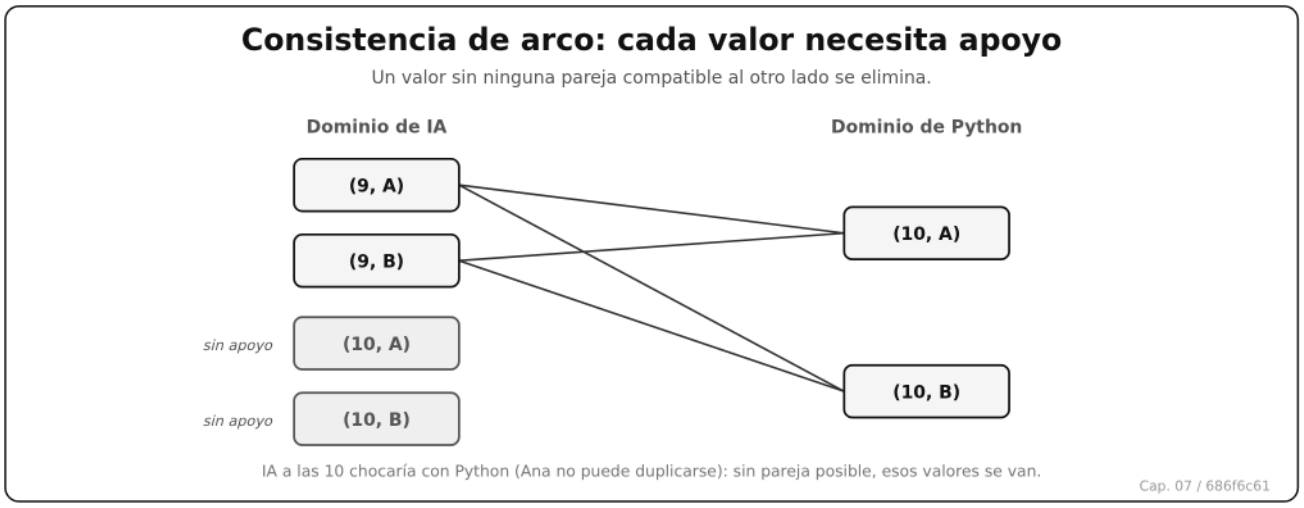
SÍMBOLO	SIGNIFICADO	EJEMPLO
X_i, X_j	Variables conectadas por una restricción.	IA y Python.
$x \in D_i$	Valor candidato para X_i.	IA a las 10:00 en sala A.
$y \in D_j$	Valor candidato para X_j.	Python a las 10:00 en sala B.
$C_{ij}(x, y)$	Restricción entre ambas variables.	IA y Python no pueden tener la misma hora.
$\exists y$	Existe al menos un valor compatible al otro lado.	IA a las 9:00 sí tiene apoyo.

Como Python ya solo puede ir a las 10:00, cualquier valor de IA a las 10:00 deja de tener apoyo. Si IA va a las 10:00, Ana tendría IA y Python a la vez. Por tanto, se elimina.

VALOR DE IA	¿HAY ALGÚN VALOR DE PYTHON COMPATIBLE?	ACCIÓN
$(9, A)$	Sí: Python puede ir a $(10, A)$ o $(10, B)$.	Se conserva.
$(9, B)$	Sí: Python puede ir a $(10, A)$ o $(10, B)$.	Se conserva.
$(10, A)$	No: Python siempre va a las 10:00.	Se elimina.
$(10, B)$	No: Python siempre va a las 10:00.	Se elimina.

Después de esta limpieza:

