CAPITULO 1

RESOLUCIÓN DE PROBLEMAS

- **Espacio de Estados**
- **Características Generales de los Procesos de Búsqueda**
- **Estrategias de Exploración del Espacio de Estados**
- **Resumen**
- **Textos Básicos**

1. RESOLUCIÓN DE PROBLEMAS

No podemos decir que algo o alguien exhibe comportamiento inteligente si no explota de manera eficaz y eficiente[1] un conjunto mínimo de conocimientos. Una arquitectura, natural o artificial, bien definida y estructurada pero vacía, no puede utilizarse para resolver problemas mientras no incorpora procedimientos de resolución y conocimientos propios del dominio de los problemas planteados.

Es claro que no todos los problemas son iguales, por lo que los tipos de conocimiento necesarios van a ser diferentes también. De hecho, para ciertas clases de problemas, lo que habitualmente entendemos como "conocimiento" puede ser sustituido por procedimientos preestablecidos que se ejecutan mecánicamente. Así, cualquier ordenador debidamente programado es capaz de "resolver" la ecuación:

$$x^2 + 4x + 3 = 0$$

Incluso, aunque con un esfuerzo de programación algo mayor, ningún ordenador tendría excesivos problemas para resolver ecuaciones como la que sigue:

$$ax^2 + bx + c = 0$$

La resolución de este tipo de problemas no requiere necesariamente la utilización de técnicas de inteligencia artificial. Es más, su empleo sería equivalente a utilizar cañones para matar moscas... No sería apropiado.

En general, el tipo de problema planteado condiciona la técnica de resolución a emplear. No obstante, la decisión de utilizar una técnica de IA o una técnica convencional de programación puede ser también cuestión de planteamiento. Como norma general, el empleo de técnicas de IA debe permitir la construcción de programas que:

- Capten generalizaciones, de forma que cada situación individual que se produzca no tenga que ser representada de forma separada. Por el contrario, todas aquellas situaciones que compartan propiedades deben ser agrupadas.

- Hagan explícito su conocimiento, al objeto de facilitar su comprensión.

- Puedan actualizarse continuamente, de forma que sea factible modificar el conocimiento sin tener que manipular ni alterar todo el programa.

- Puedan ser empleados en muchas situaciones, aún cuando las respuestas que generen sean parcialmente correctas o imprecisas.

La aplicación de estos criterios en la resolución de problemas de dominios concretos, conduce a la construcción de programas con características esencialmente

[1] Decimos que un sistema es eficaz cuando es capaz de resolver correctamente un problema. Un sistema es eficiente cuando, además de comportarse eficazmente, optimiza los recursos disponibles.

diferentes a los construidos mediante técnicas convencionales. Así, mientras los programas de IA deben ser empleados para tratar fundamentalmente dominios simbólicos, los programas convencionales son particularmente idóneos para tratar dominios numéricos. Por otra parte, la búsqueda de soluciones en los dominios apetecidos por la IA se realiza a través de procesos heurísticos[2], en los que los pasos hacia la solución suelen ir implícitos. Por el contrario, los programas convencionales emplean procedimientos algorítmicos de búsqueda, con pasos explícitos hacia la solución. Por último, mientras en los programas convencionales la información y el control se encuentran físicamente integrados en una misma estructura, en los programas de IA el conocimiento del dominio y las estructuras de control suelen estar físicamente separados, dando como resultado arquitecturas mucho más modulares.

Como siempre ocurre, la diferencia no es lo suficientemente evidente como para permitirnos establecer una frontera clara entre los "problemas de IA" (i.e., aquellos para cuya resolución está indicada la aplicación de técnicas de IA), y los "problemas convencionales" (i.e., aquellos para cuya resolución están indicadas las técnicas de programación convencional). Sin embargo, un análisis cuidadoso del dominio nos puede dar pistas a la hora de elegir una u otra filosofía, y evitar así errores provocados por el hecho de no ajustar la técnica al tipo de problema que queremos resolver.

Analicemos el siguiente problema:

"Supongamos que tenemos dos cubos inicialmente vacíos, uno de 6 litros y el otro de 8 litros. Ninguno de los cubos tiene marca ni división alguna. Disponemos de un grifo de agua que podemos utilizar. ¿Cómo podríamos llenar el cubo de 8 litros exactamente hasta la mitad?"

Es claro que en este problema hay un conjunto de acciones que pueden llevarse a cabo si se cumplen unos requisitos mínimos. La situación se ilustra esquemáticamente en la Tabla 1.1.

[2] El conocimiento heurístico es un conocimiento difícilmente formalizable, fruto de la experiencia, y que se establece implícitamente para tratar de encontrar respuestas más o menos correctas, pero siempre válidas, a problemas de dominios concretos.

Nº	PRECONDICIONES	ACCIONES
1	El cubo de 8 litros no está lleno	Llenar el cubo de 8 litros
2	El cubo de 6 litros no está lleno	Llenar el cubo de 6 litros
3	El cubo de 8 litros no está vacío	Vaciar el cubo de 8 litros
4	El cubo de 6 litros no está vacío	Vaciar el cubo de 6 litros
5	El cubo de 8 litros no está vacío, y El cubo de 6 litros no está lleno, y El contenido de ambos es menor o igual a seis litros	Vaciar el contenido del cubo de 8 litros en el cubo de 6 litros
6	El cubo de 6 litros no está vacío, y El cubo de 8 litros no está lleno, y El contenido de ambos cubos es menor o igual a 8 litros	Vaciar el contenido del cubo de 6 litros en el cubo de 8 litros
7	El cubo de 6 litros no está vacío, y El cubo de 8 litros no está lleno, y El contenido de ambos cubos es mayor o igual a 8 litros	Llenar el cubo de 8 litros con el contenido del cubo de 6 litros
8	El cubo de 8 litros no está vacío, y El cubo de 6 litros no está lleno, y El contenido de ambos cubos es mayor o igual a 6 litros	Llenar el cubo de 6 litros con el contenido del cubo de 8 litros

Tabla 1.1 Requisitos mínimos y acciones posibles relativos al problema de los dos cubos

Una secuencia de acciones que resolvería el problema inicialmente planteado (i.e., conseguir un estado final en el que el cubo de 8 litros estuviese lleno justamente hasta la mitad), podría ser la siguiente:

primero: Con los dos cubos vacíos ejecutar la acción número 2, con lo que tendríamos el cubo de 8 litros vacío, y el cubo de 6 litros lleno.

segundo: Ejecutar la acción número 6, con lo que tendríamos 6 litros en el cubo de 8 litros, y el cubo de 6 litros vacío.

tercero: Repetir la acción número 2, con lo que tendríamos 6 litros en cada uno de los cubos.

cuarto: Ejecutar la acción número 7, con lo que tendríamos el cubo de 8 litros lleno, y nos quedarían 4 litros en el cubo de 6 litros.

quinto: Ejecutar la acción número 3, con lo que quedarían 4 litros en el cubo de 6 litros, y el cubo de 8 litros estaría vacío.

sexto: Ejecutar la acción número 6, con lo que tendríamos 4 litros en el cubo de 8 litros, y el cubo de 6 litros estaría vacío, lo que representa una posible solución a nuestro problema.

Nótese que, en este ejemplo, la solución propuesta es sólo una de un conjunto de soluciones válidas. En efecto, cualquier estado en el que el cubo de 8 litros estuviese lleno por la mitad sería válido, independientemente de la cantidad de agua que hubiese en el cubo de 6 litros.

Parece que hemos sido capaces de encontrar una solución al problema planteado. Podríamos pensar ahora en cómo representar ambos, problema y solución, al objeto de facilitar la posterior labor de implementación. Al respecto, consideraremos nuestro universo de estados posibles como pares ordenados, en los que el primer elemento del

par representa el contenido del cubo de 8 litros, y el segundo elemento del par representa el contenido del cubo de 6 litros. Las acciones que suponen transiciones entre estados posibles las representaremos entre guiones. Con este formalismo, la secuencia de estados por los que el problema evoluciona, desde el estado inicial hasta que llegamos a una solución "razonable", es la siguiente:

(0,0) -2- (0,6) -6- (6,0) -2- (6,6) -7- (8,4) -3- (0,4) -6- (4,0)

Una forma de programar una solución a este problema consiste en resolverlo previamente sobre el papel y luego codificar la solución en un lenguaje cualquiera. Tal procedimiento, sin duda eficaz en el tiempo, tiene no obstante serios inconvenientes. Uno de ellos es la "rigidez" del programa resultante. Efectivamente, simplemente variando las condiciones iniciales, o la solución buscada, el problema sería diferente y, por lo tanto, el programa diseñado no sería válido. Hay que buscar otro enfoque.

Supongamos que podemos definir "a priori" un conjunto de situaciones posibles y unas "reglas del juego" potencialmente útiles, y supongamos también que podemos diseñar unas estrategias generales, que investiguen la aplicabilidad de las reglas del juego sobre los estados para obtener otros estados, de forma que sea el ordenador quien encuentre por sí mismo la solución al problema. Nótese que este nuevo enfoque es de naturaleza no determinística. Además, un programa que siguiese este planteamiento sería mucho más general que otro que siguiese la primera aproximación. De hecho, el mismo programa podría utilizarse para resolver un número, en principio, ilimitado de problemas, ya que sólo cambiarían las reglas del juego aplicables, en función del estado del problema en cada momento. Para conseguir esto, en inteligencia artificial se define el "espacio de estados " del problema.

1.1. Espacio de Estados

En inteligencia artificial es útil definir el dominio del problema que queremos resolver como un "espacio de estados". El espacio de estados es una descripción formal del universo de discurso, y está constituido por los siguientes elementos:

- Un conjunto de estados iniciales.

- Un conjunto de operadores que definen operaciones permitidas entre estados. En el espacio de estados no todas las operaciones definidas van a ser aplicables siempre. Así, en un momento dado, el conjunto de estados que representa la situación actual de nuestro problema es quien determina el subconjunto de "operadores aplicables" (o "relevantes"), del conjunto de operadores previamente definido. Para que un operador pueda ser considerado relevante debe cumplir un conjunto de requisitos mínimos, tal y como fue establecido en el ejemplo de los dos cubos.

- Un conjunto de metas (i.e. objetivos), que cumplen los requisitos suficientes para ser consideradas soluciones aceptables de nuestro problema.

El espacio de estados es útil, ya que permite describir formalmente un problema como un conjunto de transformaciones, desde unas situaciones dadas hasta unas

situaciones deseadas, a través de un conjunto de operaciones permitidas. Más concretamente, el espacio de estados nos permite contemplar el proceso global de solución de un problema como:

- (a) la aplicación de un conjunto de técnicas conocidas, cada una de ellas definida como un paso simple en el espacio de estados , y
- (b) un proceso de búsqueda, o estrategia general de exploración del espacio de estados[3].

Formalmente:

Si I define al conjunto de estados iniciales, tal que $I = [i_1, i_2, \ldots, i_n]$, O define al conjunto de operadores potencialmente útiles, tal que $O = [o_1, o_2, \ldots, o_m]$, y M define al conjunto de metas o estados finales, tal que $M = [m_1, m_2, \ldots, m_t]$, la búsqueda se define como el proceso de exploración del espacio de estados que produce O: $(I \rightarrow M)$, expresión que podemos interpretar como una evolución desde los estados iniciales hasta los estados finales obtenida tras la aplicación de un conjunto de operadores. En este mismo contexto una transición simple en el espacio de estados puede representarse como

$$o_x : (i_z \rightarrow i_w), \text{con } i_z, i_w \in I, o_x \in O$$

Además, si $i_w \in M$, entonces i_w representa una solución aceptable para nuestro problema. La llamada *prueba de meta* o *test de realización* aplicada a la descripción de un estado permitirá decidir si alguno de los nuevos estados generados se trata de un estado meta. Es conveniente mencionar aquí que cualquier estado alcanzado durante la búsqueda que no pertenezca al conjunto de metas puede ser considerado como un nuevo estado inicial del problema.

Este formalismo de resolución normalmente se traduce en la creación de programas menos eficientes que los programas convencionales pero, desde luego, mucho más flexibles y generales[4]. En cualquier caso, la resolución de problemas en inteligencia artificial requiere siempre una descripción formal y manejable del problema. En otras palabras, para abordar un problema desde la perspectiva de la IA tendremos que elaborar un *modelo computacional* del universo de discurso o dominio del problema[5].

El espacio de estados, aunque imprescindible para la representación formal de un problema de IA, tan sólo nos proporciona lo que podríamos llamar el "soporte físico" del dominio. El aspecto más dinámico de obtención de soluciones se materializa a través

---

[3] Esta es precisamente la razón de la separación física entre los conocimientos del sistema y los mecanismos de control del conocimiento.

[4] El término "eficiente" es utilizado aquí en sentido físico: eficiencia=eficacia/tiempo

[5] Los conceptos de "universo de discurso" y "dominio del problema" son prácticamente equivalentes y se refieren a un determinado ámbito de aplicación (e.g., el dominio de la Medicina,...)

de los llamados "procesos de búsqueda", que básicamente son mecanismos generales de exploración del espacio de estados.

El concepto de búsqueda está íntimamente ligado a la aplicación de operadores relevantes. Más generalmente, la búsqueda se puede asimilar a los procesos de control que guían la ejecución de un programa de IA, según los cuales el sistema debe de ser capaz, en todo momento, de decidir cuál será su próximo movimiento en el espacio de estados. Conviene que la búsqueda sea sistemática (para evitar dar "rodeos" innecesarios), y que la aplicación de cada operador provoque un movimiento que genere la aparición de un estado nuevo.

Para ilustrar estas nuevas ideas volveremos al ejemplo de los dos cubos, para el cual, el primer paso será definir su espacio de estados.

