Resolución de problemas y búsqueda

PID_00267997

Vicenç Torra i Reventós

Revisión a cargo de
Jordi Delgado Pin
Andrés Cencerrado Barraqué

Tiempo mínimo de dedicación recomendado: 8 horas

Universitat Oberta de Catalunya

Vicenç Torra i Reventós **Jordi Delgado Pin** **Andrés Cencerrado Barraqué**

La revisión de este recurso de aprendizaje UOC ha sido coordinada por el profesor: Carles Ventura Royo (2019)

Tercera edición: septiembre 2019

Avda. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Índice

Introducción

Heuristic Search Hypothesis

«The solutions to problems are represented as symbol structures. A physical symbol system exercises its intelligence in problem solving by search –that is, by generating and progressively modifying symbol structures until it produces a solution structure.»

A. Newell; H. A. Simon (1976). «Computer Science as Empirical Inquiry: Symbols and Search». *Communications of the ACM* (vol. 3, núm. 19, págs. 113-126).

Ved también

Recordad la hipótesis de Newell presentada en el módulo «Qué es la inteligencia artificial» de esta asignatura.

En el primer módulo de la asignatura hemos visto que, de acuerdo con la hipótesis de Newell, la actividad inteligente se consigue mediante el uso de patrones de símbolos para representar los aspectos significativos del dominio del problema, operaciones de estos patrones para generar las soluciones potenciales y la búsqueda para seleccionar una solución entre estas posibilidades.

En este módulo nos concentramos en el tercer elemento: la búsqueda. Veremos cómo formalizar un problema de manera que nos defina un espacio en el que intentaremos encontrar la solución. Veremos que hay diferentes tipos de problemas y cómo se puede hacer la búsqueda en sus espacios respectivos.

Para hacer todo esto, este módulo está dividido en los cuatro apartados siguientes:

a) En el primero, se verá cómo se puede formular un problema y cómo dicha formulación define el llamado *espacio de estados*. Veremos algunos ejemplos de problemas y su formulación. Además, mostraremos la representación de un problema.

b) A continuación, en el segundo apartado se muestra cómo podemos encontrar una solución a partir de la formulación del problema. Se plantea un esquema que permite hacer búsquedas en el espacio de estados de muchas maneras diferentes. Además, el esquema es general, de forma que permite resolver problemas de muchos tipos diferentes. Se incluye aquí una implementación con Python de los algoritmos de búsqueda que complementa la proporcionada en el primer apartado para un problema concreto.

c) El esquema general introducido en el apartado «Construcción de una solución» no concreta algunas cuestiones relacionadas con la estrategia a la hora de seleccionar cada paso de la búsqueda. En los apartados «Estrategias de búsqueda no informada» y «Coste y función heurística», se ven algunas de las estrategias para el caso en que para resolver un problema solo hace falta aplicar una secuencia de acciones. Primero se verán métodos que como única información usan el espacio de estados (son las estrategias de búsqueda no

informada del apartado «Estrategias de búsqueda no informada») y después los métodos que usan información adicional, como, por ejemplo, el coste o las funciones heurísticas (apartado «Coste y función heurística»).

d) El módulo sigue con el caso de los juegos cuando hay un adversario. Esto es, cómo elegir qué movimiento se tiene que hacer teniendo en cuenta que jugamos con un contrincante. Primero se considera el caso ideal en que no hay problemas ni de memoria ni de tiempo, y después se plantea qué hacer en caso de tener estas restricciones.

Objetivos

Con este módulo lograréis los objetivos siguientes:

1. Descubrir que las soluciones de un problema están en el espacio de estados.
2. Aprender que diferentes problemas se pueden resolver con unos mismos métodos.
3. Formular los problemas de forma que sea posible encontrar las soluciones.
4. Conocer algoritmos para resolver diferentes tipos de problemas.
5. Entender el funcionamiento de los programas de juegos y comprender la problemática de tener recursos limitados.

1. Introducción a la resolución de problemas y búsqueda

Como se ha dicho en la introducción, el objetivo de un programa de inteligencia artificial es resolver problemas a partir del conocimiento de que se dispone. Para poder hacerlo, hay que formalizar el proceso de resolución. Esto es, formalizar cómo encontramos una solución. La formalización se hace a partir de la modelización de las posibles situaciones con que se puede encontrar un sistema y las acciones que este sistema puede hacer en su entorno de trabajo. Como veremos, el espacio de estados es el que modeliza este conjunto de situaciones posibles y las acciones que lo relacionan. También veremos que es en este espacio donde se hace una búsqueda con objeto de encontrar una solución mediante los llamados *algoritmos de búsqueda*.

En este apartado consideraremos la formalización de un problema para poder aplicar después los algoritmos de búsqueda. Además, consideraremos algunos ejemplos con objeto de ilustrar dicha formalización. Algunos de los ejemplos se volverán a usar en los próximos apartados.

1.1. Espacio de estados y representación de un problema

Para que un sistema pueda aplicar los algoritmos de búsqueda a un problema concreto, hace falta modelizar su entorno de trabajo de manera que sea posible encontrar una solución a partir de la situación inicial en que está. Para hacer esta modelización, hay que considerar una serie de elementos: cuál es el entorno del sistema, qué puede hacer este sistema para cambiar el entorno, etc.

La modelización ha de conducir siempre a estructuras parecidas para que problemas de naturaleza muy diferente se puedan resolver con las mismas herramientas. Esto es, que se les puedan aplicar luego los métodos generales existentes (algunos de los cuales se ven en este módulo).

Ejemplos de modelizaciones

Con objeto de ilustrar el proceso de modelización, consideraremos los ejemplos siguientes:

a) Rompecabezas lineal

Dada una permutación cualquiera de la secuencia [1, 2, 3, 4], determinar cómo construiríamos otra que satisfaga una determinada propiedad (por ejemplo, que esté ordenada de mayor a menor, que represente un número múltiplo de 2, etc.). Consideraremos que, dada una permutación, para construir una nueva podemos intercambiar solo dos números que estén en posiciones consecutivas de la secuencia. Por ejemplo, ¿cómo se ordena la secuencia [3, 4, 1, 2]?

b) Camino más corto

Dado un mapa de carreteras y dos poblaciones, encontrar la ruta de distancia más corta que permite ir de la primera a la segunda. Por ejemplo, de acuerdo con un mapa, ¿como debemos ir de un lugar A a otro B haciendo el mínimo número de kilómetros posibles?

c) Integración simbólica de funciones

Dada una expresión numérica en la que aparece un símbolo de integral, encontrar una expresión equivalente en la que este símbolo no aparezca. Para hacer la integración suponemos que disponemos de un conjunto de «reglas de integración» que describen cómo transformar expresiones en otras equivalentes. Por ejemplo, ¿cuál es la integral de $x^2e^x + x^3$?

d) Demostración de teoremas

Dada una expresión, demostrar su validez. La demostración se basa en la existencia de un conjunto de axiomas iniciales. Por ejemplo, ¿es cierto $a \times (b + c) = a \times b + a \times c$?

La modelización del problema constará de tres pasos: la modelización del entorno en que se mueve el sistema, la modelización de las acciones del sistema y la definición del problema. A continuación, estudiamos cada uno de estos pasos en detalle.

1) Modelización del entorno en que se mueve el sistema

Un sistema necesita conocer el entorno en que está en un instante concreto. Dado que este entorno cambia –en particular, algunos cambios son debidos a las actuaciones del mismo sistema–, tendremos que poder representar todas las situaciones posibles con que un sistema se puede encontrar. La representación del mundo en un instante concreto se denomina *estado*. De acuerdo con esto y para considerar todas las situaciones posibles, la modelización tendrá que tener en cuenta un conjunto de estados, que corresponde a todas las situaciones posibles en todos los instantes.

Para hacer la modelización, tenemos que responder a las preguntas: ¿qué es un estado y cuáles son los estados posibles? De todas formas, para la implementación tendremos que decidir cómo se representa un estado.

El entorno en un momento dado se modeliza mediante lo que denominamos *estado*. Hay que saber cuáles son los estados posibles con que trabajará un sistema y tenemos que determinar cómo representar (implementar) un estado.

De acuerdo con esto, dado que en el caso del rompecabezas lineal, las posibles situaciones con que nos podemos encontrar son las diferentes secuencias, cada secuencia es un estado posible.

Estados posibles en el rompecabezas lineal

En el ejemplo del rompecabezas lineal las secuencias [1, 2, 3, 4], [2, 3, 4, 1], [3, 1, 2, 4] son estados posibles. De hecho, el conjunto de estados posibles es el conjunto de todas las secuencias posibles construibles con las permutaciones de las cuatro cifras. Dado que el número de secuencias posibles formadas con los cuatro dígitos {1,2,3,4} es 4!, tendremos que los posibles estados son 24 y son, evidentemente, los siguientes: {[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2], [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 1, 4, 3], [2, 3, 1, 4], [2, 3,

4, 1], [2, 4, 1, 3], [2, 4, 3, 1], [3, 2, 1, 4], [3, 2, 4, 1], [3, 1, 2, 4], [3, 1, 4, 2], [3, 4, 2, 1], [3, 4, 1, 2], [4, 2, 3, 1], [4, 2, 1, 3], [4, 3, 2, 1], [4, 3, 1, 2], [4, 1, 2, 3], [4, 1, 3, 2]}.

Normalmente, la definición de un problema contiene implícitas una serie de restricciones que remarcan la importancia de distinguir entre estados posibles y estados válidos. Una vez determinada la manera de representar un problema, los estados posibles serán aquellos que sean representables. Sin embargo, si la definición del problema restringe algunas características, nos podremos encontrar con estados que son representables, pero no son válidos. Por ejemplo, si la definición del rompecabezas lineal mencionara que los dos últimos dígitos nunca pueden sumar menos de 5, tendríamos que el estado [4, 3, 2, 1] es un estado representable (nuestra representación permite *expresarlo*), pero no sería un estado válido.

2) Modelización de las acciones del sistema

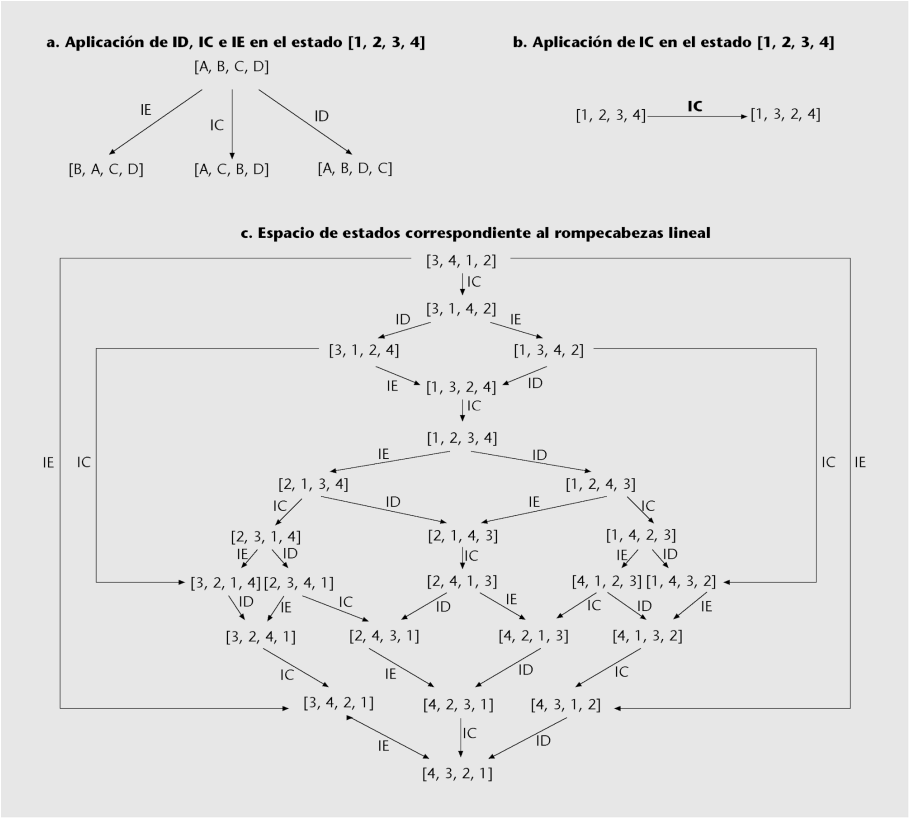
El sistema actúa en su entorno mediante una serie de acciones. Para poder encontrar el camino hacia la solución a partir de la situación actual, necesitamos saber cuáles son las acciones que puede realizar el sistema y qué efecto tienen dichas acciones en el entorno. Dado que lo que hacen las acciones es modificar el entorno, las podemos ver como formas de pasar de una situación –un estado– a otra –otro estado. Por esta razón, las acciones se modelizan como transiciones de un estado a otro. Al número de acciones que se pueden aplicar a los estados de un problema lo denominaremos *factor de ramificación*. Este factor nos será de utilidad a la hora de comparar métodos de búsqueda.

Las **acciones** de un sistema se modelizan como transiciones entre estados. El conjunto de todos los estados posibles y las acciones que actúan en estos estados define el **espacio de estados**. Las acciones son llevadas a cabo mediante **operadores** que, como veremos, en función de cómo se definan, podrán realizar más de una acción diferente.

El **factor de ramificación** es el número de acciones que se pueden aplicar a los estados de un problema.

Los estados y las acciones definen el llamado *espacio de estados*. Este se puede representar gráficamente como un grafo dirigido en que los nodos corresponden a los estados y las aristas corresponden a las acciones. Un camino en este grafo indica qué secuencia de acciones permite al sistema transformar una situación en otra. Encontrar una solución corresponde a hacer una búsqueda en este espacio de estados.

Figura 1



Posibles acciones del sistema en el rompecabezas lineal

En el sistema del rompecabezas lineal hay tres acciones posibles correspondientes a los tres intercambios que podemos hacer. Denominaremos las acciones ID, IC e IE, que corresponderán, respectivamente, a los intercambios derecho, central e izquierdo. Dado el estado [A, B, C, D], la acción IE nos conducirá al estado [B, A, C, D], la acción IC nos conducirá al estado [A, C, B, D] y la acción ID al estado [A, B, D, C]. En el esquema de la figura 1 se hace una representación gráfica de estas acciones. La representación gráfica de la aplicación de la acción IC al estado [1, 2, 3, 4] es al esquema **b**. El espacio de estados correspondiente a este problema aparece en **c**.

La entidad encargada de llevar a cabo una o varias acciones recibe el nombre de operador. La manera como definimos un operador determinará qué y cuántas acciones diferentes puede ejecutar, dependiendo de si este está parametrizado o no, es decir, de si puede recibir unos parámetros o argumentos que configuren la manera en que ejecuta la acción.

Así, por ejemplo, las acciones ID, IC e IE se pueden realizar mediante tres operadores específicos diferentes que no hace falta que reciban parámetros, dado que cada uno ha sido definido para llevar a cabo una sola acción, de manera unívoca (hacer intercambio de posición de los dos dígitos de la derecha, del centro o de la izquierda, respectivamente). Aun así, estas tres acciones las puede llevar a cabo un solo operador `intercambio(posición)` que recibe un argumento indicando qué par de dígitos se tienen que intercambiar, si los de la derecha, los del centro o los de la izquierda.

3) Definición del problema

Para poder encontrar la solución del problema necesitamos, además de la modelización de los estados y de las acciones, la definición de aquello que queremos resolver. Esto es, cuál es el problema. Para hacerlo, requerimos dos cosas:

a) Por un lado, tenemos que saber cuál es la situación actual –el llamado *estado inicial*.

b) Por el otro, necesitamos saber dónde queremos llegar –que corresponde al estado objetivo. Dado que a menudo no hay una única solución al problema, esto es, no hay un único estado que cumpla los requerimientos, el objetivo se define mediante una función. Esta función, llamada *función objetivo*, aplicada a un estado, retorna un valor booleano, que será cierto cuando este estado satisfaga los requerimientos y falso cuando no los satisfaga. En este sentido, el objetivo es una propiedad que solo satisface algunos estados.

Observación

Hay que tener cuidado de no confundir el factor de ramificación con el número de operadores. El factor de ramificación coincide con el número de acciones que se pueden aplicar a un estado, que no tiene por qué coincidir con el número de operadores definidos.

La **definición del problema** necesita una descripción del estado inicial y una función objetivo. Las **funciones objetivo** comprueban la satisfacción de los requerimientos exigidos a un estado solución.

Definición del problema del rompecabezas lineal

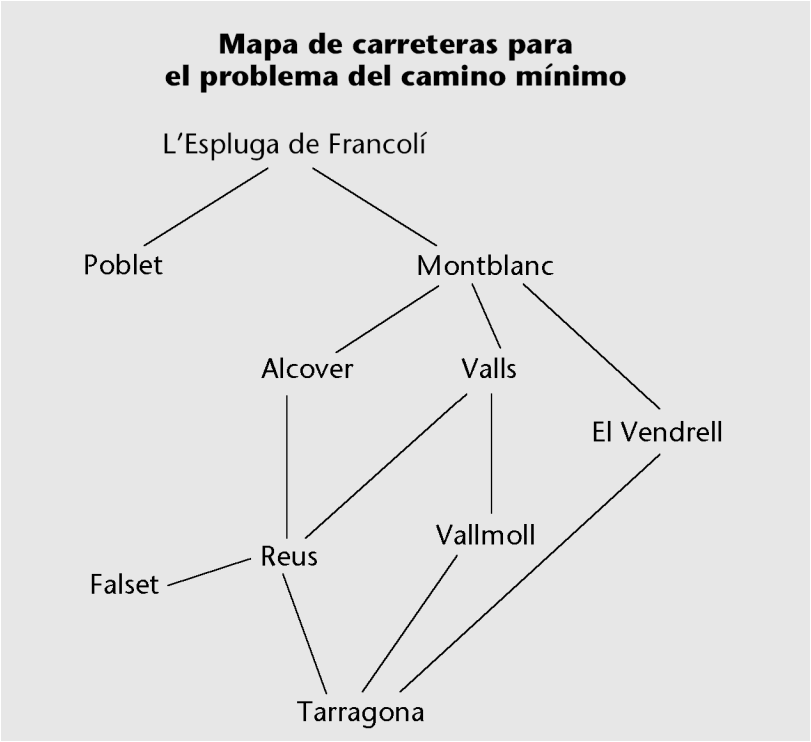
En el caso del rompecabezas lineal, un ejemplo de problema es el siguiente: dada la secuencia [2, 4, 1, 3], determinar los movimientos para conseguir una secuencia que sea múltiplo de dos. En este caso, el estado inicial es [2, 4, 1, 3] y la función objetivo aplicada a un estado corresponde a probar si la secuencia acaba en 2 o en 4. Otro problema es, dada la secuencia anterior, determinar los movimientos para conseguir la secuencia ordenada [1, 2, 3, 4]. En este caso, tenemos, como antes, que el estado inicial es [2, 4, 1, 3] y la función objetivo será comprobar que el estado es [1, 2, 3, 4].

Modelización del problema del camino más corto

La modelización de este problema se hará de manera análoga al problema del rompecabezas lineal.

Empezamos modelizando el entorno en el sistema. Aquí, las diferentes poblaciones corresponden a los varios estados con que se puede encontrar el sistema y las acciones son las carreteras que conectan dos poblaciones diferentes (que nos permiten pasar de un estado a otro). Si consideramos el mapa de carreteras para el problema del camino más corto, tendremos que los estados posibles son las poblaciones que aparecen. Las acciones que nos permiten cambiar de estado son los segmentos de carreteras. Por lo tanto, en un estado tendremos tantos operadores como caminos salgan de una población. Por ejemplo, si consideramos el mapa de carreteras para el problema del camino más corto, tendremos que al estado «Montblanc» le podemos aplicar cuatro operadores que nos llevan a los estados «L'Espluga de Francolí», «Alcover», «Valls» y «El Vendrell». De hecho, el mapa puede ser visto como el espacio de estados. Observad que este mapa tiene la misma estructura que el espacio de estados correspondiente al rompecabezas lineal representado anteriormente. El problema será definir dónde estamos y dónde queremos ir.

Figura 2



Hasta ahora hemos hablado de la modelización de los problemas en términos de estados y de acciones de estos estados con la idea puesta en un estado como representación de un entorno, digamos físico, de un sistema. No obstante, en general la abstracción de la idea de *estado* y de *acción* permite trabajar en otros dominios. No hace falta que haya un sistema que modifique el entorno, basta con las ideas de *estado* y *transformación del estado*.

Ejemplos de tipos de aplicaciones

En el caso del sistema para la integración simbólica, puede ser que lo que realmente nos interese es la integral de una determinada función y no cómo el sistema consigue encontrar dicha integral. Del mismo modo, en el caso de la demostración de teoremas nos puede interesar saber si una determinada propiedad se deduce a partir de unos axiomas (si es cierto que $a \times (b + c) = a \times b + a \times c$) y no los pasos que hacen falta para llegar a demostrarla. Esto último, suponiendo, evidentemente, que el sistema opera correctamente.

Ejemplos de abstracción de la idea de *estado*

1) Modelización del problema de integración simbólica de funciones

Dado que el sistema tiene que tratar con expresiones, definimos *estado* como una expresión numérica. Así, el conjunto de estados posibles es el conjunto de todas las expresiones que permite la notación utilizada. Por ejemplo, si consideramos sumas, restas, multiplicaciones, exponenciaciones, etc., podremos tener expresiones como por ejemplo x, x^2, $3x$, $5x^2$, e^x, e^{5x}... En relación con las acciones, tenemos que estas corresponden a cada una de las técnicas de integración que el sistema prevé. Por ejemplo, que $(1/2)x^2$ es la integral de x. El estado inicial es una expresión cualquiera y un estado final es aquel en el que no hay ningún símbolo de integral.

2) Modelización del problema de demostración de teoremas

Una manera de tratar este problema es considerar el estado como un conjunto de fórmulas lógicas. Así, un estado es aquello que sabemos en un instante concreto que se puede demostrar a partir de los axiomas. En este caso, las acciones corresponden a añadir nuevas fórmulas a un estado que se pueden deducir de las primeras.

Dada una modelización de un problema, un algoritmo de búsqueda determina la secuencia de acciones que permiten pasar del estado inicial a un estado objetivo. Esta secuencia de acciones se denomina *plan* o *camino*. A menudo la

secuencia de acciones y el estado objetivo al que se llega serán la solución al problema. Ahora bien, a veces solo interesa saber cuál es el estado objetivo o, incluso, bastaría con saber que este existe.

En general, tendremos dos tipos de aplicaciones:

1) Aplicaciones en las que interesa saber cómo se llega a un estado objetivo.

2) Aplicaciones en las que solo interesa saber si es posible llegar al estado objetivo.

A pesar de que habitualmente usamos el término *solución* para denotar el estado objetivo únicamente, cuando nos encontramos con el primer tipo de aplicaciones también lo usamos para indicar el camino que conduce hasta ella.

1.1.1. Algunas clases generales de problemas: satisfacción de restricciones y planificación

Hasta ahora hemos visto cómo se aplica la resolución de problemas a algunos ejemplos concretos. Ahora consideramos dos clases de problemas y veremos cómo aplicarles estas técnicas.

Un tipo particular de problema es el llamado *problema de satisfacción de restricciones* (CSP[1]). Todos los problemas de esta familia tienen la misma forma. Tenemos un conjunto de variables a las que tenemos que asignarles un valor. En esta caso, un estado se define como un conjunto de asignaciones a unas cuantas variables. De acuerdo con esto, un operador corresponde a la asignación de un valor a una variable. La solución de estos problemas se puede obtener utilizando los algoritmos de búsqueda y eligiendo métodos adecuados para la selección de qué variable le asignamos.

(1) *Constraint satisfaction problem* en inglés.

Esta clase de problemas tiene interés porque muchos problemas se pueden formalizar de este modo (por ejemplo, encontrar una configuración que satisfaga unas restricciones, o también un problema de juego, como, por ejemplo, el de situar n reinas en un tablero). De hecho, los problemas de satisfacción de restricciones se han aplicado a muchos problemas reales.

El telescopio espacial Hubble

La programación de las observaciones del telescopio espacial Hubble usa esta formulación de satisfacción de restricciones.

Otro tipo de problema es el de la planificación. La tarea de un planificador es encontrar una secuencia de acciones (un plan) que permita al sistema hacer una tarea concreta. La secuencia describe qué acciones se tienen que hacer y en qué momento.

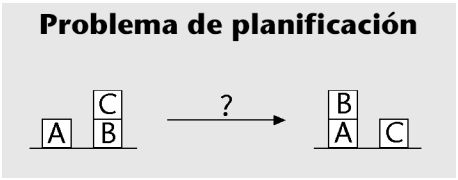
Ejemplo de planificación

Consideramos la ubicación de tres objetos A, B y C y queremos determinar los movimientos necesarios para poner cada objeto en la posición requerida. En la figura 3 se muestra dónde están los objetos y cómo se quieren dejar. Para conseguir este resultado, el sistema puede coger un objeto apilado y dejarlo sobre la superficie o coger un objeto y ponerlo sobre otro.

El espacio de planes

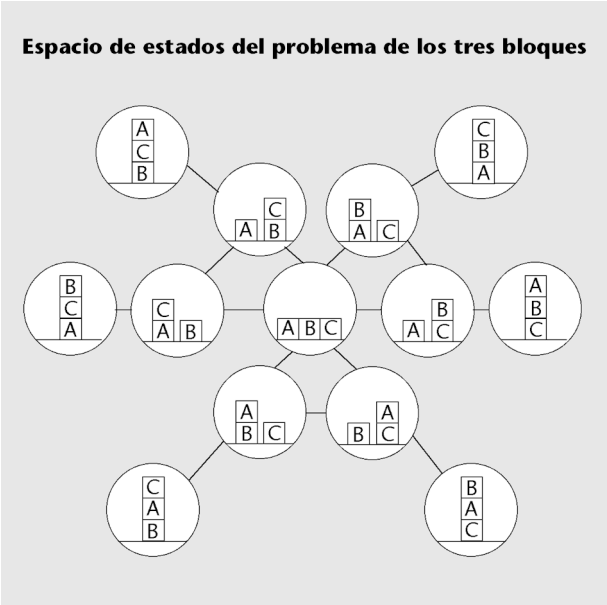
Actualmente se prefiere hacer la búsqueda en el espacio de planes en vez de hacerlo en el espacio de estados.

Figura 3



La manera más sencilla de tratar la planificación es considerando que cada posible situación de los objetos es un estado. En este caso, las acciones que relacionan los estados son las acciones que puede hacer el agente para actuar en el mundo. En la figura 4, se muestra el espacio de los estados correspondiente a un problema de planificación de acuerdo con esta formalización. El espacio de estados se ha definido usando los tres bloques ya utilizados antes: A, B y C. El problema considera los operadores `mesa (objeto)` –que deja el `objeto` sobre la mesa– y `encima (objeto 1, objeto 2)` –que toma el `objeto 1` y lo sitúa encima del `objeto 2`. El estado inicial y el objetivo para este problema se han dado en la figura 3.

Figura 4. Ejemplo de espacio de estados



La figura muestra el espacio de los estados del problema de tres bloques. Fuente: adaptado de D. S. Weld (1994). «An introduction to least commitment planning». En: *Magazine*.

A continuación, se presentan con más detalle los problemas de satisfacción de restricciones.

Problemas de satisfacción de restricciones

Un problema de satisfacción de restricciones se define mediante un conjunto de variables, un dominio para cada variable y un conjunto de restricciones que las variables han de satisfacer. El problema estará resuelto cuando todas las variables tienen un valor asignado y, además, se satisfacen todas las restricciones.

Ejemplos de problemas de satisfacción de restricciones

En primer lugar, veremos el problema muy conocido de situar ocho reinas en un tablero sin que se maten. Después, consideramos el ejemplo conocido como el problema criptoaritmético.

El problema de las ocho reinas: dado un tablero de ajedrez (de 8 × 8) se tienen que poner ocho reinas de forma que no se maten entre sí. Esto es, que dos reinas no coincidan ni en la misma fila, ni en la misma columna, ni tampoco en una diagonal.

El problema criptoaritmético: se considera una expresión aritmética correspondiente a la suma de dos números y su resultado. Sin embargo, a cada número, en lugar de asignarle dígitos, se le asigna una codificación en términos de letras del alfabeto. Así tenemos una expresión como por ejemplo:

```
  DOS
+TRES
-----
 CINCO
```

Suponiendo que para cada dígito solo hay una codificación (no pueden haber dos letras diferentes a las que se les asigne el mismo valor), el problema consiste a encontrar una asignación en las letras del alfabeto de forma que la suma cuadre. Es decir, en el caso del ejemplo, una solución sería asignar el valor 1 a la letra E, el valor 2 a la letra S y el resto de asignaciones como se indica a continuación. Con esta asignación, la suma anterior corresponde a: 952 + 3.812 = 4.764.

```
1 2 3 4 5 6 7 8 9 0

E S T C O N I R D _
```

Además de estos dos ejemplos de problemas de juego, de este mismo modo también se pueden plantear problemas de planificación temporal (diseño de horarios) o problemas de configuración (qué componentes se tienen que colocar).

Ahora debemos definir, para esta clase de problemas, los tres elementos que hacen falta para resolver un problema. Si tenemos en cuenta dichos elementos, podremos aplicar los algoritmos de búsqueda que veremos en los próximos apartados a un problema de esta clase. Recordemos que los tres elementos son: modelización del entorno en el que se mueve el sistema, modelización de las acciones del sistema y definición del problema.

Veremos que la modelización se hace basándose en unas variables y unas restricciones sobre los valores. Primero consideramos el caso general y después concretaremos para dos casos de ejemplo. Pasamos, pues, a concretar los tres elementos anteriores:

1) Modelización del entorno en el que se mueve el sistema. El estado se representará mediante la lista de variables del problema y los valores que tienen asignados las variables. En un estado concreto, solo se tendrá un subconjunto de todas las variables.

2) Modelización de las acciones del sistema. Dado que lo que queremos es que al final cada variable tenga asociado un valor, las acciones que permitimos son las de asignar un valor a una variable. Por lo tanto, tendremos tantas acciones como valores de variables haya.

3) Definición del problema. Inicialmente ninguna variable tiene asignado un valor. Queremos conseguir que todas las variables tengan un valor asignado y que, además, se cumplan las restricciones. Por lo tanto, el estado inicial serán las variables sin asignar y la función objetivo será comprobar que todas las variables estén asignadas y que los valores cumplan las restricciones.

Ejemplos de modelización de problemas de satisfacción de restricciones

1) Problema de las ocho reinas

Una manera de modelizar este problema siguiendo lo que se ha explicado hasta ahora es considerar ocho variables de forma que cada una de ellas corresponda a la columna en que está la reina de la *i-ésima* fila. Así, si consideramos las variables x_1, x_2, x_3, x_4, x_5, x_6, x_7 y x_8 podemos entender x_1 como la columna donde está la reina de la fila 1, x_2 como la columna donde está la reina de la fila 2. Así, en general, x_i es la columna donde está la reina de la fila *i-ésima*. Dado que cada reina puede estar situada, en principio, en cualquiera de las ocho columnas, el dominio de cada variable es el conjunto {1, 2, 3, 4, 5, 6, 7, 8}. Con estas variables, un estado corresponderá a tener valores asociados a unas cuantas variables (esto, de hecho, se puede ver como equivalente a tener unas cuantas reinas colocadas). Una acción consistirá en asignar un valor a una variable que aún no tenía un valor asignado. El estado inicial será no tener ninguna variable asignada y el estado final será tener valores asignados a todas las variables y que, además, se cumplan las restricciones (que las reinas no se maten).

La formulación de las restricciones será:

- Que dos reinas no estén en la misma fila: la construcción del problema ya no permite este caso.
- Que dos reinas no estén en la misma columna:
$$x_i \neq x_j \text{ para todo } i \neq j$$
- Que dos reinas no estén en la misma diagonal: hay dos casos en los que dos reinas están en la misma diagonal. Son los de la figura 5 (casos 1.º y 2.º) cuando $a = b$. Observad que suponemos que la fila j es más grande que la i ($j > i$). El primer caso se dará si:
$$x_i - x_j = j - i$$

y el segundo caso se dará si:

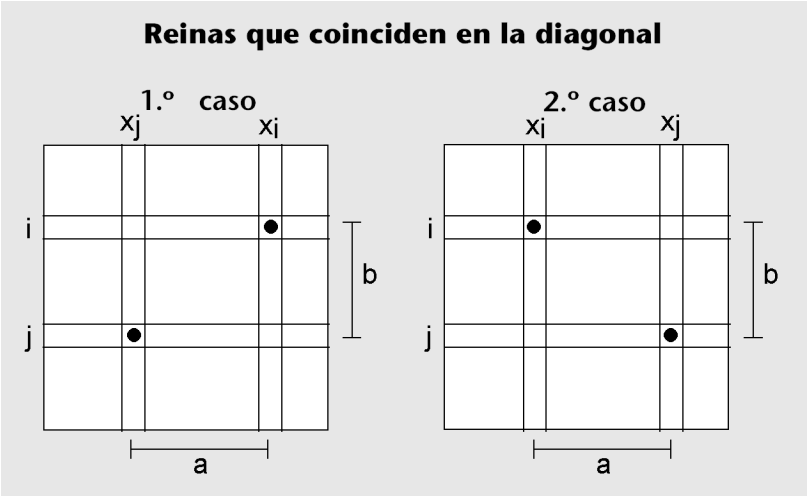
$$x_j - x_i = j - i$$

Por tanto, para exigir que dos reinas no estén en la misma diagonal necesitamos:

$$\text{si } j > i,\ x_i - x_j \neq j - i$$

$$\text{si } j > i,\ x_j - x_i \neq j - i$$

Figura 5



2) Problema criptoaritmético

También se puede formalizar de manera similar. En este caso, hemos de tener una variable para cada una de las letras que aparecen en la expresión. Dado que cada letra se puede hacer corresponder a uno de los dígitos del 0 al 9, el dominio de cada variable será el conjunto {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. En este caso, como antes, los operadores son elegir una variable y una asignación. Finalmente, para definir el problema criptoaritmético necesitamos concretar el estado inicial y la función objetivo. El estado inicial será no tener ninguna variable asignada y la función objetivo será que cada variable (cada letra) tenga un valor y que no haya dos variables a las que se les haya asignado el mismo valor. Las restricciones serán las que se deducen de la expresión (teniendo en cuenta que cuando hagamos la suma podemos llevar): (S + S) mod 10 = C; O + E + [(S + S) div 10] = N; ...

