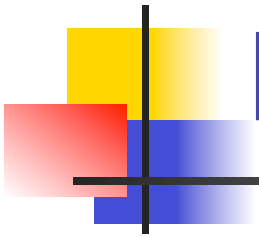
Razonamiento Automatizado

Pedro Meseguer
IIIA, CSIC
Bellaterra
pedro@iiia.csic.es



Esquema

- ¿Qué es el razonamiento automatizado?
- Algunas áreas con éxito (últimos 15 años)
  - SAT (*Boolean Satisfiability*)
  - CSP (*Constraint Satisfaction Problems*)
  - Planificación (*Planning*)
  - Juegos por ordenador (*Computer Games*)
  - Bases de datos de patrones (*Pattern Databases*)
- Algunas conclusiones

¿Qué es razonamiento automatizado?

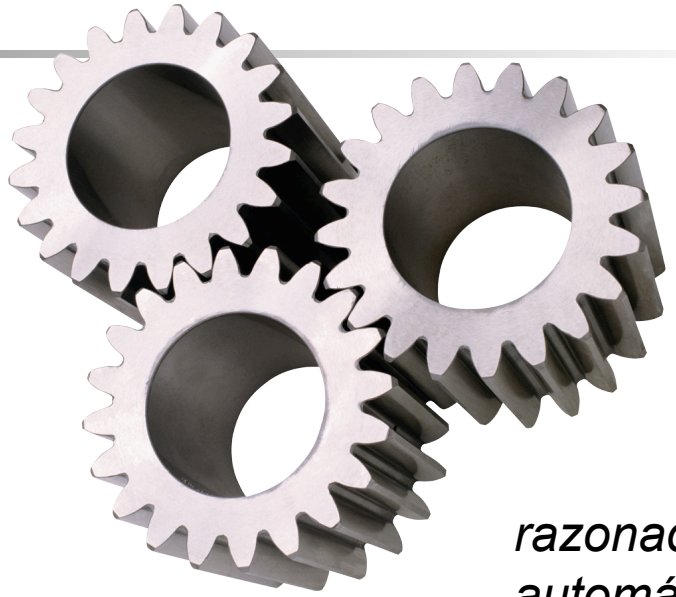
problemas reales

traducción formalismo

interpretación

manual

tema de la conferencia



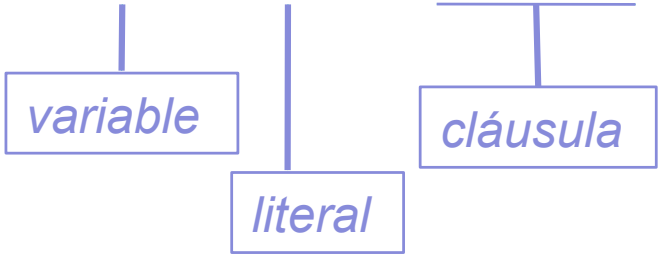
razonador automático

solución

SAT

SAT

- Dada una fórmula ϕ en lógica de proposiciones, determinar si es satisfactible o no
- ϕ en forma normal conjuntiva (CNF, cualquier fórmula se puede expresar como CNF)
- $\phi = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3)$



- SAT: encontrar asignación {1,0} a variables de ϕ, de forma que ϕ=1, o probar que tal asignación no existe(NP-completo)

¿Por qué SAT?

- Muchas aplicaciones se pueden formular como SAT:
  - Verificación de circuitos
  - Criptografía
  - Model checking

- SAT-SOLVER: Mejora de rendimiento en los últimos años

años	#variables	#cláusulas
60-70	Decenas	Cientos
80-90 (com)	Cientos	Miles
90	Miles	Cientos de miles
00	Cientos de miles	Millones

SAT: Conceptos Básicos

- Cláusula satisfecha si al menos un literal tiene valor 1
  $(x_1 \vee \neg x_2 \vee \neg x_3)$
- Cláusula insatisfecha si todos sus literales tienen valor 0
  $(x_1 \vee \neg x_2 \vee \neg x_3)$
- Cláusula unitaria si contiene un solo literal no asignado, y todos los demás literales tienen valor 0
  $(x_1 \vee \neg x_2 \vee \neg x_3)$
- Fórmula sastifecha si todas sus cláusulas lo son
- Fórmula insatisfecha si al menos una cláusula es insatisfecha

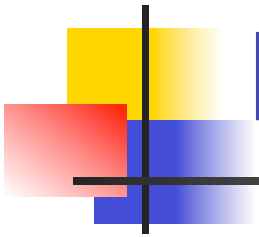
Propagación Unitaria

- Regla: el único literal no asignado de una cláusula unitaria ha de ser asignado a 1
- Propagación unitaria: iteramos la aplicación de la regla

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$$
$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$$
$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$$

- … pero también puede descubrir conflictos!

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$
$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$
$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

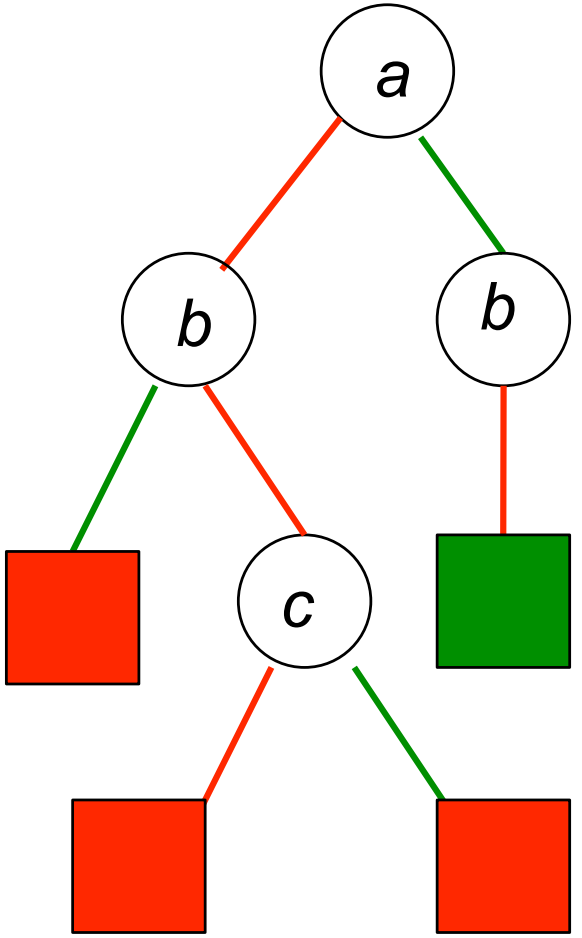


DPLL: algoritmo para SAT

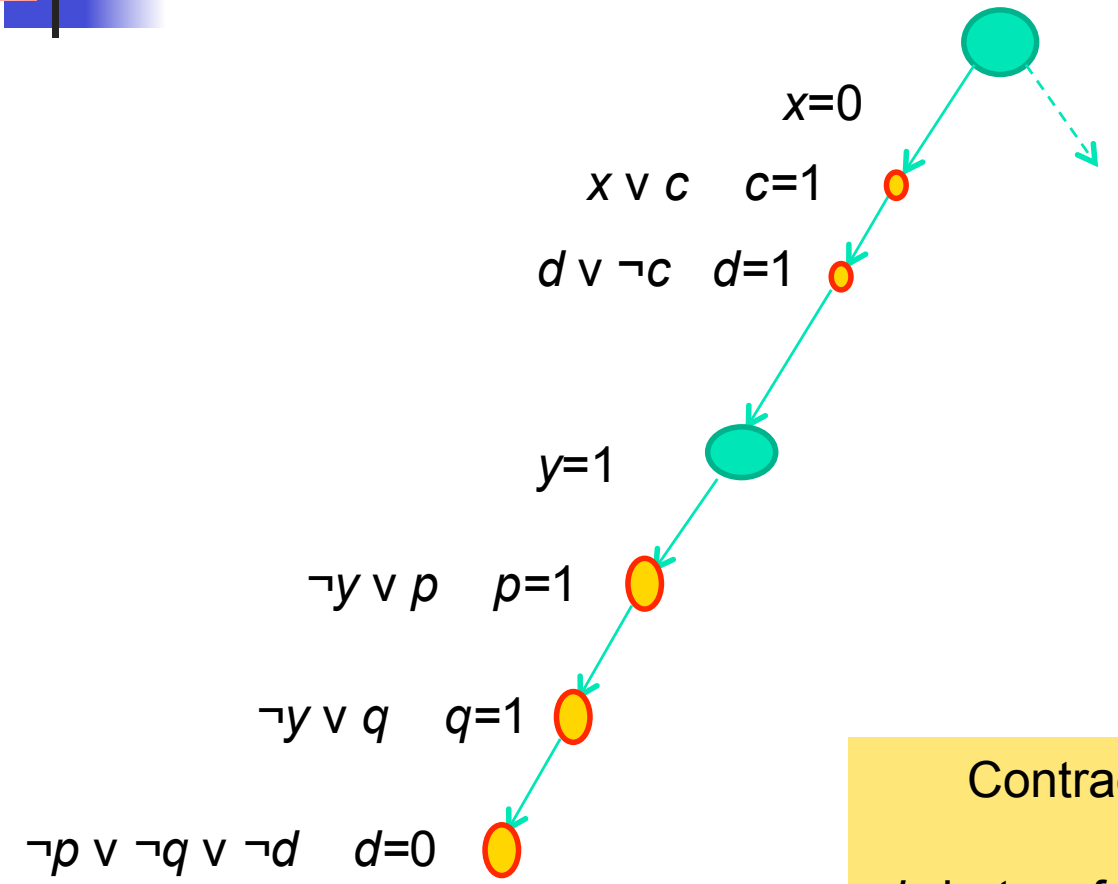
- Búsqueda primero en profundidad con backtracking
- En cada paso:
  - Selecciona asignación
  - Aplica propagación unitaria
  - Si conflicto, backtrack
    - Tras backtrack, propagación unitaria
    - Si no se puede realizar backtrack, retorna UNSAT
  - Si la fórmula satisfecha, retorna SAT
  - Sino, continua con otra asignación

DPLL: Ejemplo

$(a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge$
$(\neg b \vee \neg d \vee \neg e) \wedge (a \vee b \vee c \vee d) \wedge$
$(a \vee b \vee c \vee \neg d) \wedge (a \vee b \vee \neg c \vee e) \wedge$
$(a \vee b \vee \neg c \vee \neg e)$