Descripción del espacio de estados para el problema de los dos cubos

Estado genérico: (A, B) donde A = contenido del cubo de 8 litros
B = contenido del cubo de 6 litros

I = $(0, 0)$
M = $(4, B)$ $B \in [0, 6]$
O = [
op1 = Llenar A,
op2 = Llenar B,
op3 = Vaciar A,
op4 = Vaciar B,
op5 = Vaciar A en B,
op6 = Vaciar B en A,
op7 = Llenar A con B,
op8 = Llenar B con A
]

Precondiciones:

op1 $\Leftrightarrow$ $A \neq 8$
op2 $\Leftrightarrow$ $B \neq 6$
op3 $\Leftrightarrow$ $A \neq 0$
op4 $\Leftrightarrow$ $B \neq 0$
op5 $\Leftrightarrow$ $A \neq 0 \wedge B \neq 6 \wedge A + B \leq 6$
op6 $\Leftrightarrow$ $A \neq 8 \wedge B \neq 0 \wedge A + B \leq 8$
op7 $\Leftrightarrow$ $A \neq 8 \wedge B \neq 0 \wedge A + B \geq 8$
op8 $\Leftrightarrow$ $A \neq 0 \wedge B \neq 6 \wedge A + B \geq 6$

Buscaremos ahora una estrategia que nos permita resolver el problema[6], y trataremos de observar los criterios, ya comentados, de "sistematicidad" y de "generación de estados nuevos".

Estrategia número 1

A priori, una buena posibilidad podría ser la de aplicar el primer operador, según su número de orden, que cumpla los requisitos del estado actual. De acuerdo con esta estrategia, el primer estado es el estado inicial (0,0), y el primer operador aplicable es op1. El nuevo estado generado es el (8,0), sobre el que se puede aplicar op2 (op1 ya no puede ser aplicado puesto que el cubo *A* está lleno). El nuevo estado generado es (8,6), sobre el que se puede aplicar op3 para dar el estado (0,6). Ahora sí se puede aplicar nuevamente op1, que vuelve a generar el estado (8,6). Hemos caído en un bucle. El proceso continuaría indefinidamente con la aplicación sucesiva de op1 y de op3, y la generación sucesiva de los estados (8,6) y (0,6). La estrategia no es buena. La secuencia de estados generados es:

(0,0) -1- (8,0) -2- (8,6) -3- (0,6) -1- (8,6) -3- ...

Esta estrategia de búsqueda es sistemática, pero no observa la condición de generar estados nuevos. Además, hay operadores cuya aplicación se repite lo cual, como veremos más adelante, no es deseable. Tendremos que buscar una estrategia alternativa que trate de paliar los inconvenientes encontrados.

Estrategia número 2

Intentaremos ahora evitar la aplicación repetitiva de operadores manteniendo el criterio de sistematicidad. La estrategia podría seguir el siguiente esquema:

- Seleccionar los operadores que verifiquen las precondiciones del estado actual.
- Descartar aquellos operadores que ya hayan sido aplicados.
- Aplicar el primero de los operadores "supervivientes".

Nótese que esta estrategia nos obliga a definir estructuras adicionales que nos permitan comprobar si un operador ha sido aplicado ya o no. El procedimiento genera la siguiente secuencia de estados:

(0,0) -1- (8,0) -2- (8,6) -3- (0,6) -4- (0,0)

La búsqueda se detiene sin haberse encontrado una solución. No podemos aplicar más operadores puesto que op1, op2, op3 y op4 han sido ya aplicados, y op5, op6, op7 y op8 no verifican las restricciones impuestas por el estado actual. Además,

6 En toda esta discusión asumiremos que los operadores están ordenados según su número de identificación. Así, el primer operador será op1, y el último será op8.

todavía no hemos conseguido evitar la generación de estados ya generados. Necesitamos una nueva aproximación.

Estrategia número 3

Impediremos ahora de forma explícita los inconvenientes señalados cuando comentamos la primera estrategia. Para ello definiremos el siguiente esquema:

- Seleccionar los operadores que verifiquen las precondiciones del estado actual.
- Descartar aquellos operadores que ya hayan sido aplicados.
- Descartar aquellos operadores cuya aplicación no genere un estado nuevo.
- Aplicar el primero de los operadores "supervivientes".

Nuevamente precisamos definir aquí estructuras adicionales. Aparte de saber qué operadores hemos aplicado ya, debemos comprobar si un estado es nuevo o no.

La secuencia de estados que resulta de esta estrategia es la siguiente:

(0,0) -1- (8,0) -2- (8,6) -3- (0,6) -6- (6,0)

Al alcanzarse el estado (6,0) el proceso nuevamente se detiene ya que los operadores op1, op2, op3 y op6 han sido ya aplicados, y los operadores op4, op7 y op8 no verifican las precondiciones del estado actual. El operador op5, que todavía no ha sido aplicado, y que verifica las precondiciones del estado actual, no puede ser ejecutado, porque el esquema de control prohíbe la utilización de operadores que generen estados ya existentes en la secuencia (i.e., (6,0) -5- (0,6)).

A la vista de los resultados parece evidente que hemos de sacrificar algo si queremos encontrar una solución razonable. Analicemos ahora la siguiente estrategia.

Estrategia número 4

- Seleccionar los operadores que verifiquen las precondiciones del estado actual.
- Descartar aquellos operadores que no generen estados nuevos.
- Aplicar el primero de los operadores "supervivientes", con independencia de que ya haya sido aplicado en algún paso anterior.

Esta estrategia genera la siguiente trayectoria:

(0,0) -1- (8,0) -2- (8,6) -3- (0,6) -6- (6,0) -2- (6,6) -7- (8,4) -3- (0,4) -6- (4,0)

Ahora sí que hemos sido capaces de encontrar una solución correcta al problema planteado. Para ello, el "sistema", internamente, ha tenido que realizar las operaciones que se ilustran esquemáticamente en la Tabla 1.2.

ESTADO	OPERADOR	APLICABILIDAD	MOTIVOS	NUEVO ESTADO	SOLUCIÓN
(0,0)	OP1	SI	verifica restricciones	(8,0)	NO
(8,0)	OP1	NO	no verifica restricciones		
(8,0)	OP2	SI	verifica restricciones	(8,6)	NO
(8,6)	OP1	NO	no verifica restricciones		
(8,6)	OP2	NO	no verifica restricciones		
(8,6)	OP3	SI	verifica restricciones	(0,6)	NO
(0,6)	OP1	NO	verifica restricciones, pero estado ya generado		
(0,6)	OP2	NO	no verifica restricciones		
(0,6)	OP3	NO	no verifica restricciones		
(0,6)	OP4	NO	verifica restricciones, pero estado ya generado		
(0,6)	OP5	NO	no verifica restricciones		
(0,6)	OP6	SI	verifica restricciones	(6,0)	NO
(6,0)	OP1	NO	verifica restricciones, pero estado ya generado		
(6,0)	OP2	SI	verifica restricciones	(6,6)	NO
(6,6)	OP1	NO	verifica restricciones, pero estado ya generado		
(6,6)	OP2	NO	no verifica restricciones		
(6,6)	OP3	NO	verifica restricciones, pero estado ya generado		

Tabla 1.2 Tabla de operaciones realizadas internamente por el sistema para la estrategia de resolución número 4 (continúa en página siguiente)

ESTADO	OPERADOR	APLICABILIDAD	MOTIVOS	NUEVO ESTADO	SOLUCIÓN
(6,6)	OP4	NO	verifica restricciones, pero estado ya generado		
(6,6)	OP5	NO	no verifica restricciones		
(6,6)	OP6	NO	no verifica restricciones		
(6,6)	OP7	SI	verifica restricciones	(8,4)	NO
(8,4)	OP1	NO	no verifica restricciones		
(8,4)	OP2	NO	verifica restricciones, pero estado ya generado		
(8,4)	OP3	SI	verifica restricciones	(0,4)	NO
(0,4)	OP1	NO	verifica restricciones, pero estado ya generado		
(0,4)	OP2	NO	verifica restricciones, pero estado ya generado		
(0,4)	OP3	NO	no verifica restricciones		
(0,4)	OP4	NO	verifica restricciones, pero estado ya generado		
(0,4)	OP5	NO	no verifica restricciones		
(0,4)	OP6	SI	verifica restricciones	(4,0)	SI

Tabla 1.2 Tabla de operaciones realizadas internamente por el sistema para la estrategia de resolución número 4 (continuación)

El motivo por el cual hemos ilustrado de esta manera el hipotético proceso que el sistema ha seguido para encontrar la solución, es tratar de evidenciar las diferencias existentes entre los programas convencionales y los programas de IA. Nótese que hemos definido de manera totalmente independiente el "conocimiento del sistema" (i.e., los operadores y sus precondiciones), y la forma de utilizar este conocimiento (i.e., las "estrategias", también llamadas "mecanismos de control del conocimiento"[7]). De hecho,

[7] Aunque no es estrictamente cierto que "estrategia" y "mecanismo de control del conocimiento" sean términos equivalentes, por el momento asumiremos que sí lo son. Más adelante se discutirá la diferencia entre ambos conceptos.

esta misma estrategia podría haber sido utilizada con cualquier conjunto de operadores en cualquier dominio, ya que las instrucciones son de naturaleza completamente general.

Ciertamente estamos dando macroinstrucciones a nuestro programa, y es él, con su "conocimiento", el que se encarga de encontrar la solución (mejor dicho, una solución que satisfaga un conjunto de requisitos). También es cierto que, a medida que vamos explorando diversas estrategias de resolución, nos vamos encontrando con la necesidad de definir estructuras auxiliares que nos ayuden a comprobar si un estado ha sido generado ya, o si un operador ha sido aplicado con anterioridad.

Por último, nótese que la solución encontrada por el sistema es "aceptable", en el sentido de que efectivamente cumple los requisitos para ser "meta". Sin embargo, frente a la solución encontrada "a mano", el sistema ha necesitado explorar 9 estados y efectuar 8 operaciones, dos más de los estrictamente necesarios. El sistema ha sido capaz de encontrar una solución, pero esta solución no es la mejor. La validez de la solución encontrada dependerá mucho del tipo de problema planteado... ¿qué ocurriría si en lugar de agua tuviéramos mercurio?[8]

Las distintas estrategias de búsqueda que se presentan en este capítulo tratarán de resolver problemas de este tipo.

1.2. Características Generales de los Procesos de Búsqueda

Ya hemos mencionado que los programas de IA deben ser flexibles y generales de forma que, independientemente del universo de discurso, permitan la utilización de técnicas aplicables a la resolución de cualquier problema, y que sean de una eficiencia, por lo menos, aceptable. Surgen así las llamadas *técnicas de búsqueda de propósito general*, conocidas también con el nombre de *métodos débiles de exploración del espacio de estados*. Estas técnicas derivan de la idea de búsqueda heurística, y pueden definirse independientemente de cualquier tarea particular, o del dominio concreto considerado[9].

En la literatura podemos encontrar diversas técnicas de búsqueda de propósito general, cada una con sus ventajas, inconvenientes, e idiosincrasia particular. Pero antes de decidirnos por una u otra técnica, es conveniente estudiar su idoneidad en relación al tipo de problema y dominio planteado (ver nota a pie de página número 9). Esta idoneidad puede establecerse analizando un conjunto de cinco características esenciales que condicionan el proceso de búsqueda:

---

[8] El precio del mercurio es muy elevado, y su densidad es de 13.6 gr/cc, por lo que en 1 litro caben 13.6 Kg de mercurio. ¿Podríamos permitirnos el lujo de "tirar a la basura" 108.8 Kg ó 81.6 Kg de líquido metal, dependiendo del cubo que vaciásemos?

[9] Evidentemente, esta afirmación no debe tomarse "al pie de la letra". Claramente, el tipo de tarea a resolver, y el dominio del problema, influyen en la elección de una u otra técnica. En cualquier caso, aunque conceptualmente todas estas técnicas son igualmente aplicables, unas son más apropiadas que otras.

- Dirección del proceso de búsqueda
- Topología del proceso de búsqueda
- Representación de los estados por los que discurre la resolución del problema
- Criterios establecidos y el procedimiento definido para la selección sistemática de los operadores relevantes en función de los estados alcanzados
- Posibilidad de optimizar los procesos de búsqueda mediante el empleo de *funciones heurísticas*

Dirección del proceso de búsqueda

Existen dos direcciones fundamentales que podemos definir a la hora de configurar un proceso de búsqueda determinado:

- Desde los estados iniciales hacia los estados meta, mediante la generación de estados intermedios obtenidos tras la aplicación sucesiva de operadores relevantes
- Desde los estados meta hacia los estados iniciales, investigando qué estados previos al estado (o estados) meta, y qué operadores aplicables, nos producen una transición deseada

En cierto tipo de sistemas, que serán tratados más adelante, la primera dirección definida (i.e., desde los estados iniciales hacia los estados meta), configura un *razonamiento progresivo* o *dirigido por los datos*, mientras que la segunda (i.e., desde los estados meta hacia los estado iniciales), configura un *razonamiento regresivo* o *dirigido por los objetivos*.

La importancia de elegir una u otra dirección para organizar nuestra búsqueda se pone de manifiesto en el siguiente ejemplo:

"Supongamos que hemos quedado con un amigo para cenar en su casa, a la cual no sabemos cómo llegar, aunque tenemos su dirección. Tratando de ahorrarnos el taxi, le llamamos por teléfono para que nos dé unas someras explicaciones sobre el trayecto que debemos seguir para llegar a pie. Como no conocemos el barrio, y además las explicaciones de nuestro amigo son más bien confusas, hacemos de tripas corazón y decidimos, por fin, llamar al taxi. Después de cenar opíparamente, y dado que la noche es magnífica, tratamos de regresar a casa dando un paseo, para lo cual nuestro amigo nos marca unas cuantas referencias. Al cabo de un rato de recorrer las calles, y después de equivocar la ruta un par de veces, conseguimos finalmente entrar en nuestro portal".

El regreso a casa de nuestro ejemplo anterior es, claramente, un proceso de búsqueda dirigido por los objetivos. Parece siempre mucho más fácil llegar desde un lugar desconocido hasta nuestra casa, que desde nuestra casa hasta un lugar desconocido. Recorriendo el camino al revés, o simplemente tomando una ruta al azar,

suele ser corriente que encontremos alguna referencia conocida que nos indique, al menos, qué camino es el correcto, o simplemente dónde estamos.

El problema es netamente diferente cuando de lo que se trata es de, por ejemplo, diagnosticar una enfermedad. En este caso partimos de un conjunto de observaciones que nos permiten, por un lado, descartar un conjunto de posibilidades iniciales, y por otro, establecer un conjunto de hipótesis, compatibles con las observaciones iniciales, que deberemos investigar para tratar de localizar la enfermedad en cuestión. Este tipo de búsqueda es, claramente, un proceso dirigido por los datos.