1.1.2. Algunas consideraciones adicionales: la importancia de una representación adecuada y la cuestión del coste

Dado que los algoritmos de búsqueda se basan en el espacio de estados que se define, interesa que el espacio sea lo más reducido posible para que la búsqueda sea menos costosa. A veces, un cambio de representación puede reducir enormemente una búsqueda.

Un ejemplo del mundo real de que un cambio de representación puede reducir la búsqueda es el del problema de la planificación. Antes hemos considerado que una representación se basa en el espacio de estados. Actualmente, se considera la resolución del problema basada en una búsqueda en el espacio de planes porque es más eficiente. En este caso, el estado inicial es un plan vacío (sin ninguna acción) y el objetivo es conseguir un plan que lleve a una situación que satisfaga las condiciones que nosotros imponemos. Un estado es un plan que se ha especificado solo parcialmente y las acciones son operaciones de refinamiento del plan (por ejemplo, añadir una acción al plan).

A la hora de definir un problema hay otro aspecto que a veces se ha de tener en cuenta: el coste. En algunos problemas, no todos los caminos son igual de buenos, ni tampoco todas las acciones. Las acciones que el sistema realiza se pueden evaluar en términos de una función de coste. El coste puede corresponder a la dificultad de hacer la acción o al precio que se tiene que pagar para llevarla a cabo. El coste de un plan se corresponde a cómo sea de buena una secuencia de acciones y, a menudo, es la suma de los costes de las acciones que forman parte del plan. Un ejemplo muy conocido es cuando queremos determinar cómo ir de una población a otra y tenemos diferentes alternativas (rutas). En este caso, el coste puede corresponder al número de kilómetros entre las dos poblaciones, al tiempo necesario para hacer una ruta o al coste económico de hacer el desplazamiento.

Solución óptima del rompecabezas lineal

Si queremos encontrar la solución con el número de inversiones más bajo, podemos asignar a cada acción un coste unidad y, a un camino, la suma de los costes de las acciones que forman parte. Dado que esto último corresponderá a la longitud del camino, podremos primar aquellas soluciones con menos inversiones.

La cuestión del coste nos permitirá evaluar los diferentes algoritmos de búsqueda. Así, diremos que una solución es óptima si el camino del estado inicial al objetivo es mínimo (menor que todos los otros trayectos). Un algoritmo será óptimo si encuentra una solución óptima.

1.1.3. Análisis práctico del problema de las ocho reinas

La modelización escogida para abordar un problema es clave a la hora de llevar a cabo cualquier operación o intento de resolución. Así, pues, hay que tener cuidado con cada una de las definiciones y decisiones que se toman en la modelización, puesto que a menudo un aspecto (por ejemplo, la representación de los estados) tiene una consecuencia directa en otros (por ejemplo, la elección de la estrategia de búsqueda de una solución).

Ved también

Profundizaremos en el tipo de estrategias de búsqueda en el apartado «Construcción de una solución».

Basándonos en la modelización que acabamos de dar del problema de las ocho reinas, nos podemos hacer algunas preguntas que serán determinantes para poder implementar un procedimiento de búsqueda de soluciones:

- **¿Cuántos estados posibles tiene nuestro problema?**
  Dado que tenemos ocho variables, con nueve posibles valores para cada una (uno por columna, más el valor nulo que indica la variable sin asignación), tenemos 9^8 diferentes posibles estados.

- **¿Son válidos todos los estados posibles?**
  Los 9^8 posibles estados que acabamos de mencionar son los estados que nuestra modelización **permite representar**. Esto no quiere decir que todos sean válidos, puesto que la formalización del problema impone unas restricciones muy claras que muchos de estos estados no cumplirán (por ejemplo, cualquier estado en que dos o más variables tengan el mismo valor asociado).

- **¿Cuáles son los operadores de que disponemos? ¿Cómo cambian los estados?**

La única acción que se define en la modelización propuesta es «asignar un valor a una variable que aún no tenía ningún valor asignado». Así, pues, contamos con un único operador, que realizará esta acción. Este operador necesitará recibir como argumentos la variable que se debe modificar y el valor que asignará, de forma que transformará un estado en otro mediante la modificación de la variable.
A nivel conceptual, también es válido considerar que, dado que tenemos ocho variables, podemos disponer de ocho operadores, en el que cada uno de estos está «dedicado» a una sola variable.

- **¿Son todos los estados accesibles desde cualquier otro estado?**
  No todos los estados serán accesibles desde cualquiera otro estado, dado que la acción modelada no permite asignar un valor a una variable que ya tiene un valor asignado. Aparte de este detalle, se puede considerar que el mismo operador tiene en cuenta por sí mismo si el estado que producirá es válido o no. Si lo tiene en cuenta y, en consecuencia, nuestro operador no puede generar estados inválidos, entonces el número de estados accesibles desde cualquier otro estado es muy menor que en el supuesto de que el operador pueda producir estados inválidos.

Como se puede observar, en muchos casos estos planteamientos están sujetos a la implementación concreta que se haga del problema. Generalmente, las definiciones de problemas dejan margen para tomar diferentes decisiones que pueden ser igualmente válidas y respetar las restricciones del problema, tanto implícitas como explícitas.

De estas decisiones dependerá, en buena medida, el rendimiento que tendrán nuestros algoritmos, así que es importante hacer una buena reflexión a nivel conceptual de la solución que se propone, teniendo en cuenta las estrategias de resolución que estudiaremos en los próximos apartados.

Ved también

Para entender el código que se expone en este apartado, es aconsejable echar un vistazo al anexo que podéis encontrar al final del módulo. Explica el tratamiento de las listas de Python que hemos adoptado en el código que encontraréis de ahora en adelante.

1.1.4. Representación de un problema: implementación con Python del rompecabezas lineal

A continuación, haremos la representación del problema del rompecabezas lineal con Python. Su implementación corresponde a definir todos aquellos elementos que necesitamos para poder aplicar posteriormente los algoritmos de búsqueda. En particular, y tal como se ha descrito en el subapartado precedente, necesitamos los elementos siguientes:

- **Modelización del entorno**. Esto es, tenemos que saber cómo representaremos un estado (como hemos dicho antes, un estado corresponderá a una permutación de las cuatro cifras).

- **Modelización de las acciones del sistema**. Tenemos que definir funciones para cada uno de los operadores disponibles. Por tanto, tenemos que definir tres funciones, una para cada uno de los tres intercambios posibles.

- **Definición del problema**. Tenemos que saber cómo representaremos el estado inicial y tenemos que disponer de una función que, aplicada a un estado, nos retorne «cierto» cuando este sea un estado objetivo.

Pasamos, ahora, a la definición de estos elementos:

1) Modelización del entorno

Definición de la estructura para representar un estado. Lo más simple es representar un estado mediante una lista de cuatro números correspondientes a los cuatro dígitos de una secuencia. Con Python, representaremos el estado mediante la secuencia [A, B, C, D].

2) Modelización de las acciones del sistema

Definimos las funciones `mov_ie`, `mov_ic` y `mov_id` correspondientes a los intercambios posibles. Las funciones se aplican al estado. Por lo tanto, reciben el estado actual como parámetro y retornan el estado nuevo. Así, tendremos:

```
def mov_ie (estado, info):
    return [cadr(estado), car(estado), caddr(estado), cadddr(estado)]

def mov_ic (estado, info):
    return [car(estado), caddr(estado), cadr(estado), cadddr(estado)]

def mov_id (estado, info):
    return [car(estado), cadr(estado), cadddr(estado), caddr(estado)]
```

El esquema general que siguen las tres funciones es similar: construimos una lista a partir de los cuatro elementos que definen la nueva secuencia. La diferencia entre las tres funciones es el orden en que se toman los elementos. En todos los casos, dado que el estado es una lista, para conseguir el primer elemento basta con coger la cabecera de la lista. Así, tenemos que `car(estado)` es el primer elemento. En cambio, para conseguir el segundo elemento, tomamos la cola de la lista (todos menos el primero, o sea, del segundo hasta el último) y de esta cola tomamos el primer elemento (o sea, el segundo de la lista original). Esto es `car(cdr(estado))`. La función `cadr` es equivalente a estas dos llamadas.

Información adicional

Además de un estado, las funciones reciben otro parámetro que denominamos *información adicional*, que de momento no consideramos, pero que nos será de utilidad más adelante, en el apartado «Búsqueda con profundidad limitada» dedicado a su implementación.

De manera similar, para conseguir el tercer elemento, aplicamos dos veces la función cola (`cdr`) al estado, de forma que obtenemos la lista formada por el tercero y el cuarto elemento. Dado que el primer elemento de esta lista es el tercero, aplicando `car` tendremos el tercer elemento. Esto es: `car(cdr(cdr(estado)))` o lo que es equivalente: `caddr(estado)`. Para obtener el cuarto elemento tenemos que hacer lo mismo que en el caso anterior, pero ahora aplicando la función `cdr` una vez más. Así, tenemos que el cuarto elemento será: `car(cdr(cdr(cdr(estado))))` o de manera equivalente: `cadddr(estado)`. Si resumimos, tenemos que para conseguir los cuatro elementos que componen la lista hemos de hacer:

```
Primer  elemento   car (estado)
Segundo elemento   cadr(estado)
Tercer  elemento   caddr(estado)
Cuarto  elemento   cadddr(estado)
```

Los operadores del rompecabezas lineal los almacenamos en una lista de pares de elementos que retornará la función `rl_operadores` (donde `rl` corresponde a rompecabezas lineal). Para cada operador tenemos su nombre y la función que nos permite obtener el nuevo estado a partir de un estado inicial. Así, para el intercambio izquierda tenemos `'ie'`, que es el nombre del operador y la función correspondiente (`mov_ie` representa la función). Del mismo modo, tenemos `['ic', mov_ic]` para el intercambio central y `['id', mov_id]` para el intercambio derecho. Así, la lista de operadores será:

```
def rl_operadores():
    return [['ie', mov_ie],
            ['ic', mov_ic],
            ['id', mov_id]]
```

3) Definición del problema

Necesitamos saber cómo representamos el estado inicial y disponer de una función objetivo. La representación del estado inicial es clara una vez ha sido elegida la representación de los estados. Así, el estado inicial es la secuencia [2, 4, 1, 3].

En cuanto a la función objetivo, tenemos que cuando las soluciones del problema son aquellos estados donde la secuencia acaba en 2 o 4 una posible función objetivo es esta:

```
def rl_funcion_objetivo_par(estado):
```

```
    return cadddr(estado) == 2 or cadddr(estado) == 4
```

Esta función se cumplirá cuando el último elemento de la secuencia `cadddr(estado)` es igual a 2 o a 4. Si consideramos que solo hay un estado solución y este es el de la secuencia [1, 2, 3, 4] tendremos que una función objetivo puede ser:

```
def rl_funcion_objetivo(estado):
    return estado == [1,2,3,4]
```

Esta función simplemente comparará el estado con el único estado objetivo.

Ahora, con todos estos elementos podemos montar el problema. Para ello, definimos una función llamada *problema*:

```
def problema():
    return [rl_operadores(),
            (lambda info_nodo_padre, estado, nombre_operador: []),
            [4,3,2,1],
            (lambda estado: estado == [1,2,3,4]),
            (lambda estado: [])]
```

En la definición del problema consideramos los cuatro elementos siguientes: los operadores del problema (en este caso `rl-operadores()`), el estado inicial (la lista `[4,3,2,1]`), la función objetivo `lambda estado: estado == [1,2,3,4]` y una función, que más adelante nos será de utilidad, que asocia a cada estado información adicional (de momento, como que no la usamos, la definimos de forma que retorne `[]` para cualquier estado). El problema se representa mediante una lista con estos cuatro elementos en el orden establecido.

2. Construcción de una solución

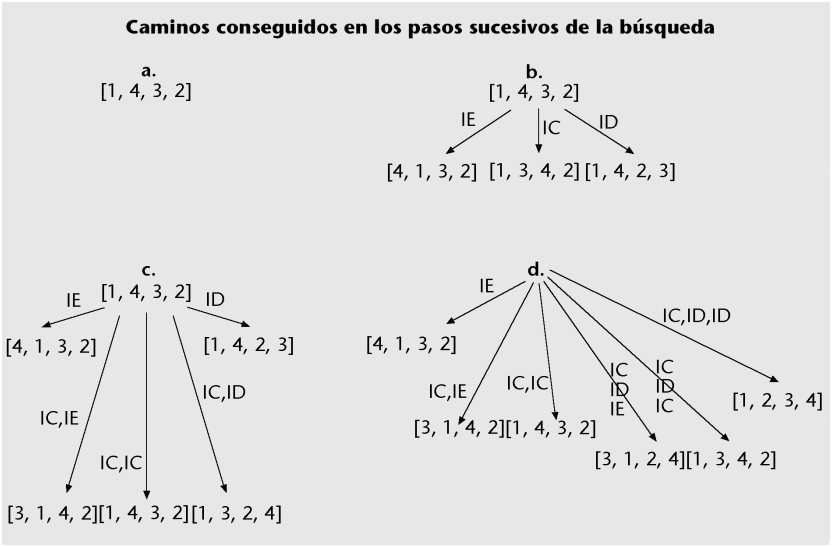
Una vez hemos definido el problema, debemos encontrar la solución. Dado que sabemos de dónde partimos (el estado inicial) y adónde queremos ir (un estado que satisfaga la función objetivo), lo que tenemos que hacer es encontrar un camino que nos lleve hasta ahí. La búsqueda corresponde a encontrar ese camino en el grafo que representa el espacio de estados.

Un algoritmo de búsqueda construye el camino (la solución) en pasos sucesivos. La idea general es disponer de un conjunto de caminos (soluciones) parciales y a cada paso prolongarlos (mejorarlos) un poco más. Esto se consigue eligiendo uno de los caminos del conjunto y considerando aquellas acciones que el sistema puede aplicar al último estado del camino. Cada acción nos determina un nuevo camino. Así, tomamos un camino y, para cada acción que se pueda aplicar al estado terminal, construimos un nuevo camino. El camino corresponde al camino original más la transición correspondiente a la acción seleccionada.

Proceso de construcción de la solución en el rompecabezas lineal

Retomamos el problema del rompecabezas lineal y lo ilustramos:

Figura 6



Consideramos el problema con un estado inicial igual a [1, 4, 3, 2] y como objetivo la secuencia [1, 2, 3, 4]. En este caso, empezamos con un único camino vacío (sin ninguna acción) que empieza y acaba en el estado inicial. Esta primera situación se representa en el paso **a**. Dado que este estado no es la solución al problema, consideramos qué acciones se le pueden aplicar a dicho estado y a qué estados nos conducirán: las tres (IE, IC, ID) son aplicables y nos conducen, respectivamente, a los estados: [4, 1, 3, 2], [1, 3, 4, 2] y [1, 4, 2, 3]. Con esta información podemos construir tres caminos alternativos (formados cada uno con una única acción) que llevan a los tres estados nuevos. En el paso **b** se meustra la representación de estos tres caminos y los estados adonde se llega a partir del estado inicial [1, 4, 3, 2].

Proseguimos seleccionando uno de los tres caminos y aplicando en el estado terminal las acciones que le son adecuadas (en este problema, lo son las tres). Suponemos que el

camino seleccionado es el segundo (el que dirige a [1, 3, 4, 2]), por lo tanto, cuando consideramos las tres acciones obtenemos tres nuevos caminos que llevan, respectivamente, a [3, 1, 4, 2], [1, 4, 3, 2] y [1, 3, 2, 4]. La representación gráfica de los caminos que tenemos en este punto se dan en el paso **c**.

El paso siguiente sigue el mismo esquema. Tenemos que considerar los caminos existentes, elegir uno y aplicar las acciones adecuadas al estado terminal. En este momento hay cinco caminos que debemos considerar (los que ya se han considerado no se tienen que considerar de nuevo, puesto que reharían caminos que ya tenemos). Son los que conducen a: [4, 1, 3, 2], [3, 1, 4, 2], [1, 4, 3, 2], [1, 3, 2, 4] y [1, 4, 2, 3]. Si ahora seleccionamos el camino hasta el estado [1, 3, 2, 4] tendremos que entre los nuevos caminos hay uno que lleva al estado objetivo. En el paso **d** se dan todos los caminos que tenemos en este punto. Se puede ver que el camino IC, ID, IC lleva a la secuencia que queríamos [1, 2, 3, 4].

Para que la parte común de los caminos no aparezca repetida y se pueda ver aquello que comparten, la implementación de la búsqueda se acostumbra a representar mediante una estructura de árbol. La raíz del árbol corresponde al estado inicial, los nodos del árbol son los estados, los arcos corresponden a las acciones y las hojas del árbol corresponden a los estados terminales de los caminos. De este modo, cuando reseguimos las ramas del árbol desde el nodo inicial a una hoja, estamos repasando un camino desde el estado inicial a un estado terminal, pasando por todos los estados intermedios. Al mismo tiempo, podemos conocer las acciones que el sistema debe hacer para llevar a cabo los cambios de estado y conseguir llegar a un estado hoja (son los arcos del árbol).

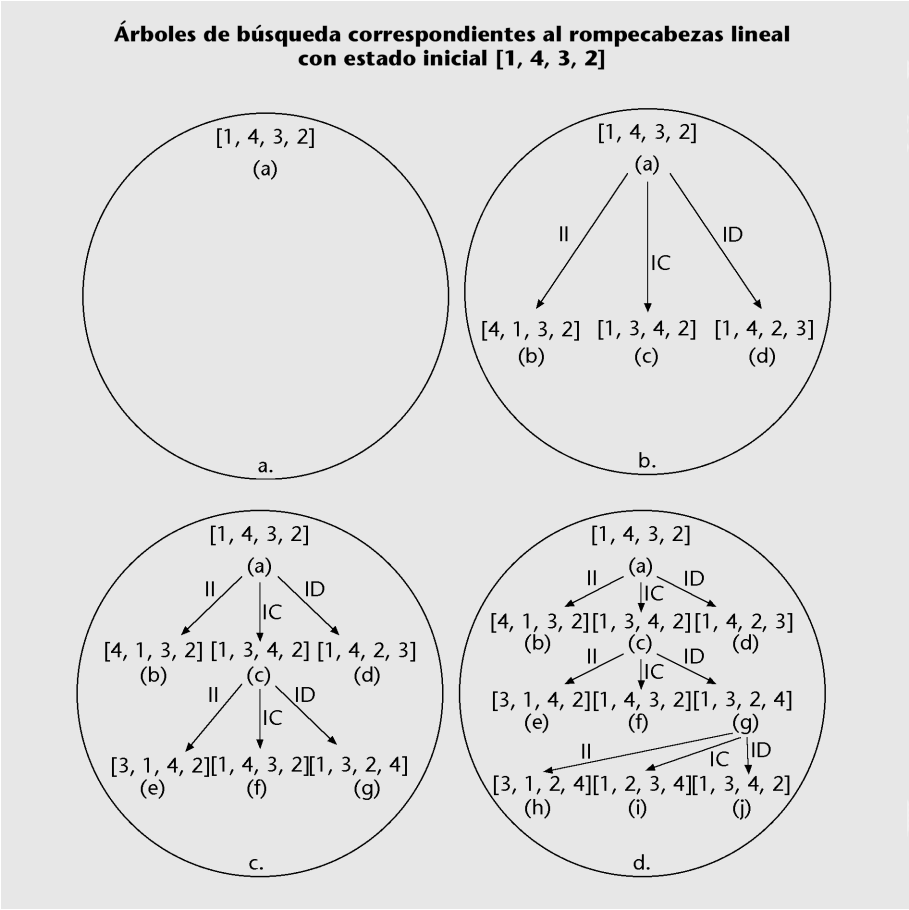
La elección de un camino corresponde a escoger un nodo hoja y la aplicación de las acciones al estado terminal corresponde a aplicar las acciones al estado asociado al nodo hoja. La aplicación de las acciones proporciona un conjunto de estados que son añadidos al árbol como descendientes del nodo terminal. El nodo ya tratado dejará así de ser hoja. De este modo, en todo momento, las hojas del árbol corresponden a caminos que aún no se han considerado, mientras que todos los nodos que no están en las hojas corresponden a nodos ya tratados. En un momento dado de la búsqueda denominaremos *frontera del árbol de búsqueda* al conjunto de nodos que no se han tratado. La consideración de un nodo de la frontera y la aplicación de las acciones que le son adecuadas se denomina *expansión del nodo*.

La expansión de un nodo consiste en aplicar los operadores al estado asociado a un nodo.

Denominaremos *frontera del árbol de búsqueda* al conjunto de nodos no expandidos.

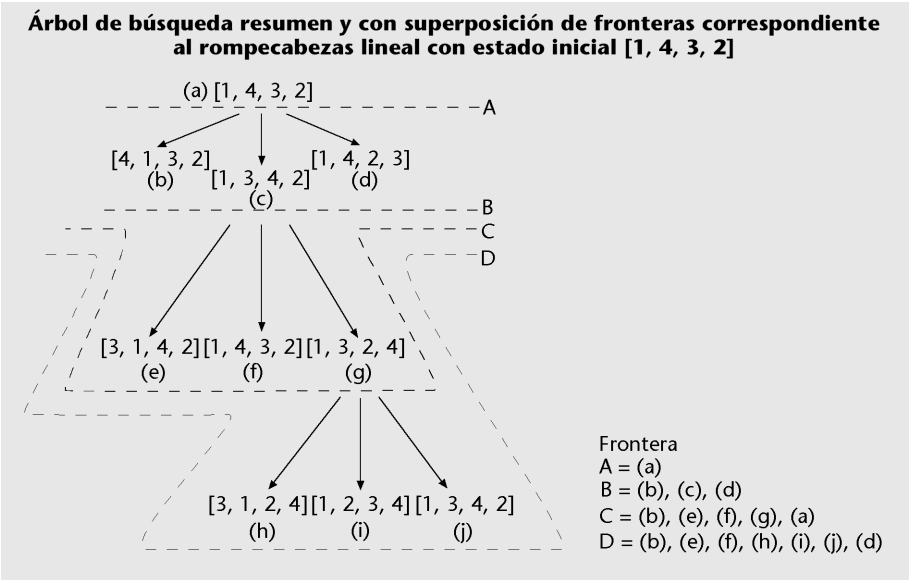
En la figura 7, se muestran los árboles de búsqueda que obtenemos siguiendo los mismos pasos que hemos considerado anteriormente. Los nodos del árbol disponen de un identificador del nodo (a), (b), ... (j) y de la información de los estados. Los arcos aparecen etiquetados con el nombre de la acción que permite pasar de un estado al siguiente.

Figura 7



En la figura 8, aparece otra vez el último árbol de búsqueda obtenido en los pasos sucesivos, ahora sobreponiendo los conjuntos frontera. Son las líneas A, B, C y D. Las fronteras son los puntos donde se tiene que elegir entre los caminos existentes. El nodo terminal del camino escogido se expande y el árbol crece con los nuevos estados que se han generado.

Figura 8



Se puede ver que en los sucesivos árboles de la figura 7, los caminos desde el nodo raíz a las hojas corresponden a los caminos que se han considerado en los pasos a, b, c y d. Así tenemos que en el paso (a) hay un único camino (que está vacío) del nodo inicial a sí mismo. En el paso (b) tenemos tres caminos, todos correspondientes a hacer una única acción, que llevan a [4, 1, 3, 2], [1, 3, 4, 2] y [1, 4, 2, 3] respectivamente. Estos caminos son los que aparecen en el segundo árbol de la figura 7, que representa los árboles de búsqueda obtenidos en los pasos sucesivos. En el paso (c) tenemos los caminos que corresponden a los que llevan a los nodos (b), (e), (f), (g), (d) del tercer árbol de la figura 7, que representa los árboles de búsqueda obtenidos en los pasos sucesivos. Lo mismo pasa con el paso (d) y el cuarto árbol de la figura 7.

A pesar de que el árbol de búsqueda tiene bastantes parecidos con el espacio de estados (de hecho, un nodo y su expansión se pueden superponer al espacio de estados), hay una diferencia importante: el grafo correspondiente en el espacio de estados tiene tantos nodos como estados hay, mientras que en el caso del árbol de búsqueda esto no es así, hay estados que pueden no aparecer y otros aparecer más de una vez. Por ejemplo, en la figura 8, el estado [1, 4, 3, 2] aparece en los nodos (a) y (f). Lo mismo pasa con las aristas. En la búsqueda podemos volver a considerar un operador aplicado a un estado al que ya se lo habíamos aplicado antes. Este sería el caso si expandiéramos el nodo (f).

En la descripción del esquema hay algunos elementos que quedan pendientes de clarificación. El más importante es la elección del nodo que tenemos que expandir. De hecho, es el punto más crítico del algoritmo, puesto que una buena elección nos permitirá llegar más rápidamente a la solución, ahorrándonos la expansión de otros muchos nodos. En los apartados siguientes se describen maneras de hacer estas elecciones. Pero antes vamos a ver una implementación del esquema general de la búsqueda.

2.1. La implementación

Para seguir el esquema planteado hasta ahora, la implementación de un algoritmo de búsqueda necesita unas estructuras de datos que correspondan al árbol de búsqueda y también las funciones para construir dicho árbol y operar con él. A continuación, hacemos una implementación de estos elementos primero en alto nivel y después con lenguaje Python. La implementación dada es genérica, y, por lo tanto, intenta ser apta para todos los algoritmos. Esto hace que no siempre sea la óptima.

Pasamos a continuación a la implementación empezando por la representación del árbol de búsqueda. El árbol necesita, en primer lugar, la estructura de datos correspondiente a un nodo. La estructura tiene que contener la información que necesitamos para reconstruir el camino que nos lleva desde el nodo inicial a un nodo solución. De hecho, dado que hasta que un nodo no es una solución no sabemos cuál de todos los posibles caminos es el bueno, es más sencillo considerar la reconstrucción del camino desde el nodo solución

al nodo inicial. Así, pues, consideramos que un nodo debe contener la información siguiente: qué otro nodo lo ha generado (su nodo padre) y cómo lo ha generado (el operador que aplicado al estado del padre retorna el estado del nodo que estamos considerando). Además, para simplificar el proceso de construcción del árbol, añadiremos el estado correspondiente a cada nodo. También incluiremos un identificador del nodo. Gracias a esto, tendremos que para saber quién es el padre de un nodo bastará con almacenar el identificador del padre. Además de toda esta información, algunos algoritmos de búsqueda necesitan información adicional de los nodos. Esta información también se guardará en el nodo.

De acuerdo con todo esto, tenemos que la estructura de datos de un nodo dispone de la información siguiente:

```
Estructura de datos nodo:
    identificador: etiqueta identificadora
    estado: representacion del nodo (array, struct, etc.)
    nodo-padre: identificador del padre
    operador-generador: identificador del operador
    informacion-adicional: estructura con informacion extra
```

El número de nodos que hay en el camino desde el nodo inicial o el coste del camino son ejemplos de información adicional.

La estructura de datos correspondiente al árbol de búsqueda tendrá que contener información de todos los nodos, diferenciando aquellos que ya han sido expandidos de aquellos que aún no lo han sido. Esto se consigue utilizando dos estructuras: una para cada conjunto de nodos.

1) Conjunto de nodos ya expandidos

Dado que las operaciones que hacemos con este conjunto son las de añadir nodos a la estructura (cuando expandimos un nodo, este tendrá que pasar de la estructura de nodos a expandir la de nodos expandidos) y obtener el nodo que corresponde a un determinado identificador (para poder reseguir el camino tenemos que poder encontrar el padre de un nodo a partir de su identificador), podemos usar simplemente una lista.

Lista en vez de tabla

De hecho, una implementación con una tabla –o una tabla de dispersión (según qué identificadores se usen)– es más eficiente. No obstante, consideramos el uso de una lista para reducir los elementos necesarios del lenguaje Python.

2) Conjunto de nodos a expandir

A este conjunto le tenemos que aplicar las operaciones de selección de un nodo (esto es, elegir uno que será el que expandiremos) y añadir un conjunto de nodos a la estructura (una vez expandido, sus hijos tienen que ser incluidos en la estructura para ser tratados más tarde). Para simplificar la operación de selección, se implementará la estructura mediante una cola ordenada (una cola con prioridades). De este modo, siempre seleccionaremos el primero de

la cola. En este caso, cuando añadimos nuevos nodos, se tendrán que colocar con cuidado para que queden en la posición más adecuada según un criterio determinado y fijado previamente.

De este modo, la estructura de datos del árbol de búsqueda tiene que ser la siguiente:

```
Estructura de datos árbol de búsqueda:
    Nodos a expandir: cola con prioridad de nodos
    Nodos ya expandidos: lista de nodos
```

Una vez tenemos la estructura de datos correspondiente al árbol de búsqueda, veamos el algoritmo para generar el árbol:

```
funcion busqueda(problema, estrategia-de-busqueda,
            arbol-de-busqueda-inicial) retorna solucion se
    arbol-de-busqueda := arbol-de-busqueda-inicial;
    solucion-encontrada := falso;
    mientras no (solucion-encontrada) y
            hay-nodos-para-expandir(arbol-de-busqueda) hacer
        nodo := seleccionar-nodo(arbol-de-búsqueda);
        añadir-nodo-a-ya-expandidos(nodo, arbol-de-busqueda);
        si solucion(problema, nodo)
            entonces solucion-encontrada := cierto;
            sino  nuevos-nodos := expandir(nodo, problema);
                  añadir-nodos-a-para-expandir (nuevos-nodos,
                                                estrategia-de-busqueda,
                                                arbol-de-busqueda);
        fsi;
    fmientras;
    si solucion-encontrada
        entonces retorna solucion(arbol-de-busqueda, nodo);
        sino retorna no-hay-solucion;
    fsi;
ffuncion;
```

La función recibe como parámetros tres elementos, el problema, la estrategia de búsqueda y el árbol de búsqueda inicial:

1) El problema, que contiene la información correspondiente al estado inicial, la función objetivo y los operadores.

2) La estrategia de búsqueda, que representa todo aquello que se refiere a la elección de un nuevo nodo.

3) El árbol inicial, que está formado por un único nodo que corresponde al estado inicial. De hecho, dado que hemos supuesto que la lista de nodos a expandir es una cola con prioridad y siempre se elige el primero, la estrategia es la función que determina cómo ordenar los nodos de la lista y no cómo se hace la selección.

A partir de estos parámetros, la función se ejecuta y lo hará siempre que no se haya encontrado ya la solución (esto lo marcará la variable booleana `solucion-encontrada`) y mientras queden nodos para expandir. Dado que estos nodos están almacenados en el árbol de búsqueda, la función `hay-nodos-para-expandir` se aplica a la variable que contiene el árbol.

En general, el procedimiento que se debe seguir es el siguiente:

1) Seleccionar el nodo (los posibles nodos están en el árbol).

2) Pasarlo al conjunto de nodos ya expandidos (estos nodos también se almacenan dentro del árbol de búsqueda).

3) Procesarlo según si es una solución o no. Para comprobar si es una solución, además del nodo, usaremos la función objetivo (que se almacena en la variable problema). Si el nodo no es una solución, tendremos que expandirlo (aquí usaremos el problema para saber qué operadores podemos aplicar al estado asociado al nodo) y añadir los nodos nuevos al árbol de búsqueda (la manera de añadirlos dependerá de la estrategia para que aquellos que se tengan que seleccionar primero queden más bien situados en la cola con prioridad).

En caso de acabar y haber encontrado una solución, el camino desde el estado inicial a la solución se construirá a partir del nodo solución y del árbol de búsqueda (el árbol contiene todos los nodos ya expandidos y, por lo tanto, todos los nodos desde el nodo inicial al final).

El algoritmo descrito aquí sigue, en líneas generales, lo que se ha explicado informalmente en el subapartado precedente, a pesar de que hay una diferencia importante. En la explicación del ejemplo, hemos detenido la operación cuando expandiendo el nodo [1, 3, 2, 4] hemos encontrado que uno de sus hijos ya corresponde a un estado solución. La función de búsqueda que presentamos aquí no se detendrá en ese punto. Una vez expandido el nodo [1, 3, 2, 4] y obtenidos los tres hijos [3, 1, 2, 4], [1, 2, 3, 4] y [1, 3, 4, 2], estos serán añadidos al árbol como nodos a expandir.

A continuación, la función procederá con la selección de otro nodo. Fijaos que la condición que activa la variable `solucion-encontrada` es que el nodo seleccionado corresponda a un estado objetivo y no a uno de los `nuevos-no-`

dos. Por lo tanto, no se detendrá hasta que al seleccionar un nodo, este sea la solución. Esta implementación que no sigue la apuntada anteriormente se hace para que algunos de los algoritmos de búsqueda que veremos más adelante sean óptimos. Es decir, no solamente encuentre una solución, sino que en el supuesto de que haya más de una, esta sea la óptima.