$$D'_{\text{IA}} = \{(9, A), (9, B)\}$$



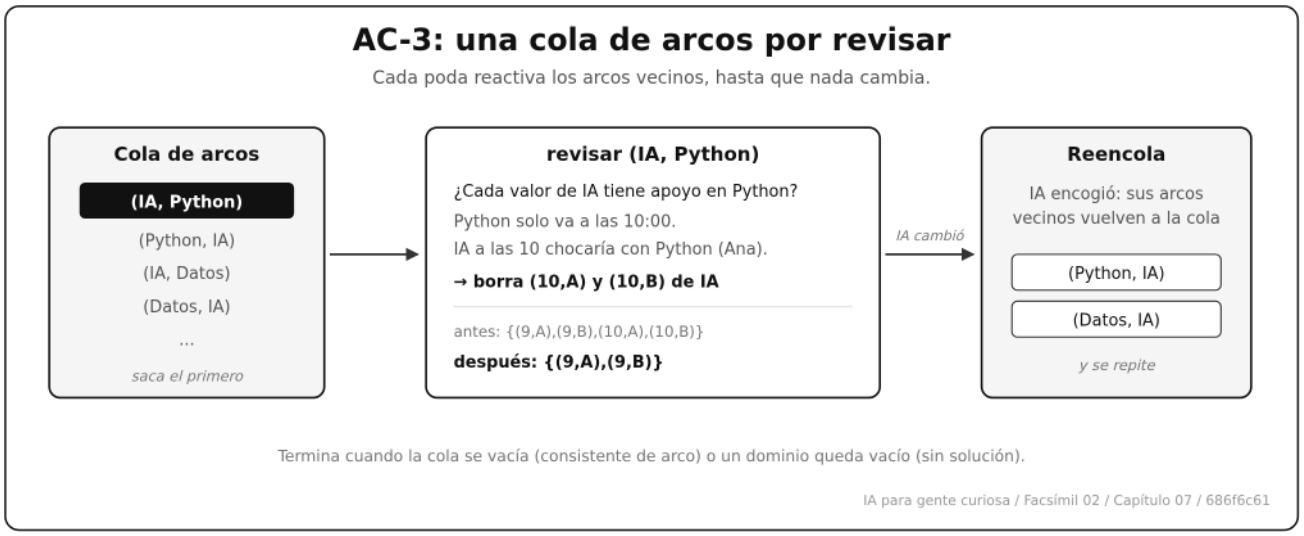
No hemos probado horarios completos. Solo hemos dicho: “si esta opción nunca puede convivir con ninguna opción vecina, fuera”.

AC-3: propagar hasta que nada cambie

Limpiar un arco una vez no basta. Cuando borras un valor de un dominio, las variables vecinas pueden perder apoyos que antes tenían, así que hay que volver a revisarlas. El algoritmo clásico que organiza esa cascada se llama **AC-3** (Mackworth, 1977) y es la receta concreta que está detrás de la consistencia de arco.[3] Funciona con una cola de arcos pendientes de revisar:

```
mete todos los arcos (Xi, Xj) en una cola
mientras la cola no esté vacía:
    saca un arco (Xi, Xj)
    si revisar(Xi, Xj) borró algún valor de Di:
        si Di queda vacío: no hay solución, para
        vuelve a meter en la cola los arcos (Xk, Xi) de los vecinos de Xi
```

`revisar(Xi, Xj)` elimina de D_i todo valor que no tenga ningún apoyo en D_j. La línea importante es la última: cada vez que un dominio encoge, los arcos que apuntan a esa variable vuelven a la cola, porque su situación ha cambiado. El proceso termina cuando la cola se vacía, y entonces todo es consistente de arco, o cuando un dominio se queda vacío, y entonces el problema no tiene solución y lo sabemos sin haber buscado. Esa repetición hasta que ya no cambia nada es lo que se llama llegar a un **punto fijo**.



AC-3 no resuelve el CSP: no asigna valores, solo poda. A veces deja los dominios tan reducidos que la solución es inmediata; a veces apenas borra nada. Por eso se combina con la búsqueda, que es la siguiente pieza.

Conviene ser honesto con lo que la propagación promete. Que un CSP sea consistente de arco **no garantiza que tenga solución**: puedes dejar todos los arcos con apoyo y aun así no existir ninguna asignación completa válida, porque la contradicción vive en una combinación de tres o más variables que la consistencia de arco, que solo mira pares, no llega a ver. Detectarla pediría consistencias de orden superior, la llamada **k-consistencia**, bastante más caras de calcular.[4] En la práctica se propaga lo justo y se deja que la búsqueda descubra esas contradicciones más profundas: propagar abarata la búsqueda, no la sustituye.

Backtracking: probar, fallar, volver

La propagación no siempre resuelve todo. Cuando quedan varias opciones posibles, necesitamos buscar. El método clásico es *backtracking*: asignar una variable, comprobar consistencia y, si algo falla, deshacer.

Russell y Norvig presentan el *backtracking* como la búsqueda básica para CSP: una asignación parcial se extiende mientras siga siendo consistente; cuando no puede extenderse, se retrocede.[5]

La condición de consistencia parcial se puede escribir así:

$$\text{consistente}(a_p) = \bigwedge_{C_k \in C_p} C_k(a_p)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
a_p	Asignación parcial.	Python $(10, A)$, IA todavía vacía.
C_p	Restricciones que ya pueden evaluarse con lo asignado.	“Python a las 10” sí; “IA distinta de Python” aún no si IA está vacía.
$C_k(a_p)$	Resultado de evaluar la restricción k.	Verdadero o falso.
$\bigwedge$	Todas las restricciones evaluables deben cumplirse.	Si una falla, se vuelve atrás.

