Inteligencia Artificial

Inteligencia artificial (Universidad de Alcalá)

Inteligencia artificial
Grado en Ingeniería Informática

Carlos Múgica Gómez

2013

Índice

Presentación

ASIGNATURA: INTELIGENCIA ARTIFICIAL

- Tema 1º Introducción, panorama histórico y conceptual
- Tema 2º Búsqueda desinformada
- Tema 3º Búsqueda heurística
- Tema 4º Problemas de satisfacción de restricciones
- Tema 5º Juegos
- Tema 6º Conocimiento, razonamiento e incertidumbre

BIBLIOGRAFÍA GENERAL

Nilsson, Inteligencia Artificial, McGraw-Hill ; http://aima.cs.berkeley.edu/index.html
Russell-Norvic, Inteligencia Artificial, Pearson
Poole-Mackworth, Artificial Intelligence, Cambridge Univ. Press
Fdez.Galán-Glez.Boticario,Mira, Problemas resueltos de Inteligencia artificial, Addison- Wesley

CALIFICACIÓN

Por evaluación continua: tres pruebas teórico- prácticas (teoría, cuestiones y problemas) a realizar los miércoles 27 de febrero,3 de abril y 8 de mayo en el aula NA3, en horario de teoría, que puntuarán el 75 % de la nota, más la entrega y defensa de dos proyectos de laboratorio, a presentar a partir de abril en fechas a precisar, que completarán el 25 % restante de la nota. Es necesario obtener en cada una de las partes un mínimo de 3 puntos sobre 10 para poder superar el conjunto en media.
Por examen final: examen teórico -práctico (75 % de la nota, en las fechas establecidas por la Escuela para los exámenes de mayo y extraordinario de junio) más entrega y defensa de proyectos, antes del correspondiente examen (25 % de la nota, con el mismo mínimo de 3 para poder promediar.

1. Introducción, panorama histórico y conceptual

Se intentan desarrollar sistemas informáticos que tengan caracteristicas que se supone que manifiestan los seres humanos al enfrentarse a problemas.
La I.A. arranca en 1956 en Darmouth College, la idea, sin embargo, empezó en la antiguedad (Golem de Praga, leyenda judía). En Egipto se hicieron mecanismos que permitian abrir puertas automáticamente.
Posteriormente se quisieron desarrollar máquinas artificiales con las que se pudiera jugar al ajedrez, pero los juegos son problemas que requieren de I.A.

1.1. Génesis de las ideas sobre la computación e inteligencia artificial

1.1.1. En el ámbito formal y abstracto:

Proceso de formalización de las Matemáticas:
A partir del S. III a.C. las matemáticas se dividen en:

- **Aritmética:** Ciencia de los números naturales y sus operaciones.
- **Geometría:** Espacio y sus conceptos.

Cuando el sedentarismo y la civilización aparecen (Mesopotamia, Egipto, China...), surgen problemas aritméticos y geométricos, hay que planificar tareas agrícolas, planificar construcciones...

Fue en la antigua Grecia donde se asentó finalmente la matemática, crearon un sistema mercantil, y un sistema legal en el cual había juicios y había que defenderse. En dichos juicios surgió la *lógica*, el arte de la argumentación.
El principal desarrollo de la lógica lo hizo Aristóteles en su libro *"Oganon"* (385 a.C), en el cual explica cómo organizar el conocimiento y cómo razonar correctamente mediante *silogismos.*
La combinación entre aritmética y geometría empíricas, junto con la lógica hicieron surgir un nuevo concepto de las matemáticas. Euclides escribió *"Elementos"* (325 a.C.) en el cual organizó los conocimientos empíricos anteriores por medio de definiciones, enunciando postulados o axiomas, dicho libro tuvo tanto éxito que hasta aproximadamente 1900 se usaba su libro para dar matemáticas, la cual era considerada una ciencia perfecta e indiscutible que permitía descubrir cosas sobre el mundo que nos rodea y que sirve como modelo para otras ciencias.
Para garantizar que las matemáticas seguian siendo una ciencia rigurosa se formalizaron. Este proceso de formalización comenzó con Boole, que algebrizó la lógica, enunciando y analizando una estructura *booleana*, que describe el razonamiento con proposiciones. De ahí se pasó a Frege, que amplió al cálculo de predicados. Los predicados son relaciones entre objetos de un dominio. Dispone de expresiones como relaciones unarias ($p(x)$) o binarias ($p(x, y)$) que se usan junto a los conectores lógicos ($\vee$, $\wedge$, $\neg$, $\rightarrow$) y los cuantificadores $\exists$ y $\forall$.

Intento formalista Hilbert 1900: Hay que formalizar por completo todas las estructuras matemáticas y sus reglas para comprobar que nos da una estructura:

- **Consistente:** El sistema resultante no ha de ser contradictorio, asi no se corre el riesgo de hacer paradojas.
- **Completa:** Cualquier enunciado formal de ese sistema que sea correcto sintácticamente ha de ser demostrable o refutable.
- **Decidible:** Si hay algún procedimiento finitista para comprobar que un enunciado es consecuencia de otro.

Para precisar el intento formalista, hubo que precisar qué entendemos por procedimiento (algoritmo) y comprobarlo finitísticamente. La hipótesis era que si existía dicho procedimiento, la respuesta la dio Gödel (1930) en el teorema de la incompletitud (ningun sistema formal que incluya lo necesario para expresar las matemáticas será consistente y completo).
En 1936 Alan Turing aclaró decidiendo lo que era computable o no con la máquina de Turing y comprobó que no existía algoritmo que pueda decidir sobre todo.
También hacia 1936, Church desarrollo el λ-cálculo, que intenta precisar un concepto de algoritmo y de lo que este puede hacer.
Toda función computable, puede realizarse mediante una máquina de Turing o mediante el λ-cálculo o funciones recursivas.

1.1.2. En el ámbito de la ingeniería y de la Física

- Autómatas mecánicos e hidráulicos (época helenística, Alejandría)
- 1642 y 1671: Calculadoras mecánicas de Pascal y de Leibniz
- Máquinas diferencial(1822) y analítica (1833-42) de Babage
- Calculadoras electromecánicas (Zuse y otros)
- Ordenador de Von Neumann (1945)

1.1.3. Neurociencia, Psicología y Economía

- Gómez Pereira (1554), Descartes (1596), problema cuerpo-mente.
- Ramón y Cajal, McCullogs, Pitts, redes neuronales.
- von Neumann y Morgenstern, teoría de juegos.

1.1.4. Características importantes de la I.A.

- Se usa más información simbólica que numérica
- Se usan métodos heurísticos más que deterministas
- Se usa conocimiento específico-declarativo e información imperfecta (incompleta o incierta)

1.2. Conceptos de inteligencia.

Por un lado, lo computable es lo que se puede "algoritmizar". Por otro lado *la inteligencia es la habilidad de llevar a cabo pensamiento abstracto cuyas funciones implican:*

- Razonamiento.
- Percepción y entendimiento de la realidad.
- Toma de decisiones.
- Resolución de problemas mediante razonamiento, usando conocimientos previos.
- Aprendizaje.
- Adaptación al medio o las circunstancias.

La inteligencia es lo que se mide en un agente cuando percibe su entorno y actua sobre él adecuándose a unos fines.

1.3. Inteligencia artificial, fines y medios. Relación con otras disciplinas.

Algunas definiciones de I.A.:

- **Sistemas que piensan como humanos:**
  Ciencia Cognitiva → Se estudia cómo funciona el pensamiento humano para poder emularlo en las máquinas.
- **Sistemas que actuan como humanos:**
  Comportamiento humano → El test de Turing consiste en que un humano no sepa diferenciar si su interlocutor es una máquina, en base a las respuestas recibidas a una serie de preguntas.
- **Sistemas que piensan racionalmente:**
  Pensamiento racional → Los silogismos de Aristóteles codifican la manera de pensar. Si las premisas son correctas, se llega a soluciones correctas. Su estudio creó la lógica, esta construye sistemas inteligentes a partir de programas codificados en notación lógica, que en teoría resuelven todo, pero no en la práctica (falta de recursos, información...).
- **Sistemas que actuan racionalmente:**
  Agente racional → Es aquel que actua intentando encontrar el mejor resultado, y en caso de incertidumbre, el mejor resultado esperado. Diseñar un agente racional implica:
  - Efectuar inferencias correctas.
  - Conocer el pensamiento humano y su conducta en la evolución.

