

Introducción a la Inteligencia Artificial

Diciembre, 2016

ÍNDICE GENERAL

1. Introducción a la Inteligencia Artificial	1
1.1. Definición.	1
1.2. Breve historia de la AI.	2
1.3. Aplicaciones de la AI.	3
1.4. Sistemas y agentes inteligentes.	5
1.4.1. Agentes reactivos.	6
1.4.2. Agentes planificadores.	9
 I Búsqueda	 11
2. Resolución de problemas mediante búsqueda	13
2.1. Introducción.	13
2.2. Agentes que <i>piensan</i> frente a agentes <i>reactivos</i>	13
2.3. Grafos de espacio de estados.	15
2.4. Terminología de grafos.	17
2.5. Problema de búsqueda.	18
2.5.1. El espacio de estados.	21
2.6. Grafos del espacio de estados y árboles de búsqueda.	22
2.7. Búsqueda en árbol	25
 3. Búsqueda primero en profundidad. Búsqueda primero en anchura. Búsqueda con profundidad iterativa.	 29
3.1. Búsqueda primero en profundidad (DFS).	29
3.1.1. Propiedades del algoritmo de búsqueda primero en profundidad (DFS). . .	30
3.1.1.1. Complejidad temporal (nodos a expandir).	32
3.1.1.2. Complejidad espacial (tamaño de la frontera).	32
3.1.1.3. Completitud.	32
3.1.1.4. Optimalidad.	32
3.2. Búsqueda primero en anchura.	33
3.2.1. Propiedades del algoritmo de búsqueda primero en anchura (BFS). . . .	34
3.2.1.1. Complejidad temporal.	34
3.2.1.2. Complejidad espacial.	35

3.2.1.3. Completitud.	35
3.2.1.4. Optimalidad.	35
3.3. Búsqueda en profundidad iterativa.	35
3.3.1. Justificación.	36
4. Búsqueda de coste uniforme (UCS).	37
4.1. Búsqueda de coste uniforme.	37
4.2. Propiedades de la UCS.	39
4.3. Resumen de algoritmos de búsqueda no informada.	40
4.4. Estados repetidos: búsqueda en grafos.	43
5. Búsqueda informada.	49
5.1. Heurística de búsqueda.	49
5.2. Búsqueda avariciosa o voraz.	51
5.3. Búsqueda A*.	52
5.4. Heurísticas admisibles. Optimalidad de A*.	54
5.5. Propiedades de A*.	57
5.6. Aplicaciones de A*.	57
5.7. Problemas relajados. Construcción de heurísticas admisibles.	58
5.8. Comparación de heurísticas.	60
5.9. Estados repetidos: búsqueda en grafos.	60
5.10. Consistencia.	62
5.10.1. Consecuencias de la consistencia.	63
5.11. Resumen de la búsqueda A*.	65
5.12. Búsqueda A* paso a paso.	65
5.12.1. Búsqueda en árbol.	65
5.12.1.1. Visualización en árbol de búsqueda.	65
5.12.1.2. Visualización de la frontera (cola con prioridades).	67
5.12.2. Búsqueda en grafo.	67
6. Búsqueda con adversarios.	71
6.1. Búsqueda con adversarios y juegos.	71
6.2. Juegos con dos agentes.	72
6.3. Tipos de juegos.	73
6.4. Decisiones óptimas en juegos. Búsqueda con adversarios.	74
6.5. Algoritmo MINIMAX.	77
6.5.1. Eficiencia del MINIMAX.	82
6.6. Decisiones imperfectas. Funciones de evaluación.	83
6.7. Poda $\alpha - \beta$	85
6.7.1. Características de la poda $\alpha - \beta$	90
6.8. La poda $\alpha - \beta$ paso a paso.	91

TEMA 1

INTRODUCCIÓN A LA INTELIGENCIA ARTIFICIAL

1.1 Definición.

Existen múltiples enfoques de la IA, cada uno de los cuales destaca ciertas características de esta disciplina de nueva creación. Nosotros definiremos la Inteligencia Artificial como *la ciencia de hacer máquinas que actúan racionalmente*. Usamos los términos racional/racionalidad de un modo técnicamente específico:

- *Racional* es todo agente que busca alcanzar unos objetivos predefinidos, cualesquiera que estos sean, de manera tal que maximiza (o minimiza) algún valor de una función (*utilidad*¹).
- *La racionalidad* solo se refiere a *las decisiones* que se toman y no a los procesos que se encuentran detrás de estas. Por tanto, se refiere a los algoritmos que se utilizan pero no a los programas o máquinas que los ejecutan.
- Los *objetivos* se expresan en función de la *utilidad* de sus resultados.
- Ser *racional* significa maximizar *la utilidad esperada*.

Se ha propuesto como nombre alternativo de la disciplina y más ajustado a la realidad el de *Racionalidad Computacional*.

Una frase que resume el significado de la AI es la que muestra el robot:

Maximiza tu utilidad esperada

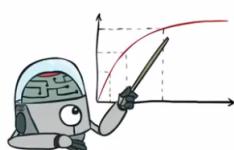


Figura 1.1.1:

¹La utilidad es un concepto extraído de la teoría microeconómica y mide el grado de satisfacción del agente. Con frecuencia asociaremos la utilidad con los costes, de modo que la utilidad del agente se maximiza si se minimizan los costes.

Conviene resaltar que hablamos de utilidad *esperada* en el sentido probabilístico del término puesto que cuando se toma una decisión solo se conoce la historia anterior al momento de la decisión y se ignora cuáles serán sus consecuencias reales. La racionalidad maximiza el resultado *esperado*, para maximizar el resultado *real* es necesaria la omnisciencia.

1.2 Breve historia de la AI.

El comienzo de la disciplina se remonta a los años 40 del pasado siglo y desde entonces podemos destacar los hechos siguientes:

- 1940-50: los inicios.
 - 1943: McCulloch & Pitts: Modelo de circuito booleano del cerebro².
 - 1950: Turing: «Computing Machinery and Intelligence³»
- Siendo de destacar el artículo de Turing en cuanto presenta por primera vez la llamada «prueba de Turing» diseñada para proporcionar una definición operacional y satisfactoria de inteligencia.
- 1950-70: el entusiasmo: ¡Mira mamá, sin manos!
 - 1950s: primeros programas de IA, incluyendo el del juego de damas de Samuel. Newell & Simon crearon el programa *Logic Theorist* capaz de demostrar muchos teoremas matemáticos.
 - 1957:
Reunión de Dartmouth donde se adoptó el nombre de “Inteligencia Artificial” .
Robinson creó con Allen Newell el *General Problem Solver* (GPS) .
- 1970-90: Enfoques basados en el conocimiento que condujeron a la proliferación de *sistemas expertos*, es decir, de conjuntos de programas que, sobre una base de conocimientos, posee información de uno o más expertos en un área específica. Se puede entender como una rama de la inteligencia artificial, donde el poder de resolución de un problema en un programa de computadora viene del conocimiento de un dominio específico:
 - *Dendral* que interpreta la estructura molecular.
 - *Dipmeter Advisor* (Asesor) fue un sistema experto temprano desarrollado en 1980 por Schlumberger Doll Research para auxiliar en el análisis de los datos recolectados durante la exploración petrolera.
 - *Caduceus* fue un sistema experto médico programado para realizar diagnósticos en medicina interna. Ha sido descrito como el sistema experto "de mayor conocimiento intensivo existente"
- 1990-: enfoque estadístico y resurgimiento de la probabilidad.

²<http://www.minicomplexity.org/pubs/1943-mcculloch-pitts-bmb.pdf>

³<http://www.csee.umbc.edu/courses/471/papers/turing.pdf>

- Aumento en la profundidad técnica.
- Agentes y sistemas de aprendizaje.
- Aplicaciones masivas iniciadas por Internet que se extienden a otros campos.

1.3 Aplicaciones de la AI.

Las principales aplicaciones de la AI fuera del mundo informático donde se han aplicado a sistemas de tiempo compartido, GUI, ... han sido y son las siguientes:

- Juegos.
 - Ajedrez.
 - Damas.
- Finanzas:
 - Los bancos usan inteligencia artificial para organizar operaciones, invertir en acciones y administrar propiedades. En agosto del 2001, robots vencen a los humanos en una competición simulada de comercio financiero.
 - Las instituciones financieras usan sistemas de redes neuronales artificiales para detectar pagos o reclamaciones fuera de lo normal, marcándolos para ser investigados por humanos.
- Medicina:
 - Una clínica puede usar inteligencia artificial para organizar las asignaciones de las camas, crear una rotación del personal, proveer información médica y otras tareas importantes.
 - Sistemas de apoyo para decisiones clínicas en el diagnóstico médico, tales como la tecnología de Procesamiento de Conceptos en el software de registros médicos electrónicos.
 - Interpretación de radiologías asistidas por computadoras. Estos sistemas ayudan a escanear imágenes digitales para señalar zonas visibles, tales como posibles enfermedades. Una aplicación típica es la detección de un tumor.
 - Análisis del ruido cardíaco.
- Industria:
 - Los *robots* se han vuelto comunes en muchas industrias. A menudo se le asignan puestos de trabajo que se consideran peligrosos para los humanos.
- Servicios de atención al cliente:
 - Un asistente en línea automatizado proporcionando servicio de atención al cliente en una página web.

- Transportes:
 - aplicaciones de conducción automática en automóviles y aviones.
 - controles para cajas de cambios automáticas en vehículos.
- Procesamiento del lenguaje:
 - Traducción.
 - Búsqueda en la Web.
 - Clasificación de textos.
- Lenguaje natural.
 - Reconocimiento de voz.
 - Texto-a-Habla
- Reconocimiento de imágenes como la Cloud Vision API de Google.
 - Reconocimiento de formas.
 - Reconocimiento de caras.
- World Wide Web: Internet es una red global por la que circula un volumen enorme de datos de todo tipo: de personas, transacciones, sensores,...a través de la cual cada día se realizan más labores de procesamiento y almacenamiento en sustitución de los ordenadores locales. Esto ha motivado el interés de muchas organizaciones públicas y privadas en desplegar herramientas basadas en el Big Data y la AI para explotar esta inmensa fuente de recursos de información.
- Toma de decisiones:
 - Spam de correos-e.
 - Recomendación de productos (Amazon).
- Lógica:
 - Demostración de teoremas.
 - Respuesta a preguntas.

1.4 Sistemas y agentes inteligentes.

El elemento clave de un sistema inteligente es el *agente racional*.

Un agente es cualquier cosa capaz de percibir su medio ambiente con la ayuda de sensores y actuar sobre ese medio utilizando actuadores. La figura 1.4.1 muestra el esquema de un agente donde el recuadro con el signo de interrogación es la *función del agente*, esto es, en términos matemáticos se puede decir que el comportamiento del agente viene dado por la *función del agente* que proyecta una percepción dada en una acción.

Un *agente humano* tiene ojos, oídos y otros sensores además de manos, piernas, boca así como otras partes del cuerpo y herramientas como actuadores.

Un *agente robot* recibe pulsaciones del teclado, archivos de información y paquetes vía red a modo de entradas sensoriales y actúa sobre el medio con mensajes en el monitor, escribiendo ficheros y enviando paquetes por la red.

Se admite la hipótesis general de que *cada agente puede percibir sus propias acciones*.

La *secuencia de percepciones* de un agente refleja el historial completo de lo que el agente ha recibido. En general, un agente tomará una decisión en un momento dado dependiendo de su secuencia completa de percepciones hasta ese instante. Si se puede especificar qué decisión tomará un agente para cada una de las posibles secuencias de percepciones, entonces se habrá explicado más o menos todo lo que se puede decir de un agente. En términos matemáticos se puede decir que el comportamiento del agente viene dado por la *función del agente* que proyecta una percepción dada en una acción.

En la figura podemos ver un ejemplo físico y su relación con el modelo de agente.

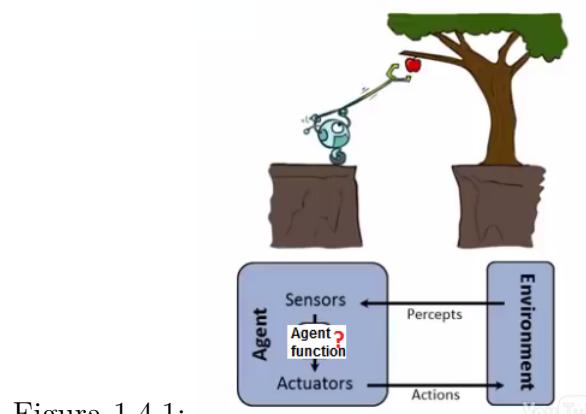


Figura 1.4.1:

Pac-Man puede utilizarse como un ejemplo de agente.

El protagonista del videojuego Pac-Man es un círculo amarillo al que le falta un sector. Aparece en laberintos donde debe comer puntos pequeños y puntos mayores. El *objetivo* del personaje es

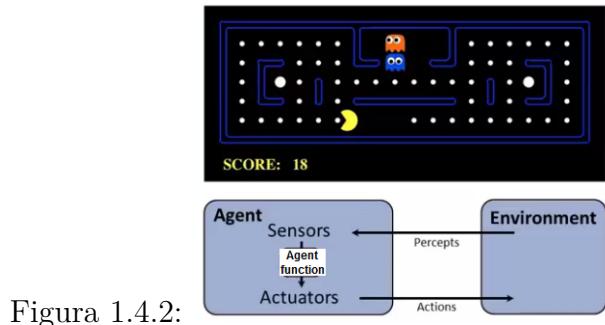


Figura 1.4.2:

comer todos los puntos de la pantalla, momento en el que se pasa al siguiente nivel o pantalla. Sin embargo, unos fantasmas o monstruos recorren el laberinto para intentar capturar a Pac-Man. Hay puntos más grandes de lo normal que proporcionan a Pac-Man la habilidad temporal de comerse a los monstruos (todos ellos se vuelven azules mientras Pac-Man tiene esa habilidad). Después de haber sido tragados, los fantasmas se regeneran en «casa» (una caja situada en el centro del laberinto). El tiempo en que los monstruos permanecen vulnerables varía según la pantalla.

En el juego el agente debe determinar en cada posición cuál debe ser su próxima acción para lo cual debe tomar en cuenta factores adicionales, tales como:

- posiciones de los puntos restantes (pequeños y mayores)
- posiciones de los fantasmas.

Podemos distinguir entre dos tipos fundamentales de agentes:

1. Agentes reactivos.
2. Agentes planificadores.

Como ejemplo consideraremos un taxi automático.

1.4.1 Agentes reactivos.

Este tipo de agentes se caracteriza porque

- Elige sus acciones basándose en las percepciones actuales (y quizá en la memoria)
- Puede tener memoria o un modelo del estado actual del mundo.
- No considera las futuras consecuencias de sus acciones.
- Considera al mundo tal como es.

El tipo de agente más sencillo es el *agente reactivo simple* que selecciona las acciones sobre la base de las percepciones actuales, ignorando el resto de las percepciones históricas cuyo esquema de funcionamiento es el siguiente:

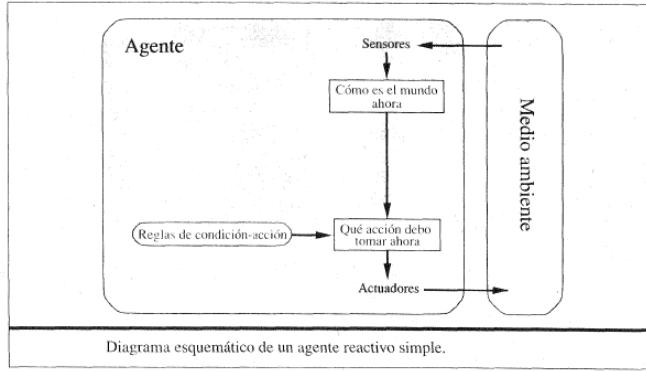


Figura 1.4.3:

función AGENTE-DIRIGIDO-MEDIANTE TABLA(*percepción*) **devuelve** una acción
variables estáticas: *percepciones*, una secuencia, vacía inicialmente
tabla, una tabla de acciones, indexada por las secuencias de percepciones, totalmente definida inicialmente

añadir la *percepción* al final de las *percepciones*
acción \leftarrow CONSULTA(*percepciones*, *tabla*)
devolver *acción*

El programa AGENTE-DIRIGIDO-MEDIANTE TABLA se invoca con cada nueva percepción y devuelve una acción en cada momento. Almacena la secuencia de percepciones utilizando su propia estructura de datos privada.

Figura 1.4.4:

P.e. el taxi automático tiene definida una tabla de *condición-regla de actuación*, esto es **if condición then acción** como p.e. **if semáforo en rojo then detenerse**.

Existen además los *agentes reactivos basados en modelos*. Un agente difícilmente puede ver la totalidad del «mundo» que lo rodea. La forma más efectiva que tienen los agentes de manejar la visibilidad parcial es almacenar información de las partes del mundo que no pueden ver. O lo que es lo mismo, el agente debe mantener algún tipo de *estado interno* que dependa de la historia percibida y que de ese modo refleje por lo menos alguno de los aspectos no observables del estado actual.

La actualización de la información del citado estado interno según pasa el tiempo requiere codificar dos tipos de conocimiento en el programa del agente:

- Primero, se necesita alguna información acerca de cómo evoluciona el mundo independientemente del agente.
- Segundo, se necesita más información sobre cómo afectan al mundo las acciones del agente.

Este conocimiento acerca de cómo funciona el «mundo», se denomina modelo del mundo. Esquemáticamente:

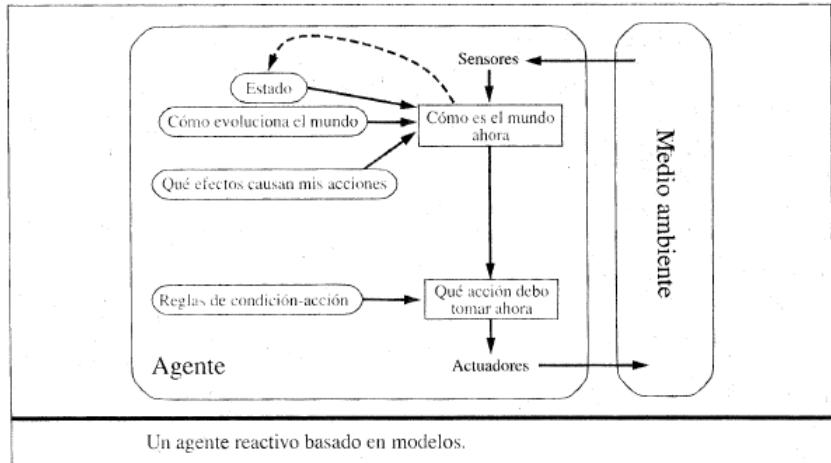


Figura 1.4.5:

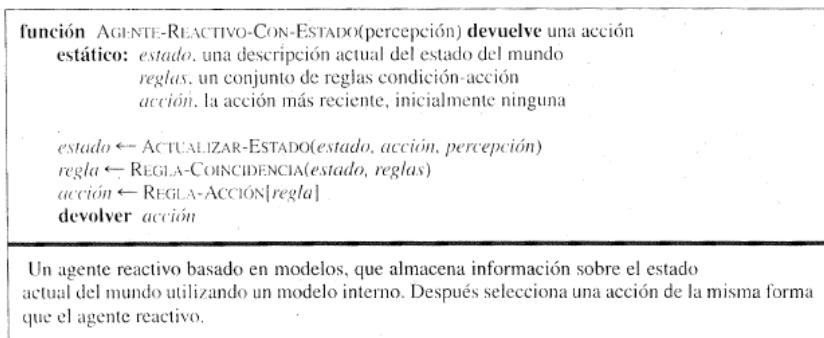


Figura 1.4.6:

Como ejemplo, el taxi es capaz de saber que, por lo general, tras 5 minutos en una autopista habrá recorrido 15 km o que la distancia de frenado para no chocar cuando se viaja a 40 km/h es de 16 m.

La parte interesante de este agente es la correspondiente a la función ACTUALIZAR-ESTADO que es la responsable de la creación de la nueva descripción del estado interno. Además de interpretar la nueva percepción a partir del conocimiento existente sobre el estado, utiliza la información relativa a la forma en la que evoluciona el «mundo» para conoce más sobre aquellas partes del «mundo» que no están visibles: para lo cual debe conocer cuál es el efecto de las acciones del agente sobre el estado del mundo.

De un modo natural surge la pregunta: *¿Puede un agente reactivo ser racional?*

En principio parece que esto es difícil salvo en casos simples y aislados. El conocimiento sobre el estado actual del mundo parece casi nunca suficiente para decidir qué hacer. Por ejemplo, en un cruce de carreteras, un taxi automático puede girar a la izquierda, a la derecha o seguir hacia delante. La decisión correcta depende del destino que quiera alcanzar el taxi. En otras palabras, además de la descripción del estado actual, el agente necesita sitar algún tipo de información sobre su meta que describa las situaciones que son deseables, por ejemplo, llegar al destino propuesto por el pasajero.

Pero dejaremos esta discusión para el capítulo siguiente.



Figura 1.4.7:

1.4.2 Agentes planificadores.

Un agente planificador es aquel que anticipa las consecuencias futuras de sus acciones y usa estas consecuencias como guía para decidir sus acciones actuales.

Esta idea de la *planificación* puede formalizarse como un **problema de búsqueda** que será lo que nos ocupe en el siguiente capítulo.

Parte I

Búsqueda

TEMA 2

RESOLUCIÓN DE PROBLEMAS MEDIANTE BÚSQUEDA

2.1 Introducción.

La idea tras el concepto de búsqueda es hacernos capaces de construir *buenos* agentes capaces de tomar *buenas* decisiones. Para ello el agente ha de ser generalmente un agente planificador capaz de *prever* el futuro de alguna forma, esto es, capaz de evaluar las consecuencias de sus acciones en diferentes entornos.

Esta idea de la planificación futura se plantea como una **búsqueda** y comenzaremos estudiando la **búsqueda no informada** (o **a ciegas**) en tres modalidades:

1. búsqueda primero en anchura;
2. búsqueda primero en profundidad y
3. búsqueda de coste uniforme.

«*Búsqueda a ciegas*» significa que en esta no se tiene información adicional acerca de los estados más allá de la que proporciona la definición del problema. Todo lo que esta búsqueda puede hacer es

- generar los sucesores y
- distinguir entre un estado objetivo de uno que no lo es.

Las consideraremos dentro de un mismo contexto y los compararemos más adelante con otros procedimientos *informados* guiados por una *heurística* en los que, por el contrario, la búsqueda es capaz de determinar si un estado es más *prometedor* que otro.

2.2 Agentes que *piensan* frente a agentes *reactivos*.

¿Por qué queremos agentes que piensan?

Fíjémonos en el robot que quiere coger una manzana del árbol. Si el código que guía su comportamiento solo le dice **if detectas manzana then acércate a manzana** o a Pac-Man diciéndole simplemente en su tabla:

- **if detectas punto then cómete el punto**
- **if ves fantasma then huye del fantasma**

Es fácil comprobar que estos simples códigos conducen a malas decisiones, como se ve en el caso de que exista un vacío entre el robot y el manzano.

Podríamos escribir un código más complejo, indicando p.e. lo que habría que hacer en caso de existir un abismo entre agente y árbol, del tipo **if** *abismo* **then** *salta hacia el árbol*.

Mejor sería que el agente supiera que es un mal plan y considerase otras alternativas para alcanzar su objetivo en uno o varios pasos.

Nuestro objetivo es codificar el comportamiento de estos agentes que *planifican por adelantado* y son capaces de emprender las acciones *contextualmente adecuadas* y donde no es necesario escribir las complicada relaciones contexto-acción, sino que estas surgen de nuestro modelo.

Las acciones del agente reactivo, por tanto, se describen básicamente mediante sentencias **if**-**then** es decir, sus acciones se toman en base a su percepción actual (ves la manzana, el punto, el fantasma) y en una tabla encuentra la acción pertinente (acercarse al árbol, comer puntos, huir del fantasma). Se trata de un mapeado directo y simple entre entradas (*percepciones*) y salidas (*acciones*), pero la tabla puede llegar a ser extraordinariamente grande. Pero, además, los **agentes reactivos** *no toman en consideración las consecuencias futuras de sus acciones*, sino solo *el estado actual del «mundo»*.

Si nos fijamos en Pac-Man con una sola regla: **if** *ves punto* **then** *cómete el punto*, en los dos casos siguientes:

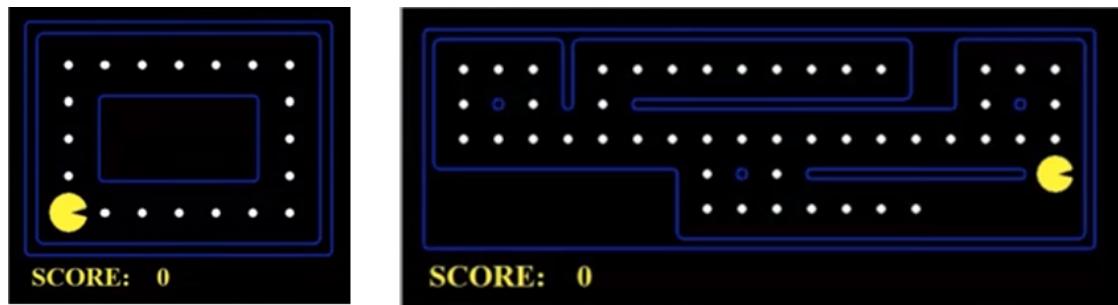


Figura 2.2.1:

el de la izquierda podría ser resuelto de una manera racional, pero no así el derecho dado que en esta circunstancia no hay reacción prevista ya que para comerse el punto que se encuentra en sus fauces, precisa de un punto a la vista que no existe.

Pac-Man se detendría, al no tener prevista la situación en que no haya punto a la vista aunque lo hay más allá de su cercanía inmediata.

Volviendo a la pregunta acerca de la racionalidad de los agentes reactivos, *la racionalidad es una función de las acciones* que el agente emprende no de la computación que fue necesaria para ello. Por tanto, un agente reactivo con una buena tabla de percepciones-acciones de tamaño suficiente puede tomar las decisiones adecuadas mientras que puede haber agentes planificadores que fracasen en su intento de ser racionales dependiendo de su funcionamiento.

Un *agente planificador*, al contrario que el reactivo, piensa en las consecuencias de sus acciones: se pregunta **what if?** Una de las formas de considerar estas consecuencias de las propias acciones es ejecutarlas¹, aun cuando lo más común es *simularlas en un modelo*. Por tanto, *un agente planificador* requiere

¹Modalidad que se trata en el llamado *aprendizaje por refuerzo*.

1. un *modelo del mundo* y
2. un *objetivo a alcanzar* - manzana - en dicho modelo o más generalmente un *test objetivo* ya que el robot se conforma con una cualquiera de las manzanas del árbol.

Algunas ideas de importancia a este respecto son:

- *Planeamiento completo y óptimo.*
 - Planeamiento completo: se refiere a la situación en que el agente encuentra una solución para *alcanzar su objetivo*.
 - Planeamiento óptimo: se da cuando no solo encuentra una solución sino que esta es, además, *óptima con respecto a una función de coste*.
- *Planeamiento completo y replanificación.*
 - El agente puede tener un *plan completo* que ejecuta para alcanzar su objetivo o
 - tener una *serie de planes* que ejecuta en secuencia uno tras otro, *replanificando*. Un ejemplo de este sería el caso 2 de Pac-Man si este dispusiera de un plan a realizar cuando no encuentra ningún punto a su vista («Plan B»).

2.3 Grafos de espacio de estados.

Sea un mundo en el que existen tres bloques de juguete: A,B,C y un robot que puede mover aquellos que tengan su cara superior libre (sobre el suelo o en la cima de una pila). Esas *acciones* las modelamos mediante el *esquema move*(x, y) - $x \rightarrow y$ - donde $x \in \{A, B, C\}$ e $y \in \{A, B, C, Floor\}$. Las instancias del esquema como *move*(A, *Floor*) se llaman **operadores**, es decir, son *modelos de acciones*. Sabemos que algunos operadores son inviables (*move*(A,A)).

Los operadores posibles se representan gráficamente en la figura siguiente:

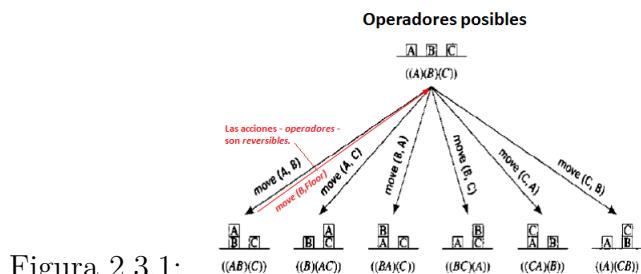


Figura 2.3.1:

A partir del estado del mundo producido por cada operador, podemos seguir un paso más allá p.e. de $((AB)(C)) \rightarrow ((A)(B)(C))$, $((ABC))$, $((A(BC))$. Una estructura útil para este propósito, es decir, para seguir las consecuencias alternativas de acciones es un **grafo dirigido** (digrafo), donde

- sus *nodos* son representaciones del *estado del mundo* del agente y
- sus *arcos* son *operadores*.