El esquema mental es:

```
elige variable
prueba valor
si sigue siendo consistente:
    propaga consecuencias
    continúa
si aparece contradicción:
    deshaz y prueba otro valor
```

No es una caja negra. Es una búsqueda ordenada, con freno y marcha atrás.

Forward checking y MAC: cuánto propagar al buscar

Durante la búsqueda hay una decisión de diseño: cada vez que asignas una variable, ¿cuánto propagas antes de seguir? Hay un espectro entre no propagar nada y propagar a fondo.

ESTRATEGIA	QUÉ PROPAGA AL ASIGNAR $X_I = V$	COSTE POR NODO	PODA
Backtracking puro	Nada; solo comprueba consistencia con lo ya asignado.	Mínimo	La menor
Forward checking	Borra de los dominios de las variables vecinas aún sin asignar los valores incompatibles con $X_i = v$.	Bajo	Media: detecta un dominio vacío un paso antes
MAC (*maintaining arc consistency*)	Ejecuta AC-3 completo tras cada asignación, propagando en cascada.	Alto	La mayor: poda más, pero cada nodo cuesta más

Forward checking es el punto dulce habitual: barato y suficiente para cortar muchas ramas muertas. Mira solo un paso, las variables directamente conectadas con la que acabas de asignar, y si alguna se queda con el dominio vacío, retrocede sin seguir bajando. MAC va más lejos: tras cada asignación vuelve a dejar todo el grafo consistente de arco, así que detecta contradicciones más profundas, pero paga el coste de propagar en cascada en cada nodo. No hay un ganador universal: en problemas muy enredados MAC compensa; en problemas sueltos, forward checking suele ir más rápido. El lab de este capítulo usa forward checking, que es lo que se ve en la traza.

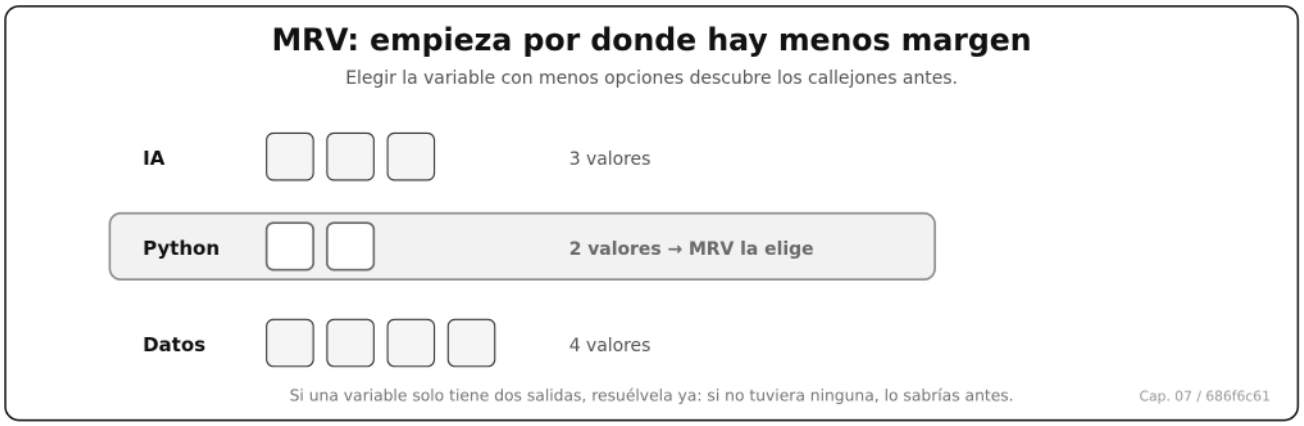
Heurísticas: fallar pronto y dejar opciones

Si hay muchas variables, el orden importa. Una buena heurística no “adivina” la solución. Lo que hace es ordenar la búsqueda para descubrir contradicciones pronto y conservar alternativas útiles.[6]

La primera heurística es MRV (*minimum remaining values*): elige la variable con menos valores legales restantes. No nos la inventamos: es el principio **fail-first** («falla primero») que Haralick y Elliott formalizaron en 1980 junto al forward checking, y dice algo casi de sentido común: si una variable va a quedarse sin opciones, mejor descubrirlo cuanto antes y no al final del árbol.[7]

$$X^* = \arg\min_{X_i \notin a_p} |D_i^{(a_p)}|$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
X^*	Variable elegida para asignar ahora.	Python, si solo tiene dos opciones.
$X_i \notin a_p$	Variable todavía no asignada.	IA o Datos si Python ya está fijada.
$D_i^{(a_p)}$	Dominio restante después de la asignación parcial.	Valores que siguen siendo legales.
$\arg\min$	Elige quien minimiza el tamaño del dominio.	“Empieza por quien tiene menos margen”.