De este modo se puede encontrar una racionalidad perfecta que siempre haga lo correcto, pero como el dominio de posibilidades es demasiado grande se estudia una racionalidad limitada.

La creación de IA trata de construir agentes artificiales (hoy por hoy, sistemas informáticos) que emulen el comportamiento de agentes naturales (personas, animales o grupos o sociedades) cuando resuelven problemas siguiendo fines, percibiendo su entorno y actuando sobre él para adaptarse con éxito al mismo. Esta descripción, genérica, es lo bastante flexible como para poder abarcar desde el funcionamiento de un simple climatizador hasta el juego de un gran maestro de ajedrez.
También es susceptible de cuantificación: se pueden determinar criterios numéricos de comparación de su grado.

Esto relaciona el campo de la I.A., hoy entroncado en la Computación y encarnado en la Informática, con los de las ciencias formales (Matemática, Lógica, Epistemología, Ontología), la Neurociencia, la Ciencia Cognitiva, la Teoría de la Decisión, la Teoría de Juegos y la Economía.

1.3.1. Paradigma simbólico

Hipótesis del símbolo físico (Newell y Simon 1976): la condición necesaria y suficiente para que un sistema físico muestre comportamiento inteligente es que sea un *Sistema de Símbolos Físico.*
Por S.S.F se entiende algo capaz de representación de signos. La hipótesis postula que el uso de símbolos basta para representar el mundo, y que el diseño de mecanismos de búsqueda simbólica, especialmente heurística, para explorar el espacio de potenciales inferencias que estos sistemas simbólicos puedan presentar basta para emular la inteligencia, con independencia del medio de implementación física.
Esto equivale a decir que:

1. Dando por **anticipado** una lista de instrucciones (lo bastante detalladas y complejas), escritas en un **reglamento escrito permanente y definitivo** a un empleado que se lo tome siempre al pie de la letra actuando a ciegas (sin entender nada de lo que haga)

2. Que opere en una oficina con ficheros muy grandes (de fichas de cartulina como las de las bibliotecas antiguas), fichas que **contengan sólo cadenas de cifras** (que pueden representar codificaciones numéricas preestablecidas de símbolos) y un escritorio sobre el que puede ejecutar las instrucciones del reglamento, consistentes sólo en **órdenes para extraer** fichas, colocarlas sobre la mesa, hacer cambios estipulados **borrando parte de su contenido** o **añadiendo contenido** de otras fichas y **devolverlas a posiciones del fichero**

3. Dándole tiempo suficiente, es posible emular el comportamiento inteligente de personas o animales que resuelven **problemas nuevos**, no planteados previamente en un ámbito dado.

1.3.2. Paradigma conexionista (redes neuronales y algoritmos genéticos)

El comportamiento inteligente puede surgir a partir de la capacidad de entrenamiento de redes de neuronas artificiales y el aprendizaje (no simbólico) por parte de las mismas, en procesos que imiten la evolución adaptativa.

1.4. Aplicaciones y logros

- **Planificación autónoma:** El programa de la NASA Agente Remoto se convirtió en el primer programa de planificación autónoma que controlaba la planificación de las operaciones a bordo de la propia nave.
- **Juegos:** Deep Blue de IBM fue el primer sistema que derrotó a un campeón mundial en una partida de ajedrez, ganando a Gary Kasparov.
- **Diagnosis:** Los programas de diagnóstico médico basados en el análisis probabilístico han llegado a alcanzar niveles similares de médicos expertos en algunas áreas de la medicina.
- **Planificación logística:** Durante la crisis del Golfo Pérsico, EEUU desarrolló la herramienta *Dynamic Analisis and Replaining Tool* para automatizar la planificación y organización logística del transporte.
- **Robótica:** HipNav es un sistema que usa técnicas de visión por computador para crear un modelo en 3D de la anatomía de un paciente, después con un control robotizado, guía el implante de prótesis de cadera.
- **Procesamiento del lenguaje y resolución de problemas:** PROVER B es un programa informático que resuelve crucigramas mejor que la mayoría de los humanos.

1.5. Campos de la Inteligencia artificial

- **Lógica:** Todo lo que un programa conoce acerca del mundo en general, los hechos de la situación especifica en la cual debe actuar y todos sus objetivos estan representados por sentencias en el lenguaje matemático-lógico. El programa decide qué hacer infiriendo esas acciones como sea necesario para alcanzar sus objetivos.
- **Búsqueda:** Los programas de IA suelen tener que examinar un gran número de posibilidades, continuamente se descubren nuevas formas de hacer las búsquedas de manera más eficiente.

- **Reconocimiento de patrones:** Cuando un programa realiza observaciones de algun tipo, suele estar programado para comparar lo que ve con un patrón. Por ejemplo, un programa de reconocimiento facial intenta encontrar un patrón en los ojos y la nariz en una imagen para encontrar una cara.
- **Representación:** Los hechos deben ser representados de algun modo, normalmente se utiliza el lenguaje matemático-lógico.
- **Inferencia:** A partir de unos hechos, se pueden inferir otros, la deducción matemático-lógica es adecuada para algunos propósitos, pero nuevos métodos de inferencia no monótona han sido añadidos a la lógica desde 1970.
- **Sentido común y Razonamiento:** Este área de la IA es la que más alejada se encuentra del nivel de un ser humano, a pesar de que ha sido un campo activo de la investigación desde 1950. Aunque ha alcanzado progresos considerables, como el desarrollo de sistemas de razonamiento no monótono.

La IA abarca muchos otros campos como por ejemplo *"Aprender de la experiencia", "Planificación", "Epistemología", "Ontología", "Heuristica"*, etc.

2. Búsqueda no heurística (o desinformada)

2.1. Formulación de problemas en espacios de estados

Los problemas que se van a tratar en este tema son los más sencillos. Se trata de situaciones en las que el agente tiene unos fines o metas, que puede perseguir buscando sin incertidumbre en la representación que posea sobre un micromundo, la representación consiste en un modelo formal del mismo (estructuras de datos sobre los que se pueda computar), que lo represente abstractamente en sus rasgos fundamentales (en relación con el problema) como un espacio de estados (estado que el agente puede identificar cuando está en ellos) donde uno o varios de los ellos son iniciales o de partida, desde los cuales se ha de llegar o encontrar, mediante transformaciones o cambios estado a uno o varios **estados meta**, solución u objetivo.
Los **estados meta** se identifican o valoran mediante algún o algunos test o criterios de que dispone el agente. Los **cambios de estado** se producen mediante la aplicación de **operadores** de cambio de estado o de **funciones** sucesor que determinan totalmente los estados a los que se puede pasar a partir de uno dado (estados sucesores del mismo), y que el agente conoce perfectamente, como conoce el coste (si existe) de dichos cambios.

1. Definir el espacio de estados.
2. Indicar el estado inicial o de partida.
3. Indicar el/los estados meta.
4. Especificar los operadores de transición o cambio de estado.
5. Indicar el coste de los cambios.
6. Test o criterios de identificación de soluciones.

2.2. Ejemplos de problemas, concepto de solución

Por **resolución del problema** se entiende el proceso de determinar la serie de acciones de cambio de estado que hay que realizar (el camino a seguir) para llegar desde un estado inicial a uno o varios estados meta, proceso sometido o no, según el contexto del problema, a la consecución de un coste mínimo (solución óptima) o a la obtención de estados que, de acuerdo con algún criterio de valoración dado, sean lo bastante satisfactorios.
Cuando un problema haya sido formulado, basta identificar cada estado con un nodo de un grafo, y cada operador de cambio de estado con la arista que une los nodos correspondientes al estado de partida y al de llegada, y se tendrá el problema convertido en uno de búsqueda en grafos.