2.1.1. Implementación con Python

A continuación, vamos a hacer una implementación con Python de la función correspondiente al esquema general de búsqueda, junto con la de las funciones auxiliares y la de las estructuras de datos necesarios. Empezamos con la función búsqueda.

```
def busqueda (problema, estrategia, arbol):
    if (not candidatos(arbol)):
        return ['no_hay_solucion']
    else:
        nodo      = selecciona_nodo(arbol)
        nuevo_arbol = elimina_seleccion(arbol)
        if solucion(problema, nodo):
            return camino(arbol,nodo)
        else:
            return busqueda(problema,
                        estrategia,
                        expande_arbol(problema,
                                      estrategia,
                                      nuevo_arbol,
                                      nodo))
```

La versión Python del esquema general de búsqueda utiliza los mismos parámetros ya descritos antes: el problema, la estrategia y el árbol. De todas maneras, la forma de la función es ligeramente diferente, puesto que esta que se presenta aquí es recursiva. Así, primero se comprueba la condición de acabamiento: si no hay ningún nodo que se pueda expandir, habremos acabado (el caso de encontrar la solución se trata más adelante, y entonces la función devuelve el resultado y la recursión acaba).

La función `candidatos` es la que comprueba si hay nodos para expandir dentro del árbol de búsqueda y, si no hay, se acaba y se devuelve la lista con un único elemento `['no-hay-solucion']`. Cuando hay nodos para expandir, se selecciona uno y se define el nuevo árbol, en el que dicho nodo se extrae de la lista de nodos a expandir. La selección la hace la función `selecciona_nodo` y el nuevo árbol sin el nodo que se selecciona devuelve la función `elimina_seleccion`. Si el nodo seleccionado es la solución (la función `solucion` comprueba si este es el caso con la ayuda del problema

y el nodo en cuestión), entonces se ha de retornar el camino hasta el nodo solución. Esto lo hace la llamada `camino(arbol,nodo)`. Si no es así, se hace una llamada recursiva a la función búsqueda con el nuevo árbol resultado de la expansión del nodo seleccionado. La función `expande_arbol` es la que nos calcula la expansión del nodo incorporando los nuevos nodos al árbol.

La función `busqueda` se llamará desde una función que solo tiene como parámetros el problema y la estrategia y que, a partir del problema, crea el árbol inicial. Este árbol estará formado por el estado inicial definido en el problema.

```
def hacer_busqueda (problema, estrategia):
    return busqueda(problema,
                estrategia,
                arbol_inicial(estado_inicial(problema),
                    info_inicial(problema)))
```

Antes de pasar a las funciones relativas al árbol de búsqueda, consideremos la definición de la función `solucion`. Esta función, dado un nodo y un problema, comprueba si el primero es una solución del segundo. La función tiene la definición siguiente:

```
def solucion (problema, nodo):
    ff = funcion_objetivo(problema)
    return ff(estado(nodo))
```

Aquí utilizamos el hecho de que la función objetivo del problema es una función. Por tanto, solo hay que aplicar la función al estado asociado al nodo. Con `estado(nodo)` obtenemos el estado correspondiente al nodo y con `funcion_objetivo(problema)` obtenemos la función que nos dirá si el estado es una solución o no. Si se trata del problema del rompecabezas lineal definido en el subapartado «Análisis práctico del problema de las ocho reinas», tendremos que la función `ff` es `rl_funcion_objetivo`:

```
def rl_funcion_objetivo(estado):
    return estado == [1,2,3,4]
```

2.1.2. El árbol de búsqueda: representación y funciones

Ahora pasamos a la representación del árbol de búsqueda. Usamos la misma representación descrita en el subapartado «La implementación», una cola con prioridad por los nodos a expandir y una lista por los nodos ya expandidos. Tanto la cola como la lista se representarán mediante una lista Python. Así tenemos lo siguiente:

```
Estructura de datos arbol de busqueda::=
            [lista_nodos_a_expandir, lista_nodos_ya_expandidos]
```

Basándonos en esta estructura, podemos definir algunas de las funciones que han aparecido y que devuelven alguno de los elementos del árbol. Se incluyen aquí otras funciones que nos harán falta más adelante:

```
def nodos_a_expandir (arbol):
    return car(arbol)

def nodos_expandidos (arbol):
    return cadr(arbol)

def selecciona_nodo (arbol):
    return car(nodos_a_expandir(arbol))

def candidatos (arbol):
    return bool(nodos_a_expandir(arbol))
```

La definición de estas funciones se basa en la selección del elemento correspondiente de acuerdo con la estructura. Así, las funciones `nodos_a_expandir` y `nodos_expandidos` retornan las listas de nodos que hay en el árbol y `selecciona_nodo` devuelve el primer nodo de los que hay para expandir. `candidatos` comprueba que la lista de nodos para expandir no esté vacía. Solo hay que subrayar que la selección del nodo cogerá siempre el primer elemento de la lista correspondiente a los nodos a expandir porque, como ya se ha dicho, esta corresponde a una cola con prioridades.

Otra función relacionada con el árbol –y que aparece en la función `busqueda` cuando encontramos la solución– es la que nos retorna el camino correspondiente al nodo.

```
def camino (arbol, nodo):
    if not id_padre(nodo):
```

```
        return []
    lp = camino(arbol, nodo_arbol(id_padre(nodo), arbol))
    return lp + [operador(nodo)]
```

Esta función reconstruye el camino del nodo que se nos da hasta la solución. Y como que dado un nodo, podemos conocer su padre, tenemos que todo el camino lo podemos escribir como el camino del nodo inicial hasta el padre del nodo actual y el operador que nos ha permitido pasar del padre al nodo actual. Así, `camino(arbol,nodo_arbol(id_padre(nodo),arbol))` corresponde al camino hasta el padre y `[operador(nodo)]` es una lista con el operador que, aplicado al padre, genera el `nodo`. La función usa la función `id_padre`, la cual, dado un nodo, retorna el identificador del padre y otra llamada `nodo_arbol`, que, dado un identificador –y el árbol–, nos encuentra el nodo correspondiente. Dado que esta última función forma parte de la estructura árbol, la veremos a continuación:

```
def nodo_arbol (id_nodo, arbol):
    check_nodo = lambda nodo: ident(nodo) == id_nodo
    a_expandir = member_if(check_nodo, nodos_a_expandir(arbol))
    if bool(a_expandir):
        return car(a_expandir)
     return find_if(check_nodo, nodos_expandidos(arbol))
```

Para encontrar el nodo, esta función primero mira si el nodo está entre los nodos a expandir y, si es así (si la variable `a_expandir` no es nula), hace la selección del nodo entre los que se tienen que expandir. Si no es así, la selección se hará entre los nodos ya expandidos. Si el nodo tampoco está en la segunda lista, la segunda selección devolverá `[]`. Las funciones que usamos aquí de `member_if` y `find_if` comprueban y seleccionan si hay un elemento que satisfaga una propiedad (definidas en el apéndice).

Ahora pasamos a la función `expande_arbol` que expande el árbol. Se puede ver que primero se calculan los nodos que se han de expandir y que después serán incorporados al árbol con la función `construye_arbol`. La función que construye el árbol necesita, además del árbol, el nodo que hemos expandido y los nuevos nodos generados. Además, para que a la hora de situar los nuevos nodos en la cola de nodos a expandir esto se haga correctamente, necesitamos la estrategia. Dado que consideramos que la estrategia es una función, la llamamos indicándole como parámetros la cola de nodos a expandir ya existentes y los nuevos nodos.

En en cuanto a la expansión del nodo, esta función usa, además de los nodos, los operadores del problema y –para algunos tipos de búsqueda– cierta información adicional asociada al problema.

```
def expande_arbol (problema, estrategia, arbol, nodo):
    nuevos_nodos_a_expandir = expande_nodo(nodo,
                                           operadores(problema),
                                           funcion_info_adicional(problema))
    return construye_arbol(arbol,
                           estrategia,
                           nodo,
                           nuevos_nodos_a_expandir)

def construye_arbol (arbol, estrategia, nodo_expandido, nuevos_nodos_a_expandir):
    elm = estrategia(car(arbol), nuevos_nodos_a_expandir)
    return cons(elm, [cons(nodo_expandido, cadr(arbol))])
```

Para completar las funciones que actúan en el árbol nos falta la que elimina el nodo seleccionado llamada `elimina_seleccion`, y la que nos genera el árbol inicial del problema.

```
def elimina_seleccion (arbol):
    return cons(cdr(nodos_a_expandir(arbol)), cdr(arbol))

def arbol_inicial (estado, info):
    infres = info(estado)
    nodo = construye_nodo(gensym(), estado, [], [], [])
    tmp = [nodo + infres]
    return [tmp]
```

La función `elimina_seleccion` construye un nuevo árbol (una lista) a partir de los nodos a expandir (después de eliminar el primero aplicando `cdr` a la lista resultante de llamar `nodos_a_expandir`) y de los nodos ya expandidos (la cola del árbol). La función `arbol_inicial` construye un nuevo árbol donde solo hay un nodo a expandir que tiene como estado el que se le indica a la función. Esta función llama a la función que calcula la información adicional para el estado inicial.

2.1.3. Los nodos: representación y funciones

La representación de los nodos sigue lo que se ha propuesto anteriormente: un identificador, el estado, el identificador del nodo padre y el operador que lo ha generado. Además, a pesar de que hoy por hoy no hace falta, conside-

raremos que es posible que el nodo contenga más campos correspondientes a información adicional (por ejemplo, para guardar el coste de llegar hasta el nodo). Para mantener toda esta información, usaremos una lista en la que los elementos que aparezcan corresponderán a los especificados anteriormente y con el mismo orden. Así, tendremos que la estructura es como sigue:

```
nodo ::=
    [Identificador, Estado, Identificador-Nodo-Padre,
    Operador-generador, Otros...]
```

Basándonos en esto, definimos un conjunto de funciones consultoras que, dado un nodo, nos devuelvan los elementos de estos nodos que podemos necesitar en la búsqueda. Estas funciones son:

```
def ident (nodo): return car(nodo)

def estado (nodo): return cadr(nodo)

def id_padre (nodo): return caddr(nodo)

def operador (nodo): return car(cdddr(nodo))

def info (nodo): return cdr(cdddr(nodo))
```

La función `info` (de información) nos devolverá simplemente la cola final de la lista. Si hay más de un elemento, la función nos retornará todos los que haya. Las otras funciones nos devuelven cada una un único elemento, que puede ser una lista. Este será el caso que tendremos, por ejemplo, si estamos buscando soluciones para el problema del rompecabezas lineal, en el que un estado es una lista.

Además de las funciones para hacer consultas de un nodo, necesitamos una función que nos permita construir un nodo. Esto nos hace falta en el momento de la expansión (llamada a la función `expande_nodo`) y de crear el árbol inicial (llamada a la función `arbol_inicial`). A esta función le indicamos cuatro elementos y una lista con la información adicional. Como antes, esta última información es una lista para no limitar el número de elementos.

```
def construye_nodo (ident, estado, id_padre, op, info):
    return [ident, estado, id_padre, op] + info
```

Por ejemplo, si queremos generar el nodo inicial correspondiente al estado [2, 4, 1, 3], haremos la llamada:

```
construye_nodo(gensym(), [2, 4, 1, 3], [], [], [])
```

Si queremos construir el mismo nodo, pero de forma que añadimos información diciendo que el coste de este nodo es cero y que su estado corresponde al estado inicial, podemos hacer:

```
construye_nodo(gensym(), [2, 4, 1, 3], [], [], [0, estado_inicial])
```

Como hemos visto, además de estas funciones, hace falta una función que realice la expansión de un nodo. Esta tarea la hace la función `expande_nodo`, que llama a la función `expande_arbol` que hemos visto anteriormente.

```
def expande_nodo (nodo, operadores, funcion):
    def elimina_estados_vacios (lista_nodos):
        return remove_if(lambda nodo: estado(nodo) == 'vacio',
                         lista_nodos)

    st        = estado(nodo)
    id_nodo   = ident(nodo)
    info_nodo = info(nodo)
    aux       = []
    for op in operadores:
        nuevo_simbolo = gensym()
        ff         = cadr(op)
        ffapp      = ff(st,info_nodo)
        aux.append(construye_nodo(nuevo_simbolo,
                                  ffapp,
                                  id_nodo,
                                  car(op),
                                  funcion([st, info_nodo],
                                          ffapp,
                                          car(op))))
    return elimina_estados_vacios(aux)
```

Esta función construirá un nodo para cada operador (aplicando cada operador al nodo que estamos expandiendo) y, a continuación, eliminará aquellos que están vacíos (o son erróneos). La eliminación la hace la función

elimina_estados_vacios, que se define localmente. Esta función es para resolver el caso de los movimientos que no se pueden hacer en un determinado estado (por ejemplo, eliminar una pieza cuando esta no existe).

La constante 'vacio' define qué se entiende por un estado que no se ha podido generar. La función que aplica cada operador al estado es una función lambda, que para cada operador genera un nodo con la función construye_nodo. Cada nuevo nodo tiene un identificador generado con gensym(), el nuevo estado que se obtiene con ff(st, info_nodo), donde ff es la función en cadr(op), el identificador del nodo que es su padre (el identificador del nodo en curso id_nodo), el nombre del operador car(op) y la nueva información asociada al nodo.

2.1.4. El problema: representación y funciones

A la hora de aplicar el esquema de búsqueda a un ejemplo concreto, se ha considerado que el problema tiene la estructura siguiente:

```
problema ::=
    [operadores, funcion, estado-inicial,
        funcion-objetivo, info-inicial...]
```

y de acuerdo con esto se han definido las funciones operadores, funcion_info_adicional, estado_inicial, e info_inicial tal como sigue:

```
def operadores (problema): return car(problema)

def funcion_info_adicional (problema): return cadr(problema)

def estado_inicial (problema): return caddr(problema)

def funcion_objetivo (problema): return car(cdddr(problema))

def info_inicial (problema): return car(cdr(cdddr(problema)))
```

2.2. Algunas consideraciones adicionales

El esquema de búsqueda presentado aquí no detalla cuál es la estrategia que tenemos que usar para seleccionar el nodo que tenemos que expandir. En los apartados que siguen se presentan diferentes alternativas para hacerlo. Para poderlas evaluar, utilizaremos varias propiedades. Por ejemplo, podemos considerar la completitud, la optimalidad, la complejidad en cuanto al tiempo y

la complejidad en cuanto al espacio. Las dos primeras son para saber si cuando hay una solución, la encontraremos, y cuando hay más de una, encontraremos la solución de más calidad. En este caso, hace falta que el problema defina la manera de evaluar las soluciones (por ejemplo, mediante las funciones de coste). Las dos últimas corresponden al tiempo y a la memoria que necesitan los algoritmos para encontrar la solución. Estos costes en tiempo y memoria se expresan en función del factor de ramificación, la longitud del camino solución, etc.

Además de estas propiedades, hay otra que interesa cuando la búsqueda se aplica a un entorno en tiempo real. Son los **algoritmos *anytime***[2]. Estos métodos de búsqueda retornan una solución para cualquier asignación de tiempo de computación y se espera que, mientras más tiempo de actuación tienen, mejor será la solución que retornarán. Para algunos de estos algoritmos se sabe que, con suficiente tiempo, encontrarían la solución óptima.

(2)En inglés, *anytime algorithms*.

Las propiedades mencionadas se definen a continuación:

1) Completitud: un algoritmo de búsqueda es completo si, cuando un problema tiene solución, este la encuentra.

2) Optimalidad: un algoritmo de búsqueda es óptimo cuando en un problema con diferentes soluciones siempre encuentra la solución de más «calidad».

3) Complejidad en cuanto al tiempo: el tiempo que tarda el algoritmo en encontrar la solución.

4) Complejidad en cuanto al espacio: la memoria que necesita el algoritmo para poder encontrar la solución.

5) Algoritmos *anytime*: un algoritmo de búsqueda es *anytime* cuando puede encontrar una solución para cualquier asignación de tiempo de computación.

La cuestión de los estados repetidos

Ya habéis visto que el árbol de búsqueda y el espacio de estados tienen bastantes parecidos, pero no son iguales porque el grafo correspondiente al espacio de estados tiene tantos nodos como estados, pero el árbol de búsqueda no porque puede haber estados que aparecen más de una vez. El hecho de tener estados repetidos hace que los algoritmos de búsqueda sean ineficientes porque expanden algunos subárboles más de una vez y, además, puede provocar que el algoritmo nunca acabe porque un estado se expande de manera repetida. Para evitarlo, se puede comprobar si un estado ya ha aparecido antes, pero esto introduce un sobrecoste adicional en el método. Por tanto, debe existir un compromiso entre el sobrecoste de comprobar si un estado se repite y que el

algoritmo acabe. En general, se pueden considerar tres alternativas para evitar la repetición de estados. Las presentamos ordenadas de menor a mayor coste de comprobación:

1) No permitir el retorno al estado del que venimos: hace falta que la función que expande un nodo compruebe que el estado correspondiente al nodo generado no coincide con el padre del nodo que estamos expandiendo. Esta comprobación se puede hacer en tiempo constante.

2) No permitir generar caminos con ciclos: hace falta que la función que expande un nodo compruebe que el estado correspondiente al nodo generado no coincide con ninguno de los antecedentes del nodo que estamos expandiendo. Esta comprobación se hará en tiempo lineal (en relación con la longitud del camino).

3) No generar ningún estado que ya haya sido generado: esto obliga a comprobar todos los estados del árbol de búsqueda. Por lo tanto, la complejidad será del orden de O(s), donde s es el número de estados en el espacio de estados.

Es importante subrayar que no es indispensable almacenar los nodos ya expandidos para representar el árbol de búsqueda, ni tampoco para poder reconstruir el camino cuando hayemos la solución. Por lo tanto, almacenar estos nodos representa un coste adicional de memoria.

En general, el uso de una técnica u otra dependerá del problema. De hecho, cuantos más estados repetidos aparezcan, más útil será almacenarlos y utilizar el tiempo para comprobar que un estado no haya aparecido antes. Además, se ha de tener en cuenta que según cuál sea la implementación elegida para los estados y el número de estados diferentes, se pueden tener métodos más o menos costosos para comprobar si un estado ya ha sido visitado o no.

3. Estrategias de búsqueda no informada

El esquema general de búsqueda planteado anteriormente no concreta qué nodo tenemos que expandir, es decir, cómo tenemos que ordenar los nodos en la lista de nodos a expandir. A continuación, veremos algunos de los métodos que se pueden utilizar para hacer esta expansión. Primero, en este apartado veremos métodos que como única información para decidir qué nodo hay que expandir y solo usan los operadores que se pueden aplicar a un determinado estado. Veremos, por ejemplo, la búsqueda en anchura y profundidad. En el próximo apartado, en cambio, se verán métodos que usan información adicional relativa al problema.

Ahora veremos dos algoritmos de búsqueda: búsqueda en anchura y búsqueda en profundidad.

3.1. Búsqueda en anchura

La búsqueda en anchura corresponde a hacer un recorrido del grafo de estados por niveles. Esto es, primero se visitan todos aquellos estados a los que se puede acceder desde el estado inicial aplicando solo un operador, después se visitan todos los estados a los que se accede aplicando dos operadores al estado inicial, y se sigue con aquellos que necesitan la aplicación de tres operadores para acceder y, así, sucesivamente.

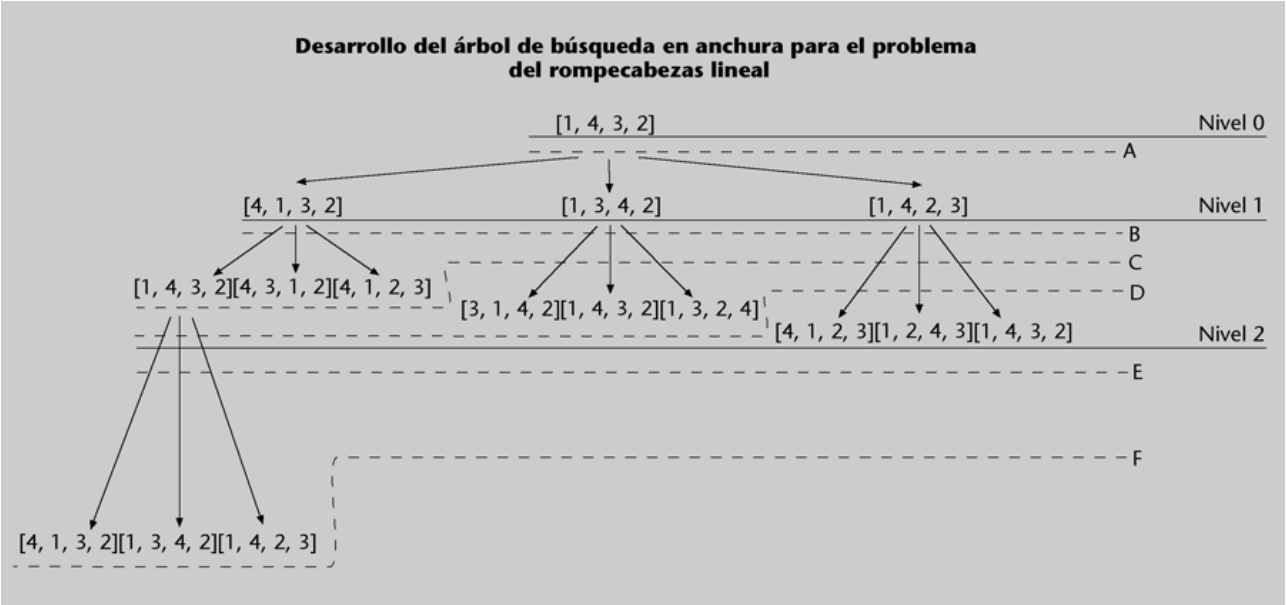
Formalmente, el algoritmo espera a expandir un nodo de un nivel determinado hasta el momento en que todos los nodos del nivel anterior ya se han expandido. Esto se puede implementar como un refinamiento del esquema general de búsqueda en el que la estrategia es almacenar el nuevo conjunto de nodos (los expandidos a partir del nodo seleccionado) después de todos los nodos que todavía quedan pendientes. Observad que, en un momento dado, los nodos todavía pendientes solo pueden ser de dos niveles consecutivos. O son del nivel d o del nivel $d + 1$. Si hay de los dos niveles, los del $d + 1$ quedarán detrás de los del nivel d. Así, pues, en general el algoritmo accede a un nodo de nivel d y los nodos resultado de la expansión (nodos del nivel $d + 1$) quedarán al final de la lista.

En la figura 9 aparece el desarrollo del árbol de búsqueda partiendo del estado inicial [1, 4, 3, 2]. Evidentemente, en este nivel la frontera es A. La inicialización del árbol nos da un único nodo que corresponde al estado [1, 4, 3, 2]. La expansión de este estado nos da tres nuevos estados, que son [4, 1, 3, 2], [1, 3, 4, 2], [1, 4, 2, 3].

Ahora la frontera del árbol de búsqueda es B. A continuación, expandiremos el primero de estos nodos y obtendremos la frontera C. Seguidamente, cuando seleccionamos un nuevo nodo para expandir, tenemos que escoger uno de los que está en el nivel anterior. Tomamos el nodo con el estado [1, 3, 4, 2], que cuando lo expandimos obtenemos la frontera D. El nodo siguiente que seleccionamos es el [1, 4, 2, 3], que es el único que queda en un nivel anterior. Cuando lo expandimos, obtenemos la frontera E.

Para implementar este mecanismo con el esquema general de búsqueda, tenemos que añadir los nodos resultantes de la expansión al final de la lista de nodos a expandir. Por lo tanto, la función estrategia es concatenar la nueva lista por el final.

Figura 9



Dado que en la búsqueda en anchura los nodos se van expandiendo por niveles, se hace un recorrido sistemático del árbol. Esto hace que, si hay una solución, el algoritmo la encontrará cuando llegue al nivel correspondiente. Por eso, este esquema de búsqueda es completo. La optimalidad, en cambio, solo ocurre si el coste del camino es una función no decreciente de la profundidad del nodo. Esto es, que si hemos encontrado una solución en el nivel d, no haya ninguno más bueno a profundidades mayores que d.

La complejidad en cuanto al tiempo del algoritmo es exponencial: si tenemos que el factor de ramificación (el número de operadores) es b y la solución está a la profundidad d, para encontrar la solución tendremos que haber expandido todos los nodos hasta el nivel d. Por tanto, dado que $1 + b + b^2 + b^3 + ... + b^d = O(b^d)$, la complejidad será del orden de b^d.

Para determinar la complejidad en en cuanto a la memoria, hemos de tener en cuenta el número de nodos que tendremos en la lista de nodos a expandir. En este método, el número de nodos dependerá del nivel. De hecho, en el nivel *i*-ésimo tendremos b^i nodos (podéis ver que en la figura anterior tenemos 3^0 nodos en la frontera del nivel 0 –la frontera A–, 3^1 en la del nivel 1 –la frontera B–, 3^2 en la del nivel 2 –la frontera E–...). Dado que la solución está en el nivel *d*, cuando encontremos la solución, habrá en la memoria del orden de $O(b^d)$ nodos.

3.1.1. Implementación

Cuando utilizamos la estrategia de búsqueda en anchura tenemos que expandir todos los nodos de un determinado nivel antes de pasar a los nodos del nivel siguiente. Para implementar esta estrategia, basta con que la función que la implementa coloque los *n* nodos resultado de la expansión detrás de los que tenemos en la lista de nodos a expandir.

Esto se puede implementar con Python mediante una concatenación de la lista correspondiente a los nodos a expandir y la lista con los nuevos nodos:

```
def rl_estrategia_anchura (nodos_a_expandir, nuevos_nodos_a_expandir):
    return nodos_a_expandir + nuevos_nodos_a_expandir
```

Así, pues, la búsqueda en anchura la podemos definir como:

```
def busqueda_anchura (problema):
    return hacer_busqueda(problema, rl_estrategia_anchura)
```

3.2. Búsqueda en profundidad

La búsqueda en profundidad siempre expande los nodos que están en un nivel más profundo dentro del árbol de búsqueda. Solo cuando se llega a un nodo al que no se le puede aplicar ningún operador, se elegirá un nodo de otro camino.

Para implementar este método con el esquema general de búsqueda, añadiremos los nodos resultado de la expansión delante de los nodos a expandir. De este modo, los nodos situados a más profundidad estarán siempre delante y se seleccionarán primero. La implementación con Python de la función de estrategia correspondiente será:

```
def rl_estrategia_profundidad (nodos_a_expandir, nuevos_nodos_a_expandir):
```

```
    return nuevos_nodos_a_expandir + nodos_a_expandir
```

Con la función de estrategia podemos definir la función de búsqueda en profundidad y, a continuación, hacer la llamada con el problema del rompecabezas lineal que hemos definido en el subapartado «Análisis práctico del problema de las ocho reinas»:

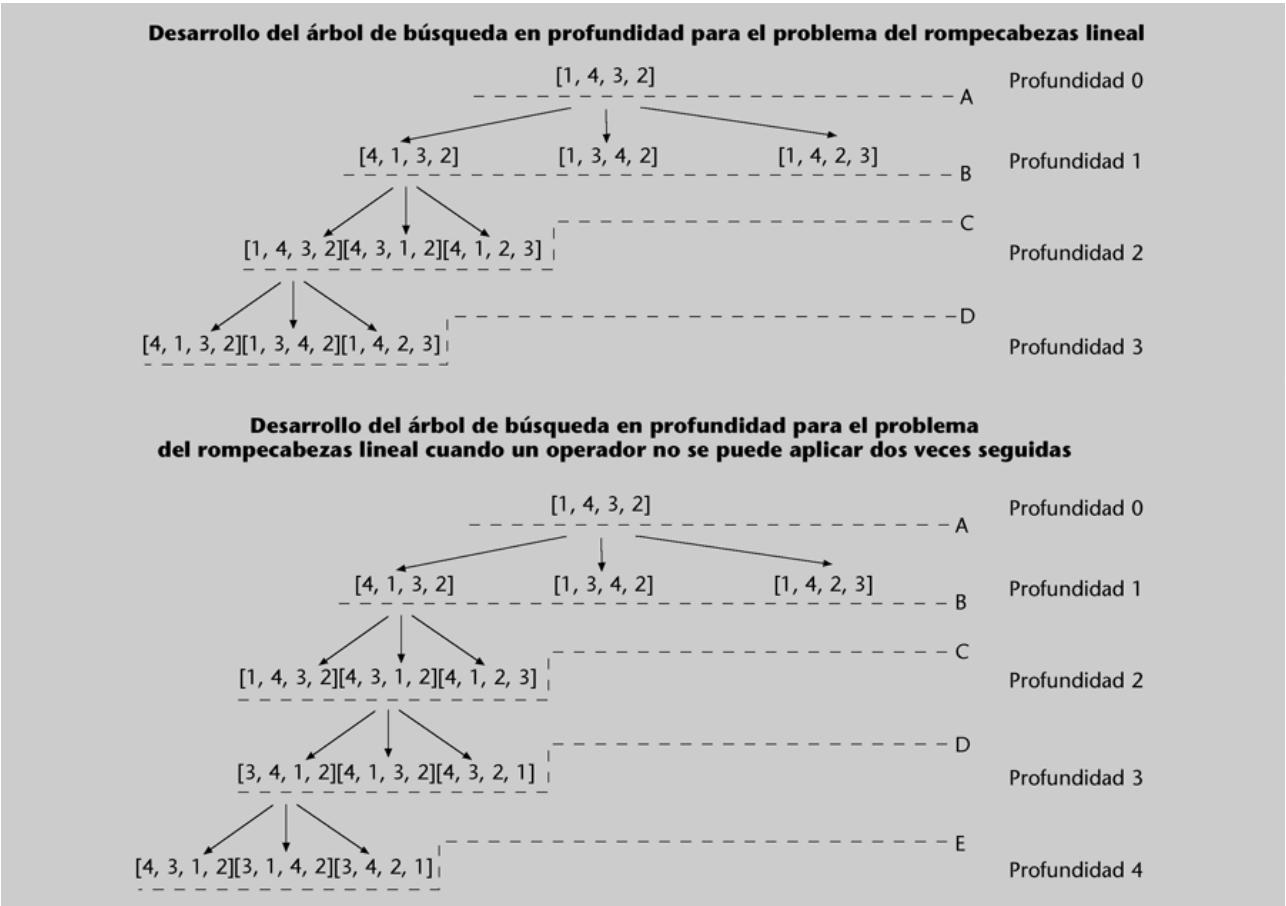
```
def busqueda_profundidad (problema):
    return hacer_busqueda(problema, rl_estrategia_profundidad)
```

Para realizar la búsqueda en profundidad del problema haremos:

```
busqueda_profundidad(problema())
```

Sin embargo, esta llamada no retorna ningún resultado porque la función queda colgada con el concatenamiento de movimientos IE.

Figura 10



En la figura 10, se representa el desarrollo del árbol de búsqueda en profundidad correspondiente al problema del rompecabezas lineal definido en la variable `problema`. Se puede observar que la búsqueda va profundizando en un mismo camino. Además, también se puede ver que el mecanismo de búsqueda no es completo porque en determinadas condiciones no acaba. Este es el caso cuando el sistema empieza a repetir los estados [1, 4, 3, 2], [4, 1, 3, 2]. En el segundo esquema de la figura, se representa el desarrollo del árbol de búsqueda en profundidad correspondiente al mismo problema cuando los operadores se restringen de forma que no se puedan aplicar dos veces seguidas (o lo que es lo mismo, no se pueda repetir el estado del padre). Si seguimos la expansión del árbol, veremos que también se entra en un ciclo.

A pesar de que no sea completa, la búsqueda en profundidad tiene una gran ventaja en relación con la búsqueda en anchura: tiene una complejidad menor en cuanto a memoria. En los dos árboles de la figura 10, se puede ver que la frontera de los árboles de búsqueda sucesivos no son exponenciales como en el caso de la búsqueda en anchura, sino que son lineales en relación con la profundidad. Así, en general, si tenemos que un problema tiene un factor de ramificación b y disponemos de operadores hasta una profundidad m (suponemos que se puede encontrar la solución en una profundidad menor que m), tendremos que la complejidad en cuanto al espacio es de $O(bm)$. Observad que en el primer árbol de la figura 10 tenemos una profundidad igual a tres: (3 – 1) + (3 – 1) + 3 nodos a expandir y que, en la figura que representa el desarrollo del árbol de búsqueda en profundidad, cuando un operador no se puede aplicar dos veces seguidas, tenemos una profundidad igual a cuatro: (3 – 1) + (3 – 2) + (3 – 1) + 3 nodos.

La complejidad en cuanto al tiempo, en cambio, será parecida al caso de la búsqueda en anchura, $O(b^m)$, aunque se puede encontrar la solución con menos tiempo. Esto será así porque el árbol se irá abriendo a medida que llegamos a nodos que no se pueden expandir. No obstante, esto no afectará a la memoria, porque no hay que almacenar la estructura de los nodos ya expandidos (y todavía menos si ya sabemos que no llevan a la solución).