La segunda es la heurística de **grado** (*degree*): si dos variables empatan en MRV, elige la que participa en más restricciones pendientes. Es el desempate natural del mismo repertorio clásico de heurísticas de CSP que recogen Russell y Norvig: la variable más conectada es la que, al fijarse, más poda en el resto del problema.

$$X^* = \arg\max_{X_i \notin a_p} \text{grado}(X_i)$$

Y la tercera es LCV (*least constraining value*): prueba primero el valor que menos opciones elimina a las demás. Es la cara simétrica del fail-first, ahora sobre los valores: en la *variable* conviene fallar pronto (MRV), pero en el *valor* conviene lo contrario, dejar abiertas el máximo de puertas para no encerrar la búsqueda antes de tiempo.

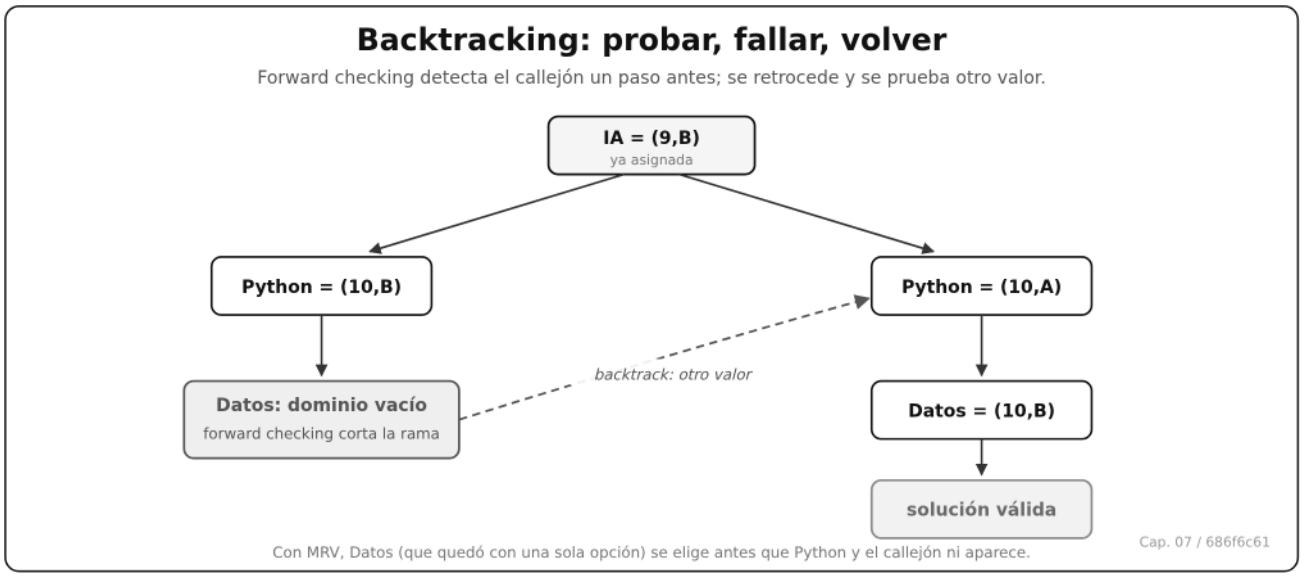
$$v^* = \arg\min_{v \in D_i} \sum_{X_j \neq X_i} \text{eliminados}(X_j \mid X_i = v)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
v^*	Valor que conviene probar primero.	IA a las 9:00 en sala A.
eliminados	Número de valores que se pierden en otra variable.	Si elegir sala A bloquea muchas opciones, es peor.
$\sum$	Suma de pérdidas sobre variables vecinas.	Total de opciones que dejamos fuera.