2.2.1. Procedimiento general de búsqueda en grafos.

```
1: Procedure Búsqueda(G, S, EsMeta)
2:     Inputs
3:         G: grafo con nodos N  y arcos A
4:         S: conjunto de nodos iniciales
5:         EsMeta : función booleana de los estados (test de solución)
6:     Output
7:         Camino de un elemento de S a nodo para el que EsMeta de verdad
8:         o ⊥ (no encontrado camino) si no hay caminos solución
9:     Local
10:      ABIERTOS: conjunto de caminos(listas de nodos de cada camino)
11:    ABIERTOS ←{⟨s⟩: s∈S}
12:    while (ABIERTOS ≠{})
13:        select y remove ⟨s0,...,sk⟩ de ABIERTOS
14:        if ( EsMeta(sk)) then
15:            return ⟨s0,...,sk⟩
16:        ABIERTOS ←ABIERTOS  ∪{⟨s0,...,sk,s⟩: ⟨sk,s⟩∈A}
17:    return ⊥
```

Los diversos métodos de búsqueda (anchura, profundidad, optimal o de coste uniforme, etc.) se diferencian esencialmente en cómo se seleccione (línea 13 del algoritmo) el elemento de la lista ABIERTOS (tratándola como cola, como pila, como lista de prioridad según el coste acumulado, etc).

2.3. Principales métodos de búsqueda no heurística

Una vez definida la formalización del problema hay que buscar una solución (que puede ser un estado o estados finales, o puede ser un camino desde el nodo de inicio al nodo meta) rastreando o explorando el grafo de búsqueda.
Idea general de la búsqueda:

1. Situarse en el estado inicial y comprobar si es solución.
2. Si no, generar sus sucesores (estados a los que puede transitar) y:
   - Almacenarlos en una lista.
   - Examinarlos como posibles metas, en el orden que indique el método que estamos siguiendo.
   - Para cada estado de la lista, examinarlo como posible meta, presentando si lo es o si no. Generar sus sucesores y ponerlos en la lista.
3. Repetir el proceso.

Los métodos de búsqueda se distinguen unos de otros por la manera de gestionar los nodos generados.

2.3.1. Búsqueda en anchura

La lista ABIERTOS se gestiona como una cola, donde los caminos se van extrayendo de la lista en el orden en que entraron a ella (FIFO), y sus nodos terminales se van examinando (mediante la función test EsMeta) para presentar al camino como solución (si lo fuera) o generar los sucesores del nodo terminal e incorporarlos a la lista ABIERTOS.

- Inconveniente: Si no se guarda la información de los nodos ya visitados, se encarece mucho el proceso porque se repiten muchos casos. Una variante del problema sería incluir una lista de cerrados en la cual se añadirian los nodos ya visitados.

La búsqueda en anchura es **completa**, es decir, si existe solución, la va a encontrar.

2.3.2. Búsqueda en profundidad

La lista ABIERTOS se gestiona como una pila (LIFO). Existen variantes de la búsqueda en profundidad:

- **Profundidad acotada:** Se procede como en la búsqueda en profundidad, pero, hasta un límite preestablecido L para la misma, es decir, "podando" las ramas del árbol de búsqueda con longitud L.
- **Profundidad iterativa:** Se establece una sucesión creciente de limites "$L_1 < L_2 < L_3 < ...$ " y se repite, **desde el principio**, una serie de búsquedas en profundidad limitada L_i hasta que una de ellas encuentre la solución.

Nota: Tanto en la búsqueda en anchura como en la de profundidad y sus variantes, se da por supuesto que los costes de las aristas (es decir, de aplicar los operadores para transitar de unos nodos a otros) son siempre 1, de modo que los "gastos" hechos al recorrer los caminos dependen sólo del número de etapas de los mismos, que coincide con la profundidad de las ramas del árbol de búsqueda correspondientes.

2.3.3. Búsqueda de coste uniforme u optimal

Búsqueda llamada "optimal", "de menor coste" o "de coste uniforme", se aplica cuando los costes de las aristas entre nodos no son constantes, de modo que el gasto hecho al seguir los caminos no depende sólo del número de etapas de los mismos. La lista ABIERTOS se trata como una lista de prioridad, eligiendo primero para ser examinado con EsMeta aquel camino de la lista que menor gasto acumulado lleve.

2.3.4. Búsqueda bidireccional

La Búsqueda bidireccional *sólo es aplicable cuando se conocen los operadores de paso inversos* (es decir, se sabe cómo generar todos los predecesores de cada nodo) *y además es conocido explícitamente algún estado solución*. En esta situación, se podría resolver el problema del modo habitual, buscando desde el estado inicial hacia la meta, o también se podría buscar desde la meta hacia el inicial.

Pero una tercera posibilidad es hacer ambas búsquedas simultáneamente, en pasos alternados, hasta que los segmentos de caminos generados coincidan en un estado común. Esto ocurrirá necesariamente si por lo menos una de las dos búsquedas se efectúa en anchura, con lo cual siempre aparecen en su lista ABIERTOS nodos de cada uno de los caminos solución.
Podrian realizar las dos búsquedas en anchura, pero hacerlo combinando los dos tipos permite economizar gasto en memoria en la dirección en que el factor de ramificación sea mayor. En cualquier caso, el número de operaciones de examen y generación de sucesores disminuirá sustancialmente, al llegar cada una de las dos búsquedas sólo hasta la mitad de profundidad.
Esta ventaja en coste computacional hay que ponerla en balance frente a la desventaja de tener que comprobar en cada paso si ha aparecido alguna coincidencia entre los nodos de las dos listas de abiertos.

2.4. Características y comparación de soluciones

Para comparar la eficacia de los métodos se estudian generalmente las siguientes características:

- **Completitud:** un método es completo si siempre encuentra solución, cuando esta existe.
- **Optimalidad:** cuando encuentra una solución, esta es la más cercana al estado de partida, sea en número de pasos, sea en coste acumulado de recorrerlos cuando este no es constante.
- **Complejidad espacial o en memoria:** dada por la estimación del tamaño de las listas a gestionar y mantener, en función del factor de ramificación y la profundidad de las soluciones.
- **Complejidad temporal o en operaciones:** dada por la estimación de la cantidad de nodos o estados a generar y examinar, también en función del factor de ramificación y la profundidad de las soluciones.

Método	Gestión abiertos	Completo	Óptimo	C. Espacial	C. Temporal
Anchura	Cola	Si	Si (*1)	$O(\pi^{p+1})$	$O(\pi^{p+1})$
Profundidad	Pila	No (*2)	No	$O(\pi \cdot p)$	$O(\pi^{p+1})$
Prof. limit.	Pila	No	No	$O(\pi \cdot l)$	$O(\pi^{l+1})$
Prof. iterat.	Pila	Si	Si	$O(\pi \cdot p)$	$O(\pi^{p+2})$
Coste uniforme	Menor gasto	Si	Si		$O(\pi^{p+1})$

(*1) Como se hace por niveles, la primera solución que encuentre será la que este a menos pasos del inicio.
(*2) Se puede solucionar haciendo búsqueda en profundidad limitada, pero la solución puede estar fuera del límite. Tambien puede hacerse busqueda en profundidad iterativa.
El Coste Temporal es proporcional al numero de nodos examinados hasta encontrar la solución.
Intervienen:

- p $\rightarrow$ profundidad.
- $\pi \rightarrow$ ramificación (nº de nodos sucesores).
- l $\rightarrow$ límite establecido.

2.5. El problema de las jarras

Se tienen dos jarras, una con capacidad para 3L y otra con capacidad 4L. ¿Cómo llegar a tener en una de las jarras 2L?