Entre ambas situaciones límite, encontramos ciertos tipos de problemas para los cuales es conveniente emplear estrategias mixtas de búsqueda. Así, en algunos casos, es conveniente iniciar un proceso dirigido por los datos y, llegado a un punto, cambiar la dirección de la búsqueda, o viceversa (i.e., iniciar un proceso dirigido por los objetivos, que permita el establecimiento de un conjunto de hipótesis razonables, y luego confrontar las hipótesis con los datos, a través de un proceso progresivo).

De lo visto anteriormente, parece claro que la dirección del proceso condiciona, al menos en parte, los resultados y la eficiencia del sistema, pero... ¿cuándo es aconsejable optar por un proceso dirigido por los datos o por un proceso dirigido por los objetivos?, ¿existe algún criterio que nos permita discriminar entre ambas opciones?.

La elección sobre la dirección de búsqueda más conveniente debe considerar tres aspectos diferentes:

- Tamaño relativo de los conjuntos *I* y *M*
- Factor de ramificación
- Inclusión de estructuras explicativas como requisito inicial en el diseño de nuestro sistema inteligente

El tamaño relativo de los conjuntos *I* y *M* es fundamental a la hora de decidir qué dirección debemos seguir. Así, es preferible explorar el espacio de estados de forma que progresemos desde un conjunto inicialmente pequeño de información de partida, hacia un conjunto mayor de estados. La pregunta que debemos responder antes de decidir sobre la dirección del proceso es... ¿qué conjunto es mayor, el conjunto *I* de estados iniciales, o el conjunto *M* de metas?.

Un segundo aspecto importante es el factor de ramificación del proceso de búsqueda[10]. Este parámetro influye en la eficacia del proceso de búsqueda, de forma que siempre trataremos de explorar el espacio de estados según la dirección del menor factor de ramificación[11].

---

[10] Definimos *factor de ramificación* como el promedio de estados que podemos alcanzar directamente desde un estado previo.

[11] Ya que, cuanto menor sea el factor de ramificación, menor será el número de alternativas posibles que deberemos explorar.

Por último, si el programa debe ser capaz de "explicar" un proceso dado de razonamiento[12], es conveniente que el razonamiento del sistema se produzca en la dirección que concuerde más aproximadamente con la forma de razonar del usuario humano[13].

Topología del proceso de búsqueda

Una forma sencilla de explorar el espacio de estados es generar dinámicamente un árbol[14], partiendo de un determinado estado, inicial o final, y expandirlo tras la ejecución de uno, o varios, operadores relevantes[15]. La Figura 1.1 muestra un posible árbol relativo al problema de los dos cubos.

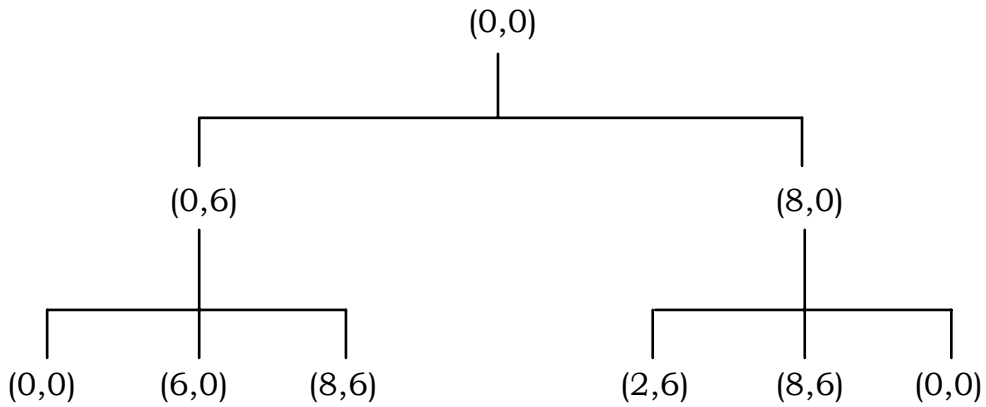


Figura 1.1 Un posible árbol relativo al problema de los dos cubos

En el árbol de la Figura 1.1 se observa que un mismo estado puede ser generado durante la exploración de diversos caminos. Ello supone un esfuerzo adicional de computación, que se traduce en una menor eficiencia del proceso de búsqueda. Este inconveniente puede mitigarse cambiando la topología del proceso, y convirtiendo el árbol en un grafo, tal y como se muestra en la Figura 1.2.

---

[12] Cuestión que no sólo es posible en los programas de IA, sino que a veces es exigible.

[13] De cualquier otra forma difícilmente podríamos conseguir una explicación adecuada, fácilmente comprensible e ilustrativa.

[14] Es importante la característica diferencial de generación dinámica del árbol. En IA los árboles de decisión son siempre implícitos, sólo se materializan cuando se ejecuta un determinado proceso inferencial.

[15] El número de operadores relevantes ejecutados depende de la técnica de exploración elegida, como veremos más adelante.

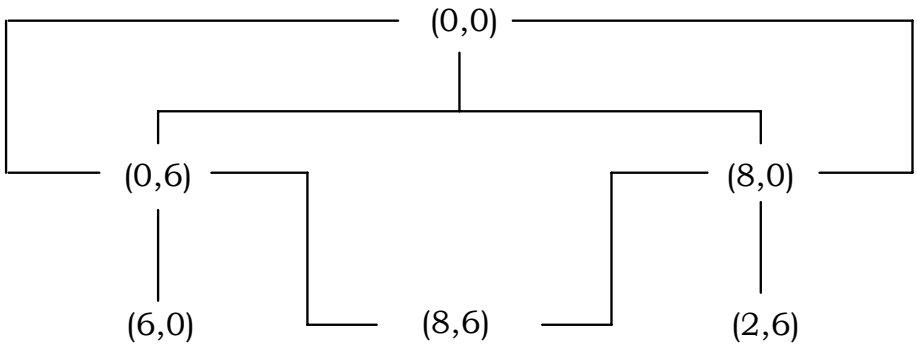


Figura 1.2 Grafo de búsqueda correspondiente al árbol de la Figura 1.1.

Para convertir un árbol de búsqueda en un grafo de búsqueda podemos utilizar el procedimiento que se describe a continuación[16]:

(a) Empezar generando (uno o más) estados, tras la aplicación de (operador u operadores) relevantes
(b) Examinar el conjunto de estados generados. Para cada uno de ellos...
    (b1) Si es nuevo, añadirlo y volver a (a)
    (b2) Si ya existía, descartarlo e ir a (c)
(c) Añadir un enlace entre el nodo que se está expandiendo y su sucesor
(d) Recorrer el nuevo camino desde el principio
    (d1) Si es más corto, insertarlo como mejor camino, y:
        - propagar el cambio
        - reorganizar el grafo si es necesario
        - volver a (a)
    (d2) Si no es más corto, simplemente volver a (a)

Este mecanismo de conversión es completo ya que, además de cambiar la topología del proceso de búsqueda, registra el mejor camino cada vez que se genera un nuevo estado.

Es claro que, en los procesos de resolución de problemas, la utilización de grafos de búsqueda reduce los esfuerzos de exploración del espacio de estados. Sin embargo, tiene el inconveniente de que obliga a comprobar si cada "nuevo" estado generado pertenece ya al conjunto de estados generados en pasos anteriores. Las topologías en árbol suelen causar problemas de memoria, pero la búsqueda puede ser más rápida. Por el contrario, las topologías en grafo, aparte de ser conceptualmente más correctas[17], minimizan los problemas de memoria. Sin embargo, al tener que efectuar comprobaciones frecuentes, la eficiencia del sistema puede disminuir. Ambos

[16] Para una discusión detallada sobre este punto, véase Rich en la Bibliografía, texto del cual los autores han recuperado el algoritmo correspondiente.

[17] ¿Para qué vamos a generar algo que ya existe?...¡Basta con saber cómo se puede hacer!

esquemas tienen ventajas e inconvenientes[18]. En último término, la elección de una u otra alternativa depende del dominio del problema planteado.

El problema de la representación

De acuerdo con las estructuras definidas en el espacio de estados, el problema de la *representación* puede estudiarse desde tres perspectivas diferentes:

- Representación de los objetos, entidades relevantes o hechos del dominio.
- Representación de las relaciones entre objetos, entidades relevantes o hechos del dominio.
- Representación de las secuencias de estados surgidas durante los procesos de búsqueda.

El primero de los puntos de vista anteriores tiene que ver con el modo de representación de los nodos, considerados como entidades del dominio de naturaleza estática. La segunda perspectiva se refiere al modo de representación de las estructuras que nos permiten transitar por el espacio de estados (i.e., los operadores). Por último, la tercera posibilidad trata de la representación de la estrategia y de los mecanismos de control necesarios para organizar convenientemente la búsqueda[19]. Aunque de las tres perspectivas barajadas la más relacionada con la búsqueda es la representación de las secuencias de estados, todas ellas están estrechamente relacionadas y, normalmente, la elección de determinados esquemas de representación para entidades y relaciones, suele condicionar el esquema de representación idóneo para las secuencias de estados.

Selección sistemática de operadores relevantes

Ya hemos comentado cómo la aplicación de un determinado operador sobre un estado dado produce un nuevo estado, pero... ¿cómo podemos reconocer, de entre el conjunto global de operadores potencialmente útiles, aquellos que realmente son aplicables a nuestro estado actual? Dicho de otra forma: dado un estado ¿cómo podemos extraer del conjunto global de operadores un subconjunto de operadores relevantes?

El problema planteado define lo que en inteligencia artificial se denomina *emparejamiento*[20], que constituye una de las tareas más costosas y lentas de los programas de IA. Existen varios tipos de emparejamiento, cada uno con sus ventajas e inconvenientes, y la elección del tipo de emparejamiento suele depender del esquema utilizado para representar el conocimiento. En este texto analizaremos dos grandes

[18] Los autores ha podido comprobar, en parte a través de las prácticas de sus alumnos, los enormes problemas de memoria que suelen causar las topologías en árbol.

[19] La representación de nodos y la representación de las relaciones entre nodos, definen el problema de la *representación del conocimiento*, que será tratado con cierto detalle un poco más adelante.

[20] Básicamente, el *emparejamiento* es el proceso de selección de operadores relevantes.

familias de modelos de emparejamiento: el emparejamiento literal y el emparejamiento con variables.

El emparejamiento literal implica realizar una búsqueda simple, a través de todos los operadores del conjunto "*O*", analizando las precondiciones de cada operador en el contexto del estado actual considerado, y extraer aquellos operadores que verifiquen dichas precondiciones[21].

El emparejamiento literal tiene varios problemas. Así, su eficiencia es muy sensible al número de operadores definidos en el conjunto "*O*". Por otra parte, los problemas realmente interesantes para la IA requieren la utilización de gran número de operadores, por lo que este método de emparejamiento suele ser intrínsecamente ineficiente en los problemas reales.

Otro punto débil del emparejamiento literal es que, para un estado particular, no siempre es evidente que un operador determinado sea aplicable. En otras palabras, a veces es difícil saber si un estado actual satisface o no las precondiciones del operador. Así, entre otros, se pueden dar casos en los que:

- El operador empareje completamente con el estado
- Las precondiciones del operador sean un subconjunto de la descripción del estado actual
- Las precondiciones del operador coincidan sólo parcialmente con la descripción del estado actual
- Las precondiciones del operador no coincidan con la descripción del estado actual

El emparejamiento literal es útil en dominios pequeños, en los que la exploración de patrones, costosa y lenta, se ve compensada por lo restringido del conocimiento involucrado.

El otro tipo de emparejamiento, el emparejamiento con variables, es de naturaleza no literal, y es especialmente útil cuando el problema que tratamos de resolver requiere una búsqueda extensa en la que haya variables involucradas. Ilustraremos el proceso con el siguiente ejemplo:

Hechos del dominio

HIJO (María , Juan)
HIJO (Juan , Pedro)
HIJO (Pedro , Tomás)
HIJA (Pedro , Rosa)
HIJA (Juan , Ana)
HIJA (Ana , Rosa)

[21] Los autores recomiendan que este tipo de emparejamiento sea visualizado con el ejemplo de los dos cubos, discutido anteriormente. En concreto, en este contexto, es particularmente ilustrativa la descripción de la estrategia número 4, que es la que conduce a la solución del problema planteado.

Operadores del dominio

op1:	HIJO (x, y)	AND	HIJO (y, z)	→	NIETO (*x*, *z*)
op2:	HIJA (x, y)	AND	HIJO (y, z)	→	NIETO (*x*, *z*)
op3:	HIJO (x, y)	AND	HIJA (y, z)	→	NIETA (*x*, *z*)

Consideremos que los hechos del dominio, tal y como se describen arriba, constituyen nuestro estado inicial. Pretendemos encontrar un estado meta que incluya los mismos hechos y, además, otro hecho que indique "quién es el nieto de Juan".

En este caso, nos interesa aplicar los operadores op1 u op2, puesto que son los que concluyen sobre la existencia de un nieto. La meta se obtendría inmediatamente con sólo sustituir *x* por Juan. Pero, para ello tendríamos que encontrar primero un *y* que verificase: HIJO (Juan, *y*) AND HIJO (*y*,*z*), para algún valor de *z*. Ahora, el proceso requiere que ejecutemos una de las acciones siguientes:

- comprobar a todos los hijos de Juan, y verificar que alguno de ellos tenga, a su vez, un hijo, o

- comprobar que, de todos los que tengan algún hijo, hay alguno que, a su vez, es hijo de Juan.

En este caso, si comenzamos investigando los hijos de Juan, y tratamos de comprobar cuál de ellos tiene un hijo, no será necesario ensayar demasiadas posibilidades. De todas formas, no siempre es fácil saber "a priori" qué predicado emparejará primero y, frecuentemente, encontraremos muchos valores que satisfagan los predicados por separado, pero muy pocos que satisfagan todos ellos.

También relacionado con el proceso de emparejamiento, y en el caso de que hayamos podido identificar más de un operador aplicable a nuestro estado actual, debemos ser capaces de poder elegir la utilización del operador que "a priori" nos ofrezca más garantías de éxito en la búsqueda de un camino adecuado hacia la meta. Un enfoque correcto de este problema, denominado *resolución de conflictos*, es fundamental en la construcción de sistemas inteligentes.