Luger resume estas heurísticas como conocimiento de control: no cambian las soluciones válidas, cambian el camino por el que llegas a ellas.[8]

Un backtracking paso a paso

Juntemos las piezas en el horario. Imagina que la búsqueda ya ha asignado IA $(9, B)$, lo que por forward checking deja a Datos con una sola franja libre, y ahora le toca a Python, que puede ir a $(10, A)$ o $(10, B)$. Mira lo que pasa según el valor que pruebe:



Si prueba Python $(10, B)$, el forward checking mira a Datos: como necesita sala B y ya están tomadas la franja $(9, B)$ por IA y la $(10, B)$ por Python, su dominio se queda vacío. No hace falta seguir bajando: esa rama está muerta. Se retrocede y se prueba Python $(10, A)$, que le deja a Datos la opción $(10, B)$, y la búsqueda completa una solución. Aquí es donde MRV se gana el sueldo: si hubiéramos elegido primero la variable con menos opciones, Datos, que tras asignar IA queda con una sola, nunca habríamos entrado en el callejón.

Otra vía: reparar en vez de retroceder

El backtracking construye la solución poco a poco y vuelve atrás cuando se atasca. Hay una familia de métodos que hace lo contrario: parte de una asignación completa, aunque tenga conflictos, y la **repara**. Es la búsqueda local, y su versión más conocida en CSP es **min-conflicts**.[9] La idea es sorprendentemente simple:

```
empieza con una asignación completa (por ejemplo, al azar)
mientras haya conflictos y no se agote el tiempo:
    elige una variable que esté en conflicto
    reasígnale el valor que deje el menor número de conflictos
```

No hay árbol ni marcha atrás: en cada paso se mueve una sola variable al valor menos conflictivo. Suena ingenuo, pero en problemas grandes y poco estructurados funciona asombrosamente bien. El caso famoso es el de las n-reinas: min-conflicts coloca un millón de reinas sin que se ataquen en un tiempo casi constante, algo impensable para el backtracking sistemático.

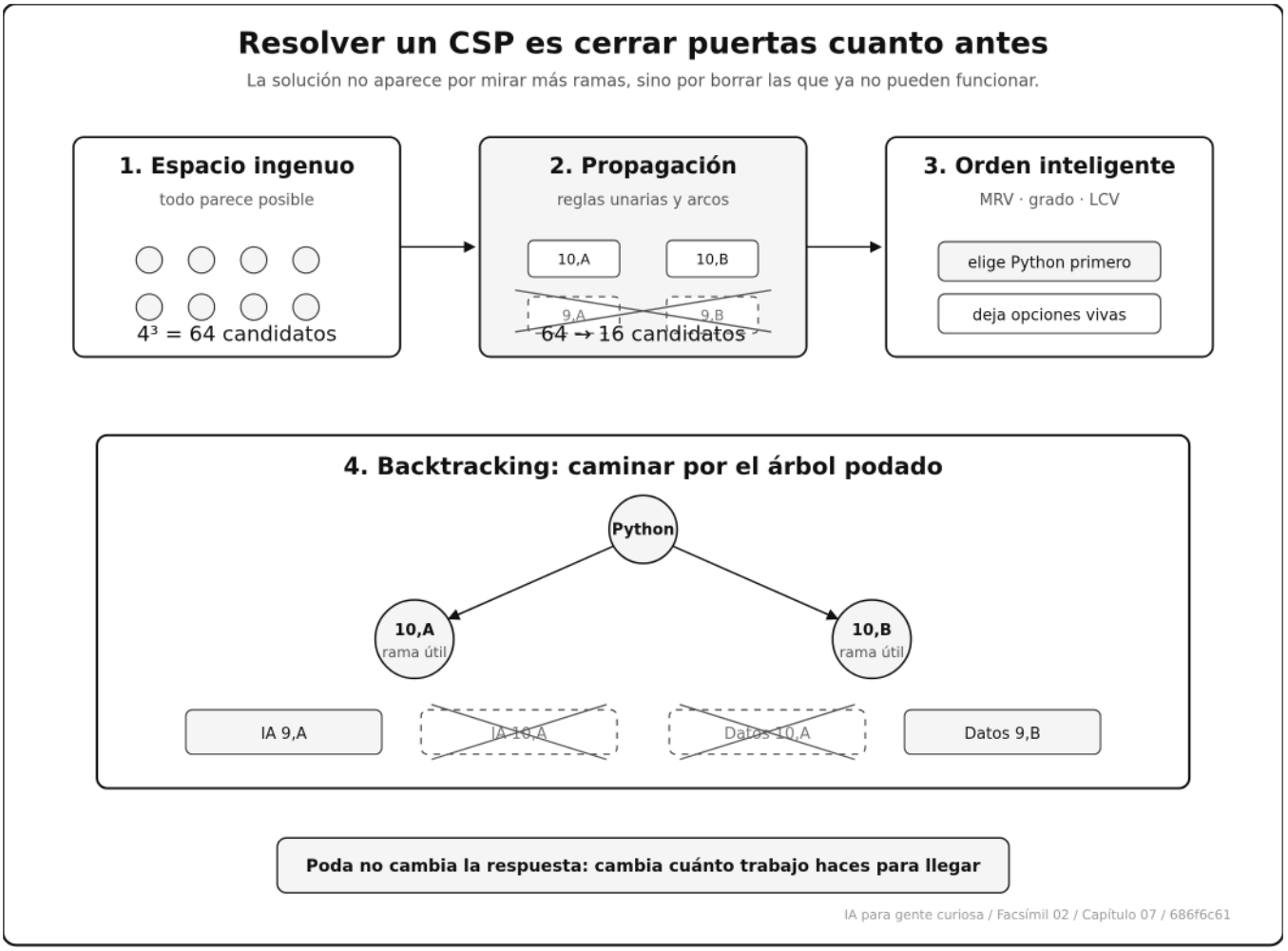
A cambio, la búsqueda local tiene dos límites que conviene decir en voz alta. Primero, es **incompleta**: si no encuentra solución, no puede afirmar que no exista, solo que ella no la halló; el backtracking sí puede demostrar la ausencia de solución recorriendo todo el árbol podado. Segundo, puede quedarse atrapada en un **mínimo local**, una asignación con pocos conflictos pero ninguno que se quite moviendo una sola variable, y por eso en la práctica se combina con reinicios aleatorios o algún paso al azar. La regla para elegir es clara: si necesitas garantía de recorrer todo el espacio o de demostrar que no hay solución, backtracking; si el problema es enorme y te basta con encontrar pronto una solución buena, búsqueda local.