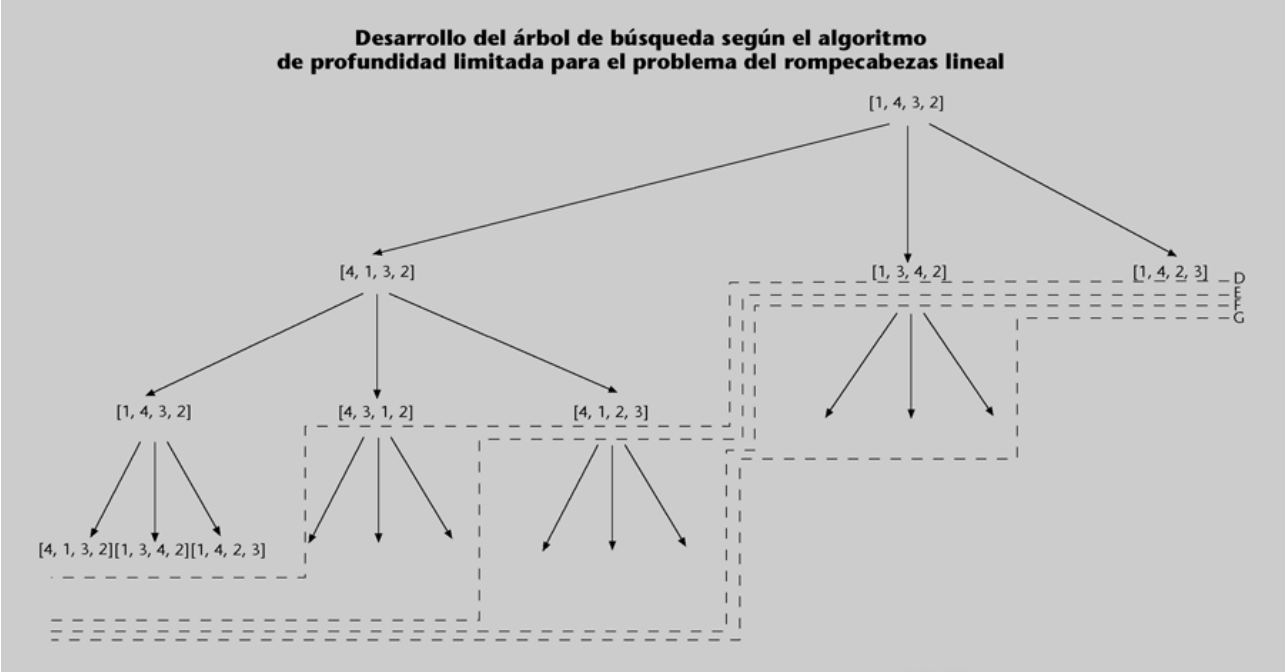
Dado que el coste de memoria de este esquema de búsqueda es sustancialmente mejor que el de la búsqueda en anchura, se han desarrollado métodos para aprovecharlo, pero sin caer en la no completitud del método. A continuación veremos dos aproximaciones.

3.2.1. Búsqueda en profundidad limitada

En este caso se aplica el algoritmo de búsqueda en profundidad, pero fijando la profundidad máxima a la que se puede llegar. Así, el algoritmo funciona del mismo modo que el método descrito anteriormente, pero ahora cuando se

llega a la profundidad prefijada, en lugar de aplicar el operador, se considera que este ya no se puede aplicar. Esto fuerza a recular y escoger un nodo de un nivel menos profundo.

Figura 11



En la figura 11, se presenta el desarrollo del árbol de búsqueda visto anteriormente para un caso del problema del rompecabezas lineal con una profundidad máxima igual a 3. Se proporcionan las fronteras del árbol de búsqueda, que seguirán las que ya hemos visto en la figura 10. Se puede ver que en todos los caminos la búsqueda se detiene en un mismo nivel.

El hecho de introducir la profundidad máxima permite que en determinadas situaciones el método sea completo (aunque no siempre lo será). Así, pues, tendremos que la búsqueda con una profundidad máxima p es completa cuando hay una solución a menos profundidad. La complejidad para este método es análoga al caso anterior, tomando ahora p en lugar de m (m era la profundidad máxima que permitían los operadores). Por tanto, la complejidad en cuanto al tiempo será $O(b^p)$ y la complejidad en cuanto al espacio será $O(bp)$.

A continuación, se indican las funciones que implementan este método de búsqueda. El control de la profundidad lo obtenemos al asociar a cada nodo su profundidad e impidiendo que el nodo se expanda cuando ya esté a la profundidad máxima (si la profundidad es superior o igual al `limit` –definimos esto como una variable global–, el movimiento no es permitido y la función devuelve entonces un movimiento vacío). Esta asociación se implementa mediante la información adicional del nodo que ya habíamos comentado en el subapartado «Análisis práctico del problema de las ocho reinas».

Movimiento vacío

La función de expandir los nodos `expande_nodo` ya tiene en cuenta cuando un nodo no se puede expandir, como ya hemos visto en el subapartado «La implementación».

Así tendremos que los operadores serán:

```
limit = 0 ## variable global, valor inicial arbitrario

def mov_ie_profLim (estado, info):
    if car(info) < limit: return [cadr(estado), car(estado), caddr(estado), cadddr(estado)]
    return 'vacio'

def mov_ic_profLim (estado, info):
    if car(info) < limit: return [car(estado), caddr(estado), cadr(estado), cadddr(estado)]
    return 'vacio'

def mov_id_profLim (estado, info):
    if car(info) < limit: return [car(estado), cadr(estado), cadddr(estado), caddr(estado)]
    return 'vacío'

def rl_operadores_profLim():
    return [['ie', mov_ie_profLim], ['ic', mov_ic_profLim], ['id', mov_id_profLim]]
```

Aquí `'vacio'` es una constante que define el estado que no se ha podido generar (o estado erróneo).

Con estos operadores se puede definir el problema de una manera similar a como se ha hecho antes (subapartado «Análisis práctico del problema de las ocho reinas»). Se ha de tener en cuenta que ahora tenemos que inicializar el nodo correspondiente al estado inicial de forma que tenga asociado el valor de su profundidad (esto es, cero). Esta función será: `lambda estado: [estado, [0]]`. También tenemos que definir una función que calcule la profundidad del nuevo nodo a partir del nodo que expandimos. Esto lo define la función:

```
def auxf(info_nodo_padre, estado, nombre_operador):
    info_nodo = cadr(info_nodo_padre)
    return [estado, [1 + caadr(info_nodo)]]
```

El resto de la definición es como antes, teniendo en cuenta que ahora los operadores son resultado de la llamada `rl_operadores_profLim`.

```
def problema_profLim():
    def auxf(info_nodo_padre, estado, nombre_operador):
        info_nodo = cadr(info_nodo_padre)
        return [estado, [1 + caadr(info_nodo)]]
```

```
    return [rl_operadores_profLim(),
            auxf,
            [4,3,2,1],
            (lambda estado: estado == [1,2,3,4]),
            (lambda estado: [estado, [0]])]
```

Con estas funciones definimos la búsqueda en profundidad limitada a partir de un problema y un límite de la manera siguiente:

```
def busqueda_profundidad_limitada (problema, lim):
    global limit
    limit = lim
    return hacer_busqueda(problema, rl_estrategia_profundidad)
```

Para llamar la función con el problema que hemos definido y una profundidad máxima de diez, tenemos que hacer:

```
busqueda_profundidad_limitada(problema_profLim(), 10)
```

Esta llamada dará el resultado siguiente:

```
['ie', 'ie', 'ie', 'ie', 'ie', 'ic', 'ie', 'id', 'ic', 'ie']
```

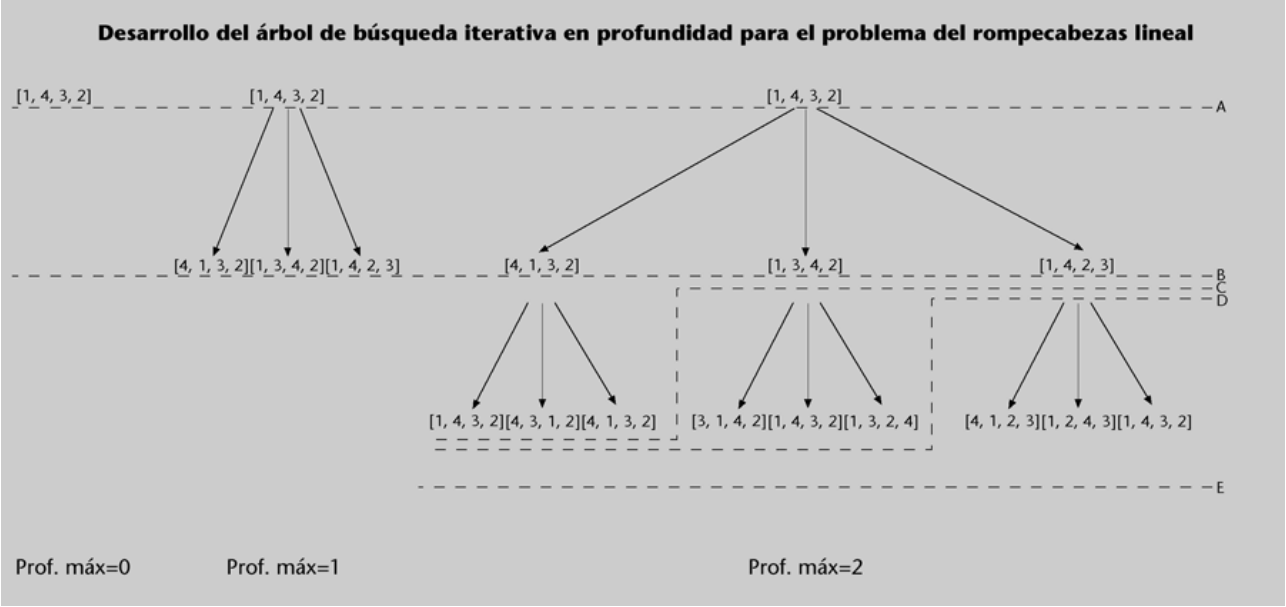
Los primeros cinco movimientos IE se deben al hecho de que, en realidad, hacemos una búsqueda en profundidad y, dado que está limitada, vuelve atrás y encuentra una solución usando otros operadores. Recordad que la búsqueda en profundidad para este mismo problema se quedaba colgada aplicando siempre el operador IE.

Dado que, en general, no se conoce en qué nivel está la solución, es difícil y problemático fijar *a priori* el nivel máximo para el algoritmo. Una alternativa a este problema es la llamada *búsqueda iterativa con profundidad*.

Búsqueda iterativa con profundidad

La estrategia consiste en hacer varias búsquedas con una profundidad limitada, utilizando cada vez valores crecientes de la profundidad máxima.

Figura 12



En la figura 12, se presentan los diferentes árboles de búsqueda (y su desarrollo) para el problema del rompecabezas lineal. En cada uno de los árboles tenemos una búsqueda en profundidad limitada con una profundidad máxima diferente.

Evidentemente, este procedimiento es completo, porque llegará un momento en el que la profundidad máxima permitida igualará la profundidad en la que está la solución.

En cuanto a los costes, hay que diferenciar entre el tiempo y la memoria. Dado que para cada profundidad máxima la búsqueda empieza de nuevo, la memoria necesaria corresponde a la que se necesita para la búsqueda en profundidad limitada cuando la profundidad corresponde a la profundidad de la solución. Así, tenemos que el coste de memoria será O($b\,d$). Para determinar la complejidad en cuanto al tiempo, se ha de tener en cuenta que ahora hay nodos que se expanden más de una vez. Si suponemos que la solución es a una profundidad d, tendremos que deberemos expandir todos los árboles hasta una profundidad máxima igual a d. Por tanto, el número de nodos que expandiremos será:

$$1 + b + b^2 + \ldots + b^{d-2} + b^{d-1} + \mathrm{b}^d$$

Esta expresión se puede reescribir como:

$$(d + 1) \cdot 1 + d \cdot b + (d - 1) \cdot b^2 + \ldots + 3 \cdot b^{d-2} + 2 \cdot b^{d-1} + b^d$$

Por tanto, el coste será: O(b^d).

El cociente de las dos expresiones nos dará el sobrecoste de repetir la búsqueda. Este cociente es aproximadamente $1 + 1/b$, lo cual muestra que el sobrecoste es pequeño ($100/b$ % aproximadamente) y que, cuanto mayor sea b, menor será. Por ejemplo si tomamos $b = 10$ y una profundidad máxima de 5, tendremos que la búsqueda iterativa en profundidad visitará 123.456 nodos, mientras que si visitamos los nodos solo una vez (el caso de la búsqueda en anchura) visitaremos 111.111 nodos. Esto representa un sobrecoste del 11 %. Con $b = 20$ y la misma profundidad, visitaremos 3.545.706 con la búsqueda iterativa y 3.368.421 con la búsqueda en profundidad. Por tanto, tendremos que el cociente es 1,052 y el sobrecoste será del 5,2 %.

Sobrecoste

Estos cálculos son, evidentemente, suponiendo que la búsqueda con profundidad máxima la hacemos con una profundidad igual a *b*. Esto solo será posible si conocemos *b*; sino haríamos una búsqueda a mayor profundidad para asegurarnos de encontrar la solución y, por tanto, el ahorro (si lo hay) sería muy menor.

Para implementar la búsqueda iterativa en profundidad utilizaremos una función que realiza la búsqueda iterativa desde una determinada profundidad (es la función `busqueda_iterativa_profundidad_desde_k`). En este caso, la búsqueda iterativa empezará desde la profundidad cero y si en esta profundidad no está la solución empezará la búsqueda desde la profundidad siguiente. Una implementación de estas dos funciones es la siguiente:

```
def busqueda_iterativa_profundidad (problema):
    return busqueda_iterativa_profundidad_desde_k(problema,0)

def busqueda_iterativa_profundidad_desde_k (problema, lim):
    resultado = busqueda_profundidad_limitada(problema, lim)
    if resultado == ['no_hay_solucion']:
        return busqueda_iterativa_profundidad_desde_k(problema, lim+1)
    return resultado
```

La llamada a la función se hará utilizando el problema donde la expansión tiene en cuenta la profundidad máxima:

```
busqueda_iterativa_profundidad(problema_profLim())
```

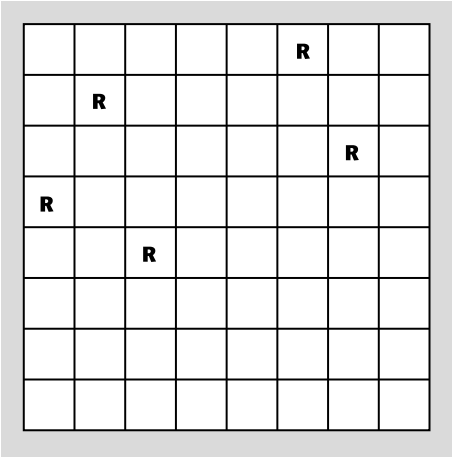
Esta llamada da la solución siguiente:

```
['ie', 'ic', 'ie', 'id', 'ic', 'ie']
```

Podéis observar que no puede haber ninguna otra solución con un camino de longitud menor. Si existiera, se habría encontrado antes, dado que en los pasos previos la longitud máxima estaba limitada a 5, 4, 3, 2, 1 y 0. Por tanto, cualquier solución de estas longitudes se habría encontrado.

3.3. Ejemplo práctico de búsqueda de solución

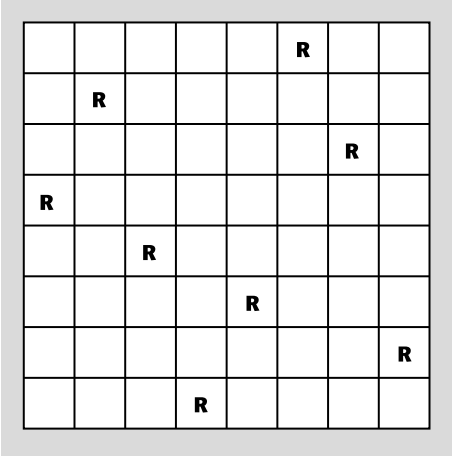
Retomando el problema de las ocho reinas del apartado «Introducción a la resolución de problemas y búsqueda», vamos a ver la evolución del algoritmo de búsqueda en profundidad para la búsqueda de una solución a partir de este estado inicial:



Para simplificar, agruparemos las variables x_1, x_2, x_3, x_4, x_5, x_6, x_7 y x_8 en un *array* o lista $x = [x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]$. Entonces, la evolución del algoritmo de búsqueda en profundidad sería la siguiente:

```
Paso 0:

Estado inicial: x=[5,1,6,0,2,NULL,NULL,NULL]

Nodos a expandir (solamente el estado inicial):

[ [5,1,6,0,2,NULL,NULL,NULL] ]

Paso 1:

Seleccionamos primer nodo de la lista a expandir.

Comprobamos si este nodo es la solución. No lo es. Lo expandimos.

Nodos a expandir:

[        [5,1,6,0,2,4,NULL,NULL],        [5,1,6,0,2,7,NULL,NULL],
[5,1,6,0,2,NULL,7,NULL], [5,1,6,0,2,NULL,NULL,3] ]

Paso 2:

Seleccionamos primer nodo de la lista a expandir.

Comprobamos si este nodo es la solución. No lo es. Lo expandimos.

Añadiremos los nuevos nodos por delante de los anteriores.

Nodos a expandir:

[           [5,1,6,0,2,4,7,NULL],           [5,1,6,0,2,4,NULL,3],
[5,1,6,0,2,7,NULL,NULL],                [5,1,6,0,2,NULL,7,NULL],
[5,1,6,0,2,NULL,NULL,3] ]

Paso 3:

Seleccionamos primer nodo de la lista a expandir.

Comprobamos si este nodo es la solución. No lo es. Lo expandimos.

Añadiremos los nuevos nodos por delante de los anteriores.

Nodos a expandir:

[            [5,1,6,0,2,4,7,3],            [5,1,6,0,2,4,NULL,3],
[5,1,6,0,2,7,NULL,NULL],                [5,1,6,0,2,NULL,7,NULL],
[5,1,6,0,2,NULL,NULL,3] ]

Paso 4:

Seleccionamos primer nodo de la lista a expandir.

Comprobamos si este nodo es la solución.

Es la solución. Problema resuelto.

Solución: [5,1,6,0,2,4,7,3]
```

Es importante destacar que, como que se puede comprobar, a la hora de expandir solo consideramos los nodos resultantes válidos (satisfacen las restricciones del problema). Así, pues, el algoritmo ha encontrado la solución siguiente:


R
R
R
R
R
R
R
R

4. Coste y función heurística

En los esquemas de búsqueda vistos hasta ahora, la información utilizada a la hora de decidir qué nodo es el que hay que expandir se limita a saber qué operadores podemos aplicar a un determinado nodo. Ahora presentamos unos nuevos esquemas que utilizan información adicional. En primer lugar, consideramos el uso del coste asociado a un operador y, a continuación, el uso de las llamadas *funciones heurísticas*.

El coste de un operador, ya comentado en el subapartado «Algunas consideraciones adicionales: la importancia de una representación adecuada y la cuestión del coste», corresponde a una evaluación de los operadores en términos de precio, dificultad, tiempo, etc. Una solución, o dicho de manera más precisa, el camino a una solución se puede evaluar basándonos en los costes de los generadores que aplicamos. Es decir, el coste de un camino se considerará la suma de los costes de los operadores que componen dicho camino. El coste de las soluciones nos permite discriminarlos y, normalmente, nos interesará la solución de coste mínimo (o una de las soluciones, si hay más de una con el mismo coste).

Funciones de coste de tres ejemplos

Consideramos, a continuación, dos de los ejemplos ya conocidos y uno de búsqueda en bases de datos de pago. A cada uno de estos ejemplos le asignamos funciones de coste.

1) Rompecabezas lineal

Podemos considerar que todos los operadores del rompecabezas tienen el mismo coste (coste unitario). Así, el coste de un camino corresponderá a la longitud del camino y una solución con menos movimientos será considerada más buena que una que tenga más.

2) Camino más corto

Como que el objetivo es encontrar la ruta de distancia mínima, el coste de un operador lo definimos como la longitud de la carretera correspondiente (en este problema, elegir un operador es escoger una carretera). Evidentemente, el coste de un camino será su longitud (suma de los segmentos de carretera seleccionados). Un problema equivalente al del camino más corto será considerar el camino de precio más bajo. En este caso, deberemos hacer una evaluación de los precios en euros para cada carretera.

3) Búsqueda en bases de datos de pago

Cuando para encontrar una determinada información en una base de datos, tenemos que acceder más de una vez haciendo varias consultas y tenemos que pagar una cierta cantidad por cada consulta, tendremos que cada consulta corresponde a un operador y que el coste de los operadores es el precio de hacer las consultas. De manera análoga a los casos anteriores, el coste de la solución corresponderá a la suma de los costes de los accesos.

Ved también

Ved los ejemplos del rompecabezas lineal y del camino más corto en el subapartado «Espacio de estados y representación de un problema» de este módulo.

Ahora veremos algunos métodos que, con objeto de encontrar la solución, utilizan el coste o una información similar.

4.1. Búsqueda de coste uniforme

Para ordenar los nodos de la lista de nodos a expandir, el algoritmo de coste uniforme utiliza el coste de su camino desde el estado inicial. Cuando se extrae el primer nodo de la lista, tenemos que este es siempre el nodo terminal del camino de menor coste.

Dado un nodo n del árbol de expansión, denotaremos con $g(n)$ el coste del camino del estado inicial al nodo n.

Figura 13

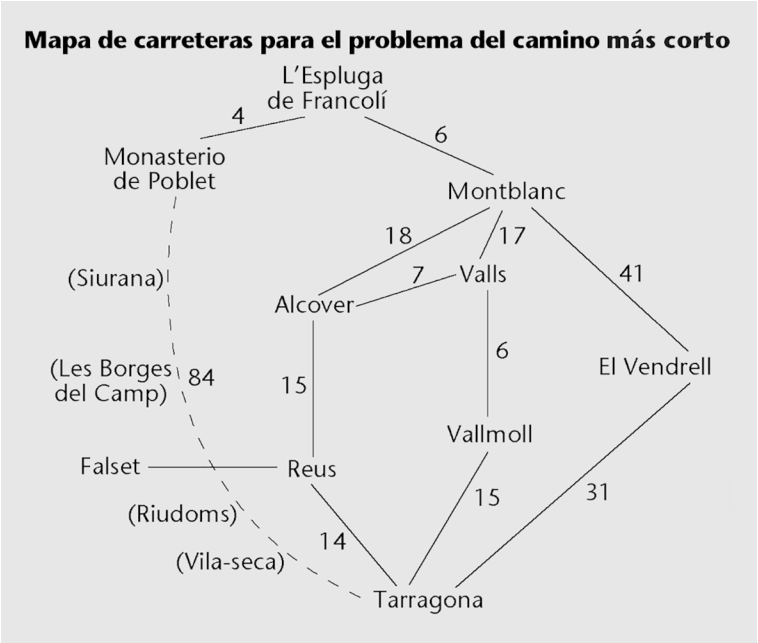
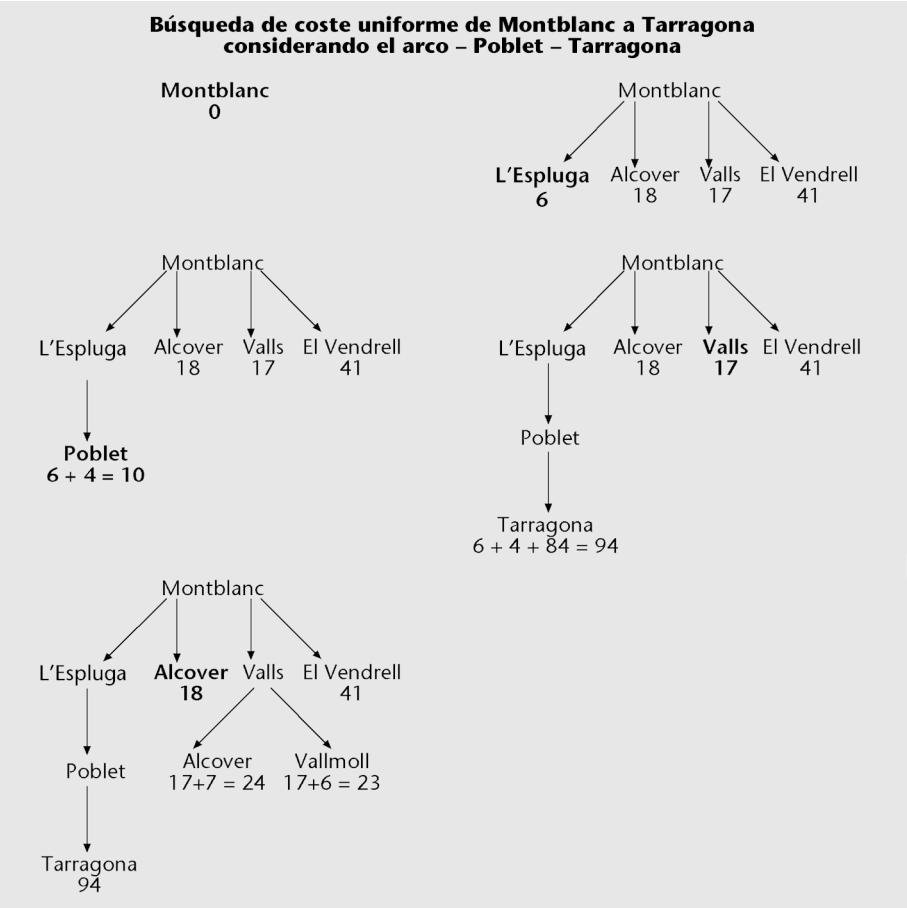


Figura 14



Búsqueda del coste uniforme para el problema del camino más corto de Montblanc a Tarragona cuando usamos el mapa de carreteras considerando el arco de Poblet a Tarragona.

Ejemplo del camino más corto

Consideramos, a continuación, el camino más corto de Montblanc a Tarragona cuando los operadores del problema son aquellos que se pueden deducir del mapa de carreteras para el problema del camino más corto. Este camino se desarrolla parcialmente en la figura que representa la búsqueda del coste uniforme y la lista de nodos a expandir aparece con detalle en la tabla siguiente. Para simplificar el ejemplo, hemos supuesto que en una población no podemos deshacer el último camino (no se puede visitar el padre de un nodo). Así, si el último arco considerado es el de Montblanc a Valls, no podemos pasar de Valls a Montblanc.

Como se ve en la figura y en la tabla, la búsqueda del coste uniforme empieza expandiendo Montblanc y obteniendo 4 poblaciones. Cada población es evaluada con la función de coste (en este caso, la distancia a Montblanc) y se inserta en la lista de nodos a expandir. Después, se seleccionará la localidad con un valor menor (l'Espluga) y se expandirá. Dado que no permitimos regresar al nodo padre, no podemos rehacer el último camino y así solo obtenemos un nuevo estado: Poblet. El coste de este estado será el de l'Espluga más el coste del operador de l'Espluga a Poblet (la distancia del arco). Así, tenemos que Poblet está evaluado en 6 + 4 = 10. Seguidamente, expandiremos el nodo con un coste menor, que es Poblet.

La figura 14 muestra los primeros pasos de la expansión. Se puede ver que, a pesar de que el algoritmo de expandir Poblet genera el estado Tarragona, este no se da como solución. Esto es así porque el esquema general de búsqueda presentado en el subapartado «La implementación» solo acaba cuando el nodo seleccionado a expandir es la solución y este no es el caso aquí. Ahora tenemos que el nodo seleccionado es Poblet y Tarragona quedará en la cola de nodos a expandir.

La tabla siguiente ofrece las fronteras que vamos obteniendo en los pasos sucesivos hasta encontrar la solución. Tenemos los estados que aparecen en la frontera y, para cada uno de ellos, entre paréntesis, su función de coste.

Ved también

Ved el subapartado «Algunas consideraciones adicionales».

Observación

Recordad que en el subapartado «La implementación» se había dicho que, para acabar, hay que esperar que el nodo seleccionado sea la solución y que además esta sea la solución óptima. Este es el caso de la búsqueda de coste uniforme como vemos aquí.

Lista de nodos

Lista de nodos pendientes de expandir para el problema del camino más corto de Montblanc a Tarragona cuando usamos el mapa de carreteras y la búsqueda de coste uniforme, considerando la carretera Poblet-Tarragona.

Aparecen subrayados los nodos seleccionados que se expanden. El orden de los nodos de la lista no es el que se daría en la cola con prioridad, sino que siguen la frontera de la figura. La cola con prioridad los ordenaría según el coste que aparece entre paréntesis.

1. Montblanc (0)

2. L'Espluga (6), Alcover (18), Valls (17), El Vendrell (41)

3. Poblet (10), Alcover (18), Valls (17), El Vendrell (41)

4. Tarragona (94), Alcover (18), Valls (17), El Vendrell (41)

5. Tarragona (94), Alcover (18), Alcover (24), Vallmoll (23), El Vendrell (41)

6. Tarragona (94), Valls (25), Reus (33), Alcover (24), Vallmoll (23), El Vendrell (41)

7. Tarragona (94), Valls (25), Reus (33), Alcover (24), Tarragona (38), El Vendrell (41)

8. Tarragona (94), Valls (25), Reus (33), Montblanc (42), Reus (39), Tarragona (38), El Vendrell (41)

9. Tarragona (94), Montblanc (42), Vallmoll (31), Reus (33), Montblanc (42), Reus (39), Tarragona (38), El Vendrell (41)

10. Tarragona (94), Montblanc (42), Tarragona (46), Reus (33), Montblanc (42), Reus (39), Tarragona (38), El Vendrell (41)

11. Tarragona (94), Montblanc (42), Tarragona (46), Falset (68), Tarragona (47), Montblanc (42), Reus (39), Tarragona (38), El Vendrell (41)

Si reseguimos el algoritmo hasta encontrar la solución, podemos observar que encuentra el camino óptimo. De hecho, la búsqueda de coste uniforme siempre encuentra la solución óptima (esto es, encuentra el camino de menor coste) cuando el coste de un camino no decrece nunca –cuando los costes de los operadores son siempre positivos. Dicho de otro modo, no decrece el coste cuando se añaden más nodos (es una función monótona).

La optimalidad del algoritmo se debe al hecho de que la búsqueda se para cuando se selecciona un nodo y este es la solución y no cuando encuentra un estado objetivo de estados resultado de una expansión. Cuando el algoritmo acaba, querrá decir que no hay ningún camino parcial con un coste menor que el del camino que hemos encontrado hacia la solución. Por tanto, si los costes son siempre positivos, cualquier otro camino siempre tendrá un mayor coste.

Solución óptima

En el ejemplo del camino más corto, hemos considerado que nunca podíamos deshacer el último trozo de camino. Es importante subrayar que si hubiéramos permitido regresar al estado padre, también habríamos encontrado la solución óptima.

La búsqueda del coste uniforme es completa y óptima cuando la función del coste es siempre positiva.

Bucles de coste positivo

Observad que el hecho de tener bucles no impide encontrar la solución (siempre que sean bucles de coste positivo) porque cuando iteramos siempre acabarán sobrepasando el coste de cualquier otro camino. Esto provocará que también se desarrolle el camino óptimo.

La búsqueda del coste uniforme se puede ver como una generalización de la búsqueda en anchura: cuando los costes de todos los operadores son iguales, tendremos una búsqueda en anchura. De hecho, la búsqueda que hemos presentado aquí tiene propiedades parecidas.

4.2. Búsqueda con función heurística: búsqueda ávida o voraz

La búsqueda del coste uniforme es ineficiente en algunos casos porque la expansión no tiene en cuenta si nos estamos acercando o no a un estado solución. En el ejemplo considerado antes, de Montblanc pasamos a l'Espluga y este estado se expande antes de expandir cualquier otro. En particular, se expande antes que Valls, cuando esta población está más cercana a Tarragona. En general, si tenemos una situación como la que aparece en la figura, que representa un caso muy ineficiente, tendremos que la búsqueda del coste uniforme expandirá muchos nodos antes de llegar al destino. Solo llegará al estado solución cuando todos los caminos parciales superen el coste del arco directo.

Figura 15

El problema que aquí se plantea aparece porque la búsqueda del coste uniforme a la hora de decidir solo tiene en cuenta qué nodo expande su coste desde el estado inicial. No se tiene en cuenta que del nuevo nodo se tiene que proseguir el camino hasta la solución. Así, en el caso de decidir entre expandir l'Espluga y Valls, si consideramos que Valls está más cerca de Tarragona que l'Espluga, lo podríamos expandir primero. Pero dado que, en general, no podemos saber hasta qué punto un estado está cerca de la solución, usaremos una función que nos hace una estimación de la proximidad del nodo a la solución. Una función de estas características se llama *función heurística*.

Una **función heurística** es una función que, aplicada a un nodo, estima el coste del mejor camino entre este nodo y un estado solución. Usaremos $h(n)$ para representar las funciones heurísticas.

Obviamente, las funciones heurísticas dependerán del problema (de las características de los operadores y de los estados que satisfacen la función objetivo). Habitualmente, tendremos que las funciones son positivas y definidas de forma que cuando son aplicadas a un estado solución retornen el valor cero ($h(n) = 0$ si n corresponde a un estado objetivo).

Ejemplos de funciones heurísticas

1) Rompecabezas lineal

El número de piezas mal puestas es una función heurística. Cuantas más piezas haya fuera de lugar, más movimientos habrá que hacer para colocarlas.