Aunque muchas veces los mecanismos idóneos de resolución de conflictos son dependientes del universo de discurso, del esquema de representación elegido, y del procedimiento de búsqueda empleado, también es cierto que toda estrategia de resolución de conflictos debe respetar los siguientes criterios generales:

- Si podemos evitarlo no aplicaremos operadores que ya hayan sido utilizados.

- Trataremos de aplicar primero operadores que emparejen con los hechos más recientemente incorporados[22] a la base de hechos[23] que describe nuestro estado actual.

[22] Lo cual nos obliga a ser capaces de identificar la secuencia temporal en la incorporación de hechos.

- Trataremos de aplicar primero operadores con precondiciones más restrictivas (i.e., operadores más específicos), que operadores de naturaleza más general.

- De no darse ninguna de las condiciones anteriores (i.e., operadores que no han sido utilizados todavía, y que emparejan con hechos incorporados al mismo tiempo, y que son igualmente específicos), seleccionar uno (o más) aleatoriamente[24].

Funciones heurísticas

Se conoce con el nombre de *función heurística*, a aquella función, de carácter numérico, que nos permite cuantificar el beneficio de una transición, efectuada en el espacio de estados del dominio del problema a resolver. Las funciones heurísticas resultan muy útiles a la hora de optimizar los procesos de búsqueda. Para ello intentan guiar la exploración del espacio de estados en la dirección más provechosa, sugiriendo el "mejor" camino a seguir cuando disponemos de varias alternativas. Volveremos más adelante sobre esta interesante cuestión.

1.3. Estrategias de Exploración del Espacio de Estados

La eficacia del proceso de búsqueda viene frecuentemente determinada por la estrategia empleada y por los mecanismos diseñados para controlar la aplicación del conocimiento del dominio.

Es claro que, para cada universo de discurso, siempre podremos establecer estrategias específicas de exploración del espacio de estados. Hablamos, en este caso, de *técnicas de búsqueda de propósito específico*. No obstante, desde la perspectiva de la inteligencia artificial, son mucho más interesantes las estrategias genéricas[25], que unen a su generalidad una eficiencia razonable. Tales estrategias genéricas se agrupan en lo que habitualmente se denominan *métodos débiles de exploración del espacio de estados*.

Cualquier método débil de exploración del espacio de estados configura una búsqueda que será de uno de los siguientes tipos:

- En anchura

- En profundidad

- Mixta profundidad-anchura

---

[23] De forma sencilla, la base de hechos constituye el esqueleto declarativo del dominio de discurso correspondiente.

[24] Ya hemos mencionado que el número de operadores que debemos aplicar depende de la estrategia de resolución elegida.

[25] Y por lo tanto independientes del dominio.

Búsqueda preferente por amplitud

La búsqueda preferente por amplitud o búsqueda en anchura trata de generar amplios y crecientes segmentos en el espacio de estados, y en cada nuevo nivel generado, se verifica si el objetivo ha sido alcanzado, antes de pasar a generar el siguiente nivel. La característica fundamental de una búsqueda en anchura es que se expanden todos los nodos de un nivel antes de acceder a nodos de niveles inferiores.

La implementación computacional del método en anchura utiliza dos listas de nodos:

- ABIERTOS: nodos que se han generado, y a los que se les ha aplicado la función de evaluación, pero que aún no han sido examinados (no se han generado sus sucesores)

- CERRADOS: nodos que ya han sido examinados. Es una lista útil con el fin de evitar la creación de ciclos en el proceso de búsqueda.

Cada nodo *n* del árbol de búsqueda, mantendrá enlaces a todos sus sucesores, pero sólo a un único predecesor. El algoritmo de búsqueda en anchura es el siguiente:

Búsqueda_en_anchura

1. Colocar el nodo inicial en la lista de ABIERTOS
2. Aplicar el test de realización a este nodo. Si es un nodo meta, salir e informar de la solución.
3. Si la lista de ABIERTOS está vacía, informar del fallo y terminar
4. Eliminar el primer nodo de ABIERTOS y añadirlo a CERRADOS. Llamar a este nodo *N*.
5. Expandir *N* generando todos sus sucesores mediante la aplicación de todos los operadores relevantes. Si no hay sucesores volver a 3. En caso contrario añadir los sucesores al *final* de la lista de ABIERTOS y actualizar sus enlaces paternos para que apunten a *N*.
6. Aplicar a los sucesores el test de realización. Si alguno de ellos es un nodo meta, salir e informar de la solución siguiendo los enlaces que llevan al nodo inicial.
7. Volver a 3.

Para ilustrar este tipo de búsqueda consideremos el siguiente espacio de estados:

Conjunto *I* de estados iniciales: [*A*]
Conjunto *M* de estados meta: [*W*, *F*, *K*]
Conjunto *O* de operadores : [op1: *B* → *D*,
op2: *P* → *T*,
op3: *A* → *B*,
op4: *B* → *Y*,
op5: *D* → *L*,
op6: *D* → *Z*,
op7: *H* → *V*,

op8: $A \rightarrow G$,
op9: $P \rightarrow N$,
op10: $B \rightarrow N$,
op11: $P \rightarrow S$,
op12: $Q \rightarrow Y$,
op13: $E \rightarrow F$,
op14: $G \rightarrow P$,
op15: $D \rightarrow R$,
op16: $Y \rightarrow W$
]

Una búsqueda en anchura sobre este espacio de estados genera el árbol de la Figura 1.3.

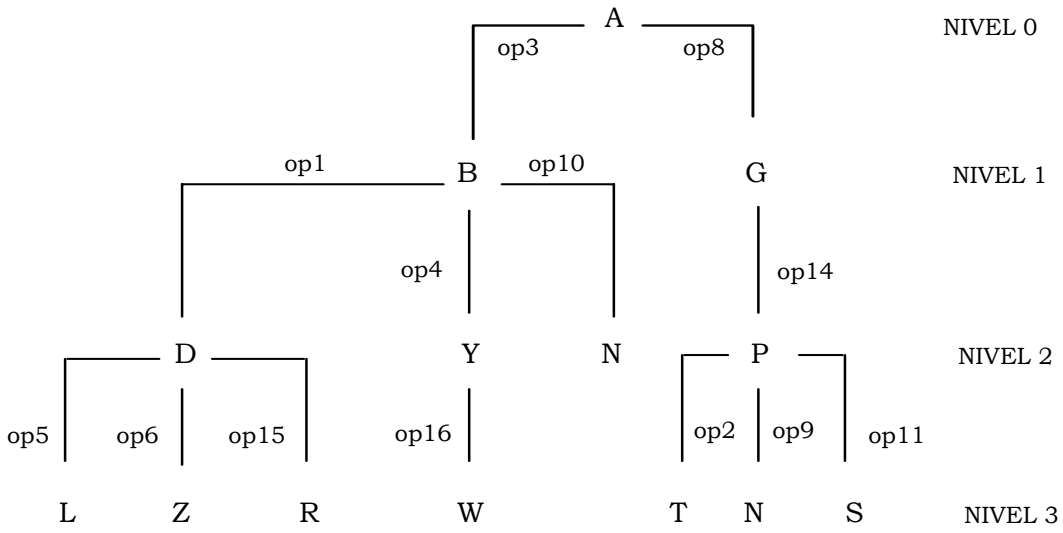


Figura 1.3 Un árbol generado por una búsqueda en anchura

En el ejemplo anterior, la solución "aceptable" que hemos encontrado está representada por el camino siguiente:

A -op3- *B* -op4- *Y* -op16-*W*

No obstante, para encontrar la solución, también hemos tenido que explorar los caminos:

A -op3- *B* -op1- *D* -op5- *L*
A -op3- *B* -op1- *D* -op6- *Z*
A -op3- *B* -op1- *D* -op15- *R*
A -op3- *B* -op10- *N*
A -op8- *G* -op14- *P* -op2- *T*
A -op8- *G* -op14- *P* -op9- *N*
A -op8- *G* -op14- *P* -op11- *S*

Los procedimientos de búsqueda en anchura, por ser exhaustivos, nos permiten asegurar que, en espacios de estados finitos y bien construidos, el sistema siempre encontrará la solución al problema planteado[26]. Nótese que estos métodos son sistemáticos, ya que se aplican todos los operadores que pueden emparejarse con todos y cada uno de los nodos del nivel considerado.

Desgraciadamente, los métodos en anchura son impracticables en dominios amplios, ya que el número de nodos generado en cada nivel sucesivo crece de manera exponencial y, consecuentemente, las necesidades de memoria, y el tiempo computacional empleado en la búsqueda, también crecen exponencialmente.

Búsqueda preferente por profundidad

A diferencia de los métodos en anchura, los métodos en profundidad seleccionan un camino determinado y siguen por él hasta agotarlo completamente. En los métodos en profundidad "puros", el test de realización se efectúa cada vez que se genera un nuevo nodo. Puede ocurrir que el camino recorrido sea resolutivo y lleve a la solución del problema o, por el contrario, que agotemos todas las posibilidades de expansión sin haber encontrado nada. En este último caso hay que efectuar una "vuelta atrás" y explorar otro camino diferente. Un algoritmo de búsqueda en profundidad es el siguiente:

Búsqueda_en_profundidad (N)

1. Colocar el nodo inicial *N* en la lista CAMINO.
2. Aplicar el test de realización a este nodo. Si es un nodo meta, salir.
3. Expandir *N* aplicando el primer operador no aplicado para generar un sucesor S.
4. Si ningún operador es aplicable, salir.
5. En caso contrario, realizar una búsqueda en profundidad partiendo de S (llamada al procedimiento Búsqueda_en_profundidad(S)).

La ruta con la solución queda almacenada en la lista CAMINO.

Como se puede observar los métodos en profundidad son sensibles a la "posición relativa" de los operadores en una lista, ya que este es el criterio que se utiliza para la selección de operadores en caso de conflicto.

La Figura 1.4 muestra el árbol obtenido tras una expansión en profundidad con los datos del ejemplo anterior. El criterio seguido para seleccionar qué operador debemos aplicar en cada momento es el de "el primero de la lista".

[26] Al menos teóricamente. En realidad, y por las razones expuestas en el texto, estos procedimientos rara vez son aplicables en problemas reales y, en todo caso, su aplicación se traduce en búsquedas ineficientes que consumen grandes recursos computacionales y de memoria.

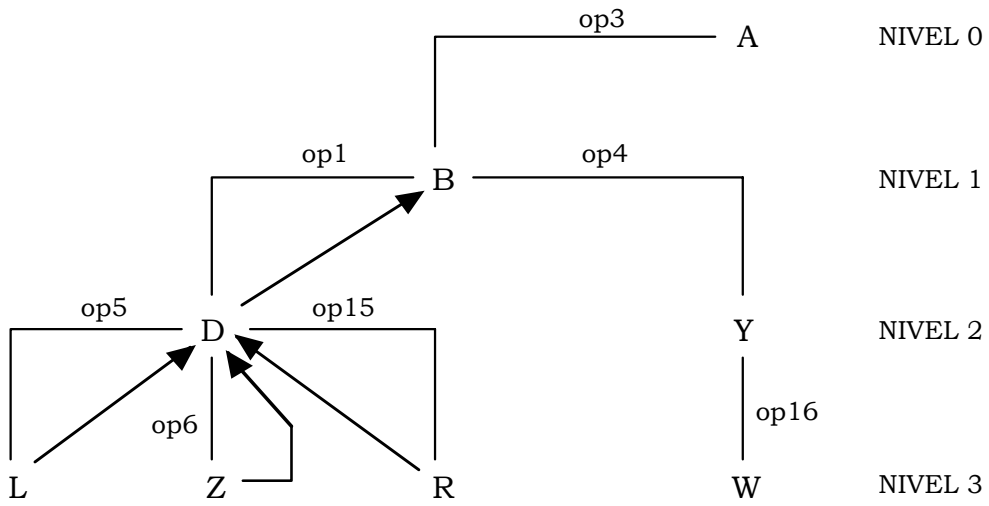


Figura 1.4 Árbol generado tras un proceso de búsqueda en profundidad

En ocasiones, los caminos generados por los métodos en profundidad no se exploran completamente, abandonándose la búsqueda tras alcanzarse sin éxito una determinada profundidad. Ello puede suponer que consideremos que un determinado camino es infructuoso cuando, en realidad, estamos muy cerca de la solución.

La búsqueda en profundidad "pura", al igual que la búsqueda en anchura, es un procedimiento sistemático, pero permite que encontremos la solución "por casualidad". Este aspecto se enfatiza si, tras cada expansión, reordenamos los operadores aplicables aleatoriamente.

Desde una perspectiva computacional, los métodos en profundidad demandan menos recursos de memoria que los métodos en anchura, ya que consideran un espacio de búsqueda más limitado. Tienen, sin embargo, el inconveniente de que pueden tratar de explorar caminos muy largos teniendo otras alternativas mejores, e incluso pueden no llegar nunca a la solución del problema[27].

Aunque puede intentarse la utilización de métodos puros de exploración del espacio de estados, lo normal es emplear estrategias mixtas, y desarrollar procedimientos que combinen, de una u otra forma, características de los métodos en anchura y en profundidad. En la literatura podemos encontrar gran cantidad de tales métodos. Aquí, describiremos los siguientes:

[27] Por ejemplo, si interrumpimos un camino en un nivel de profundidad determinado.

- Generación y prueba
- Ascensión a colinas
- Búsqueda por el mejor nodo: A*, Agendas.
- Búsqueda en grafos YO: Reducción de problemas

Generación y prueba

El método de *generación y prueba* es un procedimiento de búsqueda en profundidad casi puro, en el que deben recorrerse caminos completos antes de realizar ninguna comprobación. En este caso el test de realización se limita a responder afirmativa o negativamente acerca de la validez del camino ensayado. En su forma más sistemática, el método de generación y prueba es una búsqueda exhaustiva en el espacio del problema. El esquema del método es el siguiente:

Generación_y_prueba

1. Generar una solución posible. Para algunos problemas, esto significa generar un punto específico en el espacio del problema. Para otros, significa generar un camino desde el estado inicial.
2. Aplicar el test de realización al punto elegido, o al punto final del camino elegido.
3. Si hemos encontrado la solución, parar. En caso contrario, volver al paso 1.