P.e. en el ejemplo de los bloques es posible hacer explícita la totalidad del árbol que, además por su reducida dimensión, puede ser guardada en la memoria de un ordenador, como se ve en la figura:

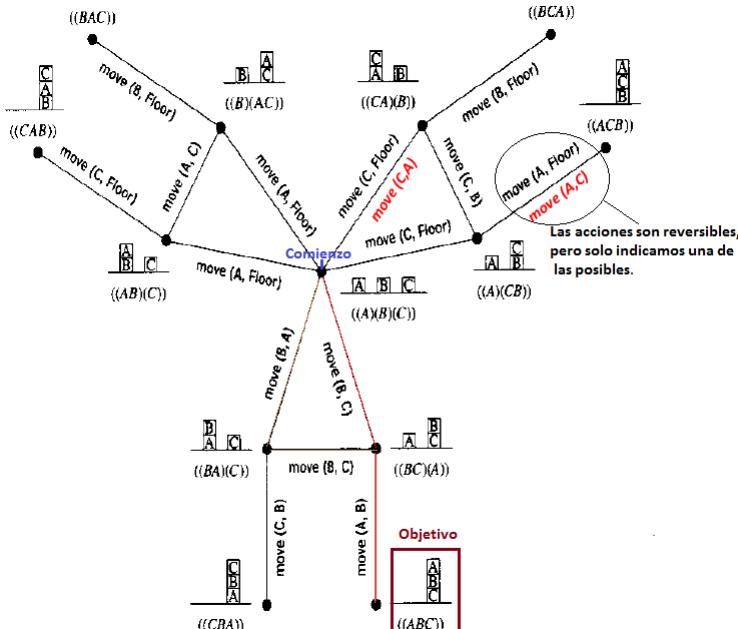


Figura 2.3.2:

El agente que desea alcanzar un *objetivo* como $((ABC))$ necesita simplemente encontrar un camino desde el nodo de comienzo hasta el nodo objetivo. Los operadores necesarios para ello - que constituyen un *plan* - están etiquetadas en los arcos del grafo.

Buscar tal secuencia se denomina *planificación* y predecir una serie de estados del mundo resultante de una secuencia de acciones se llama *predicción*.

En casos como este en que los grafos son explícitos, los métodos de búsqueda implican la propagación de «marcadores» sobre los nodos del grafo. Comenzamos etiquetando el nodo de comienzo con un 0 y propagamos en oleadas sucesivas enteros mayores a lo largo de los arcos hasta alcanzar el nodo objetivo. Luego retrocedemos desde el objetivo hasta el comienzo según enteros decrecientes y las acciones para conseguir el objetivo - plan - serán los arcos. P.e. para el problema de los bloques las etapas de propagación de marcadores serían las siguientes²:

²Como veremos luego, se trata de una búsqueda primero en anchura.

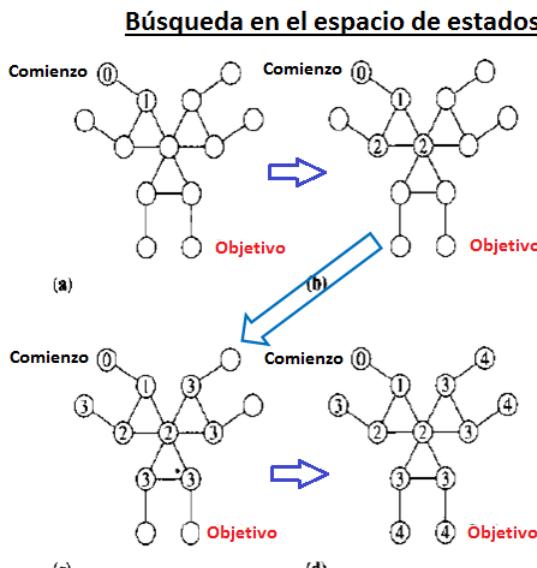


Figura 2.3.3:

2.4 Terminología de grafos.

Grafo: conjunto de *nodos* - o vértices -(no necesariamente finito).

Arco: par *ordenado* de vértices. Se representa mediante una línea terminada en flecha que va del primero al segundo.

Un grafo con arcos dirigidos se llama **grafo dirigido** o digrafo.

Para nuestros propósitos *los nodos* representan estados del mundo y los *arcos* se etiquetan con nombres de acciones.

En un arco (a, b) a se denomina *padre* de b y b es un *sucesor* (o hijo) de a .

Un par de nodos pueden ser ambos sucesores uno del otro, en cuyo caso el par de arcos se reemplaza por una **arista** sin flechas.

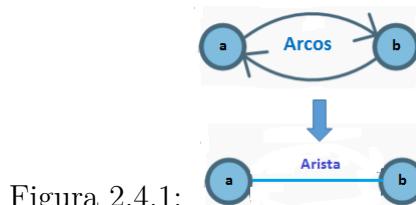


Figura 2.4.1:

Los grafos que solo contienen aristas se llaman **grafos no dirigidos**.

Un **árbol dirigido** es un caso especial de *grafo dirigido* en el que *cada nodo* (excepto el raíz) *tiene exactamente un parente*.

- El nodo sin parente se llama **raíz**.

- Los nodos sin hijos se llaman **hojas**³.
- Si no hay arcos sino aristas, tenemos un **árbol no dirigido**.
- La **profundidad** de un nodo es la de su padre +1, siendo 0 la del nodo raíz.
- Ciertos árboles son tales que todos los nodos excepto los hojas tienen el mismo número de sucesores **b** que se denomina **factor de ramificación**.
- Una secuencia de nodos $(n_1, \dots, n_i, \dots, n_l)$ en que cada n_i es sucesor de n_{i-1} se llama **camino** de longitud l desde el nodo n_1 al n_l .
Dos nodos (n_j, n_{j+m}) situados en un mismo camino son tales que el primero es un *ancestro* del segundo que, a su vez es un *descendiente* del primero.
- *Frontera*: Conjunto formado por los nodos hoja.

Coste(-) - o utilidad(+)- : con frecuencia cada acción - arco (a, b) -tiene un coste (o utilidad): $c(a, b)$.

Camino óptimo entre dos nodos: el que tiene menor coste (o mayor utilidad).

Terminología de grafos

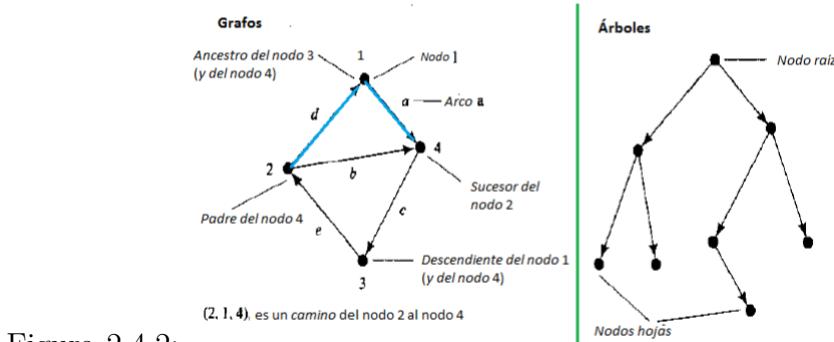


Figura 2.4.2:

2.5 Problema de búsqueda.

Muchos problemas prácticos tienen espacios de búsqueda tan grandes que es imposible representarlos mediante grafos explícitos, de modo que hemos de encontrar alguna forma de representar grafos de gran tamaño implícitamente.

Los componentes básicos de un grafo con espacio de estados implícito implica:

1. Una descripción del estado en el nodo de comienzo, plasmada en una estructura de datos.
2. Funciones que transforman la descripción de un estado en otra que describe el estado alcanzado tras una acción. Estas funciones son llamadas *operadores* y modelizan los efectos de las acciones. Cuando aplicamos un operador a un nodo generamos uno de sus nodos sucesores.

³Como veremos en árboles de búsqueda son también los últimos nodos generados que no se ha expandido aún y forman parte de la *frontera*

3. Una condición de *objetivo* que puede ser una función booleana de descripciones de estado o una lista de instancias de descripciones de estado que corresponden a estados objetivo.

¿Qué es un problema de búsqueda?

- Un *problema de búsqueda* implica tres fases:
- Formulación,
- Búsqueda y
- Ejecución.

La **formulación** del problema consiste en:

1. Definir un ***espacio de estados*** que codifica esencialmente el estado del mundo en un cierto momento de la progresión del plan del agente. Es una abstracción del mundo;
2. un ***estado inicial*** desde el que partimos;
3. Especificar las ***acciones*** que puede realizar el agente:
 - a) Reglas para las acciones permitidas que son descritas por *operadores*.
 - b) Dado que desde un nodo pueden alcanzarse en general varios nodos hijos y las acciones tienen un coste, puede definirse una ***función sucesor*** con acciones y coste que modela cómo piensa el planificador que ***funciona el mundo***. La función sucesor se encuentra en un estado del espacio de estados y desde allí nos dice
 - 1) donde debemos ir inmediatamente,
 - 2) qué acciones realizar a tal fin y, en su caso,
 - 3) cuáles son sus costes.

Por tanto, dado un estado **x**, y una acción **a**, **FUNCIÓN – SUCESOR(x)** devuelve un **conjunto de ternas** < acción, coste, sucesor > .

Mientras el espacio de estados modela cómo es el mundo, la función sucesor modeliza cómo evoluciona en respuesta a nuestras acciones;

4. definir la *utilidad* o el *coste* como función del camino y
5. un *test objetivo* que nos dice si hemos alcanzado o no nuestro objetivo.

Una *solución* es una secuencia de acciones⁴ (plan) que transforma el estado inicial en el estado objetivo (i.e. que satisface el test objetivo). Una solución es óptima si tiene utilidad máxima.

Un ejemplo tomado de Norvig es el siguiente:

Estamos viajando por Rumanía y nos encontramos en la ciudad de Arad desde donde hemos de ir a Bucarest. Disponemos de un mapa que constituye un buen ejemplo de un modelo (abstracción) de cómo es el mundo.

⁴Un *camino* en el grafo de estados.

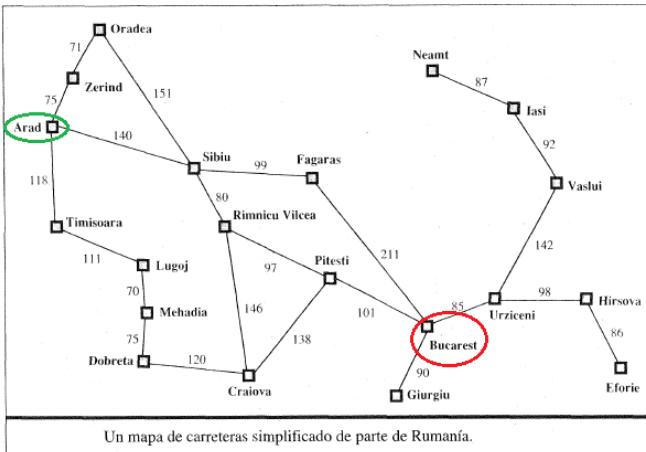


Figura 2.5.1:

Formulemos el problema de búsqueda correspondiente:

1. *Espacio de estados:* Como mínimo, ha de incluir las *ciudades* de modo que podamos aplicar el test objetivo (¿estamos en Bucarest?).

- *estado*=ciudad

2. *Acciones:* tomar carreteras en una dirección.

3. *Función sucesor(state):* lo que podemos hacer es únicamente ir de una ciudad a otra por carretera así que la función sucesor es:

- carretera par ir a una ciudad adyacente con coste=distancia. Por tanto, devuelve $\langle \text{acción} = \text{carretera}, \text{coste} = \text{distancia}, \text{sucesor} = \text{ciudad} \rangle$

4. *Estado inicial:*

- $\text{estado}_0 = \text{Arad}$.

5. *Test objetivo:*

- $\text{¿estado}(\text{ciudad}) == \text{Bucarest?}$

6. *Solución:*

- alguna secuencia de carreteras.

2.5.1 El espacio de estados.

Una importante distinción ha de hacerse entre *estado del mundo* y *estado de la búsqueda* o *estado del modelo*.

Mientras que el estado del mundo incluye hasta el menor detalle del entorno (modelo del coche, tiempo que hace, hora del día) el estado de la búsqueda solo conserva los detalles que son necesarios para planificar y es, por tanto una *abstracción* de la realidad cuyo nivel de detalle dependerá del problema en estudio. En nuestro problema, donde queremos encontrar un camino entre dos ciudades, nos interesan las ciudades en que nos encontramos el estado de la búsqueda puede definirse con las coordenadas (x, y) de la ubicación en que estamos:

- Estado de la búsqueda: **ubicación** (x, y)
- Acciones: **dirección** a seguir.
- Función sucesor: **actualizar ubicaciones posibles**.
- Test objetivo: $\text{¿}(x, y) == (\mathbf{x}_{\text{Bucarest}}, \mathbf{y}_{\text{Bucarest}})?$

El motivo por el que no es operativo utilizar explícitamente el estado del mundo es fácil de ver en el siguiente ejemplo de Pac-Man con dos fantasmas:



Figura 2.5.2:

Por tanto el *número de estados de búsqueda* es en este caso:

$$120 \times 4 \times 12^2 \times 2^{30} = 7,4217 \cdot 10^{17}$$

ya que

- *número de estados para formar parte de un camino:*

120

- *número de estados de los puntos* indicando si han sido comidos o no (booleano):

$$120 \cdot 2^{30} = 1,28849 \cdot 10^{11}$$

Una de las maneras de enfrentar el problema sería mediante la *búsqueda por fuerza bruta*, es decir, enumerar sistemáticamente todos los posibles candidatos para la solución de un problema, con el fin de comprobar si dicho candidato satisface el test objetivo. El espacio de estados es tan inmenso que resulta imposible enumerarlos todos y después ordenarlos en sus posibles secuencias. Por tanto lo que haremos será desarrollar *algoritmos*⁵ que eviten la necesidad de enumerar todos los estados, y que reduzcan las enumeraciones de estado al mínimo posible.

2.6 Grafos del espacio de estados y árboles de búsqueda.

Implícitamente el estado inicial y la función sucesor definen el espacio de estados del problema (el conjunto de todos los estados alcanzables desde el estado inicial). *El espacio de estados forma un grafo* en el cual *los nodos son estados* y *los arcos entre los nodos son acciones*. La función sucesor está codificada en los arcos.

- **En un grafo de búsqueda** (*estados interconectados por acciones*) **cada estado aparece solo una vez**. P.e. el rodeado por el óvalo en la Fig 2.6.1 puede alcanzarse por multitud de caminos, pero solo aparece una vez en el grafo de búsqueda.

El mapa de carreteras de Rumanía puede interpretarse como un grafo del espacio de búsqueda si vemos cada carretera como dos direcciones de viaje. Un **camino** es una serie de estados conectados por una secuencia de acciones. Lo mismo puede decirse del juego de Pac-Man.

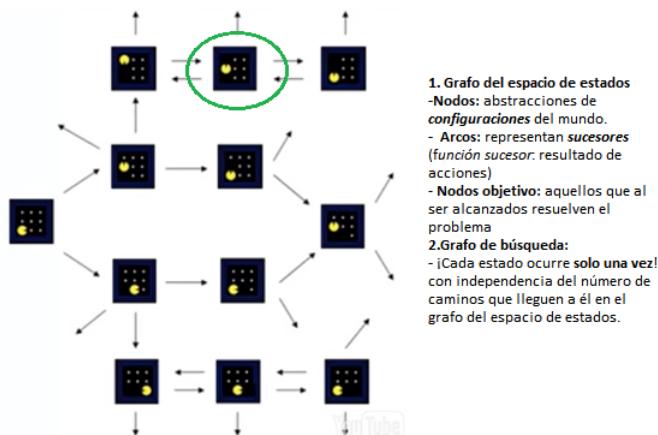


Figura 2.6.1:

Solo en muy raras ocasiones podemos construir el grafo completo en memoria (como hemos visto antes), pero es una idea útil.

Como ejemplo valga el siguiente grafo del espacio de estados:

⁵La enumeración exhaustiva es también un algoritmo pero **FBI** - de **F**(uerza) **B**(ruta) e **I**(gnorancia)

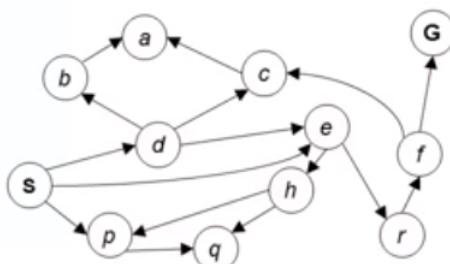


Figura 2.6.2:

Preguntémonos por lo que es relevante para nosotros en este problema: solo el *estado inicial* y *lo que pueda ocurrir a partir* de este estado, lo que nos conduce al *problema general de búsqueda en árbol* que se plantea del modo siguiente:

Búsqueda General en Árbol

```

function BÚSQUEDA-EN-ÁRBOL ( problema,estrategia ) retorna una solución o fallo
    inicializar el árbol de búsqueda con el estado inicial del problema
    loop do
        if no hay candidatos para expansión then return fallo
        Elegir nodo hoja para expansión de acuerdo con la estrategia
        if nodo satisface test-objetivo then return la solución
        else expandir el nodo y añadir los nodos resultantes al árbol
    end

```

- **Ideas Importantes :**
- Frontera
- Expansión
- Estrategia de exploración
- **Pregunta principal:** ¿Qué nodos frontera explorar?

Figura 2.6.3:

Por ello trabajaremos con un objeto denominado **árbol de búsqueda**. Para cuya construcción partimos del *estado actual (1) - ahora* - que será el *nodo raíz*. Seguidamente, en función de las acciones que se ejecuten, nos situaremos en el *futuro inmediato (2)*, *estados* que se *pueden alcanzar en solo un paso* y así continuaremos hacia abajo (3).

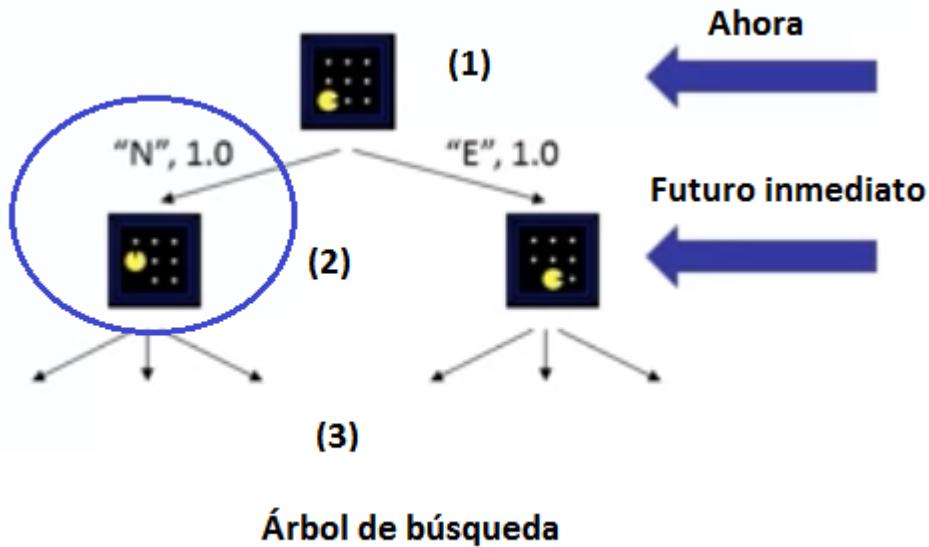


Figura 2.6.4:

Enfoquémonos en *búsqueda en un árbol* (un sólo camino desde el nodo raíz a un nodo dado)

- Los *nodos* de un árbol de búsqueda corresponden a estados de búsqueda.
- El *nodo raíz* de un árbol de búsqueda corresponde al estado de búsqueda inicial.
- *Acciones*: Expandir el nodo de búsqueda actual: Generar nodos hijo (correspondientes a nuevos estados) aplicando la función sucesor al nodo actual.
- *Estado objetivo*: Un nodo correspondiente a un estado que satisface el test de objetivo.
- *Utilidad*: - Coste del camino desde el nodo raíz al nodo actual.

Un *árbol de búsqueda* es

- Un árbol de ***what if*** y sus consecuencias.
- El *estado inicial* es el *nodo raíz*.
- los *hijos* corresponden a los *sucesores*;
- Los nodos muestran estados pero se corresponden con **planes** que alcanzan dichos estados. Cualquier plan que sea posible ejecutar se encuentra en algún lugar del árbol. P.e.en el árbol anterior, el nodo resaltado se corresponde al plan de ir en dirección Norte desde el estado inicial y llegar aquí. Hay otros modos de llegar a este estado (p.e. $N \rightarrow S \rightarrow N$), pero se hallarán en otros lugares del árbol.
- En la mayoría de los problemas es *imposible construir el árbol completo*. Lo que intentaremos es desarrollar algoritmos que nos garanticen que hemos actuado correctamente aun cuando ignoremos la mayoría del árbol completo.

Hay muchas formas de representar los nodos, pero vamos a suponer que ***un nodo es una estructura de datos*** con cinco componentes:

1. ESTADO: el estado, del espacio de estados, que se corresponde con el nodo;
2. NODO PADRE: el nodo en el árbol de búsqueda que ha generado este nodo;
3. ACCIÓN: la acción que se aplicará al padre para generar el nodo;
4. COSTE DEL CAMINO: el costo - habitualmente denotado por $g(n)$ - de un camino desde el estado inicial al nodo n y
5. PROFUNDIDAD: número de pasos a lo largo del camino desde el estado inicial.

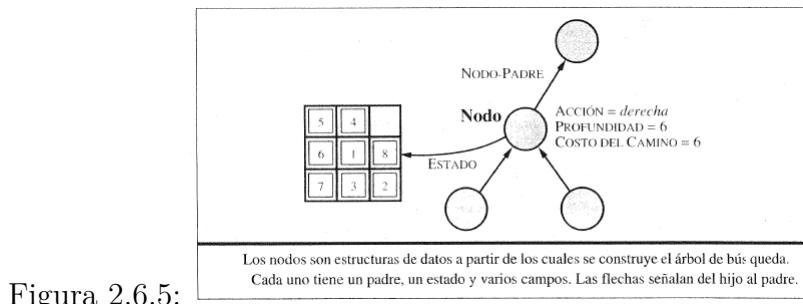


Figura 2.6.5:

Es importante recordar la distinción entre *nodos* y *estados*. Un nodo es una estructura de datos usada para representar el árbol de búsqueda (Fig. 2.4.4.). Un estado corresponde a una configuración del mundo.

2.7 Búsqueda en árbol

Partamos del problema representado por la figura 2.3.1. Comenzaremos por construir un árbol de búsqueda y analicemos algunas de sus características (Fig. 2.5.1.).



Figura 2.7.1:

Al principio, solo tendremos el *nodo inicial* o raíz (Arad). En principio, también existen todos los elementos difuminados en el gráfico que han sido definidos matemáticamente pero que no figuran todavía en nuestro código.

Seguidamente buscaremos algún procedimiento para desplazarnos hacia abajo en el árbol. A medida que avancemos iremos manteniendo una colección de planes potenciales y, por tanto, serán nodos en nuestro árbol que denominaremos **frontera**. Estos son planes que estamos considerando y que pudieran conducirnos a nuestro objetivo, pero que aún no hemos seleccionado. En nuestro avance, trataremos de **expandir** el menor número de nodos posible. En esta fase inicial, el único plan que tenemos es el vacío: estamos en Arad y no consideramos ninguna acción.

Podemos expandir el árbol incluyendo tres planes (nodos):

1. ir a Sibiu,
2. ir a Timisoara o
3. ir a Zerind.

que constituirán la nueva frontera. La frontera está constituida por un *conjunto de nodos hoja*, es decir, sin sucesores en el árbol. La estrategia de búsqueda será una función que seleccione de este conjunto el siguiente nodo a expandir ya que no podemos expandir los tres a la vez.

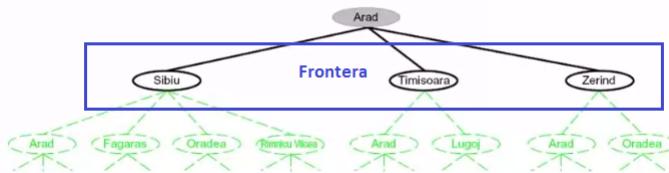


Figura 2.7.2:

Nosotros asumiremos que *esta colección de nodos se implementa como una **cola*** en la que son posibles las siguientes operaciones:

- HACER-COLA(elemento1, elemento2,...): crea una cola con los elementos dados.
- VACÍA(cola): devuelve verdadero si no hay ningún elemento en la cola.
- PRIMERO(cola): devuelve el primer elemento de la cola.
- BORRAR(primer de la cola): devuelve el primer elemento y lo borra de la cola.
- INSERTA(elemento, cola): añade elemento a la cola y devuelve la cola resultante. Puede hacerse de varias maneras como veremos (LIFO, FIFO, ...).
- INSERTAR-TODOS(elementos, cola): añade un conjunto de elementos a la cola y devuelve la cola resultante.

Ahora necesitamos elegir uno de estos tres planes y expandirlo ya que no podemos analizar los tres simultáneamente. Sea el de Sibiu el elegido.

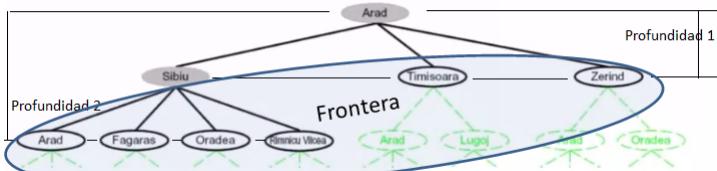


Figura 2.7.3:

Ahora la frontera está constituida por 4 nodos de profundidad 2 y 2 nodos de profundidad 1. Si en este momento descubrimos que algún (*estado==objetivo*) = *True*, hemos encontrado la solución al problema de búsqueda.

Este es el algoritmo que puede escribirse en pseudocódigo como:

```

función BÚSQUEDA-ÁRBOLES(problema,frontera) devuelve una solución o fallo
    frontera  $\leftarrow$  INSERTA(HACER-NODO(ESTADO-INICIAL[problema]),frontera)
    hacer bucle
        si VACIA?(frontera) entonces devolver fallo.
        nodo  $\leftarrow$  BORRAR-PRIMERO(frontera)
        si TEST-OBJETIVO[problema] aplicado al ESTADO[nodo] es cierto
            entonces devolver SOLUCIÓN(nodo)
        frontera  $\leftarrow$  INSERTAR-TODO(EXPANDIR(nodo,problema),frontera)
    

---


función EXPANDIR(nodo,problema) devuelve un conjunto de nodos
    sucesores  $\leftarrow$  conjunto vacío
    para cada (acción,resultado) en SUCESOR-FN[problema](ESTADO[nodo]) hacer
        s  $\leftarrow$  un nuevo Nodo
        ESTADO[s]  $\leftarrow$  resultado
        NODO-PADRE[s]  $\leftarrow$  nodo
        ACCIÓN[s]  $\leftarrow$  acción
        COSTO-CAMINO[s]  $\leftarrow$  COSTO-CAMINO[nodo] + COSTO-INDIVIDUAL(nodo,acción,s)
        PROFUNDIDAD[s]  $\leftarrow$  PROFUNDIDAD[nodo] + 1
        añadir s a sucesores
    devolver sucesores


---


Algoritmo general de búsqueda en árboles. (Notemos que el argumento frontera puede ser una cola vacía, y el tipo de cola afectará al orden de la búsqueda.) La función SOLUCIÓN devuelve la secuencia de acciones obtenida de la forma punteros al padre hasta la raíz.

```

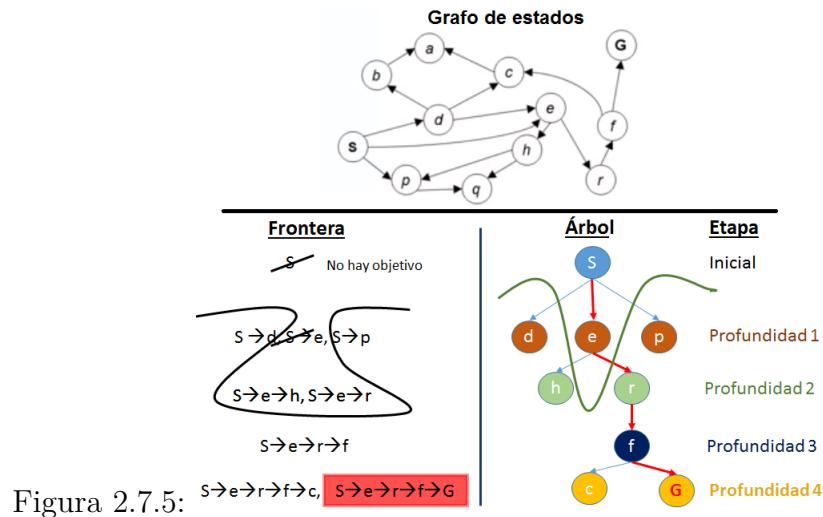
Figura 2.7.4:

En este algoritmo existen varias ideas importantes a resaltar:

- **Frontera:** conjunto de planes que aún pueden funcionar.
- **Expansión:** seleccionar nodos externos a la frontera.
- **Estrategia de exploración**

La cuestión más importante es: *¿qué nodos de la frontera explorar?*

Tomenos como ejemplo el grafo en la Fig. 2.4.2. y escribamos por fases la *frontera* y el *árbol* de búsqueda.



La figura (2.7.5) resume todo el proceso y en ella pueden seguirse las distintas etapas para llegar a la solución **G** que, como puede verse, *con la estrategia de expansión seguida, no se ha requerido la expansión completa de todo el árbol.*

de desarrollo de los nodos: en *anchura* y en *profundidad*.