2) Camino más corto

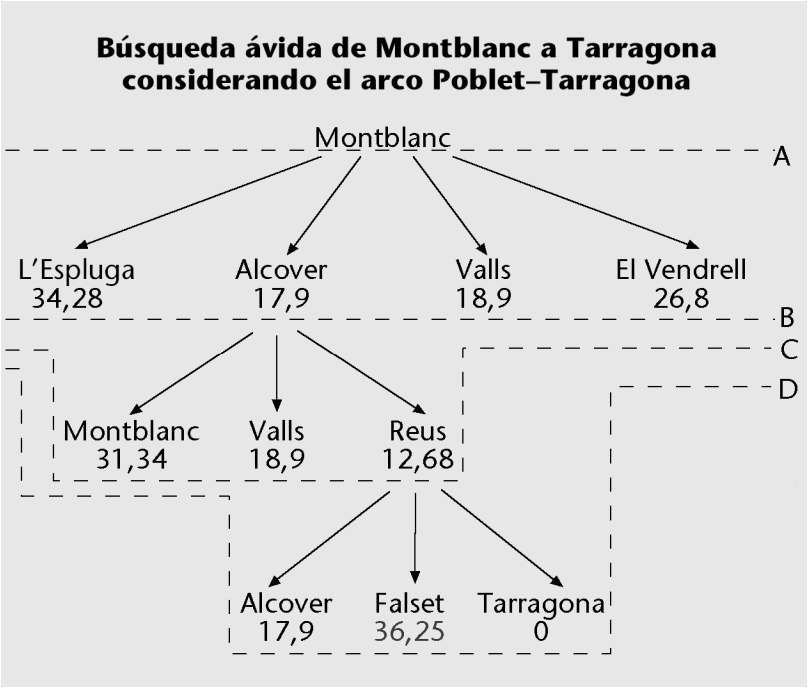
La distancia en línea recta entre una población y la localidad donde queremos ir se puede usar como función heurística. La distancia que tenemos que recorrer acostumbra a ser más grande cuanto mayor es la distancia en línea recta.

El uso de la función heurística como criterio en la ordenación de los nodos en la lista de los nodos a expandir corresponde a la busca ávida (o del primero el mejor). El hecho de elegir el nodo más cercano en pasos sucesivos provoca generalmente que una vez elegido un camino, este se siga hasta la solución. Por eso, la búsqueda ávida se asemeja a la búsqueda en profundidad. Además, tiene los mismos problemas que la búsqueda en profundidad, en el sentido de que no siempre encuentra la solución y, cuando la encuentra, puede que no sea la solución óptima.

Tabla 1

Función heurística		
Población	**Destino = Tarragona**	**Destino = Falset**
Alcover	17,9	32,5
L'Espluga	34,28	37,5
Falset	36,25	0
Montblanc	31,34	37,5
Poblet	34,2	35,5
Reus	12,68	24,25
Tarragona	0	36,25
Vallmoll	14,47	36,25
Valls	18,9	37,5
El Vendrell	26,8	60

Figura 16

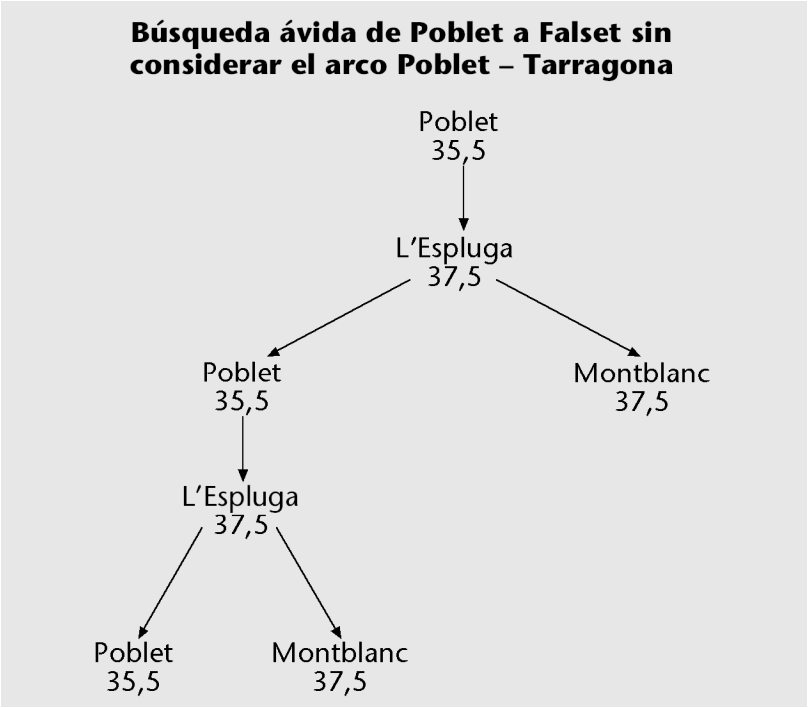


Ejemplos de aplicación de la búsqueda ávida o voraz

Siguiendo con el ejemplo, en la figura 16 se presenta la evolución del árbol de búsqueda cuando partimos de Montblanc y queremos llegar a Tarragona. Para aplicar el algoritmo, se usan los valores de la función heurística que aparecen en la tabla de la función heurística. Estos valores son los que se usan para ordenar los nodos que hay en la frontera y seleccionar en cada iteración el nodo a expandir. Se puede ver que la solución encontrada no es la óptima. El coste del camino es mayor que en el caso de la búsqueda con coste uniforme.

En la figura 17, se muestra la evolución del árbol de búsqueda cuando partimos de Poblet y queremos ir a Falset. En este ejemplo no consideramos el arco Poblet-Tarragona. El ejemplo muestra que, en este caso, el algoritmo de búsqueda entra en un bucle sin fin porque l'Espluga y Poblet son siempre los estados con un valor de la función heurística menor.

Figura 17



Con estos dos ejemplos se ha visto que la búsqueda ávida no es ni completa ni óptima. En cuanto a los costes, tenemos que si m es la profundidad máxima, donde está la solución, los costes tanto de tiempo como de espacio para el caso peor son $O(b^m)$. De todas maneras, el coste real depende de la función heurística elegida y esto puede hacer que tanto el coste en espacio como en tiempo disminuyan sustancialmente. Por ejemplo, en el caso del problema de Montblanc a Tarragona, hemos visto que el número de nodos que hemos tenido que expandir solo son tres.

4.3. Búsqueda con función heurística: algoritmo A*

Como ya hemos visto, la búsqueda ávida elige aquel nodo que estima el coste mínimo de un nodo hacia un estado objetivo y, por tanto, acorta lo que resta de búsqueda. Sin embargo, esta elección hace que el método no sea completo ni tampoco óptimo. Por otro lado, la búsqueda del coste uniforme minimiza el coste del camino del nodo origen hasta el nodo en curso. De este modo se tiene siempre el mejor coste, con independencia de lo que falte para llegar a la solución. En este sentido, los dos algoritmos de búsqueda son complementarios. Definiendo una función f para cada nodo n como:

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

donde $g(n)$ es el coste desde el estado inicial al nodo n y $h(n)$ es la estimación desde este nodo hasta la solución, tenemos que $f(n)$ es una estimación del coste del camino que va del estado inicial a un estado de solución pasando por el nodo actual n. La selección del nodo a expandir, de acuerdo con esta función, es el llamado *algoritmo A**.

La figura 18 representa gráficamente el significado de esta función para los nodos de la frontera. Así, una estimación del coste del estado inicial a un estado objetivo pasando por ($f(n)$) se puede definir como el coste desde el estado inicial a n (esta parte es $g(n)$) más la estimación del coste desde n hasta el estado objetivo ($h(n)$).

Representación gráfica de $f(n)$

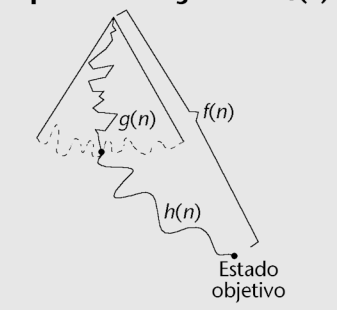


Figura 18

La optimalidad y la completitud del algoritmo A* dependen de la función heurística. Será un algoritmo óptimo y completo cuando la función heurística no sobrestime nunca el coste de llegar al objetivo. Si la función h satisface esta propiedad, diremos que es una función heurística admisible.

Una función heurística es admisible si nunca sobrestima el coste. El algoritmo A* con una función heurística admisible es completo y óptimo.

Podemos afirmar que una función heurística admisible es una función optimista, puesto que siempre nos estima el coste a la baja. Esto es, hace creer que el coste es menor de lo que realmente es.

Ejemplos de funciones heurísticas admisibles y no admisibles

1) Camino más corto

La distancia en línea recta es una función heurística admisible, puesto que la distancia real siempre será mayor que la distancia en línea recta.

2) El rompecabezas lineal

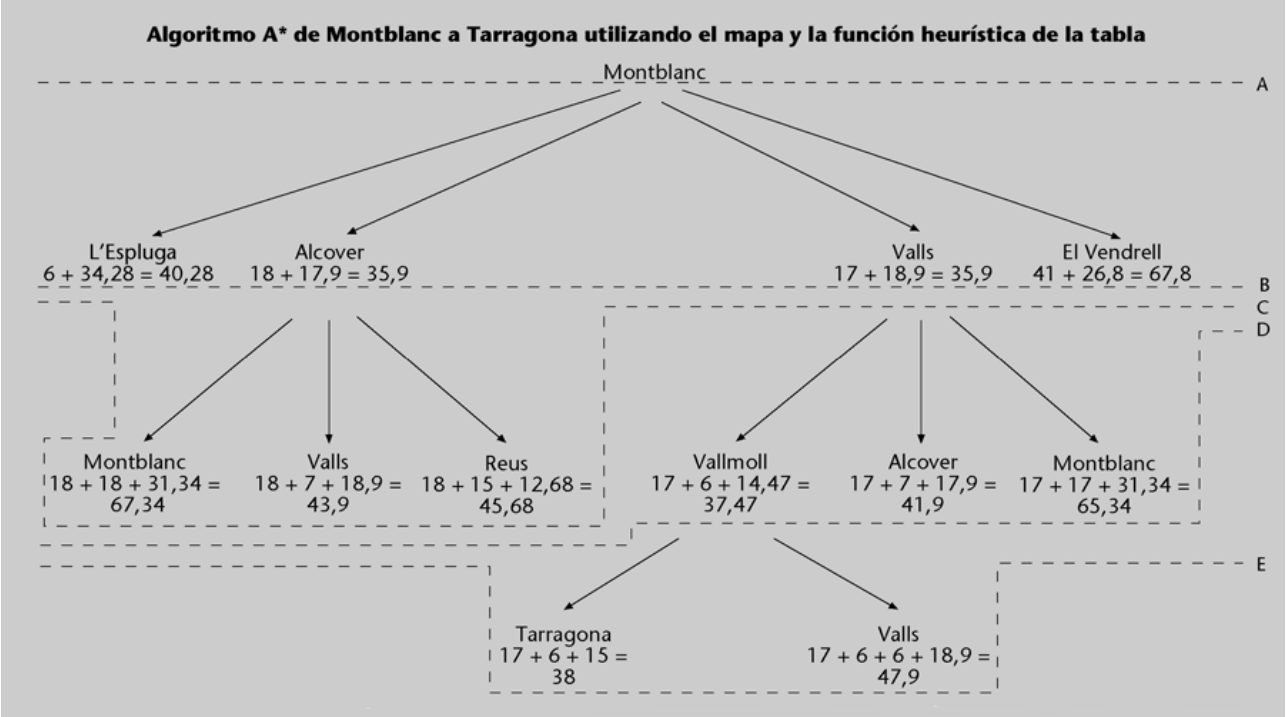
El número de piezas mal puestas es una función heurística, pero no es admisible porque podemos tener dos piezas mal puestas y que se pueda llegar a la solución con un solo movimiento (una única permutación). En cambio, el número de piezas mal puestas dividido entre dos sí que es una función admisible. Observad que para resolver tableros con dos piezas mal puestas, necesitamos como mínimo un movimiento; para resolver tableros con tres piezas, necesitamos como mínimo dos movimientos; y para los de cuatro piezas mal puestas, también necesitamos dos movimientos como mínimo. En la tabla encontramos los valores que devolvería la función heurística en tres casos diferentes y el número de movimientos necesarios.

Tabla 2

Valor de la función heurística y número de movimientos mínimo para tres ejemplos de tablero de rompecabezas lineal			
Tablero	**Número de piezas mal puestas**	**Heurística**	**Número de movimientos mínimo**
[b, a, c, d]	2	1	1
[b, c, a, d]	3	1,5	2

Valor de la función heurística y número de movimientos mínimo para tres ejemplos de tablero de rompecabezas lineal			
[b, a, d, c]	4	2	2

Figura 19



En la figura 19, se desarrolla el algoritmo A* para el caso del camino más corto con origen en Montblanc y destino en Tarragona. Dado que cada vez empezaremos con el árbol de búsqueda con un único nodo correspondiente a Montblanc, cuando expandimos este nodo obtenemos las cuatro poblaciones conectadas según el mapa. Cada una de estas poblaciones es evaluada con la función de evaluación, que consiste en sumar el coste del arco de Montblanc a la población y la heurística de la población a Tarragona. De este modo, se obtienen los valores 6 + 34,28 = 40,28 para l'Espluga, 18 + 17,9 = 35,9 para Alcover, 17 + 18,9 = 35,9 para Valls y 41 + 26,8 = 67,8 para El Vendrell. A continuación, cuando seleccionamos el mejor nodo, obtenemos Alcover (también se hubiera podido escoger Valls, puesto que tiene el mismo valor) y, por tanto, será este nodo el que expandiremos. La expansión genera las poblaciones de Montblanc, Valls y Reus. La evaluación para cada una de ellas corresponderá al coste de Montblanc a Alcover, más el coste de Alcover a la población y añadiendo la heurística de la población a Tarragona. Continuamos expandiendo el nodo Valls, que tiene asociado el valor mínimo (35,9). La expansión nos añade al árbol los nodos de Vallmoll, Alcover y Montblanc. A continuación se selecciona el nodo con una evaluación menor (Vallmoll) y se expande. Cuando lo expandimos, se obtienen dos estados (Tarragona y Valls), que

serán evaluados e insertados en la lista de nodos a expandir. Seguidamente, se seleccionará el más prometedor, que ya es un estado solución (Tarragona), y, por tanto, el algoritmo acaba.

4.3.1. Algunas cuestiones de la función heurística

Se ha dicho que una función de evaluación admisible implica que el algoritmo sea óptimo y completo. Además de influir en estos aspectos, las funciones heurísticas también lo hacen, evidentemente, en los costes de tiempo y de espacio.

Con objeto de evaluar las diferentes heurísticas de un problema, podemos usar el llamado *factor de ramificación efectivo* que denotamos con b^*.

El **factor de ramificación efectivo** se define de la manera siguiente: si la cantidad de nodos expandidos por el algoritmo en un problema es N y la profundidad donde está la solución es d, entonces el factor de ramificación efectivo b^* es el valor para que el correspondiente árbol uniforme (equilibrado) de profundidad d tenga exactamente N nodos.

Esto es, b^* es el valor que cumple:

$$N = 1 + b^* + (b^*)_2 + \ldots + (b^*)^d$$

Dado que los resultados experimentales nos dicen que para una heurística el factor b^* es bastante similar en todos los problemas, para evaluar una heurística concreta podemos considerar unos cuantos problemas pequeños y estimar b^*. Una heurística funcionará bien cuando b^* se acerca a 1.

Otro concepto importante es el de la *dominancia*. Diremos que una función heurística H domina a otra h cuando $H(n) \geq h(n)$ para todos los nodos n. En estas condiciones, se puede demostrar que A* con la función H es más eficiente que A* con h. Esto es así porque todo nodo expandido con H se expandirá también con h, pero no todo nodo expandido con h lo hará también con H. Por tanto, puede ser que h obligue a expandir más nodos.

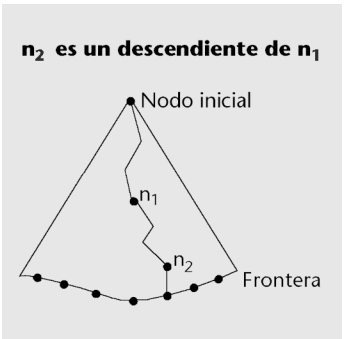


Figura 20

La dominancia y su influencia en la eficiencia del algoritmo hace que sean interesantes los dos aspectos siguientes:

A continuación demostramos que todo nodo que se expanda con la heurística H también lo hará con h: suponemos que la solución tiene un coste k. Entonces, para llegar a esta solución, tenemos que expandir todos los nodos con coste + heurística menor que k. Si tenemos un nodo n que se expande con

la heurística H, quiere decir que el coste (origen, n) + $H(n) < k$. En este caso, si $h(n) < H(n)$, entonces también tenemos el coste (origen, n) + $h(n) < k$. Por tanto, con la heurística h también se expandirá.

1) Cuando en un árbol de búsqueda el valor de f nunca decrece, diremos que la heurística es monótona. Esto es, que para todo par de nodos n_1 y n_2 donde n_2 es un descendiente de n_1 (ved la figura 20, donde n_2 es un descendiente de n_1), se cumple $f(n_1) \leq f(n_2)$.

Cuando esta condición no se satisface, podemos definir una nueva función f de la manera siguiente: $f'(n_2) = \max(f(n_1), g(n_2) + h(n_2))$.

Esta definición equivale a incrementar el valor de la función heurística para el nodo n_2, y, por tanto, corresponde a definir una nueva heurística que domina a la primera. Además, esta transformación construye una función heurística admisible cuando la heurística inicial ya lo es. Esto es así porque sabemos que todo camino que pase por n_1 ha de tener un coste mayor que $f(n_1)$, por la condición de admisibilidad de h. Por tanto, el coste real del camino que pasa por n_2 y n_1 también tiene que ser mayor que $f(n_1)$, con independencia de que $f(n_1) \geq f(n_2)$ y del valor $f(n_2)$. Por eso podemos incrementar el valor de la función aplicada al último nodo hasta $f(n_1)$.

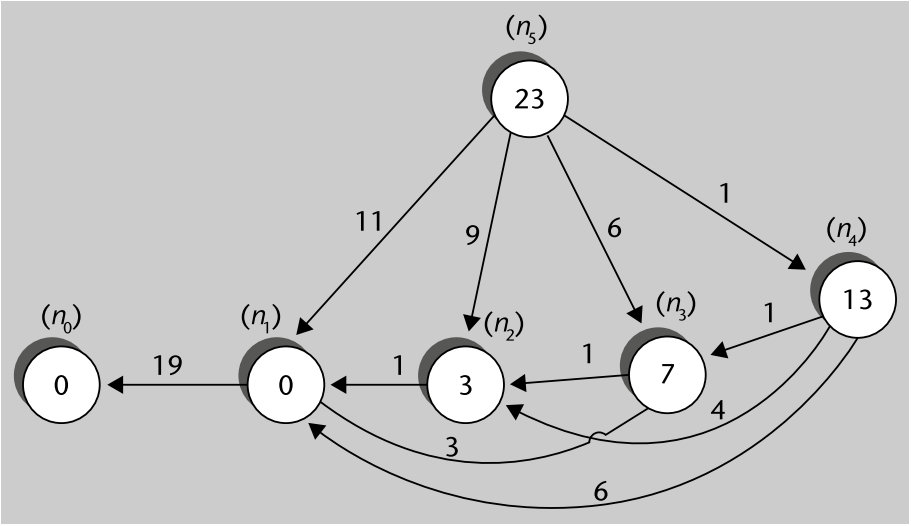
2) Si tenemos h_1, h_2, ..., h_m funciones heurísticas admisibles, entonces podemos definir una nueva heurística h de la manera siguiente: $h(n) = \max(h_1(n), h_2(n), ..., h_m(n))$. Esta función h domina a las otras y el algoritmo será, por tanto, más eficiente.

4.3.2. Consistencia del heurístico

Un heurístico de un grafo es *consistente* si es un heurístico admisible con la propiedad siguiente: para todo par de nodos de un grafo x e y, si hay un camino del nodo x al nodo y, entonces $h(x) \leq \text{coste}(x, y) + h(y)$, donde coste($x$, y) corresponde a ir de x a y. Judea Pearl demostró que podemos restringir y a ser un vecino de x para dar una definición más intuitiva, aunque equivalente: el hecho de ir de un nodo a otro vecino no tiene que hacer decrecer h más de lo que crece g. Diremos que un heurístico es *inconsistente* cuando no es consistente. Fijémonos en que la admisibilidad forma parte de la definición de consistencia, por lo que el hecho de que un heurístico sea consistente implica, por definición, que es admisible y que la A* encontrará la solución óptima si usamos este heurístico. También observamos que si consistente $\Rightarrow$ admisible, entonces *no admisible* $\Rightarrow$ *inconsistente*.

Alberto Martelli investigó este problema y definió una familia de grafos G_i, donde $\{i \geq 3\}$, denominados *grafos de Martelli*, con determinadas propiedades especiales. Dado el grafo de Martelli G_5:

Figura 21



donde el número de aristas corresponde al coste de ir de un nodo al otro y el heurístico es el número dentro de cada nodo. Así, $h(n_5) = 23$, $h(n_4) = 13$, etc. El objetivo es encontrar el camino de menor coste para ir de n_5 a n_0.

Demostrar que h es admisible es sencillo, puesto que el coste óptimo de llegar de cualquier nodo a n_0 no puede ser inferior a 19, es decir, $h^*(n_j) \geq 19$, para todo j entre 1 y 5. Esto es debido a la arista que va de n_1 a n_0. Y todos los heurísticos son menores que 19: $h(n_j) < 19 \leq h^*(n_j)$, para todos los nodos excepto n_5. El camino óptimo para n_5 tiene un coste 23, como veremos después cuando apliquemos el A*, algo que resulta bastante evidente. Dado que $23 = h(n_5) \leq h^*(n_5) = 23$, h es admisible. La inconsistencia de h se ve, puesto que $13 = h(n_4) > \text{coste}(n_4, n_3) + h(n_3) = 1 + 7$. Hay más inconsistencias.

Fijémonos en que en el algoritmo A* utilizamos dos listas de nodos abiertos y cerrados, revisando los costes en caso de mejorarlos. En cambio, si miráis el pseudocódigo de la Wikipedia, encontraréis un algoritmo donde, cuando se expande un nodo, si alguno de sus vecinos corresponde al conjunto de cerrados, ignoramos el vecino en cuestión, sin revisar para nada sus costes.

Consideraremos dos algoritmos A* diferentes:

- **A* completo:** Cuando consideramos los sucesores del nodo actual, si encontramos uno que ya está en el conjunto de cerrados, calculamos su coste y si mejora el coste que tenía cuando lo cerramos, sacamos el sucesor del conjunto de cerrados y lo volvemos a situar en el conjunto de abiertos con el coste revisado.

- **A* simplificado:** Cuando consideramos los sucesores del nodo actual, si encontramos uno que ya está en el conjunto de cerrados, lo ignoramos.

Este es el caso del pseudocódigo mencionado que podéis encontrar en la Wikipedia.

El A* simplificado solo funciona bien si el heurístico es consistente. Si aplicamos el A* simplificado a G_5, veremos que el camino que encuentra es el camino de n_5 a n_1, y de n_1 a n_0, con coste 30, que no es el óptimo. Si aplicamos el A* completo, encontramos el camino óptimo, de coste 23 (ejercicio para el lector: aplicad los algoritmos manualmente).

Detalle del A* simplificado y completo aplicados a G_5

Si consideramos el A* simplificado en el grafo anterior G_5, entonces las listas de nodos abiertos y cerrados que obtenemos en cada iteración son los siguientes (el nodo subrayado es el nodo de la lista de nodos abiertos que se expande en la iteración siguiente por el hecho de ser el de coste mínimo):

Abiertos(0): (n_5, 0 + 23) → ($\underline{n_5}$, 23)
Cerrados(0):

Abiertos(1): (n_1, 11 + 0), (n_2, 9 + 3), (n_3, 6 + 7), (n_4, 1 + 13) → ($\underline{n_1}$, 11), (n_2, 12), (n_3, 13), (n_4, 14)
Cerrados(1): (n_5, 23)

Abiertos(2): (n_0, 11 + 19 + 0), (n_2, 12), (n_3, 13), (n_4, 14) → (n_0, 30), ($\underline{n_2}$, 12), (n_3, 13), (n_4, 14)
Cerrados(2): (n_1, 11), (n_5, 23)

Abiertos(3): (n_0, 30), ($\underline{n_3}$, 13), (n_4, 14)
Cerrados(3): (n_1, 11), (n_2, 12), (n_5, 23)

Abiertos(4): (n_0, 30), ($\underline{n_4}$, 14)
Cerrados(4): (n_1, 11), (n_2, 12), (n_3, 13), (n_5, 23)

Abiertos(5): ($\underline{n_0}$, 30)
Cerrados(5): (n_1, 11), (n_2, 12), (n_3, 13), (n_4, 14), (n_5, 23)

Por tanto, vemos que como no se recalculan los costes para los nodos ya visitados, el A* simplificado decide que el camino para ir del nodo 5 al nodo 0 es el siguiente:

$$n_5 \rightarrow n_1 \rightarrow n_0$$

El coste de este camino es 30 (11 + 19) y no es el camino mínimo existente entre los nodos n_5 y n_0. En cambio, utilizando el A* completo, sin restricciones, obtenemos un despliegue de la búsqueda más largo, pero correcto:

Abiertos(0): (n_5, 0 + 23) → ($\underline{n_5}$, 23)
Cerrados(0):

Abiertos(1): (n_1, 11 + 0), (n_2, 9 + 3), (n_3, 6 + 7), (n_4, 1 + 13) → ($\underline{n_1}$, 11), (n_2, 12), (n_3, 13), (n_4, 14)
Cerrados(1): (n_5, 23)

Abiertos(2): (n_0, 11 + 19 + 0), (n_2, 12), (n_3, 13), (n_4, 14) → (n_0, 30), ($\underline{n_2}$, 12), (n_3, 13), (n_4, 14)
Cerrados(2): (n_1, 11), (n_5, 23)

Abiertos(3): (n_0, 30), (n_1, 9 + 1 + 0), (n_3, 13), (n_4, 14) → (n_0, 30), ($\underline{n_1}$, 10), (n_3, 13), (n_4, 14)
Cerrados(3): (n_2, 12), (n_5, 23)

Abiertos(4): (n_0, 9 + 1 + 19), (n_0, 30), (n_3, 13), (n_4, 14) → (n_0, 29), ($\underline{n_3}$, 13), (n_4, 14)
Cerrados(4): (n_1, 10), (n_2, 12), (n_5, 23)

Abiertos(5): (n_0, 29), (n_1, 6 + 3 + 0), (n_2, 6 + 1 + 3), (n_4,14) → (n_0, 29), ($\underline{n_1}$, 9), (n_2, 10), (n_4, 14)
Cerrados(5): (n_3, 13), (n_5, 23)

Abiertos(6): (n_0, 29), (n_0, 6 + 3 + 19 + 0), (n_2, 10), (n_4, 14) → (n_0, 28), ($\underline{n_2}$, 10), (n_4, 14)

Cerrados(6): (n_1, 9), (n_3, 13), (n_5, 23)

Abiertos(7): (n_0, 28), (n_1, 6 + 1 + 1 + 0), (n_4, 14) → (n_0, 28), (n_1, 8), (n_4, 14)
Cerrados(7): (n_2, 10), (n_3, 13), (n_5, 23)

Abiertos(8): (n_0, 28), (n_0, 6 + 1 + 1 + 19 + 0), (n_4, 14) → (n_0, 27), (n_4, 14)
Cerrados(8): (n_1, 8), (n_2, 10), (n_3, 13), (n_5, 23)

Abiertos(9): (n_0, 27), (n_1, 1 + 6 + 0), (n_2, 1 + 4 + 3), (n_3, 1 + 1 + 7) → (n_0, 27), (n_1, 7), (n_2, 8), (n_3, 9)
Cerrados(9): (n_4, 14), (n_5, 23)

Abiertos(10): (n_0, 27), (n_0, 1 + 6 + 19 + 0), (n_2, 8), (n_3, 9) → (n_0, 26), (n_2, 8), (n_3, 9)
Cerrados(10): (n_1, 7), (n_4, 14), (n_5, 23)

Abiertos(11): (n_0, 26), (n_1, 1 + 4 + 1 + 0), (n_3, 9) → (n_0, 26), (n_1, 6), (n_3, 9)
Cerrados(11): (n_2, 8), (n_4, 14), (n_5, 23)

Abiertos(12): (n_0, 26), (n_0, 1 + 4 + 1 + 19 + 0), (n_3, 9) → (n_0, 25), (n_3, 9)
Cerrados(12): (n_1, 6), (n_2, 8), (n_4, 14), (n_5, 23)

Abiertos(13): (n_0, 25), (n_1, 1 + 1 + 3 + 0), (n_2, 1 + 1 + 1 + 3) → (n_0, 25), (n_1, 5), (n_2, 6)
Cerrados(13): (n_3, 9), (n_4, 14), (n_5, 23)

Abiertos(14): (n_0, 25), (n_0, 1 + 1 + 3 + 19 + 0), (n_2, 6) → (n_0, 24), (n_2, 6)
Cerrados(14): (n_1, 5), (n_3, 9), (n_4, 14), (n_5, 23)

Abiertos(15): (n_0, 24), (n_1, 1 + 1 + 1 + 1 + 0) → (n_0, 24), (n_1, 4)
Cerrados(15): (n_2, 6), (n_3, 9), (n_4, 14), (n_5, 23)

Abiertos(16): (n_0, 24), (n_0, 1 + 1 + 1 + 1 + 19 + 0) → (n_0, 23)
Cerrados(16): (n_1, 4), (n_2, 6), (n_3, 9), (n_4, 14), (n_5, 23)

Cuando se selecciona el (n_0, 23) el A* completo finaliza y encuentra que el camino entre los nodos n_5 y n_0 es el siguiente:

$$n_5 \rightarrow n_4 \rightarrow n_3 \rightarrow n_2 \rightarrow n_1 \rightarrow n_0$$

El coste de este camino es 23 (1 + 1 + 1 + 1 + 19) y se trata del camino óptimo entre los nodos n_0 y n_5. Observad un par de detalles en el paso a paso del algoritmo: los nodos revisitados con un coste menor se eliminan de la lista de nodos cerrados y de los nodos duplicados de la lista de nodos abiertos se mantiene únicamente el nodo de coste menor. Un ejemplo del primer caso lo podemos encontrar en la lista de nodos Abiertos(3), donde hay (n_1, 10) y, por lo tanto, se elimina (n_1, 11) de la lista de cerrados. Un ejemplo del segundo caso lo encontramos en la lista de nodos Abiertos(4), donde tenemos (n_0, 29) y (n_0, 30) y solo mantenemos (n_0, 29).

4.3.3. Implementación

A continuación, vamos a hacer una implementación con Python de la búsqueda A*. Empezamos definiendo el problema asociando información adicional a los nodos, como en el caso de la búsqueda en profundidad limitada. Ahora tendremos asociado a cada nodo el valor del coste del nodo y el de $f(n)$ (la función heurística más el coste del nodo). En cada expansión recalcularemos estos valores a partir del que teníamos en el nodo padre y de la heurística del nuevo estado. La función para calcular estos valores para cada nodo y la función para definirlo por el estado inicial se definen en el problema llamado `problema_busquedaAstar()`.

```
def problema_busquedaAstar():
```

```
    rl_ops = rl_operadores()
    def aux_func(info_nodo_padre, estado, operador):
        estado_padre = car(info_nodo_padre)
        g          = caadr(info_nodo_padre)
        g_mas_padre = g + 1
        g_mas_h    = g_mas_padre + heuristico(estado)
        return [g_mas_padre, g_mas_h]
    estado_inicial = [4,3,2,1]
    check_estado_final = lambda estado: estado == [1,2,3,4]
    return [rl_ops, aux_func, estado_inicial,
            check_estado_final,
            lambda estado: [0, heuristico(estado)
```

Para probar el algoritmo, necesitamos una función heurística. Construimos una muy dirigida hacia conseguir el mejor resultado.

```
def heuristica(estado):
    if estado == [4,3,2,1]: return 6
    elif estado == [4,3,1,2]: return 5
    elif estado == [4,1,3,2]: return 4
    elif estado == [1,4,3,2]: return 3
    elif estado == [1,3,4,2]: return 2
    elif estado == [1,3,2,4]: return 1
    elif estado == [1,2,3,4]: return 0
    else: return 10
```

Además de estas definiciones, necesitamos la función correspondiente a la estrategia:

```
def rl_estrategia_Astar (nodos_a_expandir, nuevos_nodos_a_expandir):
    union = nuevos_nodos_a_expandir + nodos_a_expandir
    return quicksort(selecciona_estimacion(union),union)
```

Esta función empieza uniendo los nodos que teníamos que expandir con los nuevos nodos y, después, los ordena mediante una función que implementa el método de ordenación `quicksort`. Esta función recibe dos listas como parámetros, una con los nodos que hay que ordenar y otra (de la misma longitud) con los valores que tienen que guiar la ordenación. Así, si hacemos:

```
quicksort([10, 4, 3],['n1', 'n2', 'n3'])
```

nos retornará:

```
['n3', 'n2', 'n1']
```

puesto que 'n_1' se asocia al valor 10, 'n_2' al valor 4 y 'n_3' al valor 3.

El `quicksort` se compone con la función `selecciona_estimacion` que construye una lista con los valores *f*(*n*) para cada nodo de la nueva lista de nodos a expandir (la lista `union_nodos`).

```
def selecciona_estimacion (union_nodos):
    return mapcar(lambda nodo: caddr(cdddr(nodo)), union_nodos)
```

Esta función selecciona el segundo elemento del campo de información adicional, puesto que es ahí donde se guarda *f*(*n*).

```
def quicksort (para_ordenar, elems):
    if not elems:
        return []
    pivot = car(para_ordenar)
    elemp = car(elems)
    pequeños = selecciona_menorigual(pivot, cdr(para_ordenar), cdr(elems))
    grandes = selecciona_mayor(pivot, cdr(para_ordenar), cdr(elems))
    return quicksort(selecciona_estimacion(pequeños),pequeños) +
            cons(elems, quicksort(selecciona_estimacion(grandes),grandes))
```

La función `quicksort` sigue el esquema habitual: selecciona el primer elemento de la lista que se debe ordenar (que hace de pívot); separa los elementos más pequeños que el pívot de aquellos que son más grandes; ordena cada uno de los dos conjuntos y, finalmente, devuelve una nueva lista que concatena los pequeños ordenados, el elemento pívot y los grandes ya ordenados.