Representación de la información

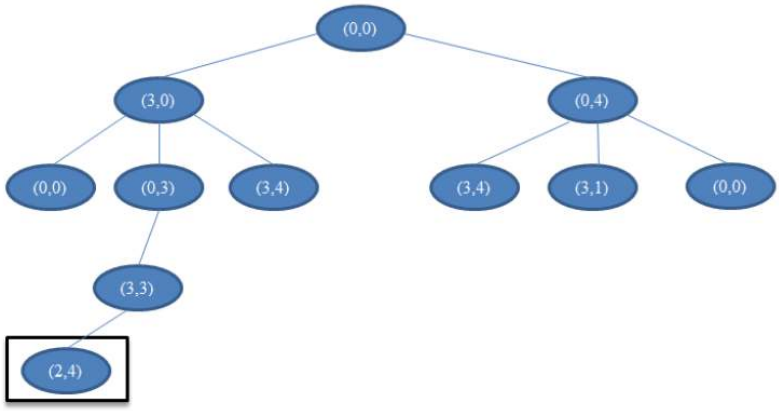
- Lista (X, Y)
- $X, Y \in$ Naturales
- $X = \{0,1,2,3,4\} \rightarrow$ Jarra grande
- $Y = \{0,1,2,3\} \rightarrow$ Jarra pequeña
- Estado inicial $= (0, 0)$
- Estado final $= (X, 2) \cup (2, Y)$

Operadores

- $(X, Y) \rightarrow (4, Y)$ Llenar la primera jarra
  si $X < 4$
- $(X, Y) \rightarrow (X, 3)$ Llenar la segunda jarra
  si $Y < 3$

- $(X, Y) \to (0, Y)$ Vaciar la primera jarra
  si $X > 0$
- $(X, Y) \to (X, 0)$ Vaciar la segunda jarra
  si $Y > 0$
- $(X, Y) \to (4, Y - (4 - X))$ Verter de la segunda hasta llenar la primera
  si $(X + Y \geq 4) \wedge (Y > 0) \wedge (X < 4)$
- $(X, Y) \to (X - (3 - Y), 3)$ Verter de la primera hasta llenar la segunda
  si $(X + Y \geq 3) \wedge (X > 0) \wedge (Y < 3)$
- $(X, Y) \to (X + Y, 0)$ Verter todo el agua de la segunda en la primera
  si $(X + Y \leq 4) \wedge (Y > 0)$
- $(X, Y) \to (0, X + Y)$ Verter todo el agua de la primera en la segunda
  si $(X + Y \leq 3) \wedge (X > 0)$

Resolución



3. Búsqueda heurística (o informada)

En los métodos de búsqueda no informada, la búsqueda se realizaba explorando sistemáticamente los nodos del grafo, a veces teniendo en cuenta lo ya gastado en la exploración de la parte del camino recorrido (coste uniforme), pero sin utilizar ninguna información disponible (que en ocasiones puede existir) sobre preferencia de unas vías sobre otras o sobre proximidad de las metas en las partes de los caminos aún por recorrer.
Cuando se dispone de alguna clase de información en este segundo sentido se habla de **heurísticas** y los métodos correspondientes son los **métodos heurísticos**.
En ellos lo que se hace es utilizar dicha información para elegir el desarrollo de unos estados frente a otros, tomando antes para desarrollar de la lista de abiertos aquellos que aparenten estar más cerca de la meta.

3.1. Funciones heurísticas. Búsqueda heurística

La información sobre proximidad a la meta consiste en una función numérica **h** del espacio de estados $[0, \infty) = \mathbb{R}^+ \cup \{0\}$, que estime la "distancia" de cada estado al objetivo más próximo, valiendo **0** en las metas.
Esta estimación es información incierta, y puede conducir a engaños. Al usarla se sacrifica certeza para obtener economía "podando" el árbol de búsqueda para centrarse en las ramas o direcciones más prometedoras.
Combinando la información heurística (sobre el futuro) con la del gasto realizado (sobre el pasado), se puede establecer métodos seguros y económicos, que combinan las ventajas respectivas de ambas informaciones.

3.2. Métodos voraces

3.2.1. Búsqueda en Escalada o irrevocable (Hill Climbing)

Esta búsqueda rudimentaria, sin memoria, consiste en seguir siempre el camino que, según lo apuntado por la función heurística, parezca mejor sucesor, mientras éste mejore al estado en que se esté:

1. Empezar con ACTUAL (Lista formada por el estado inicial).
2. Hasta que ACTUAL = meta o no haya cambios en ACTUAL, hacer:
   a) Si su estado final es una meta presentar la lista de ACTUAL y parar
   b) Si no, tomar los sucesores de ACTUAL y usar h para puntuar cada uno de ellos.
   c) Si uno de los sucesores tiene mejor puntuación que ACTUAL, hacer ACTUAL igual a la lista anterior aumentada en el sucesor.
3. Presentar FALLO y terminar

3.2.2. Búsqueda Primero el mejor

Como la búsqueda en escalada, usa la heurística para elegir siempre al nodo sucesor aparentemente mejor situado, pero este método usa memoria (manteniendo abiertas las posibilidades alternativas mediante una lista de ABIERTOS) teniendo así varios caminos posibles, no sólo uno sin alternativas.
Funciona de modo análogo al de la búsqueda desinformada *optimal* gestionando la lista ABIERTOS como una cola de prioridad, pero usando sólo la función heurística h para ordenarla y priorizar, examinando antes los nodos que tengan mejor valor heurístico.
La obtención de resultado, su calidad, dependerá de lo fidedigna que sea la función heurística.

1. Empezar con ABIERTOS (Lista formada por el estado inicial).
2. Mientras ABIERTOS no esté vacío hacer:
   a) Quitar de ABIERTOS la lista con el mejor nodo terminal y ponerla en ACTUAL.
   b) Si el nodo terminal de ACTUAL es meta, presentar su lista y terminar.
   c) Si no, tomar los sucesores de dicho nodo, usar h para puntuar cada uno de ellos e incorporar las listas resultantes de ampliar ACTUAL con dichos nodos a la lista ABIERTOS en orden.
3. Presentar FALLO y terminar.

Si se dispone de una buena heurística, la búsqueda primero el mejor puede lograr en la práctica sustanciales descensos en el coste computacional de su ejecución, pero no queda garantizado que el camino obtenido sea óptimo. Hay que tener además en cuenta el coste que suponga la evaluación de h en cada nodo.

3.2.3. El algoritmo A^*

La idea en la que se basa la búsqueda A^* consiste en intentar combinar las ventajas de la búsqueda desinformada de coste uniforme (completitud y optimalidad) con las de la búsqueda heurística *primero el mejor* (eficiencia, por la poda del árbol de búsqueda).
Esto se consigue priorizando en la lista de ABIERTOS aquellos caminos en que resulte menor una combinación de los valores del gasto hecho en la parte del camino recorrido (suma de los costes de sus aristas) con la distancia restante hasta la meta según la información dada por h en el trozo de camino por recorrer.

$$f(n) = g(n) + h(n)$$

Siendo $g(n)$ el coste para alcanzar el nodo n desde el inicio y $h(n)$ el coste estimado para alcanzar la meta desde n.

1. Empezar con ABIERTOS (Lista formada por el estado inicial).
2. Mientras ABIERTOS no esté vacío hacer:
   a) Quitar de ABIERTOS la lista con el mejor nodo terminal y ponerla en ACTUAL.
   b) Si el nodo terminal de ACTUAL es *meta*, presentar su lista y terminar.
   c) Si no, tomar los sucesores de dicho nodo, usar f para puntuar cada uno de ellos e incorporar las listas resultantes de ampliar ACTUAL con dichos nodos a la lista ABIERTOS en orden.
3. Presentar FALLO y terminar.

Si la heurística es adecuada, con este método se consigue un algoritmo completo, óptimo y con un coste computacional menor, equivalente en la práctica una disminución del factor de ramificación.

3.3. Propiedades de las heurísticas

Se denota con h^* a la heurística ideal, que daría la información perfecta sobre el coste óptimo, por el mejor camino, desde cada nodo hasta la meta más cercana, y con $f^*(n) = g(n) + h^*(n)$ al coste total de la mejor solución que pase por n.
Dada una posible función heurística h, puede ocurrir:

a) Que $h(n) = 0$ para todo n, es decir, que la heurística no dé información. En este caso, A^* se convierte en la *búsqueda optimal*.

b) Que $h(n)=h^*(n)$ para cada n, en cada paso se sabría qué dirección hay que seguir para llegar a la meta más cercana del inicio por el mejor camino.

c) Que $h(n)>h^*(n)$ para algún n. Entonces A^* puede no resultar óptimo.

d) Que $h(n)\leq h^*(n)$ para cada n. Entonces A^* termina y resulta ser completo, óptimo y con coste computacional tanto más reducido cuanto mayor sea h, oscilando entre la búsqueda no heurística de *coste uniforme* del caso a) y la ausencia de búsqueda del caso b). Estas heurísticas se denominan **admisibles**.