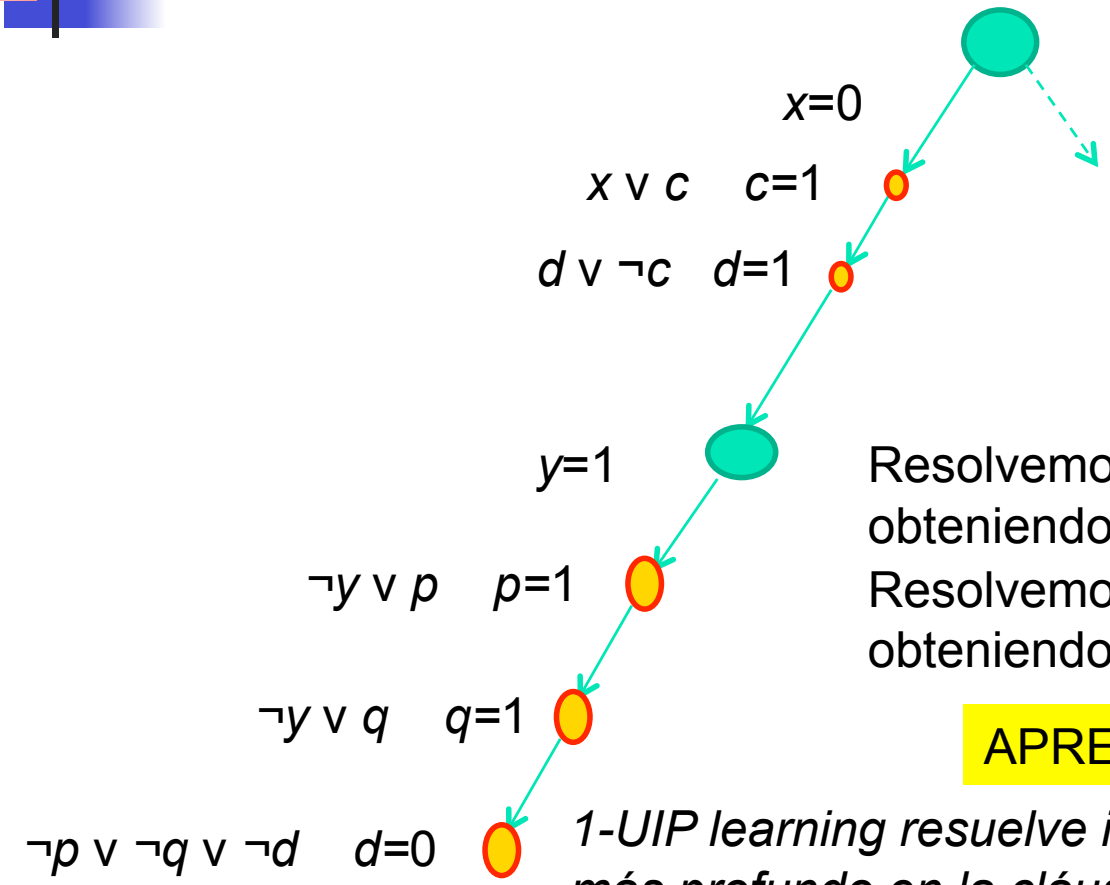
SAT: aprendizaje de cláusulas



Contradicción:

d cierto y falso a la vez

SAT: aprendizaje de cláusulas



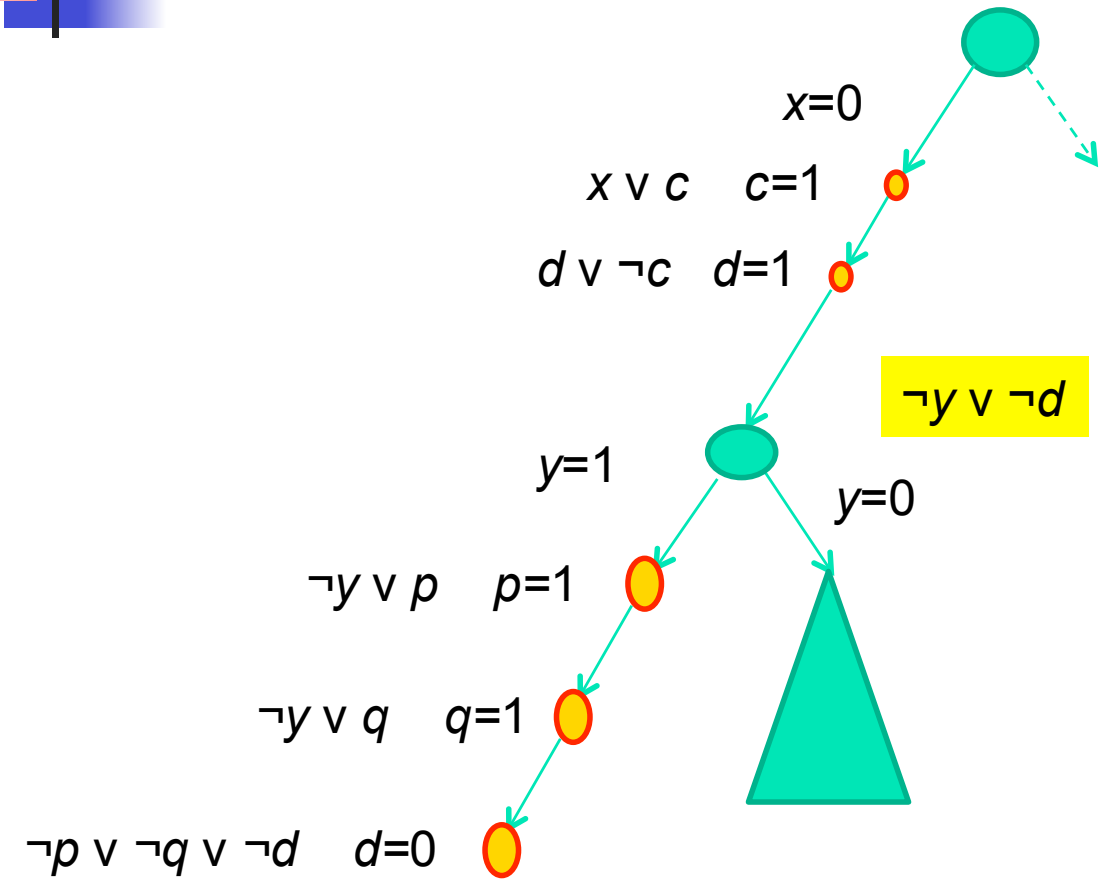
- Una nueva cláusula tras una contradicción
- Involucra una secuencia de pasos de resolución

Resolvemos $(\neg p \vee \neg q \vee \neg d)$ con $(\neg y \vee q)$
obteniendo $(\neg p \vee \neg y \vee \neg d)$
Resolvemos $(\neg p \vee \neg y \vee \neg d)$ con $(\neg y \vee p)$
obteniendo $(\neg y \vee \neg d)$

APRENDEMOS $(\neg y \vee \neg d)$

1-UIP learning resuelve iterativamente el literal implicado más profundo en la cláusula falsa, hasta que esta contiene sólo un literal del nivel en que se detectó la contradicción

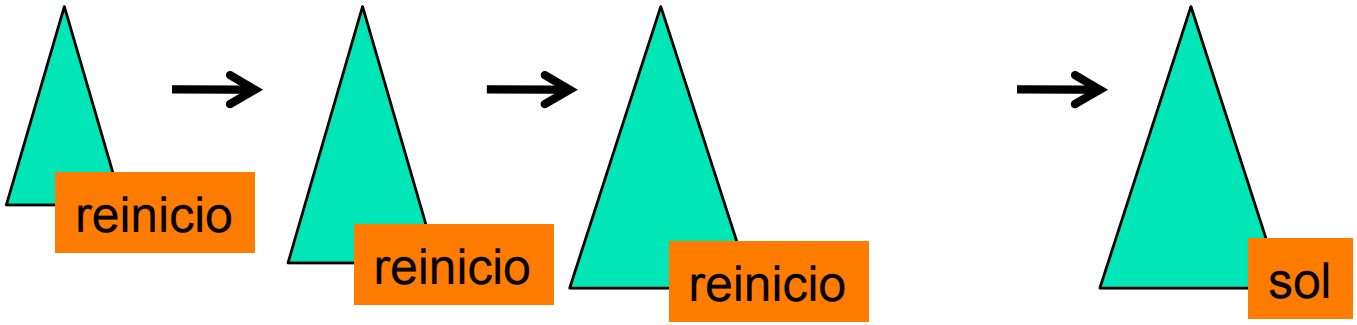
SAT: backtracking no cronológico



- La nueva cláusula es incondicional
- Tras aprender la nueva cláusula, se hace backtracking al nivel en donde esa cláusula es unitaria (como si se hubiera conocido la cláusula al principio)
- Se fuerza el literal
- Se continua la búsqueda

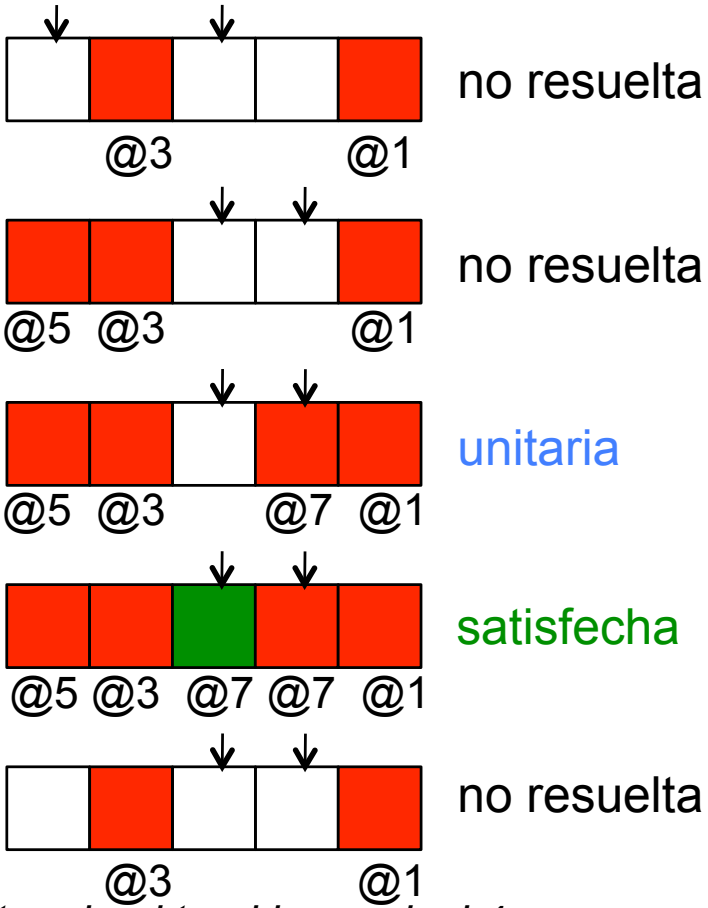
SAT: Reinicios (*Restarts*)

- 50% ejecuciones < 1.000 backtracks
  pequeño % > 10.000 backtracks
- Reinicio: abandona la búsqueda actual y comienza de nuevo



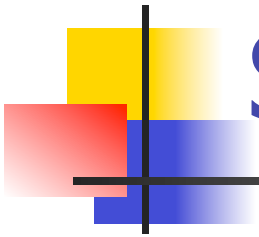
- Mantiene las cláusulas aprendidas antes del reinicio
- Ayuda a reducir la variancia y añade robustez al Solver

SAT: *Two-watched literals*



tras backtracking a nivel 4

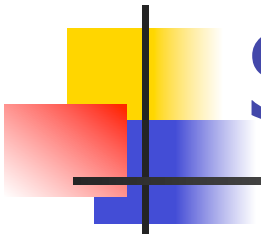
- Por cada cláusula 2 *watched literals* (para detectar cuando se vuelve unitaria)
- Cuando se asigna var *x*, sólo los *watched literals* de *x* han de ser evaluados
- Si hay *backtracking*, no se ha de hacer nada



SAT: Heurísticas

- VSIDS
  - Ordena variables por num literales en las cláusulas iniciales
  - Incrementa contadores por nuevas cláusulas
  - Periodicamente, divide todos los contadores por una cte

- Heuristica "quasi-estática"
  - Combina estado inicial con nuevas cláusulas
  - Usar un *heap* para encontrar la primera variable: una búsqueda lineal en cada decisión puede dominar run-time.