Las funciones que seleccionan los nodos con una estimación más pequeña o más grande que el nodo pívot son estas:

```
def selecciona_menorigual (pivot, para_ordenar, elems):
    if not para_ordenar:
        return []
```

```
    ll = selecciona_menorigual(pivot,cdr(para_ordenar),cdr(elems))
    if car(para_ordenar) <= pivot:
        return cons(car(elems),ll)
    return ll

def selecciona_mayor (pivot, para_ordenar, elems):
    if not para_ordenar:
        return []
    ll = selecciona_mayor(pivot,cdr(para_ordenar),cdr(elems))
    if car(para_ordenar) > pivot:
        return cons(car(elems),ll)
    return ll
```

Con estas definiciones podemos definir la búsqueda de la manera siguiente:

```
def busqueda_Astar (problema):
    return hacer_busqueda(problema, rl_estrategia_Astar)
```

La aplicación de esta función al problema que hemos definido antes nos da el resultado siguiente:

```
busqueda_Astar(problema_busquedaAstar())
['id', 'ic', 'ie', 'ic', 'id', 'ic']
```

En este apartado, no hemos hecho las implementaciones de la búsqueda uniforme y ávida, pero seguirían el mismo esquema presentado aquí.

4.4. Otros métodos de búsqueda heurística

Un problema que comparten la mayoría de los algoritmos de búsqueda es que tienen que guardar en la memoria todos los nodos que se tienen que considerar para su expansión en los pasos sucesivos del algoritmo. Normalmente, la memoria necesaria para almacenar estos nodos crece de manera lineal en relación con el tiempo de ejecución y de manera exponencial en relación con la dimensión del problema. Esto significa que muchos de los algoritmos no se pueden aplicar a problemas de dimensión elevada. Últimamente, se han diseñado versiones de algoritmos de búsqueda clásicos que intentan resolver dicho problema. Una de las alternativas es aprovechar las ideas de la búsqueda iterativa con profundidad limitada (ved el subapartado «Busca en profundidad limitada»). La aplicación al algoritmo A* define los algoritmos llamados

[3]Acrónimo del inglés *iterative-deepening* A*.

*IDA**[3]: se poda una rama del árbol de búsqueda según un umbral que se va modificando en iteraciones sucesivas. La primera versión de estos algoritmos fue del año 1985 (de Korf), pero hay versiones posteriores.

Dado que los métodos como los IDA* infrautilizan la memoria disponible, se han desarrollado otros métodos que la aprovechan, pero evitando el *colapso* del sistema. Son los métodos denominados *reducción de nodos*[4]. Cuando la memoria está completamente llena, los hijos del nodo que tiene la peor evaluación de entre los que están en la memoria se borran y se modifica el valor del nodo padre. Se le asigna el valor del hijo, pero con un valor menor. De este modo el nodo tiene una estimación según la expansión que se ha realizado. Así, pues, estos algoritmos combinan fases de expansión y de reducción de nodos.

(4)Del inglés *node retraction*.

Además de estos métodos, hay otras alternativas para resolver los problemas de espacio, como, por ejemplo, la búsqueda en el perímetro (un tipo de búsqueda bidireccional) y la búsqueda tabú. Uno de los métodos de búsqueda en el perímetro empieza haciendo una búsqueda hacia atrás (por ejemplo, una búsqueda en anchura) a partir de un estado objetivo hasta que casi se ocupa toda la memoria. A partir de este momento, se aplica una búsqueda hacia adelante desde el estado inicial hasta que encuentra un estado de la frontera del árbol de la otra búsqueda.

Así se optimiza el uso de la memoria y, además, el coste de la búsqueda es menor que hacerla solo en una dirección. De todas formas, hacer una búsqueda hacia atrás no siempre es posible. A veces hay muchos estados objetivos y no podemos seleccionar solo uno y otras veces no es fácil aplicar los operadores hacia atrás. Este sería el caso del problema de la integración. En otros casos, como en los ejemplos del rompecabezas lineal o el problema del camino más corto, hacer una búsqueda atrás es más fácil. En estos casos, los operadores son reversibles (podemos definir el operador que va hacia atrás) y, además, hay un único estado final.

5. Búsqueda con adversario: los juegos

En el mundo de la inteligencia artificial, los juegos son una de las aplicaciones más antiguas, porque constituyen problemas abstractos y completamente formalizables. Uno de los primeros programas de ajedrez fue desarrollado alrededor del año 1950 por C. Shannon (el padre de la teoría de la información).

Desde un punto de vista formal, la dificultad principal para tomar decisiones en los juegos es la presencia de un adversario, puesto que este intenta movimientos que nos resultan menos favorables, lo cual introduce incertidumbre en el modelo. Otro tipo de incertidumbre que aparece en algunos juegos es el componente aleatorio (por ejemplo, los dados). Los algoritmos para tomar decisiones en los juegos han de tener en cuenta estos dos componentes.

A pesar de que los juegos están resueltos desde un punto de vista formal (hay un algoritmo para condiciones ideales de tiempos de computación y de memoria), desde el punto de vista computacional no es tan sencillo. Los juegos son problemas difíciles de resolver cuando queremos obtener una solución en un tiempo razonable y que utilice la memoria disponible en una cierta máquina. Esto es así porque los árboles de búsqueda que se deberían expandir para encontrar la solución óptima son inmensos. Por ejemplo, en el caso del ajedrez, una partida corriente corresponde a unos cien movimientos (cincuenta por jugador) y la media de su factor de ramificación es de en torno a treinta y cinco (cuando se considera un tablero se pueden hacer, de media, unos treinta y cinco movimientos diferentes). Esto corresponde a un árbol de búsqueda con 35^{100} nodos diferentes (que corresponde a $2{,}55 * 10^{154}$ nodos). Evidentemente, no podemos expandir todo el árbol y después aplicar el algoritmo que nos dé el mejor movimiento posible. Por tanto, nos encontramos con el hecho de que la incertidumbre en los juegos no es debida a que tengamos carencia de información, sino porque no se pueden calcular las consecuencias exactas de un movimiento debido a problemas de tiempo y memoria.

A continuación, describiremos los algoritmos de búsqueda para juegos. Empezaremos con los que proporcionan una decisión perfecta porque consideran que ni el tiempo ni el espacio son recursos limitados. Después, pasaremos a tener en cuenta las restricciones de los recursos. Para acabar, se considerará el elemento aleatorio.

5.1. Decisiones perfectas

La formalización de un juego cuando no se tiene que considerar el azar es similar a la formalización de un problema de búsqueda como los que hemos visto hasta ahora. Por tanto, tendremos que definir: la modelización del en-

torno en que se mueve el sistema (los estados), la modelización de las acciones del sistema y la definición del problema. Pasamos ahora a definir cada uno de los elementos.

1) Los estados. En nuestro caso, esto corresponderá a representar la información correspondiente a un instante del juego. Por ejemplo, si es un juego con un tablero y unas fichas, tendremos que representar el tablero y cómo están colocadas las fichas.

2) Las acciones. Esto corresponderá a modelizar los movimientos que pueden hacer los jugadores.

3) El problema. Como en cualquier problema de búsqueda, vendrá definido por el estado inicial y la función objetivo. En un juego, el estado inicial corresponderá a la posición del juego (por ejemplo, la situación del tablero) en un momento dado en que queremos que nuestro programa tome una decisión. Es importante subrayar que la posición inicial no siempre será la misma (y, evidentemente, no corresponde a la situación del juego a comienzos de la partida) porque cada vez que el programa tenga que decidir tendremos una posición inicial diferente. La función objetivo la descompondremos en dos funciones: un test de finalización y una función de utilidad. El test de finalización aplicado a un estado nos dirá si todavía se puede continuar jugando. La función de utilidad aplicada a un estado nos evaluará diciéndonos hasta qué punto es bueno.

Una vez hecha la formalización, si se tratase de un típico problema de búsqueda, bastaría con encontrar un camino desde el estado inicial a un estado objetivo. Sin embargo, en este caso no es así porque el camino que planificamos para encontrar la mejor solución será malogrado por el otro jugador.

Un algoritmo que nos permite encontrar la mejor jugada teniendo en cuenta las opciones de los jugadores y las posibles evoluciones de la partida una vez se ha movido es el llamado *algoritmo minimax*. Este algoritmo es para juegos de dos jugadores en los que no interviene el azar. Además, el algoritmo representa que no tenemos restricciones ni de memoria ni de tiempo y que, por tanto, podemos explorar todas las alternativas de los jugadores.

El mecanismo general de este algoritmo se basa en una función de utilidad que obtiene valores altos cuando gana un jugador y bajos cuando gana el otro. De este modo, para un determinado jugador el algoritmo elegirá aquella opción que le permite conseguir optimizar la utilidad.

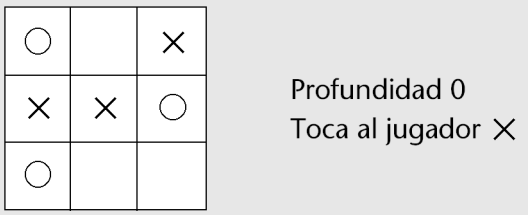
Antes de pasar a ver una implementación del algoritmo, veremos cómo se puede calcular la mejor solución para un juego y un tablero concretos.

Modelización del juego del tres en raya

Consideraremos el juego del tres en raya (con jugadores *x* y *o*, como es habitual), en la versión en la que los jugadores van tirando por turnos poniendo sus fichas hasta que uno de los dos gana (poniendo sus tres fichas haciendo una línea o raya) o el tablero está completamente lleno. Si el tablero está lleno y ninguno de los dos jugadores ha ganado, diremos que los dos jugadores empatan. Estimamos un problema en el que la partida ya ha empezado y le toca al jugador *x*: tomaremos el tablero que se muestra en la figura 22.

Figura 22

Tablero inicial para el tres en raya



La modelización del problema del tres en raya es bastante evidente:

1) Estados. Los tableros que pueden aparecer en el juego definirán el conjunto de estados posibles. Así, tendremos que son estados todos los tableros de 3 × 3 en los que las posiciones están vacías o llenas con alguna de las dos fichas posibles (*o* o *x*).

2) Acciones. Serán acciones a considerar las de añadir una pieza al tablero en una posición vacía.

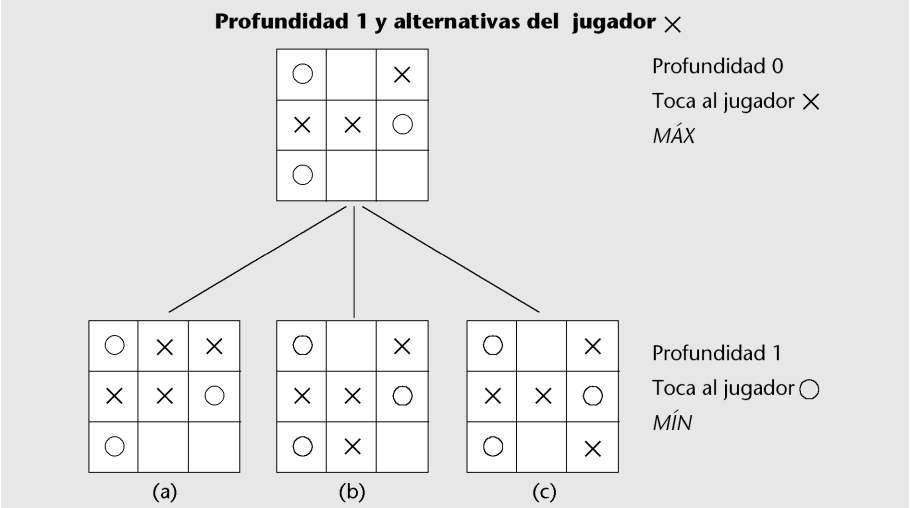
3) El problema. El estado inicial será un tablero concreto. En nuestro caso, tomaremos el del tablero representado en la figura 22. El test de finalización será, evidentemente, comprobar si alguien ya ha hecho tres en raya o si ya no se pueden poner más fichas.

Como se ha dicho antes, la función de utilidad tiene que permitir evaluar los estados de forma que si gana un jugador se obtengan valores altos y para el otro bajos. Aquí supondremos que el jugador *x* es a quien le interesa una puntuación alta (denominaremos a este jugador MÁX, de máximo). Entonces, haremos que la función nos retorne un valor de 1 si gana *x*. En cambio, al jugador *o* lo llamaremos MÍN (de mínimo) y definiremos la función de utilidad de forma que si gana MÍN retorne -1. En el supuesto de que los dos jugadores queden empatados, la función de utilidad devolverá 0. Esta definición es consistente con la idea de que el jugador *x* prefiere ganar a empatar, y empatar a perder (dado que al jugador MÁX prefiere el valor 1 al valor 0, y el valor 0 al –1, esto es, siempre le interesa el valor más alto). De manera análoga, *o* prefiere ganar a empatar y empatar a perder (prefiere el valor de –1 al 0, y el 0 al 1).

Es importante señalar que la función de utilidad solo se aplica a estados terminales (cuando el test de finalización se cumple): tableros donde gana un jugador o cuando están empatados porque no se puede poner ninguna otra pieza. La función de utilidad no se puede aplicar (no tiene sentido) cuando hay tableros con posiciones vacías en que no gana nadie.

Seguimos con el ejemplo del juego del tres en raya:

Figura 23



El algoritmo minimax toma las decisiones considerando las diferentes alternativas. Ahora veremos la manera en que se toman las decisiones utilizando el tablero inicial:

1) En este caso, tenemos que el jugador *x* puede hacer tres movimientos diferentes que llevan a los tres tableros de la figura anterior, con profundidad 1 y donde se representan las alternativas del señor *x* –denotaremos estos tableros con (a), (b) y (c).

2) Una vez tenemos estas alternativas, las hemos de evaluar para poder decidir cuál es la mejor.

3) Cuando sean evaluadas, de entre los tres tableros elegiremos aquel que tenga un valor más alto (porque *x* es el jugador máximo).

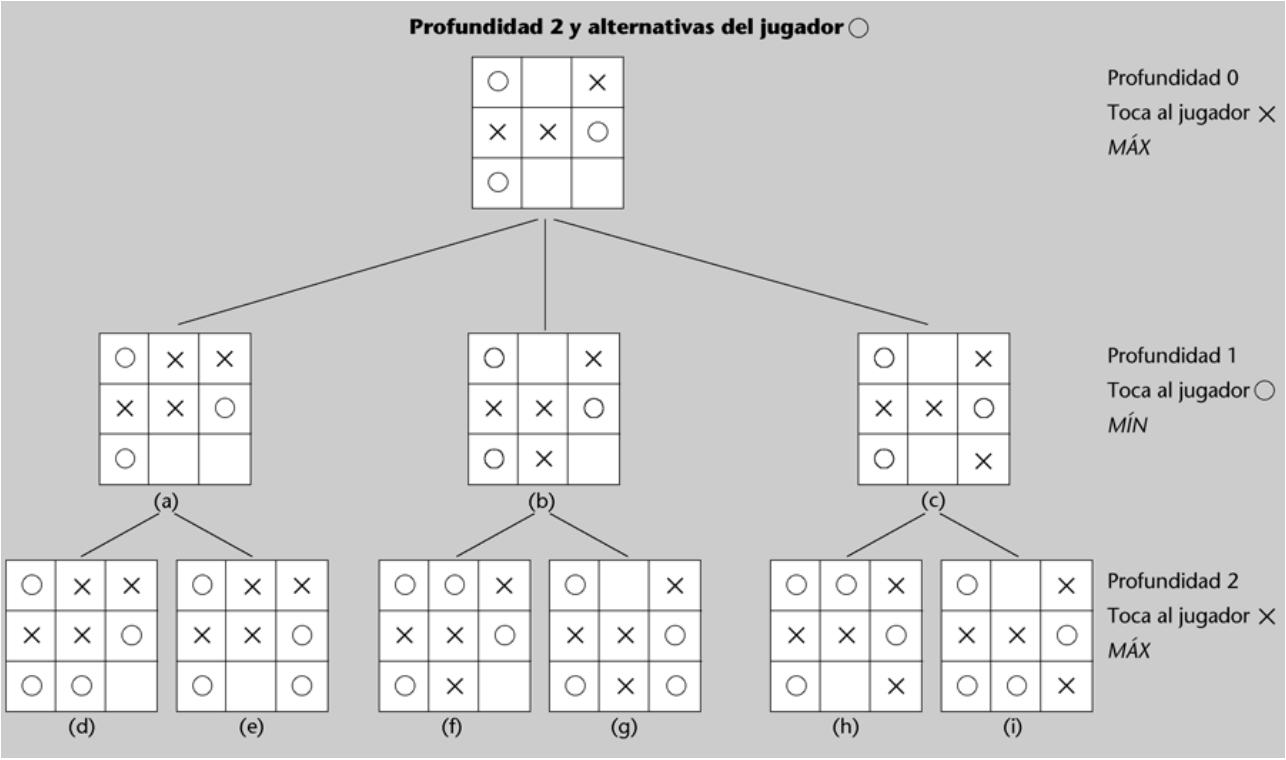
Desafortunadamente, la evaluación de los tres tableros no es trivial porque no son posiciones en las que un jugador gane (tampoco empatan). Así que no se puede aplicar la función de utilidad.

Cuando la evaluación no se puede hacer mediante la función de utilidad, podemos aplicar el mismo esquema que hemos aplicado al tablero inicial: considerar qué jugadas puede hacer el jugador, evaluar las jugadas y elegir la que le

va mejor al jugador. En nuestro caso, esto representa que para cada uno de los tres tableros tenemos que considerar todos los movimientos que puede hacer el jugador –en este caso es el jugador *o*– y evaluarlos.

La expansión de estos tres tableros se muestra en la figura con profundidad 2, donde se representan las alternativas del señor *o*. Para cada tablero, obtenemos dos nuevos. Por ejemplo, para el tablero (a) obtenemos los nuevos tableros (d) y (e).

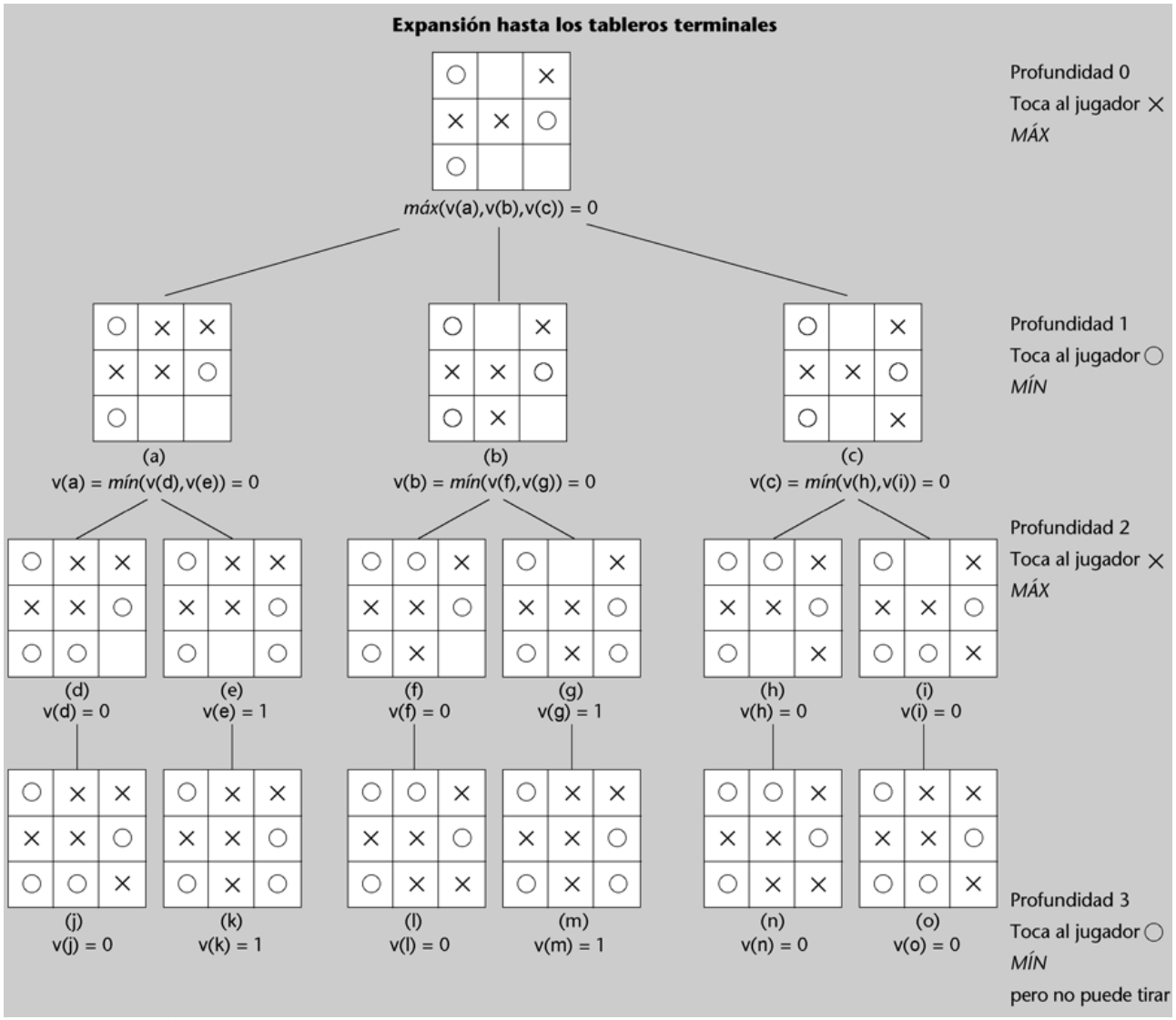
Figura 24



Si los nuevos tableros fueran terminales, los evaluaríamos. Para determinar la evaluación de los tableros (a), (b) y (c) que nos quedaba pendiente, elegiremos en cada caso el mejor de los valores de acuerdo con el jugador que tiene que jugar. Dado que ahora el jugador es *o*, elegiremos el movimiento que tiene un valor más bajo. Así, para determinar el valor del tablero (a), calcularemos el mínimo de los valores de los tableros (d) y (e). Del mismo modo:

$$v(\text{b}) = \min(v(\text{f}), v(\text{g})) \text{ y } v(\text{c}) = \min(v(\text{h}), v(\text{i})).$$

Figura 25



Desafortunadamente, en este caso los nuevos tableros tampoco son terminales y no podemos aplicar la función de utilidad. La alternativa es volver a aplicar el mismo procedimiento: calcular los posibles movimientos, evaluarlos y elegir el mejor (de acuerdo con a quién le toca tirar). La figura 25 muestra la expansión de los últimos estados de la figura con profundidad 2 y donde se representan las alternativas del señor *x*. Ahora, para cada tablero que expandimos (d), (e), (f), (g), (h) e (i) solo hay un movimiento posible (es decir, solo hay una posición vacía para cada tablero) y, además, los tableros que se generan son terminales. Por tanto, los podemos evaluar con la función de utilidad. La figura muestra estas evaluaciones: $v(j) = v(l) = v(n) = v(o) = 0$ porque corresponden a tableros en que se empata, y, en cambio, $v(k) = v(m) = 1$ porque gana el jugador *x*.

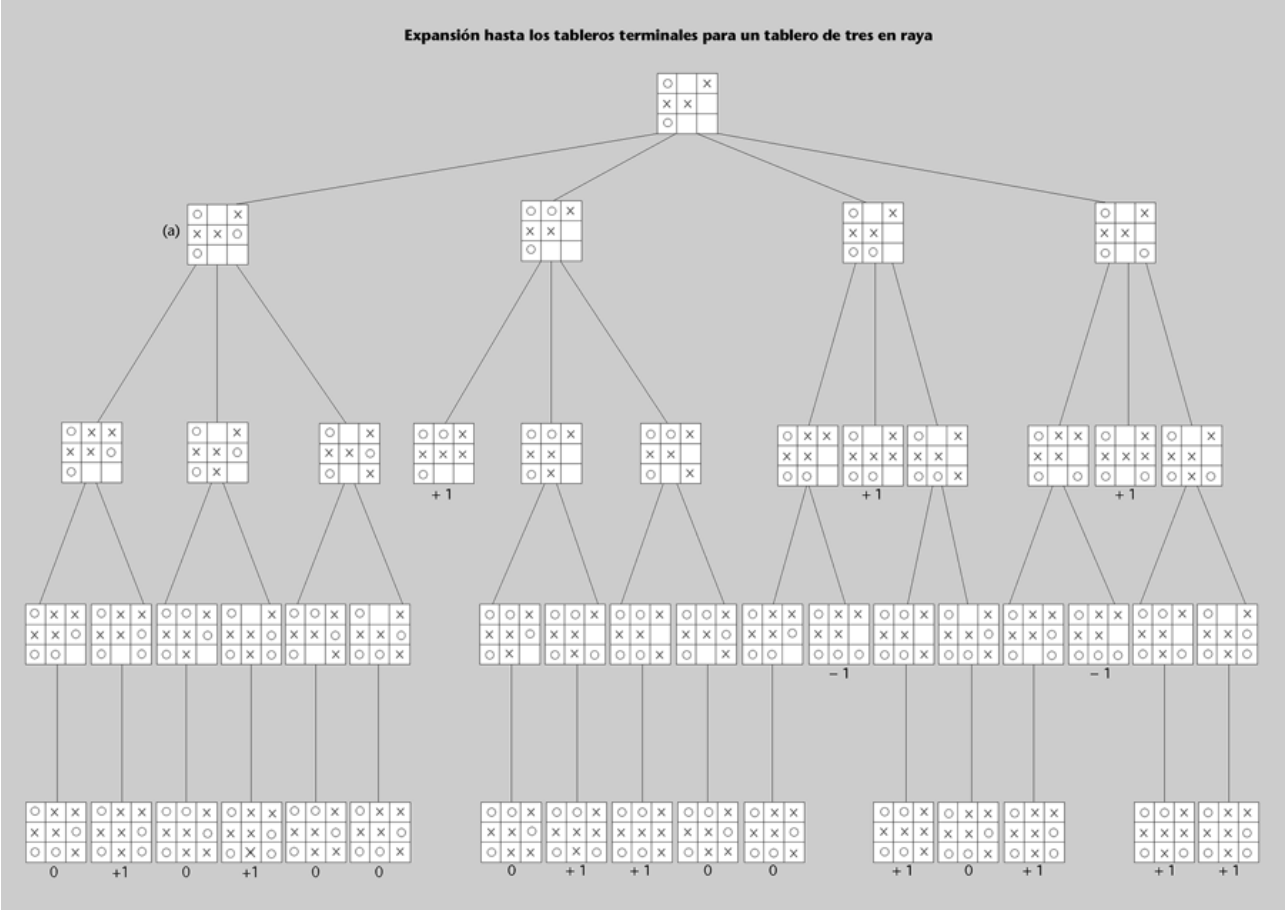
Una vez determinado el valor de los tableros obtenidos a profundidad 3, tenemos que calcular la evaluación a profundidad 2. El jugador *x* (a quien le toca jugar a profundidad 2) tiene que elegir la alternativa más buena (la del

valor máximo) pero, como que en este caso solo hay una, la evaluación de los tableros (d), (e), (f), (g), (h) e (i) corresponderá a la de los tableros (j), (k), (l), (m), (n) y (o).

Una vez evaluados los tableros de profundidad 2, evaluamos los de profundidad 1. Los habíamos dejado pendientes porque no se podía aplicar la función de utilidad a los de profundidad 2. Como se ha dicho, $v(a) = \min(v(d), v(e))$, $v(b) = \min(v(f), v(g))$ y $v(c) = \min(v(h), v(i))$ para reflejar que consideramos qué hará el jugador *o* en cada caso: elegir de entre las diferentes alternativas la que lleva a un valor más bajo. Ahora bien, esta elección es significativa solo para los tableros (a) y (b), porque para el tablero (c) las dos opciones tienen la misma evaluación. En el tablero (a), el jugador elegiría (d) con objeto de no perder (el tablero (d) está evaluado en 0 y el tablero (e) en 1). En el tablero (b) se elegiría (f) por la misma razón (el tablero (f) está evaluado en 0 y (g) en 1).

Una vez hecha la evaluación de los tableros (a), (b) y (c), el jugador *x* puede elegir entre las tres alternativas que se planteaban inicialmente (profundidad 0). En este caso, y suponiendo que los dos jugadores juegan siempre lo mejor posible, se ve que el resultado final será de empate: en los tres casos se llega al empate. Por tanto, cualquier movimiento es adecuado.

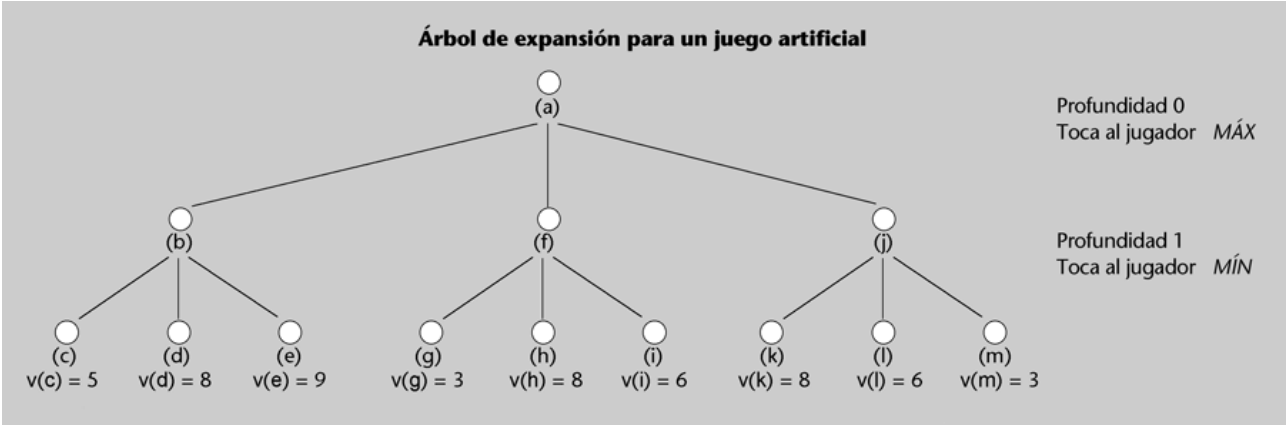
Figura 26



La figura 26 presenta el árbol de expansión para un tablero similar al de la figura 22 pero con una ficha menos. En este caso, se puede ver que la evaluación del árbol permite elegir al jugador *x* un movimiento que le permite no perder la partida. En este árbol, el tablero (a) es el de la figura 22 y, por tanto, su evaluación corresponde a la evaluación que aparece en la figura que presenta la expansión hasta los tableros terminales (figura 25).

La figura 27 muestra el árbol de expansión correspondiente a otro juego. Se consideran dos niveles y la función de utilidad aplicada a todos los nodos del último nivel. Si suponemos que el jugador a quien le toca mover en el estado inicial (a) es el jugador MÁX, tendremos que con el esquema planteado dicho jugador elegirá como mejor opción la correspondiente al estado (b).

Figura 27



La implementación

A continuación, hacemos la formalización del algoritmo minimax. Este algoritmo corresponde al proceso que se ha seguido más arriba para decidir el mejor movimiento para el tablero de la figura 22.

```
def decision_minimax (toca, pieza_min, pieza_max, tablero):
    f_aux_max = lambda th: [valor_minimax(pieza_max, \
                                          pieza_min, \
                                          pieza_max, th), th]
    f_aux_min = lambda th: [valor_minimax(pieza_min, \
                                          pieza_min, \
                                          pieza_max, th), th]
    if gana(pieza_min, tablero) or gana(pieza_max, tablero) or lleno(tablero):
        return funcionUtilidad(pieza_min, pieza_max, tablero)
    elif toca == pieza_min:
        return cadr(selecciona_min(mapcar(f_aux_max, \
                                          opciones_tablero(pieza_min, tablero))))
    else:
        return cadr(selecciona_max(mapcar(f_aux_min, \
```

```
                                        opciones_tablero(pieza_max, tablero))))
```

La función recibe como parámetros un tablero, las dos piezas que juegan (`pieza_min` y `pieza_max`) y la pieza a la que le toca jugar (`toca`).

Máximo y mínimo

Las piezas corresponderán, respectivamente, al máximo y al mínimo, a pesar de que, de hecho, cuando se hace la llamada no afectará al resultado. Dará lo mismo `decision_minimax('o', 'o', 'x', tablero)` que `decision_minimax('o', 'x', 'o', tablero)`. Observad que si hacemos `decision_minimax('x', 'o', 'x', tablero)` con el tablero `[['o', '_', 'x'], ['x', 'x', 'o'], ['o', '_', '_']]` –el caso de la figura 25–, las evaluaciones de los tableros son como aparecen en la figura. En este caso, cuando se elija el movimiento de *x*, se elegirá un tablero con el valor más alto. En cambio, si la llamada es `decision_minimax('x', 'x', 'o', tablero)`, tendremos que las evaluaciones de los tableros serán las mismas pero cambiadas de signo. Pero en este caso, dado que `'x'` es la pieza del jugador mínimo (`pieza_min`) y `'o'` es la del jugador máximo (`pieza_max`), en cada nivel se intercambian los papeles y esto hace que la elección sea la misma. Tened en cuenta que la función de utilidad que se hace más adelante da –1 para la pieza mínima, con independencia de si es `'o'` o `'x'`.