Qué se mide en una búsqueda CSP seria

Si quieres saber si una estrategia de CSP mejora algo, no basta con decir “encuentra solución”. Hay que medir el trabajo que ha evitado. Dos estrategias pueden devolver las mismas cuatro soluciones y, aun así, una visitar diez nodos y otra sesenta.

MÉTRICA	QUÉ MIDE	POR QUÉ IMPORTA
Nodos visitados	Asignaciones parciales exploradas.	Aproxima el trabajo real del backtracking.
Valores podados	Opciones eliminadas de dominios futuros.	Enseña si la propagación está haciendo algo útil.
Dominios vacíos	Contradicciones detectadas pronto.	Indica dónde se corta una rama.
Profundidad máxima	Cuánto llega a comprometerse la búsqueda.	Ayuda a entender si falla pronto o tarde.
Orden de variables	Qué decide MRV o grado en cada paso.	Permite depurar heurísticas de selección.
Soluciones encontradas	Asignaciones completas válidas.	Resultado final, pero no única métrica.

Esta traza es especialmente útil cuando un CSP real no encuentra solución. Sin traza solo tienes un “no”. Con traza puedes ver si el problema está sobre-restringido, si una variable se queda sin dominio demasiado pronto o si una restricción global está eliminando casi todo.



En el día a día

En planificación de turnos, propagación es borrar turnos imposibles antes de construir el calendario: quien no trabaja sábados pierde todos los valores de sábado; quien necesita descanso tras una noche pierde la mañana siguiente.

En configuración de producto, *backtracking* aparece cuando eliges módulos compatibles. Si activar “facturación avanzada” exige “plan empresa”, y el cliente tiene “plan básico”, esa rama se corta antes de seguir configurando complementos.

En agentes con herramientas, la analogía práctica es clara: no pruebes acciones imposibles. Antes de pedirle al modelo que elija una herramienta, filtra por permisos, estado, coste y disponibilidad. Poole, Mackworth y Goebel conectan esta idea con la separación entre representación e inferencia: si el conocimiento del dominio está bien representado, el razonamiento puede descartar opciones temprano.[10]

Por qué debería importarte

Porque la diferencia entre resolver y no resolver suele estar en la poda. Dos formulaciones con las mismas soluciones pueden comportarse de forma muy distinta si una detecta contradicciones pronto y la otra las descubre al final.