SAT: Evolución

Instance	Posit'94	Grasp'96	Chaff'03	Minisat'03	Picosat'08
ssa2670-136	13,57	0,22	0,02	0,00	0,01
Bf1355-638	310,93	0,02	0,02	0,00	0,03
design_3	>1800	3,93	0,18	0,17	0,93
design_1	>1800	34,55	0,35	0,11	0,68
4pipe_4_ooo	>1800	>1800	17,47	110,97	44,95
fifo8_300	>1800	>1800	348,50	53,66	39,31
w08_15	>1800	>1800	>1800	99,10	71,89
9pipe_9_ooo	>1800	>1800	>1800	>1800	>1800
c6288	>1800	>1800	>1800	>1800	>1800

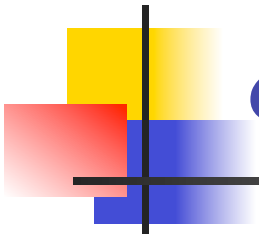
CSP

CSP *(X, D, C)*

- $X = \{x_1, x_2, \ldots, x_n\}$ variables
- $D = \{D_1, D_2, \ldots, D_n\}$ dominios (discretos y finitos)
- $C = \{c_1, c_2, \ldots, c_r\}$ restricciones

- Restricción *c*: *var*(*c*) : variables involucradas en *c*
  rel(*c*): tuplas de valores permitidas por *c*

- Solución: asignación de valores a variables que satisface todas las restricciones (NP-completo)



¿Por qué CSP?

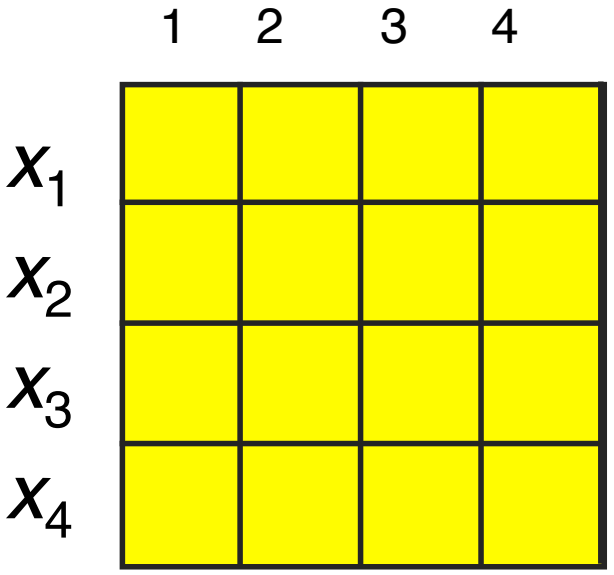
- Modelo muy usado para problemas industriales
  - Scheduling
  - Turnos de trabajo
  - Logística
- Éxito comercial (ILOG)
- Expresividad
- Programación declarativa
  - Variables
  - Dominios
  - Restricciones

CSP: 4-reinas

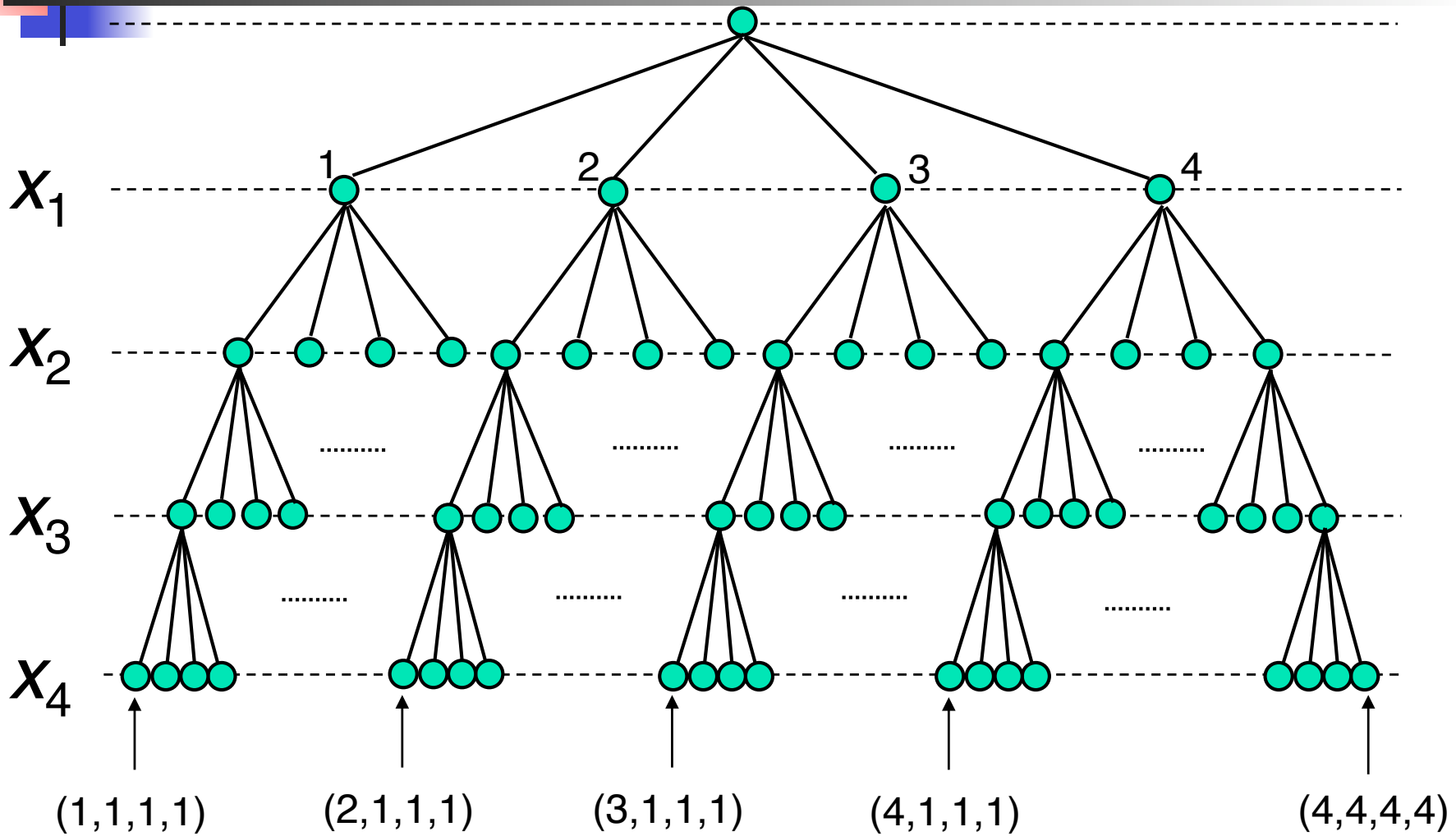
Objetivo: poner 4 reinas en un tablero de ajedrez 4 x 4 de forma que no se ataquen

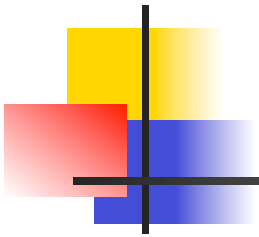
Formulación:

- Variables: una reina por fila
- Dominios: columnas
- Restricciones: dos reinas han de estar en
  - Diferentes columnas
  - Diferentes diagonales



CSP: Árbol de búsqueda





CSP: *Backtracking*

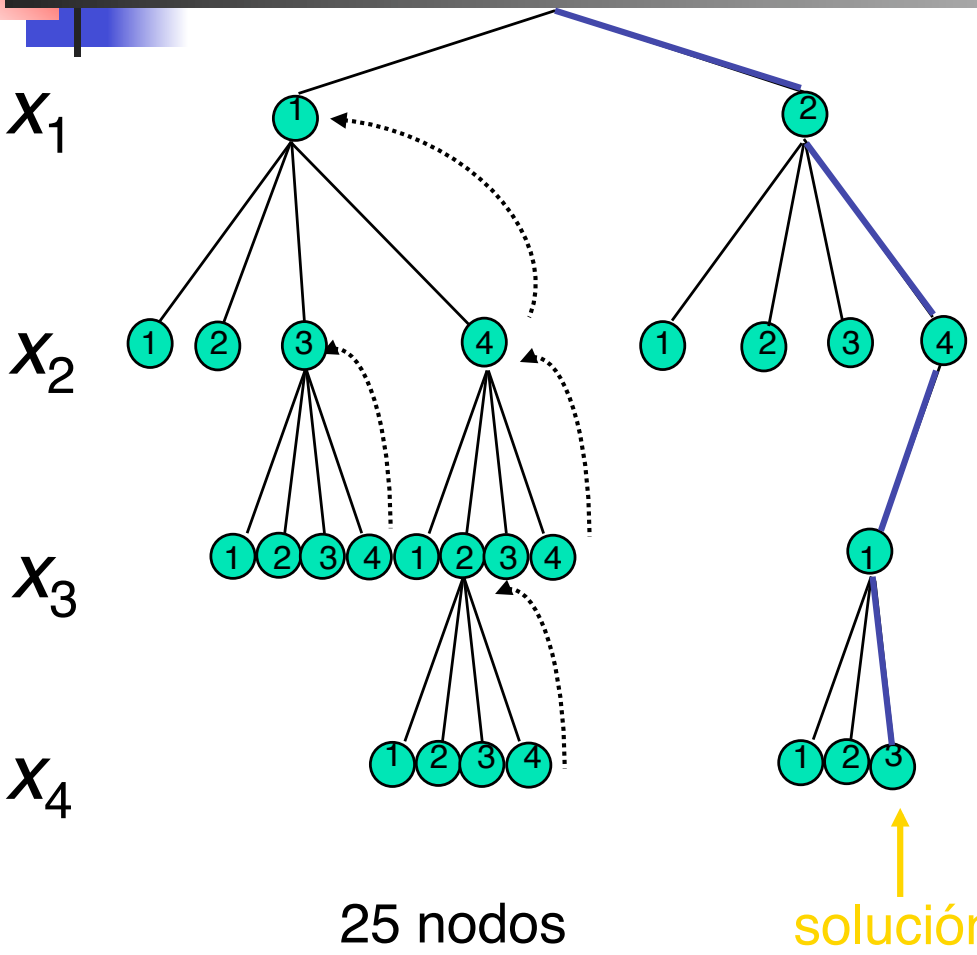
Visita nodos primero en profundidad (DFS)

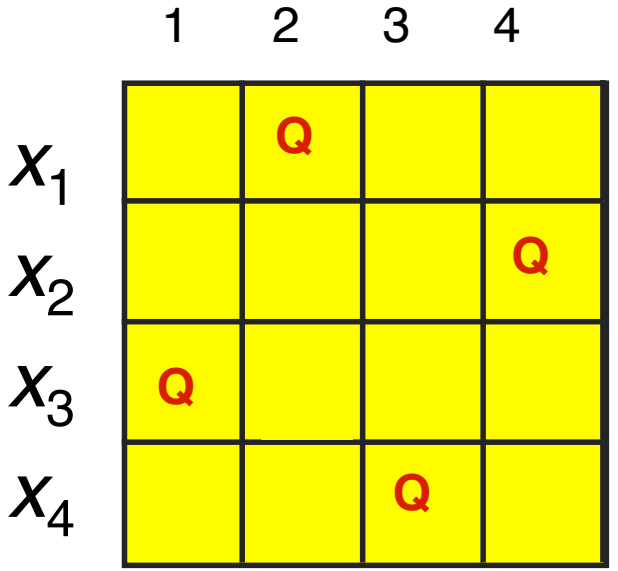
En cada nodo:

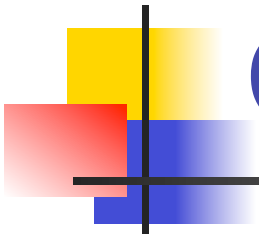
- Verifica cada restricción totalmente asignada
- Si es consistente, continua DFS
- Sino, poda la rama actual y continua DFS