La función mirará primero si se puede poner una pieza y, si es así, generará todos los posibles movimientos –esto lo hace `opciones_tablero(pieza_min, tablero)`– y los evaluará. Para hacer esto último se aplica a cada tablero hijo `th` la función:

```
lambda tablero_hijo: [valor_minimax(pieza_max, pieza_min, \
                                    pieza_max, tablero_hijo), tablero_hijo]
```

que genera para cada tablero un par con la evaluación del tablero y después el tablero mismo. La función `selecciona_min` (respectivamente, la función `selecciona_max`) retorna el mejor tablero. Esto es, el que tiene una evaluación más baja (respectivamente, el que tiene una evaluación más alta).

La evaluación de los tableros la hace la función que se denomina `valor_minimax`. Esta función mirará si con el tablero que tenemos ya podemos determinar el valor (si algún jugador gana o si el tablero está lleno). Si es así, se retorna el valor y, en caso contrario, se consideran las diferentes alternativas (se evaluarán haciendo una llamada recursiva a la misma función `valor_minimax`) y después se elige la mejor. La elección, como antes, la hará `selecciona_min` y `selecciona_max`.

```
def valor_minimax (toca, pieza_min, pieza_max, tablero):
    f_aux_max = lambda th: [valor_minimax(pieza_max, pieza_min, pieza_max, th), th]
    f_aux_min = lambda th: [valor_minimax(pieza_min, pieza_min, pieza_max, th), th]
    if gana(pieza_min, tablero) or gana(pieza_max, tablero) or lleno(tablero):
        return funcionUtilidad(pieza_min, pieza_max, tablero)
    elif toca == pieza_min:
```

```
        return car(selecciona_min(mapcar(f_aux_max, \
                                  opciones_tablero(pieza_min, tablero))))
    else:
        return car(selecciona_max(mapcar(f_aux_min, \
                                  opciones_tablero(pieza_max, tablero))))
```

Las funciones para seleccionar el tablero mínimo y máximo aparecen definidas a continuación. Estas funciones toman una lista de pares valor-tablero y retornan el par que tiene un valor más bajo o más alto (según sea `selecciona_min` o `selecciona_max`). Las dos funciones tienen un funcionamiento parecido, cogen los dos primeros elementos y miran cuál es el mejor. Si es el segundo, hacen una llamada recursiva a la misma función una vez eliminado el primero. `cdr(pares_v_t)` eliminará el primer elemento. En cambio, si el mejor elemento es el primero, harán la llamada eliminando el segundo. De hecho, esto se hace construyendo una nueva lista en la que el primer elemento es el que antes era primero, `car(pares_v_t)` y el resto de la lista son todos los que teníamos a partir del tercero `cddr(pares_v_t)`. Así, la nueva lista es:

```
cons(car(pares_v_t),cddr(pares_v_t))
```

La condición de finalización de la función es cuando la lista solo tiene un elemento (cuando la cola de la lista está vacía – `cdr(pares_v_t)==[]`. En este caso el primero (y único) elemento es la solución.

```
def selecciona_min (pares_v_t):
    if cdr(pares_v_t) == []:
        return car(pares_v_t)
    elif caar(pares_v_t) > caadr(pares_v_t):
        return selecciona_min(cdr(pares_v_t))
    else:
        return selecciona_min(cons(car(pares_v_t),cddr(pares_v_t))

def selecciona_max (pares_v_t):
    if cdr(pares_v_t) == []:
        return car(pares_v_t)
    elif caar(pares_v_t) < caadr(pares_v_t):
        return selecciona_max(cdr(pares_v_t))
    else:
        return selecciona_max(cons(car(pares_v_t),cddr(pares_v_t)))
```

Las funciones que hemos explicado antes son generales para implementar el minimax para cualquier juego. Ahora indicamos las propias del tres en raya. Empezamos con la función de utilidad, la que comprueba si el tablero está lleno y la de generar todos los posibles movimientos para un tablero dado. Para

hacer estas funciones, tenemos que definir la estructura de datos del tablero. La representación del tablero será una lista en que cada elemento corresponderá a una fila. La representación de la fila también utilizará una lista. Así, tendremos que un tablero es una lista de listas. Las posiciones vacías las representaremos con el símbolo `'_'`. Por tanto, el tablero de la figura inicial se representará:

```
[['o', '_', 'x'], ['x', 'x', 'o'], ['o', '_', '_']]
```

Pasamos ahora a la definición de las funciones:

```
def funcionUtilidad (pieza_min, pieza_max, tablero):
    if gana(pieza_min, tablero):
        return -1
    elif gana(pieza_max, tablero):
        return 1
    elif lleno(tablero):
        return 0
    else: return None

def lleno (tablero):
    return ('_' not in car(tablero)) and \
        ('_' not in cadr(tablero)) and \
        ('_' not in caddr(tablero))

def gana (pieza, tablero):
    return alguna_columna(pieza,tablero) or \
        alguna_diagonal(pieza,tablero) or \
        alguna_fila(pieza,tablero)

def alguna_diagonal (pieza, tablero):
    diag1 = (pieza == car(car(tablero))) and \
            (pieza == cadr(cadr(tablero))) and \
            (pieza == caddr(caddr(tablero)))
    diag2 = (pieza == caddr(car(tablero))) and \
            (pieza == cadr(cadr(tablero))) and \
            (pieza == car(caddr(tablero)))
    return diag1 or diag2

def alguna_fila (pieza, tablero):
    fila1 = (pieza == car(car(tablero))) and \
            (pieza == cadr(car(tablero))) and \
            (pieza == caddr(car(tablero)))
    fila2 = (pieza == car(cadr(tablero))) and \
            (pieza == cadr(cadr(tablero))) and \
            (pieza == caddr(cadr(tablero)))
    fila3 = (pieza == car(caddr(tablero))) and \
```

```
             (pieza == cadr(caddr(tablero))) and \
             (pieza == caddr(caddr(tablero)))
    return fila1 or fila2 or fila3

def alguna_columna (pieza, tablero):
    col1 = (pieza == car(car(tablero))) and \
           (pieza == car(cadr(tablero))) and \
           (pieza == car(caddr(tablero)))
    col2 = (pieza == cadr(car(tablero))) and \
           (pieza == cadr(cadr(tablero))) and \
           (pieza == cadr(caddr(tablero)))
    col3 = (pieza == caddr(car(tablero))) and \
           (pieza == caddr(cadr(tablero))) and \
           (pieza == caddr(caddr(tablero)))
    return col1 or col2 or col3

## Dado un tablero, genera todos los movimientos posibles.
def opciones_tablero (pieza, tablero):
    if tablero == []:
        return []
    caps = opciones_ningun_tablero(pieza, car(tablero))
    faux1 = lambda el_resto: cons(car(tablero), el_resto)
    faux2 = lambda una_cap: cons(una_cap, cdr(tablero))
    if caps == []:
        return mapcar(faux1,opciones_tablero(pieza, cdr(tablero)))
    else:
        return mapcar(faux2, caps) + mapcar(faux1, opciones_tablero(pieza, cdr(tablero)))

def opciones_ningun_tablero (pieza, ningun_tablero):
    aux = lambda x: cons(car(ningun_tablero), x)
    if ningun_tablero == []:
        return []
    elif car(ningun_tablero) == '_':
        llaux = mapcar(aux, opciones_ningun_tablero(pieza,cdr(ningun_tablero)))
        return cons(cons(pieza,cdr(ningun_tablero)),llaux)
    else:
        llaux = mapcar(aux, opciones_ningun_tablero(pieza,cdr(ningun_tablero)))
        return llaux
```

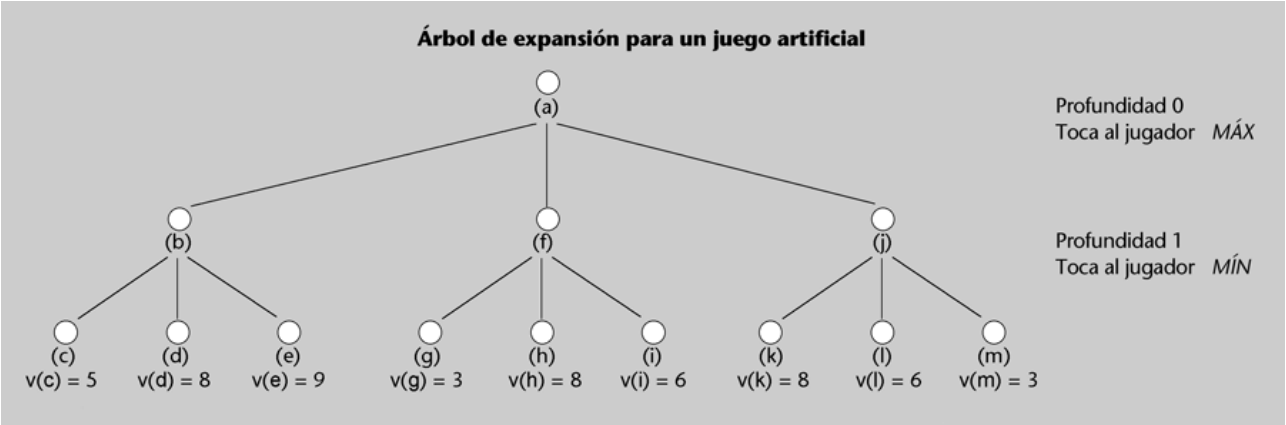
5.1.1. La poda α-β

Si hacemos un análisis cuidadoso del ejemplo del árbol de expansión para el juego artificial de la figura 27, encontraremos que hay nodos del árbol que no habría que expandir porque antes de visitarlos ya sabemos que nunca serán seleccionados. Cuando ocurra este caso, el algoritmo de poda α-β no evaluará las ramas ahorrándonos la correspondiente expansión del árbol.

Procedimiento minimax

Antes de pasar a ver el algoritmo, reseguimos el procedimiento de minimax aplicado a la figura 27. Recordamos esta figura:

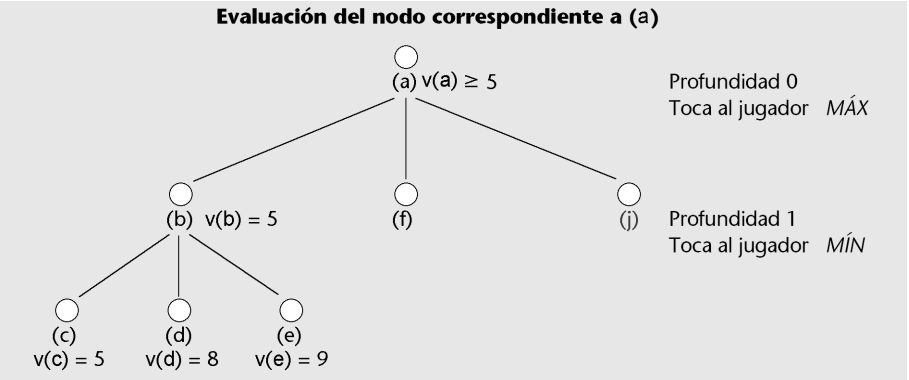
Figura 27



Si empezamos la evaluación del árbol (consideraremos la evaluación de los nodos de izquierda a derecha), nos encontraremos que cuando expandimos el nodo (a) obtenemos los nodos (b), (f) y (j). Para evaluar (b), lo tenemos que expansionar y asignar a (b) el mínimo de los valores de los tres hijos. Así, v(b) = min(v(c), v(d), v(e)). Por tanto, v(b) = 5.

Antes de pasar a la evaluación del nodo (f), podemos observar que como que en el nodo (a) el jugador que tiene que mover es MÁX, la evaluación de (a) no será nunca más pequeña que 5, porque si los nodos (f) y (j) se evaluaran con un valor más pequeño que 5, cuando MÁX decida el movimiento del nodo (a), elegirá la opción (b) en lugar de (f) o (j). Así, pues, ponemos v(a) ≥ 5 en el nodo (a) para indicarlo. La figura 28 representa esta situación.

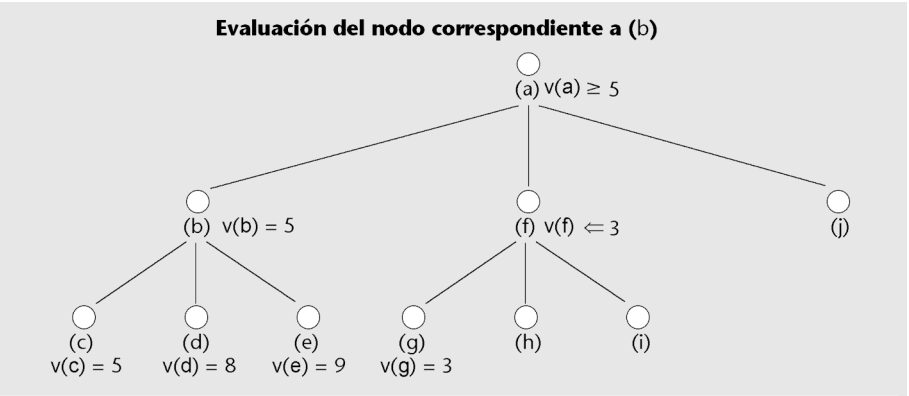
Figura 28. Evaluación del nodo (a)



Árbol de expansión, evaluación del nodo correspondiente a (b). Hito inferior para la evaluación de (a).

Pasamos, ahora, a evaluar el nodo (f). En este caso, empezamos con la evaluación del nodo (g). Dado que su evaluación es 3, podemos afirmar que la evaluación de (f) será más pequeña o igual que 3 ((f) es un estado que corresponde al jugador MÍN). Por tanto, podemos escribir v(f) ≤ 3 (la figura 29 corresponde a este nuevo paso).

Figura 29. Evaluación del nodo (b)



Hito inferior para la evaluación de (a) y superior para v(f).

Observad que:

- Si la evaluación del nodo (h) es **más pequeña** que la de (g), entonces el jugador MÍN cuando esté en el estado (f) elegirá (h) en lugar de (g).

- Si la evaluación del nodo (h) es **más grande** que la de (g), entonces el jugador MÍN preferirá (g).

Poda del ejemplo del árbol de expansión para el juego artificial

Una vez sabemos que la evaluación de (f) es menor que 3, dado que conocemos que la del nodo (a) es más grande que 5 y que (a) es un nodo que corresponde al jugador MÁX, también sabemos que el jugador MÁX cuando esté en el nodo (a) no elegirá nunca (f). Sean las que sean las evaluaciones de (h) y de (i), la evaluación de (f) siempre será más pequeña o igual que 3 y, por tanto, el jugador MÁX elegirá siempre (b) antes de que (f). Por tanto, no hay que evaluar los nodos (h) e (i) porque su evaluación es irrelevante a la hora de tomar la decisión. El hecho de no considerar los nodos (h) e (i) se denomina poda. En el caso de la evaluación del nodo (j), empezamos haciendo la evaluación de (k). En este caso, encontramos que es 8 y, por tanto, v(j) ≤ 8. Pero esto no nos permite podar porque 8 es más grande que el hito que tenemos ahora para v(a). Por tanto, si se confirmara este valor, el jugador MÁX preferiría (j) a (b). A continuación, evaluamos el nodo de (l) y nos da 6. A pesar de que el valor es menor que 8 –y, por tanto, tendremos

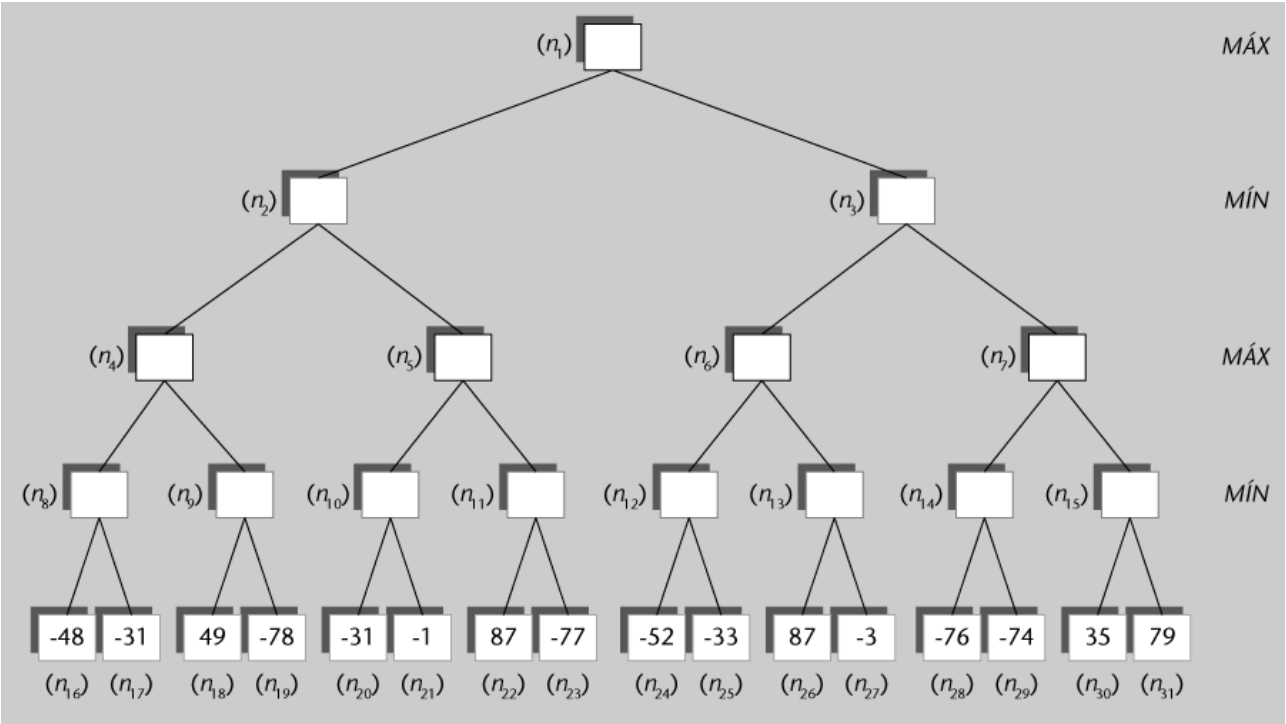
una modificación del hito porque el jugador MÍN preferirá (l) a (k)–, este valor no nos permite podar el árbol porque esta rama es la alternativa más buena en este momento para el jugador MÁX.

Seguidamente, evaluamos el estado (m) y esto nos retorna 3. Este valor dejará $v(j) \leq 3$ y permitiría –si hubieran otros nodos hermanos de (m)– hacer una poda. No es este el caso y, por tanto, no podemos podar.

Ejemplo de minimax con algoritmo de poda α-β

A continuación, vemos otro ejemplo de aplicación del algoritmo minimax con el algoritmo de poda α-β para el ejemplo de un juego artificial representado en la figura 30.

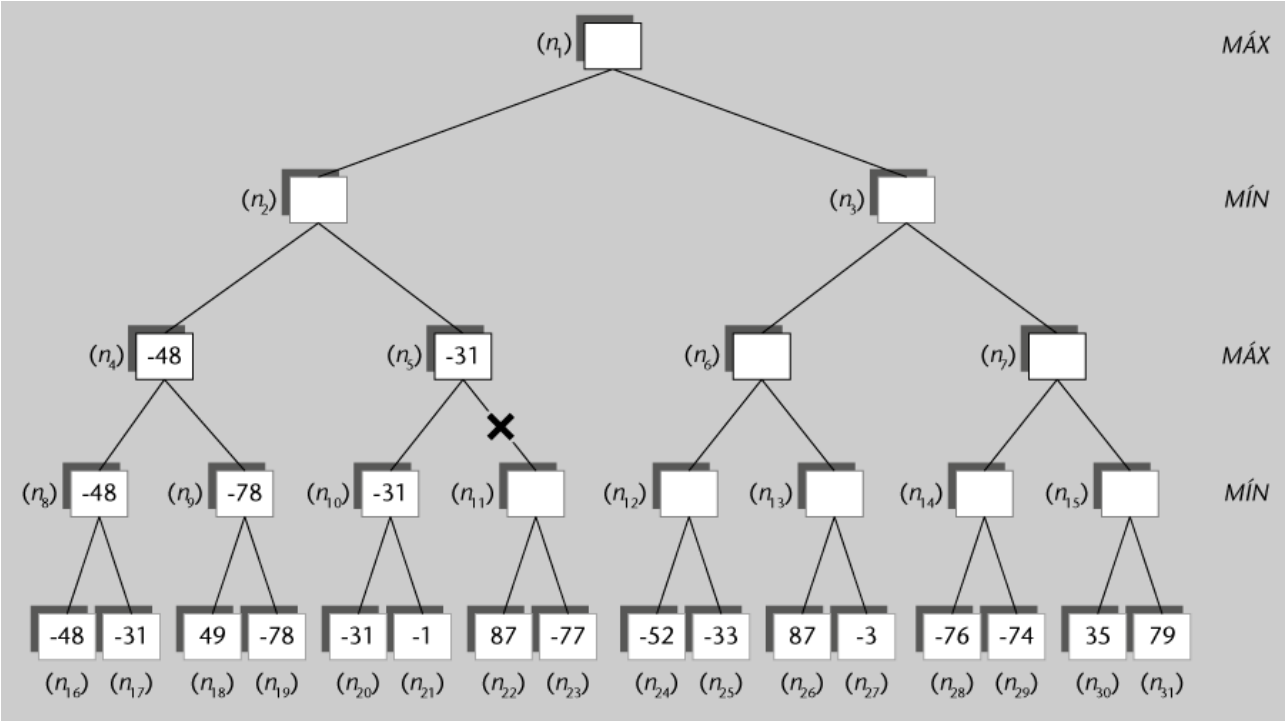
Figura 30



El árbol se va resolviendo de izquierda a derecha y de abajo hacia arriba. Por ejemplo, en el primer paso, el valor del nodo (n_8) se elige entre el mínimo de los valores de los nodos (n_{16}) y (n_{17}). Después, para el nodo (n_9) se evalúa primero el nodo (n_{18}), que tiene valor 49. Dado que este valor es superior al valor asignado al nodo (n_8), el nodo (n_9) también explora el nodo (n_{19}) por si encuentra un valor más pequeño. Por lo tanto, el nodo (n_{19}) no es expurgado y, de hecho, como que se trata de un valor más pequeño, se asigna al nodo (n_9). A continuación, se asigna a (n_4) el valor máximo de los nodos (n_8) y (n_9). Se sigue con la evaluación del nodo (n_{10}), que primero explora el nodo (n_{20}), con valor -31. Dado que -31 es más grande que -48 (el valor de (n_4)), el algoritmo también explora el nodo (n_{21}) por si encuentra un valor mejor. No es ese el caso, así que se acaba asignando el valor -31 al nodo (n_{10}). En este punto es cuando encontramos el primer momento en que se puede expurgar una rama del árbol, tal como se ve en la figura 31. Cuando se evalúa el nodo (n_5), como

que se trata de un nodo MÁX, este tendrá un valor a decidir entre el nodo (n_{10}), con valor -31, y el nodo (n_{11}), con un valor por explorar. Dado que el nodo (n_2) es un nodo MÍN y el valor del nodo (n_4) es -48, más pequeño que -31, el nodo MÁX (n_5) no conseguirá ninguna mejora aunque el nodo (n_{11}) tenga un valor más grande que -31. Así, pues, el nodo (n_5) decide expurgar la rama correspondiente al nodo (n_{11}) y asignar el valor del nodo (n_{10}) al nodo (n_5).

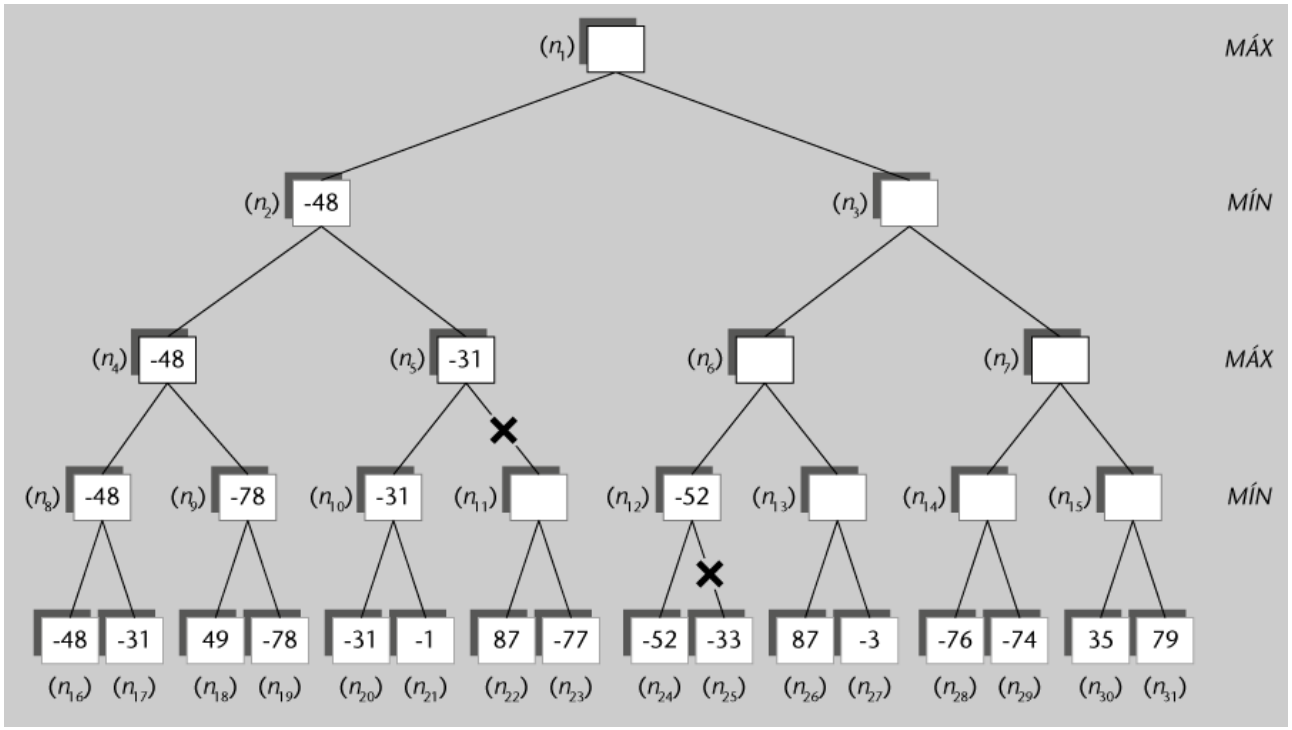
Figura 31



En las figuras siguientes (figuras 32, 33 y 34) solo se visualizan los pasos en los que se produce alguna expurgación y solo se explicará el detalle de estos pasos.

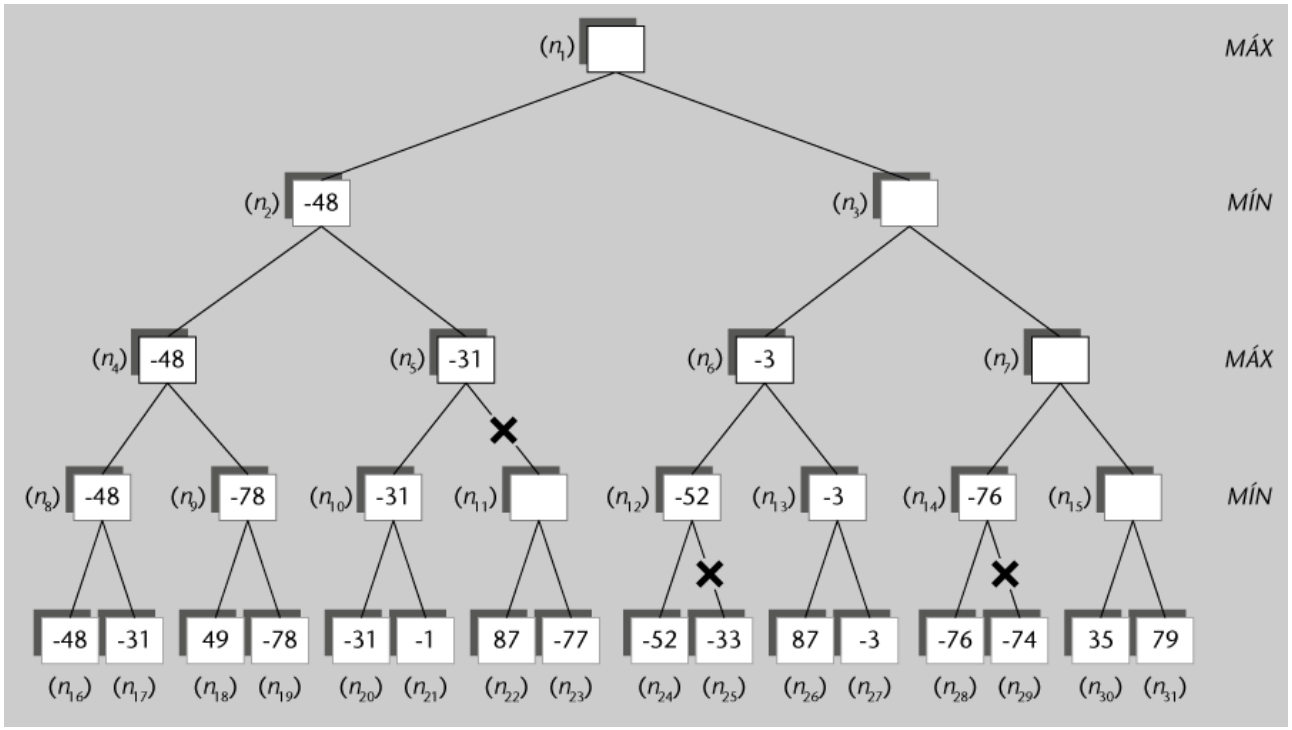
En la figura 32 se puede ver que, una vez que el nodo (n_{12}) ha explorado el nodo (n_{24}), con valor -52, se puede expurgar la rama correspondiente al nodo (n_{25}), dado que el valor -52 ya es más pequeño que el valor que se tiene en el nodo (n_2), con valor -48. Dado que por la parte izquierda del árbol, el mejor valor que ha podido subir el jugador MÍN (con la oposición del jugador MÁX) ha sido -48 y en la raíz está el jugador MÁX, al jugador MÍN le bastará con intentar subir cualquier valor que sea inferior a -48 por la parte derecha del árbol.

Figura 32



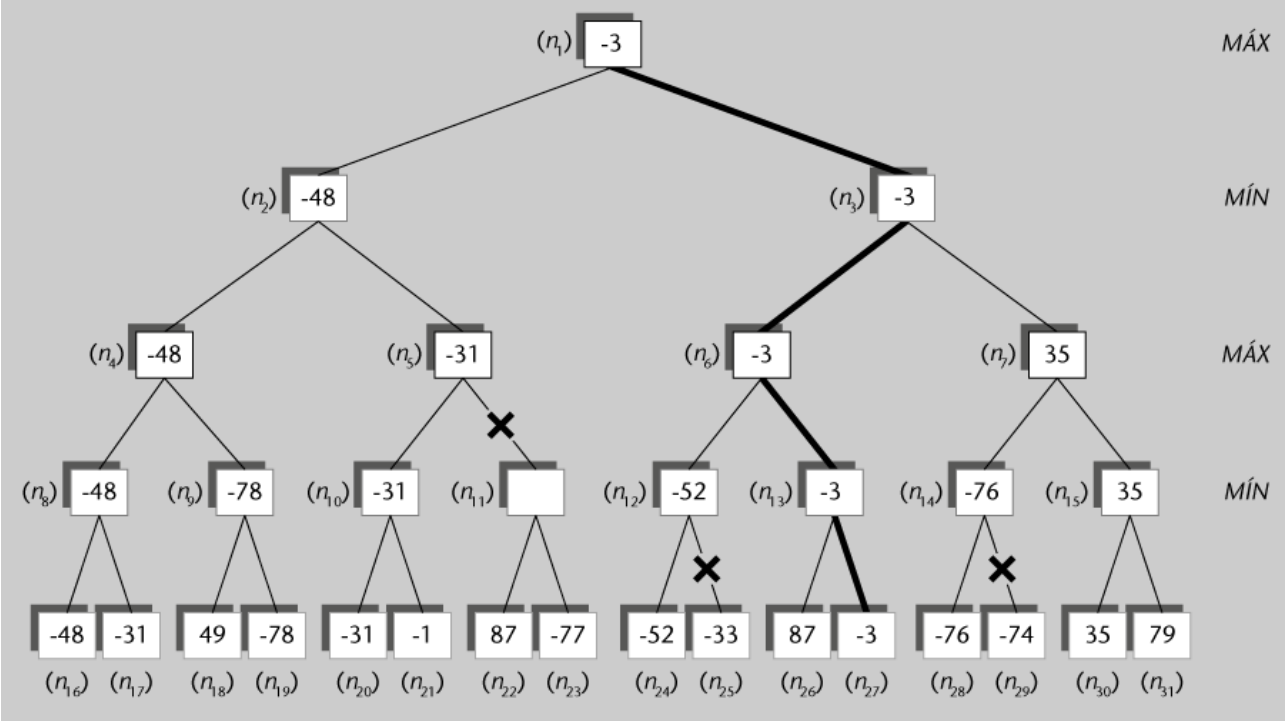
Más adelante, como se observa en la figura 33, por el mismo motivo que anteriormente, el nodo (n_{14}) puede expurgar la rama correspondiente al nodo (n_{29}) puesto que cuando ha evaluado el nodo (n_{28}) ha obtenido un valor (-76) que es más bajo que el del nodo (n_2), con valor -48.