Con el procedimiento de generación y prueba, si la generación de caminos se hace sistemática y exhaustivamente, siempre encontraremos la solución, si existe. Sin embargo, si el dominio del problema es muy amplio, la exploración del espacio de estados mediante este método puede requerir demasiado tiempo.

Los procedimientos de búsqueda ensayados hasta ahora tienen una característica común: forman parte de las denominadas *búsquedas ciegas*. Con ellas aparecen, como ya hemos visto, algunos problemas. En IA, parte de estos problemas se pueden resolver empleando las denominadas *técnicas heurísticas* de búsqueda, que suelen mejorar la eficiencia de los procesos de resolución de problemas, sacrificando frecuentemente la exhaustividad de la respuesta, dejando de considerar algunos caminos que parece improbable que conduzcan a la solución.

Las *estrategias de búsqueda informada* tratan de optimizar los procesos de búsqueda, utilizando funciones heurísticas que les permiten guiar la exploración del espacio de estados en la dirección más provechosa, sugiriendo el “mejor” camino a seguir cuando disponemos de varias alternativas.

Más sobre funciones heurísticas

Las *funciones heurísticas* son funciones de carácter numérico, computables eficientemente, que incorporan conocimiento específico del problema para estimar el coste de alcanzar una meta desde un estado dado. Generalmente, se simbolizan con la letra *h*. Desde el punto de vista formal, *h* podrá ser cualquier función, el único requisito es que $h(n)=0$ cuando *n* sea un estado meta.

Sea $h^*(n)$ la función que devuelve el coste **real** de un camino de coste mínimo desde el estado *n* hasta el estado meta. Una heurística *admisible* es aquella que nunca sobrestima $h^*(n)$, es decir,

$$\forall n \; h(n) \leq h^*(n)$$

Para la mayoría de los algoritmos de búsqueda informada y, concretamente para los que veremos en este capítulo, se exige además a la función heurística la *restricción de monotonía*. Se dice que una función heurística *h* satisface esta restricción si cualesquiera que sean los nodos n_i, n_j, tales que n_j es un sucesor de n_i, se cumple que

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

siendo $c(n_i, n_j)$ el coste de recorrer el camino que nos lleva de n_i a n_j.

El problema del 8-puzzle fue uno de los primeros problemas resueltos mediante búsqueda heurística. Es uno de los llamados *problemas juguete*, utilizados para investigar la eficacia y la eficiencia de los algoritmos de búsqueda[28]. El 8-puzzle es un tablero de 3x3 casillas con 8 fichas numeradas del 1 al 8, y una casilla vacía. El problema consiste en deslizar las fichas horizontal o verticalmente y colocarlas en el espacio vacío. El objetivo del problema es conseguir que el puzzle esté ordenado según algún criterio prederteminado, partiendo de una posición inicial cualquiera. La Figura 1.5 muestra una representación esquemática del problema, en la cual se observa una configuración inicial y una configuración meta deseada.

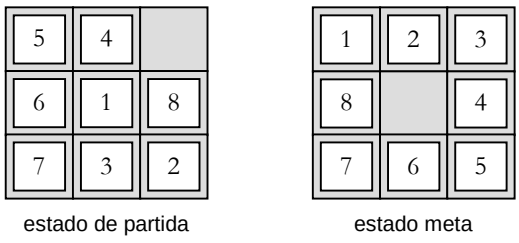


Figura 1.5 Ejemplo típico del ejemplo del 8-puzzle.

La búsqueda de la solución a este problema mediante métodos no informados genera una gran cantidad de estados, por lo que procede encontrar una buena función heurística. Las dos siguientes serían buenas candidatas:

[28] Otros problemas de este tipo son los siguientes: 8 reinas, misioneros y caníbales, 2 cubos, etc. En la bibliografía recomendada se pueden encontrar soluciones a estos problemas.

- h_1=número de piezas que están en lugar incorrecto. En la figura 2.5 ninguna de las 8 piezas está en la posición correcta, por lo que en el estado de partida h_1=8. h_1 es una heurística aceptable (es admisible, se hace 0 en la meta y es monótona) puesto que es evidente que cuando una pieza está en un lugar incorrecto habrá que realizar, por lo menos, un movimiento.

- h_2=sumas de las distancias que separa a las piezas de sus posiciones meta. Puesto que las piezas no pueden desplazarse en diagonal, esta distancia se obtendrá como la suma de las distancias verticales y horizontales. A esta distancia se le conoce como *distancia de Manhattan*. Se puede comprobar que h_2 también es aceptable. La configuración de partida tiene una distancia de Manhattan de

$$h_2 = 2+3+2+1+2+2+1+2 = 15$$

Evidentemente, cuanto más precisa sea la estimación que la función heurística realiza sobre los correspondientes nodos del espacio de estados, más directa debería ser la resolución del problema. Reduciendo al absurdo el problema, la mejor función heurística sería la que nos permitiese decidir un camino concreto hacia la solución sin tener que efectuar ninguna búsqueda.

Una fuente de heurísticas admisibles se encuentra a partir de la *abstracción* de problemas. De forma intuitiva, una abstracción de un problema es una transformación del mismo en la que se eliminan ciertos detalles del problema original. Formalmente, sea el problema de búsqueda una terna (S, c, G), donde S es el conjunto de estados que describen las situaciones del mundo; $c: S\text{x}S \rightarrow R$ es una función de coste positiva que representa el coste de aplicar la acción correspondiente para transitar de un estado a otro, y $G \subseteq S$ es el conjunto de estados meta. Una función ϕ: $S \rightarrow S'$es una abstracción del problema (S, c, G) al problema (S', c', G') si y solo si se cumplen las dos condiciones siguientes:

- ϕ reduce costes: $(\forall s,t \in S)\ \ c'(\phi(s), \phi(t)) \leq c(s,t)$

- ϕ expande metas: $(\forall g \in G)\ \phi(g) \in G'$.

Las funciones que nos proporcionan el coste preciso de la ruta en abstracciones del problema resultan ser heurísticas admisibles en el problema original.

Un ejemplo de abstracción es aquella que, en el problema original del 8-puzzle, permite que las fichas se puedan mover a cualquier posición adyacente, de forma que

- S' es el conjunto de todas las configuraciones posibles del puzzle estando permitido el deslizamiento de unas piezas sobre otras,

- $c'(s, t)$=1, si podemos pasar del estado s al estado t moviendo una de las piezas a una posición adyacente aún cuando ésta no este vacía, y $c'(s, t)= \infty$ en otro caso, y

- G' es igual a G pero ignorando la posición de la casilla vacía, es decir, sólo se impone que las piezas sigan un cierto orden.

Esta transformación es una abstracción ya que reduce costes y expande metas a partir del problema original. En este caso, h_2 proporciona la cantidad exacta de pasos a la solución en la abstracción del problema original que acabamos de realizar. Si cambiamos de nuevo las reglas del problema permitiendo que una pieza se pueda desplazar a cualquier posición aunque no sea adyacente, la cantidad exacta de pasos nos la daría h_1.

Evidentemente, cuanto mayor sea la abstracción del problema, menor será la precisión de la heurística derivada. Así se puede observar que h_2 siempre va a proporcionar estimaciones más precisas que h_1.

En la literatura se pueden encontrar otras aproximaciones para la generación de heurísticas tales como el uso de información estadística.

A continuación veremos distintos métodos de búsqueda informada que hacen uso de las funciones heurísticas.

Ascensión a colinas

El procedimiento de *ascensión a colinas* es una variante del método de profundidad en el que la selección del siguiente nodo a expandir se realiza de acuerdo con alguna medición heurística que permite estimar la distancia que queda por recorrer hasta la meta. En su forma más sencilla, el algoritmo de la ascensión a colinas es el siguiente:

Ascensión_a_colinas_simple (N)

1. Colocar el nodo inicial N en la lista CAMINO.
2. Aplicar el test de realización a este nodo. Si es un nodo meta, salir.
3. Repetir hasta encontrar un sucesor
   3.1. Aplicar a N un operador no aplicado para generar un sucesor S.
   3.2. Si no hay ningún operador aplicable, salir.
   3.3. En caso contrario, aplicar la función heurística a S.
   3.4. Si mejora la función heurística del estado actual, ir a 4.
   3.5. Si no volver a 3.1
4. Continuar el proceso de búsqueda partiendo de S
   (llamada al procedimiento Ascensión_a_colinas_simple (S))

La Figura 1.6 ilustra un ejemplo de este algoritmo. En este caso cada nodo se ha caracterizado por un valor de función heurística. Se puede observar, que a diferencia de la búsqueda en profundidad, no se explora completamente una rama hasta agotarla, sino que el proceso de expansión termina en el momento en que se encuentra un nodo sucesor que no mejora el *estado* actual. Este es el caso de los nodos *B*, *T* y *N*.

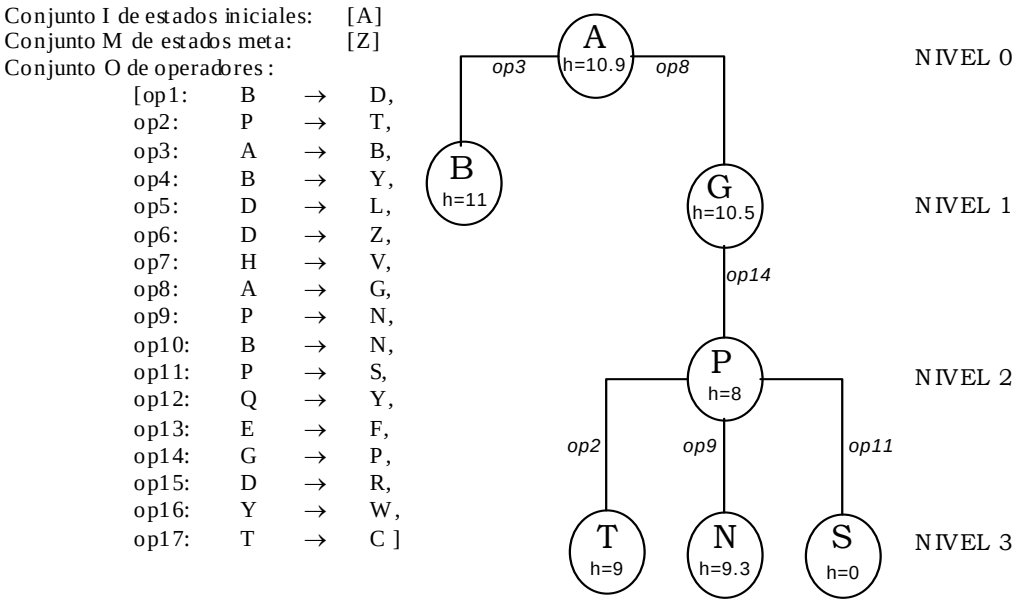


Figura 1.6 Un árbol generado por ascensión a colinas simple

La solución se encuentra a través del camino: *A* -op8- *G* -op14- *P* -op11- *S*

Una variante útil del método de ascensión simple, consiste en considerar todos los posibles movimientos a partir del estado actual, y elegir el mejor de ellos como el nuevo estado. Este método se denomina método de *ascensión por la máxima pendiente* o *búsqueda del gradiente*. Nótese el contraste con el método básico, en el que el primer estado que parezca ser mejor que el actual se selecciona como estado actual. El algoritmo funcionaría así:

Ascensión_por máxima_pendiente (N)

1. Colocar el nodo inicial *N* en la lista CAMINO.
2. Aplicar el test de realización a este nodo. Si es un nodo meta, salir.
3. Aplicar a *N* todos los operadores disponibles y generar todos sus sucesores S_i.
4. Si no hay ningún operador aplicable, salir.
5. En caso contrario, aplicar la función heurística a todos los S_i.
6. Si la función heurística de algún S_i mejora la función heurística del estado actual, llamar a este nodo MEJORNODO e ir a 8
7. En caso contrario, salir.
8. Continuar el proceso de búsqueda partiendo de MEJORNODO (llamada a Ascensión_por_máxima_pendiente (MEJORNODO))

Tanto la ascensión a colinas básica como la de máxima pendiente pueden no encontrar una solución si caen en un estado del que no es posible generar nuevos estados mejores que él. Esto ocurre cuando el proceso de búsqueda se encuentra con un máximo local, una meseta o una cresta. Considerando el espacio de estados como una

superficie n-dimensional en la que cada punto está definido por los valores que definen ese estado y el valor de la función heurística de ese estado[29], se define:

- Un *máximo local* es un estado puntual mejor que cualquiera de sus vecinos, pero peor que otros estados más lejanos. Llegados a un punto de estas características, cualquier operación que hagamos, cualquier movimiento que intentemos, nos llevará a un estado aparentemente peor, aunque en realidad nos estemos aproximando a la solución. Cuando los máximos locales aparecen cerca de la solución final se denominan *estribaciones*.

- Una *meseta* es una área plana del espacio de estados en la que todos los estados individuales tienen un mismo valor de la función heurística. En las mesetas no es posible determinar cuál es la mejor dirección para continuar la búsqueda.

- Una *cresta* es un tipo especial de máximo local. Es un área del espacio de estados, que tiene estados con mejores valores de la función heurística que los de regiones colindantes, y además posee una inclinación, pero la orientación de esta región alta hace que sea imposible atravesar la cresta mediante transiciones simples.

En la Figura 1.7 y en la Figura 1.8 se representan estas situaciones. Las superficies se corresponden con el espacio de estados para un problema de búsqueda en el que los estados puedan representarse mediante dos variables *x* e *y*. El valor de la calidad de un nodo – su cercanía a la meta– toma valores en el eje *Z*, y los sucesores de un estado dado, se localizan en las posiciones adyacentes.