Una heurística h es **más informativa** que otra h**'** si cumplen $\mathbf{0} \leq h'(n) \leq h(n) \leq h(n)^*$ para cada n. Cuanto más informativa sea una heurística, más económica resultará la búsqueda mediante ella con A^*, es decir, menos nodos habrá que examinar para encontrar la solución.
Una heurística es **consistente** cuando para cada par de nodos adyacentes n y n' con el segundo sucesor del primero cumple que $h(n)\text{-}h(n') \leq C(n,n')$.
Es **monótona** si se cumple que siempre que n' sea un sucesor de n se tendrá que

f(n') ≥ f(n).

Cuando una heurística es consistente, entonces también es monótona y se cumple que la primera vez que un nodo sea escogido de la lista ABIERTOS para ser examinado, se habrá llegado desde el inicial hasta él por el camino más corto posible (es decir, que en la aplicación de A^* no habrá que "rectificar" ningún camino).

3.4. Relajación y diseño de heurísticas

A un problema con menos restricciones en las acciones se le llama **problema relajado**. *El coste de una solución óptima en un problema relajado es una heurística admisible para el problema original.* La heurística es admisible porque la solución óptima en el problema original es, por definición, una solución en el problema relajado y por lo tanto debe ser al menos tan cara como la solución óptima del problema relajado.
Un problema con la generación de nuevas funciones heurísticas es que a menudo se falla al conseguir una heurística "claramente mejor". Si tenemos disponible un conjunto de heurísticas admisibles h_1, h_2, h_m para un problema y ninguna de ellas predomina sobre las otras, podemos obtener lo mejor de ellas definiendo una nueva heurística

$$h(\mathrm{n}) = \max\{h_1(\mathrm{n}),...,h_m(\mathrm{n})\}$$

Como las heurísticas componentes son admisibles, h es admisible, también se puede demostrar que h es consistente, y además, h predomina sobre todas sus heurísticas componentes.

3.5. Búsqueda heurística con memoria limitada

3.5.1. Algoritmo IDA* o A*PI

La forma más simple de reducir la exigencia de memoria para A^* es adaptar la idea de la profundización iterativa al contexto de búsqueda heurística, resultando así el A^* de Profundidad Iterativa. La principal diferencia entre A*PI y la profundidad iterativa estándar es que el corte utilizado es el f-coste $(g+h)$ más la profundidad. En cada iteración, el valor del corte es el f-coste más pequeño de cualquier nodo que excedió el corte en la iteración anterior.

3.5.2. Algoritmo SMA* o A*MS

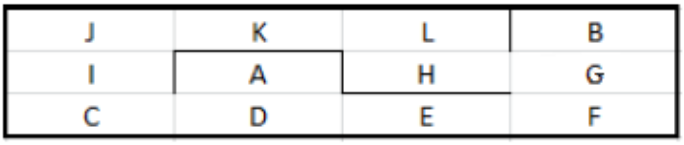
A*PI sufre de utilizar *muy poca* memoria, entre iteraciones, sólo conserva un número, el límite del f-coste actual. Dos algoritmos que usan toda la memoria disponible son A*M (A^* con Memoria acotada) y A*MS(A*M Simplificado).
A*MS avanza como A^*, expandiendo la mejor hoja hasta que la memoria esté llena. En este punto A*MS retira el peor nodo hoja (f-coste más alto), entonces

devuelve a su padre el valor del nodo olvidado, de esta forma, el antepasado sabe la calidad del mejor camino en el subárbol. Con esta información A^*MS vuelve a generar el subárbol sólo cuando *todos los otros caminos* parecen peores que el olvidado. Dicho de otra forma, si todos los descendientes de n son olvidados, no sabremos por qué camino ir desde n, pero aun tendremos una idea de cuánto vale la pena ir desde n a cualquier nodo.
A^*MS es completo si hay alguna solución alcanzable, es decir si la profundidad del nodo objetivo más superficial es menor que el tamaño de memoria. Es óptimo si cualquier solución óptima es alcanzable.
Sobre un problema muy dificil a A^*MS se le fuerza a ir hacia delante y hacia atras continuamente, el tiempo extra requerido para repetir los subárboles olvidados pueden hacer a *un problema intratable desde el punto de vista de tiempo de cálculo.*

3.6. Búsqueda local

En muchos problemas, el camino al objetivo es irrelevante, y sólo importa la configuración final, para estos problemas podemos considerar una clase diferente de algoritmos que no se preocupen de los caminos. Los **algoritmos de búsqueda local** funcionan con un sólo **estado actual**.
Estos algoritmos no son sistemáticos, pero tienen dos ventajas clave, usan muy poca memoria y pueden encontrar a menudo soluciones razonables en espacios de estados grandes o infinitos, para los que los algoritmos sistemáticos son inadecuados.

3.7. El problema del laberinto



- Estado inicial = A $(2, 2)$
- Estado final = B $(1, 4)$

Las transiciones posibles son a nodos adyacentes en los que no haya "muro". Una heurística posible sería usando las cordenadas del estado actual y el estado meta, para orientar la búsqueda.

Búsqueda en profundidad (con lista de CERRADOS)

ACTUAL	ABIERTOS	SOLUCIONES
A	D	D
D	A,E,C	A,E,C
C	I,D	A,E,I,D
I	J,C	A,E,J,C
J	K,I	A,E,K,I
K	J,L	A,E,J,L
L	K,H	A,E,J,K,H
H	L,G	A,E,J,K,L,G
G	H,F,B	A,E,J,K,L,H,F,B
B	G	

9 pasos → A,D,C,I,J,K,L,H,G,B

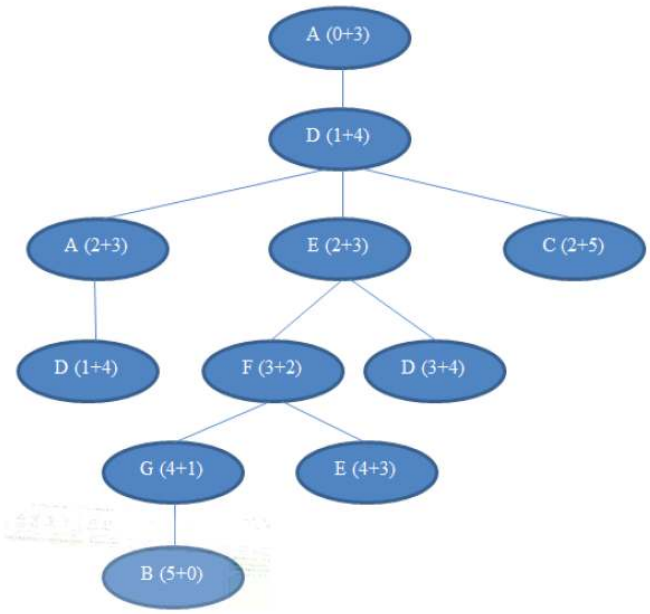
Búsqueda de coste uniforme (sin lista de CERRADOS)

ACTUAL	ABIERTOS	SOLUCIONES
A	D	D
D	A,E,C	A,E,C
A		

A*

Usando como función heurística la distancia al nodo si no hubiese "muros"

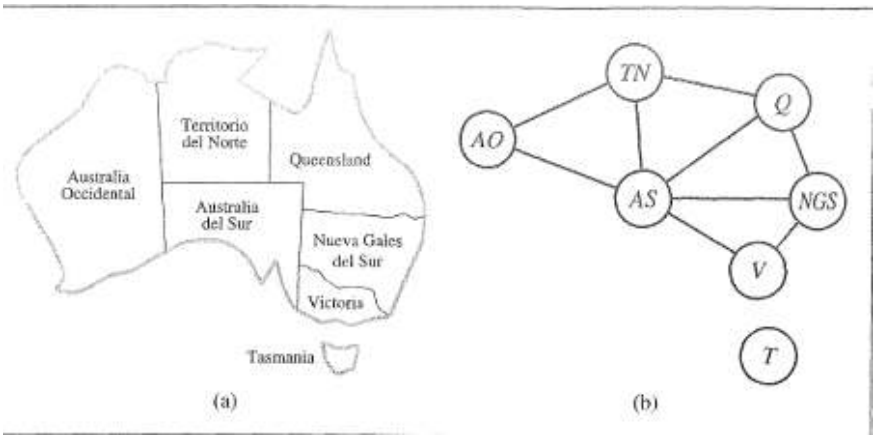
ESTADO	$h(n)$
A	3
B	0
C	5
D	4
E	3
F	2
G	1
H	2
I	4
J	3
K	2
L	1



4. Problemas de Satisfacción de Restricciones (P.S.R.)

Formalmente, un **problema de satisfaccion de restricciones** está definido por un conjunto de **variables** $X_1, X_2, ..., X_n$ y un conjunto de **restricciones** $C_1, C_2, ..., C_m$. Cada variable X_i tiene un **dominio** no vacío D_i de **valores posibles**. Cada restricción C_i implica algún subconjunto de variables y especifica las combinaciones aceptables de valores para este subconjunto. Un **estado** del problema está definido por una **asignación** de valores a una o a todas las variables, $\{X_i = v_i, X_j = v_j, ...\}$. A una asignación que no viola ninguna restricción se le llama **asignación consistente** o legal. Una asignación completa es una asignación en la que se menciona cada variable, y una **solución** de un PSR es una asignación completa que satisface todas las restricciones. Algunos PSRs también requieren una solución que maximiza una **función objetivo**.