Complejidad: $O(d^n)$

CSP: ejemplo de *backtracking*







CSP: Arco Consistencia

- Arco consistencia: inferencia dentro de cada restricción
- Cada valor del dominio de una variable que no aparezca en la solución de la restricción se elimina
- Efecto combinado de varias restricciones: poda significativa del dominio

Ejemplo: Ecuaciones

$x + y = 9$

$2x + 4y = 24$

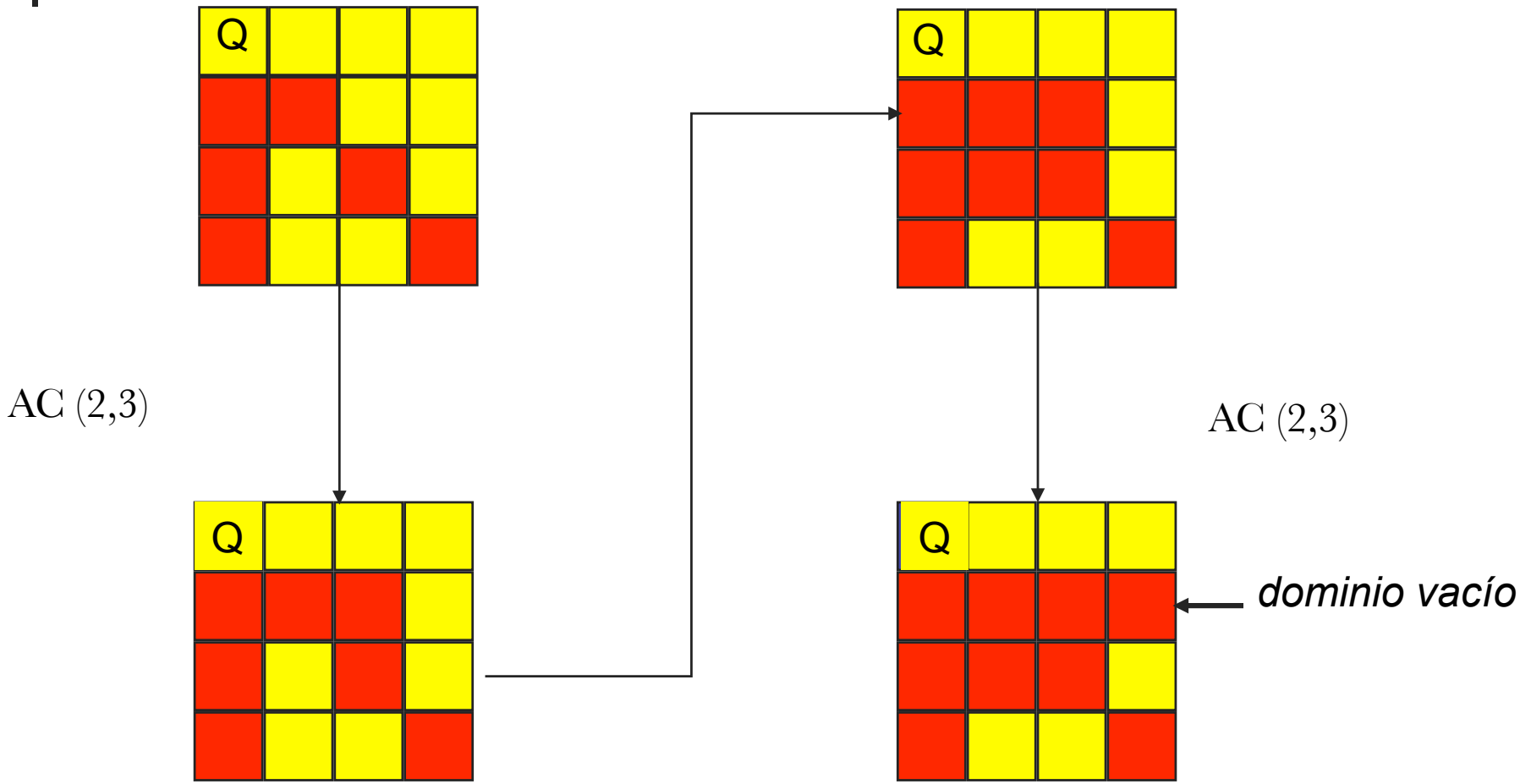
x	0	1	2	3	4	5	6	7	8	9
y	0	1	2	3	4	5	6	7	8	9

Punto fijo

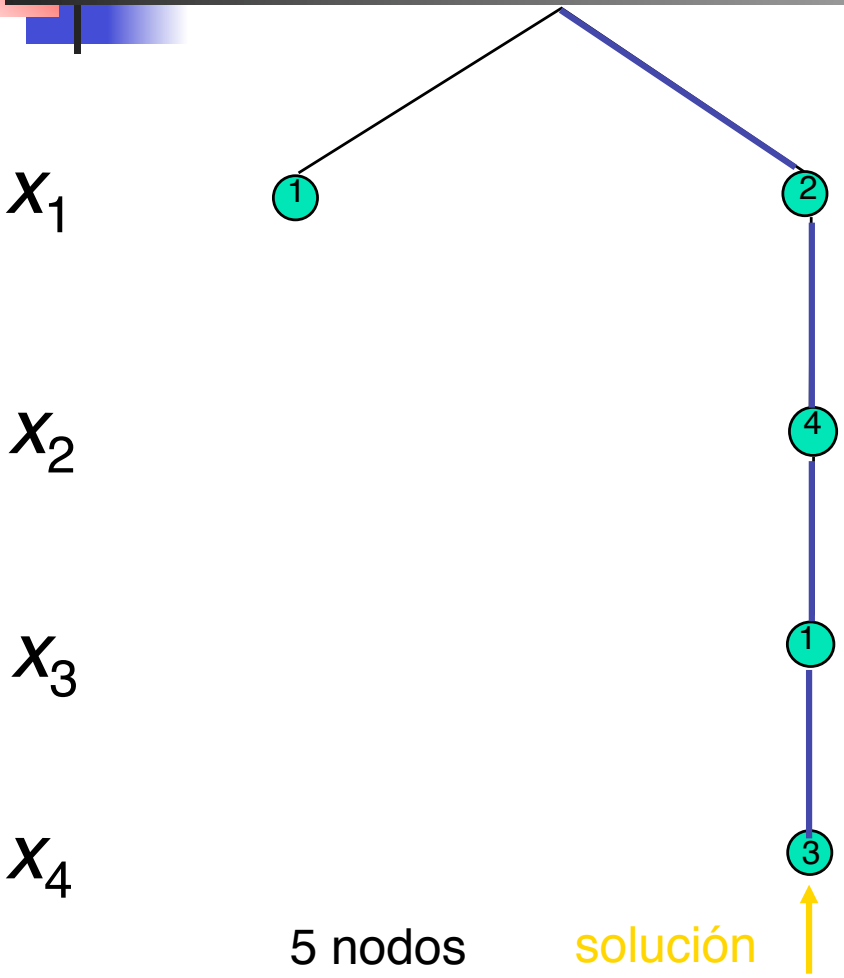
CSP: *Maintaining Arc Consistency*

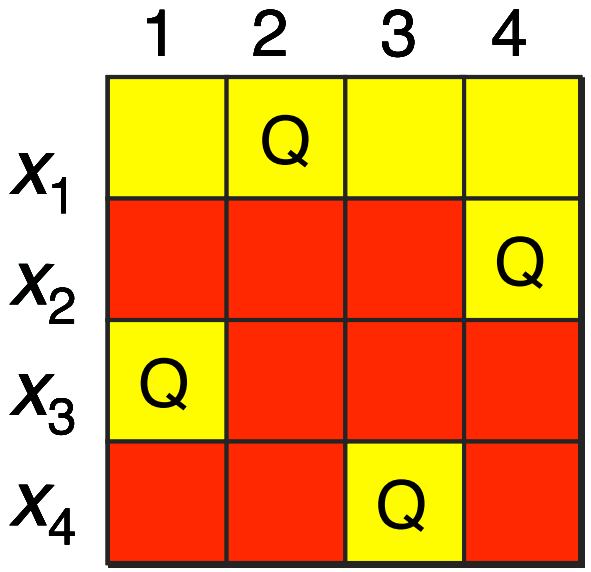
- MAC es combinación
  - Búsqueda: BT (backtracking)
  - Inferencia en cada nodo: AC (arco consistencia)
  - Preproceso: subproblemas se hacen AC
- Cuando un dominio se vuelve vacío
  - No hay soluciones en la rama actual
  - Backtrack
- Restaurar:
  - Valores eliminados en el nivel i se han de restaurar cuando se hace backtracking al nivel i o anterior

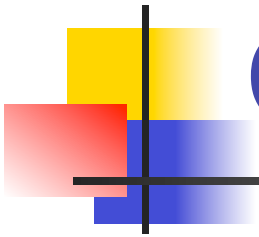
CSP: AC en 4 reinas



CSP: Ejemplo MAC en 4 reinas

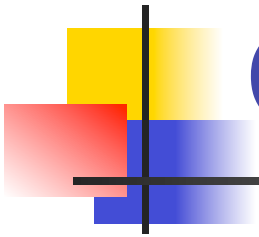






CSP: Extensiones

- Variables
  - Inicialmente, naturales (y reales)
  - Conjuntos
  - Multi-conjuntos
- Restricciones
  - Inicialmente, obligatorias (duras, *hard*, problema de satisfacción)
  - Ahora, no obligatorias (blandas, *soft*, problema de optimización)

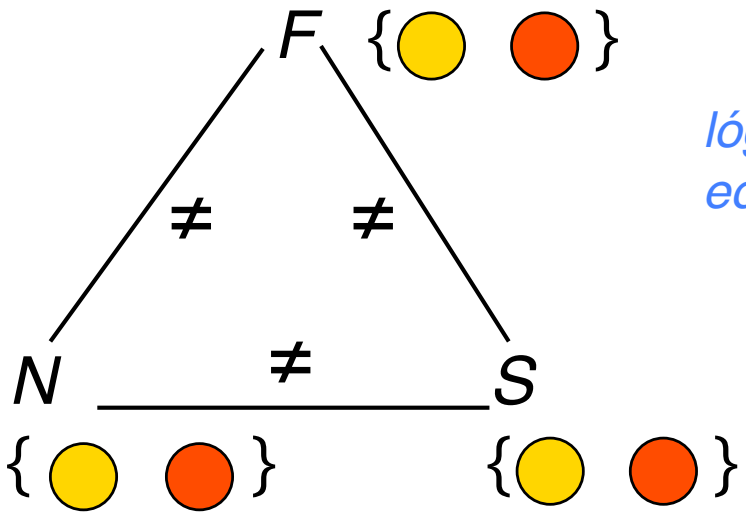


CSP: Restricciones Globales

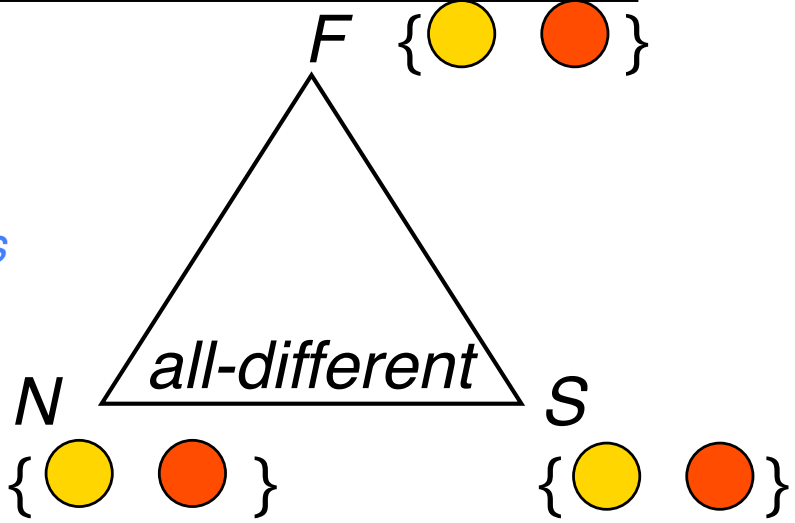
- Restricciones entre más de dos variables
- Pueden (o no) ser equivalentes semánticamente a un conjunto de restricciones binarias
- AC sobre una restricción global es más potente (poda más) que AC sobre las restricciones binarias equivalentes
- Implementadas con propagadores
- Semántica para disminuir la complejidad de AC
- Esenciales para modelizar y resolver problemas reales
- Actualmente unas 400 en http://