TEMA 3

BÚSQUEDA PRIMERO EN PROFUNDIDAD. BÚSQUEDA PRIMERO EN ANCHURA. BÚSQUEDA CON PROFUNDIDAD ITERATIVA.

En el capítulo anterior hemos planteado una búsqueda sin más información sobre el problema que la relativa a las acciones que pueden llevarse a cabo (*legales*). este tipo de búsqueda se llama *no informada*. Además, hemos elegido nuestro desplazamiento por los nodos sin una estrategia clara. Parece conveniente explorar las posibilidades que se nos presentan de realizar nuestra búsqueda con una estrategia definida.

Imaginemos la búsqueda en un árbol como un proceso minero en que deben encontrarse ciertos minerales que están en el interior de una montaña en posiciones desconocidas. En el peor de los casos para encontrar el mineral deberíamos desmontar toda la montaña.

Resulta obvio desarrollar dos estrategias de excavación (Fig. 3.1.1.) que *parten de la cúspide de la montaña*:

1. una que profundiza verticalmente (*búsqueda en profundidad*) y
2. una segunda que va desmontando la montaña por estratos horizontales (*búsqueda en anchura*).

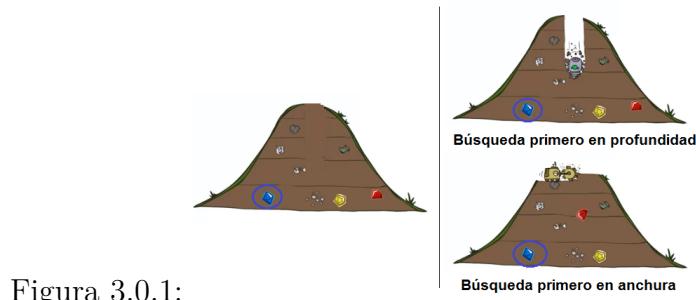


Figura 3.0.1:

3.1 Búsqueda primero en profundidad (DFS).

Centrémonos en el primer tipo de búsqueda: la *búsqueda primero en profundidad* que ilustraremos con el ejemplo expuesto en el apartado 2.5. La figura 3.1.2. muestra tanto el grafo como el árbol *completo* de búsqueda.

Este tipo de búsqueda genera los sucesores de un nodo uno por uno aplicando operadores individuales y se deja una marca en el nodo indicando que es posible que puedan aplicarse otros operadores al mismo nodo.

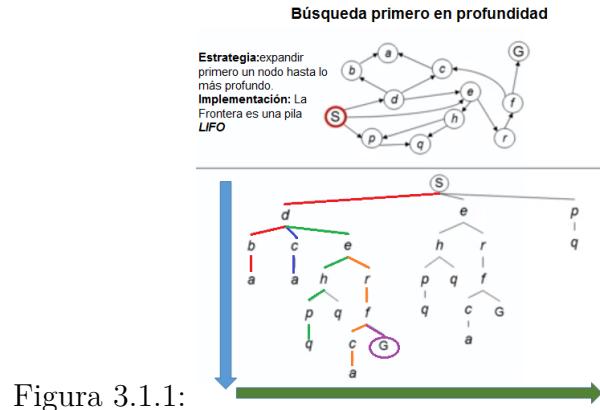


Figura 3.1.1:

Comenzando por el nodo inicial - **S** -, expandiremos en una primera etapa hasta la frontera (**d, p, e**). Hemos de profundizar inicialmente sobre uno solo de ellos, sin embargo los tres nodos son igualmente válidos para seguir profundizando. Supongamos que, *como regla general, elegimos el situado más a la izquierda en el árbol* - **d** - y sobre este expandiremos en cualquier profundidad. En la figura se dibujan los diferentes caminos explorados hasta encontrar el objetivo **G**. La estrategia es clara: hemos ido directamente hacia abajo por la izquierda del árbol hasta llegar a un nodo sin hijos y, si no es este el objetivo, volvemos al nivel inicial. El algoritmo procesa todos los nodos a la izquierda de la primera solución.

Ejemplo: En el gráfico de la Fig. 3.1.3. ¿qué solución encontrará una búsqueda primero en profundidad?

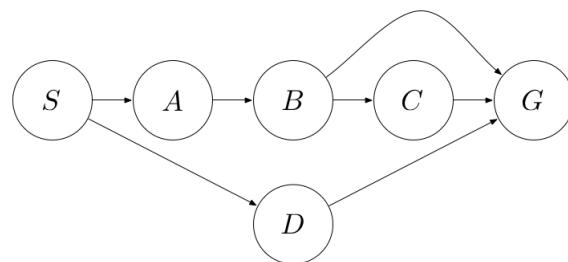


Figura 3.1.2:

SOLUCIÓN: $\circ \quad S \rightarrow A \rightarrow B \rightarrow C \rightarrow G$

3.1.1 Propiedades del algoritmo de búsqueda primero en profundidad (DFS).

1. ¿Es **completo**? es decir, si existe una solución, ¿está garantizado que la encuentre? Es complicado, pero habremos de responder a esta pregunta en cada uno de los problemas que enfrentemos.

2. ¿Es óptimo?, es decir, ¿encuentra la solución de menor coste?
3. ¿Complejidad temporal? Se refiere al número máximo de nodos que se expanden.
4. ¿Complejidad espacial? Se refiere al tamaño máximo de la frontera¹.

Trataremos de responder a algunas de estas preguntas a través del siguiente esquema:

Sea **b** el **factor de ramificación** del árbol, esto es, el *número de hijos de cualquier nodo*. Cuando - como es lo habitual en árboles no uniformes - no es el mismo en todos, se toma un valor promedio, esto es, en dichos casos el **factor de ramificación a una profundidad determinada d se calcula del siguiente modo**:

Sea T el número de nodos del árbol no uniforme,

- En un árbol uniforme el número de nodos y la profundidad del árbol - Fig. 3.1.3 - se relacionan por:

$$T = 1 + b + b^2 + \dots + b^d$$

- Haremos lo mismo con el árbol no uniforme - Fig. 3.1.4 - para *un factor promedio \bar{b}* :

$$T = 1 + \bar{b} + (\bar{b})^2 + \dots + (\bar{b})^d \quad (3.1.1)$$

Siendo **d** es el número máximo de niveles del árbol y el problema puede tener soluciones a varias profundidades.

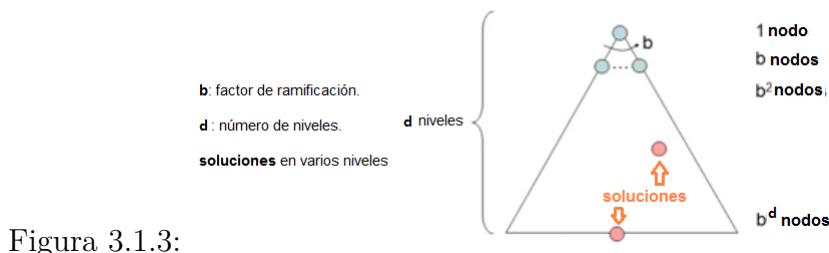


Figura 3.1.3:

Siendo esto así, el número de nodos en el árbol será:

$$1 + b + b^2 + \dots + b^d = \frac{b^{d+1} - 1}{b - 1}$$

Por tanto, es *exponencial*.

Aplicado al problema de la Fig. 3.1.2.

¹Como es sabido en Informática suele existir el dipolo tiempo de proceso- espacio de memoria. Suele ser posible cambiar uno por otro: la reducción de una de las variables aumenta la otra (reducir memoria aumenta el tiempo de procesamiento).

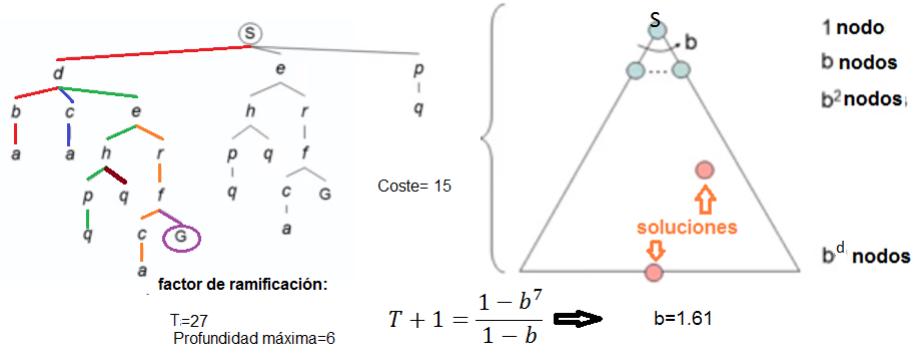


Figura 3.1.4:

3.1.1.1. Complejidad temporal (nodos a expandir).

Como hemos dicho empezaremos por aquellos que tengan un prefijo izquierdo en el árbol. Esto implica que es posible que tengamos que procesar la totalidad del árbol lo que no es nada deseable dado el carácter exponencial del problema lo que implica *un tiempo de proceso O(b^d)*.

3.1.1.2. Complejidad espacial (tamaño de la frontera).

Nos interesa saber el espacio que ocupa la frontera. Recordemos que la frontera contiene *todas las cosas que aún no hemos hecho* en el nivel en que estamos.

Puede haber **b sucesores en cada nivel** y como mucho **d niveles** y como solo hay hermanos en el camino hacia el nodo raíz, será **O(b · d)**. Por tanto, la frontera es pequeña.

3.1.1.3. Completitud.

La dificultad aquí estriba en el valor de **d**. Si **d** es infinito, el algoritmo no es completo, por tanto debemos evitar la existencia de *ciclos*² de donde nunca saldríamos. No obstante, existen procedimientos generales para evitar la redundancia y podemos descartar esta situación. En estas circunstancias - eliminando los ciclos- **es completo**.

3.1.1.4. Optimalidad.

Supongamos que el coste de cada acción es 1. El **coste mínimo equivale al de la solución más superficial** (menor profundidad desde el nodo raíz). Tal como lo hemos presentado, **no** garantiza el coste mínimo, sino que **encuentra la solución más a la izquierda con independencia de su profundidad o coste**. **No** es por tanto óptima.

²Camino donde no se repite ninguna arista ni vértice y coinciden los vértices inicial y final

3.2 Búsqueda primero en anchura.

En el capítulo anterior hemos realizado una estrategia de búsqueda que generaba un grafo de estados explícito aplicando todos los operadores posibles al nodo raíz después a los nodos sucesores de este y así sucesivamente. La búsqueda avanza uniformemente desde el nodo de comienzo. Este tipo de búsqueda se denomina *Búsqueda Primero en Profundidad (DFS)*.

Cuando se aplica la función sucesor a un nodo, se generan el conjunto completo de nodos que puede producir la aplicación de todos los operadores posibles a dicho nodo de partida.

Ahora *avanzaremos a lo ancho* del árbol, bien de derecha a izquierda, bien en sentido contrario. En este tipo de búsqueda el gráfico que muestra la estrategia de búsqueda - consistente en *expandir el nodo más superficial primero* - es el siguiente:

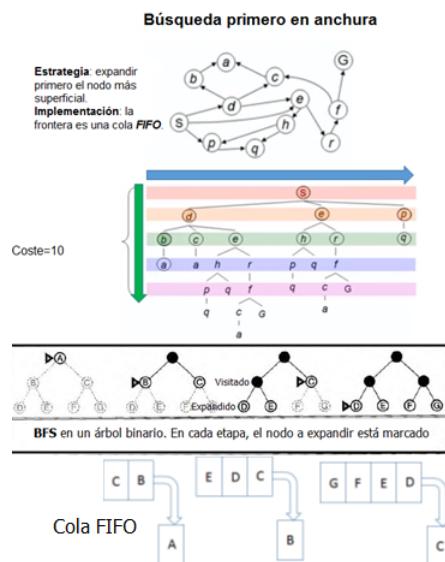


Figura 3.2.1:

En esta ocasión, se procede por capas: primero recorremos el primer nivel completamente y solo entonces comenzaremos con el segundo y así sucesivamente. Los nodos se recorren por niveles. En este algoritmo la frontera es una cola FIFO (primero en entrar, primero en salir) en la que se insertan los nodos al final sin importar el orden de inserción (pues todos los nodos pertenecen al mismo nivel).

Ejemplo. ¿Qué solución daría el ejemplo de la Fig. 3.1.3. cuando se busca primero en profundidad?

La respuesta es ahora :

• $S \rightarrow D \rightarrow G$

Veamos sobre un ejemplo cómo funcionaría el algoritmo **BFS** y su cola asociada:

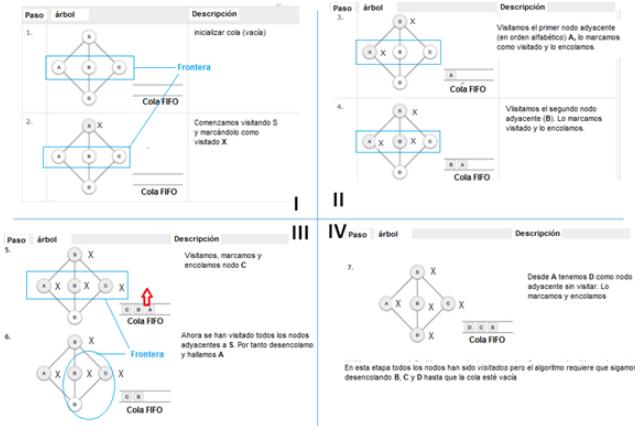


Figura 3.2.2:

3.2.1 Propiedades del algoritmo de búsqueda primero en anchura (BFS).

Es claro que el algoritmo BFS procesa todos los nodos por encima de la solución menos profunda. De nuevo nos encontramos con un caso exponencial como en DFS con la diferencia de que no tenemos que llegar a la máxima profundidad m , sino que esto dependerá de la profundidad a la que se encuentre la solución que llamaremos s .

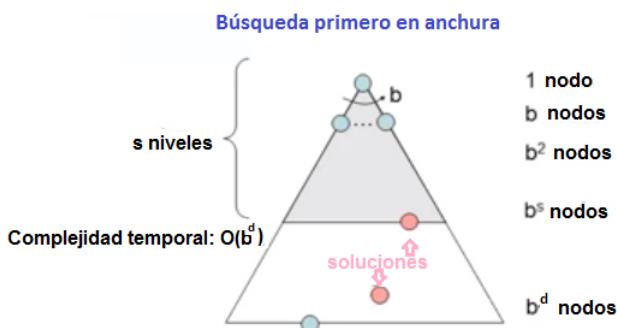


Figura 3.2.3:

3.2.1.1. Complejidad temporal.

Siendo b el factor de ramificación:

Por tanto ahora la complejidad temporal será $O(b^s)$ dependiendo de la profundidad $-s-$ a la que se encuentre la *solución más superficial*.

3.2.1.2. Complejidad espacial.

Para analizar el tamaño de la frontera, preguntémonos ¿qué había en la frontera cuando alcanzamos la solución?. Es claro que si el nivel en que hallamos la solución es s , en general, no habremos hecho todas las acciones en dicho nivel y todo lo que hayamos hecho en el nivel s tiene hijos en los niveles inferiores. Por tanto la frontera se parece al último nivel que expandimos, quizá un nivel por debajo de este. En este caso *el número de nodos en la frontera* puede ser:

$$\mathbf{b}^s = \mathbf{O}(\mathbf{b}^s)$$

Será exponencial, aproximadamente como el último nivel : $\mathbf{O}(\mathbf{b}^s)$ como la complejidad temporal.

3.2.1.3. Completitud.

Es claro que si existe alguna solución, esta se ubica en un nivel *finito*, por tanto **es completo**. Al contrario de los prefijos más a la izquierda que pueden ser infinitos, los prefijos en altura son finitos.

3.2.1.4. Optimalidad.

Si *los costos son generales*, *no es óptimo*, pero *si los costes son todos 1, sí es óptimo*.

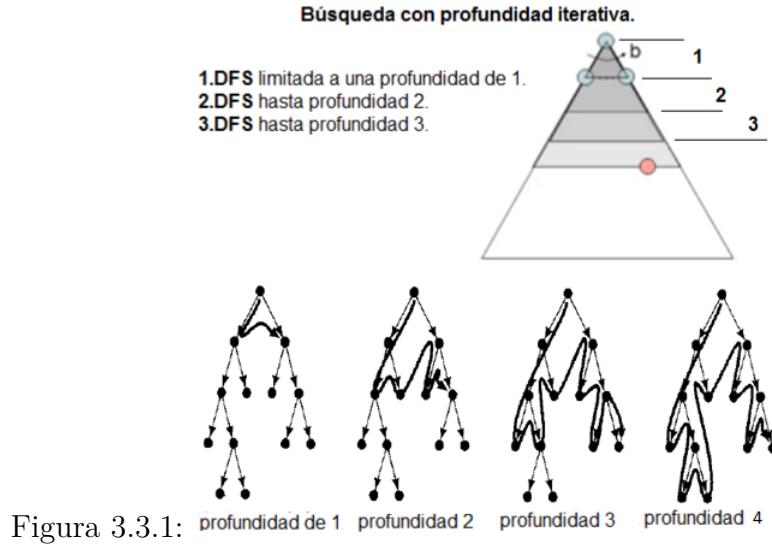
3.3 Búsqueda en profundidad iterativa.

Se trata de tener lo mejor de los dos tipos de búsqueda: *DFS* y *BFS*. La idea que guía a los algoritmos de profundidad iterativa es conseguir las ventajas espaciales de DFS con las temporales de BFS y las garantías de esta respecto a completitud y optimalidad.

El modo de conseguirlo es fácil:

1. se comienza por una búsqueda en profundidad limitada a una profundidad de 1. Si se encuentra la solución, perfecto, si no
2. se hace una búsqueda en profundidad hasta profundidad 2, si no se encuentra la solución
3. se hace una búsqueda en profundidad hasta profundidad 3

y así sucesivamente.



3.3.1 Justificación.

Como hemos visto, el número de nodos expandidos hasta el nivel s en *BFS* es:

$$N_{BF} = 1 + b + b^2 + \dots + b^s = \frac{b^{s+1} - 1}{b - 1}$$

Para el cálculo de los nodos expandidos en búsqueda en *profundidad iterativa (IDS)*, notemos que los nodos expandidos por una búsqueda *DFS* completa hasta el nivel j será:

$$N_{DFj} = \frac{b^{j+1} - 1}{b - 1}$$

En el peor de los casos, la búsqueda en profundidad iterativa cuando hay un objetivo a la profundidad d debe ejecutar búsquedas *DFS* completas para profundidades hasta d . La suma de los nodos expandidos para todas estas búsquedas es:

$$N_{ID} = \sum_{j=0}^d \frac{b^{j+1} - 1}{b - 1} = \frac{1}{b - 1} \sum_{j=1}^d (b^{j+1} - 1) = \\ \frac{1}{b - 1} [b \left(\frac{b^{d+1} - 1}{b - 1} \right) - (d + 1)]$$

y para d grande tendremos que

$$\lim_{d \rightarrow \infty} N_{ID}/N_{BF} = \frac{b}{b - 1}$$

que para $b = 10$, significa que expandimos un 11% más nodos que en *BFS*.

Generalmente la mayor parte del trabajo se realiza en el último nivel en que se busca, por tanto no es tan malo.

TEMA 4

BÚSQUEDA DE COSTE UNIFORME (UCS).

Hasta ahora hemos fijado como objetivo *minimizar el número de acciones a tomar*, y BFS es capaz de hallar esta solución. Esto es equivalente a asignar el mismo coste (1) a todas las acciones. Pero supongamos que cada arco del grafo (acción) tiene un coste diferente (Fig.4.0.1). Los algoritmos anteriores no son válidos en este caso.

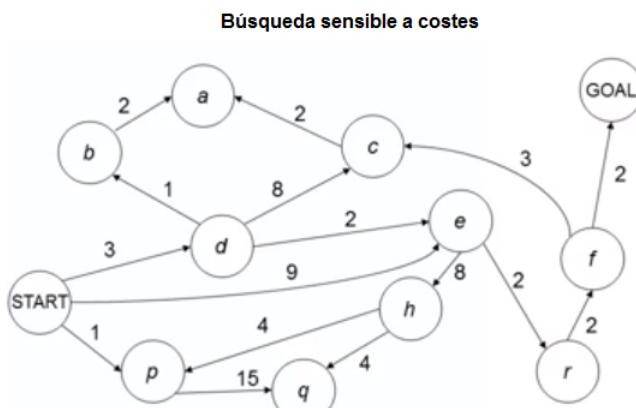


Figura 4.0.1:

Como ilustración pensemos en el mapa de Rumanía, en que la función a minimizar con los métodos de búsqueda antes expuestos sería el *número de trayectos entre ciudades* efectuados, cuando, en realidad, parece mejor criterio el de *minimizar la distancia recorrida* (coste).

4.1 Búsqueda de coste uniforme.

Es similar a BFS pero ahora *se priorizan las capas de menor coste*, de manera que ahora se ejecutan primero las acciones más baratas aun cuando haya múltiples pasos en el árbol.

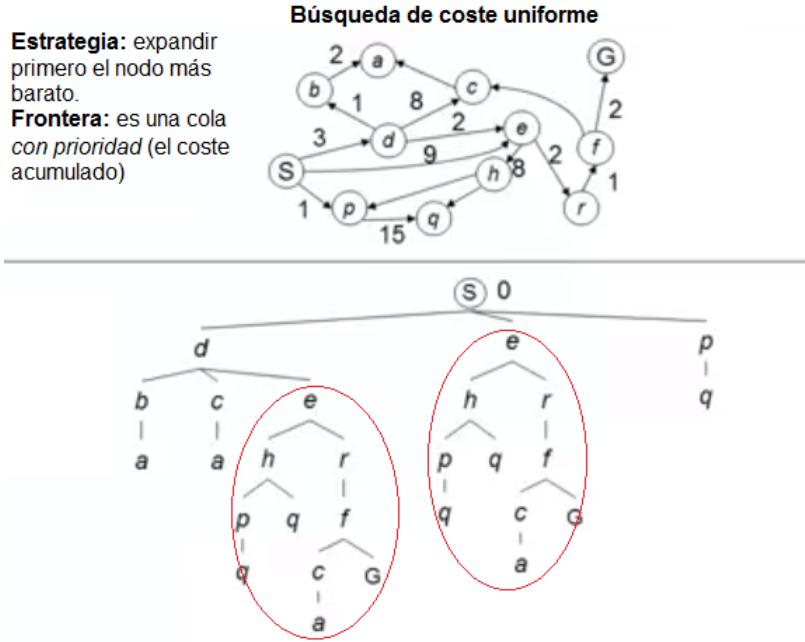


Figura 4.1.1:

Puesto que hemos de expandir el nodo más barato como estrategia, elegiremos el de **menor coste acumulado** que, en ese primer paso será p ($g=1$). Puesto que no es el objetivo, iremos después a d ($g=3$) y como (*estado==objetivo*)=Falso, seguiremos pero ahora no iremos a e ($g=9$), sino a b cuyo coste acumulado es $7 < 9$. En el siguiente paso expandiremos el nodo e (el más barato) y así sucesivamente, hasta llegar a la solución (Fig.4.1.2) que conduce a **G**.

Se ve que nos movemos desde el nodo raíz hacia abajo de un modo un tanto destaladado siguiendo los contornos de coste.

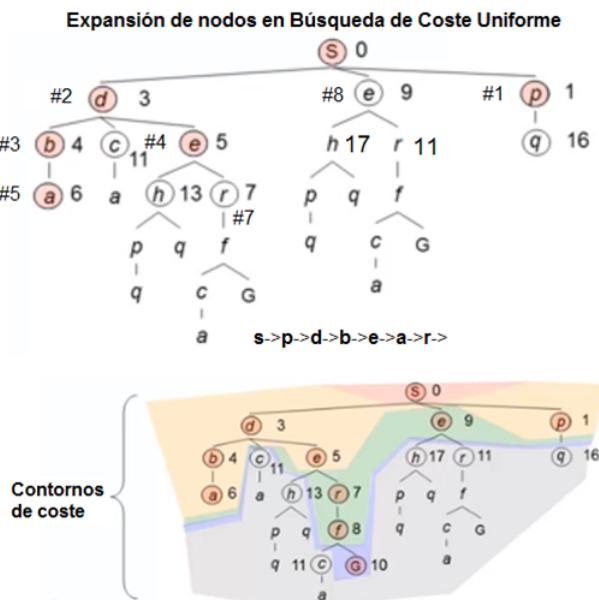


Figura 4.1.2:

4.2 Propiedades de la UCS.

Sabemos que expandimos primero los nodos de menor coste, sea este $g = 1$ y seguimos por los de $g = 2, 3, \dots$ y así hasta el nodo que nos de la solución de menor coste.

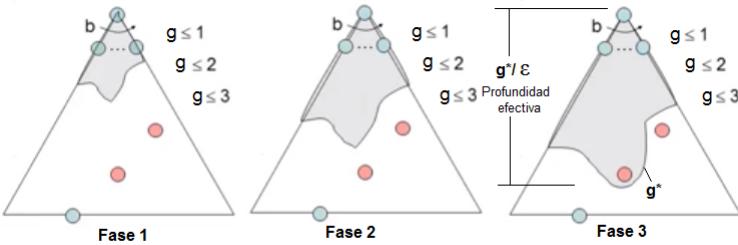


Figura 4.2.1:

Expansión en UCS

Si la solución tiene un coste g^* y los arcos cuestan como mínimo ϵ - es decir, el coste mínimo de cada acción - la *profundidad efectiva* será

$$\frac{\text{coste mínimo solución}}{\text{coste mínimo acción}} = \frac{g^*}{\epsilon}$$

A partir de aquí, podemos decir que la **complejidad temporal**, *como en la BFS*, será exponencial en la profundidad efectiva:

$$O(\frac{g^*}{\epsilon})$$

La **complejidad espacial** será, por las mismas razones,

$$O(\frac{g^*}{\epsilon})$$

Completitud. Suponiendo que la mejor solución tiene un *coste finito* y los costes de los arcos son positivos, **es completo**, es decir, garantiza encontrar una solución.

Optimalidad. Puede demostrarse que es óptima y se demostrará en próximos capítulos ya que *se trata de un caso particular de la búsqueda A** ('A estrella'). Por tanto, no devuelve la solución más superficial - lo que hacia *BFS* -, sino la de menor coste.

Podemos resumir las ventajas e inconvenientes de la UCS:

1. Ventajas:

- Es completa.
- Es óptima.

2. Desventajas:

- Busca en *cualquier dirección*.
- *No tiene información sobre la ubicación del objetivo.*

Esto significa que, partiendo del estado inicial, examina cualesquiera direcciones que sean baratas sin importarle si ayudan a avanzar hacia el objetivo o se aleja de él, pero esto puede corregirse, como veremos.

Ejercicio: Hallar la solución del grafo siguiente con el método UCS.

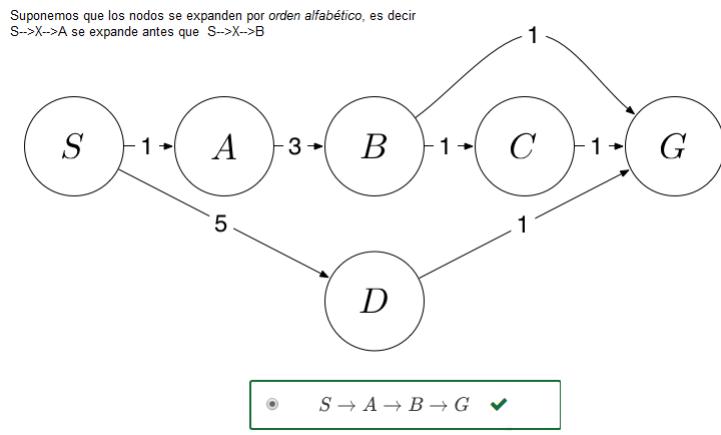


Figura 4.2.2:

4.3 Resumen de algoritmos de búsqueda no informada.

La tabla siguiente resume las características de cada algoritmo visto hasta ahora:

Resumen de algoritmos de búsqueda no informada

Criterio	Primero en anchura	Coste uniforme	Primero en profundidad	Profundidad iterativa
Completo?	Sí *	Sí *	No	Sí
Tiempo	b^{d+1}	$b^{\lceil C/\varepsilon \rceil}$	b^m	b^d
Espacio	b^{d+1}	$b^{\lceil C/\varepsilon \rceil}$	$b \cdot m$	$b \cdot d$
Óptimo?	Sí *	Sí	No	Sí*

Figura 4.3.1:

El siguiente ejemplo servirá de resumen a todo lo visto hasta ahora y dará alguna orientación acerca del manejo de las colas para gestionar la frontera de los algoritmos.

Ejemplo¹: Tenemos una pila de cuatro tortitas de diferentes tamaños y deseamos ordenar las de mayor en la base a menor en la cúspide. Para ello solo disponemos de una espátula que introduciremos en cualquier *posición* de la pila y voltearemos todas las que se sitúen sobre esta.

Se trata de hallar la solución de *mínimo coste* a este problema de búsqueda.

El **estado** es el orden de las tortitas.