Es bueno visualzar un PSR como un **grafo de restricciones**. Los nodos del grafo corresponden a variables y los arcos a restricciones.



A un PSR se le puede dar una **formulación incremental** como en un problema de búsqueda estándar si:

- **Estado inicial:** La asignación vacía {}, en la ninguna variable está asignada.
- **Función de sucesor:** Un valor se puede asignar a cualquier variable no asignada, a condición de que no suponga ningún conflicto con variables previamente asignadas.
- **Test objetivo:** La asignación actual es completa.
- **Costo del camino:** Un coste constante para cada paso.

Como cada solución debe ser una asignación completa, estas aparecen a profundidad n siendo n el número de variables, además, el árbol de búsqueda sólo se extiende a profundidad n, por ello, los algoritmos de búsqueda en profundidad son populares para estos problemas.
El camino que alcanza la solución es irrelevante, de ahí que podamos usar una **formulación completa de estados**, en la que cada estado es una asignación completa que podría satisfacer o no las restricciones. Los métodos de búsqueda local trabajan bien para esta formulación.

4.1. Tipos de P.S.R.

4.1.1. Por tipo de dominio:

Discreto
La clase más simple de PSR implica variables **discretas** y **dominios finitos**. Si el tamaño máximo del dominio de cualquier variable en un PSR es d, entonces el número de posibles asignaciones completas es $O(d^n)$, es decir, exponencial en el número de variables.
Los PSR de dominio finito incluyen a los **PSRs booleanos** cuyas variables pueden ser $verdaderas$ o $falsas$, estos incluyen como casos especiales algunos problemas NP-completos.
En el caso peor no podemos esperar resolver los PSRs con dominios finitos en menos de un tiempo exponencial. A pesar de ello, se pueden resolver problemas de órdenes de magnitud mayores que los resolubles con los algoritmos de búsqueda general (no heurística).
Las variables discretas pueden tener también **dominios infinitos**, con estos no es posible describir restricciones enumerando todas las combinaciones permitidas de valores, por ello se debe usar un **lenguaje de restricción**. Existen algoritmos de solución especiales para **restricciones lineales** sobre variables enteras.

Continuo
Los problemas de satisfacción de restricciones con **dominios continuos** son muy comunes y son ampliamente estudiados en el campo de la investigación operativa. La categoría más conocida son los problemas de **programación lineal**, en donde las restricciones deben ser desigualdades lineales que forman una región *convexa*. Estos problemas pueden resolverse en tiempo polinomial en función del número de variables.

4.1.2. Por el número de variables en las restricciones:

Restricción unaria
El caso más simple es la restricción unaria que restringe los valores de una sola variable.

Restricción binaria
Una restricción binaria relaciona dos variables. Un PSR binario es un problema

con restricciones sólo binarias y puede representarse como un grafo de restricciones.

Restricción n-aria
Las restricciones de alto nivel o n-arias implican tres o más variables. Estas pueden representarse en un **hiper-grafo de restricciones**. No obstante, si se introducen suficientes variables auxiliares se puede transformar cualquier problema de dominio finito y orden n-ario en un conjunto de relaciones binarias.

Un ejemplo de restricciones de alto nivel es el que nos dan los **puzles cripto-aritméticos**.

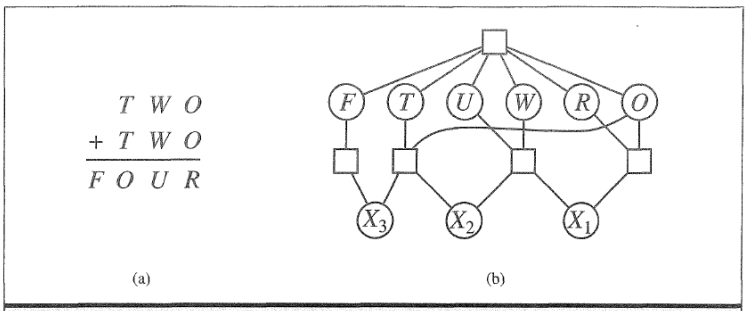


Figura 5.2 (a) Un problema cripto-aritmético. Cada letra significa un dígito distinto; el objetivo es encontrar una sustitución de las letras por dígitos tal que la suma que resulta es aritméticamente correcta, con la restricción añadida de que no se permite la administración de ceros. (b) El hiper-grafo de restricciones para el problema cripto-aritmético muestra la restricción *TodasDif* así como las restricciones añadidas en las columnas. Cada restricción es una caja cuadrada relacionada con las variables que restringe.

El de la imagen puede ser representado como la restricción de seis variables $TodasDif(F, T, U, W, R, O)$, o puede representarse como una colección de restricciones binarias del tipo $F \neq T$. Las restricciones añadidas sobre las columnas del puzle también implican variables, y pueden representarse por:

$O + O = R + 10 \cdot X_1$
$X_1 + W + W = U \cdot X_2$
$X_2 + T + T = O \cdot X_3$
$X_3 = F$

Donde X_1, X_2 y X_3 son **variables auxiliares** que representan el dígito 0 o 1, que se transfiere a la siguiente columna.

4.1.3. Por el tipo de restrición:

Las restricciones hasta ahora descritas han sido todas restricciones **absolutas**, la violación de las cuales excluye una solución.
Muchos PSR incluyen restricciones de **preferencia** que indican que soluciones

son preferidas. Las restricciones de preferencia pueden codificarse como costos sobre las asignaciones de variables individuales. Con esta formulación los PSR con preferencia pueden resolverse utilizando métodos de búsqueda de optimización, basados en caminos o locales.

4.2. Los P.S.R. como problemas de búsqueda en espacio de estados

Un PSR queda convertido de modo natural en un problema de búsqueda en espacio de estados, resoluble de forma incremental si se toma:

- **Como estados:** las asignaciones parciales.
  - Inicial = asignación nula.
  - Meta = asignación completa.
- **Como transiciones entre estados:** la asignación a una variable aún no asignada, hecha respetando todas las restricciones.

Ventajas:

- La formulación es automática, idéntica para todos los PSR.
- No importa el orden de las variables en la asignación.
- No importa el camino seguido
- No pueden aparecer estados repetidos en la búsqueda
- La profundidad es limitada, igual al número de variables

Solución incremental: Es natural resolverlo por búsqueda en profundidad con vuelta atrás cuando haya "fallo", sin que sea necesario usar lista de cerrados (*algoritmo de backtracking*).

4.3. Solución incremental, búsqueda en profundidad con vuelta atras

Formulando los PSR como problemas de búsqueda, el factor de ramificación que se obtiene es demasiado alto. Esto es porque no se aprovecha la **conmutatividad** de los PSR.
Un problema es conmutativo si el orden de aplicación de cualquier conjunto de acciones no tiene ningun efecto sobre el resultado.
El término **búsqueda con vuelta atrás** se utiliza para la búsqueda en profundidad que elige valores para una variable a la vez y vuelve atras cuando una variable no tiene ningún valor legal para asignarle. Hay distintas formas de resolver un PSR de búsqueda con vuelta atras, y debemos elegir una heurística genérica para mejorar su eficiencia, basándonos sólo en la estructura del problema.