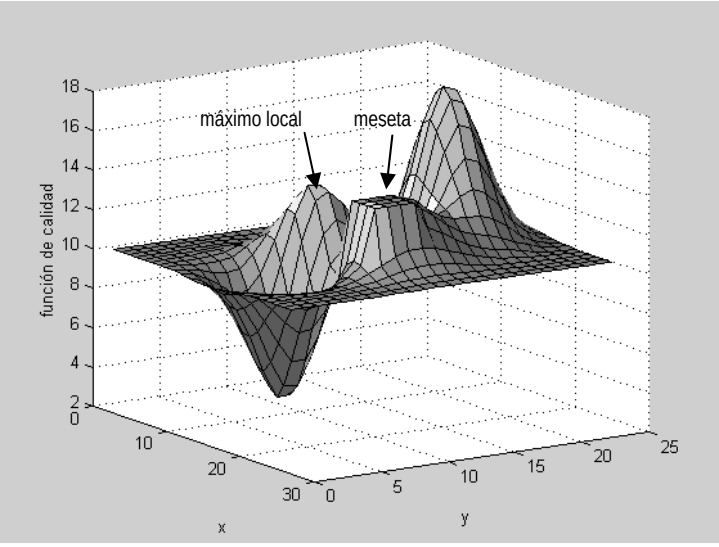


Figura 1.7 Representación de un "máximo local" y una "meseta", en el método de búsqueda de ascensión a colinas

[29] En el ejemplo de los jarros de agua, cada punto de la superficie de búsqueda está definido por el contenido del cubo de 8 litros C_8, el cubo de 6 de litros C_8 y el valor de la función heurística para el estado (C_8, C_6)

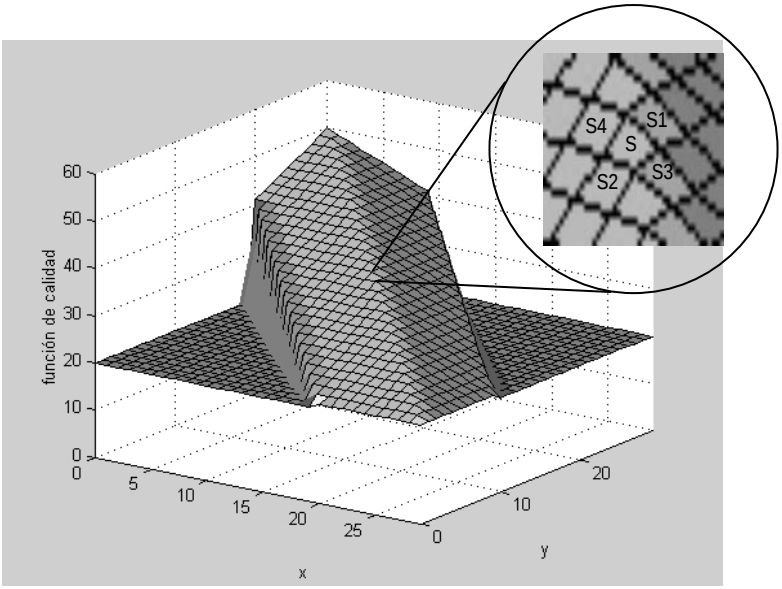


Figura 1.8 Representación de una "cresta", en el método de búsqueda de ascensión a colinas (S: estado actual, S1, S2, S3 y S4: estados sucesores de S)

Estos inconvenientes del método tienen difícil solución. Como norma general, cuando aparece alguna de estas situaciones se pueden intentar las siguientes estrategias:

- Regresar a un nodo previo e intentar una dirección diferente puede resolver el problema de los máximos locales.

- Realizar un gran salto en el espacio de búsqueda, para tratar de alcanzar una región diferente del espacio de estados, puede resolver el problema de las mesetas.

- Aplicar más de un operador antes de comparar los valores de la función heurística puede resolver el problema de las crestas.

*Búsqueda por el mejor nodo: A**

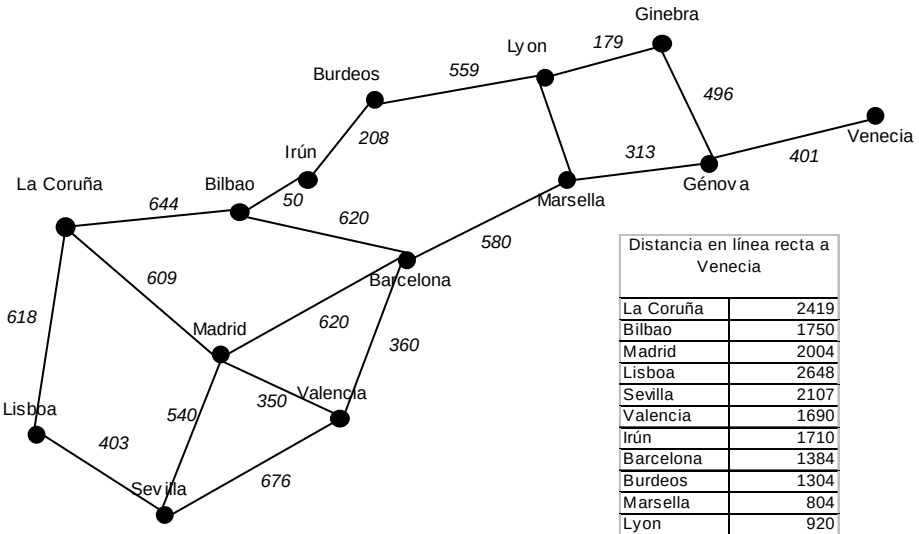
Los métodos de búsqueda por el mejor nodo combinan algunas de las ventajas de los métodos en profundidad y en anchura. Básicamente, se trata de organizar una búsqueda mixta profundidad-anchura guiada hacia el nodo más prometedor, con independencia de la rama del árbol a la que pertenezca el nodo en cuestión. A esta familia de métodos pertenecen el algoritmo A* y las Agendas que veremos a continuación.

El algoritmo A* trata de expandir el nodo más cercano a la meta, de entre los nodos que se encuentran en las rutas menos costosas que parten del estado inicial. Para ello, utiliza una función de evaluación *f* definida como

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

donde $g(n)$ es el coste de la ruta que va del nodo de partida al nodo actual n, calculada como la suma de los costes de cada una de las acciones individuales que se emprenden a lo largo de la ruta; $h(n)$ es la *función heurística*, que proporciona una estimación del coste mínimo adicional de llegar desde el nodo actual n al nodo meta. Nótese que g no es una estimación de nada; se sabe exactamente su valor.

El algoritmo A* resulta útil, por ejemplo, para hallar una ruta entre dos ciudades que minimice el coste del viaje. En la Figura 1.9 se presenta el grafo de distancias entre algunas ciudades europeas. El objetivo es encontrar una ruta lo más óptima posible desde La Coruña hasta Venecia, utilizando como algoritmo de búsqueda A*.



Distancia en línea recta a Venecia	
La Coruña	2419
Bilbao	1750
Madrid	2004
Lisboa	2648
Sevilla	2107
Valencia	1690
Irún	1710
Barcelona	1384
Burdeos	1304
Marsella	804
Lyon	920
Genova	370
Ginebra	673

Figura 1.9 Distancias entre ciudades europeas

En la Figura 1.10 se pueden observar las etapas de la búsqueda, donde los nodos se identifican con $f = g + h$. Los valores de la función g se obtienen directamente del grafo de distancias. Una buena función heurística para problemas de determinación de ruta como este, es la distancia en línea recta a la meta. Es decir,

$h(n)$ = distancia en línea recta entre n y la ubicación de la meta

cuyos valores aparecen reflejados en la tabla de la Figura 1.9. Se puede observar que ésta es una heurística aceptable ya que generalmente una carretera que va de *A* a *B* tiende a hacerlo de la forma más directa posible.

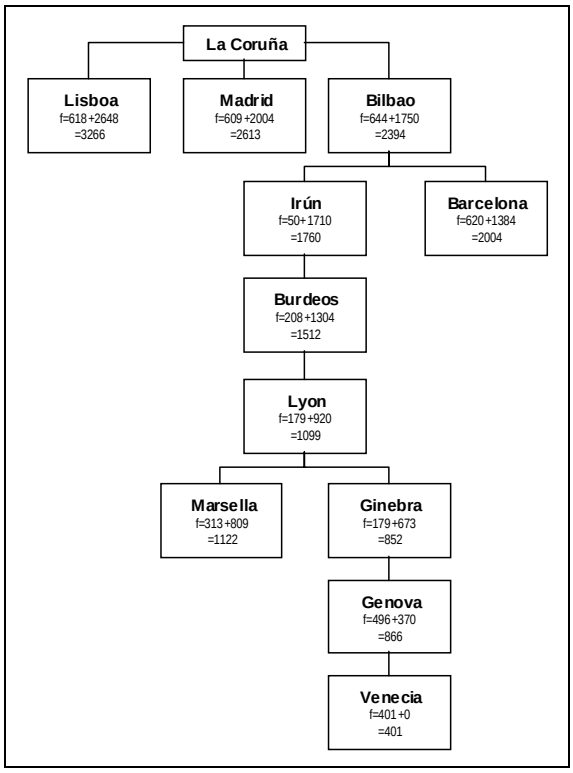


Figura 1.10 Árbol de búsqueda que resulta de la aplicación del algoritmo A*

En cada nivel se selecciona aquella ciudad que presenta un menor coste tanto con respecto a la ciudad origen (función *g*) como a la ciudad destino (función *h*). En este problema en particular, la heurística produce un coste de búsqueda mínimo: obtiene una solución sin tener que expandir un solo nodo que no esté en la ruta de la solución. Nótese que merced a la inclusión en la función de evaluación del coste acumulado desde la ciudad origen, el algoritmo prefiere el camino que pasa por Irún en detrimento de la ruta que pasa por Barcelona. Si sólo hubiese tenido en cuenta el valor de la función heurística, la elección sería Barcelona y el camino resultante La Coruña – Bilbao – Barcelona – Marsella – Génova – Venecia sería consecuentemente más largo.

El algoritmo A* utiliza un grafo dirigido para representar el espacio de estados. En su implementación, hace uso de las listas de nodos ABIERTOS y CERRADOS anteriormente descritas. Básicamente se procede por pasos, expandiendo un nodo en cada paso hasta que se genere un nodo que corresponda a un estado meta. En cada paso, se toman los nodos más prometedores que se han generado hasta el momento, pero que no se han expandido (ABIERTOS). Se generan los sucesores del nodo elegido, se les aplica la función de evaluación y se elige de nuevo el siguiente nodo a expandir. Cada nodo *n* en el grafo, mantendrá enlaces a todos sus sucesores, pero sólo a un único predecesor: aquel que se encuentre en el camino óptimo para llegar desde el nodo inicial al estado *n*.

Búsqueda_A*(N)

1. Poner el nodo inicial en ABIERTOS. Asignarle *g*(n)=0, y *h* al valor que corresponda. Calcular *f*(n). Inicializar CERRADOS como la lista vacía.
2. Si ABIERTOS está vacío, informar del fallo y terminar.
3. Elegir de ABIERTOS el nodo con mejor valor *f*. Llamarle MEJORNODO. Quitarlo de ABIERTOS. Meterlo en CERRADOS.
4. Si MEJORNODO es un nodo meta, salir e informar de la solución siguiendo los enlaces que llevan del nodo meta al nodo inicial.
5. Expandir MEJORNODO generando todos sus sucesores (si no tiene sucesores, ir al paso 2).
6. Para cada sucesor:
   6.1. Poner SUCESOR apuntando a MEJORNODO.
   6.2. Calcular *g*(SUCESOR)=*g*(MEJORNODO) + coste(MEJORNODO, SUCESOR)[1].
   6.3. Si SUCESOR existe en ABIERTOS[2], llamar a este nodo VIEJO. Añadir viejo a la lista de sucesores de mejornodo. Si *g*(VIEJO)>g(SUCESOR)[3], entonces hacer que el enlace paterno de viejo apunte a mejornodo. Hacer g(VIEJO)=g(sucesor), y calcular f(viejo)=g(SUCESOR)+h(VIEJO) Eliminar sucesor.
   6.4. Si SUCESOR existe en CERRADOS, llamar a este nodo VIEJO. Añadir viejo a la lista de sucesores de mejornodo. Si *g*(VIEJO)>g(SUCESOR), entonces hacer que el enlace paterno de viejo apunte a mejornodo. Hacer g(VIEJO)=g(sucesor), y calcular f(viejo)=g(SUCESOR)+h(VIEJO). Eliminar sucesor. Propagar el nuevo mejor coste realizando un recorrido en profundidad a partir de VIEJO[4] (ver procedimiento Propagar_Mejora (VIEJO))
   6.5. Si SUCESOR no está ni en ABIERTOS ni en CERRADOS, calcular h(SUCESOR) y f(SUCESOR), introducirlo en ABIERTOS y añadirlo a la lista de sucesores de MEJORNODO

[1] Donde coste(n,m) es el coste de aplicar el operador que nos lleva de n a m

[2] El nodo ya existe y lo actualizamos como hijo de MEJORNODO

[3] Debemos decidir si el nuevo padre que hemos encontrado para ese nodo es mejor que el que tenía, y si es así actualizar el coste y enlace paterno, de modo que en el grafo queden reflejados sólo los caminos óptimos.

[4] Si acabamos de encontrar un mejor camino a VIEJO debemos propagar la mejora a todos sus sucesores.