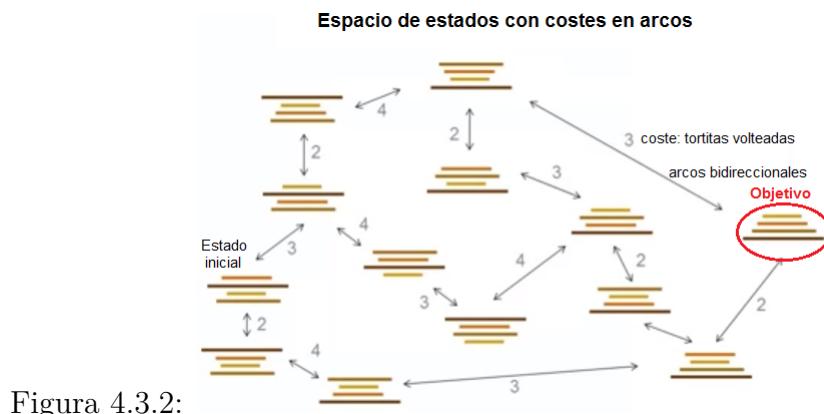
Las **acciones** son las *posiciones* en que puedo insertar la espátula.

La **función sucesor** da la vuelta en el orden definido por la posición de la espátula (acción) y determina la nueva ordenación.

El **coste** puede definirse de diferentes formas:

1. El número de volteos que debo dar. Esto equivale a dar un coste unitario a cada acción.
2. El número de tortitas volteadas, que nos parece más adecuado.

Veamos el espacio de estados de este problema:



Utilizaremos el algoritmo de búsqueda ya conocido:

¹De un artículo de William Henry Gates (*Bill Gates*):

<https://people.eecs.berkeley.edu/~christos/papers/Bounds%20For%20Sorting%20By%20Prefix%20Reversal.pdf>

```

function BUSQUEDA-ARBOL(problema, estrategia) return solución o fallo
    inicializa el árbol con el estado inicial del problema
    loop do
        if no hay candidatos para la expansión then return fallo
        elegir nodo para expandir según estrategia
        if nodo contiene estado objetivo then return solución
        else expandir el nodo y añadir los nodos resultantes al árbol de búsqueda
    end

```

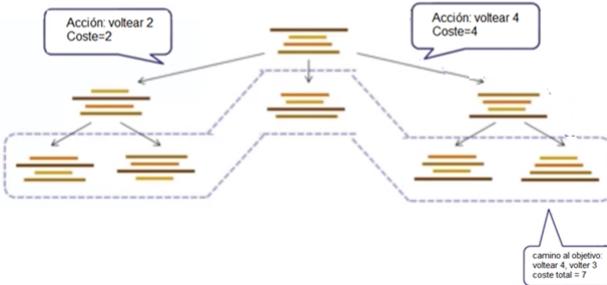


Figura 4.3.3:

Cola única.

Como sabemos este algoritmo se repite de manera simple en ciclos:

- se toma algo de la frontera,
- se generan sus hijos según la *función-sucesor* y se ponen en la *frontera*
- las acciones que han generado los hijos tienen costes y estados resultantes que se ponen en la *frontera*,
- se toma algo de la frontera,
- se generan sus hijos según la *función-sucesor* y se ponen en la *frontera*
- las acciones que han generado los hijos tienen costes y estados resultantes que se ponen en la *frontera*,
-

La frontera va yendo hacia abajo paulatinamente, más o menos rápido según el algoritmo utilizado y a medida que avanzamos acumulamos los costes. Algunos algoritmos son sensibles a los costes (UCS) y otros no (BFS, DFS), pero *todo plan tiene un coste*.

Todos estos algoritmos sin información son básicamente iguales excepto por la estrategia de selección de la frontera. Conceptualmente son *colas con diferentes prioridades*. Desde el punto de vista de su implementación, podemos pasar la estrategia de búsqueda como un parámetro. Sin embargo, en BFS y DFS no se necesita mantener una cola con prioridad, que añade una carga al programa, sino que basta con gestionar una pila. La gestión de una cola con prioridades introduce una penalización en el tiempo de ejecución de $\log(n)$.

4.4 Estados repetidos: búsqueda en grafos.

Hasta este punto hemos ignorado una de las complicaciones más importantes del proceso de búsqueda: la posibilidad de perder tiempo expandiendo estados que ya han sido visitados y expandidos. Para algunos problemas, esta posibilidad o aparece; el espacio de estados es un árbol y hay sólo un camino a cada estado.

Para algunos problemas la repetición es inevitable como en el siguiente ejemplo:

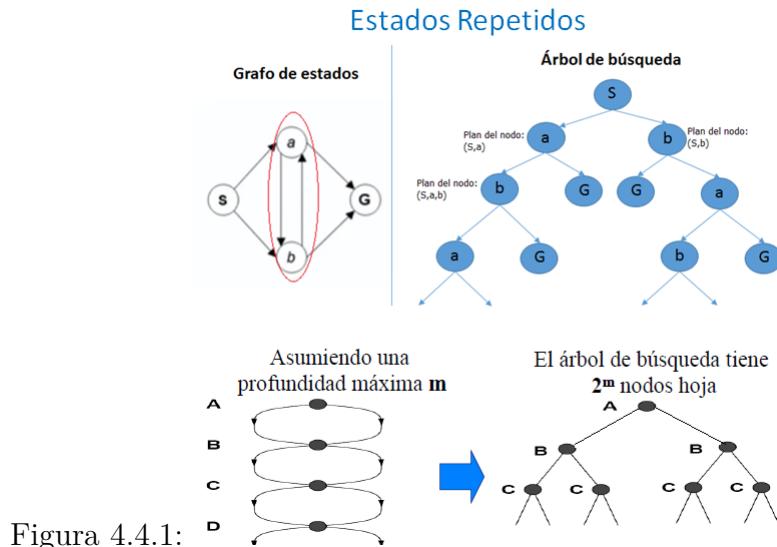


Figura 4.4.1:

Es claro que eliminando estados repetidos reduciríamos el árbol de 2^d a $d+1$ hojas. La detección de estados repetidos por lo general significa la comparación del nodo a expandir con aquellos que han sido ya expandidos; si se encuentra un emparejamiento, entonces el algoritmo ha descubierto dos caminos al mismo estado y puede desechar uno de ellos.

Si nos fijamos en la Fig. 4.1.2 vemos que los dos subárboles que arrancan en e se repiten aun cuando el nodo e está a diferente profundidad. El árbol más profundo no nos dirá nada que ya no sepamos.

Hay tres maneras de evitar estados repetidos:

1. No volver nunca al estado del que se proviene. La función sucesor no generará ningún sucesor que sea el mismo estado que el padre.
2. No crear caminos cíclicos. La función sucesor no generará ningún sucesor de un nodo que sea el mismo estado que alguno de sus ancestros.
3. No generar ningún estado que ya ha sido generado antes. Esto requiere conservar en memoria todos los estados generados.

Para implementar esto último, el algoritmo usa con frecuencia una tabla *hash* (lista-cerrada) que almacena todos los nodos que se generan lo que proporciona una eficiencia razonable a la comprobación de estados repetidos. No obstante, con esto

- a) hay requerimientos adicionales de memoria,
- b) es *óptimo* únicamente en *UCS* y en *BFS* con coste de paso constante y
- c) *no es óptimo* con *otras estrategias*.

Para la búsqueda *DFS*, los únicos nodos en memoria son aquellos del camino desde la raíz hasta el nodo actual. La comparación de estos nodos permite al algoritmo descubrir los caminos que forman ciclos y que pueden eliminarse inmediatamente.

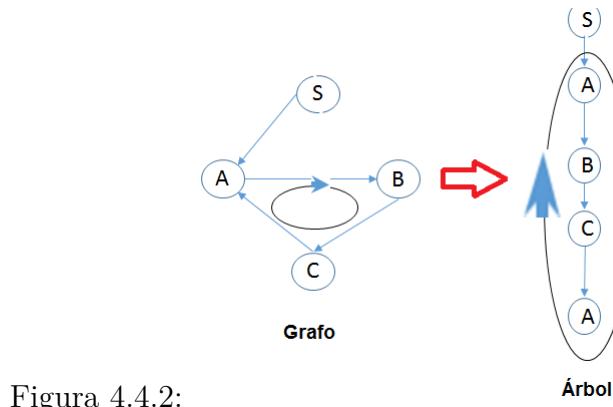


Figura 4.4.2:

Esto, sin embargo, solo sirve para evitar que los ciclos formen árboles de búsqueda infinitos, pero esto no evita problemas como el de la figura 4.4.1 en que se generan caminos exponencialmente. Para evitar estos casos se guardan los nodos ya expandidos en la memoria: *los algoritmos que olvidan su historia, están condenados a repetirla*. Podemos crear una *lista-cerrada*² que contenga los *nodos ya expandidos*. Si el nodo actual se encuentra en *lista-cerrada* se elimina en lugar de expandirlo. Así tendremos un nuevo algoritmo denominado **BÚSQUEDA-GRAFOS** que es mucho más eficaz que BÚSQUEDA-ÁRBOLES.

La idea del algoritmo es fácil: «*no expandir nunca dos veces el mismo estado*».

¿Es este algoritmo completo? ¿Es posible que perdamos la solución por no expandir ciertos nodos en el árbol? Esto no puede suceder porque no expandimos lo que ya se expandió en otro nodo.

Si examinamos la optimalidad de este algoritmo, es difícil de dilucidar. Cuando el algoritmo desecha un camino, siempre desecha el recién descubierto que puede ser el de menor coste. Sin embargo, puede demostrarse que esto no puede suceder cuando se utiliza la *UCS* o la *BFS* con costes constantes que no solo son *estrategias óptimas en árboles*, sino también en grafos. La búsqueda con profundidad iterativa, por otra parte, utiliza la expansión primero en profundidad y fácilmente puede seguir un camino subóptimo, lo que exige comprobaciones adicionales.

En la figura se muestra el algoritmo general de búsqueda en grafos:

²A veces se denomina a la frontera: *lista-abierta*. No obstante, es preferible llamarlo *conjunto cerrado* ya que si se trata como una lista los resultados son peores.

```

problema = {nodo-raíz, expandir, test-objetivo}; estrategia
function búsqueda-en-grafo (problema, estrategia)
/* devuelve solución o fallo
lista-abierta contiene los nodos de la frontera de árbol-de-búsqueda*/
Iniciarizar árbol-de-búsqueda con nodo-raíz
Iniciarizar lista-abierta con nodo-raíz
Iniciarizar lista-cerrada a una lista vacía
Loop do
  If (lista-abierta está vacía) then return fallo
  Elegir dentro de lista-abierta, de acuerdo a la estrategia, un nodo a expandir.
  If (nodo satisface test-objetivo)
    then return solución (camino desde nodo-raíz a nodo)
    eliminar nodo de lista-abierta
    If (nodo no está en lista-cerrada)
      then añadir nodo a lista-cerrada
      expandir nodo
      añadir nodos hijo a lista-abierta
end

```

Algoritmo general de búsqueda-en-grafo

Figura 4.4.3:

En otras palabras el algoritmo consiste en:

1. Crear un árbol de búsqueda que consiste solo en el nodo de inicio S y poner S en un una lista llamada LISTA-ABIERTA.
2. Crear una lista LISTA-CERRADA inicialmente vacía.
3. if LISTA-ABIERTA=vacía then return FALLO.
4. Elegir primer nodo n de LISTA-ABIERTA y moverlo a LISTA-CERRADA.
5. if n satisface TEST-OBJETIVO then return SOLUCIÓN (camino desde S a n).
6. Expandir n generando un conjunto M de sucesores y añadirlos al árbol creando arcos desde n a cada miembro de M y añadir a LISTA-ABIERTA.
7. Reordenar la LISTA-ABIERTA de acuerdo con algún esquema (Heurística p.e. u otro criterio)
8. Volver al paso 3.

El algoritmo vale tanto para BFS como para DFS. En BFS los nodos nuevos se ponen al final de la LISTA-ABIERTA (FIFO) Y EN DFS al comienzo. En A* se ordenan según el mérito heurístico.

Ejemplos:

El siguiente mapa es una muestra reducida del metro de Madrid en el que en las estaciones marcadas puede hacerse transbordo de una línea a otra de las que en ella confluyen. Encontrar el camino con menos transbordos para llegar desde AEROPUERTO-T4 a CANAL (Fig.4.4.4)

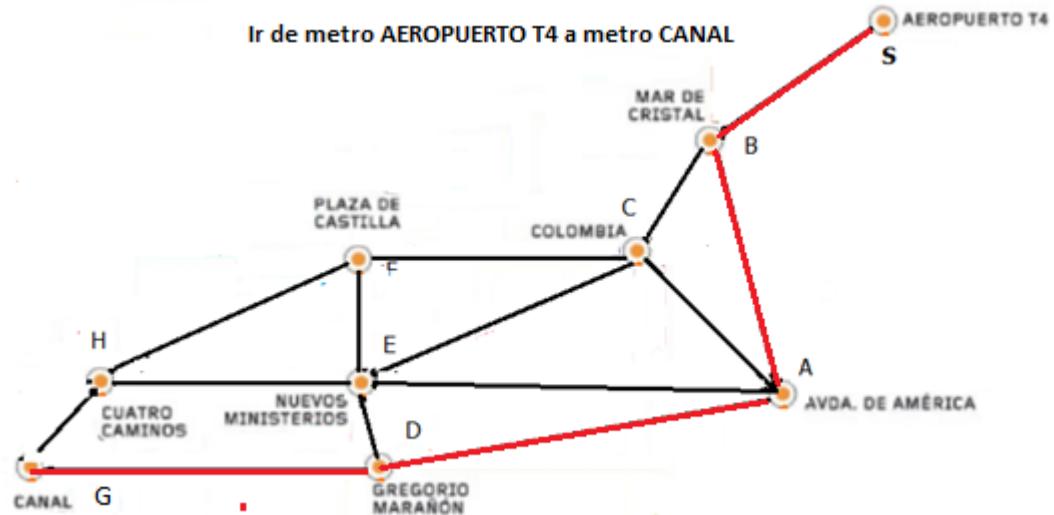


Figura 4.4.4:

Si lo planteamos como búsqueda en árbol, muchos de los nodos están repetidos (Plaza Castilla, Nuevos Ministerios,...). Operando como grafo, evitando repeticiones, llegamos a la solución mostrada en el gráfico, junto con la evolución de la LISTA-CERRADA.

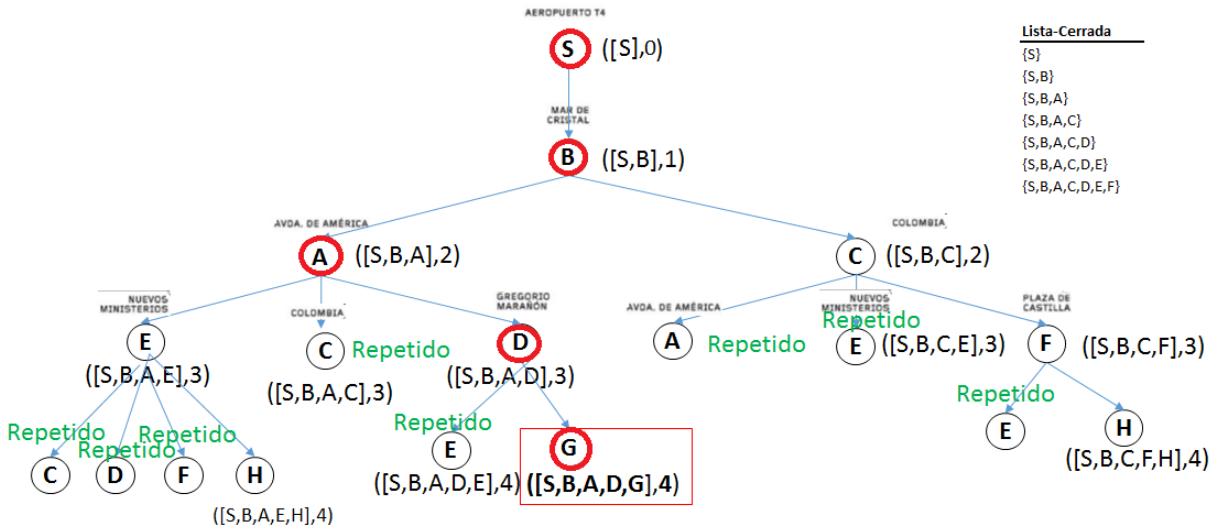


Figura 4.4.5:

EJERCICIOS.

1. Cuántos nodos hay en el siguiente árbol de búsqueda completo para el grafo del espacio de estados mostrado en la figura. El estado inicial es S.

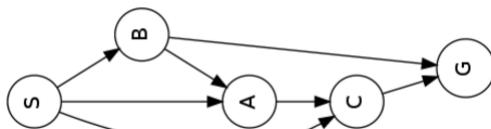


Figura 4.4.6:

Solución: dibujemos el árbol de búsqueda, vemos que son 7 nodos.

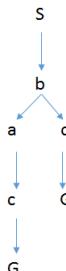


Figura 4.4.7:

2. Considerar el grafo para BFS de la figura 4.5.3. Con las mismas indicaciones que el problema anterior, hallar el camino final devuelto por BFS.

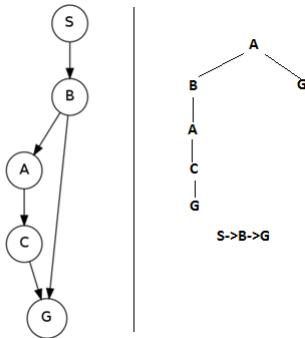


Figura 4.4.8:

3. Considerar el grafo para BFS de la figura 4.5.4. Con las mismas indicaciones que el problema anterior, hallar el camino final devuelto por BFS.

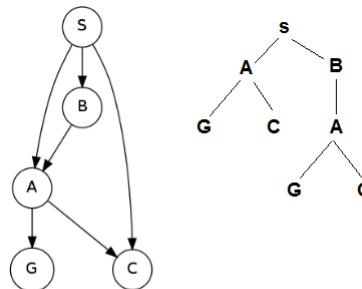
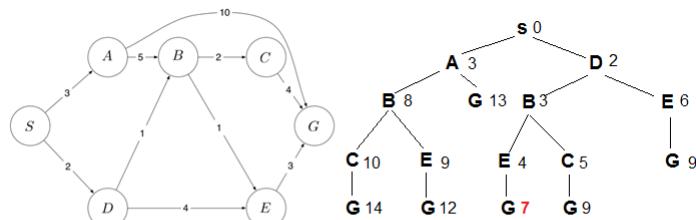


Figura 4.4.9:

✓

4. En el grafo de la figura, con la convención de ordenamiento establecida anteriormente, encontrar los caminos para las siguientes modalidades de búsqueda:

1. DFS
2. BFS
3. UCS



Soluciones:

DFS:
 $S \rightarrow A \rightarrow B \rightarrow C \rightarrow G$

BFS:
 $S \rightarrow A \rightarrow G$

UCS:
 $S \rightarrow D \rightarrow B \rightarrow E \rightarrow G$

Figura 4.4.10:

TEMA 5

BÚSQUEDA INFORMADA.

Los problemas de *búsqueda informada* se construyen a partir de los tipos de problemas que hemos visto antes e inyectándoles información acerca de la localización de las soluciones de modo que se reduzcan las exploraciones necesarias en el árbol de búsqueda.

La idea básica de la búsqueda informada es la **heurística**. Expondremos este tipo de búsqueda del siguiente modo:

1. Heurística.
2. Búsqueda avaricia o voraz.
3. Búsqueda *A**(‘A estrella’).

5.1 Heurística de búsqueda.

Como hemos dicho, se trata de añadir algo adicional a los algoritmos de búsqueda anteriores: una función que nos permite examinar un estado y decir hasta qué punto nos estamos acercando al objetivo. Esta es la idea fundamental de la búsqueda informada: algo que no solo nos dice si *estado==objetivo*, sino si estamos cerca de este. El objeto clave es una *heurística de búsqueda*.

Una **heurística de búsqueda** es

- una función que **estima** cuan próximos estamos al objetivo. Toma como argumento uno de los estados del mundo y nos devuelve un número: $W : \stackrel{h}{\rightarrow} \mathbb{R}$

La heurística toma un estado del mundo y nos devuelve un número $h = h(e)$ que si es grande indica que estamos lejos del objetivo buscado. Es claro que, con esta herramienta, podemos priorizar en la frontera aquellas áreas que la heurística nos señala como más próximos al objetivo.

- y está diseñada **para un problema concreto**.

Veamos un ejemplo: Pac-man tiene el problema de llegar al punto. Puede emprender una búsqueda de las que hemos visto pero sería de ayuda saber lo lejos que se encuentra del punto en cada posición. Para ello podría utilizar algo como la *distancia de Manhattan*. Es claro que, aunque los muros le pueden hacer ir más lejos, si esta distancia es pequeña estará cerca del punto (Fig. 5.1.1). No es perfecto, pero sí útil.



Figura 5.1.1:

Podríamos haber utilizado también la distancia euclídea, pero esta distancia no toma en cuenta que la existencia de muros impide caminar en diagonal.

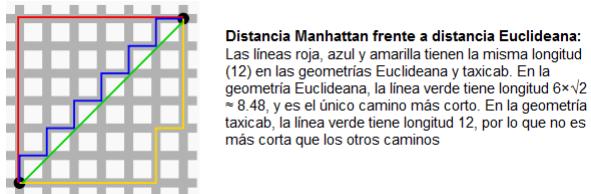
Volviendo a nuestro problema de Rumanía, podemos utilizar como heurística la distancia en línea recta desde nuestra ubicación -ciudad - a Bucarest. (Fig. 5.1.2.). Como puede observarse esta distancia es 0 para Bucarest-Bucarest que es lo esperable de una heurística: el valor numérico es nulo en el objetivo. Asimismo puede verse que cuanto más cerca estamos del objetivo, menor es la distancia en línea recta.



Figura 5.1.2:

¿Qué heurística podríamos utilizar en el caso de las tortitas? Es claro que ni la distancia euclídea ni la de Manhattan nos sirven. La respuesta es: *el número de la tortita más grande que aún está fuera de su posición*. La idea tras esta heurística es que si la tortita más grande está fuera de orden, la única manera de llevarla a su posición es situarla en la cúspide y voltear toda la pila.

Nota: Distancia de Manhattan (o taxicab).



5.2 Búsqueda avariciosa o voraz.

Es un tipo de búsqueda que utiliza la heurística pero es algo imperfecta. Ilustremos el caso con la Fig. 5.1.2. partiendo de Arad. Utilizaremos la heurística allí expuesta para priorizar los elementos que salen de la frontera.

En la figura hemos representado el camino más corto entre Arad y Bucarest en azul (el inferior), aunque la búsqueda DFS nos daría como solución el camino superior- pasa por Fagaras - que tiene menos tramos (3 frente a 4).

Para la búsqueda avariciosa procederemos así:

- expandir el nodo que parece más próximo
 - inicialmente es Sibiu.
 - seguidamente será Fagaras (que sabemos no está en el camino óptimo)
 - por último llegamos a Bucarest pero por un camino más largo que la ruta azul.

¿Qué ha fallado en esta búsqueda? por lo pronto *hemos alcanzado el objetivo pero no hemos minimizado las distancias* (costes).

Al contrario que en UCS, aquí *las heurísticas no se suman* y siempre nos encaminamos al estado que parece estar más próximo al objetivo.

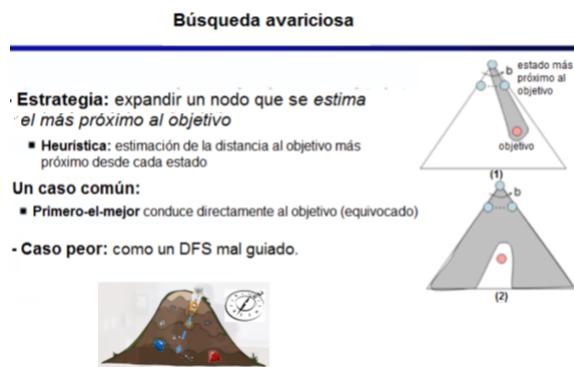


Figura 5.2.1:

Como se ve en la figura es similar a una DFS centrada en el estado más próximo al objetivo. Pero la búsqueda avariciosa es tan buena como la heurística que la guía y, en general, las heurísticas no son perfectas. Es incluso imaginable un caso como el (2) de la Fig. 5.1.3. en que se comporta

como una DFS pésimamente dirigida. Cuando funciona, lo hace bien, pero en otras ocasiones falla estrepitosamente . Este tipo de búsqueda no selecciona los casos en que es preciso volver hacia atrás.

Ejemplo: ¿Qué solución proporciona la búsqueda avariciosa al caso de la Fig 4.4.8?

Si usamos como heurística la *mínima distancia e cada nodo al nodo G* tendremos que esta viene dada por la tabla siguiente:

Nodo	Distancia mínima a G
A	9
B	4
C	4
D	7
E	3
G	0

La solución será $S \rightarrow D \rightarrow E \rightarrow G$ que tiene una distancia real de 9, mientras que la solución UCS es $S \rightarrow D \rightarrow B \rightarrow E \rightarrow G$ con distancia 7 (< 9).

5.3 Búsqueda A*.

Todo lo anterior nos servirá para desarrollar el algoritmo fundamental de búsqueda en inteligencia artificial el **A***(A estrella).

Hasta ahora hemos estudiado, entre otros, dos casos extremos:

1. UCS que es un proceso parsimonioso y constante que conduce inexorablemente al objetivo y
2. la búsqueda avariciosa que avanza rápido pero a veces acierta y otras no.

Nos proponemos construir un algoritmo que combine la seguridad de UCS con la rapidez de la búsqueda avariciosa.

¿Cómo pueden combinarse ambas búsquedas?

Comparemos el modo de actuar de cada algoritmo:

1. UCS elige los caminos con *menor coste acumulado* primero medida por $g(n)$.
2. búsqueda avariciosa ordena por *proximidad al objetivo* medida por $h(n)$.

Fijémonos en el caso expuesto en la Fig.5.3.1. resaltando en la parte resaltada de la figura inferior que es el un camino al objetivo.

Combinado UCS y búsqueda avariciosa

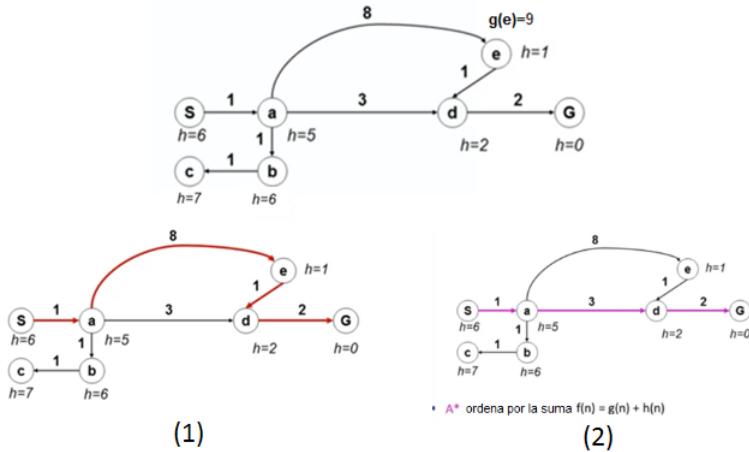


Figura 5.3.1:

¿Qué hace A^* ? ordena por la suma $g(n) + h(n)$ es decir, el coste acumulado hasta el nodo (coste pasado) , más la heurística (coste estimado futuro), esto es, el *coste estimado más barato del camino que pasa por n*. Por tanto A^* no irá hacia c muy pronto puesto que $h(c)+g(c) = 7+3 = 10$ la heurística - coste futuro -es grande y tampoco se precipitará a la rama $a - e$ (1) porque su coste es elevado. tenderá a ir por el centro (2).

Veamos sobre un ejemplo sencillo cómo funciona A^* en la figura siguiente:

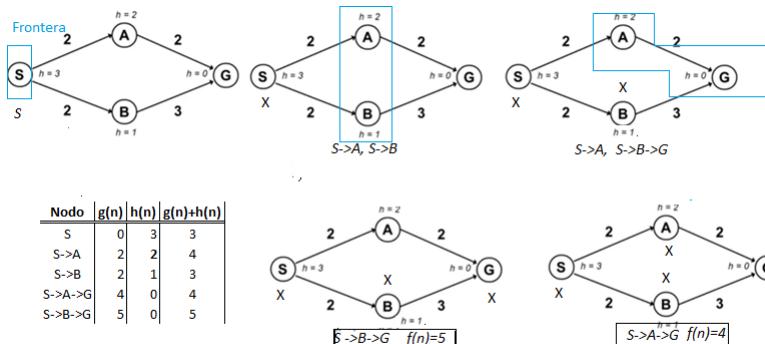


Figura 5.3.2:

Hemos ido priorizando en la frontera aquellos nodos con menor coste estimado. Sin embargo aun cuando hemos llegado primero a la solución $S \rightarrow B \rightarrow G$ esta no es la que tiene un mínimo de $f(n)$ sino la $S \rightarrow A \rightarrow G$.

Ejercicio. En el árbol de búsqueda A^* siguiente, tras expandir S hay dos nodos en la frontera: $S \rightarrow D$ y $S \rightarrow A$. Complete los valores indicados en la figura.