En sistemas modernos, esto se traduce a coste real: menos llamadas a herramientas, menos tokens, menos latencia, menos acciones rechazadas tarde. La programación con restricciones no es una reliquia; es una forma de diseñar sistemas que no gastan energía explorando lo que ya sabemos que no puede funcionar.[11]

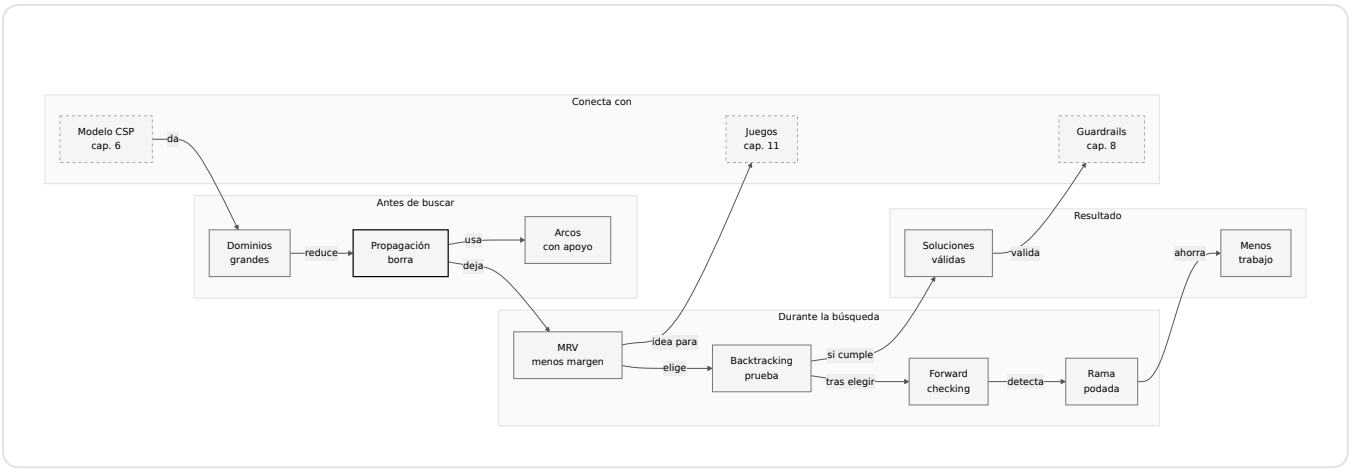
Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Propagar una sola vez y olvidarme	Al borrar valores de un dominio, otras restricciones pueden empezar a borrar nuevos valores.	Repite hasta que no cambie nada o hasta encontrar un dominio vacío.
Confundir forward checking con consistencia de arco	Forward checking mira consecuencias inmediatas de una asignación; consistencia de arco limpia apoyos entre pares de variables.	Recuerda: forward checking ocurre tras elegir; AC mira compatibilidad entre dominios.
Elegir variables en orden arbitrario	Puedes dejar el cuello de botella para el final y descubrir tarde que todo era imposible.	Usa MRV y, si hay empate, grado.
Probar primero valores muy restrictivos	Un valor que bloquea muchas opciones puede encerrar la búsqueda sin necesidad.	Usa LCV cuando conservar alternativas importe.

Cómo encaja todo

Este mapa muestra el paso de modelar a resolver. Venimos de variables, dominios y restricciones; ahora añadimos mecanismos que reducen búsqueda: propagación antes de probar, MRV para elegir dónde duele más y backtracking para volver atrás con criterio.

La idea se reutiliza después en guardrails, planificación y agentes: antes de gastar pasos caros, intenta eliminar opciones imposibles con información barata.


Conecta con
Modelo CSP cap. 6
da
Juegos cap. 11
Guardrails cap. 8
Antes de buscar
Dominios grandes
reduce
Propagación borra
usa
Arcos con apoyo
deja
Resultado
Soluciones válidas
valida
ahorra
Menos trabajo
Durante la búsqueda
MRV menos margen
idea para
elige
Backtracking prueba
si cumple
tras elegir
Forward checking
detecta
Rama podada

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Propagación	Reducción de dominios usando restricciones.
Backtracking	Probar, comprobar, avanzar y volver atrás si aparece contradicción.
Forward checking	Eliminar valores futuros incompatibles con una asignación recién tomada.
Consistencia de arco	Cada valor de una variable debe tener algún valor compatible en la variable vecina.
AC-3	Algoritmo que logra la consistencia de arco con una cola de arcos, hasta que ningún dominio cambia.
MAC	Mantener la consistencia de arco durante la búsqueda: aplicar AC-3 tras cada asignación.
MRV	Elegir primero la variable con menos valores legales restantes.
Grado	Elegir la variable que participa en más restricciones pendientes.
LCV	Probar primero el valor que menos opciones elimina a las demás variables.
Nodo de búsqueda	Asignación parcial visitada por el backtracking.
Búsqueda local	Parte de una asignación completa y la mejora reparando conflictos, sin construir un árbol.
Min-conflicts	Búsqueda local para CSP: reasigna una variable en conflicto al valor que deja menos conflictos.
k-consistencia	Consistencias de orden superior que miran k variables a la vez; más fuertes y más caras que la de arco.
Traza de propagación	Registro de decisiones, podas, dominios vacíos y soluciones.