4.3.1. Heurísticas naturales en el método de vuelta atras

En la resolución incremental de PSR (que parece una búsqueda desinformada) se pueden considerar heurísticas naturales (genéricas) basadas sólo en la estructura del problema que mejoran su eficiencia:

Orden de asignación a las variables

La heurística de **mínimos valores restantes** (MVR) escoge la variable que con mayor probabilidad causará un fracaso (es decir, prioriza aquellos nodos cuyos valores posibles sean menores), con lo cual podamos el árbol de búsqueda. Si hay una variable X con cero valores legales restantes, la heurística MVR seleccionará X y el fallo será descubierto evitando búsquedas inútiles por otras variables.

El **grado heurístico** intenta reducir el factor de ramificación sobre futuras opciones en caso de empate en el MVR seleccionando la variable que esté implicada en el mayor número de restricciones.

La heurística del **valor menos restringido** como contraposición al grado heurístico, que podría ser útil en determinados casos, trata de dejar la máxima flexibilidad a las asignaciones de las siguientes variables utilizando aquellas que menos valores legales eliminan de los nodos adyacentes.

Propagación de información mediante las restricciones

Mirando las restricciones en la búsqueda, o antes de que haya comenzado esta podemos reducir drásticamente el espacio de esta.

- **Comprobación hacia delante:** Siempre que se asigne una variable X, este proceso de comprobacion mira cada variable no asignada Y que esté relacionada con X por una restricción y suprime del dominio de Y cualquier valor que sea inconsistente con el valor elegido de X.

	AO	*TN*	*Q*	*NGS*	*V*	*AS*	*T*
Dominios iniciales	R V A	R V A	R V A	R V A	R V A	R V A	R V A
Después de *AO=rojo*	Ⓡ	V A	R V A	R V A	R V A	V A	R V A
Después de *Q=verde*	Ⓡ	A	Ⓥ	R A	R V A	A	R V A
Después de *V=azul*	Ⓡ	A	Ⓥ	R	Ⓐ		R V A

Figura 5.6 Progreso de una búsqueda, que colorea un mapa, con comprobación hacia delante. *AO = rojo* se asigna primero; entonces la comprobación hacia delante suprime *rojo* de los dominios de las variables vecinas *TN* y *AS*. Después *Q = verde*; *verde* se suprime de los dominios *TN*, *AS*, y *NGS*. Después *V = azul*; *azul* se suprime del dominio de *NGS* y *AS*, y se obtiene AS sin valores legales.

- **Arco consistente:** Ofrece una buena relación entre eficiencia y completitud. Se refiere a un arco dirigido en el grafo de restricciones. La comprobación de la consistencia del arco puede aplicarse como un paso de preproceso antes de comenzar la búsqueda o como un paso de propagación después de cada asignación durante la búsqueda. Este proceso debe repetirse hasta que no permanezcan inconsistencias.

Vuelta atrás inteligente
Hasta ahora el algoritmo avanzaba hasta encontrar un fallo y entonces volvía al nodo anterior y lo intentaba con un valor diferente, este método se conoce como **vuelta atrás cronológica**.
Una aproximación más inteligente a la vuelta atrás cronológica es retroceder hasta el conjunto de variables que causaron el fracaso, llamado **conjunto conflicto**. El conjunto conflictivo de una variable X es el conjunto de variables previamente asignadas que tienen relación con X mediante alguna restricción. El método **salto-atrás** retrocede a la variable *más reciente* en el conjunto conflicto. Este sistema es redundante con una búsqueda de comprobación hacia delante. En resumen: Sea X_j la variable actual y $conf(X_j)$ su conjunto conflicto. Si para todo valor posible para X_j falla, saltamos atrás a la variable más reciente en X_i, en $conf(X_j)$, y el conjunto

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\}$$

El **salto-atras-dirigido-por-conflicto** va hacia atrás derecho al punto en el árbol de búsqueda, pero no nos impide cometer los mismos errores en otra rama. **Aprender la restricción** modifica el PSR añadiendo una nueva restricción inducida por estos conflictos.

4.4. Búsqueda local

Los algoritmos de búsqueda local resultan ser muy eficaces para resolver PSR.
Parten de la precondición de que una solución parcial puede incluir un conflicto. Esta nueva formulación de estados quedaría así:

- **Estado inicial:** Escogido al azar.
- **Función sucesor:** Elige una variable y trabaja cambiando su valor.
- **Test objetivo:** Asignaciones completas (consistentes o no).

En la elección de un nuevo valor para una variable, la heurística más obvia será seleccionar el valor que cause menos conflictos (heurística de **mínimos conflictos**). Estos son muy eficaces en PSR cuando se ha dado un estado inicial razonable. La generación de los sucesores mediante la herística de mínimos conflictos sería:

- **Variable elegida:** Aquella (distinta de la última modificada) que participa en más restricciones no satisfechas en el estado.
- **Valor elegido:** Aquel valor (distinto del que tenía) que produce el menor número de restricciones incumplidas.

4.5. Aprovechamiento de la estructura de los problemas

Si cualquier solución para una parte del PSR combinado con cualquier solución para la otra parte del PSR es solución para el PSR, estos son **subproblemas independientes**, pero en la mayoría de los casos los subproblemas de un PSR están relacionados.
El caso más simple de resolver es cuando el grafo de restricciones forma un **árbol**, dos variables cualesquiera están relacionadas a lo sumo por un camino. *Cualquier PSR estructurado por árbol puede resolverse en un tiempo lineal al número de variables.*

1) Elegimos cualquier variable como raíz, y ordenamos las variables de la raíz a las hojas de tal forma que el padre de cada nodo en el árbol lo precede en el ordenamiento.

2) Para j desde n hasta 2, aplicar la consistencia de arco a arco (X_i, X_j), donde X_i es el padre de X_j, quitando los valores del DOMINIO $\{X_i\}$ que sea necesario.

3) Para j desde 1 a n, asigne cualquier valor para X_j consistente con el valor asignado para X_i, donde X_i es el padre de X_j.

Ahora que tenemos un algoritmo eficiente para árboles, podemos considerar si los grafos de restricción más generales pueden *reducirse* a árboles de alguna manera.

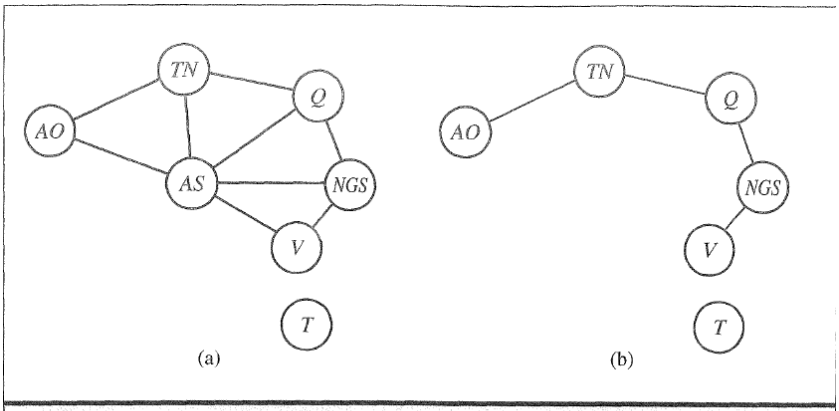


Figura 5.11 (a) Grafo restricción original de la Figura 5.1. (b) Grafo restricción después de quitar *AS*.

La primera forma de hacer esto implica valores de asignación a algunas variables de manera que las restantes formen un árbol, el algoritmo general para ello es el siguiente:

1) Elegir un subconjunto S de VARIABLES {psr} tal que el grafo de restricciones se convierta en un árbol después de quitar S. Llamamos a S un **ciclo de corte**.

2) Para cada asignación posible a las variables en S que satisface todas las restricciones sobre S,

   a) quitar de los dominios de las variables restantes cualquier valor que sea inconsistente con la asignación para S, y,

   b) si el PSR restante tiene una solución, devolverla junto con la asignación para S.

La segunda aproximación esta basada en la construcción de una **descomposición en árbol** del grafo restricción en un conjunto de subproblemas relacionados. Cada subproblema se resuelve independientemente y las soluciones que resultan son combinadas. Una descomposición del árbol debe satisfacer las siguientes exigencias:

1) Cada variable en el problema original aparece en al menos uno de los subproblemas.