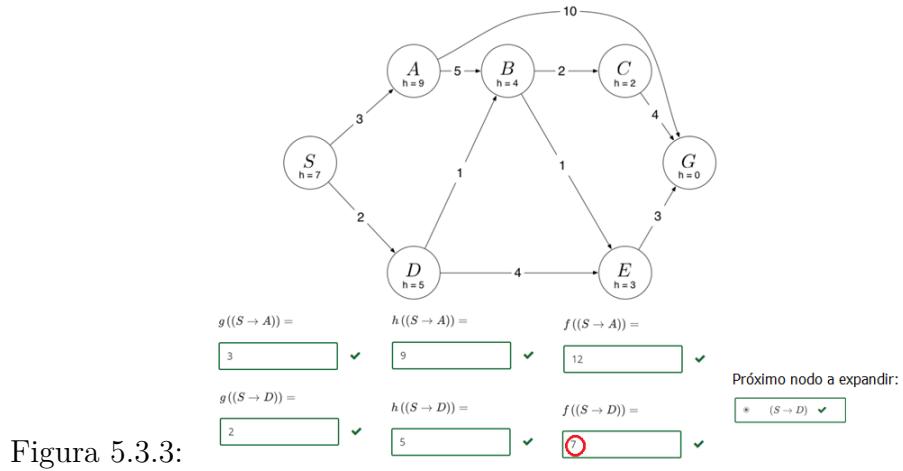
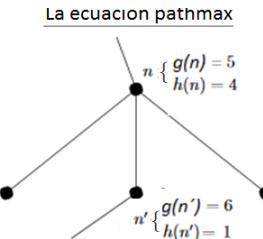


Figura 5.3.3:

Si nos fijamos en los árboles de búsqueda, veremos que, en su mayor parte, la función $f(n) = g(n) + h(n)$ no disminuye, esto es, $f(n)$ es monótona no decreciente. Si la heurística pertenece a las pocas que no son monótonas, es fácil volverlas monótonas. Sean p , e dos nodos n - padre y n' - hijo. Sea que $g(n) = 3$ y $h(n) = 4$, de donde, $f(n) = 7$ y $g(n') = 4$, $h(n') = 2 \rightarrow f(n') = 6$. Pero como todo camino a través de n' es también un camino que pasa por n , es claro que el valor 7 es despreciable porque sabemos que el coste real es al menos 7. Por tanto, cada vez que generamos un nodo podemos comprobar si su coste $f(n')$ es menor que el de su padre $f(n)$ y si esto es así usamos el coste del padre:

$$f(n') = \max(f(n), g(n') + h(n')) \quad (5.3.1)$$

La ec. 5.3.1 se llama ecuación **pathmax**.



Por tanto $f(n) = 9$ y $f(n') = 7$.

Que $f(n) = 9$ nos dice que el coste de un camino pasando por n es como mínimo 9 (porque $h(n)$ es admisible).

Pero n' está en el camino que pasa por n . Por tanto decir que $f(n') = 7$ no tiene sentido

$$f(n') = \max\{f(n), g(n') + h(n')\}$$

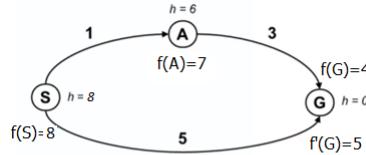
Figura 5.3.4:

De este modo podemos dibujar en contornos en el espacio de estados en que f es creciente (Fig. 5.10.5).

5.4 Heurísticas admisibles. Optimalidad de A*.

Nos preguntamos si el algoritmo A estrella es óptimo.

Si nos fijamos en la figura 5.4.1. vemos que A^* da como solución $S \rightarrow G$ con coste 5 mientras que $S \rightarrow A \rightarrow G$ tiene un coste de 4. El error se origina en que el coste real del camino malo - 5 - es menor que el coste estimado del camino bueno - 7. En particular el valor $h = 6$ de la heurística en el nodo A sobreestimó el coste de alcanzar G e hicimos caso a la heurística.



$h(A)=6$: Solución de A^* : $S \rightarrow G$. $C=5$.
Coste de $S \rightarrow A \rightarrow G$: 4
Coste real del camino malo: 5
Coste estimado del camino bueno: 7

Figura 5.4.1: Si $h(A) < 3$: la solución es $S \rightarrow A \rightarrow G$

Se necesita que nuestras estimaciones no superen los costes reales. Si el valor $h(A) < 3$ la solución es la correcta.

Esto nos lleva a la idea de **admisibilidad**:

- una heurística pesimista que sobreestima costes es inadmisible porque es subóptima y deja atrapados a los buenos planes en la frontera. Esta heurística siempre nos dice que estamos más lejos del objetivo de lo verdadero.
- una heurística optimista es admisible porque retarda los planes malos en la frontera pero no sobreestima los costes reales y finalmente, el camino de coste mínimo es el ganador.

Por tanto *una heurística es admisible si el coste estimado no supera el coste real al objetivo más próximo:*

$$h(\mathbf{n}) \leq h^*(\mathbf{n})$$

Ejemplos de heurísticas admisibles

1. La distancia Manhattan en el problema de la Fig. 5.1.1. $h = \Delta x + \Delta y$ es admisible porque moviéndose Pac-man en direcciones perpendiculares, al haber muros, siempre recorrerá más que la distancia Manhattan.
2. En el problema del mapa de carreteras, la distancia en línea recta es una función heurística admisible.

Demostremos que *si la heurística es admisible A^* es óptimo*, esto es que:

- Si p^* es el camino óptimo con coste C^* .
- Si p' es un camino subóptimo con coste $c(p') > C^*$

Demostraremos que cualquier subcamino p'' de p^* en la frontera será expandido antes que p' y por tanto A* no tomará p' como parte de la solución.

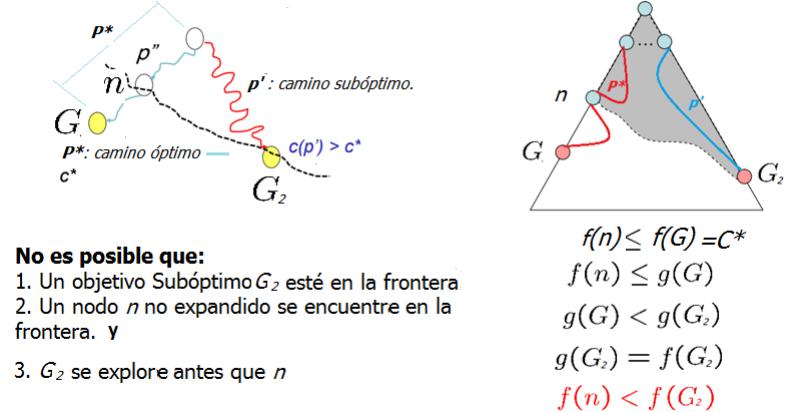


Figura 5.4.2:

Teorema. Si se usa búsqueda en árbol¹, y h es admisible A* es completa y óptima

Demostración:

Sea C^* : coste de la solución óptima. $g(n)$: coste hasta el nodo n . $h(n)$: heurística en nodo n .

- Considérese G_2 un nodo objetivo que está en un camino subóptimo (i.e. $g(G_2) > C^*$, $h(G_2) = 0$ - se cumple para todo nodo objetivo) que está en la frontera del árbol de búsqueda:

$$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^* \Rightarrow f(G_2) > C^* \quad (5.4.1)$$

- Considérese el nodo n del conjunto frontera del árbol de búsqueda que está en un camino solución óptimo.

- Dado que n está en el camino solución óptimo, $g(n) = g^*(n)$
- Dado que h es *admissible*: $h(n) \leq h^*(n)$

$$f(n) = g(n) + h(n) \leq g^*(n) + h^*(n) = C^* \Rightarrow f(n) \leq C^* \quad (5.4.2)$$

Sumando (5.4.1) + (5.4.2): tenemos:

$$f(n) \leq C^* < f(G_2) \quad (5.4.3)$$

y se explora n antes que G_2 c.q.d.

¹En la búsqueda en grafo, no se verifica. Para este caso se requiere la consistencia o monotonía de $h(n)$.

5.5 Propiedades de A*.

Si lo comparamos con UCS encontraremos algunas similitudes. UCS comenzará en el nodo raíz y descenderá de un modo un tanto zigzagueante hacia abajo buscando el camino de menor coste.

Por su parte A* profundizará más cerca de los objetivos y buscará más superficialmente lejos de estos. Pero siendo la heurística imperfecta en la cúspide buscará extensivamente.

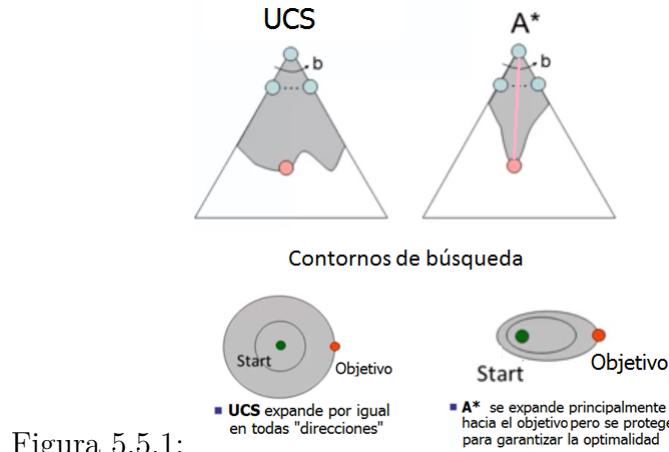


Figura 5.5.1:

Ejercicio: En el grafo adjunto responder a las preguntas:

1. ¿Es la heurística admisible?. Respuesta: sí porque $h \leq h^*$
2. ¿Cuál es la solución A*? Respuesta: $S \rightarrow D \rightarrow B \rightarrow E \rightarrow G$
3. ¿Qué nodos se expandirán en la búsqueda? Respuesta: $S \rightarrow D \rightarrow B$

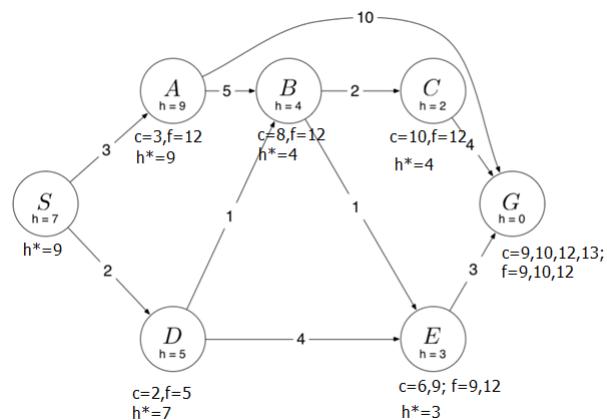


Figura 5.5.2:

5.6 Aplicaciones de A*.

1. Problemas de encaminamiento/ruteo.

2. Video juegos.
3. Planificación de recursos.
4. Análisis del lenguaje.
5. Traducción-máquina.
6. Reconocimiento de voz....

5.7 Problemas relajados. Construcción de heurísticas admisibles.

Lo más difícil de algoritmo A* es encontrar las heurísticas admisibles. Con frecuencia estas son soluciones de los llamados *problemas relajados* en los que son posibles nuevas acciones. A los problemas como el analizado con menos restricciones, se les llama **problemas relajados** y con frecuencia el coste de una solución óptima del problema relajado sirve como heurística para el problema real. La idea es que aumentando las posibilidades de actuar, reducimos la cota inferior de los costes del problema más complejo.

P.e. el ejemplo de Pac-man, usar la distancia Manhattan es equivalente a resolver el problema relajado en que no existen paredes.

Ilustremos este problema usando el conocido 8-puzzle² donde se trata de llegar desde la posición inicial - vacío en el centro - a la final - vacío en la esquina NE.



Figura 5.7.1:

Nuestro problema consiste en encontrar la solución que precise el mínimo número de movimientos (coste = 1).

Necesitamos una heurística, para lo cual se nos ocurren algunas ideas:

- *Número de cuadrados mal colocados.* Veamos si cumple las condiciones requeridas para ser una heurística para el *estado inicial*:

- ¿Proporciona un número? Sí, un número ≤ 8
- ¿Es admisible? Para ello h de ser menor que el número que el número de pasos requerido para llevar el número a su posición. Con un solo movimiento puedo llevar un cuadrado y solo uno a suposición o no hacerlo. Por tanto estando los 8 fuera de sitio en el estado inicial, 8 es menor que los movimientos necesarios.

²Aun cuando los 9 cuadrados pueden colocarse en $9!$ posiciones diferentes, puede demostrarse desde una ordenación dada solo pueden alcanzarse $9!/2$ posiciones.

Por tanto $h(\text{inicio}) = 8$

Se trata de un ejemplo de *problema relajado* que puede resolverse en 8 pasos. Concretamente el problema relajado consiste en partir de un marco vacío y 8 cuadrados y situarlos en la posición del estado objetivo.

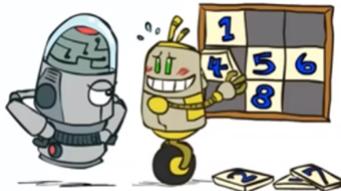


Figura 5.7.2:

Sin embargo, es una heurística exageradamente relajada.

- Podemos imaginar una versión menos relajada en la que cada cuadrado pueda moverse en cualquier dirección ignorando la existencia del resto de los cuadrados P.e. podríamos llevar el cuadrado 1 a la posición final como si el 2, que la ocupa, no existiera. Preciso un paso a la izquierda - 1 \leftarrow y dos arriba - 2 \uparrow . Podríamos posiblemente, sumar todas las distancias Manhattan de los cuadros. $h(\text{inicio}) = 3 + 1 + 2 + \dots + 2 = 18$.

Pero, ¿cómo estar seguro de que es admisible? Hay dos maneras:

- mostrando el caso de relajación, como se ha hecho o
- por demostración directa.

Si comparamos el resultado de ambas heurísticas observamos el cumplimiento de un principio general:

«cuanto menos relajado sea el problema que sirve de heurística - más cercano está al coste real -, menos trabajo se necesita para llegar a la solución».

¿Podemos usar el coste real de llegar al objetivo como heurística? Desde luego que sí y, obviamente, será admisible. Como contrapartida, aun cuando se expandan pocos nodos, en cada uno de ellos será preciso lanzar un trabajo para calcular el coste de alcanzar el objetivo. Esto nos lleva a señalar que en A* existe un compromiso entre

- la calidad de la estimación;
- la rigidez - o su opuesta, la relajación - de la heurística y
- el trabajo por nodo.

A medida que la heurística se aproxima al coste real

1. se expanden menos nodos pero
2. cada nodo exige más trabajo para calcular su heurística.

5.8 Comparación de heurísticas.

1. Factor de ramificación efectivo (b^*):

- $N = \text{Número de nodos expandidos por A}^*$.
- $d = \text{Profundidad de la solución}$
- $b^* = \text{factor de ramificación de un árbol uniforme de profundidad } d, \text{ donde } N = \# \text{ nodos que se necesitan expandir para llegar a la solución óptima. Por la ec. (3.1.1.) tendremos que}$

$$b^* + b^{*2} + \dots + b^{*d} = b^* \frac{(b^*)^d - 1}{b^* - 1} = N$$

Ejemplo: $d=5, N=52 \rightarrow b^* = 1.92$. Promediar b^* para diferentes ejemplos del mismo problema. De manera ideal: $b^* \text{ lo más cercano posible a } 1$.

2. Con frecuencia es posible establecer una relación de dominancia entre heurísticas. Decimos que la heurística a domina a b cuando

$$\forall n, h_a(n) \geq h_b(n) \quad (5.8.1)$$

3. Si usamos búsqueda A^* y h_a domina a h_b

- h_a nunca expande más nodos que h_b
- Generalmente, $b_{a*} \leq b_{b*}$

4. Usar h_a si h_a domina a h_b y los costes computacionales de computar ambas heurísticas son comparables.

5. Hay casos en que (5.8.1) solo se verifica en algunos nodos del árbol pero podemos construir una heurística válida tomando la que tiene mayor valor en cada nodo. Si hay disponibles varias funciones heurísticas admisibles, entonces $\max \{h_1(n), h_2(n), \dots, h_k(n)\}$ es admisible.

5.9 Estados repetidos: búsqueda en grafos.

Hasta este punto hemos ignorado una de las complicaciones más importantes del proceso de búsqueda: la posibilidad de perder tiempo expandiendo estados que ya han sido visitados y expandidos. Para algunos problemas, esta posibilidad o aparece; el espacio de estados es un árbol y hay sólo un camino a cada estado.

Para algunos problemas la repetición es inevitable como en el siguiente ejemplo:

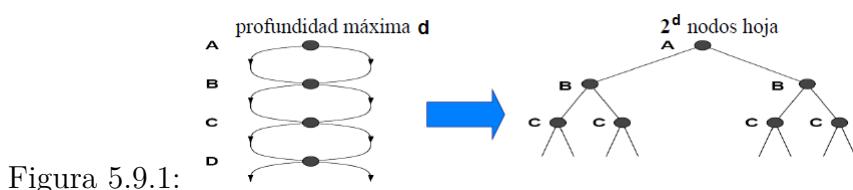


Figura 5.9.1:

Es claro que eliminando estados repetidos reduciríamos el árbol de 2^d a $d+1$ hojas. La detección de estados repetidos por lo general significa la comparación del nodo a expandir con aquellos que han sido ya expandidos; si se encuentra un emparejamiento, entonces el algoritmo ha descubierto dos caminos al mismo estado y puede desechar uno de ellos.

Si nos fijamos en la Fig. 4.1.1. vemos que los dos subárboles que arrancan en e se repiten aun cuando el nodo e está a diferente profundidad. El árbol más profundo no nos dirá nada que ya no sepamos.

Para la búsqueda *DFS*, los únicos nodos en memoria son aquellos del camino desde la raíz hasta el nodo actual. La comparación de estos nodos permite al algoritmo descubrir los caminos que forman ciclos y que pueden eliminarse inmediatamente.

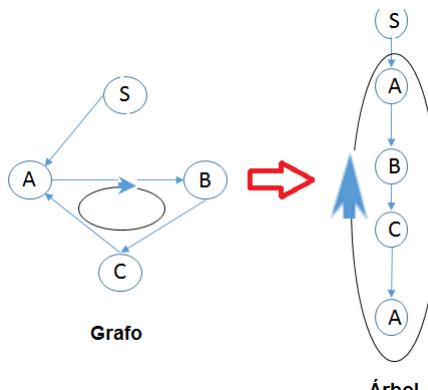


Figura 5.9.2:

Esto, sin embargo, solo sirve para evitar que los ciclos formen árboles de búsqueda infinitos, pero esto no evita problemas como el de la figura 4.4.1. en que se generan caminos exponencialmente. Para evitar estos casos es guardar más nodos en la memoria: *los algoritmos que olvidan su historia, están condenados a repetirla*. Podemos crear una **lista-cerrada**³ que contenga los nodos ya expandidos. Si el nodo actual se encuentra en *lista-cerrada* se elimina en lugar de expandirlo. Así tendremos un nuevo algoritmo denominado **BÚSQUEDA-GRAFOS** que es mucho más eficaz que BÚSQUEDA-ÁRBOLES.

La idea del algoritmo es fácil: «*no expandir nunca dos veces el mismo estado*».

¿Es este algoritmo completo? ¿Es posible que perdamos la solución por no expandir ciertos nodos en el árbol? Esto no puede suceder porque no expandimos lo que ya se expandió en otro nodo.

Si examinamos la optimalidad de este algoritmo, es difícil de dilucidar. Cuando el algoritmo desecha un camino, siempre desecha el recién descubierto que puede ser el de menor coste. Sin embargo, puede demostrarse que esto no puede suceder cuando se utiliza la *UCS* o la *BFS* con costes constantes que no solo son estrategias óptimas en árboles, sino también en grafos. La búsqueda con profundidad iterativa, por otra parte, utiliza la expansión primero en profundidad y fácilmente puede seguir un camino subóptimo, lo que exige comprobaciones adicionales.

En la figura se muestra el algoritmo general de búsqueda en grafos:

³A veces se denomina a la frontera: *lista-abierta*. No obstante, es preferible llamarlo *conjunto cerrado* ya que si se trata como una lista los resultados son peores.

```

problema = {nodo-raíz, expandir, test-objetivo}; estrategia
function búsqueda-en-grafo (problema, estrategia)
/* devuelve solución o fallo
lista-abierta contiene los nodos de la frontera de árbol-de-búsqueda*/
Iniciarizar árbol-de-búsqueda con nodo-raíz
Iniciarizar lista-abierta con nodo-raíz
Iniciarizar lista-cerrada a una lista vacía
Loop do
    If (lista-abierta está vacía) then return fallo
    Elegir dentro de lista-abierta, de acuerdo a la estrategia, un nodo a expandir.
    If (nodo satisface test-objetivo)
        then return solución (camino desde nodo-raíz a nodo)
        eliminar nodo de lista-abierta
        If (nodo no está en lista-cerrada)
            then añadir nodo a lista-cerrada
            expandir nodo
            añadir nodos hijo a lista-abierta
end

```

Algoritmo general de búsqueda-en-grafo

Figura 5.9.3:

5.10 Consistencia.

El gráfico muestra un caso en que la búsqueda en grafos no funciona a pesar de ser la heurística admisible.

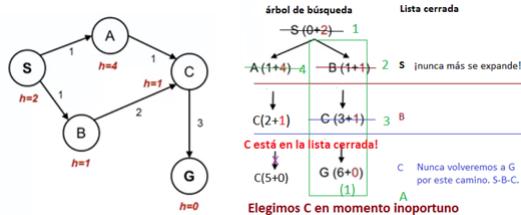


Figura 5.10.1:

Esto nos indica que para grafos precisamos algo más que la admisibilidad: la **consistencia**.

Recordemos que la admisibilidad consistía en que *el coste estimado no superase el coste real al objetivo más próximo*. **La consistencia**, por su parte, requiere que la heurística sea menor o igual que el coste real pero *para cada arco*, no para el camino hasta el objetivo.

Pero la heurística, hasta aquí, no costea arcos, sino caminos. Tomaremos el *coste heurístico* del arco A, C como la diferencia

$$\text{coste}(A, C) = h(A) - h(C) \quad (5.10.1)$$

mientras que $c(A, C)$ es el coste real. La condición de *consistencia* es, por tanto

$$\text{coste}(A, C) = h(A) - h(C) \leq c(A, C) \quad (5.10.2)$$

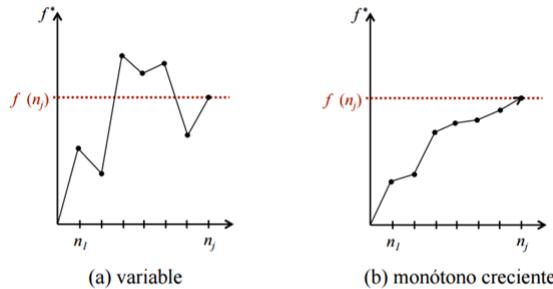


Figura 5.10.2:

Podemos ver que en la Fig.5.9.1. la heurística es inconsistente ya que $h(A) = 4$, $h(C) = 1 \Rightarrow h(A) - h(C) = 3 > c(A, C) = 1$

Valores de f en árboles de búsqueda A^*

Posibles “tipos” de variación de los valores de f a lo largo de un camino desde la raíz hasta un nodo n_j



Funciones heurísticas consistentes

Definición:

Si para todo nodo n_i y todo sucesor n_j de n_i se cumple que

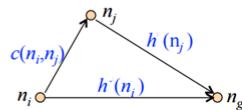
$$h(n_i) \leq h(n_j) + c(n_i, n_j)$$

 entonces h es consistente

Interpretación intuitiva:

- h es consistente si cumple la desigualdad triangular

Figura 5.10.3:



5.10.1 Consecuencias de la consistencia.

- El valor de $f(n) = g(n) + h(n)$ a lo largo de un camino no decrece nunca ($f(n)$ es monótona no decreciente). En efecto de 5.9.2. para un nodo n y cualquier sucesor del mismo n_i se verifica:

$$\left. \begin{array}{l} h(n) \leq h(n_i) + c(n, n_i) \\ g(n) \geq g(n_i) \end{array} \right\} \Rightarrow f(n) \geq f(n_i) \quad (5.10.3)$$

- Si una heurística es consistente, es admisible.

Sea $h^*(n)$ el coste del camino más barato hasta el objetivo. Demostremos por inducción que $h(n) \leq h^*(n)$.

Caso base: si hay 0 pasos desde n , n es el objetivo y $h(n) = 0 \leq h^*(n)$.

Sea que n está $am + 1$ pasos del objetivo, por tanto $h(n_{m+1}) \leq h(n_m) + c(n_{m+1}, m_m) \leq h^*(n_m) + c(n_{m+1}, m_m) \leq h^*(n_{m+1})$ c.q.d.

- La *monotonía* $def(n)$ garantiza la *optimalidad* de A^* .

Esbozaremos su demostración seguidamente.

- Sea que O es una solución óptima cuyo coste del camino es $f^*(O) = g(O)$ y $h(O) = 0$.
- Sea SO un estado objetivo subóptimo, esto es tal que $g(SO) > f^*(O)$
- A^* termina su búsqueda con SO .

Sea n un nodo hoja en el camino óptimo a O , entonces se verifica que

- $f^* \geq f(n)$ por la admisibilidad de h
- $f(n) \geq f(SO)$ n no fue elegido para expansión.
- Por tanto $f^* \geq f(n) \geq f(SO)$
- Pero $f(SO) = g(SO)$ para ser óptimo $h(SO) = 0$ y se cumpliría $f^* \geq g(SO)$ lo que es *contradicitorio con el carácter óptimo de la solución O*

El carácter monótono de $f(n)$ significa que el primer nodo objetivo expandido debe ser el correspondiente a la solución óptima.

- **Comportamiento de A^* con una heurística consistente**
 - 1 En **búsqueda en árbol** A^* expande los nodos en valores crecientes de f
 - 2 Para cada estado s , los nodos que alcanzan optimalidad en s se expanden antes que los que son subóptimos.
 - **Resultado:** la búsqueda en grafo es óptima

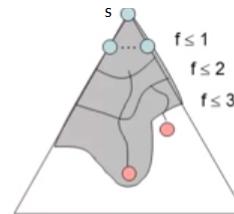


Figura 5.10.4:

- Si h es una heurística monótona, la exploración se realiza en curvas de nivel con valores crecientes de $f(n)$.

En una búsqueda de coste uniforme [$h(n) = 0$], las curvas de nivel son “concéntricas” alrededor del estado de partida.

En heurísticas mejores estas curvas forman bandas que se extienden hacia el estado objetivo

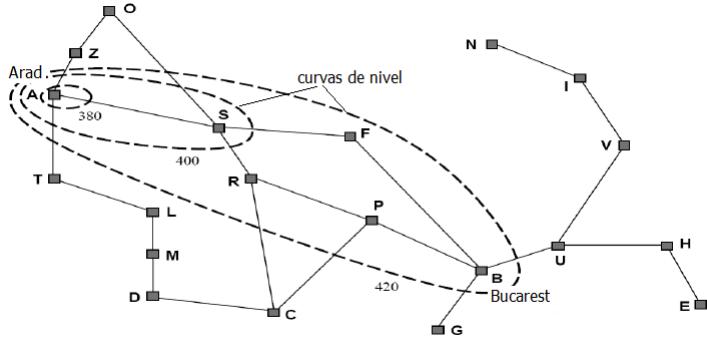


Figura 5.10.5:

5.11 Resumen de la búsqueda A*.

Optimalidad.

1. La *búsqueda en árbol* es óptima si la heurística es *admissible* (UCS es caso particular con $h=0$).
2. La *búsqueda en grafo* es óptima si la heurística es *consistente* (UCS es caso particular con $h=0$).
3. La consistencia *implica* admisibilidad.
4. En general las heurísticas «naturales» admisible procedentes de la relajación de problemas tienden a ser consistentes.

5.12 Búsqueda A* paso a paso.

5.12.1 Búsqueda en árbol.

Sea el ejemplo de *búsqueda en árbol* siguiente (Fig. 5.12.1):

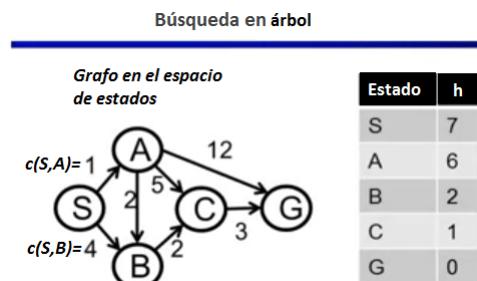


Figura 5.12.1:

5.12.1.1 Visualización en árbol de búsqueda.

Si visualizamos la búsqueda en el árbol iremos pasando por las siguientes etapas (Fig. 5.12.2):

- El estado inicial S tiene un coste estimado $f(S) = g(S) + h(S) = 0 + 7 = 7$.
- A^* considerará todos los estados del árbol aún no expandidos - S - únicamente en esta fase y se expande a los nodos A y B y dado que $f(A) = 7 > f(B) = 6$ y, por tanto, A^* considerará B para expansión.
- B tiene como sucesor a C con $f(C) = 7$. Ahora tenemos dos candidatos a expandir A y C con el mismo valor $f = 7$ - , de modo que expandiremos A por el criterio de ordenación alfabética.
- A tiene tres sucesores B, C y G y, siendo el menor valor $f(B)=5$ es este el que consideraremos en la siguiente etapa. Es importante destacar que el hecho de haber alcanzado G no basta para terminar la búsqueda, para ello *hemos de expandir el nodo objetivo*.
- Ahora expandiremos $[S, A, B, C]$ que solo tiene un sucesor G. Nos encontramos con un empate entre $([S, A, C], 7)$ y $([S, B, C], 7)$ que se rompe alfabéticamente en favor del primero que tiene un sucesor G.
- Todavía no alcanzamos el objetivo y ahora expandimos $([S, B, C], 7)$ obteniendo $([S, B, C, G], 9)$.
- Finalmente nos quedan tres nodos que acaban en G y puesto que hemos de expandir $([S, A, B, C, G], 8)$ - el de menor coste - es este el camino solución.