CSP: *all-different*

Var: *F, N, S*; Val: {  }; Ctrs: $N \neq S \neq F \neq N$



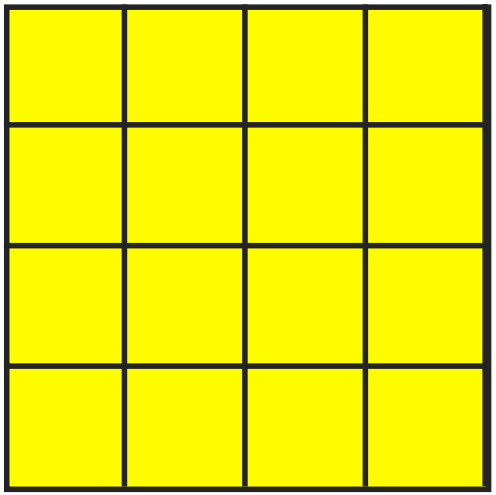
lógicamente equivalentes



3 restricciones binarias, son AC, no hay poda

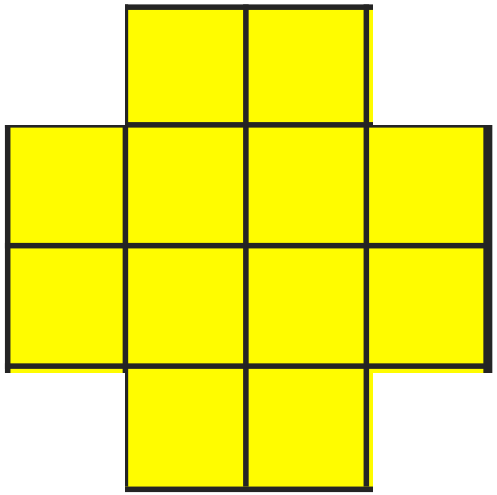
1 restricción ternaria, no es AC, poda → dominio vacío
no hay solución!!

CSP: Simetrías



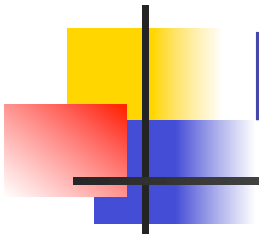
No hay solución con reina en esquina arriba izquierda

POR SIMETRÍA



- Explotación de simetrías: aumenta la eficiencia
- Simetrías: aparecen en muchos problemas reales

Planificación



Planificación

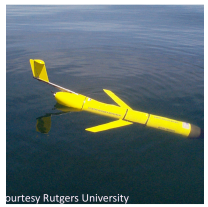
- Dado (*A, O, I, G*), (átomos, operadores, estados inicial y final) encontrar secuencia de operadores o_1, o_2,…,o_k que conectan estados *I* y *G*.

- Operador *o* (*Strips*): tres listas
  - Precondiciones: *Pre*(*o*)
  - A añadir: *Add*(*o*)
  - A borrar: *Del*(*o*)

- Complejidad: PSPACE-completo

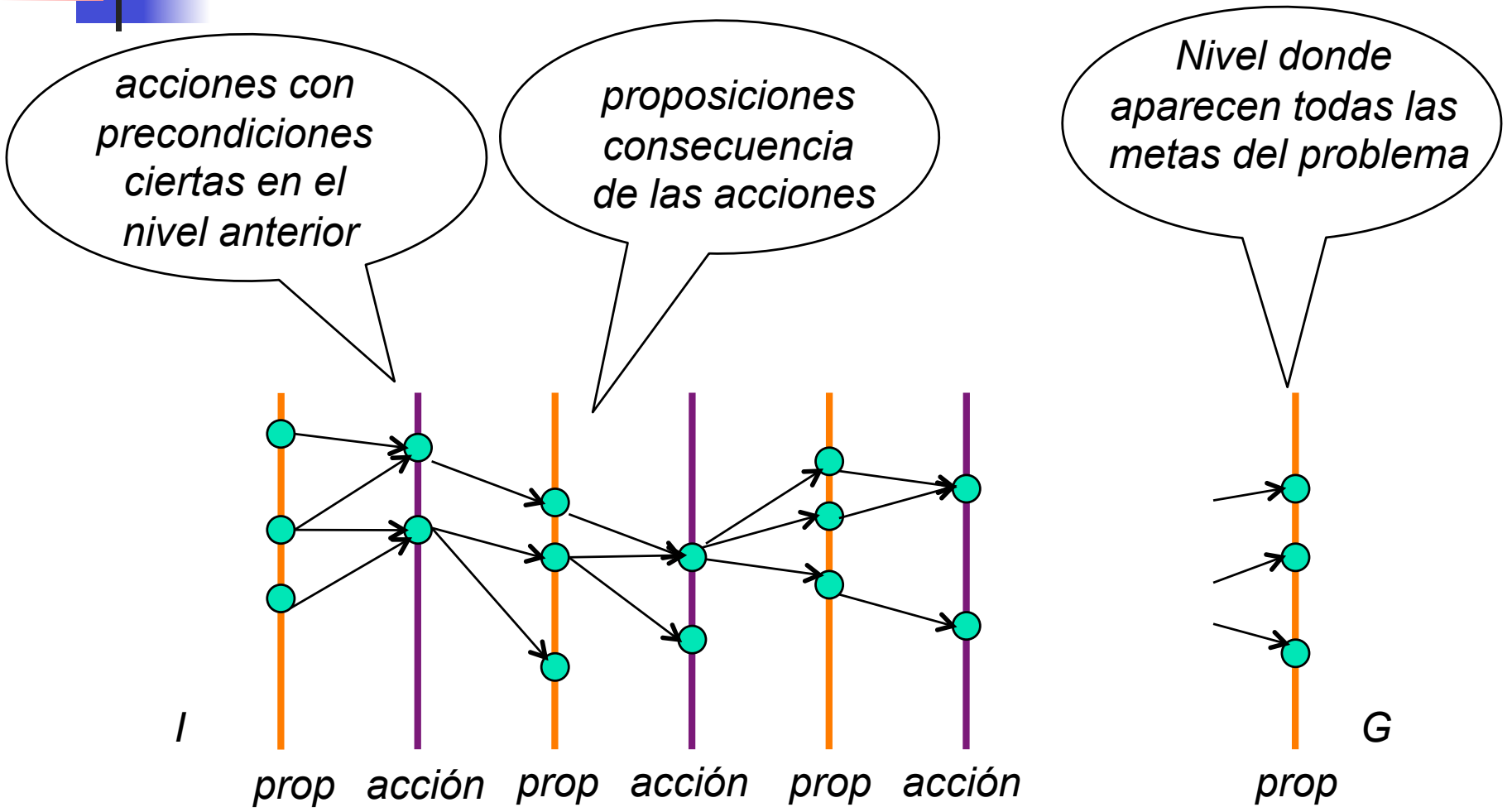
¿Por qué planificación?

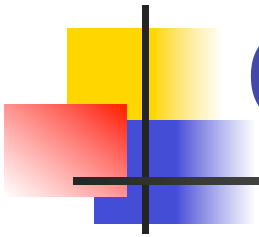
- Se han utilizado con éxito
  - Misiones espaciales
  - …
  - Tareas especiales



- Aumento de rendimiento
- Muchos modelos:
  - clásico, conformante, contingente, con costes,…
  - buena expresividad

Grafo de planificación

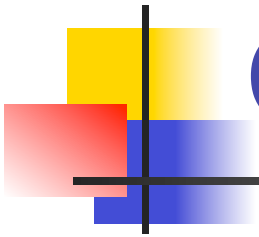




Construcción del plan

- Plan:
  - búsqueda hacia atrás en el grafo (IDA*)
  - iteración en la selección de acciones (no mutex)

- Acciones *mutex*:
  - precondiciones y efectos son inconsistentes, o
  - tienen precondiciones *mutex*
- Proposiciones mutex:
  - generadas por acciones que son *mutex*



Grafo relajado

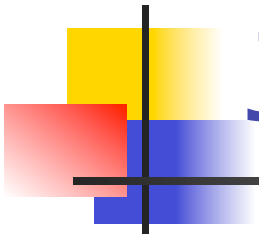
- Grafo de planificación SIN los borrados (*del*) de los operadores
- Sírve para generar heurísticas
- Las heurísticas se aplican en la búsqueda del plan
- Se han desarrollado gran cantidad de heurísticas

Otras aproximaciones

- Traducción
  - SAT
  - CSP
  - Model checking
- Planificación híbrida: combina técnicas
- Planificación jerárquica: descomposición de tareas
- Planificación basada en costes
- Planificación con recursos
- Planificación con incertidumbre

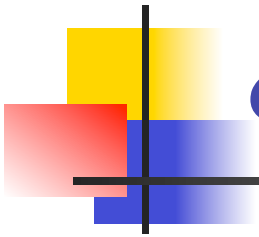
¡¡ENORME CANTIDAD DE TRABAJO DESARROLLADA POR ESTA COMUNIDAD !!

Juegos por ordenador



Juegos por Ordenador

- 2 jugadores
- Perfecta información:
  - cada jugador conoce *toda* la información del contrario
  - no hay elementos aleatorios
- Ejemplos: ajedrez, damas, othello
- Juegos con información incompleta: poker, bridge



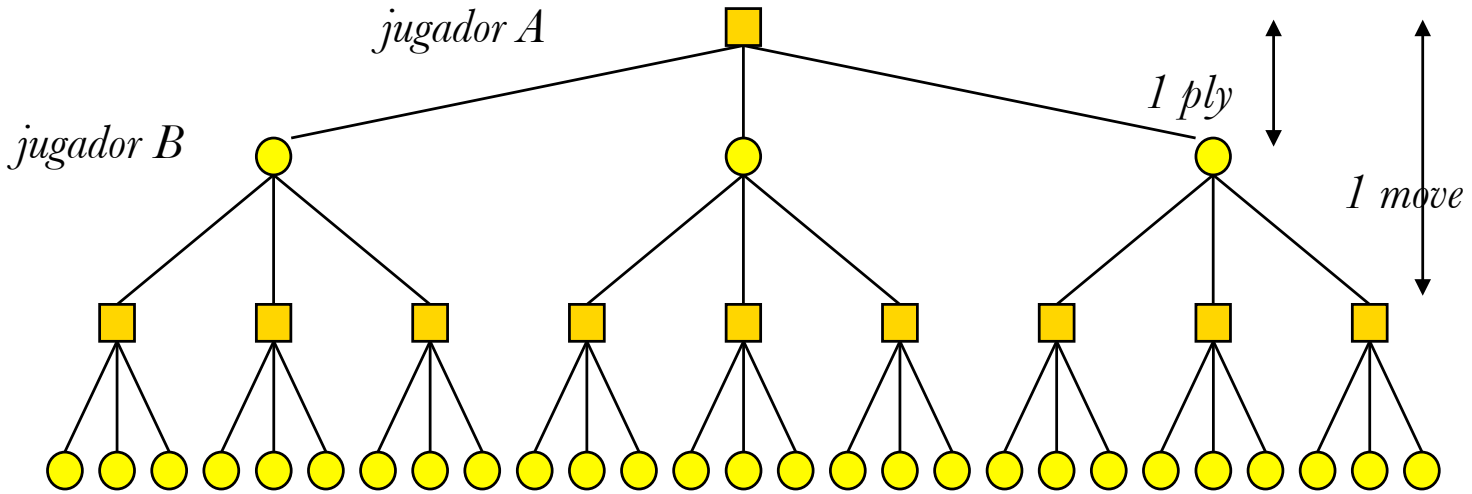
¿Por qué juegos?