Antes de pasar página

- [ ] ¿Puedo explicar por qué propagar no es lo mismo que buscar? (Si no, vuelve a «Propagación: borrar antes de buscar».)

- [ ] ¿Sé escribir la condición de consistencia de arco? (Si no, vuelve a «Consistencia de arco: cada valor necesita apoyo».)
- [ ] ¿Entiendo cómo AC-3 propaga con una cola de arcos hasta el punto fijo? (Si no, vuelve a «AC-3: propagar hasta que nada cambie».)
- [ ] ¿Entiendo qué hace backtracking cuando encuentra una contradicción? (Si no, vuelve a «Backtracking: probar, fallar, volver».)
- [ ] ¿Distingo forward checking de MAC y sé cuándo conviene cada uno? (Si no, vuelve a «Forward checking y MAC: cuánto propagar al buscar».)
- [ ] ¿Sé cuándo usar MRV, grado y LCV? (Si no, vuelve a «Heurísticas: fallar pronto y dejar opciones».)
- [ ] ¿Distingo cuándo conviene búsqueda local (min-conflicts) en vez de backtracking? (Si no, vuelve a «Otra vía: reparar en vez de retroceder».)
- [ ] ¿Sé leer los eventos de una traza de backtracking con MRV y forward checking? (Si no, vuelve a «Backtracking: probar, fallar, volver».)

En resumen

IDEA FUERZA	DETALLE
Propagar es borrar imposibles.	Antes de buscar, las restricciones reducen dominios.
La consistencia de arco exige apoyo.	Un valor sin valor compatible en la variable vecina se elimina.
Backtracking evita comprometerse con errores.	Si una rama contradice reglas, se deshace y se prueba otra.
Las heurísticas ordenan la búsqueda.	MRV, grado y LCV no cambian las soluciones, pero reducen trabajo inútil.
La traza convierte el solver en algo depurable.	Sin traza solo sabes si hay solución; con traza ves por qué se poda cada rama.

Para saber más

Dechter, R. (2003). *Constraint processing*. Morgan Kaufmann.

Freuder, E. C. (1978). Synthesizing constraint expressions. *Communications of the ACM*, 21(11), 958-966. https://doi.org/10.1145/359642.359654

Haralick, R. M. y Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3), 263-313. https://doi.org/10.1016/0004-3702(80)90051-X

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

Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8(1), 99-118. https://doi.org/10.1016/0004-3702(77)90007-8

Minton, S., Johnston, M. D., Philips, A. B. y Laird, P. (1992). Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3), 161-205. https://doi.org/10.1016/0004-3702(92)90007-K

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.

Rossi, F., van Beek, P. y Walsh, T. (Eds.). (2006). *Handbook of constraint programming*. Elsevier.

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

Notas

1. Dechter, R. (2003). *Constraint processing*. Morgan Kaufmann. La obra presenta la propagación y la búsqueda como dos piezas complementarias: una reduce el espacio y la otra explora lo que queda. ↩

2. Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8(1), 99-118. https://doi.org/10.1016/0004-3702(77)90007-8 ↩

3. Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8(1), 99-118. https://doi.org/10.1016/0004-3702(77)90007-8 ↩

4. Freuder, E. C. (1978). Synthesizing constraint expressions. *Communications of the ACM*, 21(11), 958-966. https://doi.org/10.1145/359642.359654 ↩

5. Russell, S. y Norvig, P. (2021). *Artificial intelligence: a modern approach* (4.ª ed.). Pearson. En el tratamiento de CSP, el backtracking aparece como búsqueda en profundidad sobre asignaciones parciales, reforzada con propagación y heurísticas. ↩

6. Pearl, J. (1984). *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley. Aunque el libro se centra en búsqueda heurística general, su idea central aplica aquí: una buena estimación reduce exploración inútil. ↩

7. Haralick, R. M. y Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3), 263-313. https://doi.org/10.1016/0004-3702(80)90051-X. El artículo introdujo el forward checking y el principio fail-first, base de las heurísticas de orden de variables en CSP. ↩

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

9. Minton, S., Johnston, M. D., Philips, A. B. y Laird, P. (1992). Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3), 161-205. https://doi.org/10.1016/0004-3702(92)90007-K ↩

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

11. Rossi, F., van Beek, P. y Walsh, T. (Eds.). (2006). *Handbook of constraint programming*. Elsevier. ↩