Figura 33



Finalmente, como se observa en la figura 34, no se produce ninguna expurgación más y se acaban hallando las decisiones óptimas que tomará cada uno de los jugadores (mostrado con línea más gruesa). Así, pues, en el primer turno el jugador MÁX elegirá la rama derecha. A continuación, el jugador MÍN decidirá optar por la rama izquierda. Seguidamente, el jugador MÁX escogerá la rama derecha. Finalmente, el jugador MÍN optará por la rama derecha.

Figura 34



La implementación

La implementación del algoritmo de poda α-β se basa en dos constantes `alfa` y `beta`. La primera corresponde a la meta de lo mejor que puede conseguir el jugador MÁX y `beta` a la meta de lo mejor que puede conseguir MÍN. Por tanto, `alfa` tenderá a crecer (el jugador MÁX cambiará la meta si puede conseguir un valor mayor) y beta tenderá a decrecer (el jugador MÍN cambiará la meta si puede conseguir un valor menor).

Un jugador podará una rama cuando `alfa` ≥ `beta`. Esto es así porque el jugador MÁX no elegirá nunca un valor peor que aquel que ya tiene asegurado. Así, si hay un valor (`beta`) menor que `alfa`, al jugador MÁX no le interesa y por eso lo poda. Por tanto, si `beta` ≤ `alfa` se poda. Por otro lado, el jugador MÍN tampoco elegirá nunca un valor peor que el que tiene asegurado. Así, si hay un valor (`alfa`) mayor que `beta`, no interesará al jugador MÍN. Por tanto, cuando `alfa` ≥ `beta` se poda.

A continuación, indicamos el algoritmo de poda:

```
funcion poda-alfa-beta (nodo, alfa, beta) se
    si limit(busqueda) entonces retorna valor(nodo)
    si nivel(nodo)=min entonces
        repetir hasta (no queden hijos) o (alfa ≥ beta) hacer
            r:=poda-alfa-beta(hijo, alfa, beta)
            si r<beta entonces beta:=r
        fin repetir
        retorna beta
    sino ;; nivel(nodo)=max
        repetir hasta (no queden hijos) o (alfa ≥ beta) hacer
                r:=poda-alfa-beta(hijo, alfa, beta)
                si r>alfa entonces alfa:=r
                fin repetir
            retorna alfa
            fsi
ffuncion
```

La llamada a este procedimiento se tiene que hacer asignando al nodo el correspondiente al estado inicial, a `alfa` un valor muy pequeño (– infinito) y a `beta` un valor muy grande (infinito). Estas asignaciones de `alfa` y `beta` permitirán que las metas se vayan modificando a medida que se expanden los nodos.

5.2. Decisiones imperfectas

Ajedrez

Este es el caso del juego del ajedrez que se ha comentado en la introducción.

El algoritmo que hemos presentado representa que buscamos todos los nodos hasta llegar a los estados terminales. Sin embargo, esto no siempre es posible. El algoritmo minimax tiene un coste exponencial puesto que tiene que recorrer todos los nodos de todos los niveles antes de tomar una decisión. Así, si la búsqueda se hace hasta una profundidad d y tenemos un factor de ramificación de b, el minimax tiene un coste $O(b^d)$ –expandimos todos los nodos hasta la profundidad d. Por tanto, en la mayoría de los casos no es posible expandir todo el árbol. Aquí $b = 35$ y $d = 100$, por lo que se tendrían que expandir del orden de $35^{100} = 2{,}55 * 101^{54}$ nodos. De manera indicativa, en la tabla que veremos se muestran los tiempos de ejecución del tres en raya para tableros con diferente número de posiciones vacías. También se puede ver la evolución exponencial en relación con la profundidad a la que tiene que llegar el minimax.

La poda α-β reduce el número de nodos que se consideran y reduce el coste del algoritmo. De hecho, en general tenemos que su coste en el caso ideal es $O(b^{d/2})$. Esto permite reducir considerablemente el tiempo de ejecución.

Esta reducción corresponde al caso ideal y necesita que los nodos estén ordenados de forma que siempre se pueda podar al máximo. En el ejemplo que hemos considerado en la figura 27, la expansión de los nodos (f) y (j) da 3 nodos que se evalúan con los mismos valores (3, 8 y 6). Se puede observar que, mientras que la ordenación (3,8,6) permite podar, la de (8, 6, 3) no. De hecho, en el caso del jugador MÍN nos interesará encontrar primero los valores más bajos. Inversamente, en el caso del jugador MÁX interesará encontrar primero los valores más altos. Desafortunadamente, no es posible tener *a priori* estas ordenaciones porque esto representaría tener ya los valores y esto es, precisamente, lo que estamos buscando.

Para tratar el caso de tener limitaciones de tiempo y de memoria, se consideran las alternativas siguientes:

1) No desarrollar todo el árbol: parar la expansión de los nodos.
2) Aplicar una función de evaluación heurística a las hojas del árbol.

La necesidad de la función heurística aparece como un efecto secundario del hecho de no expandir todo el árbol. Como no se llega a los nodos terminales, no podemos aplicar la función de utilidad porque en los nodos intermedios del árbol no se dará una situación en la que uno de los dos jugadores gane o en la que los dos jugadores empaten. Cuando se pare el desarrollo del árbol, encontraremos nodos en los que todavía es posible hacer movimientos. La función de evaluación heurística dará una estimación de la utilidad que se puede conseguir a partir de esa posición.

Dado que el rendimiento del programa dependerá en gran medida de la función herística, esta tendrá que reflejar las posibilidades de ganar. Evidentemente, la función se tendrá que definir de forma que se pueda calcular de manera eficiente y, además, tiene que coincidir en los nodos finales con la función de utilidad. Una manera de definir estas funciones es que refleje la probabilidad de ganar para ese estado.

Para limitar la búsqueda, hay varias alternativas. Una de ellas es fijar la profundidad máxima de forma que el tiempo no exceda el tiempo máximo disponible. Otra alternativa es aplicar una variante de la búsqueda iterativa en profundidad de forma que cuando el tiempo se acaba se retorna la búsqueda más profunda completada. Se define el minimax con profundidad limitada y, iterativamente, se va recalculando cada vez con una profundidad mayor. Esto corresponde a un algoritmo *anytime* (definido en el subapartado «Algunas consideraciones adicionales»).

El hecho de acotar la búsqueda provoca que las decisiones que se tomen no siempre sean óptimas.

Cabe resaltar que, cuando hay restricciones de tiempo o memoria, también es posible aplicar la poda α-β. En este caso, dado que el coste del algoritmo es $O(b^{d/2})$ en lugar de $O(b^d)$ como en el minimax, tenemos que para un mismo tiempo, podremos explorar hasta el doble de profundidad.

Tabla 3. Tabla de costes para la función de minimax aplicada al tres en raya

Tablero	Tiempo de ejecución (s)	Nodos expandidos
`[['*', '*', '*'], ['*', '*', '*''], ['*', '*', '_']]`	0,031	1
`[['*', '*', '*'], ['*', '*', '*'], ['*', '_', '_']]`	0,028	4
`[['*', '*', '*''], ['*', '*', '*'], ['_', '_', '_']]`	0,029	15
`[['*', '*', '*'], ['*', '*', '_'], ['_', '_', '_']]`	0,03	64
`[['*', '*', '*'], ['*', '_', '_'], ['_', '_', '_']]`	0,04	325
`[['*', '*', '*'], ['_', '_', '_'], ['_', '_', '_']]`	0,134	1.884
`[['*', '*', '_'], ['_', '_', '_'], ['_', '_', '_']]`	0,428	12.043
`[['*', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]`	2,2995	84.040
`[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]`	19,139	549.945

Las posiciones ocupadas se han indicado con un asterisco (para evitar que las posiciones llenas provoquen tres en raya).

Procedimiento de cálculo de la tabla

Esta tabla se ha calculado ejecutando con Linux `time python3 <fitxer.py>`, donde el fichero tenía una llamada a `decision_minimax('o', 'x', 'o', tablero)`, para cada uno de los tableros indicados, con un fichero diferente para cada tablero. Para saber la cantidad de nodos expandidos, se ha utilizado una variable global `nodos_expandidos` que se incrementa para cada nodo que se visita:

```
def valor_minimax (toca, pieza_min, pieza_max, tablero):
    global nodos_expandidos
    nodos_expandidos += 1
    f_aux_max = lambda th: [valor_minimax(pieza_max, pieza_min, pieza_max, th), th]
    f_aux_min = lambda th: [valor_minimax(pieza_min, pieza_min, pieza_max, th), th]
    ...
```

5.3. Juegos con elementos de azar

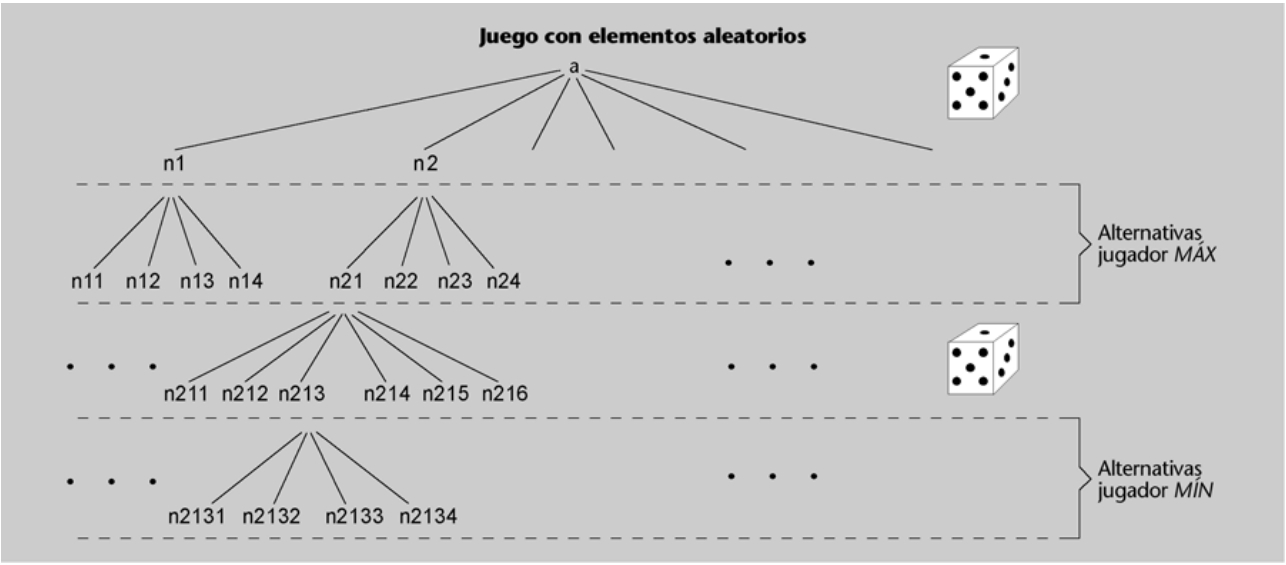
Cuando en una partida se tiene que considerar un elemento aleatorio (por ejemplo, un dado), el desarrollo del árbol de búsqueda ha de tener en cuenta dicho elemento. Así, cada vez que le toca mover a un jugador, tendremos que considerar las diversas posibilidades del elemento aleatorio y, para cada una de estas, desarrollar el árbol correspondiente. Una vez se dispone de las evaluaciones de todos los hijos, la evaluación del nodo se consigue haciendo la media de las evaluaciones de los hijos teniendo en cuenta la probabilidad de

cada componente aleatorio. Así, pues, el desarrollo del árbol de búsqueda corresponderá a una sucesión de nodos correspondientes a mínimos, elemento aleatorio, máximo, elemento aleatorio, etc.

Ejemplo de juego con elemento aleatorio

Un ejemplo de este proceso se muestra en la figura 35. Consideramos un problema en el que antes de mover, un jugador tiene que tirar un dado y según lo que sale, se podrá hacer un movimiento u otro. Estimamos que, en cada posición del juego, se pueden hacer cuatro movimientos. Así, en el caso de encontrarnos en la situación representada en el nodo *a*, tenemos que el dado nos puede ofrecer seis alternativas diferentes, correspondientes a los seis valores que pueden salir. Para cada puntuación del dado, se pueden hacer cuatro movimientos. Así, si en el dado saliera un 1, nos encontraríamos en el nodo n_1, y los cuatro movimientos posibles nos llevan a n_{11}, n_{12}, n_{13} y n_{14}, respectivamente. La figura también muestra los casos en los que en el dado sale 2, 3, ..., 6. Para cada uno de estos nodos se aplicará el mismo proceso.

Figura 35



Para poder hacer la evaluación de los nodos, hemos de tener en cuenta, por un lado, cuando un jugador corresponde al jugador mínimo o al máximo. Pero además tenemos que incluir el elemento de azar. Como se ha dicho, esto se hace definiendo el valor esperado como la media de las evaluaciones de los nodos hijos. Así, si el nodo "a" corresponde al jugador máximo, tendremos que la evaluación de n_1 será el máximo de las evaluaciones de n_{11}, n_{12}, n_{13} y n_{14}; la evaluación de n_2 será el máximo de las evaluaciones de n_{21}, n_{22}, n_{23} y n_{24}; y así sucesivamente para los demás. Usando estas evaluaciones tendremos que la evaluación de "a" será el valor esperado de las evaluaciones de los nodos n_1, n_2, ..., n_6. Dado que la probabilidad de que salga el valor y $\in \{1, 2, 3, 4, 5, 6\}$ cuando se tira el dado siempre es 1/6, la evaluación del nodo *a* será $1/6v(n_1) + 1/6v(n_2) + 1/6v(n_3) + 1/6v(n_4) + 1/6v(n_5) + 1/6v(n_6)$ donde $v(n)$ corresponde a la

evaluación del nodo *n*. Una vez sabemos cómo evaluar un nodo a partir de las evaluaciones de los nodos hijos, el proceso de propagación de las evaluaciones y de selección del mejor movimiento es como en el caso del minimax.

Actividades

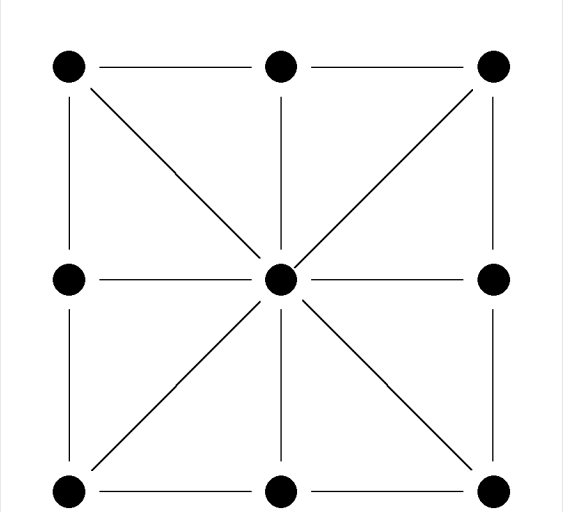
1. Rehaced la búsqueda del coste uniforme en el mismo ejemplo considerado en el subapartado «Búsqueda de coste uniforme», pero ahora permitiendo que el hijo de un nodo tenga el mismo estado que su padre.

2. Utilizando el mapa de carreteras para el problema del camino más corto (subapartado «Búsqueda de coste uniforme») y la tabla de heurísticas (subapartado «Búsqueda con función heurística: búsqueda ávida o voraz»), determinad el camino más corto de Vallmoll a Falset con el algoritmo A*.

3. Considerad el juego de 2 jugadores siguiente: Hay 23 cerillas sobre la mesa y, por turnos, cada uno de los jugadores coge 1, 2, 3, 4 o 5 cerillas. Pierde el que coge la última cerilla (cuando uno coge, como mínimo tiene que quedar 1 sobre la mesa). Pensad cómo haríamos el programa para que dadas *n* cerillas decida cuántas se tienen que coger.

4. Considerad e implementad el juego del tres en raya cuando los jugadores tienen en todo momento 3 piezas y se trata de desplazarlas en el tablero. El tablero es de la forma de la figura y una pieza solo se puede desplazar de una posición (uno de los puntos marcados en el dibujo) a otra si hay un arco que une las dos posiciones.

Tablero para el tres en raya con solo 3 piezas por jugador



Ejercicios de autoevaluación

1. Considerad el problema siguiente:

Dado un laberinto, encontrad la secuencia de movimientos que permite ir de la entrada a la salida. Responded las cuestiones siguientes:

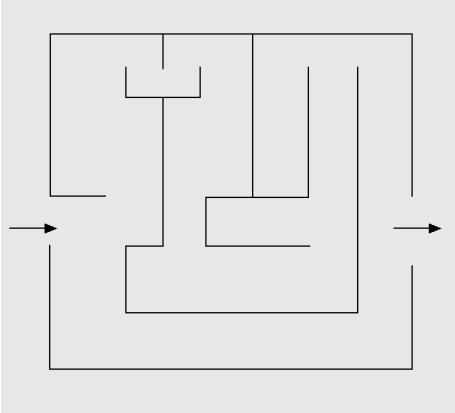
a) Formulad el problema de forma que pueda ser resuelto mediante los métodos de búsqueda.

b) De acuerdo con la formulación, describid el espacio de estados del problema.

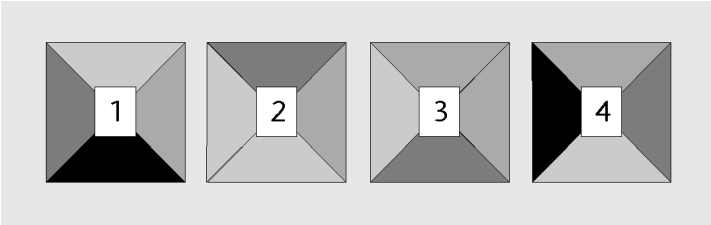
c) Mostrad cómo se aplica esta formulación al laberinto de la figura.

d) ¿Qué métodos de búsqueda podemos utilizar para resolver el problema?

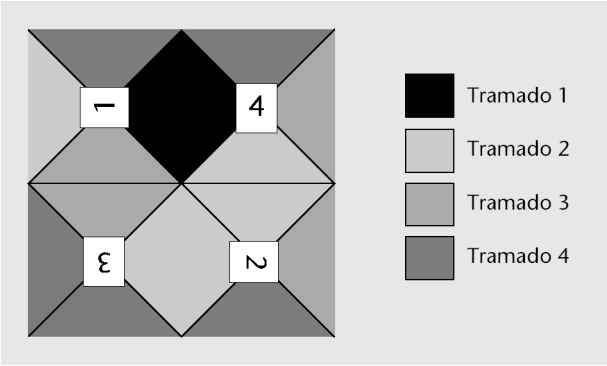
e) Si queremos que el número de posiciones del laberinto atravesadas sea mínimo, ¿podemos aplicar el algoritmo A*? Indicad una función heurística admisible para este problema.



2. Considerad el problema de poner las cuatro piezas mostradas en la figura formando un cuadrado. Se ha de cumplir que las tramas de dos piezas que se tocan tienen que ser iguales.



Una solución a este problema es la de la figura siguiente. Fijaos que en esta solución solo aparecen las piezas que había en la figura de las cuatro piezas. Por tanto, el problema consiste en saber qué pieza se tiene que poner en qué posición y la orientación de la pieza.



Contestad las preguntas siguientes:

a) Formulad el problema de forma que pueda ser resuelto mediante los métodos de búsqueda. ¿Cuál es el factor de ramificación de vuestra formulación?

b) ¿Qué algoritmos de búsqueda podemos aplicar para resolver este problema?

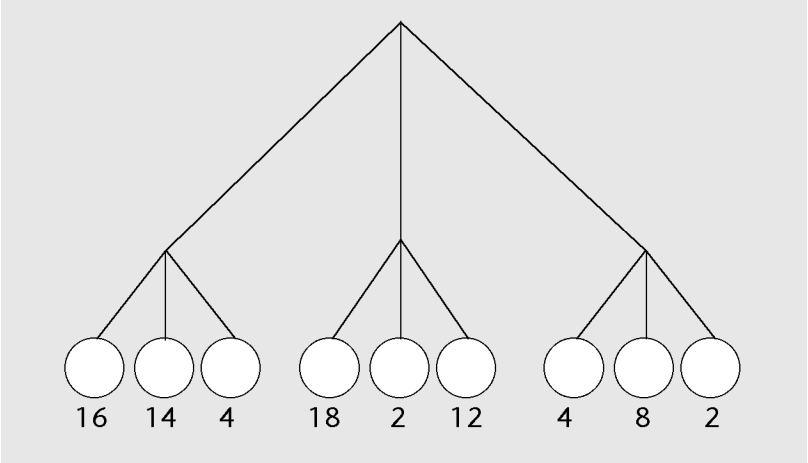
c) Para la representación elegida, ¿hay una solución con el método de búsqueda en anchura? ¿Y con la búsqueda en profundidad? ¿Y con profundidad limitada? ¿Y en búsqueda iterativa con profundidad limitada?

d) ¿Se puede formular el problema como un problema de satisfacción de restricciones?

e) ¿Se pueden aplicar los algoritmos genéticos para resolver este problema?

3. Considerad cómo hacer la poda α-β en el árbol de la figura siguiente. Los números debajo de los nodos del árbol corresponden a la función de evaluación en las hojas. Suponemos que

el nivel más alto del árbol es de nivel MÁX y el nivel inferior es de nivel MÍN. Entonces, ¿qué nodos se podarán?



Solucionario

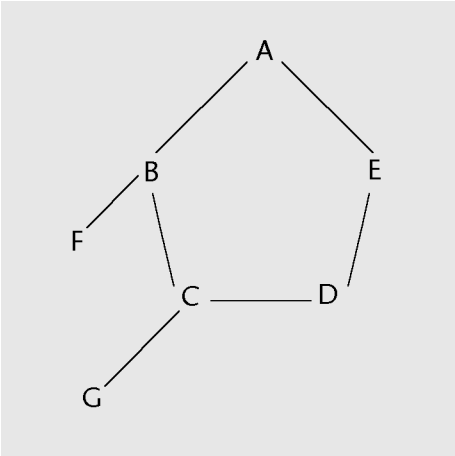
Ejercicios de autoevaluación

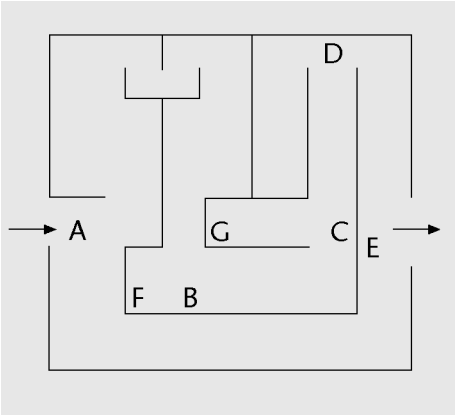
1. Las respuestas a las cuestiones correspondientes al problema del laberinto son estas:

a) Para formular el problema tenemos que modelizar el entorno en el que se mueve el sistema, sus acciones y definir el problema.

- La modelización del entorno se hará representando el laberinto en forma de grafo de forma que a cada cruce (lugar donde hay varios caminos) se le asocia un nodo (también representamos los caminos sin salida con nodo), y las aristas del grafo serán los caminos que conectan dos cruces. Hemos de destacar que con esta representación, una vez hemos elegido un camino, lo tendremos que seguir hasta el próximo cruce. De este modo, solo tenemos los diferentes cruces como estados del problema (ved la figura).
- Las acciones que consideramos son las de tomar un determinado camino que sale de un cruce. Por tanto, en un cruce podremos considerar tantas acciones como caminos salgan. Una acción nos lleva de un estado a otro (de un cruce a otro).
- El problema viene definido por la situación inicial (la entrada al laberinto) y la situación final (la salida del laberinto). A estas posiciones se les asigna un nodo del grafo que representa el problema (si es que no son cruces).

b) La formulación presentada hace que el espacio de estados corresponda al grafo definido más arriba, donde los nodos corresponden a los cruces y las aristas a los caminos entre cruces. Encontrar una solución corresponde a encontrar un camino entre el nodo que representa la entrada y el que representa la salida.





c) En el caso del laberinto, el grafo de estados será el de la figura anterior, donde A, B, ... son los cruces de la figura del laberinto. En la figura siguiente se sobrepone el nombre de los cruces al laberinto.

d) Notad que con la formulación planteada el problema es equivalente al del camino más corto.

Por tanto, para resolver este problema podemos aplicar todos los mecanismos de búsqueda explicados en los apartados «Estrategias de búsqueda no informada» y «Coste y función heurística» de este módulo. Hemos de tener en cuenta que a causa de que puede haber ciclos en el grafo, la búsqueda en profundidad puede dar problemas.

e) Es posible aplicar el algoritmo A* a este problema. Si queremos que el número de posiciones que la solución atraviesa sea mínimo, tenemos que asociar a cada arco del grafo el número de posiciones atravesadas. Este valor será el coste de cada arista. En este problema, una función heurística admisible será, como en el caso del problema del camino más corto, la de la distancia en línea recta a la solución (midiendo la distancia como posiciones atravesadas).

2. Las respuestas a los apartados del segundo ejercicio son estas:

a) Para representar un estado podemos suponer que siempre tenemos las cuatro piezas puestas formando el cuadrado pero que las adyacentes no siempre tienen tramas coincidentes. Además, cada pieza tendrá su orientación. Así, pues, la figura que presenta una posible solución del problema la representaremos diciendo qué pieza está en qué posición repasando las posiciones de izquierda a derecha y de arriba abajo, y cuántas rotaciones de 90 grados ha hecho cada pieza. Tendremos que la situación de las piezas corresponderá a la lista (1 4 3 2) y que las rotaciones de las piezas serán (3 1 2 0). Esto último quiere decir que la pieza 1 ha rotado tres veces, la pieza 2 lo ha hecho una vez, la pieza 3 ha rotado dos veces y la pieza 4 no ha rotado ninguna vez.

A nuestro sistema le permitiremos dos tipos de acciones: (1) intercambiar dos piezas cualesquiera y (2) rotar una pieza 90 grados en el sentido de las agujas del reloj. Así, pues, tendremos seis operaciones de intercambio (intercambiar la posición 1 con la 2 o la 3 o la 4; intercambiar la posición 2 con la 3 o la 4; intercambiar la 3 con la 4) y cuatro rotaciones (podemos rotar la pieza situada en la primera posición, la situada en la segunda, la situada en la tercera y la situada en la cuarta posición). Esto quiere decir que el factor de ramificación será 10.

Para formalizar el problema, podemos considerar el estado inicial como un estado en el que las cuatro piezas tienen una orientación. Por simplicidad, podemos considerar la ubicación (1 2 3 4) y la orientación (0 0 0 0). La función objetivo será comprobar si las tramas adyacentes son iguales.

Tenemos que destacar que se podrían utilizar otras representaciones (por ejemplo, no siempre considerar las cuatro piezas ya puestas formando un cuadrado, sino que unas estarán puestas y otras no) y que estas nos llevarían a otras acciones (por ejemplo, acciones para poner o sacar piezas). De hecho, la representación elegida también permite otros tipos de acciones. En particular, podríamos considerar rotaciones en el sentido contrario a las agujas del reloj.

b) La formulación anterior nos permite aplicar los algoritmos de búsqueda no informada. La definición de una función de coste y una función heurística nos permitirá aplicar algoritmos de búsqueda informada. Una función de coste puede ser el número de movimientos.

c) La búsqueda en anchura encontrará la solución con un menor número de acciones (sean las que sean las acciones permitidas). La búsqueda en profundidad no devolverá ninguna solución porque cada vez aplicará la misma acción (se quedará colgada) (salvo que controlemos los estados repetidos). La búsqueda en profundidad limitada solo encontrará una solución si el límite es mayor que el número de acciones necesarias para construir la solución. La búsqueda iterativa en profundidad también encontrará la solución con menor número de acciones.

d) Sí que se puede formular como un problema de satisfacción de restricciones. Una manera de hacerlo es considerar ocho variables, cuatro correspondientes a las piezas que hay en las cuatro posiciones del tablero (por ejemplo, pos_1, pos_2, pos_3 y pos_4), y cuatro posiciones correspondientes a la rotación de cada pieza ($rotp_1$, $rotp_2$, $rotp_3$ y $rotp_4$). Así, la solución de la figura que presenta una solución del problema se puede representar como: $pos_1 = 1$, $pos_2 = 4$, $pos_3 = 3$ y $pos_4 = 2$; $rotp_1 = 3$, $rotp_2 = 1$, $rotp_3 = 2$, $rotp_4 = 0$. Las restricciones serán que una pieza no puede estar en dos posiciones a la vez ($pos_i \neq pos_j$ cuando $i \neq j$) y que los valores posibles para las variables pos_i son 1, 2, 3 y 4 y los de las variables $rotp_i$ son 0, 1, 2 y 3. Además, se tienen que considerar restricciones de las variables para que las tramas adyacentes sean iguales.

3. En el hijo de la izquierda no se podrá podar nada. La evaluación de sus tres hijos (los nodos que cuando se evalúan dan 16, 14 y 4) dará como resultado el valor más pequeño de todos. Así que retornará 4. Esta elección se hace teniendo en cuenta que el nodo es de MÍN.

Cuando consideramos el segundo hijo, tendremos que el procedimiento podará cuando al evaluar el segundo nodo el resultado será 2, por tanto menor que el valor de 4 que habíamos obtenido para el primer hijo.

El tercer hijo no podrá podar porque hasta que no encuentre un valor menor que el 4 del primer hijo, no puede podar. El tercer hijo del tercer hijo (el que tiene un valor de 2) permitiría podar si hubiera más hijos después del último nodo, pero como este no es el caso, no se podará nada.

Glosario

admisibilidad *f* Dicho de la función heurística que es admisible si nunca sobrestima el coste.

árbol de búsqueda *m* Representación en forma de árbol del proceso de búsqueda en un espacio de estados.

espacio de estados *m* Conjunto de estados posibles y sus relaciones.

estado *m* Situaciones posibles en las que se puede encontrar un sistema.

expansión de un nodo *m* Aplicación del estado asociado a un nodo a todos los operadores de que disponemos.

factor de ramificación *m* Número de acciones que se pueden aplicar a los estados de un problema.

frontera *f* Conjunto de nodos de un árbol de búsqueda que aún no se han expandido.

función heurística *f* Función que, aplicada a un nodo, estima el coste del mejor camino entre este nodo y un estado solución.

Bibliografía

Bibliografía básica

Russell, S.; Norvig, P. (1995). *Artificial Intelligence: A modern approach*. Prentice-Hall.

Bibliografía complementaria

Pearl, J. (1984). *Heuristics*. Addison-Wesley.

Anexo

El código de este módulo está hecho con Python y lo hemos intentado simplificar al máximo. Hemos usado las listas de Python como estructura de datos fundamental y, para poder trabajar con estas listas de una manera todavía más simple, hemos reproducido la manera de construir y acceder a las listas característica del lenguaje de programación Lisp.

Si `L` es una lista, la función `car(L)` retorna el primer elemento de la lista `L` (`L[0]`) o la lista vacía `[]` si `L` está vacía. La función `cdr(L)` retorna una lista igual que `L`, pero sin el primer elemento (`L[1:]`, retorna la lista vacía si `L` tiene un solo elemento o si está vacía). Construiremos las listas con la función `cons(x,L)` que con Python se puede expresar como `[x] + L`.

```
Si L = [1, 2, 3, 4, 5]
    car(L) = 1
    cdr(L) = [2, 3, 4, 5]
y, en general, se verifica que:
    cons(car(L), cdr(L)) = L
```

También hemos añadido algunas funciones auxiliares, como `gensym`, que genera identificadores (*strings*) con un componente aleatorio para garantizar que no han aparecido antes, y funciones de orden superior, que aplican funciones a listas o a elementos de listas. Son fáciles de entender y pensamos que el código es suficientemente autocontenido.

```
import sys
import random
sys.setrecursionlimit(1000000)
# No es lo mismo que el gensym de Lisp, pero es más que
# suficiente para lo que necesitamos
def gensym ():
    return 'symb' + str(int(10000000*random.random())).rjust(7,'0')

def car(lst): return ([] if not lst else lst[0])

def cdr(lst): return ([] if not lst else lst[1:])

def caar(lst): return car(car(lst))

def cadr(lst): return car(cdr(lst))
```

```
def cdar(lst): return cdr(caro(lst))



def cddr(lst): return cdr(cdr(lst))

def caddr(lst): return car(cdr(cdr(lst)))

def cdddr(lst): return cdr(cdr(cdr(lst)))

def caadr(lst): return car(car(cdr(lst)))

def cadadr(lst): return car(cdr(car(cdr(lst))))

def cadddr(lst): return car(cdr(cdr(cdr(lst))))

# Este cons no hace lo mismo que el cons de Lisp, pero ya nos va bien
def cons(elem, lst):
    tmp = lst.copy()
    tmp.insert(0,elem)
    return tmp

def member_if (prd, lst):
    ll = lst.copy()
    leng = len(lst)
    while (leng > 0):
        elem = ll[0]
        if prd(elem):
            return ll
        ll.pop(0)
        leng -= 1
    return []

def find_if (prd, lst):
    for elem in lst:
        if prd(elem):
            return elem
    return []

def remove_if (prd, lst):
    results = []
    for elem in lst:
        if not prd(elem):
            results.append(elem)
    return results

def mapcar (f, lst):
```

```
    return list(map(f,lst))
```