- Programas campeones del mundo: damas, othello

- Programa con muy buen rendimiento: ajedrez
  - importante: desde 1956, ajedrez era un objetivo para IA
  - *Deep Blue* ganó a Kasparov en 1997
  - "*quantity had become quality*"

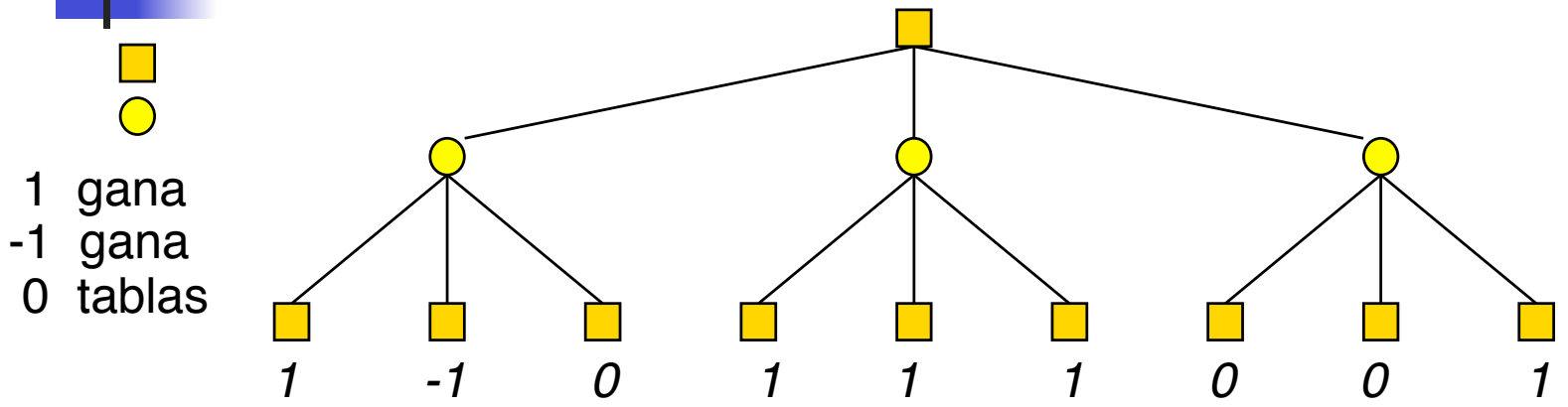
- Importancia económica directa

Árbol de juegos

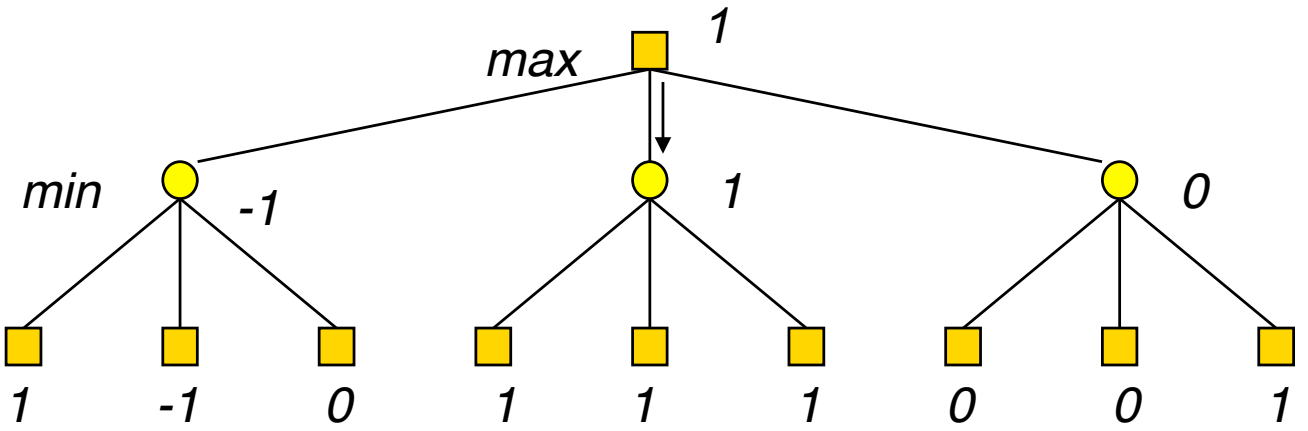


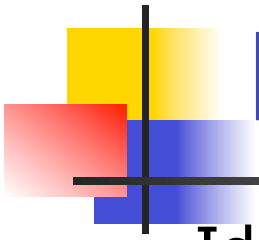
- Alternan jugadores por niveles
- Sucesores de un nodo: todas los movimientos legales que ese jugador puede hacer

MiniMax



Cuál es el mejor movimiento?
Supongo que el oponente elige el movimiento que más le conviene.





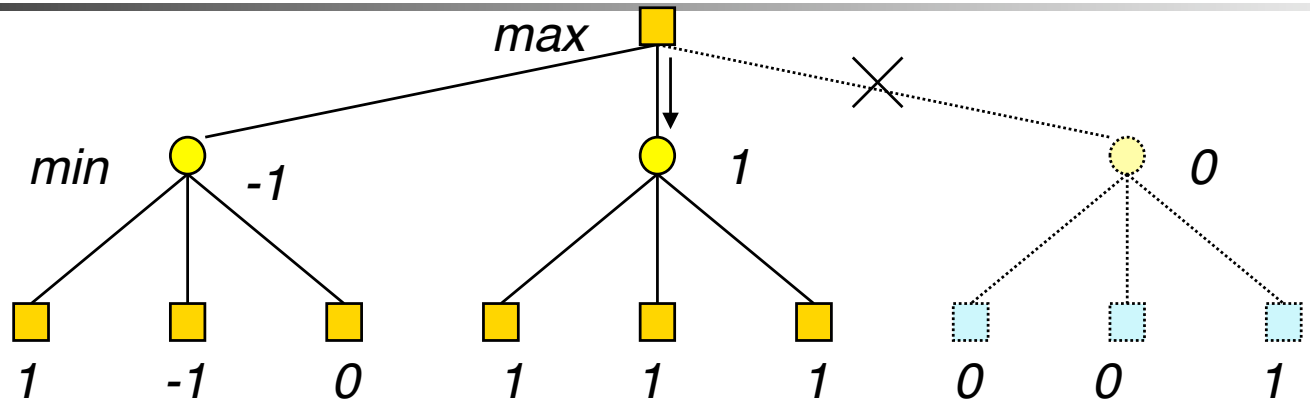
MiniMax

- Idea: propagar hacia atrás el valor de los hijos
- Nodos terminales**:** valor 1, -1 (gana A o gana B)
- Dos tipos de nodos:

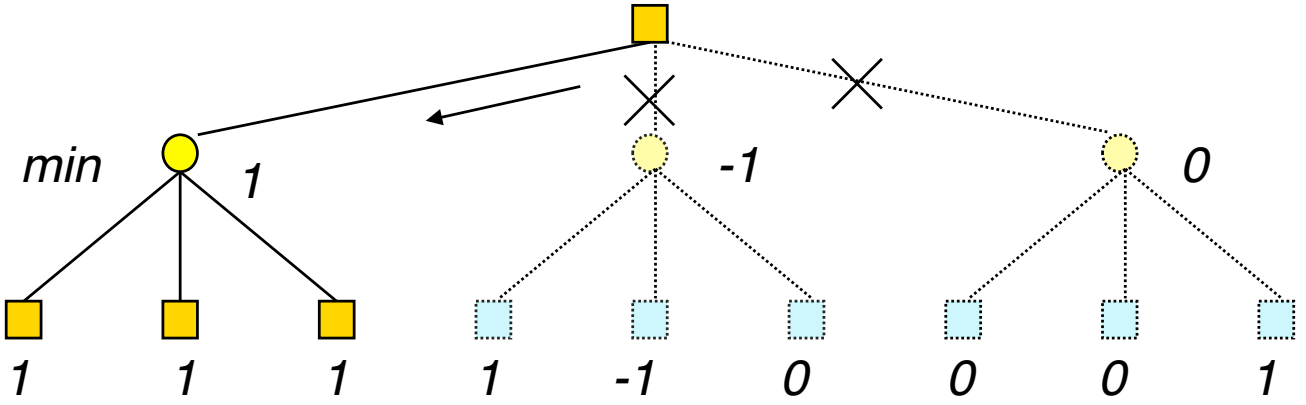
  max: gana con 1

  min: gana con -1
- Propagación hacia atrás:
  - **max**: el máximo de los valores de los hijos
  - **min**: el mínimo de los valores de los hijos

Alfa-Beta



Con una buena ordenación de sucesores,



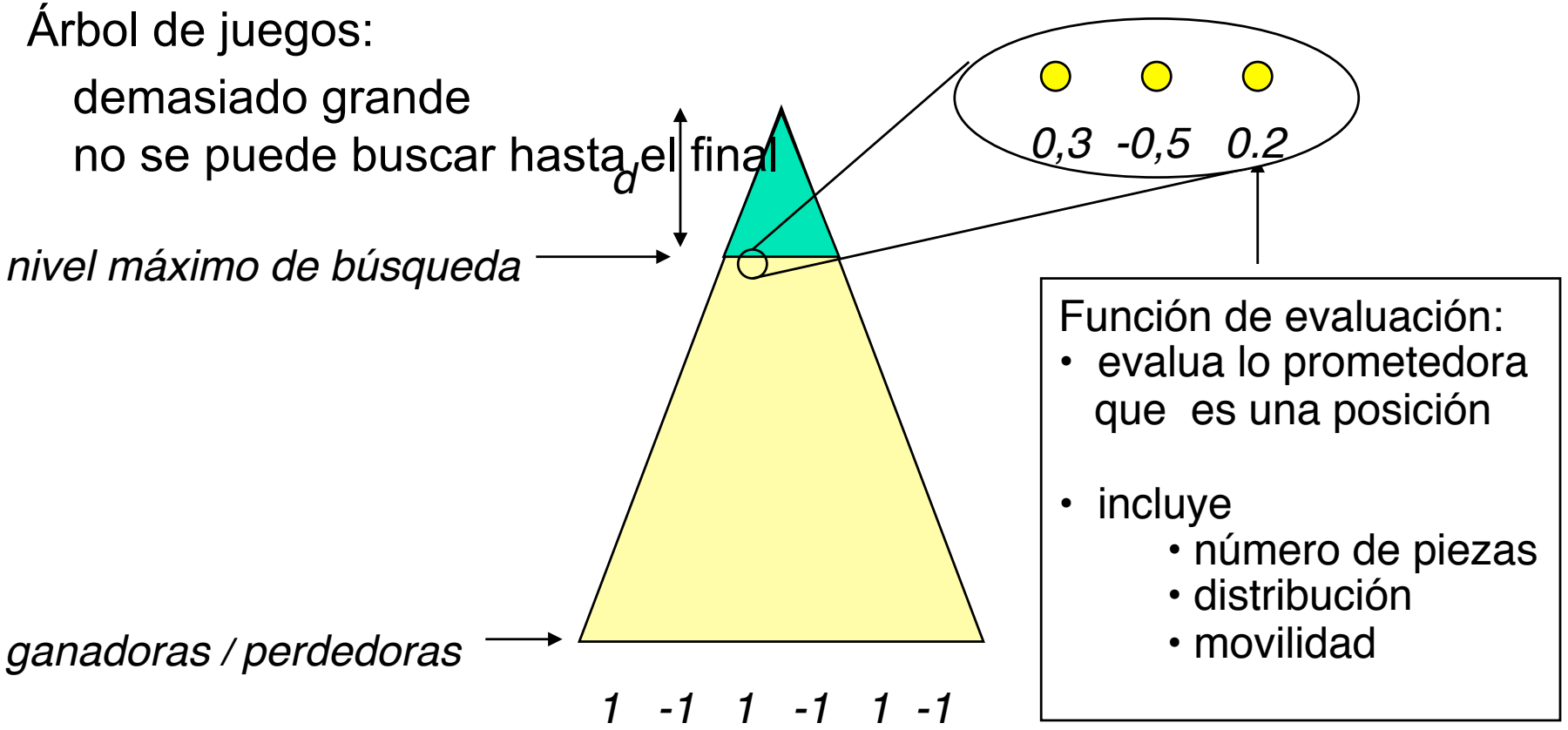
Poda muy importante: $b \approx 36 \quad \rightarrow \quad b \approx 6$