El procedimiento adicional "Propagar_Mejora(VIEJO)", se utiliza para propagar la mejora obtenida sobre el coste del nodo apuntado por VIEJO. VIEJO apunta a sus sucesores. Cada sucesor, a su vez, apunta a sus sucesores, y así sucesivamente. Por tanto, para propagar el nuevo coste hacia abajo, podemos hacer una búsqueda en profundidad, empezando en VIEJO, cambiando el valor *g* de cada nodo (y por tanto su valor *f*). La propagación termina cuando se alcanza bien un nodo sin sucesores, bien un nodo para el que ya se ha encontrado un camino equivalente o mejor[30]. Es fácil

[30] Esta segunda comprobación garantiza que el algoritmo acabará aunque haya ciclos en el grafo. Si hay un ciclo, la segunda vez que visitemos un nodo veremos que el camino no es mejor que la primera que lo visitamos.

examinar esta condición. El enlace paterno de cada nodo apunta hacia atrás a su mejor predecesor conocido. Conforme propagamos a un nodo siguiente, debemos mirar si su predecesor apunta al nodo del que estamos viniendo. Si lo hace así, debemos continuar la propagación. Si no, su valor g ya refleja el mejor camino del que forma parte. Por tanto la propagación debe parar ahí. Pero es posible que al propagar un nuevo valor de *g* hacia abajo, el camino que estamos siguiendo pueda volverse mejor para un nodo que el camino a través del antecesor actual. Por tanto debemos comparar los dos. Si el camino a través del antecesor actual es aún mejor, debemos detener la propagación. Sin embargo, el camino a través del cual estamos propagando es mejor que el camino del antecesor actual debemos cambiar el antecesor y continuar la propagación. En detalle, el pseudocódigo del algoritmo de propagación es el siguiente:

```
Propagar_Mejora(VIEJO)

1. Para cada SUCESOR n_i de VIEJO:
   1.1. Si el puntero al padre apunta a VIEJO, actualizar g(n_i) = g(VIEJO) +
        coste(VIEJO, n_i) y f(n_i).
        1.1.1. Si n_i está en CERRADOS, Propagar_Mejora (n_i)
   1.2. Si el puntero al padre no apunta a VIEJO
        1.2.1. Si g(VIEJO) < g(padre de n_i), poner como padre de n_i a VIEJO,
               g(n_i) = g(VIEJO) + coste (VIEJO, n_i), actualizar f(n_i)
               1.2.1.1. Si n_i está en CERRADOS, Propagar_Mejora (n_i).
```

Es conveniente realizar algunas observaciones sobre el papel de las funciones *g*, *h* y *f* en el algoritmo A*. La función *g* nos permite escoger el nodo a expandir sobre la base, no sólo de cuan bueno es el nodo en sí mismo (medido por *h*), sino también sobre la base de cuan bueno era el camino hasta el nodo. Al incorporar *g* en *f*, por tanto, no siempre elegiremos como nuestro siguiente nodo a expandir el nodo que parece más cercano a la meta. Esto es útil si nos interesa minimizar el coste del camino solución. Pero si sólo nos importa llegar a una solución de la forma que sea, podemos definir siempre *g* como 0. Si queremos encontrar un camino que tenga el menor número de pasos, entonces debemos asignar una constante, usualmente 1, al coste de ir desde un nodo a su sucesor. Si, por otra parte, queremos encontrar el camino de menor coste y unos operadores cuestan más que otros, debemos el coste de ir de un nodo a otro tendrá que reflejar los costes de los operadores. Así pues, el algoritmo A* puede usarse tanto si estamos interesados en encontrar un camino de coste total mínimo, como si simplemente queremos encontrar cualquier camino de la forma más rápida posible.

En cuanto a *h*, si es un estimador perfecto del coste del camino hasta la meta, entonces A* convergerá inmediatamente hacia la meta sin búsqueda. Si *h* siempre es 0, la búsqueda estará controlada por *g*. Si *g* también es 0, la estrategia de búsqueda se realizará al azar. Si *g* es siempre 1, se realizará una búsqueda en anchura.

El algoritmo A* es *completo* y *admisible*. Es completo porque siempre acaba encontrando un camino solución, si éste existe. Es admisible porque para cualquier grafo, termina siempre obteniendo un camino óptimo desde un estado inicial hasta un estado meta, con tal de que exista alguno de estos caminos. Se puede demostrar que el algoritmo A* es admisible siempre que la función heurística lo sea.

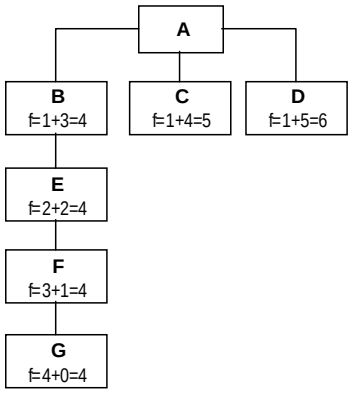


Figura 1.11 La heurística *h* sobreestima a *h**

Un ejemplo que muestra el efecto de la sobreestimación en el algoritmo aparece reflejado en la Figura 1.11. Supongamos que la solución óptima se encuentra generando directamente un sucesor del nodo *D*, con un camino solución de coste 2. Sin embargo, ya que la función $h(n)$ sobreestima $h^*(n)$, es decir, devuelve un coste superior al real, el nodo *D* se considera peor en el primer paso y se expande el nodo *B*. Si *B* se continúa expandiendo de tal forma que todos los nodos presenten una función de evaluación mejor que la de *D*, la búsqueda acabará si se encuentra la solución por este camino, sin haber explorado *D*.

Sin embargo, en la mayoría de los problemas reales, la única forma de garantizar que *h* nunca sobreestime $h^*(n)$, es hacer que su valor sea siempre 0. Esto significa una vuelta al algoritmo primero en profundidad, lo cual es admisible pero no eficiente. Sin embargo, la utilidad del algoritmo A* se mantiene gracias a un corolario sobre su admisibilidad que afirma que: Si *h* raramente sobreestima a *h** más de δ, entonces al algoritmo A* raramente encontrará una solución cuyo coste sea más que δ mayor que el coste de la solución óptima.

Por último, es importante resaltar que el algoritmo A* no tiene por qué resultar adecuado para cualquier problema de búsqueda y su utilidad se deberá evaluar dentro del contexto de cada problema. Un ejemplo en el que el algoritmo A* no resultaría adecuado sería en el caso de tener que realizar una búsqueda de ruta óptima entre dos ciudades en la que los movimientos hacia la meta se realizasen en tiempo real a medida que se selecciona un punto del camino –se expande un nodo-. La función *g* perdería importancia, puesto que sólo nos interesaría el camino que queda por recorrer, y ya que los "movimientos" en el espacio de búsqueda se realizan de forma efectiva, resultaría más apropiado un algoritmo que, como el de ascensión a colinas, escoja de entre los estados sucesores del nodo actual.

Búsqueda por el mejor nodo: Agendas

Hasta ahora se ha asumido que, el hecho de que varios caminos lleven de modo independiente al mismo estado no refuerza el mérito de ese estado. Siguiendo con el ejemplo de la Figura 1.9 y la Figura 1.10, el hecho de que desde La Coruña podamos llegar a Génova por varios caminos, no aumenta el valor de la función de evaluación

aplicada a esa ciudad. Sin embargo, esto no siempre se cumple. Existen situaciones en las que no existe una única y simple función heurística que mida la distancia entre un nodo dado y un objetivo. Además, el que distintos caminos recomienden el cambio a un mismo estado mediante la realización de una tarea, puede ser importante si cada uno de ellos proporciona una razón de porqué ese estado puede conducir a resolver el problema. Cuantas más razones haya, aumenta la posibilidad de que la tarea lleve a la solución. En estos casos, es necesario alguna estructura que nos permita almacenar las tareas propuestas, junto a las razones que se han propuesto para ellas, y algún mecanismo que nos permita gestionar este conjunto de tareas.

Las *agendas* son, básicamente, listas de tareas que puede realizar un sistema. Cada una de las tareas de la agenda suele llevar asociada una lista de razones por las cuales se presume que acometer esa tarea es conveniente[31], y un valor que representa el peso total de la evidencia que sugiere que la tarea es útil.

El *método de búsqueda conducido por agendas* es un procedimiento por el mejor nodo en el que:

- Debemos elegir la mejor tarea de la agenda
- Debemos ejecutar la tarea seleccionada, asignando para ello los recursos necesarios

El término *tarea* puede tener distintos significados. Según los casos, puede entenderse como una descripción explícita de lo que debe hacerse a continuación. En otros casos, simplemente puede ser una mera indicación acerca de cuál debe ser el siguiente nodo que debe ser expandido. Por otra parte, una misma tarea puede llevar asociadas distintas justificaciones, y no todas las justificaciones de una misma tarea tienen por qué *pesar* lo mismo.

Para poder elegir fácilmente la mejor tarea, la agenda se suele mantener ordenada por los valores de sus tareas. De este modo, cada vez que se crea una nueva tarea, ésta es insertada en el lugar apropiado. Además, puede ocurrir que las justificaciones de una tarea cambien. En este caso debemos recalcular su correspondiente peso y trasladarla a una nueva localización en la lista. El esquema del método es el siguiente:

[31] Estas razones o motivos que se asocian a las tareas de la agenda son, recuentemente, utilizadas para elaborar justificaciones y explicar procesos. De hecho, el término *justificaciones* también se emplea para designar a estas estructuras.

Búsqueda_conducida_mediante_agenda

Hasta que se alcance un estado objetivo o la agenda esté vacía hacer:

1. Identificar la mejor tarea de la agenda
2. Ejecutarla
3. Si se han generado nuevas tareas para cada una de ellas
   - 3.1. Si no estaba ya en la agenda
     - 3.1.1. Añadirla la nueva tarea a la lista
   - 3.2. Si estaba ya en la agenda
     - 3.2.1. Si no tiene la misma justificación
       - 3.2.1.1. Añadir la nueva justificación[1]
4. Si se ha añadido una tarea o una justificación calcular el peso asignado a estas tareas combinando la evidencia de todas sus justificaciones

[1] Para poder insertar una justificación necesitaremos construir también una lista de justificaciones

El método propuesto, sin embargo, provoca el gasto de una gran cantidad de tiempo en mantener ordenada la agenda. En la práctica, se utiliza una estrategia modificada que consiste en comparar los nuevos valores de las tareas sólo con algunos elementos superiores de la agenda (usualmente, entre cinco y diez) Si es mejor, se inserta el nodo, en su lugar adecuado en cabeza de la lista. En caso contrario, se deja donde estaba o simplemente se inserta al final de la agenda. Además, de vez en cuando, se debe recorrer la agenda y reordenarla adecuadamente. Esta estrategia puede, ocasionalmente, causar que se ejecute una tarea que no sea la mejor, pero su coste es significativamente menor que el del algoritmo original.

Existen algunos dominios de problemas en los que no es apropiado utilizar un mecanismo de agenda. Este es el caso de los sistemas de razonamiento no monótono, en los que las razones y justificaciones que apoyan en un momento dado a una tarea en concreto, no tienen por qué mantenerse un tiempo después. A pesar de estas dificultades las estructuras de control dirigidas por agendas resultan de gran utilidad. Constituyen un mecanismo adecuado para integrar en un mismo programa información proveniente de distintas fuentes y para resolver problemas en los que algunas tareas proporcionan evidencia negativa sobre otras tareas.

Búsqueda en grafos YO: Reducción de problemas

Hasta ahora, se han considerado estrategias de búsqueda aplicables cuando el espacio de estados se puede representar mediante grafos O (o grafos OR). Estas estructuras reflejan el hecho de que se puede obtener el camino desde un nodo a un estado objetivo si se conoce cómo llegar a ese objetivo a través de cualquiera de las ramas que cuelgan del nodo inicial.

Otro tipo de estructura, los grafos Y/O—o grafos AO, del inglés AND/OR—, son útiles para problemas que pueden resolverse descomponiéndolos en un conjunto de problemas más pequeños, para cada uno de los cuales debemos, a su vez, obtener una solución. Esta descomposición o *reducción* genera arcos Y. Un arco Y puede apuntar a cualquier número de sucesores, de forma que todos ellos deben resolverse para que el arco apunte a una solución. Pero, en un grafo Y/O, estos arcos Y conviven con arcos O, que indican los diversos caminos por los que se puede resolver el problema desde el nodo de partida. Un ejemplo de grafo Y/O se muestra en la Figura 1.12.

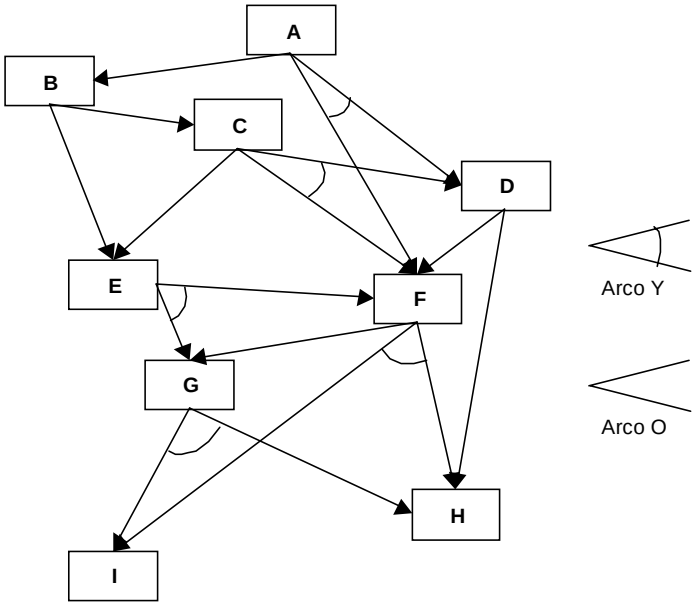


Figura 1.12 Ejemplo de grafo YO

En este ejemplo, se puede ver que los nodos *D* y *F* están conectados mediante un arco Y al nodo *A*, mientras que el nodo *B* está conectado a través de un arco O, de tal forma que para resolver *A* debemos resolver bien *B* o bien el conjunto *D*-*F*. También, el nodo *H* pertenecería a un conjunto de nodos Y en relación con su antecesor *F*, pero sería un nodo O en relación con su antecesor *D*. Un ejemplo práctico en el que el problema se puede representar mediante un grafo YO es la integración de funciones.