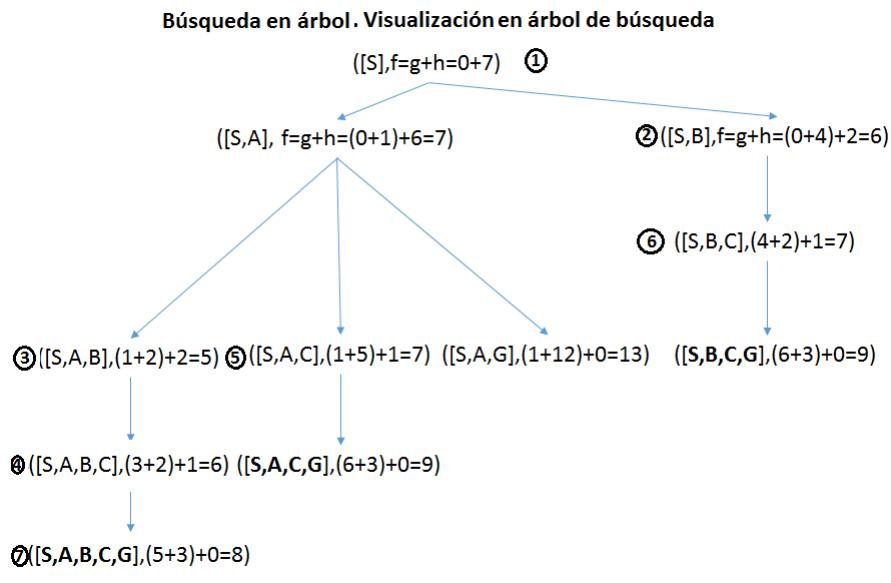


Figura 5.12.2:

Por tanto la solución que nos devuelve el problema es $([S, A, B, C, G], 8)$ pero ¿es óptima?

Lo será si la heurística h es admisible (árbol).

Admisible es ya que $h(n)$ es, para todo nodo n, inferior al coste real de alcanzar el estado objetivo como puede comprobarse.

5.12.1.2. Visualización de la frontera (cola con prioridades).

Podemos visualizar el proceso de búsqueda siguiendo la evolución de la frontera que es una cola de prioridades, lo que se traduce en lo siguiente:

1. **Frontera:** $([S], 0+7)$

Sale: $([S], 0+7)$ el nodo con *mayor prioridad*.

2. **Frontera:** $([S, A], (0+1)+6), ([S, B], (0+4)+2)$

Sale: $([S, B], 4+2)$

3. **Frontera:** $([S, A], (0+1)+6), ([S, B, C], (4+2)+1)$

Sale: $([S, A], 1+6)$

4. **Frontera:** $([S, A, B], (1+2)+2), ([S, A, C], (1+5)+1), ([S, B, C], (4+2)+1), ([S, A, G], (1+12)+0)$

Sale: $([S, A, B], 3+2)$

5. **Frontera:** $([S, A, C], (1+5)+1), ([S, B, C], (4+2)+1), ([S, A, B, C], (3+2)+1), ([S, A, G], (1+12)+0)$

Sale: $([S, A, C], 6+1)$

6. **Frontera:** $([S, B, C], (4+2)+1), ([S, A, B, C], (3+2)+1), ([S, A, C, G], 6+3, 0), ([S, A, G], (1+12)+0)$

Sale: $([S, B, C], 6+1)$

7. **Frontera:** $([S, A, B, C], (3+2)+1), ([S, A, C, G], 6+3, 0), ([S, B, C, G], 6+3, 0), ([S, A, G], (1+12)+0)$

Sale: $([S, A, B, C], 5+1)$

8. **Frontera:** $([S, B, C], (4+2)+1), ([S, A, B, C, G], (5+3)+0), ([S, A, G], (1+12)+0), ([S, A, C, G], (6+3)+0)$

Sale: $([S, B, C], 6+1)$

9. **Frontera:** $([S, B, C, G], (6+3)+0), ([S, A, B, C, G], (5+3)+0), ([S, A, C, G], (6+3), 0), ([S, A, G], (1+12)+0)$

Sale: $([S, B, C], 1+6)$

10. **Frontera:** $([S, A, B, C, G], (5+3)+0), ([S, A, C, G], (6+3), 0), ([S, B, C, G], (6+3)+0), ([S, A, G], (1+12)+0)$

Sale: $([S, A, B, C, G], 8)$

5.12.2 Búsqueda en grafo.

La diferencia con la búsqueda en árbol es que cuando vamos a expandir un nodo *comprobamos si ya hemos expandido un nodo que termine en el mismo estado del nodo que estamos a punto de expandir* y, si es así, no expandimos dicho nodo sino que simplemente lo sacamos de la cola con prioridades.

Visualizaremos esto en el árbol de búsqueda (Fig. 5.12.3).

Comenzaremos igual que en la búsqueda en árbol.

1. El estado inicial S tiene un coste estimado $f(S) = g(S) + h(S) = 0 + 7 = 7$.

2. A* considera el estados del árbol aún no expandidos - S - y lo expande los nodos A y B, Ahora, disponemos de una LISTA-CERRADA que lleva la cuenta de los estados que se han expandido en el pasado, de modo que LISTA-CERRADA= S . Dado que $f(A) = 7 > f(B) = 6$ A* considerará B para expansión.
3. B tiene como sucesor a C con $f(C) = 7$. LISTA-CERRADA= S, B . Ahora tenemos dos candidatos a expandir A y C con el mismo valor f - 7 - , de modo que expandiremos A por el criterio de ordenación alfabética.
4. A tiene tres sucesores B, C y G y, siendo el menor valor $f(B)=5$ es este el que consideraremos en la siguiente etapa. Hemos expandido A y LISTA-CERRADA= S, B, A .
5. Ahora expandiríamos $[S, A, B]$ pero $B \in$ LISTA-CERRADA y no lo expandimos. Nos encontramos con un empate entre $([S, A, C], 7)$ y $([S, B, C], 7)$ que se rompe alfabéticamente en favor del primero que tiene un sucesor G $\rightarrow ([S, A, C, G], 9)$.
6. Hemos pues expandido $([S, A, C], 7)$ y LISTA-CERRADA= S, B, A, C .
7. Finalmente nos quedan tres nodos $([S, B, C], (4+2)+1)$, $([S, A, G], (1+12)+0)$ y $([S, A, C, G], (6+3)+0)$ y habríamos de expandir el de menor coste $([S, B, C], (4+2)+1)$ pero $C \in$ LISTA-CERRADA y nos movemos al más cercano $([S, A, C, G], 9+0)$ que, al expandirlo, nos indica que es la solución.

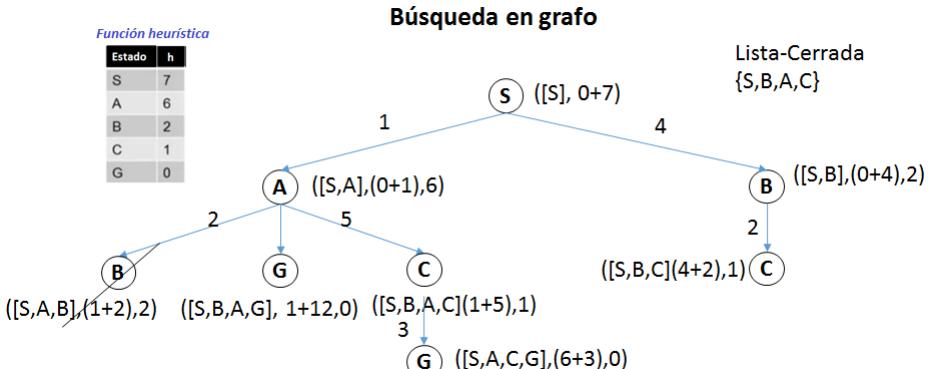


Figura 5.12.3:

¿Es óptima esta solución? Recordemos que la búsqueda en árbol anterior daba como solución $([S, A, B, C, G], 8)$, cuyo coste estimado es inferior al de $([S, A, C, G], 9)$. La explicación de esta diferencia es que *la heurística no es consistente* motivo por el cual no se exploró la solución óptima $([S, A, B, C, G], 8)$.

La condición de consistencia es

$$\forall s, s' \quad h(s) - h(s') \leq c(s, s')$$

Pero para $s=A$ y $s'=B$ tenemos que:

$$\begin{aligned} h(A) - h(B) &= 6 - 2 = 4 \\ c(A, B) &= 2 \end{aligned}$$

y, por tanto, no es consistente.

Ejercicios.

1. En el grafo siguiente responder a las preguntas:

- ¿Es la heurística consistente?
- ¿Qué solución devuelve A*?

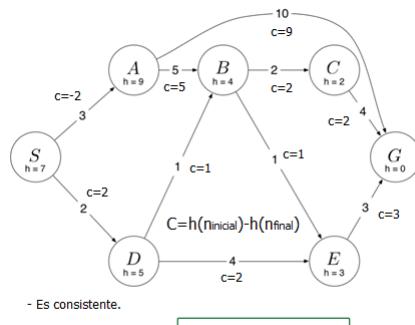


Figura 5.12.4:

TEMA 6

BÚSQUEDA CON ADVERSARIOS.

6.1 Búsqueda con adversarios y juegos.

Ahora nos interesamos por una clase de búsqueda que incluye la existencia de algún tipo de adversario del agente en el mundo.

Nos centraremos en los entornos *multiagente*, en los cuales cualquier agente tendrá que considerar las acciones de otros agentes y cómo afectan a su propio bienestar.

La imprevisibilidad de estos otros agentes puede introducir muchas posibles **contingencias** en el proceso de resolución de problemas.

Hay que distinguir entre entornos *cooperativos* y *competitivos*. Los entornos competitivos, en los cuales los objetivos del agente están en conflicto, dan ocasión a problemas de *búsqueda con adversarios*, a menudo conocidos como **juegos**.

Un ejemplo de aplicación de este tipo de búsquedas lo constituyen los juegos tales como las damas, el ajedrez o el Go.

Si revisamos el estado del arte en estos juegos nos encontramos con lo siguiente:

1. El juego de las damas fue totalmente resuelto en 2007.
2. En el ajedrez no se ha llegado tan lejos, pero sí se han conseguido sistemas capaces de derrotar a los humanos más expertos.
3. El juego del Go es de gran complejidad y su factor de ramificación b es de 300. En estos momentos los humanos vencen a las máquinas, pero se están haciendo progresos mediante el uso de herramientas tales como los métodos Monte Carlo de expansión.

Por tanto, cuando *los agentes tienen objetivos diferentes* nos encontramos ante una **búsqueda con adversarios** con las siguientes características:

- Idealización en la que los jugadores con objetivos divergentes *alternan turnos* en los que realizan acciones.
- Los agentes sólo pueden realizar *movimientos legales* (tal como definen las reglas del juego).
- Los movimientos se eligen de acuerdo a una *estrategia* que especifica un movimiento por cada posible movimiento del oponente.
- El juego termina cuando uno de los agentes *alcanza su objetivo* (medido por una *función de utilidad*).

6.2 Juegos con dos agentes.

Supongamos que sobre un tablero cuadriculado existen dos robots *Black* y *White* que pueden moverse a las casillas adyacentes a su posición actual. Los agentes se mueven alternativamente con *White* el primero. El objetivo de *White* es situarse en la misma casilla que *Black* y el de *Black* es evitarlo.

1. *White* hace un movimiento,
2. *White* observa el movimiento que hace *Black* y
3. *White* repite el proceso de planeación

En un ciclo *observa/planea/actúa*.

White planea sus acciones construyendo un árbol de movimientos posibles como el de la figura (parcial):

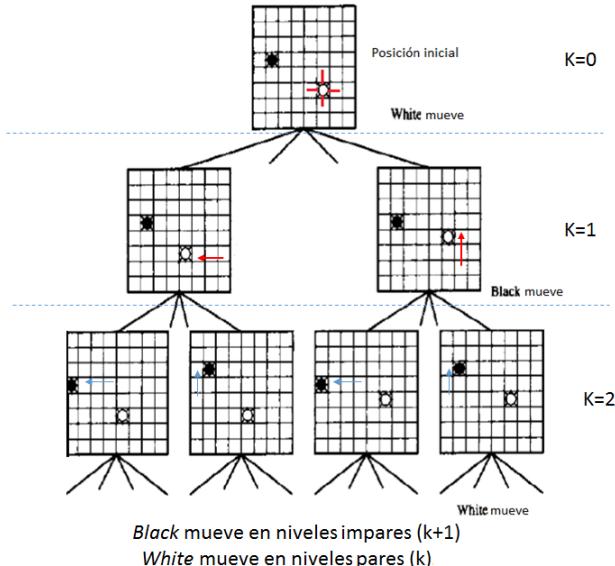


Figura 6.2.1:

Para elegir su primer movimiento, *White* analiza el árbol teniendo en cuenta que *Black* actuará para evitar que *White* alcance su objetivo.

En algunas situaciones como esta es posible que el agente encuentre un movimiento que le permita alcanzar su objetivo con independencia de lo que el otro haga en cualquiera de sus movimientos. Lo más común, sin embargo, es que ningún agente sea capaz de alcanzar su objetivo debido a que sea computacionalmente inviable desarrollar completamente los árboles de decisión por lo que lo habitual es utilizar métodos con un *horizonte* (ámbito de búsqueda) *limitado*. Podemos usar métodos BFS, DFS o heurísticos pero hemos de fijar una *condición de terminación* que puede basarse en tiempo de ejecución, capacidad de almacenamiento o profundidad del nodo más profundo, p.e.

Una vez terminada la búsqueda podemos seleccionar el primer movimiento que se estima mejor lo que se hace mediante una *función de evaluación* aplicada en los nodos hoja del árbol.

6.3 Tipos de juegos.

Los juegos pueden clasificarse con diferentes **criterios**:

- *Número de jugadores*

- Dos jugadores en el que cada uno pretende ganar al otro (ej. backgammon, ajedrez, damas)
- Multijugador donde pueden darse estrategias mixtas (competitivas / cooperativas, ej. alianzas temporales).

- Propiedades de la *función de utilidad*:

- *Suma cero*: Los agentes tienen utilidades contrarias: la misma función de utilidad que uno quiere maximizar y el otro minimizar. *La suma de utilidades de los agentes es cero*— $U_1 + U_2 = 0$ —, independientemente del resultado del juego. Ej. ajedrez, damas.

- *Suma constante*: La suma de utilidades de los agentes es constante, independientemente del resultado del juego. Es equivalente a juegos de suma cero ya que $U_1 + (U_2 - C) = 0$.

- *Suma variable*: Las utilidades de los agentes son independientes. No son de suma cero. Pueden tener estrategias óptimas complejas, que pueden implicar tanto *colaboración* (ej. monopoly, backgammon), como *indiferencia* o *competencia*, según la fase del juego.



- **Juegos de suma cero:**
Los agentes tienen utilidades **opuestas** (lo que uno gana lo pierde el otro)
Competencia pura



- **Juegos en general:**
Los agentes tienen utilidades **independientes**
Son posibles las situaciones de cooperación, indiferencia, competición y otras

Figura 6.3.1:

- *Información* de la que disponen los jugadores:

- Información *perfecta* (ej. ajedrez, damas, go)
- Información *parcial* (ej. casi todos los juegos de cartas)

- Elementos de *azar*:

- *Deterministas* (ej. ajedrez, damas, go)
- *Estocásticos* (ej. parchís, backgammon)
- *Tiempo* ilimitado / limitado (ej. ajedrez con reloj).
- *Movimientos* ilimitados / limitados (ej. ajedrez con máximo de 40 movimientos).

El juego de los dos robots en el tablero es un juego con *dos agentes, información perfecta y suma cero*.

6.4 Decisiones óptimas en juegos. Búsqueda con adversarios.

Lo que queremos es desarrollar algoritmos para calcular una **estrategia** que recomiende un movimiento en cada estado.

Este problema es distinto al de búsqueda expuesto anteriormente en que desarrollábamos un *plan* -secuencia de acciones a ejecutar - que garantizaban el éxito. Ahora, esto no es así puesto que no controlamos a nuestro adversario. Lo que necesitamos ahora es una función que nos diga en cada estado lo que debemos hacer, esto es, una **estrategia contingente**.

Plantearemos un nuevo problema de búsqueda con los siguientes elementos:

- **Estados:** \mathbf{S} (inicial s_0).
- **Jugadores:** $\mathbf{P}=\{1, 2, \dots, N\}$.
- **Acciones:** \mathbf{A} que pueden depender del estado y/o del jugador.
- **Función sucesor:** **mover** (*estado-actual \rightarrow Estado-sucesor*): $\mathbf{S} \times \mathbf{A} \rightarrow \mathbf{S}$
- **Test terminal:** función booleana que determina si el estado del juego es *terminal* (es decir, el juego ha concluido): $\mathbf{S} \rightarrow \{\text{t}, \text{f}\}$
- **Utilidad** (función objetivo o de pago): Valoración numérica de los estados terminales. $\mathbf{S} \times \mathbf{P} \rightarrow \mathbf{R}$
- **Árbol del juego:** El estado inicial y los movimientos legales alternados de cada parte definen el árbol del juego.

Su *solución* consiste en una **estrategia que relaciona estados con acciones**: $\mathbf{S} \rightarrow \mathbf{A}$.

Aplicaremos esta definición al siguiente juego sencillo:

1. Dos jugadores (*MAX* y *MIN*) realizan movimientos alternados hasta que uno de ellos gana (el otro pierde) o hay un empate.
2. Cada jugador tiene un modelo perfecto del entorno determinista y de los efectos que producen los movimientos legales.
3. Puede haber limitaciones computacionales / temporales a los movimientos de los agentes.

Por tanto se trata de un juego:

- Con *dos agentes*
- *Información perfecta*
- *Determinista*
- *De suma cero*

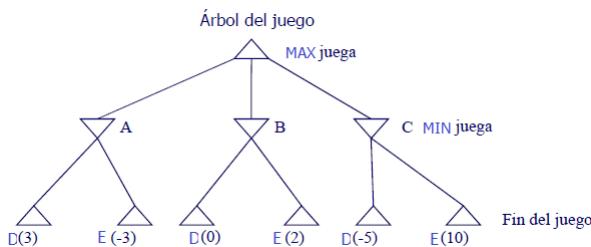
Una posible representación de este tipo de juegos incluye:

1. La *matriz de balance final*: Valor de la función de utilidad para cada jugador, dadas las acciones de los otros jugadores que representa el resultado. En este caso las sumas que MIN (positivas en la matriz) paga a MAX.

Es decir, si MAX juega A y MIN responde con B, MAX pierde 3 que gana MIN . En este caso, siendo de *suma cero*, los valores positivos son ganancias para MAX y pérdidas para MIN y viceversa.

		MIN				
		D	E			
MAX		A	ganancia de MAX=3	ganancia de MAX=-3	MIN	
		B	ganancia de MAX=0	ganancia de MAX=2		
MAX		C	ganancia de MAX=-5	ganancia de MAX=10	MIN	
		A	ganancia de MIN=-3	ganancia de MIN=3		
		B	ganancia de MIN=0	ganancia de MIN=-2		
		C	ganancia de MIN=5	ganancia de MIN=-10		

2. El *árbol del juego*, en este caso con *dos niveles*:



A cada uno de los estados terminales asociamos un valor del estado o máxima utilidad terminal obtenible.

Sea el caso de Pac-Man que quiere llegar a comerse un punto moviéndose a *derecha* o a *izquierda*.

Consideremos inicialmente este caso como si hubiera **un solo agente** que trata de **maximizar** el *valor terminal* de la búsqueda. En este caso, las ramas de la figura siguiente terminan cuando se alcance el fin del juego (Pac-Man se come el punto) o no terminan nunca. Asociaremos valores a cada uno de los estados terminales.

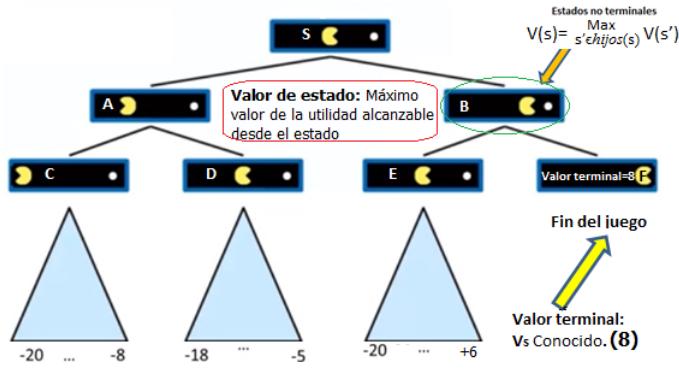


Figura 6.4.1:

Para estos estados terminales, el valor del juego es conocido. Pero ¿qué sucede con los demás estados? Desde el estado B puedo alcanzar tanto el estado F como todo lo que se encuentre en algún camino por debajo de E, es decir desde el nodo B puedo alcanzar *cualquier valor terminal que se encuentre por debajo de E* como 6 o -20. No obstante, si quiero maximizar mi utilidad entre 6¹ y 8 elegiré siempre 8. El estado **F** tendrá un valor: será 8 el valor de F.

Por tanto puedo escribir que en el caso de **un solo agente** el *valor de un estado s* (el B, el inicial,...) será el *valor máximo que tengan los estados sucesores de s -{s'}*:

$$V_s = \max_{s' \in \text{sucesores}(s)} V(s') \quad (6.4.1)$$

Pasemos al caso en que existen **dos adversarios** (jugadores): *Pac-Man* y un *fantasma*. Pac-Man puede moverse a izquierda y a derecha, como antes, pero ahora el fantasma también puede moverse en dichas direcciones. Sigue existiendo un árbol de estados posibles futuros, solo que ahora también el fantasma tiene elecciones que hacer. Las características de ramificación son las mismas que las del agente único. La diferencia con aquel es **quién controla lo que ocurre en cada estado (nodo)**, Ambos adversarios están de acuerdo respecto a lo que *puede pasar*, pero están en desacuerdo acerca de lo que *quieren que pase*. Es imposible que uno de los agentes elija y alcance el valor terminal 8 porque su adversario puede oponerse.

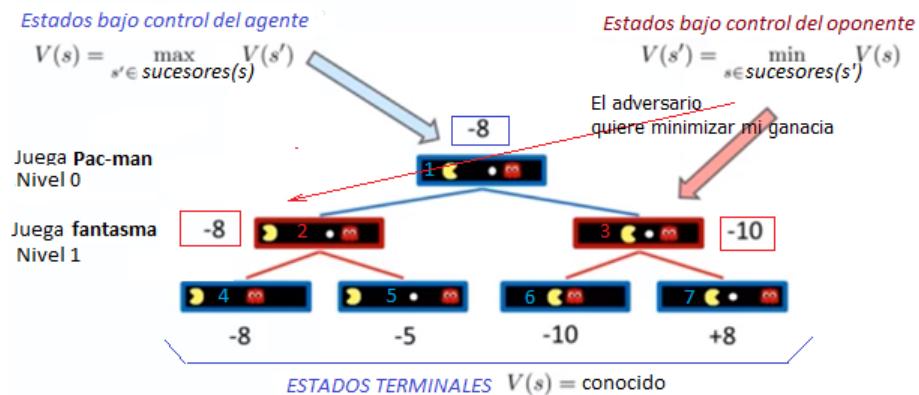


Figura 6.4.2:

¹Máximo valor por debajo de E.

En este caso, el valor de estado sigue siendo el **mejor resultado obtenible frente a un adversario óptimo**, pero ahora incorporamos la idea de *razonamiento del adversario*.

De nuevo aquí están dados los valores terminales del juego. Si suponemos que el juego termina tras una acción de Pac-Man y una respuesta del fantasma, reducimos el árbol al de la Fig. 6.4.2 en el que puede verse cómo existen nodos bajo control de nuestro adversario (2 y 3).

Si p.e. alcanzamos el estado 3 bajo control del oponente, el fantasma se moverá a la izquierda y no podremos visitar el nodo 7 cuyo $V(7) = 8$, sino que forzosamente iremos al nodo 6 cuyo **valor (minimax)** es **-10** y el valor de 3 será -10.

Los cálculos en estos nodos bajo control adversario - 2 y 3 -son fáciles para el agente *fantasma* que quiere **minimizar** la ganancia de *Pac-Man*:

$$\forall s' \text{ bajo control adversario } V(s') = \min_{s' \in \text{sucesores}(s')} \begin{cases} V_4 = -8 \\ V_5 = -5 \end{cases} \Rightarrow V(s' = 2) = -8 \quad (6.4.2)$$

Para los que están bajo control de *Pac-Man* - 1-, en cambio, el agente *Pac-Man* querrá **maximizar** sus ganancias:

$$\forall s \text{ bajo control del agente } V(s) = \max_{s' \in \text{sucesores}(s)} \begin{cases} V_2 = -8 \\ V_3 = -10 \end{cases} \Rightarrow V(s = 1) = -8 \quad (6.4.3)$$

Esto implica que las únicas acciones que llevamos a cabo *para encontrar el valor de un estado* consisten en calcular *los valores máximos y mínimos* de otros estados que se encuentran *por debajo*, de manera que el valor del nodo raíz será siempre el valor de un (o conjunto) nodo terminal del árbol del juego.

6.5 Algoritmo Minimax.

Examinemos un juego que satisface las anteriores condiciones conocido como es el tres en raya:

- Jugadores: 2 (*MAX* y *MIN*).
- Estados: se definen por la matriz que contiene los *X* y *O* que hay en el tablero y sus posiciones:

$$\begin{bmatrix} X & \emptyset & \emptyset \\ \emptyset & O & \emptyset \\ \emptyset & \emptyset & \emptyset \end{bmatrix}, \begin{bmatrix} X & \emptyset & X \\ \emptyset & O & \emptyset \\ \emptyset & \emptyset & \emptyset \end{bmatrix}, \begin{bmatrix} X & O & X \\ \emptyset & O & \emptyset \\ \emptyset & \emptyset & \emptyset \end{bmatrix}, \dots$$

- Estado inicial: Tablero 3x3 vacío:

$$\begin{bmatrix} \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset \end{bmatrix}$$

- Movimientos: Poner una ficha (*MAX*: \times , *MIN*: \circ) en una de las casillas vacías.

- Test terminal: 3 fichas del mismo jugador están alineadas.
- Utilidades terminales:

Utilidad terminal para	MAX	MIN
Gana MAX	+1	-1
Gana MIN	-1	+1
Empate	0	0

Es fácil ver que se trata de un juego

- con *dos agentes*;
- con *información perfecta*: cada jugador conoce perfectamente el estado del juego²;
- *determinista*: los cambios en el entorno son totalmente previsibles, conocidas las acciones³.
- De *suma-cero* (si ganar vale 1, lo que gana MAX lo pierde MIN).

A este tipo de juegos es aplicable el método MINIMAX, adoptando el convenio de en las posiciones -s- favorables a MAX: $V(s) > 0$ y en las favorables a MIN: $V(s) < 0$ ya que siendo de suma nula $V_{MAX} = -V_{MIN}$.

¿Cuáles serían las estrategias óptimas?

MAX mueve primero y alterna sus movimientos con MIN. En el árbol del juego, los nodos de profundidad par (impar) corresponden a MAX (MIN). Una *dupla de profundidad* k en el árbol del juego corresponde a los nodos del árbol del juego de profundidades $2k$ y $2k+1$.

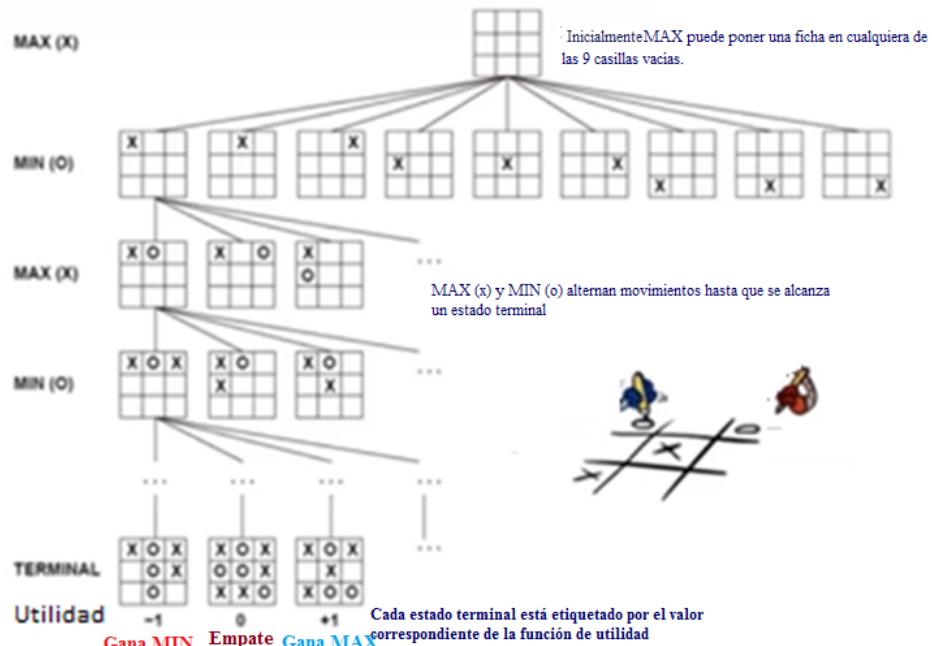


Figura 6.5.1:

²En el póquer hay *información imperfecta*: cada jugador ignora la mano que tiene cada uno de los demás jugadores.