Función de evaluación

Árbol de juegos:

demasiado grande

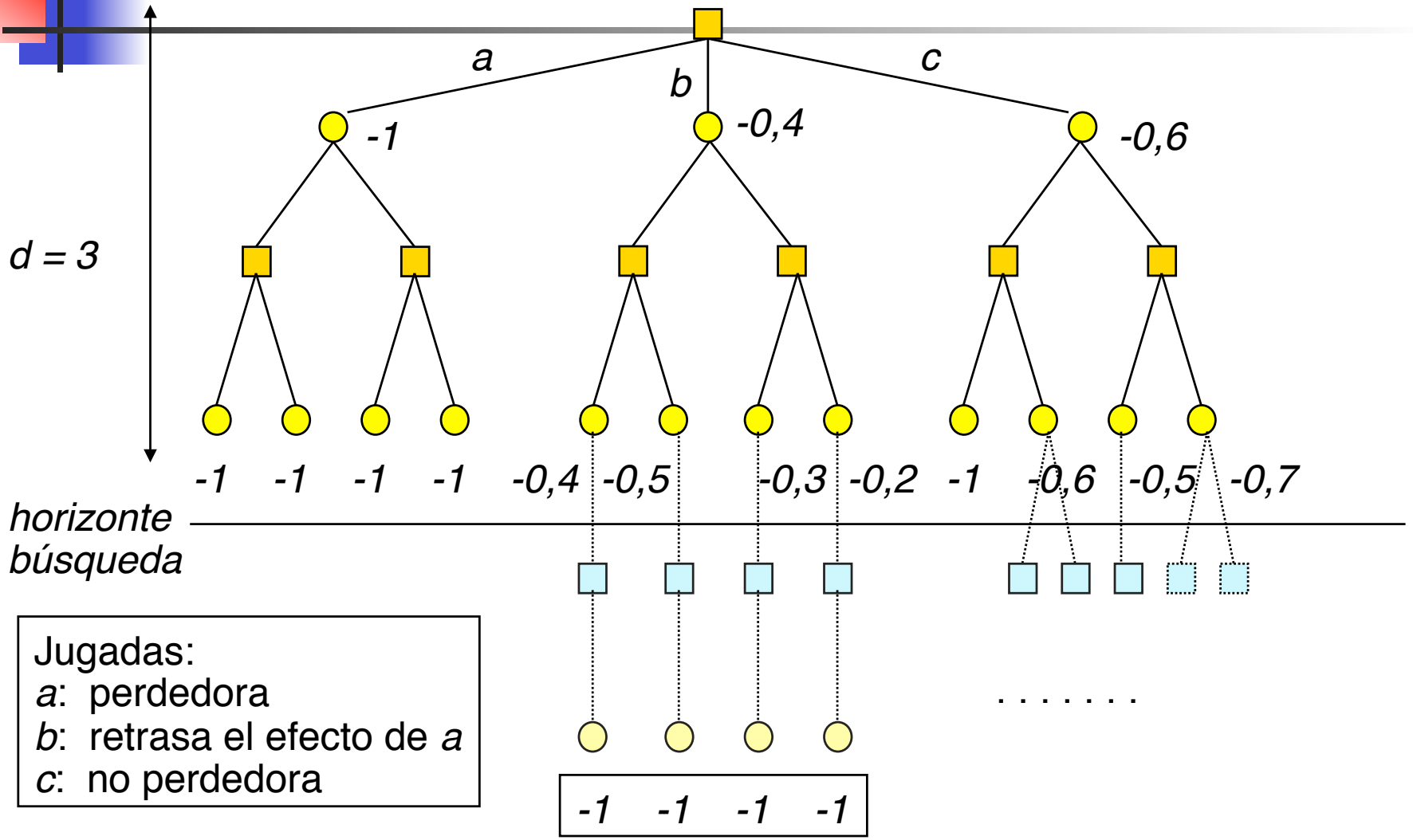
no se puede buscar hasta el final



Función de evaluación:

- evalua lo prometedora que es una posición
- incluye
  - número de piezas
  - distribución
  - movilidad

Efecto horizonte

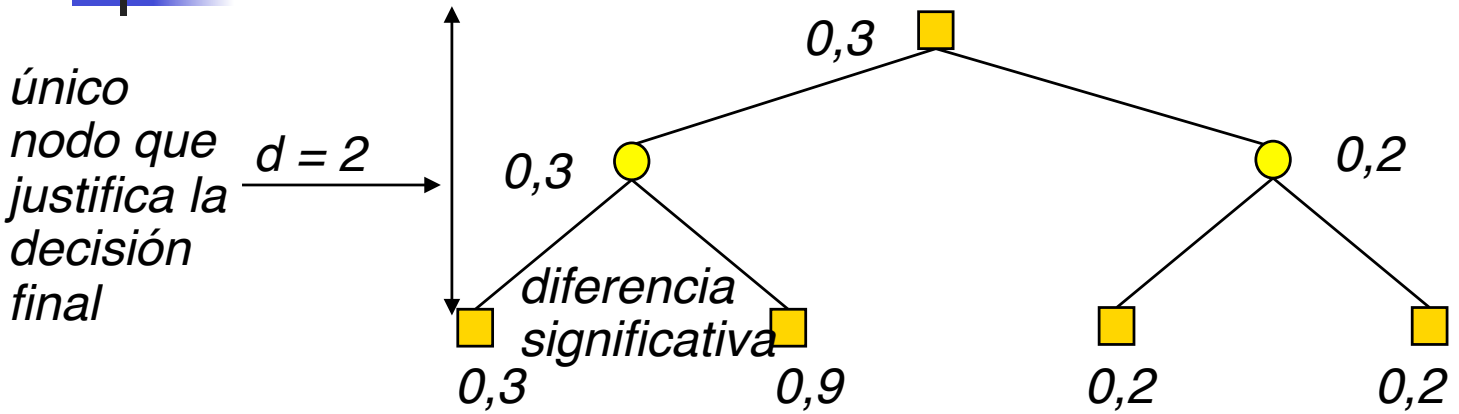


Jugadas:
a: perdedora
b: retrasa el efecto de *a*
c: no perdedora

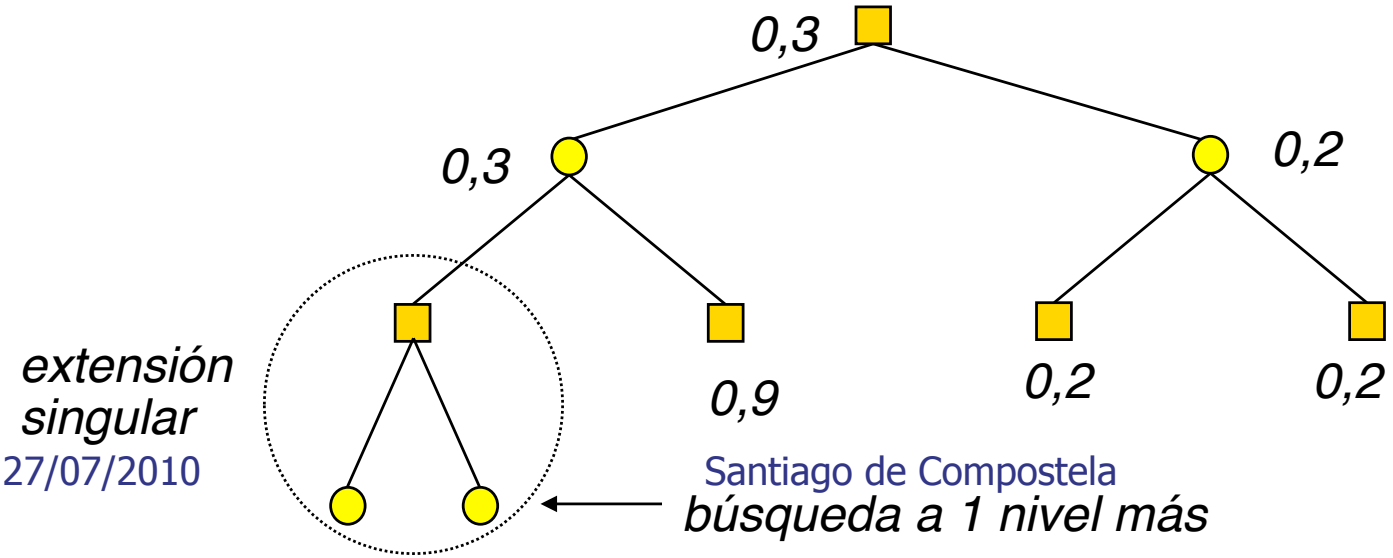
Efecto horizonte en ajedrez

- Efecto horizonte: una jugada que inicialmente parece buena puede resultar mala
  - el peligro no se ve, está tras el horizonte
  - la función de evaluación es incapaz de detectarlo
- Solución fuerza bruta: buscar más profundo
  - coste computacional
  - en ajedrez, hay muchos movimientos que pueden retrasar una jugada de peligro
- Aproximación selectiva: identificar los nodos en los que se debe profundizar más → extensiones singulares

Extensiones singulares



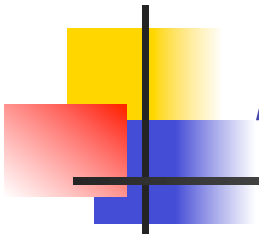
Buscar más debajo de ese nodo



Deep Blue

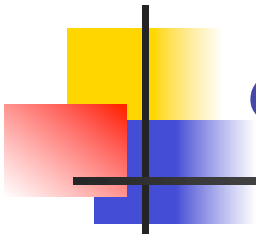
- Proyecto de IBM
- Alfa-beta paralelo:
  - *Hardware* especializado (2 millones posiciones / sg)
  - Extensiones singulares, quiescencia
  - Función de evaluación sofisticada
  - Biblioteca: aperturas y finales
- Resultados:
  - Venció a Kasparov en 1997
  - Ratio 2700, Kasparov 2800
  - *quantity had become quality*

Pattern Databases



Pattern Databases

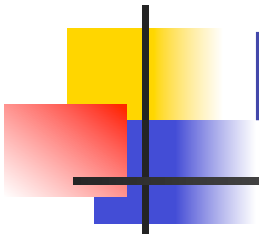
- Almacena en un fichero la heurística
  - Es muy grande pero factible dada la tecnología actual
- Indexa el fichero por unas pocas piezas (*pattern*)
  - Para que la explosión combinatoria no sea inmanejable
- Generación fichero: búsqueda en anchura (desde la solución hacia atrás)
  - Cuando aparece configuración (*pattern*) nueva, se guarda su coste
  - Es una cota inferior de ese mismo *pattern* en otro contexto
- El fichero se genera *off-line*
- El coste se amortiza en muchas resoluciones



¿Por qué *Pattern Databases*?