En líneas generales, un grafo solución desde un nodo *S* hasta un conjunto *N* de nodos puede obtenerse partiendo del nodo *S* y seleccionando un arco a partir de él. Para cada nodo sucesor a los que ese arco va dirigido, seleccionaremos un arco que parta de él, y así sucesivamente, hasta que, eventualmente, todos elementos del conjunto *N* estén entre los sucesores producidos. En la Figura 1.13 se muestran dos grafos solución distintos partiendo del nodo *A* hasta el conjunto solución { *H* , *I* }

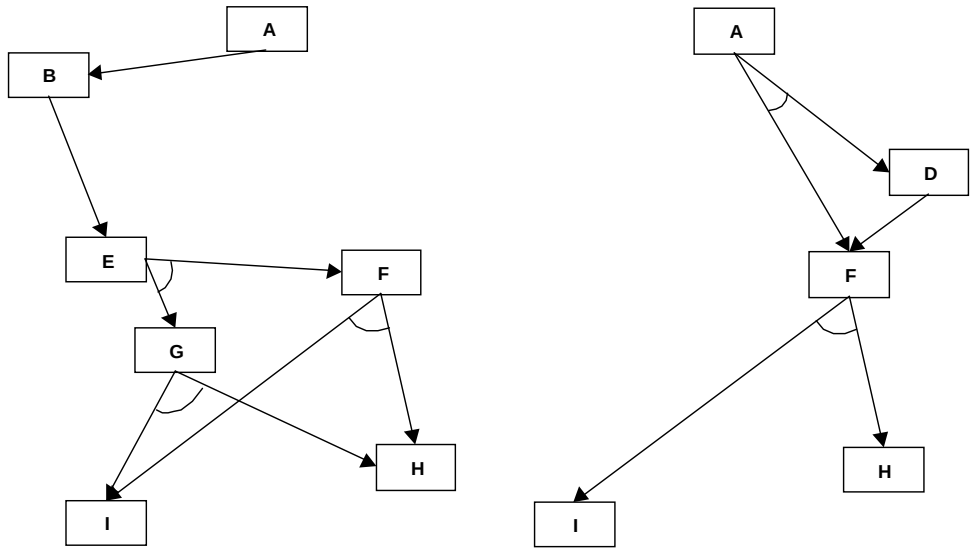


Figura 1.13 Ejemplo de dos grafos solución

Para encontrar soluciones en un grafo Y/O, se necesita un algoritmo similar a una búsqueda por el mejor nodo, pero con la capacidad de manipulación apropiada de los arcos Y. Este algoritmo debería ser capaz de encontrar un camino a partir del nodo inicial hacia un conjunto de nodos que representen los estados solución. Nótese, que puede resultar necesario considerar más de un estado solución ya que cada brazo de un arco Y puede conducir a su propio nodo solución.

El algoritmo de búsqueda Y/O, cuyo seudocódigo se muestra más abajo, se puede ver como una reiteración de dos operaciones. En la primera, una operación de formación del grafo de arriba hacia abajo (pasos 3.1-3.3 del algoritmo), encuentra el mejor de los grafos soluciones parciales. Para ello, seguirá unas marcas que le indicarán el mejor de los grafos soluciones parciales para cada nodo obtenido hasta el momento. Se expande uno de los nodos hoja de ese mejor grafo parcial y se asigna un coste a los sucesores generados.

La segunda operación (paso 3.4) es un procedimiento que se ejecuta de abajo a arriba, en el que se revisan costes, se marcan los conectores que determinan los mejores grafos parciales y se establece si alguna de las partes ya se ha resuelto. Comenzando por el nodo que se acaba de expandir, el procedimiento revisa los costes utilizando los nuevos costes asignados a sus sucesores. Se marca el arco que conecta el nodo con el mejor de sus sucesores. Éstas serán las marcas que determinen el mejor grafo parcial. Este coste revisado se propaga hacia arriba en el grafo.

El algoritmo YO* utiliza un valor denominado COSTE MÁXIMO para detener el proceso de búsqueda. Si el coste estimado de una solución se hace mayor que el valor de COSTE MÁXIMO, la búsqueda se abandona. COSTE MÁXIMO debería elegirse de forma que se corresponda con un umbral tal que, cualquier solución con un coste mayor sea demasiado “cara” para ser práctica, incluso en el caso de que pueda encontrarse.

En lugar de las dos listas de ABIERTOS y CERRADOS que se utilizan en el algoritmo A*, el algoritmo YO* usa únicamente la estructura GRAFO, que representa la porción del grafo de búsqueda que se ha generado hasta ese momento. Cada nodo de este grafo apuntará a sus sucesores y predecesores inmediatos. También tendrá asociado un valor q — obtenido a partir de una heurística h — que representa una estimación del coste del camino existente entre el nodo y el conjunto de nodos solución. Se impondrá de nuevo a h la restricción de monotonía, es decir, para cualquier arco en el grafo dirigido desde el nodo n a los sucesores $n_1, \ldots, n_k$ se deberá cumplir que

$$h(n) < c + h(n_1) + \ldots + h(n_k)$$

donde c es el coste asociado al conector. En este algoritmo, sin embargo, no se utiliza una función g, como en el caso A*.

Ahora, ya es posible establecer el algoritmo de exploración YO*:

Búsqueda_YO*

1. Inicializar GRAFO con el nodo inicial *S*. Asociarle un coste q(S) = h(S).

2. Si *S* es una solución, señalar *S* como RESUELTO.

3. Repetir hasta que *S* se etiquete como RESUELTO o q(S) > COSTE MÁXIMO:

   Atravesar GRAFO empezando en S y siguiendo el mejor camino actual[1]. Acumular el conjunto de nodos de ese camino que aún no se han expandido o no se han etiquetado como RESUELTO.

   Elegir de este conjunto el nodo con mayor valor h. Llamarle NODO.

   Generar los sucesores de NODO. Si no existen, asignar q(NODO) = COSTE MÁXIMO[2]. Si existen sucesores, para cada sucesor hacer:

      Añadir SUCESOR al GRAFO.

      Asignar q(SUCESOR) = h(SUCESOR).

      Si h(SUCESOR) = 0, etiquetarlo como RESUELTO

   Propagar la nueva información descubierta hacia arriba en el grafo:

      Inicializar G[3] con NODO.

      Hasta que G esté vacío, hacer:

         Seleccionar un nodo de G tal que ninguno de sus descendientes en GRAFO se encuentre en G. Si no existe, seleccionar un nodo cualquiera de G. Llamar a este nodo ACTUAL. Quitarlo de G.

         Calcular el coste de los arcos que parten de ACTUAL: para cada arco j, calcular $q_j(ACTUAL)=c+q(n_{1j})+\ldots+ q(n_{kj})$ donde c es el coste del arco (coste de aplicación de un operador), $\{n_{1j}, \ldots, n_{kj}\}$ es el conjunto de nodos Y conectados a ACTUAL a través del arco j[4]. Asignar $q(ACTUAL)=\min\{q_j(ACTUAL)\}$ calculados para todos los arcos j de ACTUAL.

         Marcar el mejor camino para ACTUAL siguiendo el arco de menor coste establecido en el paso anterior. Se borrarán marcas previas que indicaban otro camino mínimo.

         Etiquetar ACTUAL como RESUELTO si todos los nodos conectados a él a través del arco de menor coste han sido etiquetados como RESUELTO.

         Si ACTUAL se ha etiquetado como RESUELTO o si q(ACTUAL) ha cambiado, añadir a G todos los antecesores directos de ACTUAL[5] cuyo valor q < COSTE MÁXIMO.

[1] Este camino se habrá determinado y marcado en ciclos previos de ejecución de los pasos siguientes del algoritmo.
[2] Equivalente a decir que NODO no es resoluble.
[3] G contendrá el conjunto de nodos recientemente etiquetados como RESUELTO, o aquellos cuyo valor de h ha cambiado y que, de esta forma, necesitan propagar una mejora a sus padres.
[4] Nótese que el cálculo del coste de un arco O es una particularización de este método general.
[5] Debemos propagar este cambio hacia atrás en el grafo, para lo cual añadimos los antecesores del nodo que ha cambiado al conjunto G de nodos a revisar.

Como ejemplo de uso de este algoritmo, consideremos de nuevo el grafo de la Figura 1.12 y supongamos que disponemos de las estimaciones siguientes:

$h(A)=0$	$h(B)=2$	$h(C)=1$
$h(D)=1$	$h(E)=4$	$h(F)=4$
$h(G)=2$	$h(H)=0$	$h(I)=0$

donde los nodos *H* e *I* son soluciones y que el coste de cada arco es *n* siendo *n* el número de sucesores conectados mediante ese arco. Además, la función *h* satisface la restricción de monotonía y se hace 0 en las soluciones.

Los grafos de exploración obtenidos después de sucesivos ciclos del bucle externo (paso 3 del algoritmo) se muestran en la Figura 1.14. En esos grafos, junto a cada nodo, aparecen los valores revisados de *q* correspondientes; se usan las flechas gruesas para los arcos marcados que forman el camino mínimo, y los nodos etiquetados con RESUELTO, aparecen resaltados.

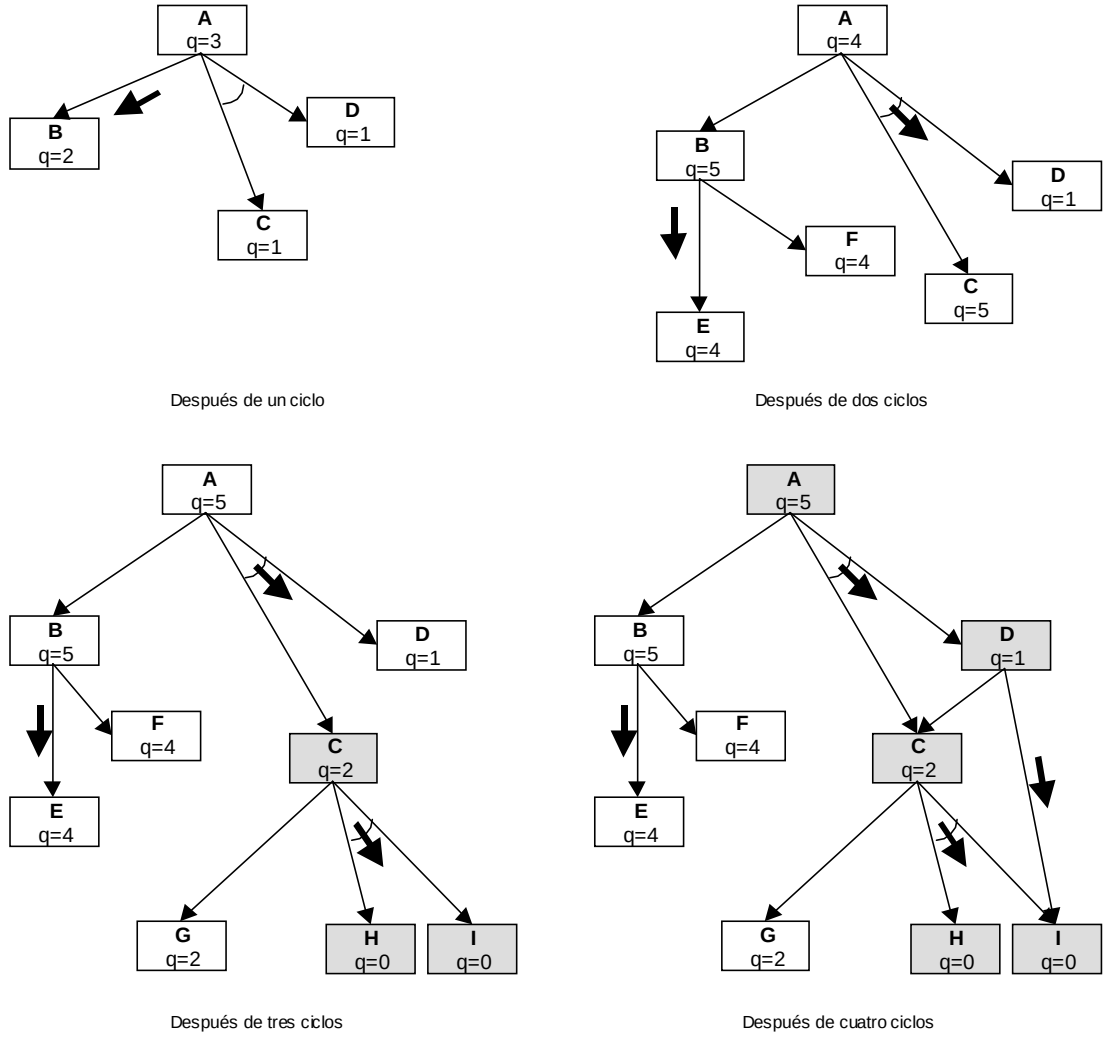


Figura 1.14 Grafos de exploración después de sucesivos ciclos de YO*

Durante el primer ciclo, expandimos el nodo *A*; en el siguiente ciclo, expandimos el nodo *B*; a continuación el *C* y, por último, el *D*. Después de la expansión de *D*, el nodo *A* recibe la señal de RESUELTO. El grafo solución (con coste mínimo igual a 5) se obtiene siguiendo la pista de los arcos marcados.

1.4. Resumen

En este capítulo hemos descrito algunas de las peculiaridades inherentes a la resolución de problemas mediante técnicas de IA. Comenzamos el tema con el enunciado de un problema típico, para cuya resolución es preciso ensayar diferentes estrategias. A continuación formalizamos la resolución de problemas mediante la conceptualización del espacio de estados, en el que distinguimos estados iniciales, estados meta y operaciones permitidas. Los problemas que surgen de la selección y aplicación de operadores, que nos permiten transitar por el espacio de estados, nos llevan directamente a definir los procesos de búsqueda, en los que se discuten algunas características esenciales como son la dirección del proceso, la topología, la representación de estados, los criterios de selección de operadores, o el uso de las llamadas funciones heurísticas. Concluimos el tema con la descripción y desarrollo de algunas técnicas empleadas para la resolución de problemas de IA. En concreto nos centramos en la descripción de los métodos débiles de exploración. Dentro de estos, hemos visto las búsquedas ciegas–anchura, profundidad– que se limitan a recorrer el espacio de estados en un cierto orden en busca de la solución. Los métodos informados –generación y prueba, ascensión a colinas, A*, agendas, YO*–, por el contrario, materializan búsquedas más dirigidas mediante el uso de funciones heurísticas. Estas funciones intentan guiar la exploración del espacio de estados en la dirección más provechosa, haciendo uso de conocimiento sobre el problema y sugiriendo el "mejor" camino a seguir cuando disponemos de varias alternativas. Hemos visto las características que deben cumplir estas funciones y el método de abstracción de problemas para la generación de heurísticas. Existen otros métodos de búsqueda no tratados en este texto de los cuales podrá encontrarse amplia información en la bibliografía que sigue inmediatamente y en el capítulo de Bibliografía Recomendada.

1.5. Textos Básicos

- Nilsson, "Problem-Solving Methods in Artificial Intelligence", McGraw-Hill, eds., 1971
- Nilsson, "Principios de Inteligencia Artificial", Díaz de Santos, 1987
- Rich,"Inteligencia Artificial", G.Gili, eds., 1988
- Rich, Knight,"Inteligencia Artificial", McGraw-Hill, eds., 1994
- Rusell, Norvig, "Artificial Intelligence: A Modern Approach", Prentice Hall, eds., 1995

- Prieditis, “Quantitatively relating abstractness to the accuracy of admisible heuristics”, Artificial Intelligence, vol.14, pp.165-175, 1995.