³El póquer tiene *elementos estocásticos* ya que cada jugador ignora la mano que tienen los demás jugadores. Solo puede hacer conjeturas probabilísticas al respecto conociendo algunos datos como las cartas propias y los descartes de cada jugador.

Veamos una estrategia válida para que MAX seleccione el primer movimiento. Si el agente pudiera elegir entre los nodos hoja (terminales) es claro que elegiría el de utilidad máxima:M

$$\text{Máxima } \underset{n \in \text{Hojas}}{\text{Utilidad}(n)} \text{ si } n \text{ es un nodo terminal}$$

Dado que MAX puede elegir este nodo *si es su turno de jugar*, el valor *hacia atrás* de un nodo padre de hojas MIN es igual al máximo de la evaluación *estática* de los nodos hoja. Por otro lado, si MIN ha de elegir entre nodos hojas (terminales) elegiría los de menor valor para MAX. Como MIN puede elegir estos nodos *si es su turno*, al nodo MIN padre de los nodos MAX se le asigna un valor *hacia atrás* igual al mínimo de la evaluación estática de los nodos hoja. Después de asignar valores *hacia atrás* a todos los nodos hojas de esta dupla, retrocedemos otro nivel, suponiendo que MAX vuelve a elegir el nodo MIN sucesor de mayor valor, mientras que MIN lo hace con el de valor mínimo. Continuamos retrocediendo en el árbol hasta que, finalmente, se les asigna valor *hacia atrás* a los sucesores del nodo de comienzo y, por tanto, MAX elegirá el valor máximo.

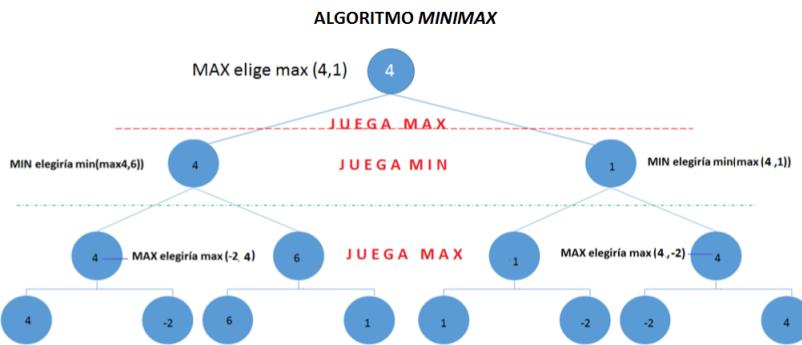


Figura 6.5.2:

- Descripción formal del juego:

1. Estado inicial: Configuración inicial del tablero + identidad del primer jugador.
2. Función sucesor: $Sucesores(n)$
3. Test terminal: $Terminal(n)$
4. Función de utilidad: $Utilidad(n)$, solo si n es un nodo terminal.

- Estrategia óptima para MAX: Estrategia con un resultado al menos tan bueno como cualquier otra estrategia, asumiendo que MIN es un oponente infalible.

- Estrategia minimax:

Usar el **valor minimax de un nodo** para guiar la búsqueda: **Utilidad de un nodo** (desde el punto de vista de MAX) asumiendo que **ambos jugadores juegan óptimamente** desde ahí hasta el fin del juego

$$MINIMAX(n) \left\{ \begin{array}{ll} Utilidad(n) & \text{si } n \text{ es un nodo terminal} \\ \max\{minimax(s); s \in \text{sucesores}(n)\} & \text{si } n \text{ es un nodo MAX} \\ \min\{minimax(s); s \in \text{sucesores}(n)\} & \text{si } n \text{ es un nodo MIN} \end{array} \right. .$$

(6.5.1)

La idea básica es elegir el movimiento con el máximo valor del MINIMAX: la mayor ganancia posible frente al mejor juego adversario. Se eligen movimientos que conduzcan a la victoria aunque MIN trate de bloquearlos.

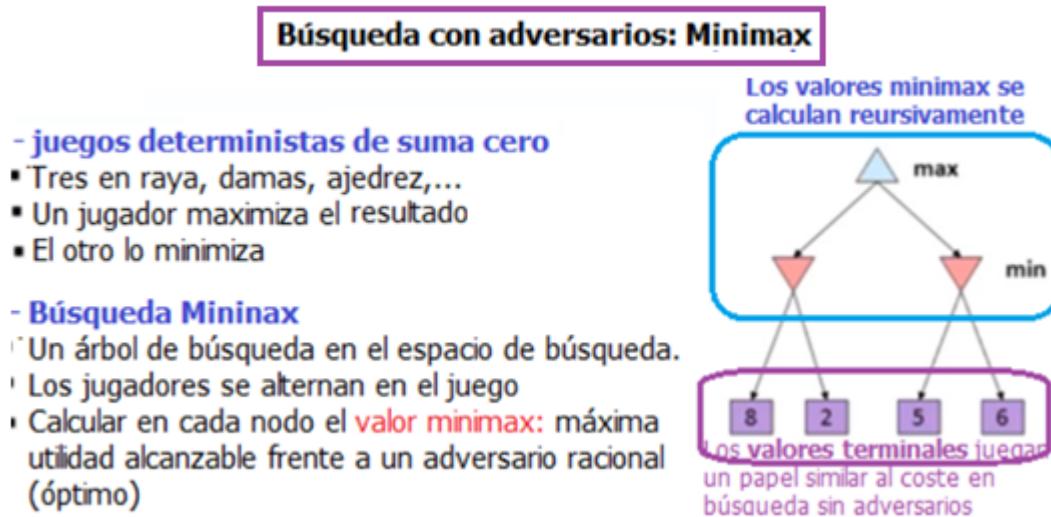


Figura 6.5.3:

El algoritmo MINIMAX es *recursivo*, corre de *abajo arriba* y propaga el árbol hacia el nodo raíz y calcularemos el mínimo de los hijos en un nodo min y su máximo en los nodos max. El valor calculado en el nodo raíz será el *valor MINIMAX tanto del nodo raíz como de otros nodos*.

Este algoritmo se expone en el cuadro siguiente:

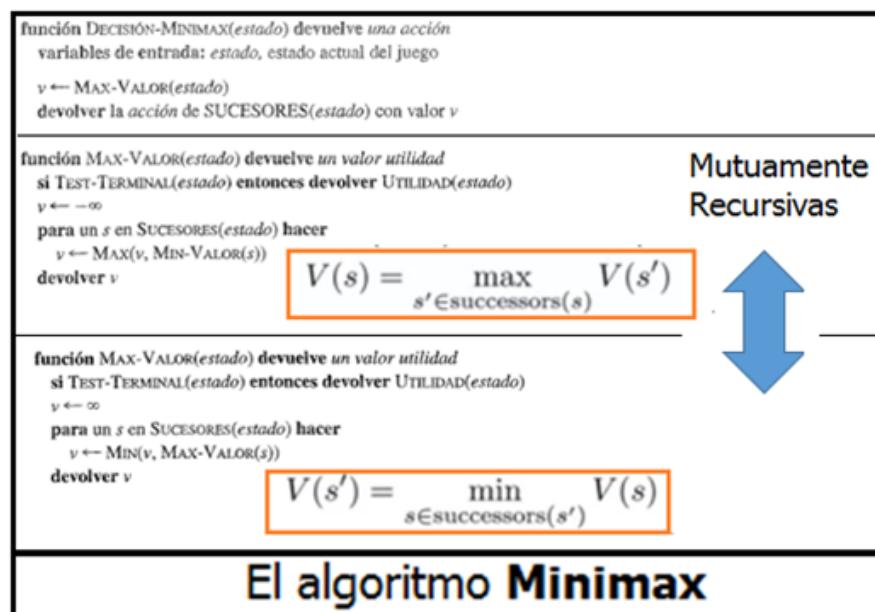


Figura 6.5.4:

Apliquemos el algoritmo al problema expuesto en la sección 3:

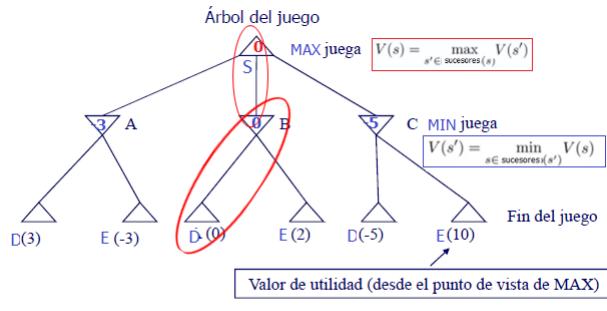


Figura 6.5.5:

Estos mismos cálculos pueden plantearse en la matriz de balance final:

		MIN		Mínimos
		D	E	
MAX	A	3	-3	-3
	B	0	2	0
	C	-5	10	-5

En cada una de las filas - que representan las jugadas de MAX - añado una columna de mínimos valores que obtendría el jugador, lo que refleja la estrategia que seguiría MIN. La estrategia a seguir por MAX, sería, evidentemente, jugar **B** que es el mayor de los mínimos.

Si lo miramos desde la perspectiva de MIN, la matriz de balance sería:

		MIN		Mínimos
		D	E	
MAX	A	-3	3	
	B	0	-2	
	C	5	-10	
	Mínimos	-3	-10	

y si MIN comenzara el juego, optaría por la estrategia D.

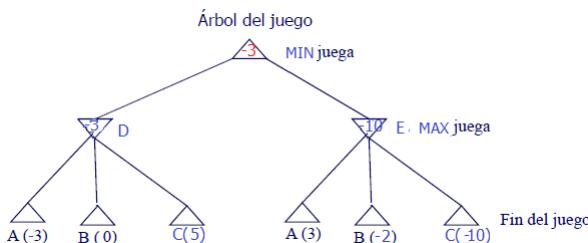


Figura 6.5.6:

6.5.1 Eficiencia del MINIMAX.

Es como el DFS: opera recursivamente en cada nodo y va a través de los hijos de izquierda a derecha, de modo que como en DFS:

- **Completo** sólo si el árbol del juego es finito (notar que puede haber estrategias óptimas finitas para árboles infinitos)
- **Óptimo** sólo si el oponente es óptimo nos devuelve el mayor valor del juego. Si el oponente es subóptimo, podemos usar sus debilidades para encontrar estrategias mejores *distintas de las generadas por el MINIMAX*.
- **Complejidad temporal:** $O(b^m)$
- **Complejidad espacial:** $O(b \cdot m)$

m = profundidad máxima del árbol del juego.

Ejemplo: ajedrez $b \approx 35$, $m \simeq 100$ siendo m en este caso el número de movimientos que hay en un juego. Como vemos el ajedrez es completamente imposible de explorar en su totalidad, pero ¿se necesita realmente explorar todo el árbol?

Ejemplo:

En el caso siguiente:

1. ¿Cuál es el valor MINIMAX del árbol? Respuesta: 8

2. ¿Qué acción tomará MAX según la estrategia MINIMAX? Respuesta: Izquierda de modo que deja a MIN solo las alternativas 10 y 8 (MIN elegirá 8, si es óptimo).

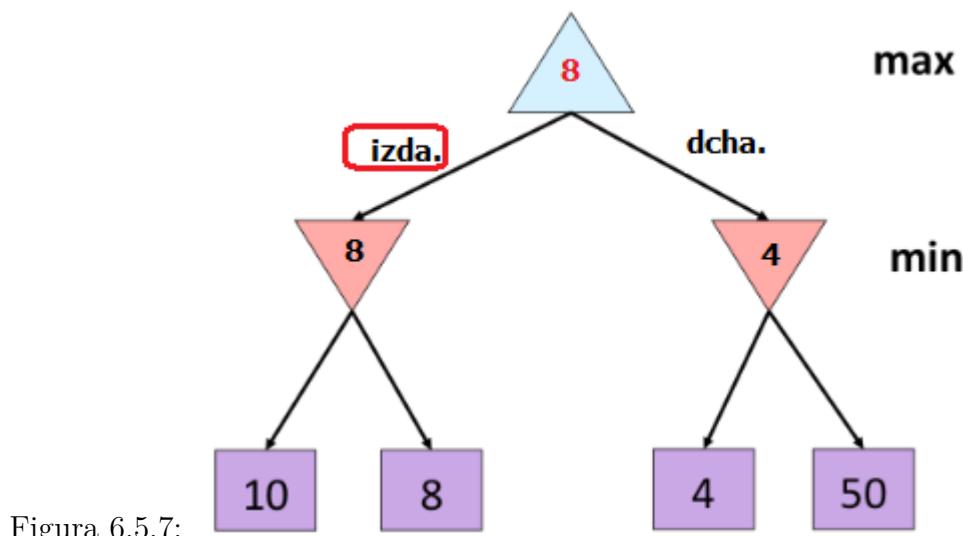


Figura 6.5.7:

6.6 Decisiones imperfectas. Funciones de evaluación.

¿Cómo podemos enfrentar problemas como el del ajedrez donde, probablemente, la complejidad temporal nos impide explorar todo el árbol?

La solución - bastante sencilla - consiste en hacer una **búsqueda limitada en profundidad**.

- Solo descenderemos por el árbol *mientras tengamos tiempo* y ahí truncaremos el árbol de búsqueda.
- Sustituiremos el valor terminal, que no alcanzaremos, por una **función de evaluación** de estados **no terminales**.

y **aplicaremos el minimax como si estos estados intermedios fueran terminales**.

Si p.e. en un juego de ajedrez, tenemos 100 seg. para el movimiento y podemos explorar $10,000 \text{ nodos} \cdot \text{seg}^{-1}$. Podríamos explorar 10^6 nodos por movimiento lo cual, alcanzando profundidades de 8, puede dar lugar a un nivel decente de juego.

Existen dos factores que inciden en la eficiencia de este tipo de algoritmos:

1. la profundidad: cuanto mayor, más eficiente;
2. la exactitud de la función de evaluación.

En general existe una compensación entre ambas: cuanta mayor profundidad alcancemos, menor será la influencia de la calidad de la función de evaluación y, en sentido contrario, si somos capaces de desarrollar una función de evaluación complicada y exacta, ganaremos tiempo al no tener que profundizar demasiado en el árbol de búsqueda.

Por tanto, en este tipo de búsqueda necesitamos *una función* (de evaluación) capaz de decirnos si un *estado no terminal* es bueno o malo. Idealmente, *una función de evaluación debería devolver un valor igual al valor minimax de un estado que no es terminal*, de un modo similar a la heurística de A*. Se trata de una *función de truncación*.

En la práctica se suele buscar una función que, en promedio, sea positiva cuando el valor MINIMAX es positivo y negativa cuando lo es el MAXIMIN.

¿Cómo son estas funciones de evaluación?

1. Evaluación basada en utilidad.

Una forma puede ser calcular el valor esperado de la utilidad estimando las “probabilidades” de diferentes resultados finales desde la posición actual:

$$E[U(n)] = \sum_{\text{resultados}} \text{Prob}(n \rightarrow \text{resultado}) \cdot U(\text{resultado}) \quad (6.6.1)$$

2. Evaluación basada en características.

Si pensamos en el ajedrez, podemos encontrar algunas propiedades que son importantes. P.e. si somos el blanco y tenemos más caballos que el negro, puede ser positivo, pero no tanto como tener más reinas. Consultando con expertos en el juego podemos llegar a reunir un grupo de *características* que puedan ayudarnos a controlar el tablero. Estas

características son ellas mismas funciones que reciben como argumento el estado - n - del juego y devuelven un valor numérico - $f_i(n)$. Una forma típica de combinarlas es linealmente:

$$Eval(n) = w_1 f_1(n) + w_2 f_2(n) + \dots + w_n f_n(n) \quad (6.6.2)$$

Aquellas características consideradas más importantes, reciben un peso mayor medido por w_i y donde p.e. las funciones podrían ser:

$$\begin{aligned} f_1(n) &= (\text{nº de reinas blancas} - \text{nº de reinas negras}) \\ f_2(n) &= (\text{nº de caballos blancos} - \text{nº de caballos negros}) \end{aligned}$$

En general pueden combinarse como

$$Eval(n) = F[f_1(n), \dots, f_n(n)] \quad (6.6.3)$$

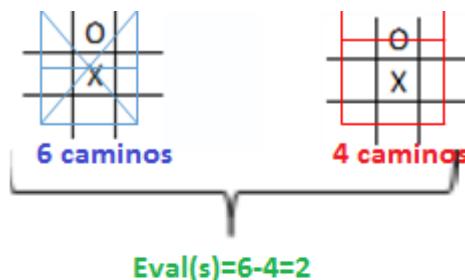
Si nos fijamos en *tres en raya*, una característica que favorece al jugador es la diferencia entre el número de filas, columnas y diagonales completas disponibles para cada jugador, es decir:

$$Eval(n) = NFCyD_{MAX}(n) - NFCyD_{MIN}(n) \quad (6.6.4)$$

y

$$Eval(n) = \begin{cases} \infty \text{ si } n \text{ es nodo ganador para MAX} \\ -\infty \text{ si } n \text{ es nodo ganador para MIN} \end{cases}$$

Por tanto, los siguientes estados tendrán los valores indicados:



$$Eval(s')=\infty$$

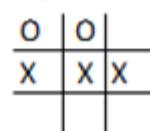


Figura 6.6.1:

En la Fig. 1.5.1. podemos ver algunos valores de esta función de evaluación para algunos casos.

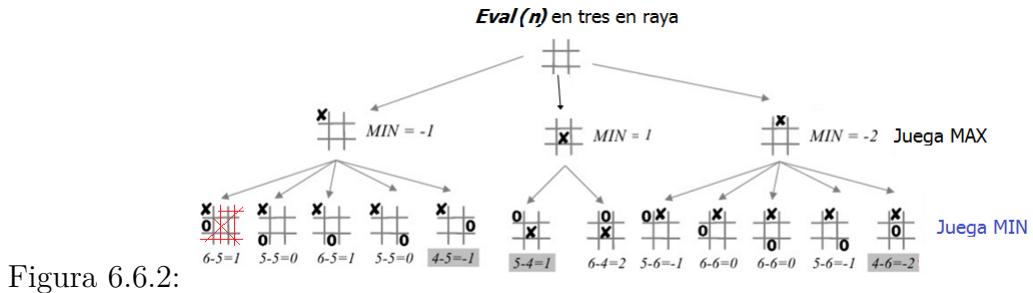
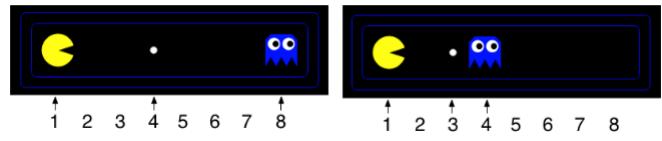


Figura 6.6.2:

La función (6.6.3) F contiene conocimiento experto. F puede ser aprendido (aprendizaje automático) de la experiencia (ej. se puede dejar al ordenador jugar contra sí mismo).

Ejercicio: De las situaciones mostradas en la figura, ¿qué función de evaluación dará mejor puntuación a la izquierda?



- a) $1/(\text{distancia de Pac-Man al punto})$
- b) Distancia de Pac-Man al fantasma.
- c) Distancia de Pac-Man al fantasma+ $1/(\text{distancia de Pac-Man al punto})$
- d) Distancia de Pac-Man al fantasma+ $1000/(\text{distancia de Pac-Man al punto})$

Respuesta: b. (izda. 8, dcha. 4) y c (izda. $8+1/4$; izda $4+1/3$).

6.7 Poda $\alpha - \beta$.

Los métodos de búsqueda expuestos anteriormente separan completamente el proceso de *generación del árbol* de búsqueda de la *evaluación de la posición*. La evaluación solo comienza cuando el árbol ha sido generado. Como veremos en esta sección pueden conseguirse ahorros en el tamaño de la búsqueda si llevamos a cabo las evaluaciones de los nodos hojas y los cálculos de valores *hacia atrás* simultáneamente con la generación del árbol.

Para ello nos basaremos en un hecho observable: *para calcular el valor MINIMAX de un nodo muchas veces no es necesario explorar exhaustivamente*. Por ello vamos a buscar procedimientos para «*podar*» el árbol de aquellas ramas que no precisan ser analizadas.

En la Fig. 1.6.1. presentamos un árbol de juego en que vamos explorando *todos* los nodos en el orden señalado *hacia atrás* (de abajo arriba):



Figura 6.7.1:

Si observamos en detalle el árbol podemos ver que, yendo de abajo arriba, a la profundidad 2, *es totalmente imprescindible visitar los nodos 1, 2 y 3* ya que aún no he visitado ningún camino previo. En efecto siendo $V(1) = 3$ tengo que seguir visitando 2 y 3 por si alguno fuera menor que este. Una vez visitados, $V(10) = \min_{s \in \text{hijos}(10)} V(s) = 3$. Por tanto ahora sabemos que

$$V(13) = \max_{s \in \text{hijos}(13)} V(s) \geq \alpha = 3$$

Siendo hasta el momento, α la mejor opción ya explorada hasta la raíz para MAX a lo largo del camino. Para su adversario MIN, β es la mejor opción ya explorada a lo largo del camino que es $V(10) = 3$ hasta ahora.

Seguimos explorando nodos terminales y, tras visitar el nodo 4, *no es preciso explorar los nodos 5 y 6* ya que, puesto que $V(4) = 2$ tendremos que

$$V(11) = \min_{s \in \text{hijos}(11)} V(s) \leq \beta = 2 \text{ y}$$

$$V(\text{raíz} = 13) = \max_{s \in \text{hijos}(13)} V(s) \geq \alpha = 3$$

y, de la exploración anterior, el nodo padre de 10 y 11 ha de tener un valor mayor o igual que el de estos por lo cual, siendo $\alpha = V(\text{raíz}) = 3 > V(11) = \beta = 2$ es claro que desde el nodo inicial siempre se elegirá el nodo 10 antes que el 11 ya que $V(10) > V(11)$ cualesquiera que sean los valores de los nodos 5 y 6 y, por tanto, *estas ramas pueden «podarse»*. Pero aún nos queda analizar la rama que comienza en 12 para la cual los nodos 7, 8 y 9 sí han de ser analizados en su totalidad ya que

$$V(12) = \min_{s \in \text{hijos}(12)} V(s) = \min(14, 4, 2) = \beta$$

$$V(\text{raíz} = 13) = \max_{s \in \text{hijos}(13)} V(s) \geq \alpha = 3$$

Veamos qué ocurre en cada uno de estos nodos:

1. en 7 solo sabemos que $V(12) \leq 14$ y , si este fuera el menor valor de los nodos hijos de 12, $V(13) = \alpha$ debería ser ≥ 14 . Como $\alpha(= 3) < \beta(= 14)$ hemos de seguir desarrollando.
2. en 8 $V(8) \leq 5$ pero, también ahora $\alpha(= 3) < \beta(= 5)$ y, por último,
3. en 9 $V(9) \leq 2$ y ahora $\alpha(= 3) > \beta(= 2)$ y, es este el valor del nodo 12 - $\min_{s \in \text{hijos}(12)} V(s)$.

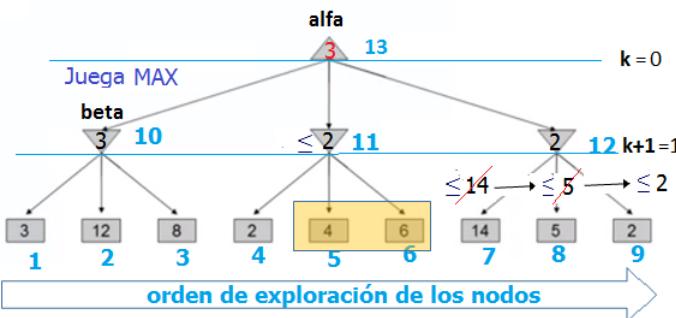


Figura 6.7.2:

Por tanto, hemos visto que la posibilidad de poda depende de los valores de los estados y su posición en el árbol. Hemos podido podar porque el nodo 11 es un candidato para MAX peor que el nodo 10.

Vayamos ahora al tres en raya y empecemos por la última etapa del juego - (B) en la figura 6.7.3):

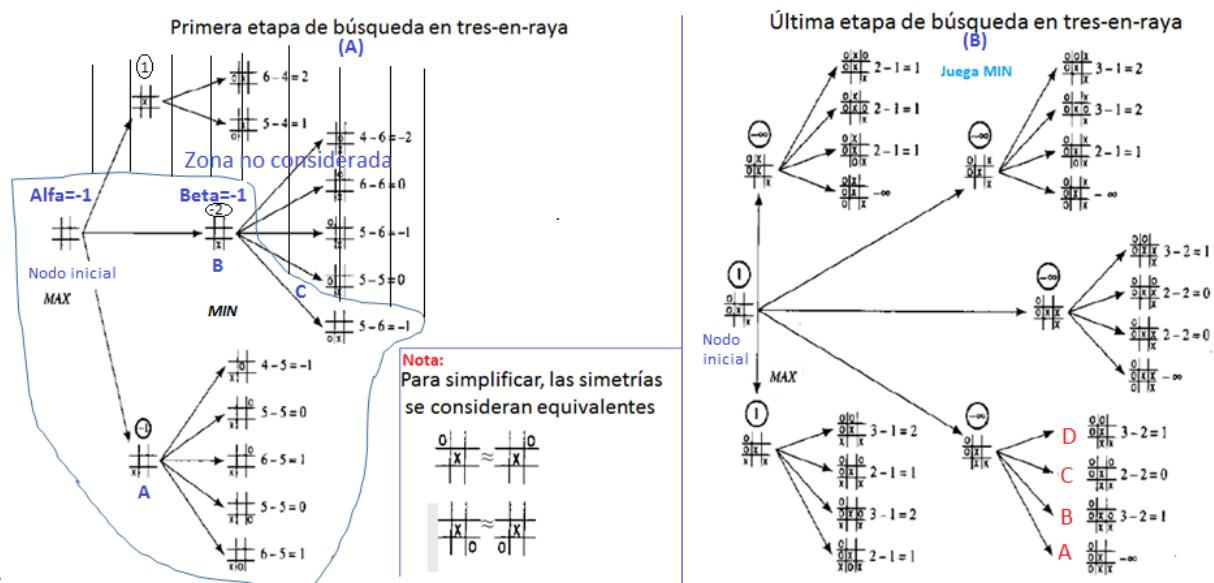


Figura 6.7.3:

y supongamos que un nodo hoja *se evalúa en el momento en que es generado*, entonces en cuanto se genere el nodo A, no tiene sentido *generar* ni *evaluar* B,C,D ya que estando A disponible - nodo de menor valor posible -, MIN no preferirá ningún otro nodo de mayor valor. Es importante observar que descartar los nodos B,C,D no afecta al valor obtenidos por MAX para el primer movimiento.

Esto se basa en el nodo A representa una victoria de MIN. Sin embargo, el mismo ahorro puede conseguirse cuando *ninguna de las posiciones en el árbol representan una victoria* ni para MAX ni para MIN.

Para ello consideremos en la figura 6.7.3 *solo la zona delimitada por la curva del primer movimiento del 3-en-rayas - (A)*. Supongamos que la búsqueda ha procedido como *DFS* y que cuando se genera un nodo, este se evalúa estáticamente. Supongamos asimismo que en cuanto a una posición se le puede dar un valor *hacia atrás*, se calcula este valor. Ahora consideremos la situación que ocurre en *DFS* inmediatamente después de que el nodo A y sus sucesores han sido generados, pero antes de que lo haya sido B. Al nodo A se le da un valor *hacia atrás* de -1. En este punto sabemos que el valor *hacia atrás* del nodo inicial está acotado inferiormente

$$\text{por } -1 \quad (V(n_{raíz}) \geq -1) \text{ ya que } V(n_{raíz}) = \max_{s \in \text{hijos}(n_{aiz})} V(s) \geq V(A) = \alpha = -1.$$

Dependiendo de los valores *hacia atrás* del resto de los sucesores del nodo inicial, el valor final *hacia atrás de este* puede ser mayor que -1, pero no menor. Llamamos a este límite inferior un **valor alfa** para el nodo inicial. cada vez que generamos descendientes y calculamos el valor hacia atrás de un nodo MAX - el raíz - puede cambiar el valor α .

Ahora procedemos DFS hasta que el nodo B y *su primer descendiente C* se generan. Al nodo C se le da un valor estático de -1 de manera que sabemos que el valor del nodo B está acotado *superiormente* por -1. Dependiendo de los valores *hacia atrás* del resto de los sucesores del nodo B, el valor final *hacia atrás de este* puede ser *menor que -1*, pero no mayor y por tanto, el nodo B nunca será preferido al A. Llamamos a este límite superior de B un *valor beta* para el nodo B. Por tanto, nótese que en este punto que el valor final hacia atrás del nodo B no puede exceder al valor *alfa* del *nodo inicial* y, por tanto, podemos abandonar la búsqueda por debajo de B. Está garantizado que el nodo B no se volverá preferible al nodo A.