- Porque han permitido resolver en tiempos inimaginables problemas de búsqueda enormes

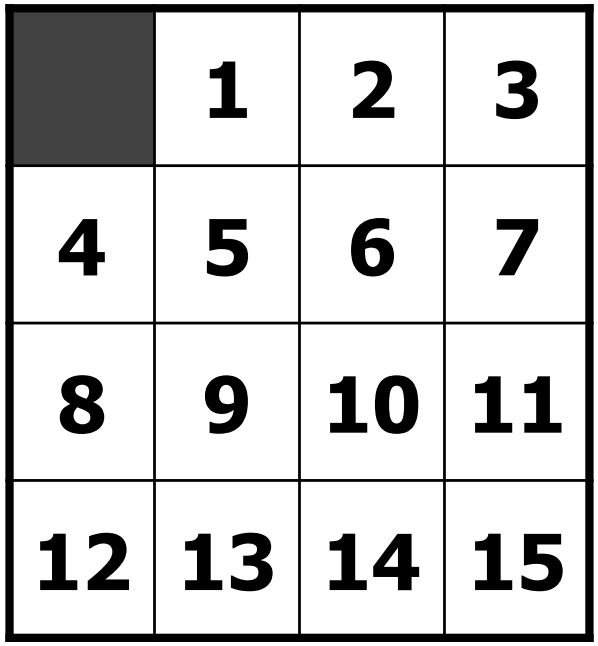
- Ejemplos:
  - Cubo de Rubik (43×10^{18} estados): camino más largo 26 (63 horas, 128 procesadores, 7 terabits de disco)
  - Puzle de 15 (10^{13} estados): se resuelve en milisegundos
  - Puzle de 35 (10^{41} estados): cotas coste solución óptima (1 mes, 16 GB ram, 4 terabytes disco)
  - Torres de Hanoi (4 postes, 30 discos, 4^{30} estados): sol óptima (17 días, 400 GB disco)



Problemas de búsqueda en IA

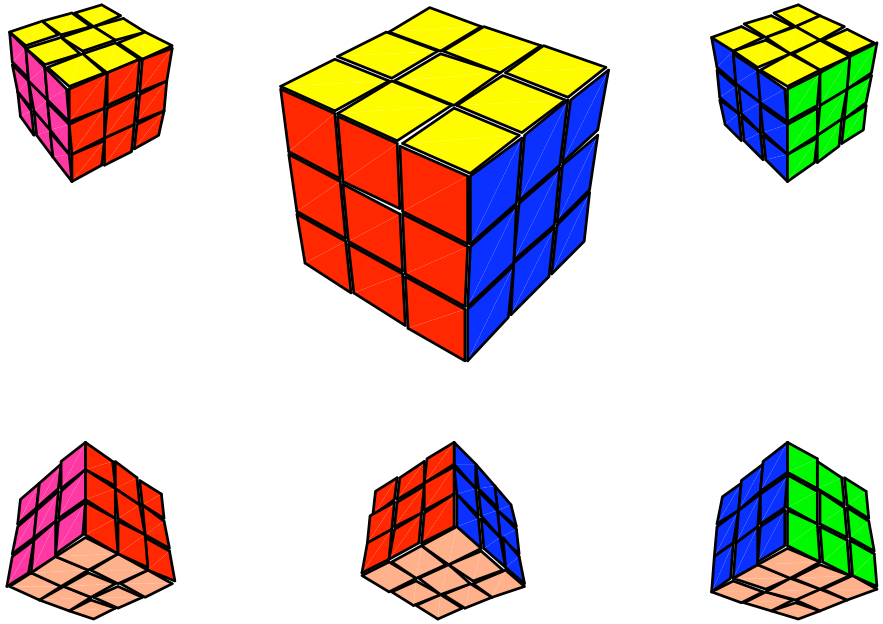
- Puzles de fichas deslizantes
  - 8 puzle (3x3)
    - Pequeño: 8! / 2 ≈ 1.8 x10^5 estados
  - 15 puzle (4x4)
    - Grande: 16! / 2 ≈ 10^{13} estados
  - 24 puzle (5x5)
    - Enorme: 25! / 2 ≈ 7.8 x 10^{25} estados
- Cubo de Rubik
  - 2x2x2
  - 3x3x3

15 Puzle



- Inventado por Sam Loyd en 1878
- $16! / 2 \approx 10^{13}$ estados
- Media de 53 movimientos para resolverlo
- Diámetro conocido (longitud máxima de camino óptimo): 87
- Factor de ramificación: 2.13

3x3x3 Cubo de Rubik

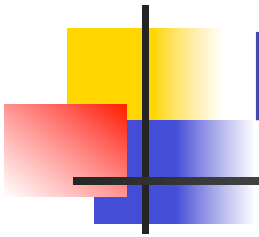


- Inventado por Rubik en 1974
- 4.3 x 10^{19} estados
- *Half Turn Metric*
- Media de 18 movimientos para resolverlo
- Diámetro: 26
- Factor de ramificación: 13.35

Heurística Manhattan

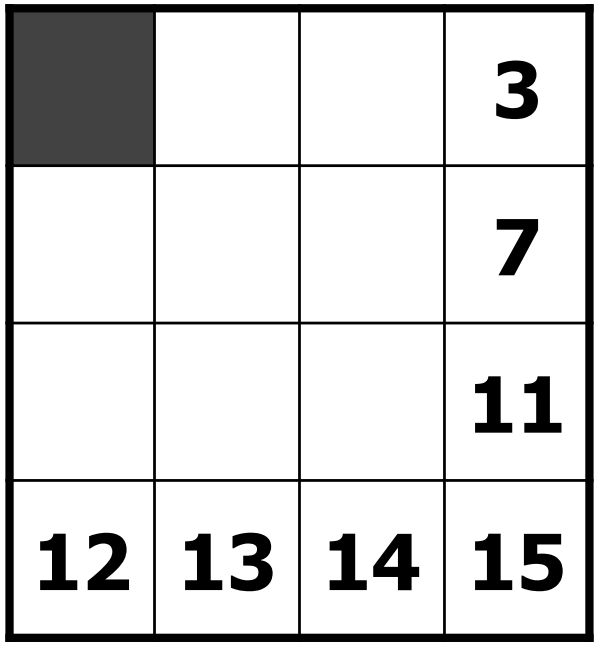
- Heurística admisible
  - Suma del número óptimo de movimientos de cada pieza para alcanzar su posición en el estado objetivo, permitiendo que se mueva libremente
- Razonablemente buena para puzles
  - Usada por A* pasra resolver el 8 puzle
  - Usada por Richard Korf con IDA* para resolver por primera vez el 15 puzle.
- No es muy buena para el cubo de Rubik
  - Se debe dividir el resultado final por 8 (cada movimiento involucra 8 piezas) resultando en un valor más bajo que el esperado (≈5,5)

Mejores heurísticas



- *Pattern Databases*
  - Almacena el número de movimientos de TODAS las piezas para resolver subpartes del puzle
  - Este coste es una heurística admisible
  - Produce heurísticas mucho mejores que Manhattan
- Permite combinar
  - Máximo
  - Suma

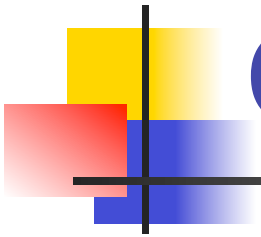
Ejemplo



- *Fringe pattern*: nos fijamos solo en esas 7 piezas y el hueco
- Cada configuración de esas 8 piezas en un *pattern* ($COM_{16,8}$)
- Almacenamos el coste mínimo (= movimientos de todas las piezas del puzle) para ir del *pattern* al objetivo (495MB)
- Resultados originales 15 puzle:
  - Nodos generados divididos por 346
  - Tiempo dividido por 6

Combinar con máximo

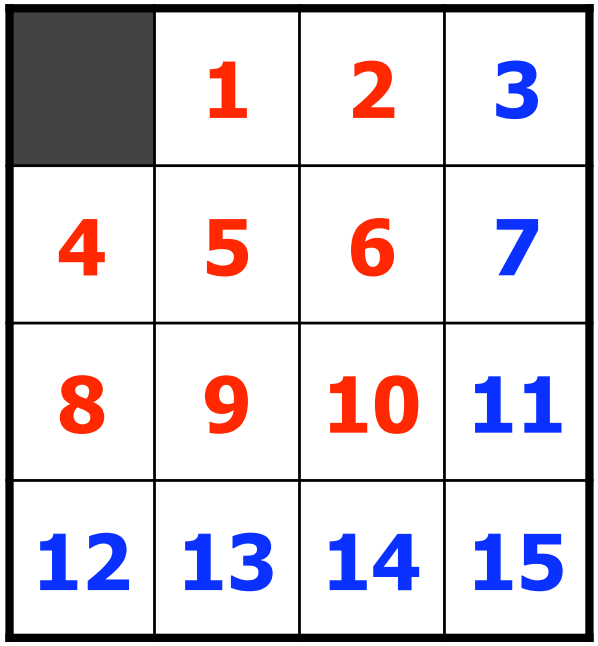
- Tenemos dos pattern databases PDB_1 y PDB_2
- Sea c el estado actual, queremos $h(c)$
  - Por PDB_1 obtenemos $h_1(c)$
  - Por PDB_2 obtenemos $h_2(c)$
  - $h(c) = \max \{h_1(c), h_2(c)\}$
- 15 Puzle:
  - Nodos generados: divididos por 1.000
  - Tiempo: dividido por 12



Combinar con suma

- Problema con escalado de PDBs
- Combinar con suma: no se pueden sumar heurísticas de dos *pattern databases* porque puede haber costes repetidos
- Salvo si …. se guardan costes SOLO del *pattern,* entonces es suficiente con que los patterns sean disjuntos
- Para sumar, se guarda el número de movimientos de las piezas del *pattern* (no de todas las piezas)

Ejemplo



- Dos *pattern*: azul y rojo
- Generaremos 2 *pattern databases* (azul y rojo)
- En cada una, almacenamos el coste mínimo (= movimientos de todas las piezas del *pattern*) para ir del *pattern* al objetivo
- Resultados originales 15 puzle:
  - Nodos generados divididos por 11.000
  - Tiempo dividido por 2.000

Algunas Conclusiones

Conclusiones

- Razonamiento automatizado:
  - Rendimiento
    - Buenas implementaciones
    - Competiciones
  - Expresividad
    - Librerías de problemas
  - Inferencia sofisticada
  - Tecnología madura

Conclusiones

- Desafíos:
  - Simplicidad de uso
  - Encapsular: *Black box*
  - Modelización
  - Diseminación