2) Si dos variables están relacionadas por una restricción en el problema original, deben aparecer juntas (junto a la restricción) en al menos uno de los subproblemas.

3) si una variable aparece en dos subproblemas en el árbol, debe aparecer en cada subproblema a lo largo del camino que une a esos subproblemas.

Resolvemos cada subproblema de manera independiente, si alguno de ellos no tiene solución, sabemos que el problema entero no tiene solución. Si podemos resolver todos los subproblemas, entonces intentamos construir una solución global tratando a cada subproblema como una "megavariable" cuyo dominio es el conjunto de todas las soluciones para el subproblema.

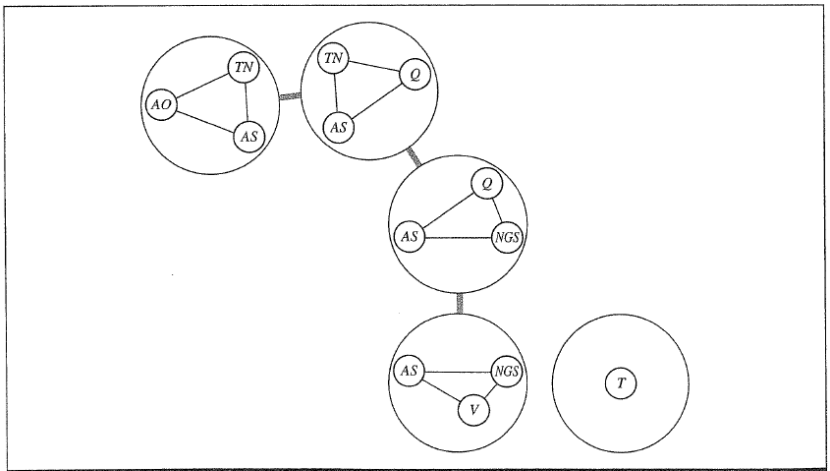


Figura 5.12 Una descomposición en árbol del grafo restricción de la Figura 5.11(a)

El grafo de restricciones admite muchas descomposiciones en árbol: el objetivo de la elección de una descomposición es hacer los subproblemas tan pequeños como sea posible. La **anchura del árbol** de una descomposición en árbol de un grafo es menor que el tamaño del subproblema más grande, la anchura de árbol del grafo en si mismo está definida por la anchura del árbol mínimo entre todas sus descomposiciones en árbol. De ahí que *los PSR con grafos de restricciones de anchura de árbol acotada son resolubles en tiempo polinomial.*

5. Juegos (Búsqueda con adversarios)

5.1. Los juegos como problemas de búsqueda reactiva o con adversarios

Los entornos competitivos en los cuales los objetivos del agente están en conflicto, dan ocasión a problemas de **búsqueda entre adversarios** a menudo conocidos como **juegos**. La **teoría matemática de juegos** ve a cualquier entorno multiagente como un juego a condición de que el impacto de cada agente sobre los demás sea significativo, sin tener en cuenta si los agentes son **cooperativos** o **competitivos**.
En IA los juegos son entornos deterministas, totalmente observables en los cuales hay dos agentes cuyas acciones deben alternar y que los valores utilidad, al final del juego, son siempre iguales y opuestos.

5.1.1. Tipos de juegos

Los juegos pueden clasificarse según distintas características:

- Número de jugadores.
- Con una cantidad de jugadas finita o no en cada turno.
- Con información perfecta o no (Ajedrez o poker).
- Con intervención o no del azar.
- Con posibilidad o no de formar coaliciones.
- De suma cero o no

Los teóricos de juegos denominan **juegos de suma cero** a juegos de dos jugadores, por turnos, deterministas y de **información perfecta**.

5.2. Decisiones óptimas en juegos

Consideraremos juegos con dos jugadores, llamados MAX y MIN. MAX mueve primero y se van turnando hasta que el juego termina. Al final del juego, se conceden puntos al ganador y se penaliza al perdedor. Una definición formal de un juego puede ser una clase de problemas de búsqueda con los siguientes componentes:

- El **estado inicial** que incluye la posición del tablero e identifica al jugador que mueve.
- Una **función sucesor**, que devuelve una lista de pares (*movimiento, estado*), indicando un movimiento legal y el estado resultante.

- Un **test terminal** que determina cuando se termina el juego. A los estados donde el juego ha acabado se les denomina **estados terminales**.
- Una **función utilidad** que da un valor numérico a los estados terminales.

El **árbol de juegos** queda definido por el estado inicial y los movimientos legales para cada lado.

5.2.1. Estrategias óptimas

MAX debe encontrar una **estrategia** que especifique el movimiento de MAX en el estado inicial, en respuesta a cada movimiento de MIN, en respuesta a la respuesta de MIN respecto al movimiento anterior, etc.
Considerando un árbol de juegos, la estrategia óptima puede determinarse examinando el **valor minimax** de cada nodo, este corresponde con la utilidad para MAX de estar en el estado correspondiente, *asumiendo que ambos juegan óptimamente*. Entonces MAX se querrá mover hacia un estado de valor máximo y MIN hacia uno de valor mínimo.

$$\text{VALOR-MINIMAX}(n) = \begin{cases} \text{UTILIDAD}(n) & \text{si } n \text{ es un estado terminal} \\ \max_{s \in \text{Sucesores}(n)} \text{VALOR-MINIMAX}(s) & \text{si } n \text{ es un estado MAX} \\ \min_{s \in \text{Sucesores}(n)} \text{VALOR-MINIMAX}(s) & \text{si } n \text{ es un estado MIN} \end{cases}$$

5.2.2. El algoritmo minimax

El **algoritmo minimax** calcula la decisión minimax del estado actual. Usa un cálculo simple recurrente de los valores minimax de cada sucesor, implementando directamente las ecuaciones de la definición. La recursión avanza hacia las hojas del árbol, entonces los valores minimax **retroceden** por el árbol cuando la recursión se va deshaciendo.
Primero realiza una exploración en profundidad completa del árbol de juegos, si la profundidad máxima es m y hay b movimientos legales en cada punto, entonces la complejidad en tiempo del algoritmo es $O(b^m)$. La complejidad en espacio para un algoritmo que genere todos los sucesores a la vez sería $O(bm)$ y para el que los genere uno por uno $O(m)$.

5.3. Minimax con poda alfa-beta. Eficiencia de la poda alfa-beta

El problema de minimax es que el número de estados que tiene que examinar es exponencial en el número de movimientos, pero es posible calcular la decisión minimax correcta sin mirar todos los nodos en el árbol de juegos. La técnica se denomina **poda alfa-beta** y puede aplicarse a árboles de cualquier profundidad y a menudo es posible podar subárboles enteros. El principio general es: considere un nodo n en el árbol, tal que el jugador tiene una opción de movimiento a

ese nodo. Si el jugador tiene una mejor selección m en cualquier punto, entonces *n nunca será alcanzado en el juego actual*. Este método recibe su nombre de los dos parámetros que describen los valores que aparecen a lo largo del camino:

- α Es el valor de la mejor opción para MAX
- β Es el valor de la mejor opción para MIN

Propagacion de los valores α-β

- Inicialmente se dan los valores más críticos:
  - -∞ como α y ∞ como β.
  - Estos valores se dan en la raíz y en los sucesores asignados que no sean hojas
- El valor α-β de una hoja es la puntuación de la misma.
- En nodos de nivel MAX se actualiza α de manera que $\alpha = \max(\alpha, \textit{valor } \alpha\text{-}\beta(hijo))$ mientras $\alpha < \beta$ y queden hijos.
  Si se acaban los hijos se devuelve α. Si en algún momento $\alpha \geq \beta$, se para y se devuelve β como valores α-β del nodo (condición de corte).
- En nodos de nivel MIN se actualiza β de manera que $\beta = \min(\beta, \textit{valor } \alpha - \beta(hijo))$ mientras $\alpha < \beta$ y queden hijos.
  Si se acaban los hijos se devuelve β. Si en algún momento $\alpha \geq \beta$, se para y se devuelve α como valores α-β del nodo (condición de corte).

La eficacia de la poda alfa-beta es muy dependiente del orden de generación de los nodos.