Esta reducción en la búsqueda se ha conseguido controlando los límites de los valores *hacia atrás*. En general, cuando los sucesores de un nodo reciben valores hacia atrás, los límites de los valores hacia atrás pueden ser revisados pero obsérvese que:

- los *valores alfa* de los nodos *MAX* (incluyendo el raíz) *no pueden disminuir* y
- los *valores beta* de los nodos *MIN* *no pueden aumentar*.

Estas dos restricciones nos permiten definir las siguientes reglas para descontinuar la búsqueda:

1. Se puede podar por debajo de un nodo *MIN* si su valor beta es menor o igual que al menos un alfa antecesor $\beta_p \leq \alpha_{p-1}$. El valor final *hacia atrás* de este nodo MIN puede por tanto fijarse en su valor beta que puede no ser el mismo que se obtendría por una búsqueda MINIMAX pero también proporciona el mejor movimiento.
2. Se puede podar por debajo de un nodo *MAX* si su valor alfa es mayor o igual que al menos un beta antecesor $\alpha_p \geq \beta_{p-1}$. El valor final *hacia atrás* de este nodo MAX puede por tanto fijarse en su valor alfa.

Por tanto, iremos de nodo en nodo y en cada uno de ellos calcularemos las tres variables: V , α y β y donde se verifique que

$$\begin{aligned} \text{Nodos MAX } V \geq \beta &\rightarrow \text{poda } \beta \\ \text{Nodos MIN } V \leq \alpha &\rightarrow \text{poda } \alpha \end{aligned}$$

y no podaremos en el resto. Cuando empecemos asignaremos al nodo raíz los valores iniciales $V = \mp\infty$, $\alpha = \mp\infty$ y $\beta = \pm\infty$ según sea MAX ($-\infty$) o MIN. estos valores se actualizarán a medida que alcancemos nodos terminales.

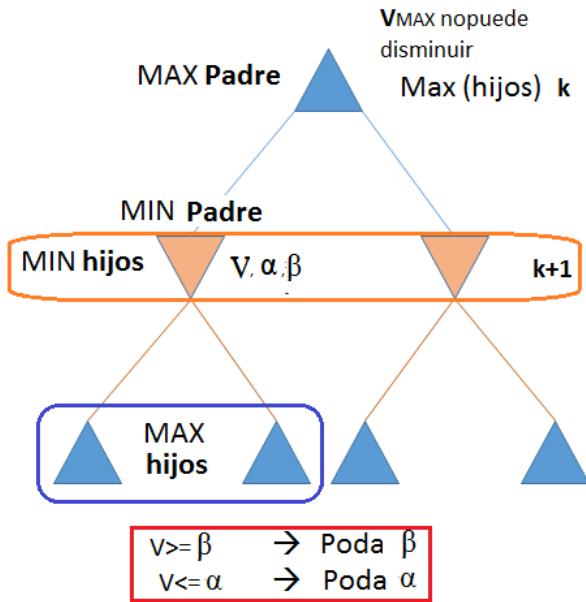


Figura 6.7.4:

Durante la búsqueda, los valores de *alfa* y *beta* se calculan del modo siguiente:

- El valor *alfa* de un nodo *MAX* se hace igual al *mayor* valor *hacia atrás* final de sus sucesores.
- El valor *beta* de un nodo *MIN* se hace igual al *menor* valor *hacia atrás* final de sus sucesores.

El proceso termina cuando todos los sucesores del nodo raíz han recibido valores finales *hacia atrás* y el mejor primer movimiento es el que produce el sucesor que tenga el mayor valor *hacia atrás*. Como se ha dicho, este procedimiento produce el mismo resultado que el MINIMAX con mucha menos búsqueda.

En la práctica es un esquema muy simple:

Desarrollar la búsqueda hacia atrás del MINIMAX con una excepción: si en el curso de la actualización del valor minimax de un nodo, su valor atraviesa un cierto valor (hacia arriba o abajo), entonces no se necesita mayor exploración por debajo de ese nodo. Su valor actual puede transmitirse a su padre como si se hubieran examinado todos los hijos.

El algoritmo - recursivo - puede expresarse en pseudocódigo:

$\mathbf{V}(n; \alpha, \beta)$: recibe dos parámetros α y β y evalúa $V(n)$ valor MINIMAX del nodo n, si $\alpha \leq V(n) \leq \beta$.

Si no devuelve bien α ($V(n) \leq \alpha$), bien β ($V(n) \geq \beta$).

Claramente, si $n = \text{nodo raíz}$, su valor MINIMAX se obtendrá mediante $\mathbf{V}(n; -\infty, +\infty)$.

1. **if** n es terminal **then** devolver $V(n)$ =evaluación *estática* de n.
Else sean $n_1, \dots, n_k, \dots, n_b$ los sucesores de n ordenados. Hagamos $k \leftarrow 1$
if n es un nodo MAX **then** ir al paso 2.1
else - es MIN - ir a 2.2

2.1. $\alpha \leftarrow \max[\alpha, V(n_k; \alpha, \beta)]$	2.2. $\beta \leftarrow \min[\beta, V(n_k; \alpha, \beta)]$
3.1. Si $\alpha \geq \beta$, devolver β . Si no continuar	3.2. Si $\beta \leq \alpha$, devolver α . Si no continuar
4.1. Si $k = b$ devolver α ; si no avanzar a n_{k+1} es decir $k \leftarrow k + 1$ e ir al paso 2.1.	4.2. Si $k = b$ devolver β ; si no avanzar a n_{k+1} es decir $k \leftarrow k + 1$ e ir al paso 2.2.

A medida que realizamos la búsqueda en el nodo n (MIN) como empezamos con ∞ la estimación de los sucesores de n va disminuyendo y su valor le interesa a MAX pero solo en tanto no haya a lo largo del camino hacia el nodo ningún valor que para MAX sea mejor.

6.7.1 Características de la poda $\alpha - \beta$.

- La poda no afecta al resultado final, pero sí puede afectar a otros nodos intermedios. Por este motivo, la versión más *ingenua* de la poda *puede dejar al agente sin estrategia. No basta con estar acertado en la raíz, sino también en los sucesores de esta*. En la figura siguiente, el nodo a la derecha de D puede podarse ya que al nodo raíz (MAX) solo le importa que $V(B) \leq 10 = V(A)$ y, por lo tanto, no necesita conocer el valor exacto $V(B)$ ya que en el nodo raíz que se encuentra por encima de A y B *cualquier valor de $V(B) > 10$ nunca será elegido* por MAX y $\text{MAX}(V(A), V(B)) = \text{MAX}(10, \leq 10) = 10$

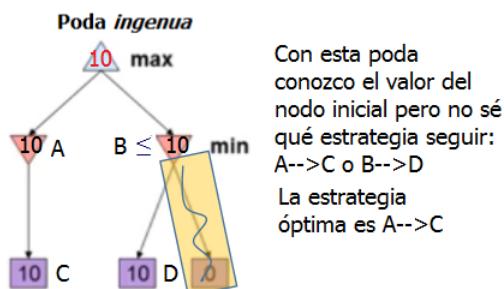


Figura 6.7.5:

- La eficacia de la poda depende del orden en la búsqueda: Es buena si los movimientos buenos se exploran primero.
 - Caso peor: No hay mejora
 - Ordenación aleatoria: $O(b^{3d/4}) \Rightarrow b^* = b^{3/4}$
 - Ordenación perfecta (los sucesores mejores están los primeros): $O(b^{m/2}) \Rightarrow b^* = b^{1/2}$.

El uso de heurísticas sencillas lleva a menudo a b^* cercano al óptimo (ej. examinar primero los movimientos que fueron los mejor considerados en el anterior turno).

6.8 La poda $\alpha - \beta$ paso a paso.

La figura siguiente muestra paso a paso la aplicación de la poda alfa-beta seg\xf1n el algoritmo expuesto.

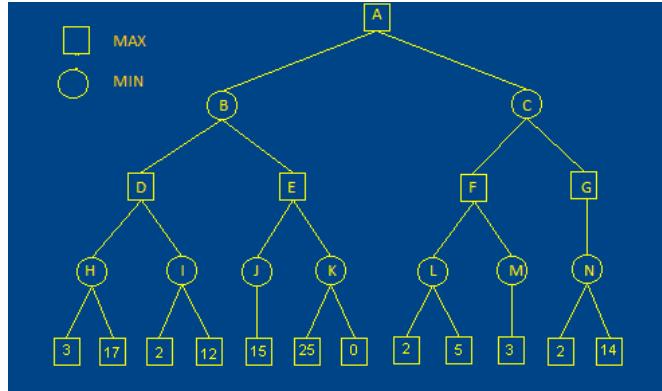


Figura 6.8.1:

El desarrollo de Fig (6.8.1) es el siguiente:

1. Como sabemos, en cada nodo N el valor estimado se encuentra acotado entre dos valores $\alpha \leq V(N) \leq \beta$

Inicialmente no hemos explorado ning\xf1n nodo terminal, por lo que partimos del nodo ra\xedz con $-\infty \leq V(A) \leq \infty$. Como contiene un rango v\xedlido de valores, este nodo pasará hacia abajo los valores de $\alpha = -\infty$ y $\beta = +\infty$ y así sucesivamente de padre a hijo hasta llegar al nodo terminal 3 (Fig 6.8.2).

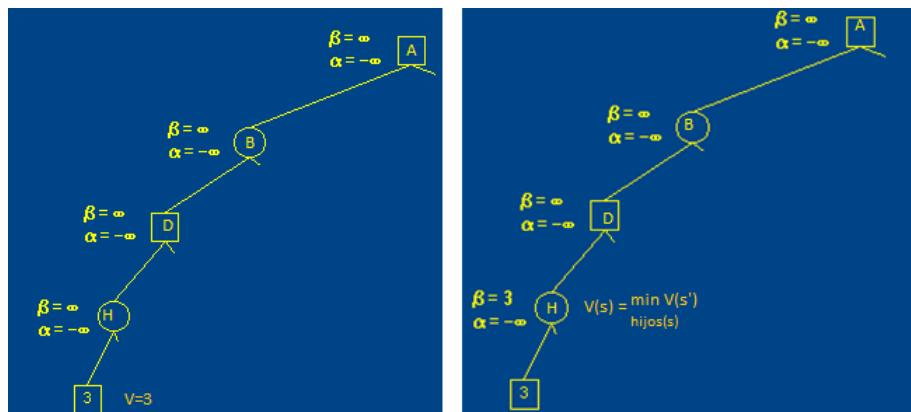


Figura 6.8.2:

2. Evaluamos el nodo terminal -MAX - 3 ya que es un nodo hoja y se pasa hacia el nodo anterior que es MIN, por lo que su valor MINIMAX debe ser $V(H) \leq 3$, por lo que hacemos $\beta = 3$.

3. Seguidamente generamos el siguiente nodo hijo que devuelve al nodo padre el valor 17. Como el padre es MIN, este valor que es mayor que β se ignora y por tanto $V(H) = 3$ (Fig. 6.8.3) que se pasa para cambiar α al nodo D - padre de H .

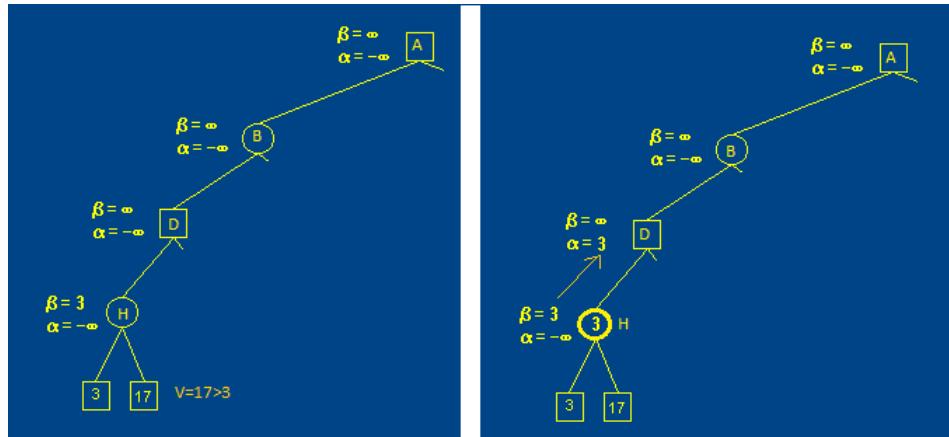


Figura 6.8.3:

Nótese

- que β no ha cambiado porque los nodos MAX solo aplican restricciones al límite inferior;
- que los valores α, β pasados hacia abajo en el árbol no se modifican, pero sí los que pasan hacia arriba:
 - el valor final de β en un nodo MIN se pasa hacia el padre para un posible cambio en su α y, del mismo modo,
 - el valor final de α en un nodo MAX se pasa hacia el padre para un posible cambio en su β .

4. Ahora el nodo D tiene una cota inferior y expandimos su hijo I (Fig. 6.8.4) y, como no es terminal, generaremos primero 2 y correremos la función de evaluación que nos devuelve $V=2$.

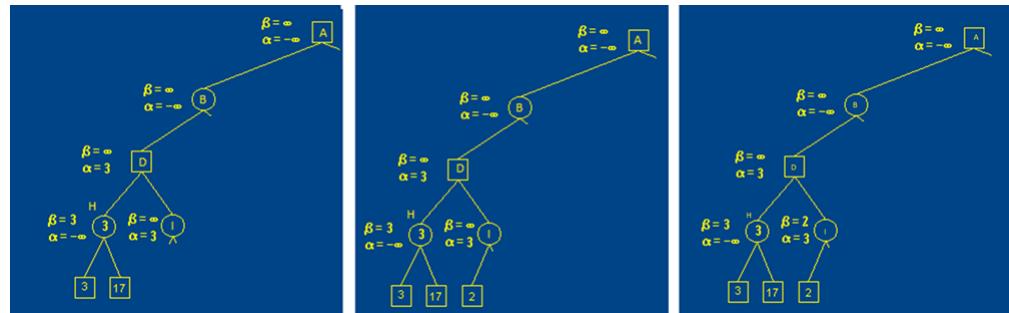


Figura 6.8.4:

Como I es MIN sabemos que $V(I) \leq 2$ de modo que cambiaremos $\beta = 2$.

5. Obsérvese que ahora en I : $3 \leq V(I) \leq 2$ lo cual no es posible y, por tanto, dejamos de evaluar hijos de este nodo y devolvemos el valor 2 (Fig. 6.8.5). En realidad, desconocemos

el verdadero valor de $V(I)$. El valor del otro hijo de I puede ser 1, 0 o -100 , pero incluso en dichos casos, la búsqueda no nos ayudaría en nada a encontrar la solución óptima. El valor 2 ya convierte al subárbol en improductivo y podamos a los restantes hijos.

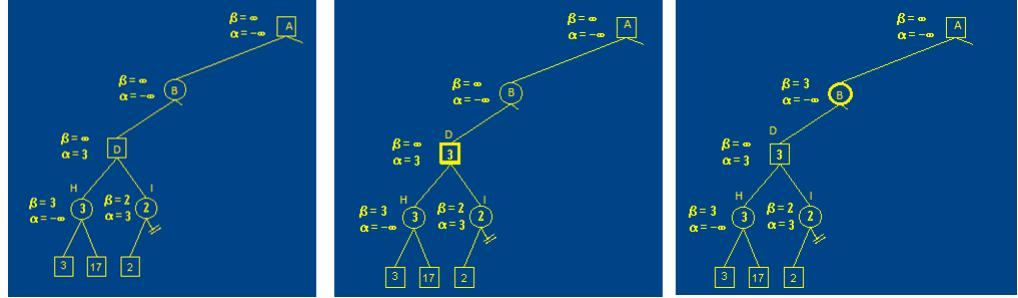


Figura 6.8.5:

Hemos hecho una *poda- β*

6. Hacia arriba, el valor α del nodo padre D ya tiene el valor 3 y no se modifica por lo que podemos fijar $V(D) = \alpha = 3$.
7. Ahora que hemos dado un valor al nodo D , nos movemos hacia arriba (nodo B) que, siendo MIN, nos lleva a que $V(B) \leq 3 \rightarrow \beta = 3$ y siendo válida la desigualdad , continuamos expandiendo el nodo B (Fig. 6.8.6) generando el nodo MAX E , después su hijo J y, finalmente el nodo terminal 15. A lo largo de este camino descendente nos hemos limitado a pasar los límites α y β .

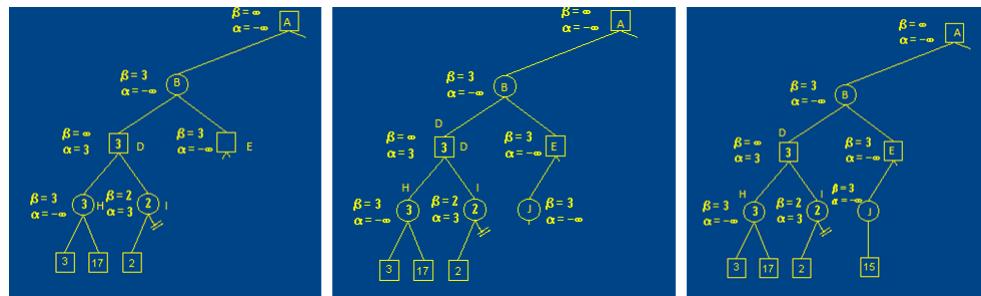


Figura 6.8.6:

8. Este es el único nodo hijo de J y no hemos cambiado el límite β . Como no hemos excedido el límite, devolveremos el valor actual MIN para el nodo. esto es distinto del caso en que se podaba en que devolvíamos el valor β por las razones que se explican más adelante.

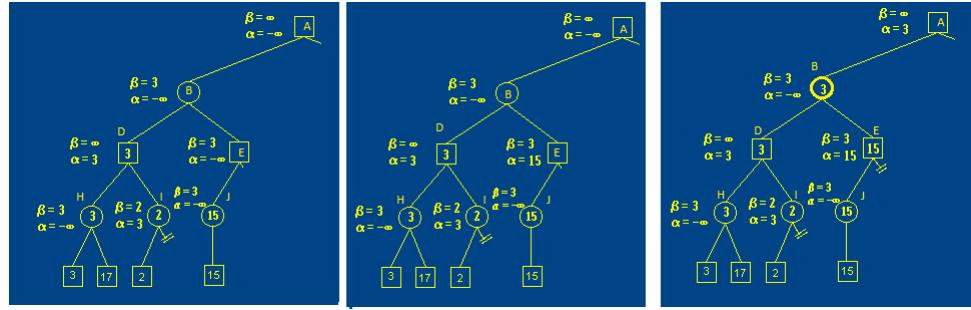


Figura 6.8.7:

Como se cruzan los valores (es imposible que $15 \leq V \leq 3$) se podan el resto de los nodos y se devuelve el valor que excede el límite. Si hubiéramos devuelto el valor β del nodo MIN hijo (3) en lugar del valor actual (15), no habríamos podido podar aquí. Ahora el nodo B MIN conoce a todos sus hijos y puede tomar el valor mínimo: $V(B) = 3$. Además, hemos terminado con el hijo B del nodo raíz A e iremos a por el segundo.

- Pasamos los valores $\alpha = 3$ y $\beta = \infty$ hacia abajo y generamos el segundo hijo C , el primer hijo de este (F), el primer hijo de este último (L) y así hasta llegar a la profundidad 4 cuya función de evaluación nos devuelve 2.

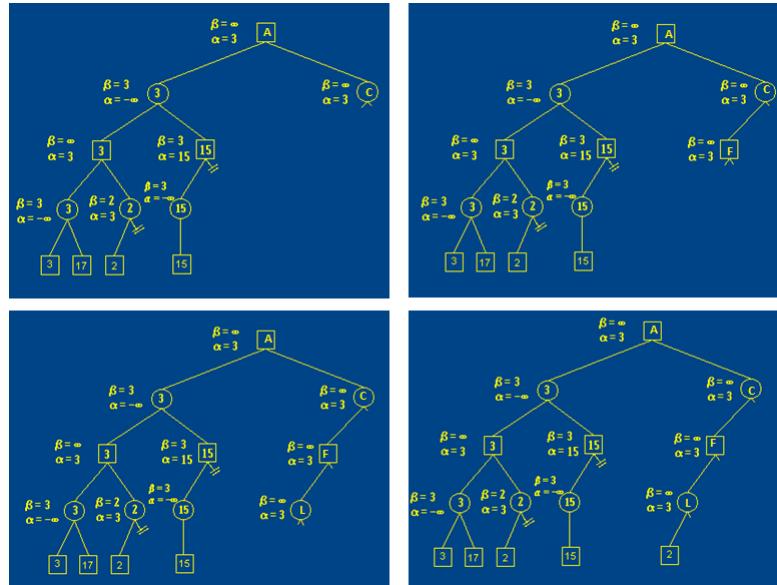


Figura 6.8.8:

- Este valor es subido hasta el padre L para cambiar su $\beta = 2$ y siendo $\alpha > \beta$ podemos los demás hijos de L . Como $\beta = 2$ es menor que el α del parent (F), no modificamos este valor.

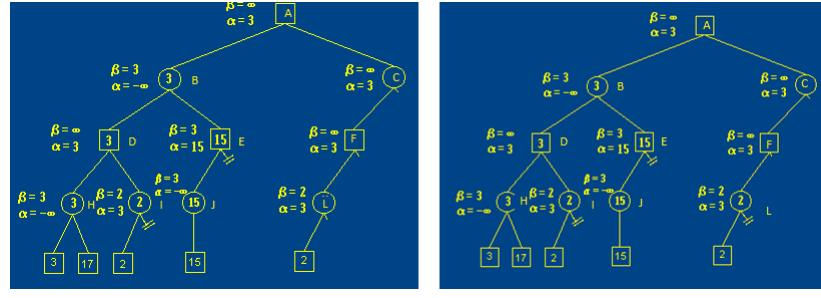


Figura 6.8.9:

11. Ahora generamos el siguiente hijo de F (nodo M) y su hijo en profundidad 4 (Fig. 6.8.10) cuya función de evaluación nos devuelve $V=3$.

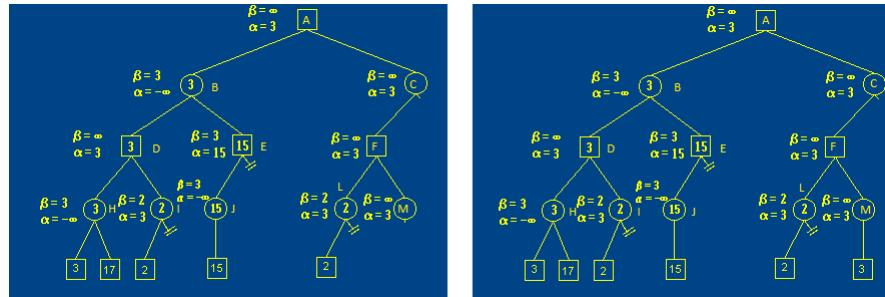


Figura 6.8.10:

12. El nodo padre MIN (M) usa este valor para su límite superior β (Fig.6.8.11). Ahora, en el nodo M $\alpha = \beta = 3$. ¿Debemos podar? La respuesta es *sí* porque aunque no hemos excedido los límites, la igualdad de α y β nos dice que no podemos hacer nada mejor en este subárbol, pero sí lo podemos hacer peor. Por ello acabamos con este nodo y devolvemos el valor 3 para MIN.

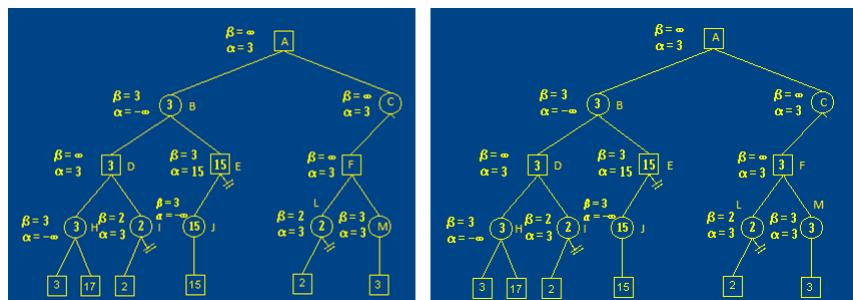


Figura 6.8.11:

13. El valor se devuelve a su nodo padre (F) y su $\beta = 3$ y, puesto que de nuevo $\alpha = \beta$, podamos. El valor final de A será 3.

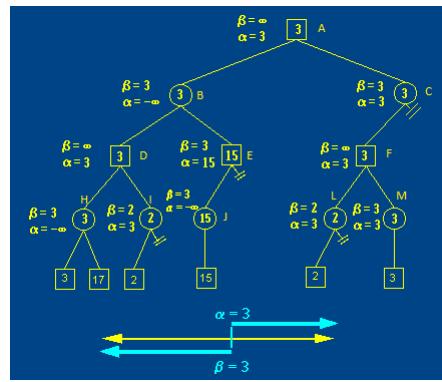


Figura 6.8.12:

Ejercicios.

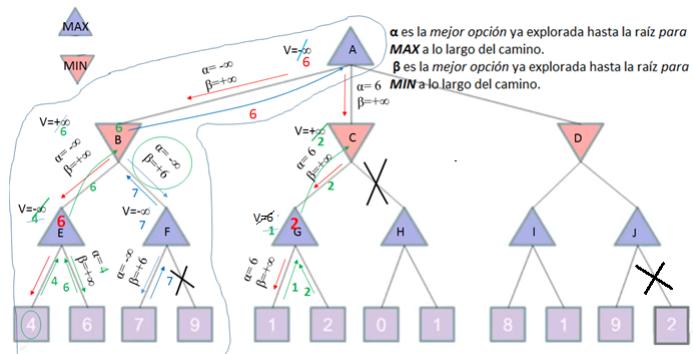


Figura 6.8.13:

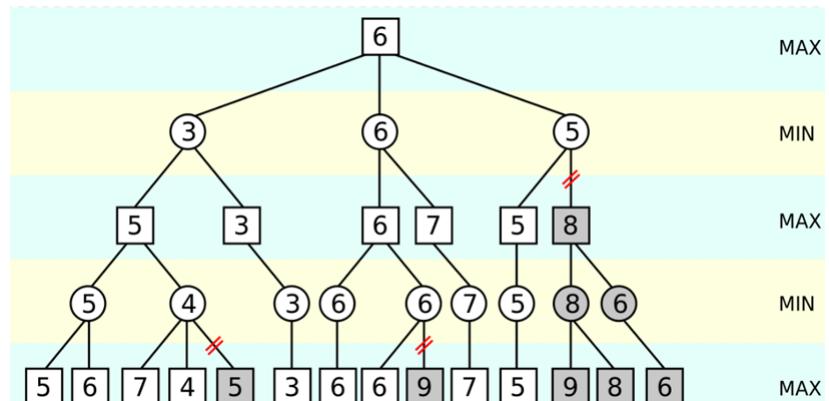


Figura 6.8.14:

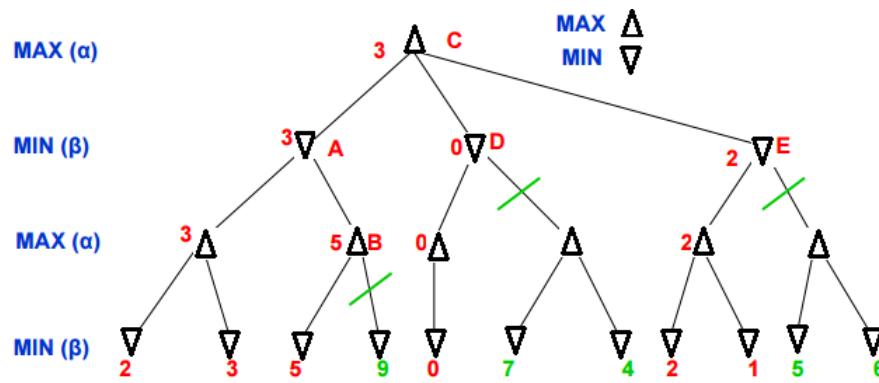


Figura 6.8.15:

Ejemplo de poda alfa-beta

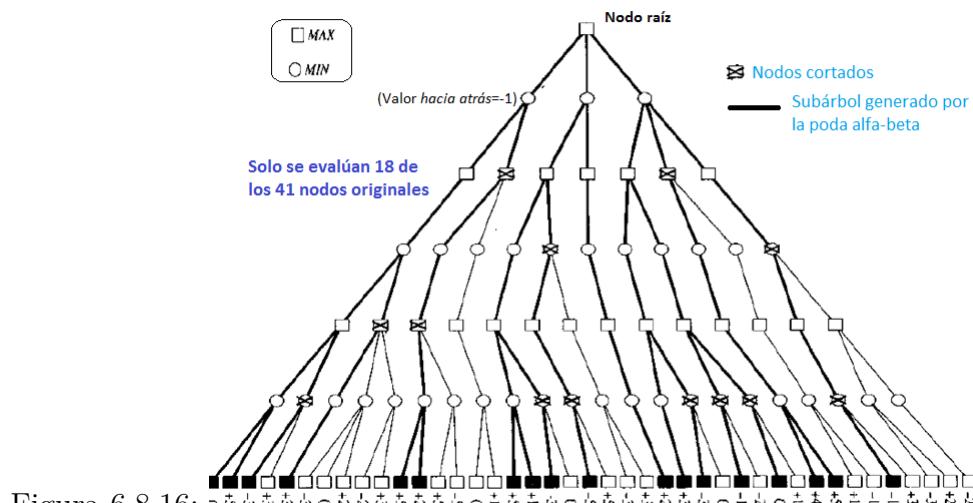


Figura 6.8.16: