Inteligencia Artificial e Ingeniería del Conocimiento I

BÚSQUEDA

Julio J. Rubio García

Área de Ciencias de la Computación e Inteligencia Artificial

Pedro R. Muro Medrano

Jose A. Bañares Bañares

Área de Lenguajes y Sistemas Informáticos

v 1.0

Borrador del 21 de mayo de 1997

Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

TEMA 1 Primera aproximación a Common Lisp

1.1. Introducción.

Common Lisp, como el resto de lenguajes de la familia Lisp, tiene la característica de ser un lenguaje de programación interactivo. Es decir, el modo habitual de trabajar en Common Lisp consiste en introducir (desde el teclado o bien por lectura de un fichero) una expresión, que es procesada (evaluada, según la terminología Lisp) y por último se procede a la escritura del resultado (en la pantalla o en otro fichero), quedando a continuación el sistema a la espera de que se le facilite una nueva expresión. En el modo más simple de trabajo se utilizan como dispositivos estándar de entrada y salida, respectivamente, el teclado y el monitor. En este caso las primeras sesiones de trabajo en un entorno Common Lisp pueden recordar al manejo de una calculadora de bolsillo o de un sistema operativo como D.O.S. o Unix, puesto que todos ellos están basados en un bucle de lectura-evaluación-escritura.

Veamos algunos ejemplos de uso de un entorno Common Lisp. En estos ejemplos el símbolo > es utilizado como "prompt" del entorno (puede variar según las implementaciones), el texto escrito a continuación debe ser entendido como una expresión tecleada por el usuario y el símbolo ===> habitualmente no aparece explícitamente en los entornos y representa aquí el proceso de evaluación. Por último, lo escrito a la derecha de ===> es lo que el sistema Lisp mostraría en la fase de escritura del bucle de lectura-evaluación-escritura. Un primer ejemplo:

```
> 3  ===> 3
```

Si tecleamos un número (una secuencia de dígitos) el sistema lo escribe tal cual; dicho de otro modo: la evaluación de un número produce el mismo número. Otro ejemplo:

```
> (* 3 3) ===> 9
```

Este ejemplo admite una explicación simple (el producto de 3 por 3 es 9), pero contiene la mayor parte de los elementos esenciales de Lisp. Así en él aparecen, además de los números, un símbolo (`*`) y una lista (un paréntesis que se abre seguido por una secuencia de elementos Lisp, separados por espacios en blanco, y un paréntesis que se cierra). Además se puede observar que la notación en Lisp es prefija: en primer lugar aparece el operador (`*`) y a continuación los argumentos. El siguiente ejemplo nos permitirá explicar un poco mejor como ha funcionado el *evaluador* (la parte del sistema Lisp que realiza la evaluación) en este caso.

```
> (+ (* 3 3) (* 5 5))     ===> 34
```

En este ejemplo la expresión introducida por el usuario es una lista compuesta por tres elementos Lisp: el primero es un símbolo (el operador `+`) y el segundo y el tercero son a su vez listas. Esta es una estructura típica en Lisp (la de listas anidadas) que conlleva una multiplicación del número de paréntesis, una de las características externas más llamativas de Lisp. Ante una expresión como la anterior, el evaluador, tras verificar que el primer elemento de la lista es un operador conocido, procede a evaluar secuencialmente el resto de elementos de la lista. En nuestro caso, la evaluación de `(* 3 3)` produce `9` y la de `(* 5 5)` produce `25`. A los resultados así obtenidos se les aplica el operador de la lista inicial (`+`) y se obtiene el resultado final, `34`. Obsérvese que el evaluador es recursivo: se llama a sí mismo para evaluar los "argumentos". En cambio, el bucle de lectura-evaluación-escritura no es recursivo. En particular, nótese que los resultados de la evaluación de `(* 3 3)` de `(* 5 5)` no han sido escritos en la pantalla. La evaluación de esas expresiones (y por tanto la dada en el segundo ejemplo) responde exactamente al mismo patrón explicado para la entrada más compleja: se comprueba que el primer elemento de la lista es un operador (`*`) y se procede a evaluar los argumentos; pero, como ya hemos indicado, un número es evaluado a sí mismo, con lo que reencontramos la explicación simple dada más arriba (el producto de 3 por 3 es 9).

El ejemplo anterior recoge ya la mayor parte de la sintaxis y de la descripción del comportamiento de los programas Lisp. Pero si tuviesemos que limitarnos a utilizar los operadores predefinidos del lenguaje tendríamos una capacidad expresiva no mucho mayor que la de una calculadora (sofisticada) de bolsillo. Sin embargo, Lisp permite la definición de funciones de usuario que van a facilitar un desarrollo por refinamientos sucesivos de programas complejos y van a ser la herramienta fundamental de organización del software en Lisp. Veamos un ejemplo:

```
> (defun cuadrado (x)
      (* x x))            ===> CUADRADO
```

La primitiva `DEFUN` de Common Lisp va a permitir ligar un cierto símbolo (`cuadrado` en el ejemplo) con una definición de función (en este caso, la función que a `x` le hace corresponder `(* x x)`). La sintaxis es fácilmente deducible del ejemplo: se trata de una lista en la que `defun` es seguido de un símbolo (el nombre de una función), de una lista de símbolos (los nombres de los parámetros formales) y de una expresión Lisp cualquiera (el cuerpo de la función). Obsérvese el formato en que ha sido escrita la expresión de entrada: el cuerpo de la función aparece en una línea diferente. En Lisp, los elementos de una lista pueden estar separados por cualquier número de espacios en blanco o de saltos de línea. Pero la forma de escribir la definición de esa función no es casual: hay unas reglas de escritura del código y de sangrado (indentación) que permiten hacer los programas más fácilmente legibles. Durante el proceso de evaluación de la expresión anterior tiene lugar algo que no aparecía en los ejemplos anteriores: se produce un *efecto lateral*. Un efecto lateral es algo que acontece durante el proceso de evaluación y que es distinto de la construcción del valor a devolver (el valor que será escrito en la fase de escritura del bucle de lectura-evaluación-escritura). En el caso de `defun` el efecto lateral obtenido es la ligadura del nombre de la función con la función en sí, ligadura que altera el contexto de modo que el nombre podrá ser utilizado en las siguientes expresiones del usuario (hasta que termine la sesión). Habitualmente `defun` es utilizado para conseguir ese efecto lateral. Pero las normas generales de Lisp deben ser respetadas, por lo que la evaluación de la expresión debe producir un valor que será a continuación escrito. El valor que se devuelve es el nombre de la función, como queda recogido en el ejemplo. Nótese que el entorno puede decidir escribir el símbolo nombre en mayúsculas. En Lisp los símbolos pueden ser escritos (por el usuario o por el sistema) indistintamente con mayúsculas o minúsculas (como los identificadores en Pascal, por ejemplo, pero no como sucede en C).

Tras evaluar la expresión anterior, el usuario puede utilizar el símbolo `cuadrado` como si se tratase de un operador predefinido, como puede comprobarse en los siguientes ejemplos:

```
> (cuadrado 3)       ===> 9
> (+ (cuadrado 3) (cuadrado 5))    ===> 34
```

1.2. Primera visita a la familia de elementos Lisp y al evaluador.

Después de la anterior introducción informal, podemos resumir lo que sabemos del entorno Lisp.

Los elementos Lisp que conocemos se agrupan del siguiente modo:

- Atomos, que a su vez se dividen en:
  - Números
  - Símbolos
- Listas

Sin entrar en detalles léxicos, los números son secuencias de dígitos (también se pueden manejar en Lisp números reales, racionales y complejos, con los formatos correspondientes) y los símbolos (su apariencia externa, más bien) son secuencias de letras y dígitos (en los que la diferencia entre mayúsculas y minúsculas no es tenida en cuenta). Es importante no confundir los símbolos con las cadenas de caracteres (estructuras de datos que serán explicadas más adelante). Las listas se definen del siguiente modo: un paréntesis que se abre, seguido de un número cualquiera de elementos Lisp separados por los delimitadores Lisp (recuérdese: cualquier número de espacios en blanco o saltos de línea) y un paréntesis que se cierra. Es importante hacer notar que la definición de las listas es recursiva, lo que permite el anidamiento con cualquier grado de profundidad de listas (ya hemos visto ejemplos de estas estructuras anidadas más arriba). Este hecho hace que el tratamiento recursivo sea en Lisp más cómodo y más frecuentemente utilizado que en otros lenguajes de programación.

Entender Lisp es equivalente a comprender como funciona el evaluador. Para ello vamos a retomar como se comporta el evaluador ante cada una de las familias de elementos introducidas antes.

- Atomos:
  - Números: son evaluados a sí mismos. Ejemplo:

```
> 4  ===> 4
```

  - Símbolos: se comprueba si tienen algún valor asociado; si no tienen ningún valor, se produce un error:

```
> a  ===> ¡¡¡¡ ERROR !!!!
```

Aquí hemos supuesto que los ejemplos que vamos escribiendo son las únicas expresiones introducidas en una misma sesión, desde que se ha entrado a ella. Lo que sucede tras haberse producido un error durante el bucle de lectura-evaluación-

escritura depende del entorno concreto con el que estemos trabajando. Habitualmente, se continúa en el bucle de lectura-evaluación-escritura o existe algún modo sencillo de volver a él.

Si un símbolo tiene un valor asociado, este valor es el resultado de la evaluación. Para encontrar una muestra de esta situación entre los ejemplos anteriores, hay que mirar con un poco más de detalle el proceso de evaluación de una expresión en la que aparezca una función definida por el usuario. Así, tras la evaluación de:

```
> (defun cuadrado (x)
      (* x x))  ===> CUADRADO
```

si introducimos:

```
> (cuadrado 3) ===> 9
```

en el proceso (interno) de evaluación de esa expresión, se produce la ligadura del parámetro formal `x` con el (resultado de evaluar el) argumento `3`, de modo que cuando se evalúa el cuerpo de la función `(* x x)`, la evaluación de `x` (en cada ocasión), produce el valor `3`, obteniéndose finalmente el comportamiento observado (escritura de `9`). Es importante resaltar que la ligadura del parámetro formal `x` con el parámetro real `3` es *local* al cuerpo de la función, de modo que su evaluación fuera de ese contexto producirá un error:

```
> x  ===> ¡¡¡¡ ERROR !!!!
```

Por supuesto, también existen modos de ligar valores a símbolos de modo global. Pero este aspecto será tratado más adelante.

Continuamos explicando el modo de evaluación del resto de elementos Lisp que conocemos: las listas. En cierto modo, aquí se recoge el núcleo de la programación Lisp, puesto que en Lisp definir un programa es escribir una lista (una definición de función que, habitualmente, se apoya en muchas otras) y ejecutar un programa no es más que evaluar una lista (una llamada a una función con ciertos argumentos).

– Listas: se comprueba que el primer miembro es un símbolo que tiene asociado un operador (predefinido o definido previamente por el usuario) y si esto no es cierto (bien porque el primer miembro no sea un símbolo, bien porque éste no tenga asociada una función) se produce un error:

```
> (5 3)          ===> ¡¡¡¡ ERROR !!!!
> (cuadrad 3)    ===> ¡¡¡¡ ERROR !!!!
```

En otro caso, se procede a evaluar secuencialmente de izquierda a derecha el resto de miembros de la lista, siguiendo (recursivamente) las mismas reglas que estamos dando; si en algún momento se produce un error, el proceso es detenido y el usuario es informado de que se ha producido un error. En otro caso (es decir, si cada uno de los miembros de la lista han sido evaluados con éxito), se procede a aplicar el operador que aparecía en cabeza (primer lugar) de la lista a los valores obtenidos en el proceso de evaluación anterior. Aquí pueden volver a aparecer nuevos errores, por ejemplo debidos a que el número de argumentos o el tipo de los argumentos definitivos (es decir, tras su evaluación) no sean los adecuados. Si no se produce ningún error adicional, se alcanza un elemento lisp que será el resultado de la evaluación de la lista inicial. La situación puede ser explicada con más detalle en el caso de que el operador haya sido definido por el usuario (si es predefinido, el manual del lenguaje es el que nos indicará el comportamiento del operador), en la línea que ha sido esbozada más arriba cuando se ha hablado de la evaluación de símbolos con ligaduras (locales), pero por el momento nos contentamos con dar esta información somera.

Esta es la regla general, la que se aplica por defecto, para la evaluación de listas. Pero ya conocemos un ejemplo de evaluación de una lista que no ha seguido el mismo proceso. Así cuando explicamos el ejemplo:

```
> (defun cuadrado (x)
     (* x x))
===> CUADRADO
```

no se produjo ningún error, mientras que si se hubiese seguido el procedimiento general se hubiese producido un error al intentar evaluar el símbolo `cuadrado` que no tiene asociado ningún valor. Esto es así porque una lista encabezada por `defun` es tratada de un modo especial por el evaluador. En particular, los argumentos provisionales de `defun` no serán evaluados. Un símbolo como `defun`, que tiene asociado un operador predefinido y tal que si aparece en cabeza de una lista las reglas generales de evaluación de listas no son aplicadas sobre ella, se denomina *elemento especial*. Cada elemento especial requiere una información adicional sobre cómo se evalúan las listas en las que aparece como primer miembro.

Otra observación que puede ser extraída de las explicaciones anteriores es que, de hecho, los símbolos son tratados por el evaluador de modo diferente según

aparezcan como primer miembro de una lista o en cualquier otra situación (dentro o fuera de una lista). En el primer caso, el evaluador busca si el símbolo tiene asociada una función, con el objeto de aplicarla a (los valores obtenidos por evaluación de) sus argumentos. En el segundo, intenta encontrar el valor que tiene ligado en el contexto actual. Ambos procesos son totalmente independientes. Por ejemplo:

```
> cuadrado      ===> ¡¡¡¡ ERROR !!!!
```

Esta consideración permite ver que un símbolo es en Lisp un objeto muy rico (mucho más que un identificador en lenguajes como C o Pascal) en el que hemos podido distinguir, hasta el momento, tres componentes:

- su nombre (la secuencia de caracteres que nos permite referirnos a él)
- su valor (como en el caso de la ligadura local de `x` y `3`)
- su valor funcional (que puede ser creado o modificado usando `defun`)

Prácticamente ya conocemos todos los elementos básicos que nos van a permitir programar en Lisp. Al ser tan reducido el número de herramientas con las que contamos, va a ser obligatorio que algunos de estos elementos puedan ser utilizados para varios fines. Esto es lo que sucede con los símbolos y las listas. Los símbolos pueden ser utilizados como:

- Identificadores: para nominar funciones (como `cuadrado`), parámetros formales (como `x` en la definición de `cuadrado`) o variables (todavía no tenemos ningún ejemplo de este último uso).
- Datos, como veremos en breve.

Las listas a su vez pueden ser utilizadas como:

- "Programas", es decir expresiones que pueden ser evaluadas, como en la mayoría de los ejemplos anteriores.
- Delimitadores sintácticos, como en el caso de las listas de parámetros formales.
- Estructuras de datos, como veremos al final del tema.

Vamos a continuación a explicar cómo es posible ligar, de modo global un símbolo y un elemento Lisp. Veamos algunos ejemplos:

```
> (setq x 5)    ===> 5
> x             ===> 5
> (+ x 8)       ===> 13
```

El operador predefinido `setq` es un elemento especial. Su primer argumento *debe* ser un símbolo (en otro caso se produce un error) que no será evaluado y su segundo argumento es cualquier elemento Lisp, que será evaluado. El resultado de evaluar este segundo argumento será el resultado de la evaluación de la lista inicial y además, *como efecto lateral*, dicho resultado será ligado en el contexto activo en ese momento (el global en el ejemplo) al símbolo que era el primer argumento del `setq`. Algunos ejemplos más:

```
> (setq y  (cuadrado (+ 1 2)))     ===> 9
> (+ (cuadrado x) y)                ===> 34
```

Esta primitiva nos permite manipular los símbolos como variables (su tercer uso como identificadores enumerado más arriba). Sin embargo, para utilizar los símbolos como datos en sí todavía nos faltan recursos. Por ejemplo si quisiésemos ligar al símbolo `z` el símbolo `caballo` no podríamos teclear algo como:

```
> (setq z caballo)  ===> ¡¡¡¡ ERROR !!!!
```

La razón es que, según se ha explicado más arriba, el evaluador intentará encontrar el valor asociado al símbolo `caballo` lo que producirá un error. Por ello, si queremos utilizar un símbolo como un dato, es obligatorio que podamos *impedir la evaluación*. Esto es posible gracias al elemento especial `quote` que no evalúa su único argumento, que puede ser cualquier elemento Lisp. Así:

```
> (quote caballo)   ===> CABALLO
```

Lo que utilizado en conjunción con `setq` permite ya manipular símbolos como datos:

```
> (setq z (quote caballo))    ===> CABALLO
> z  ===> CABALLO
```

Puesto que `quote` es una primitiva muy utilizada (Lisp es un lenguaje muy adecuado para la manipulación simbólica, lo que no es ajeno a su amplio uso en Inteligencia Artificial), los diseñadores de Lisp han previsto una abreviatura, la comilla simple `'`. Así:

```
> 'caballo                ===> CABALLO
> (setq pieza 'alfil)     ===> ALFIL
> pieza                   ===> ALFIL
```

Obsérvese en cambio:

```
> 'pieza  ===> PIEZA
```

1.3. Condicionales, recursividad y listas como estructuras de datos.

Tras la primera visita al evaluador vamos a terminar el tema con una breve introducción a la programación en Lisp. Para ello en primer lugar necesitaremos una primitiva de selección condicional (un `if`) y esto a su vez requiere de la noción de predicado: función que devuelve un valor booleano. Así pues debemos comenzar por explicar cuáles son los valores booleanos en Lisp. El valor "falso" es representado en lisp por el símbolo `nil`. Este símbolo es una constante predefinida cuyo valor es el mismo símbolo `nil`:

```
> nil      ===> NIL
```

El valor booleano "cierto" *por defecto* es el símbolo `t` (por "true") que también se comporta respecto al evaluador como una constante de valor ella misma:

```
> t  ===> T
```

Hemos indicado que `t` es el valor "cierto" por defecto debido a que en Lisp cualquier elemento distinto de `nil` es considerado como "cierto" (es una situación similar a la que se produce en C, donde el entero 0 es el valor booleano "falso", mientras que cualquier número distinto de cero es considerado "cierto", aunque el valor "cierto" *por defecto* es 1). Eso significa que el valor `t` es sólamente elegido por Lisp para ser devuelto cuando ningún otro valor distinto de `nil` es significativo en ese contexto. Para terminar con esta breve introducción a los valores booleanos en Lisp, es necesario observar que el símbolo `nil` tiene un sinónimo: la lista vacía `()`. Es decir, el símbolo `nil` y la lista vacía `()` son elementos indistinguibles en Lisp. Se trata de un solo elemento que es el único que es simultáneamente un átomo (de hecho, un símbolo) y una lista. Esto conlleva varias consecuencias. Así la regla de evaluación de listas se generaliza del siguiente modo para la lista sin ningún miembro:

```
> () ===> NIL
```

Y la afirmación de que cualquier elemento distinto de `nil` en Lisp representa el valor booleano "cierto" no nos debe hacer pensar que toda lista en Lisp representa el valor "cierto"; esto sólo es verdad para las listas que tienen algún miembro, las que no son la lista vacía.

El Lisp existen muchos predicados predefinidos que permiten conocer propiedades o comparar elementos. Veamos algunos:

```
> (equal 'a 'b)      ===> NIL
> (equal 'a 'a)      ===> T
```

La función predefinida `equal` sirve para determinar si dos elementos Lisp son iguales. Existe otro operador de comparación de la igualdad que sólo puede ser utilizado con números:

```
> (= 3 8)                 ===> NIL
> (= 9 (cuadrado 3))      ===> T
```

Aunque en Lisp no hay un "tipado explícito" de los datos y variables, sí que existe un "tipado implícito" que consiste en que ciertos operadores predefinidos producen error si sus argumentos (definitivos: tras la evaluación) no son del tipo adecuado. Así:

```
> (= 9 'y)      ===> ¡¡¡¡ ERROR !!!!
> (= 9 y)       ===> T
```

ya que, si continuamos en la misma sesión desde el comienzo del tema,

```
> y  ===> 9
```

Lo mismo sucede con otros operadores numéricos, ya sean de comparación (`>`, `<`, `>=`, `<=`) o aritméticos (`+`, `*`, `-`, ...).

Una vez que conocemos ciertos predicados, podemos pasar a considerar la primitiva condicional más elemental en Lisp: `if`. Veamos algunos ejemplos:

```
> (setq x 7)   ===> 7
> (if (< x 5)
      'menor_que_cinco
      'mayor_o_igual_que_cinco)
===> MAYOR_O_IGUAL_QUE_CINCO
> (setq x 3)   ===> 3
> (if (< x 5)
      'menor_que_cinco
      'mayor_o_igual_que_cinco)
===> MENOR_QUE_CINCO
```

El condicional `if` es un elemento especial. Tiene tres argumentos, de los cuales el primero (la condición) será siempre evaluado; su valor tendrá una interpretación booleana. Si dicho valor es "cierto" (es decir, distinto de `nil`, la lista vacía), el segundo argumento del `if` (la rama "entonces") será evaluado y su valor será devuelto como valor del `if`; el tercer argumento del `if` no será evaluado. Si el valor

del primer argumento es `nil` (la lista vacía), el segundo argumento no será evaluado, pero sí lo será el tercero (la rama "si no"), cuyo valor será el del `if`.

Esto nos permite programar una función que calcule el máximo de dos números enteros:

```
> (defun mayor (x1 x2)
     ; x1 y x2 deberán ser números enteros
     (if  (< x1 x2)
          x2
          x1))
===> MAYOR
> (mayor 2 3)  ===> 3
> (mayor (* 2 7)  3)     ===> 14
> (mayor 6 'numero) ===> ¡¡¡¡ ERROR !!!!
```

En Lisp los comentarios (aquellas partes del texto que no serán tomadas en consideración por el evaluador) se indican mediante `;`, el punto y coma. Cuando aparece en una línea el carácter reservado `;` se entiende que el texto que aparece tras él en esa línea es un comentario. En el ejemplo anterior se ha escrito como comentario la especificación de entrada de la función de `mayor`: si sus argumentos (definitivos) no son números enteros su comportamiento está indefinido. En particular, si se introduce un símbolo como argumento (como en la última línea del ejemplo) se producirá un error.

Con estos pocos preliminares, podemos ya escribir programas que realicen cálculos repetitivos. Para ello nos apoyamos en la recursividad, que en Lisp puede ser comprendida sin más que aplicar las reglas generales de evaluación explicadas más arriba. Por ejemplo, la siguiente función permite calcular el factorial de un número entero positivo:

```
> (defun factorial (n)
     ; n deberá ser un número entero positivo
     (if  (= n 0)
          1
          (* n (factorial (- n 1)))))   ===> FACTORIAL
> (factorial 4)
===> 24
```

Para terminar esta brevísima introducción a la programación en Common Lisp, vamos a mostrar cómo las listas pueden ser empleadas como estructuras de datos en Lisp. Por ejemplo, si deseamos almacenar una colección de números enteros en Pascal o C emplearíamos una estructura de la familia "vector" (un "array"). En Lisp esto puede ser conseguido utilizando una lista. Para tratar con una lista de enteros es necesario impedir su evaluación:

```
> (1 2 3) ===> ¡¡¡¡ ERROR !!!!
> '(1 2 3)     ===> (1 2 3)
```

Ahora necesitamos operadores predefinidos que nos permitan acceder a los elementos de una lista. El operador `first` extrae el primer miembro de una lista, mientras que el operador `rest` devuelve una lista como la inicial, pero sin su primer miembro. Estos dos operadores utilizados combinadamente permiten acceder a cualquier miembro de una lista:

```
> (first '(7 17 -3))                ===> 7
> (rest '(7 17 -3))                 ===> (17 -3)
> (first (rest '(7 17 -3)))         ===> 17
> (first (rest (rest '(7 17 -3)))) ===> -3
```

Obsérvese el siguiente ejemplo de comportamiento de `rest`:

```
> (rest '(-3)) ===> NIL
```

Por último el predicado `endp` nos informa de si una lista es vacía (esto es, igual a `nil`):

```
> (endp '(7 17 -3))      ===> NIL
> (endp '())             ===> T
> (endp ())              ===> T
> (endp 'nil)            ===> T
> (endp nil)             ===> T
> (endp (rest '(-3)))    ===> T
```

Estas pocas herramientas para el tratamiento de listas, utilizadas junto a la recursividad, nos permiten ya implementar algoritmos para el procesamiento de datos estructurados. Por ejemplo, la siguiente función calcula la suma de los elementos de una lista de enteros (se impone que si la lista de entrada es vacía se devuelva 0).

```
> (defun suma-lista-enteros (lista-enteros)
    (if  (endp lista-enteros)
```

```
            0
            (+ (first lista-enteros)
               (suma-lista-enteros (rest lista-enteros)))))
===> SUMA-LISTA-ENTEROS
> (suma-lista-enteros '(7 17 -3))  ===> 21
```

Ejercicios. Se pide escribir funciones Lisp que realicen la tarea que se indica. Para cada una de ellas se deberán escribir unos cuantos ejemplos significativos de llamadas a la función.

1) Diseñar un predicado (función que devuelve valores booleanos) que, a partir de una lista y un elemento Lisp cualquiera, decida si el elemento es `equal` a alguno de los elementos de la lista (si la lista es vacía se entiende que el resultado de la búsqueda debe ser `nil`).

2) Diseñar una función que, a partir de una lista de enteros positivos, calcule el elemento máximo (se impone que si la lista es vacía se devuelva el entero `-1`). Indicación: utilícese la función `mayor` que aparece en el texto más arriba.

3) Diseñar una función que, a partir de una lista no vacía de enteros, calcule el elemento mínimo. Indicación: diséñense dos funciones auxiliares, un predicado que verifique si una lista tiene un único elemento y una función que calcule el mínimo de dos números enteros.

TEMA 2. Sistemas de producción y búsqueda

2.1. Resolución de problemas en Inteligencia Artificial.

En general, podemos afirmar que un problema consiste en:

- una descripción de la situación de la que se parte;
- una descripción de la situación a la que se quiere llegar;
- una descripción de los medios de que disponemos para alcanzar nuestro objetivo.

En el contexto de la Informática, a partir de un problema, se intenta construir un sistema que lo resuelva. Resumidamente, las acciones para construir un sistema que resuelva un problema serían:

- Definir el problema con precisión (especificación del problema), habitualmente a partir de un enunciado del problema expresado en lenguaje natural:
    - de qué se parte;
    - cuál es el objetivo.
- Analizar el problema: información para elegir las técnicas.
- Aislar y representar el conocimiento necesario para resolver el problema.
    - Elegir la mejor técnica que resuelva el problema y aplicarla al problema particular.

Particularizar estas acciones al contexto de la Inteligencia Artificial consiste en elegir las técnicas de resolución entre aquellas que son utilizadas en esta disciplina. En nuestro caso, nos centraremos en los siguientes temas en las técnicas relacionadas con los sistemas de producción y la búsqueda en espacios de estados.

En los temas siguientes utilizaremos como ejemplos para ilustrar las nociones teóricas algunos problemas que no son aplicaciones reales de la Inteligencia

Artifical. Se trata más bien de ejemplos que han sido elegidos porque son suficientemente manejables, tanto en cuanto a la facilidad de su enunciado como en cuanto al volumen de datos que es necesario explorar para obtener una solución.

He aquí algunos de ellos:

- El 8-puzzle.
  - Entrada: Un tablero 3x3 donde aparecen distribuidos los dígitos 1, 2, 3, 4, 5, 6, 7 y 8, quedando por tanto una de las casillas del tablero en blanco. Por ejemplo, un tablero tal y como aparece en la Figura 1.

1	4	3
7		6
5	8	2

Figura 1

  - Salida: Alcanzar una disposición distinta de los dígitos en el tablero. Habitualmente se toma como configuración de llegada el tablero que tiene el dígito 1 en la primera casilla de la primera línea (esquina superior izquierda) y los dígitos 2, 3, …, 8 se distribuyen uno a continuación de otro en el sentido de las agujas del reloj, quedando la posición central del tablero sin ocupar (ver Figura 2)

1	2	3
8		4
7	6	5

Figura 2

  - Medios: si una casilla del tablero está al lado de la casilla vacía se puede desplazar sobre ella. (En particular, está prohibido usar un destornillador para sacar las piezas y resituarlas como a uno le parezca …)

- El problema de las garrafas de vino.
  - Entrada: dos garrafas vacías, una de cuatro litros y otra de tres.
  - Salida: la garrafa de cuatro litros contiene exactamente dos litros de vino.
  - Medios: se dispone de un depósito con mucho vino (más de 100 litros …) y las únicas operaciones permitidas son llenar cada garrafa en el depósito, vaciarlas en el depósito y pasar contenido de una garrafa a la otra, hasta

que la primera se vacíe o la segunda se llene. (En particular, no es lícito suponer que se dispone de otra garrafa que puede contener exactamente dos litros …)

- El problema del granjero, el lobo, el carnero y la lechuga.
  - Entrada: un granjero, un lobo, un carnero y una lechuga se encuentran en la orilla derecha de un río.
  - Salida: todos ellos han pasado a la orilla izquierda del río.
  - Medios y restricciones: se dispone de una barca que debe ser conducida por el granjero y que sólo tiene capacidad para uno de los otros elementos; el lobo se comerá al carnero si los deja juntos sin compañía en uno de los lados; el carnero se comerá la lechuga si los deja solos.

Ejercicios. A partir de las siguientes descripciones de los problemas que se indican, detallar con mayor precisión la entrada, la salida y los medios de resolución.

1) *El problema del viajante*. Un representante de comercio tiene que visitar una serie de ciudades viajando entre ellas por carretera. Dispone de un plano de la región con la información de las distancias entre las ciudades. Desea partir de la ciudad en la que está para volver a ella tras haber visitado el resto de ciudades, pero habiendo hecho el mínimo número de kilómetros posible.

2) *Las n reinas*. Se trata de situar n reinas (siendo n un número entero positivo mayor que 3) del ajedrez en un tablero n x n (el normal tiene 8 x 8 casillas) de modo que no se "coman" entre sí.

2.2. Sistemas de producción y búsqueda.

Si particularizamos la definición de problema dada en el apartado anterior del siguiente modo:

- Entrada: una descripción del estado inicial del mundo.
- Salida: una descripción (parcial) del estado del mundo deseado.
- Medios: una descripción de acciones que puedan transformar un estado del mundo en otro, acciones que son consideradas como operadores o reglas.

Tendremos el marco conceptual que permite resolver el problema por medio de un sistema de producción.

Un *sistema de producción* consiste en:

- una base de datos/hechos/conocimiento con información sobre el problema;
- un conjunto de reglas (operadores);
- una estrategia de control;
- un aplicador de reglas: ciclo de reconocimiento-actuación.

La parte más importante de los sistemas de producción es la estrategia de control (también llamada, en el contexto de los sistemas expertos, mecanismo o motor de inferencia). Es la que establece el proceso de búsqueda de la solución. Si la descripción del problema viene dada por una noción de estado (del mundo) podemos entender que un sistema de producción es empleado para una búsqueda en un espacio de estados (así se relacionan estos dos paradigmas clásicos de la Inteligencia Artificial). Con esta perspectiva, uno de los cometidos de la estrategia de control consiste en decidir qué operador aplicar en cada momento:

- ¿se puede aplicar?
- ¿produce algún cambio en el estado?
- ¿qué estado elegir para aplicar los operadores?
- ¿qué sucede si hay varios operadores posibles a aplicar? (resolución de conflictos)

Los operadores, que admiten una interpretación como reglas, son acciones simples que pueden transformar un estado en otro. Generalmente se distinguen en los operadores una parte izquierda (el patrón o condición), que determina la aplicabilidad de la regla (se dice también que si se verifica la condición la regla queda *sensibilizada*), y una parte derecha que describe la operación o acción a llevar a cabo al aplicar el operador (o *disparar* la regla). Así, una regla u operador consta de dos partes:

- Precondición: descripción parcial del estado del mundo que debe ser verdad para realizar una acción.
- Instrucciones para crear el nuevo estado.

Taxonomías de los sistemas de producción/búsqueda/estrategia de control:

- Hacia adelante / hacia atrás / bidireccional
- Irrevocable / tentativa
- Informada / no informada

Ventajas de los sistemas de producción:

- Separación de conocimiento (reglas) y control (ciclo de reconocimiento-actuación).
- Modularidad de las reglas/operadores:
    - no hay interacciones sintácticas entre reglas;
    - la comunicación se establece a través de ciertas estructuras de almacenamiento comunes, que se denominan de modo genérico *memoria de trabajo*. (No es estrictamente necesario que exista una variable o estructura de datos explícita con ese nombre.)
- Control dirigido por patrones:
    - programas en IA necesitan flexibilidad;
    - reglas pueden dispararse en cualquier secuencia;
    - estado > reglas aplicables > camino de la solución.
- Traza y explicación:
    - secuencia de aplicación de reglas;
    - cada regla es una pieza de conocimiento con su justificación para un cambio de estado.
- Independencia del lenguaje:
    - el modelo es independiente de la representación elegida para reglas y memoria de trabajo;
    - es posible siempre que el lenguaje soporte el reconocimiento de patrones.

- Modelo plausible del mecanismo humano de resolución de problemas.

Intérpretes generales de sistemas de producción:

- Lenguajes basados en reglas:
    - OPS5 (Brownston, en 1985, de CMU, en Common Lisp usa RETE);
    - CLIPS (C Language Integrated Production System) (de NASA, permite la integración con lenguajes como C y Ada).
- Lenguajes basados en lógica: el más extendido es PROLOG.
- Armazones de sistemas expertos:
    - EMYCIN;
    - en entornos de IA (KEE, KnowledgeCraft, LOOPS, …).
- Arquitecturas generales de resolución de problemas:
    - GPS (Newell, Shaw, Simon) 1963;
    - SOAR (Newell, Laird, …) 1987.

2.3. Representación de problemas.

Una vez decidido que el problema planteado va a ser resuelto por medio de un sistema de producción/búsqueda en el espacio de estados, debemos elegir una representación adecuada para el problema. Esto debe ser resuelto en tres niveles:

- Conceptual.
- Lógico.
- Físico.

y en cada uno de ellos deberemos elegir representaciones para dos cosas:

- Estados.

- Operadores.

El nivel conceptual hace referencia a qué consideramos como estado del problema y qué como regla u operador válido, pero independientemente de cualquier estructura de datos o descripción de algoritmos que vayamos a usar. En el nivel lógico se elige una estructura de datos para los estados y se determina el formato de codificación de los operadores (puede ser también una estructura de datos o se puede hacer proceduralmente). Por último en el nivel físico se concreta para el lenguaje de programación elegido la implementación de estados y operadores determinada en el nivel lógico. Los niveles lógico y físico están confundidos en ocasiones y, además, la fase clave para un procesamiento eficiente está en el nivel conceptual.

Por ejemplo, en el 8-puzzle se podría decir que el estado incluye información sobre el material o el tamaño del tablero, etc. Sin embargo, nada de eso es relevante. Basta con conocer la "situación relativa" de los dígitos.

En general, la noción de estado elegida debería describir:

- todo lo que es necesario para resolver el problema;
- nada que no sea necesario para resolver el problema.

Para el problema del 8-puzzle:

- localización de cada casilla y el hueco;
- nada más.

Obsérvese que, un vez fijado el nivel conceptual, todavía tenemos muchas opciones en el nivel lógico:

- matriz 3 x 3;
- vector de longitud 9;
- conjunto de hechos: {(superior-izda = 2), (superior-centro = 1), …}.

(A su vez, dependiendo de las estructuras permitidas en un lenguaje de programación concreto, para cada una de ellas son posibles diferentes elecciones, pero esto nos interesa menos: es más un problema de programación que de IA.)

En cualquier caso, la elección de la representación para los estados está muy ligada a la que se haga para los operadores, puesto que ambos elementos deben "cooperar" para la resolución del problema.

En cuanto a los operadores, deben ser tan generales como sea posible para así reducir el número de reglas distintas. En el caso del 8-puzzle:

- 4 x 9! operadores para pasar de un estado cualquiera a sus, como máximo 4, estados sucesores;
- 4 x 8 operadores que mueven cualquier ficha arriba, abajo, derecha o izquierda;
- 4 operadores que mueven el hueco arriba, abajo, derecha o izquierda.

Evidentemente, la última representación es la más adecuada.

La elección de una representación para los estados y los operadores define implícitamente una relación de adyacencia en el conjunto de estados factibles para el problema. A este conjunto con la estructura de adyacencia definida se le denomina *espacio de estados*. El espacio de estados es una noción clave en los sistemas de producción, puesto que es el lugar en el que se va a desarrollar la búsqueda, y en el que se van a poder comprender las distintas variantes involucradas (en particular, las taxonomías: hacia adelante/hacia atrás, etc.). Un fragmento del espacio de estados para el 8-puzzle, donde se ha elegido como representación para los operadores los 4 operadores citados antes, puede verse en la Figura 3.

Veamos qué sucede con la representación del problema de las garrafas de vino. Conceptualmente, los únicos datos relevantes en cada momento para el sistema son las cantidades de vino que contienen las garrafas de cuatro y tres litros (puesto que la cantidad de vino que haya en el depósito no influye en la resolución del problema, como ha quedado fijado al explicar los medios de que se dispone en el problema). Los operadores sobre esos estados serían:

- llena-cuatro: llenar, cogiendo del depósito, la garrafa de cuatro litros;
- llena-tres: llenar, cogiendo del depósito, la garrafa de tres litros;
- vacia-cuatro: vaciar en el depósito la garrafa de cuatro litros;
- vacia-tres: vaciar en el depósito la garrafa de tres litros;

- echa-la-cuatro-a-la-tres: echar de la garrafa de cuatro litros en la garrafa de tres;
- echa-la-tres-a-la-cuatro: echar de la garrafa de tres litros en la garrafa de cuatro.

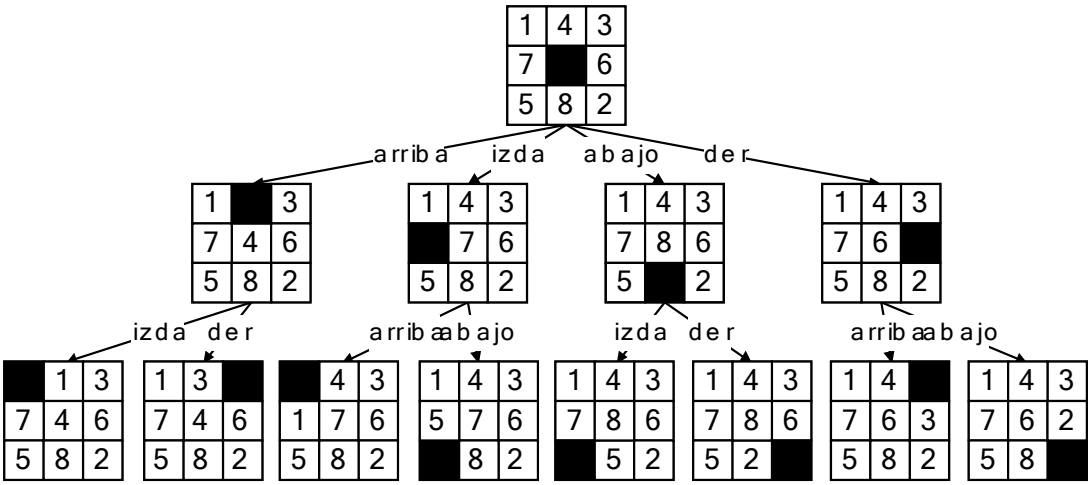


Figura 3

En el nivel lógico, un estado sería identificado como un par de números (x,y) donde, por ejemplo, x es el número de litros que contiene la garrafa de cuatro litros e y el número de litros de la garrafa de tres litros. En este nivel cada uno de los 6 operadores anteriores deberían ser especificado como una regla, con una precondición (o premisa) y una acción asociada (o consecuencia). Por ejemplo:

llena-cuatro (x, y):
precondición: x < 4;
acción: construir el estado (4,y).

Ejercicios.

1) Especificar como reglas el resto de operadores.
2) Dibujar el espacio de estados correspondiente, a partir del estado inicial (0,0) hasta que encontréis un estado objetivo, es decir un estado de la forma (2,y).

Para estudiar el nivel físico, que es el más cercano a la implementación del sistema, vamos a analizar las estructuras de datos en Lisp.

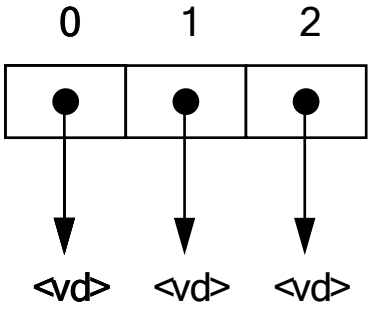
2.4. Estructuras de datos en Lisp: vectores, registros, listas.

2.4.1. Vectores en Lisp.

La función que permite construir vectores en Lisp es `make-array`. Esta función toma un único argumento cuyo valor debe ser un número entero positivo; llamemos n a dicho número. Devuelve como valor un vector con n componentes que contendrán un valor por defecto (que depende del entorno concreto con el que estemos trabajando y que será denotado `<vd>` en las figuras). Atención: los índices de los elementos en el vector comienzan en 0 y terminan en n-1 (como en C; de hecho, todo el tratamiento de vectores en Lisp es mucho más cercano al que se hace en C que al de otros lenguajes como Pascal). Por ejemplo,

```
> (make-array 3)      ===> dependiente del entorno
```

construye un vector que puede ser representado como aparece reflejado en la Figura 4. Nótese que puesto que el vector así construido no ha sido ligado a ningún símbolo, ni pasado como parámetro, es un elemento que no puede volver a ser reutilizado.



valores por defecto en la implementación

Figura 4

Para acceder a las distintas componentes de un vector, Common Lisp proporciona la primitiva `aref`, que toma dos argumentos: el primero debe ser un elemento vector (es decir, creado por `make-array`) y el segundo debe ser un número entero entre 0 y n-1 si n es la dimensión del vector que es el primer argumento. Por ejemplo, si consideramos que el símbolo `v` está ligado al vector que aparece en la Figura 5 tendríamos:

```
> (aref v 0)   ===> A
> (aref v 1)   ===> B
```

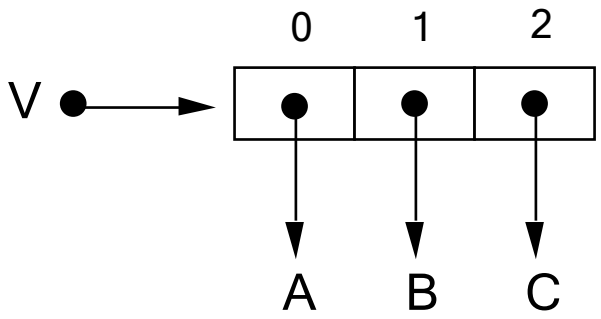


Figura 5

Para modificar el contenido de alguna componente de un vector utilizamos la primitiva `setf` que es como `setq` salvo que su primer argumento no tiene por qué ser un símbolo: puede ser cualquier *variable generalizada* (ver la Figura 6).

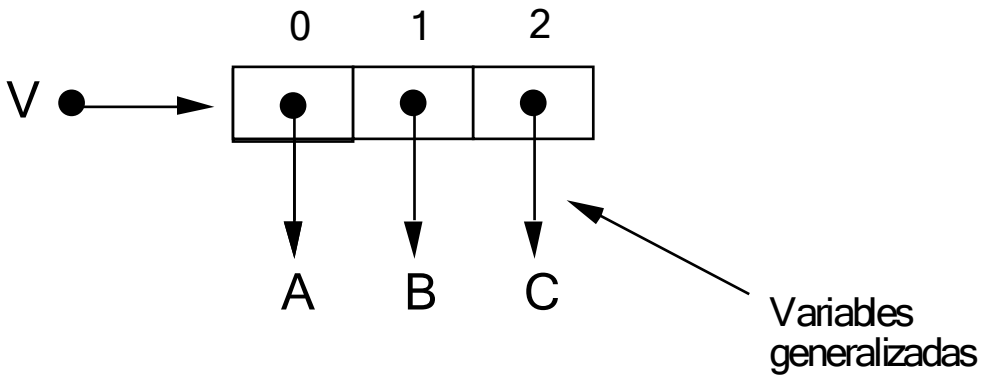


Figura 6

Siguiendo con el mismo ejemplo, si evaluamos:

```
> (setf (aref v 0) (+ 4 6))
===> 10
```

tendremos (ver Figura 7):

```
> (aref v 0)
===> 10
```

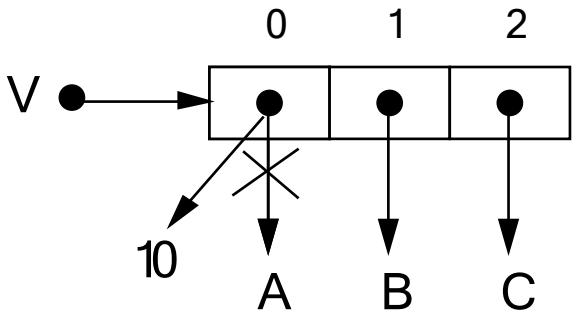


Figura 7

Para terminar esta breve introducción a los vectores en Lisp, hagamos notar que el efecto que percibe el programador es que los vectores son transferidos como parámetros por referencia. Por ejemplo, si evaluamos:

```
> (defun modifica-primero (vector elemento)
      (setf (aref vector 0) elemento))
===> MODIFICA-PRIMERO
```

y a continuación

```
> (setq v (make-array 3))     ===> dependiente del entorno
```

tendremos una situación como la reflejada en la Figura 8.

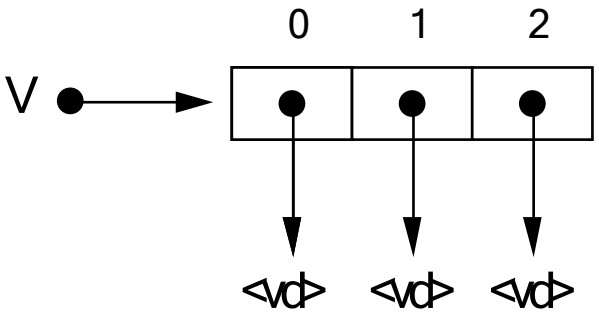


Figura 8

Si evaluamos a continuación:

```
> (modifica-primero v 'a)     ===> A
```

tendremos un proceso como el recogido en la Figura 9, por lo que el resultado observable para el programador es el siguiente (que puede resumirse diciendo que el vector `v` ha pasado por referencia):

```
> (aref v 0)   ===> A
```

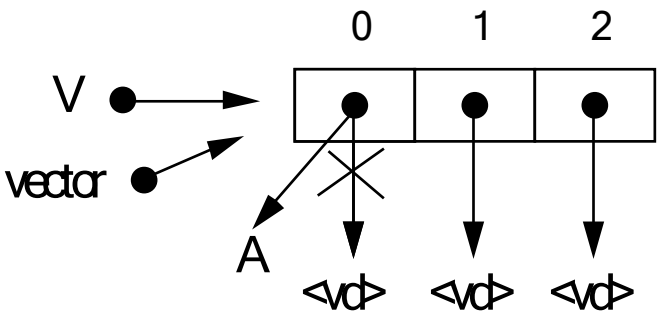


Figura 9

2.4.2. Registros en Lisp.

Los registros en Lisp toman el nombre de "structs" (estructuras). La primitiva para definir una familia de estructuras, todas referidas por un mismo nombre, es `defstruct`. Este *elemento especial* toma como argumentos cualquier número (mayor o igual que dos) de símbolos, *que no serán evaluados*. El primer símbolo será interpretado como el nombre de la estructura y los siguientes serán los nombres de los campos en la estructura. El valor devuelto por una llamada a `defstruct` es el nombre de la estructura definida. Por ejemplo, la evaluación de:

```
> (defstruct libro autor titulo)
===> LIBRO
```

define una familia de registros, de nombre `libro` y con dos campos llamados `autor` y `titulo`. La evaluación de una llamada a `defstruct` tienen varios efectos laterales. Entre ellos, la definición de una función para la construcción de registros de esa familia (el nombre de esta función se construye adjuntando `make-` como prefijo al nombre de la estructura; en el ejemplo anterior: `make-libro`) y la definición de funciones de acceso a cada uno de los campos de la estructura (en el ejemplo, `libro-autor` y `libro-titulo`).

Para construir un ejemplar de una estructura, llamamos al *elemento especial* de construcción (como `make-libro` en el ejemplo) con un número par de argumentos (se le puede llamar sin ningún argumento). Los argumentos que aparecen en lugar impar *no serán evaluados* (y son símbolos construidos poniendo '`:`' como prefijo del nombre de los campos; en el ejemplo, `:autor` o `:titulo`). Los argumentos de lugar par pueden ser elementos Lisp cualesquiera y serán evaluando, pasando sus valores a ser las componentes de los campos del ejemplar de la estructura que es construido. Por ejemplo, la evaluación de:

```
> (setq x (make-libro :autor 'yo))
===> dependiente de la implementación
```

tiene un efecto que ilustramos en la Figura 10.

Ahora, para acceder a los campos de esa estructura podemos emplear los operadores de acceso `libro-autor` o `libro-titulo`. Por ejemplo:

```
>    (libro-autor x)     ===> YO
```

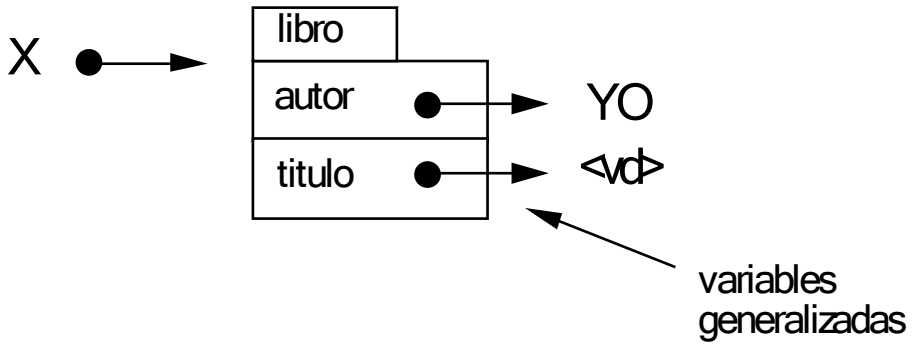


Figura 10

Como hemos recogido en la Figura 10, la ligadura existente entre los campos y las componentes en una estructura debe ser interpretada como una variable generalizada. Por tanto, para modificar la información almacenada en un campo debe utilizarse la primitiva `setf`, como en el caso de los vectores. Por ejemplo (véase la Figura 11):

```
> (setf (libro-titulo x) 'nada)
===> NADA
> (libro-titulo x)
===> NADA
```

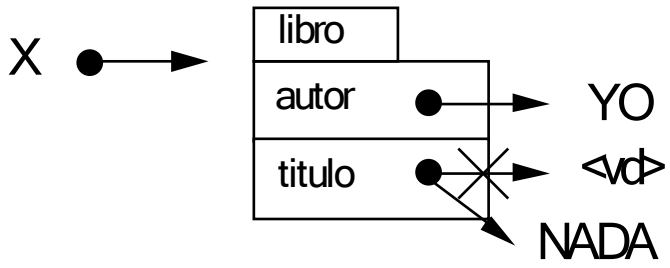


Figura 11

Para terminar, hacer notar que las estructuras, como los vectores, podemos interpretar que *pasan por referencia* como parámetros.

2.4.3. Listas como estructuras de datos en Lisp.

Al final del tema anterior ya hemos introducido de modo breve las listas como estructuras de datos. Allí explicamos las operaciones de acceso `first` y `rest` y el predicado que permite decidir si una lista es la vacía: `endp`. Para construir listas utilizábamos las características del bucle de lectura-evaluación-escritura y el elemento especial `quote` que impide la evaluación. Pero para poder diseñar algoritmos de manipulación de listas como estructuras de datos son necesarias operaciones de construcción de listas. A continuación explicamos las más básicas e importantes.

La primitiva `cons` es una función que construye una nueva lista al añadir el elemento que es su primer argumento como primer miembro en la lista que es su segundo argumento. La función `list` toma cualquier número de argumentos y construye una lista con ellos. Por último, `append` es la función que concatena dos listas. Una cuestión que es frecuentemente olvidada por los programadores principiantes en Lisp es que estas tres primitivas son funciones y, como tales, no tienen ningún efecto lateral. En particular, no modifican el valor de ninguno de sus argumentos. Obsérvese el siguiente comportamiento:

```
> (setq lista '(1 2 3))  ===> (1 2 3)
> (cons 0 lista)         ===> (0 1 2 3)
> lista                  ===> (1 2 3)
```

Véase en los siguientes ejemplos las diferencias de comportamiento entre `cons`, `list` y `append`.

```
> (cons '(a)  '(b c))    ===> ((A) B C)
> (list '(a)  '(b c))    ===> ((A) (B C))
> (append '(a)  '(b c))  ===> (A B C)
```

Para terminar indiquemos otros operadores de acceso como `second` y `third` que permiten acceder, respectivamente, al segundo y tercer elemento de una lista. Por ejemplo:

```
> (third '(a b c))  ===> C
```

Obsérvese el siguiente comportamiento:

```
> (third '(a b))    ===> NIL
```

y el par de ejemplos siguientes:

```
> (second '(a b))   ===> B
```

```
> (rest '(a b))      ===> (B)
```

2.4.4. El nivel físico de la representación para el problema de las garrafas.

Al elegir una representación física en Lisp lo más adecuado es aplicar los principios básicos sobre la abstracción de datos. En lo que sigue nos centramos tan solo en los estados, dejando de lado la representación física de las reglas. Así pues, lo mejor es aislar las operaciones de acceso para los estados, independientemente de cual sea la representación física para ellos. A los estados (x,y) los denominaremos `garrafas`. En este caso tan sencillo las operaciones son las habituales para tratar pares de elementos. Por ejemplo, una especificación podría ser:

- `(defun construye-garrafas (x y) …)` ; devuelve la representación del estado (x,y);
- `(defun garrafa4 (garrafas) …)` ; devuelve x si el argumento representa a (x,y);
- `(defun garrafa3 (garrafas) …)` ; devuelve y si el argumento representa a (x,y);
- `(defun modifica-garrafa4 (garrafas n) …)` ; si el estado era (x,y) lo modifica para construir (n,y) y devuelve el estado así modificado como valor;
- `(defun modifica-garrafa3 (garrafas n) …)` ; si el estado era (x,y) lo modifica para construir (x,n) y devuelve el estado así modificado como valor.

En realidad, las dos últimas operaciones no son necesarias para programar el sistema de producción (basta con poder acceder a las componentes, para verificar las condiciones de las reglas, y con construir nuevos estados, por disparo de reglas), pero las incluimos para poder explicar ciertas características de las representaciones que serán necesarias en temas siguientes.

Ejercicios. Utilizando los operadores anteriores, se pide:

- definir una función `estado-inicial` que construya y devuelva el estado inicial para el problema de las garrafas de vino;
- definir un predicado `estado-objetivo?` que a partir un estado decida si es o no un estado objetivo para el problema de las garrafas de vino.

Ahora el nivel físico de representación consiste en elegir una estructura de datos para los estados e implementar utilizándola cada una de esas funciones (las dos últimas pueden ser en realidad procedimientos funcionales).

Por ejemplo, si decidimos representar las `garrafas` por medio de registros podríamos definir:

```
> (defstruct garrafas tres cuatro)      ===> GARRAFAS
```

y

```
> (defun  construye-garrafas (x y)
     (make-garrafas :cuatro x :tres y))
===>      CONSTRUYE-GARRAFAS

> (defun  modifica-garrafa4 (garrafas n)
     (setf (garrafas-cuatro garrafas) n)
     garrafas)
===> MODIFICA-GARRAFA4
```

En la última operación hemos utilizado que en el cuerpo de un `defun` puede aparecer cualquier secuencia de elementos Lisp, que serán evaluados uno tras otro y el valor resultado de la última evaluación será el devuelto por la llamada a la función (se trata de un *bloque implícito;* es lo que se denomina en la terminología Lisp un *`progn` implícito*). Obsérvese que la secuencia no está encerrada en paréntesis adicionales. De hecho, sería un error escribir en el cuerpo de la anterior función:

```
((setf (garrafas-cuatro garrafas) n)
    garrafas)
```

¿Por qué?

Ejercicios.

1) Completar la implementación de las operaciones con esa representación.

2) Utilizar una representación con vectores (de dos elementos) e implementar en ella las operaciones.

No es difícil repetir el proceso anterior para una representación de las `garrafas` como listas. La única complicación reside en que si queremos interpretar los operadores de modificación en un sentido estricto (es decir, de modo que su parámetro sea efectivamente modificado y no solamente utilizado para construir un

nuevo estado) no disponemos de instrumentos para hacerlo. Por ejemplo, un código como el siguiente:

```
> (defun  modifica-garrafa4 (garrafas n)
     (setq garrafas (list n (second garrafas)))
     garrafas)
===> MODIFICA-GARRAFA4
```

no tendría el efecto perseguido (¿por qué?). Se puede decidir entonces reconstruir el estado en lugar de modificar el que pasa como argumento. Pero en ese caso `modifica-garrafa4` y `modifica-garrafa3` son operaciones secundarias que pueden ser programadas en función de `construye-garrafas`, `garrafa4` y `garrafa3`.

Ejercicios.

1) Programar `modifica-garrafa4` y `modifica-garrafa3` en términos de `construye-garrafas`, `garrafa4` y `garrafa3`, entendiéndose que "modificar" es interpretado como "construir a partir de".

2) Representar `garrafas` por medio de listas e implementar las operaciones primarias `construye-garrafas`, `garrafa4` y `garrafa3`.

Las listas sí pueden ser modificadas en Lisp al estilo de lo que sucede con los registros y vectores, pero un uso inadecuado de esa posibilidad puede conllevar ciertas dificultades bastante graves, por lo que esa posibilidad debe ser utilizada con precaución. La razón por la que las listas pueden ser modificadas reposa en que los primitivas que hemos presentado como de acceso (`first`, `second`, `third`, `rest`) en realidad también *denotan* variables generalizadas, por lo que pueden ser modificadas de un modo destructivo (irrecuperable) por medio del elemento especial `setf`. Por ejemplo, podríamos definir:

```
> (defun  modifica-garrafa4 (garrafas n)
     (setf (first garrafas) n)
     garrafas)
===> MODIFICA-GARRAFA4
```

con el siguiente comportamiento:

```
> (setq g '(1 2))                  ===>      (1 2)
> (modifica-garrafa4 g 4)          ===>      (4 2)
> g                                ===>      (4 2)
```

Ejercicio. Estudiar la representación de los estados para el 8-puzzle. Llegar hasta el nivel físico de la representación.

2.5. Espacios de estados.

Como hemos definido en 2.3, el espacio de estados es el conjunto de estados factibles para un problema junto a la estructura de adyacencia definida implícitamente por los operadores o reglas. Según la estructura que los operadores definan sobre el conjunto de estados hablaremos de árbol o grafo de estados. Las posibilidades relevantes (por cómo influyen en el proceso de búsqueda) son:

- Arbol
- Grafo dirigido acíclico
- Grafo dirigido (no necesariamente acíclico)
- Grafo no dirigido

Empezando por el final, la estructura de *grafo no dirigido de estados* aparece cuando si de un estado `e` se puede pasar a un estado `e'` por aplicación de un operador siempre existe otro operador (o el mismo) que permite pasar de `e'` a `e`. Este es el caso, por ejemplo, del 8-puzzle con la representación con la que venimos trabajando. Con esta estructura, si no se toman las medidas oportunas, la búsqueda puede caer en ciclos que impidan la terminación de la ejecución.

Si no se da la circunstancia anterior, existen estados `e`, `e'` tales que de `e` se puede pasar a `e'`, pero de `e'` no se puede pasar a `e`. En este caso, la dirección u orientación de la adyacencia es importante y se habla de *grafo dirigido de estados*. Por ejemplo, en la representación dada para el problema de las garrafas, del estado (3,0) se puede pasar al estado (0,0) (aplicando el operador vacia-cuatro), pero no se puede pasar (directamente; esto es, por aplicación de una única regla) de (0,0) a (3,0). Pero, en general, que no se pueda pasar directamente no significa que no se pueda pasar por una secuencia de aplicaciones de reglas. Eso posibilita la aparición de ciclos (dirigidos) como el siguiente: (0,0) --> (0,3) --> (3,0) --> (0,0). En este caso, la aparición de bucles infinitos en la búsqueda que se citaron para los grafos no dirigidos de estados también pueden reproducirse y será necesario tenerlo en cuenta en el diseño de estrategias de control.

Si en el grafo dirigido no aparecen ciclos (dirigidos) hablaremos de *grafo dirigido acíclico de estados*. Para poner un ejemplo de un tal espacio de estados, pensemos en la siguiente representación para el problema de las n reinas. Conceptualmente, los estados son configuraciones del tablero en las que, como mucho, aparecen n reinas (pudiendo no aparecer ninguna: el estado inicial). Tenemos n^2 reglas que consisten en poner una reina en cada una de las casillas, siendo las únicas precondiciones el que no haya ya una reina en esa casilla y que todavía queden reinas disponibles (o dicho de otro modo: que en el tablero aparezcan menos de n reinas). En esta situación es claro que se puede llegar a una configuración de más de un modo, pero que no podemos encontrar ciclos dirigidos (puesto que cada aplicación de una regla aumenta el número de reinas en el tablero y nunca quitamos reinas). Si el espacio de estados es un grafo dirigido acíclico no aparecerán problemas de no terminación de la ejecución como los evocados en los dos casos anteriores. Sin embargo, si en el sistema de producción no se toman las precauciones adecuadas, un mismo estado (y todo el subgrafo de sus descendientes) puede ser explorado en muchas ocasiones con el consiguiente perjuicio en la eficiencia (en tiempo y en espacio) del sistema.

Por último, si el espacio de estados es un grafo dirigido acíclico en el que a cada estado se puede llegar sólo de una manera tenemos el caso de un *árbol de estados*. Esta es la estructura combinatorial más sencilla y la que produce menos problemas al realizar la búsqueda en ella. Para encontrar un ejemplo, pensemos en la misma representación para el problema de las n reinas que hemos dado en el caso del grafo dirigido acíclico. La única modificación es que ahora para que una regla sea disparada es necesario que haya una y exactamente una reina en cada una de las filas precedentes a la de la casilla que corresponde a la regla (o que el tablero esté vacío en el caso de reglas correspondientes a casillas de la primera fila). Esta ligera modificación hace que el espacio de estados sea un árbol, lo que simplifica considerablemente la labor de búsqueda de una solución. Este ejemplo muestra claramente que la fase de diseño conceptual es la más importante para conseguir un sistema de producción eficaz.

Es claro que un sistema de producción que considere los espacios de estados más generales (los que corresponden a grafos no dirigidos) funcionará correctamente para aquellos espacios más especializados. Sin embargo, en ocasiones, cuestiones de eficiencia aconsejan que las estrategias de control se particularicen para espacios de estados que son árboles o grafos dirigidos acíclicos.

Otras clasificaciones de los espacios de estados se refieren a su simplicidad, su finitud o infinitud, y a la existencia de pesos en los arcos.

Comenzando de nuevo por el final, los espacios de estados con pesos en las aristas aparecen en sistemas de producción en los que el disparo de las reglas se supone que tiene un coste. Habitualmente en estos casos los problemas de búsqueda aparecen bajo la forma de problemas de optimización.

Para hablar de *espacios de estados infinitos* (todos los que aparecen en los problemas anteriores son finitos) vamos a introducir toda una clase de sistemas de producción que tienen un interés teórico (son una fuente de ejemplos para ilustrar las diferencias entre distintos tipos de espacios de estados y distintas estrategias de control), histórico (fueron introducidos por el lógico Post en los años 30 para estudiar problemas de calculabilidad teórica y son la base matemática para los actuales sistemas de producción) y práctico (muchos sistemas importantes, en particular los relacionados con el tratamiento del lenguaje natural, están directamente basados en ellos). Nos refererimos a los *sistemas (simbólicos) de reescritura*. En ellos los estados son palabras (secuencias de caracteres) en un cierto alfabeto (los elementos del alfabeto son los caracteres) y las reglas son del tipo siguiente: `<cadena1> --> <cadena2>`. Una de estas reglas se activa sobre una cadena cuando ésta contiene como subcadena a `<cadena1>`; el nuevo estado obtenido por aplicación de la regla es la cadena inicial en la que se ha reemplazado una ocurrencia de `<cadena1>` por `<cadena2>`.

Por ejemplo, en un sistema de reescritura con alfabeto `{A,B}` y reglas `A --> AA`, `AB --> B`, cualquier estado inicial que contenga una `A` tendrá una rama infinita. Esto puede producir problemas de no terminación de la búsqueda de una índole diferente a los comentados en los casos de los grafos con ciclos. Pero que un espacio de estados sea infinito no significa que en él la búsqueda no pueda tener éxito. En el ejemplo anterior, si el estado inicial es `ABAB` y los estados objetivos son aquellos en los que no aparece el caracter `A`, es sencillo encontrar una secuencia de aplicación de reglas que llevan del estado inicial al estado objetivo `BB`.

En el mismo ejemplo anterior puede verse que una misma regla puede estar sensibilizada sobre un mismo estado por varias razones distintas, lo que producirá varios disparos distintos de la misma regla sobre el mismo estado. Así sobre `ABAB` la primera regla está sensibilizada dos veces y los estados que se obtienen por los disparos son, respectivamente, `AABAB` y `ABAAB`. En el caso anterior, los dos estados construidos son distintos pero puede también suceder que dos disparos distintos de la misma regla sobre el mismo estado produzcan un único estado. Es el caso de la

primera regla del ejemplo, que es sensibilizada tres veces sobre `AABAB`. Las dos sensibilizaciones correspondientes a las dos primeras ocurrencias de A producirán como resultado un solo estado: `AAABAB`. En este último caso (cuando entre dos vértices aparece más de una arista dirigida) se dice que el *grafo de estados* es *no simple*. En la mayoría de las aplicaciones los *espacios de estados* que aparecen son *simples* (compruébese con todos los ejemplos aparecidos hasta el momento), por lo que en el resto del texto nos olvidaremos de la posibilidad de que algunos grafos de estados pueden ser no simples.

Terminamos este apartado haciendo una comparación entre la teoría algorítmica de grafos y la búsqueda en Inteligencia Artificial. Los principales esquemas algorítmicos (recorrido en anchura, en profundidad, caminos óptimos, algoritmos voraces, etc.) son comunes a ambos campos, pero en la teoría de grafos se supone que el grafo que se recorre, o en el que se busca, es explícitamente conocido. Se trata de un dato para el problema que habitualmente está almacenado en una estructura de datos (lo que excluye el tratamiento de grafos infinitos). Sin embargo, la búsqueda en espacios de estados no supone que el árbol o grafo esté previamente generado. No asume tampoco que se tengan que generar todos los estados (sólo serán tratados los estados necesarios para resolver el problema). Esto es imprescindible si se tiene en cuenta que los problemas que ataca la Inteligencia Artificial suelen ser sumamente costosos en espacio y tiempo. Lo anterior es cierto incluso en los ejemplos que parecen más inofensivos. Por ejemplo, el grafo del 8-puzzle tiene … ¡362.880 vértices! (explosión combinatoria)[1]. De hecho, en la búsqueda en espacios de estados, ni siquiera es necesario que el espacio de estados aparezca de ningún modo reconocible dentro de los algoritmos (aunque el espacio de estados siempre será un marco en el que entender el desarrollo de la ejecución de los algoritmos). Si el espacio de estados es (parcialmente) almacenado o tenido en cuenta por el algoritmo de búsqueda, obtenemos la clase más importante de sistemas de producción / estrategias de búsqueda: la *exploración en grafos de estados*. (Habitualmente en este tipo de sistemas de producción el almacenamiento consiste en listas de *nodos*: estados enriquecidos con algún tipo de información que, en particular, codifica las relaciones de adyacencia; volveremos sobre esta importante noción en los temas siguientes.)

[1] El espacio de estados del 8-puzzle es en realidad 9!. Pero se puede demostrar que sólo la mitad de los etados 9!/2 es alcanzable desde un estado inicial.

2.6. ¿Qué significa encontrar una solución?

Para alcanzar una solución del problema debemos disponer de una descripción de un estado deseado del mundo. Está descripción puede ser completa (por ejemplo, diciendo exactamente cuál es el estado objetivo, caso del 8-puzzle) o puede ser parcial (describiendo algunas propiedades de los estados que consideramos objetivo, como sucede en el problema de las garrafas). En este segundo caso, la forma más habitual de describir los estados objetivos es una función que recibe como argumento un estado y devuelve `verdad` si el estado es un estado objetivo y `falso` en otro caso.

En general, la solución consiste en una secuencia de operadores que transforman el estado inicial en el estado objetivo. Un ejemplo de solución para el 8-puzzle (tal y como ha sido enunciado más arriba) aparece en la Figura 12.

Pero podemos resolver de este modo dos problemas distintos:

- *El problema de decisión*: consiste simplemente en decir si sí o no se puede alcanzar el objetivo desde el estado inicial.
- *El problema de explicación*: en este caso se debe indicar cómo se ha llegado desde el estado inicial hasta el objetivo.

En ocasiones, el estado objetivo mismo ya contiene toda la información necesaria para resolver los dos problemas (por ejemplo, en las n reinas), pero en general no será cierto (por ejemplo el 8-puzzle). En este último caso, el sistema de producción debe disponer de un modo de almacenamiento de los caminos que van siendo explorados.

En cualquiera de los dos problemas, como hemos dicho, la resolución consiste en encontrar una secuencia de operadores que transformen el estado inicial en el estado objetivo, por lo que tiene sentido estudiar en ambos casos las siguientes restricciones sobre las *secuencias solución*:

- Encontrar la secuencia más corta.
- Encontrar la secuencia menos costosa (en este caso se supondrá que los disparos de las reglas tienen asociado un coste o, equivalentemente, que el espacio de estados tiene pesos en las aristas).

- Encontrar cualquier secuencia lo más rápido posible.

En general, al evaluar una estrategia habrá que tener en cuenta el siguiente compromiso:

tiempo de cálculo <==> calidad de la solución.

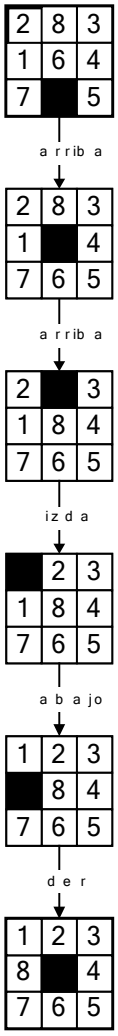


Figura 12

Una última variante para todos los problemas de búsqueda requiere no sólo que se encuentre *un* camino solución, sino *todos* los que existan (por ejemplo, en el problema de las n reinas no se trataría de encontrar una configuración del tablero en la que las n reinas no se coman, sino todas las configuraciones con esa propiedad). Estas variantes se denominan genéricamente *problemas de la Biblioteca Británica* (con tiempo y papel suficiente, unos cuantos monos provistos de máquinas de escribir serían capaces de re-escribir todos los volúmenes del British Museum …).

En el resto del texto, las estrategias explicadas no se ocupan de este problema (sino tan sólo de encontrar *un* camino solución), pero no es difícil hacer en ellas las modificaciones oportunas para que cubran los respectivos problemas de la Biblioteca Británica.

TEMA 3. Estrategias de control no informadas

3.1. Esquema algorítmico genérico de búsqueda.

Veamos una primera aproximación a un algoritmo genérico de búsqueda en un espacio de estados. Como ya hemos indicado al final del apartado 2.5, habitualmente los algoritmos de búsqueda trabajan con estructuras de datos llamadas nodos que contienen en particular información sobre un estado. El esquema algorítmico siguiente está expresado usando esas estructuras.

PRINCIPIO
1 ABIERTOS := (nodo-inicial)
RESUELTO := FALSO
2 **mientras que** ABIERTOS no es vacía **Y NO** RESUELTO **hacer**
3 N := quitar primer elemento de ABIERTOS; E := estado asociado a N
4 **si** E es un estado objetivo
5 **entonces** RESUELTO := verdad
si no para cada operador O **hacer**
6 **si** O se puede aplicar a E
7 **entonces** crear un nodo correspondiente al estado obtenido por aplicación de O a E y añadir ese nodo a ABIERTOS
si RESUELTO
8 **entonces** devuelve el estado objetivo (y si se requiere una explicación, el camino por el que hemos llegado a él)
9 **si no** informa de que el objetivo no puede ser alcanzado
FIN

En los pasos 6-7 se encuentra implicada la parte del sistema de producción que dispara las reglas: el ciclo de reconocimiento (en el paso 6) y actuación (aplicación de O a E). El modo concreto en que se organice ese condicional dependerá de las decisiones de diseño que hayamos adoptado previamente. En particular, esta

versión del algoritmo parece excluir que un mismo operador pueda ser aplicado varias veces sobre el mismo estado (véase el ejemplo del sistema de reescritura tratado en 2.5), pero no es difícil adaptarla para que cubra todos los casos.

El tipo de búsqueda depende del paso número 7 del algoritmo anterior. Se tratará de *búsquedas no informadas* si el algoritmo no conoce nada del problema en concreto que debe resolver. En nuestro contexto eso quiere decir que el paso 7 se realiza con criterios independientes del dominio del problema.

Un concepto importante que aparece cuando la estrategia de control reposa sobre la noción de nodo es el de *árbol de búsqueda*. El árbol de búsqueda es la estructura combinatorial que va siendo construida en el proceso de búsqueda. Sus vértices son nodos y existe una arista dirigida de un nodo a otro si el segundo es construido a partir del primero en el paso 7 del algoritmo anterior. Esta noción está íntimamente relacionada con la de espacio de estados, pero es necesario no confundir ambas. En particular, la estructura combinatorial construida en la ejecución del algoritmo es siempre un árbol puesto que los nodos en el paso 7 son contruidos y no reutilizados. Esto es independiente de que nodos distintos (porque han sido construidos en distintos momentos de la ejecución) tengan asociado el mismo estado. Si esta situación se produce, significa que en el espacio de estados hay ciclos (dirigidos o no) y por tanto que se trata de un grafo de estados y no de un árbol de estados.

Otra noción importante es la de *profundidad de un nodo* (en el árbol de búsqueda): se trata del número de aristas que tiene el camino que lo une con el nodo raíz (ese camino es único, por tratarse de un árbol).

3.2. Búsqueda en anchura.

En este tipo de búsqueda, el paso 7 del algoritmo anterior se particulariza del siguiente modo: los nuevos nodos son añadidos *al final* de la lista ABIERTOS. Así se consigue que los nodos en ABIERTOS estén ordenados según su profundidad, en orden decreciente: los menos profundos al principio, los más profundos al final. La lista ABIERTOS tiene de este modo una forma de acceso que la convierte en una *cola* (o fila de espera): los datos que primero entran (paso 7) son los primeros en salir (paso 3).

Los nodos de igual profundidad se ordenan arbitrariamente. En realidad, en el algoritmo precedente, ese orden arbitrario es directamente heredado del que exista

entre los operadores del sistema de producción (en el bucle '**para cada** operador O **hacer**').

Mediante esta estrategia, el árbol de búsqueda se va generando por niveles de profundidad. Hasta que todos los nodos de un nivel no han sido revisados no se revisa ninguno del siguiente nivel.

Para facilitar la comprensión del proceso de búsqueda vamos a introducir algunos espacios de estados artificiales (que no provienen de ningún sistema de producción real), que serán árboles o grafos etiquetados por letras. Las letras representan los estados. Por ejemplo, en la figura 1 aparece un árbol de estados, con estado inicial I y con un único estado objetivo O.

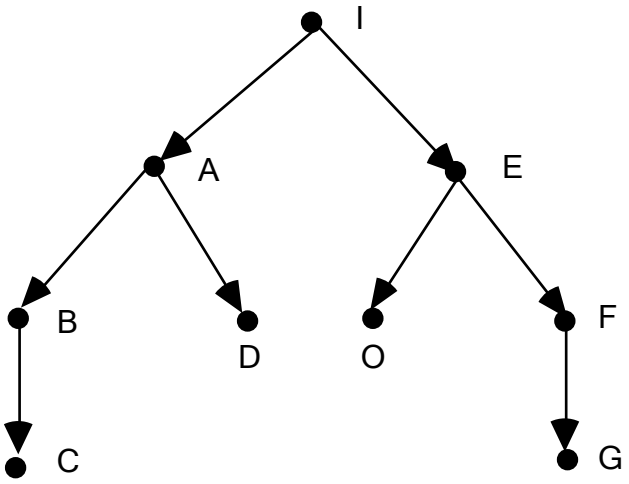


Figura 1

Introduciremos ahora una notación para describir el árbol de búsqueda. En el árbol de búsqueda no se dibujarán los nodos (cuya estructura, en general, puede ser compleja) sino tan solo los estados (letras, en los ejemplos artificiales) y junto a ellos la información relevante sobre los nodos (en el caso de la búsqueda en anchura, ninguna). La información relativa a la adyacencia entre los nodos queda recogida en el gráfico por las flechas dirigidas. En el árbol de búsqueda aparecerán vértices correspondientes a todos los nodos que van siendo generados (es decir, todos aquellos que en algún momento del proceso hayan estado en la lista de ABIERTOS) y se numerarán secuencialmente (con números subrayados) los nodos conforme son examinados.

El árbol de búsqueda correspondiente a la búsqueda en anchura en el espacio de estados de la figura 1 ha sido recogido en la figura 2. Puesto que los nodos de igual profundidad generados a partir de un mismo padre se ordenan de un modo

arbitrario, es necesario indicar cuál es el criterio seguido. En el ejemplo, hemos expandido los estados de "izquierda a derecha".

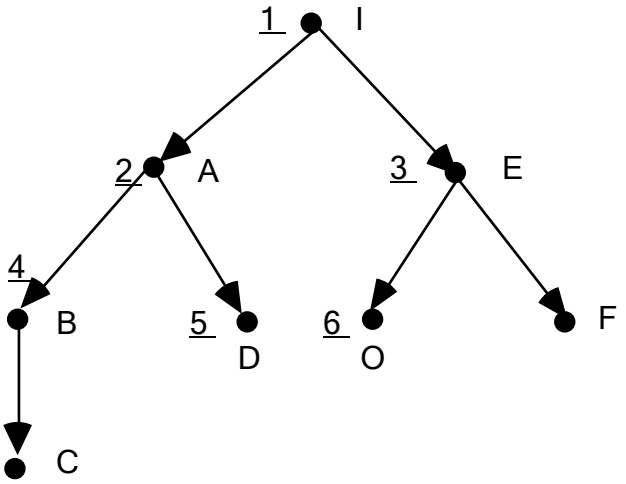


Figura 2

Ejercicio. Dibujar, con las convenciones anteriores, el árbol de búsqueda en anchura para el espacio de estados de la figura 1, con expansión de "derecha a izquierda". Compárese el número de estados generados y examinados con respecto al árbol de la figura 2.

La búsqueda en anchura es una *estrategia exhaustiva*: el recorrido por niveles examina todos los estados que son accesibles desde el estado inicial. En particular, si un estado objetivo es alcanzable, la búsqueda en anchura lo encuentra. La demostración es sencilla: el estado objetivo aparecerá en un cierto nivel de profundidad y, en un número finito de pasos (suponiendo un número finito de operadores y un número finito de posibles aplicaciones de un operador sobre un estado), llegará a ese nivel. Este mismo razonamiento muestra que la búsqueda en anchura siempre encuentra la secuencia solución más corta (una de las que tienen el mínimo número de aristas). Atención: esto no significa en absoluto que realice un número pequeño de operaciones. Se trata de una propiedad de la secuencia solución encontrada, no del proceso por el que ha sido construida (de hecho, veremos que la búsqueda en anchura puede llegar a ser extremadamente ineficaz).

De lo anterior se sigue que, si existe solución, incluso en un espacio de estados con ciclos, como el del problema de las garrafas (la Figura 3 presenta *parte* de ese espacio de estados), el algoritmo de búsqueda en anchura no entrará en bucles infinitos, aunque sí que explorará en numerosas ocasiones la misma zona del espacio de estados, lo que puede perjudicar considerablemente su eficiencia.

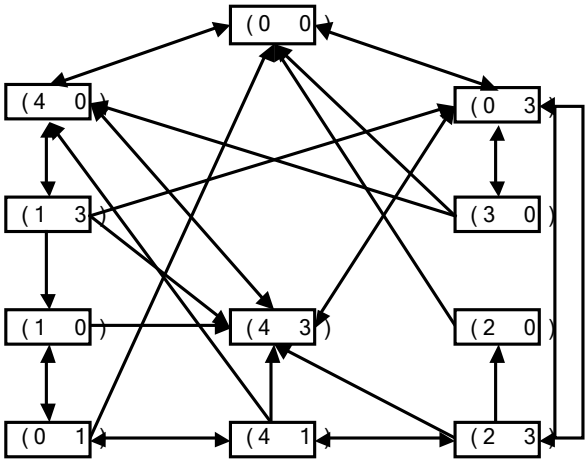


Figura 3

Ejercicio. Estudiar como sería el árbol de búsqueda para el problema de las garrafas con la exploración en anchura en la versión que conocemos.

Sin embargo, si el estado objetivo no es alcanzable y el espacio de estados contiene ciclos dirigidos, la búsqueda en anchura en esta versión no terminará nunca la ejecución. Como ejemplo, consideremos una variante del problema de las garrafas en el que las dos garrafas tienen capacidades de 4 y 2 litros y el objetivo es conseguir que en la garrafa de cuatro litros haya exactamente 3. Es fácil comprobar que este problema no tiene solución (las garrafas siempre contendrán un número par de litros) y en él la búsqueda en anchura, si no tomamos las precauciones adecuadas, entraría en un bucle infinito.

Ejercicio. Representar ese problema de las garrafas. Dibujar el correspondiente grafo de estados. Comprobar que el árbol de búsqueda en anchura es infinito.

Por tanto, la versión genérica explicada en el apartado anterior deberá ser adaptada para cubrir todos los casos posibles de los espacios de estados. Sin embargo, esta versión es muy adecuada, por ser más eficaz que sus variantes, cuando el espacio de estados es un árbol. Este es, por ejemplo, el caso del problema de las n reinas, con una de las representaciones explicadas en el apartado 2.5.

Ejercicios. Consideremos el problema de las 4 reinas. Se pide:

1) comenzar a dibujar el espacio de búsqueda en anchura con la representación del problema del apartado 2.5;

2) modificar la definición de regla sensibilizada para evitar que se examinen configuraciones del tablero que no pueden llevarnos a una solución;

3) estudiar como cambia el árbol de búsqueda en anchura con la modificación del ejercicio anterior.

En cuanto a la estimación de la eficacia de la búsqueda en anchura, podemos identificar genéricamente la complejidad en tiempo con el número de nodos generados (esta simplificación ignora, entre otras cosas, los costes de la verificación de la aplicabilidad de las reglas y de aplicación de las mismas) y la complejidad en espacio con la longitud máxima que puede alcanzar la lista ABIERTOS.

En las complejidades influyen dos factores que son denotados b y d. El número b es el *factor de ramificación* (branching factor): media del número de nodos generados desde un nodo. El número d mide la *profundidad* del estado objetivo (aquí se supone que existe solución): mínimo del número de aplicaciones de reglas necesarias para llegar del estado inicial a un estado objetivo.

No es difícil entonces comprobar que la complejidad en tiempo es de orden exponencial $O(b^d)$. Puesto que en ABIERTOS se extraen los elementos de uno en uno pero se añaden (en media) de b en b, concluimos que la complejidad en espacio es también exponencial $O(b^d)$. Esta complejidad espacial hace que el algoritmo sea impracticable para problemas grandes. Por ejemplo, para el problema del 8-puzzle, b = 3, d = 20 y b^d = 3.486.784.401. En el caso de las garrafas de vino, b = 3, d = 6, b^d = 729.

En resumen, las características de la búsqueda en anchura en esta primera versión son las siguientes. Si existe una solución en el espacio de estados (independientemente de la estructura combinatorial de éste), la encuentra y además con la secuencia solución más corta. Si el espacio de estados es un grafo dirigido acíclico la ejecución termina en todos los casos (si no existe solución, habría que imponer además que el grafo fuese finito), pero puede ser muy ineficaz por explorar en muchas ocasiones las mismas ramas (en particular, en ABIERTOS pueden aparecer muchos nodos con el mismo estado asociado). Si el grafo tiene ciclos dirigidos (o es un grafo no dirigido), el algoritmo puede no terminar si no existe

solución y si ésta existe el algoritmo puede ser muy ineficaz por las mismas razones que en el caso de los grafos dirigidos acíclicos. Por todo lo anterior, vemos que este algoritmo sólo es recomendable en el caso de que el espacio de estados sea un árbol. Por eso, lo denominaremos *búsqueda en anchura en árbol*, aunque ya hemos visto que puede funcionar correctamente (es decir, encontrar una solución) en casos más generales. Independientemente de estas consideraciones, podemos afirmar que la búsqueda en anchura es demasiado ineficaz como para ser aplicada en casos prácticos reales.

Ejercicio. Como ejemplo de que en un espacio de estado infinito si existe una solución la búsqueda en anchura la encuentra, descríbase el árbol de búsqueda en anchura del sistema de reescritura presentado en el apartado 2.5, donde el estado inicial es `ABAB` y son estados objetivo aquellas cadenas en las que no aparece `A`.

Para estudiar la implementación en Common Lisp de la búsqueda en anchura en árbol vamos a presentar dos apartados. En el primero, se explica como trabajar con variables locales y como realizar iteraciones (el algoritmo de 3.1 puede ser fácilmente trasformado utilizando como auxiliar un algoritmo recursivo, pero razones de eficiencia desaconsejan esa trasformación) en Common Lisp, así como ciertas primitivas auxiliares que serán empleadas en la implementación. En el siguiente, aparece el texto de un fichero evaluable Common Lisp con el esquema del programa de busqueda en profundidad y ciertos ejemplos de funcionamiento. Para favorecer la portabilidad del fichero, los comentarios han sido escritos sin acentos y sin los caracteres especiales del castellano (ñ, …).

3.3. Más sobre Common Lisp: variables locales, iteración, miscelánea.

Imaginemos que queremos intercambiar los valores (globales) de dos símbolos `x` e `y`.

```
> (setq x 8)    ===>    8
> (setq y 5)    ===>    5
```

Podríamos hacerlo utilizando una variable global `tmp` con la siguiente secuencia de instrucciones:

```
> (setq tmp x)  ===>     8
> (setq x y)    ===>     5
> (setq y tmp)  ===>     8
```

¿Podemos hacerlo con una única evaluación? Para ello necesitamos componer secuencialmente varias instrucciones Common Lisp. Esto es lo que permite hacer la primitiva `progn` (recuérdese que el cuerpo de `defun` es un *`progn` implícito*). Se trata de una *función* que permite construir un *bloque* de sentencias: cada uno de los objetos que son sus argumentos son evaluados, devolviéndose como valor el del último objeto evaluado. Por ejemplo (estamos suponiendo que vamos evaluando todas las expresiones desde el principio del apartado):

```
> (progn
      (setq tmp x)
      (setq x y)
      (setq y tmp))
 ===>     5
```

Sin embargo, la anterior solución no es totalmente satisfactoria, pues la variable global `tmp` habrá cambiado su valor definitivamente, lo cual es un efecto lateral que no estaba especificado al enunciar el problema al comienzo del apartado. Si queremos tener una versión más segura debemos ser capaces de definir bloques de sentencias con contexto local. Esto puede lograrse utilizando la primitiva `let`. Así para resolver el problema anterior, podríamos evaluar:

```
> (let ((tmp x))
      (setq x y)
      (setq y tmp))
===> 8
>    tmp  ===> 5
```

Nótese que el valor de `tmp` al terminar la evaluación del `let` es el mismo que había antes de la evaluación: durante la evaluación del `let` el valor *local* de `tmp` ha hecho *sombra* al valor *global* de `tmp`, pero al terminar la evaluación la ligadura local ha sido desactivada (es el mismo efecto que puede observarse con la ligadura de los parámetros formales durante la llamada a una función definida con `defun`). Obsérvese también que el cuerpo del `let` (el texto que sigue a la ligadura de las variables locales) es un `progn` implícito (sin paréntesis adicionales, por tanto). Por último, véase que el primer argumento para `let`, que no será evaluado (eso implica que `let` es un elemento especial), es una lista de listas (una lista de listas con dos

miembros en las que el primer miembro es un símbolo, para ser más precisos). Ello es debido a que en un `let` podemos crear cualquier número de ligaduras locales. Véase como ejemplo la siguiente función que calcula la suma del doble y el triple de un número dado.

```
> (defun suma-d-t (x)
    (let ((doble (* 2 x))
          (triple (* 3 x))  )
      (+ doble triple)))
```

Es importante entender como se comporta el evaluador respecto a la lista de listas que es el primer argumento para `let`. Se procede a evaluar cada uno de los segundos miembros de las listas y *tras* ese proceso de evaluación se realizan las ligaduras locales de los valores y los símbolos. Obsérvese el siguiente ejemplo:

```
> (setq x 'exterior)          ===> EXTERIOR
> (let ((x 'interior)
          (y x)  )
         (list x y))          ===> (INTERIOR  EXTERIOR)
```

Por ello, no es posible reutilizar en el primer elemento de un `let` una variable previamente definida en ese mismo `let`. Por ejemplo, la siguiente versión de la función anterior es incorrecta:

```
> (defun suma-d-t (x)
   (let ((doble (* 2 x))
         (triple (+ doble x))  )
        (+ doble triple)))
```

porque el valor local de `doble` todavía no es conocido cuando se evalúa `(+ doble x)`. Para conseguir el efecto deseado se pueden anidar dos `let`, como en:

```
> (defun suma-d-t (x)
   (let ((doble (* 2 x)))
    (let ((triple (+ doble x)))
     (+ doble triple))))
```

o alternativamente utilizar el objeto especial `let*` que se comporta como `let` salvo que las ligaduras locales, entre los valores de los segundos miembros y los símbolos, se producen en orden secuencial: se evalúa el primer segundo miembro, se realiza la ligadura con el primer símbolo, se procede a evaluar el siguiente segundo miembro, etc. Véase una nueva versión correcta de la anterior función:

```
(defun suma-d-t (x)
  (let* ((doble (* 2 x))
         (triple (+ doble x))  )
        (+ doble triple)))
```

Veamos ahora una primitiva que permite la realización de iteraciones en Common Lisp. Se trata del bucle `do`. Supongamos que deseamos pasar las n primeras componentes de un vector a una lista. Una posible definición sería:

```
> (defun vector-a-lista (v n)
   (let ((lista (list )))
    (do  ((i 0 (+ i 1)))
        ((= i n) lista)
      (setq lista (cons (aref v i) lista)))))
```

En primer lugar usamos la primitiva `let` para ligar la variable local `lista` con la lista vacía. Para construir esta última llamamos a la función `list` sin ningún argumento. (Análogamente podríamos haber empleado `nil` o `()`.) El cuerpo del `let` está constituido por un bucle `do`. Se trata de un elemento especial que evaluará selectivamente algunas partes de sus argumentos. En primer lugar nos encontramos con la lista de variables que dirigen el bucle: `((i 0 (+ i 1)))`. Se trata de una lista de listas, al estilo de `let`, pero ahora las listas internas tienen tres miembros: un símbolo (`i` en el ejemplo), un objeto de inicialización (`0`) y un objeto de actualización (`(+ i 1)`). Al entrar al `do` se evalúan todos los objetos de inicialización y tras esa evaluación se realizan las ligaduras *locales* (al `do`) de los valores a los correspondientes símbolos (obsérvese que el comportamiento es más parecido al de `let` que al de `let*`). Una vez realizada esa fase de inicialización, se procede a tratar el segundo argumento del `do` (en el ejemplo, `((= i n) lista)`), segundo argumento que denominaremos *cláusula de parada*. Se trata de una lista con dos partes bien diferenciadas. El primer miembro (`(= i n)`), la *condición de parada*, será evaluado. Si su resultado es "cierto" (es decir, es distinto de `nil`, la lista vacía) se procede a evaluar el resto de elementos de la cláusula de parada, que denominamos *cuerpo de la cláusula de parada*. En el ejemplo, sólo aparece el símbolo `lista`, que será evaluado y su valor será devuelto como valor del `do`, que terminará de ser evaluado. En general el cuerpo de la cláusula de parada es un `progn` implícito. Si la condición de parada se evalúa a `nil` se procede a evaluar el *cuerpo del* `do` que es el texto que aparece desde que se cierra la cláusula de parada hasta que se cierra el `do`. En el ejemplo, se trata sólo de un elemento Lisp: `(setq lista (cons (aref v i) lista))`. En general, se tratará de un `progn`

implícito. Una vez evaluado el cuerpo, se procede a evaluar los objetos de actualización (en el ejemplo: `(+ i 1)`) y tras la evaluación de todos ellos, sus valores son ligados a los símbolos correspondientes, procediéndose a continuación como tras la inicialización.

Realicemos un par de observaciones sobre la función elegida como ejemplo. En primer lugar, nótese que la asignación del cuerpo del bucle `(setq lista (cons (aref v i) lista))` modifica la ligadura local de `lista` creada en el `let` y no otra global que pudiese existir. Como muestra el ejemplo, no es correcto identificar `let` con variables locales y `setq` con variables globales, puesto que `setq` lo que modifica es el valor del símbolo en el *contexto activo* (que puede ser local o global) y no en el contexto global. Por otra parte, al utilizar para ir ampliando la lista la primitiva `cons` resultará que los elementos aparecerán en la lista en orden inverso a como aparecieran en el vector. Si deseamos mantener el orden de aparición de los elementos, podríamos utilizar en el cuerpo de la cláusula de parada la primitiva `reverse`, que es una función que invierte los elementos de una lista.

Utilizando el hecho de que el `do` puede tratar con más de una variable en la lista que es su primer argumento, podemos escribir una versión diferente de la anterior función en la que no aparece `let` y el cuerpo del `do` no contiene ningún objeto Lisp:

```
> (defun vector-a-lista (v n)
    (do ((i 0 (+ i 1))
         (lista (list ) (cons (aref v i) lista))  )
      ((= i n) (reverse lista))))
```

Obsérvese que si esta versión es correcta es debido a que todos los objetos de actualización son evaluados uno tras otro antes de realizar las nuevas ligaduras. Nótese también que hemos utilizado la primitiva `reverse` descrita más arriba.

Terminamos este apartado con una serie de procedimientos de entrada/salida y otras primitivas que serán utilizadas en el programa cuyo texto aparece en el apartado siguiente.

Vamos a introducirlas por medio de un procedimiento que, a partir de un símbolo leído desde el teclado, imprime otros símbolos en la pantalla. Se trata de un procedimiento que simplemente procesa un objeto Lisp leído desde el teclado. Obsérvese que no tiene argumentos. Pueden definirse tales subalgoritmos cuando las entradas van a ser leídas por el teclado, si se usan variables globales o cuando la parametrización depende de las funciones o procedimientos a los que se llame desde su cuerpo. A continuación encontramos la primitiva de escritura en pantalla

`format`. Se trata de un procedimiento que tiene al menos dos argumentos. Si el primero es el símbolo `t` eso indica que deseamos mostrar la información en el dispositivo estándar de salida (por defecto, la pantalla). En este caso el valor devuelto por `format` es `nil`. El segundo argumento debe ser una cadena de caracteres (delimitada por comillas) que se denomina *cadena de control* del `format`. Dentro de la cadena de control aparecen los caracteres estándar, que serán escritos literalmente en pantalla, y los caracteres de control que son precedidos por el carácter reservado `~`. En el ejemplo, aparece el carácter de control `%` que hace que el cursor salte de línea y también el carácter de control `s`. Este último indica que `format` va a tener más argumentos: tantos como veces aparezca la subcadena `~s`; dichos argumentos serán evaluados y sus valores reemplazarán a las subcadenas `~s` correspondientes. Véase su uso en el objeto `(format t "~%Mala entrada: ~s" entrada)`.

```
(defun pregunta ()
   (format   t
          "~% Introduzca una vocal ('Fin' para terminar): ")
   (let ((entrada (read)))
     (if (eq entrada 'fin)
         (progn
           (format t "~%Si esta seguro pulse 'y', si no 'n'")
           (if (y-or-n-p)
               (progn
                 (format t "~%Pulse espacio para terminar")
                 (read-char)
                 (values))
             (pregunta)))
       (progn
         (case entrada
           (a (print 'agua))
           (e (print 'este))
           (i (print 'idioma))
           (o (print 'oso))
           (u (print 'unico))
           (otherwise
             (format t "~%Mala entrada: ~s" entrada)))
         (pregunta)))))
```

Siguiendo en el cuerpo de `pregunta` (es un `progn` implícito), encontramos un `let` en el que la variable local `entrada` es ligada con el resultado devuelto por la primitiva de lectura `read`. Si se utiliza sin argumentos, `read` lee un objeto Lisp del teclado y lo devuelve (sin evaluar) como valor. En el cuerpo del `let` aparece un

condicional y en su condición utilizamos la primitiva `eq`. Se trata de una función similar a `equal`, pero que es más eficaz si lo que queremos es comparar la igualdad de un símbolo con otro objeto Lisp (este es el caso en el ejemplo). Este `if` tiene dos ramas que son sendos `progn`. Esta es una de las escasas estructuras Lisp en las que es normal encontrar un `progn` explícito: `if` no puede tener más de tres argumentos.

En la rama "entonces" de ese condicional, pedimos una confirmación de si se desea terminar la evaluación. Empleamos para ello la primitiva `y-or-n-p` que es un predicado que lee del teclado un carácter. Si la tecla pulsada ha sido `y` devuelve `t`, si `n` devuelve `nil` y en cualquier otro caso vuelve a solicitar un carácter. Si introducimos `n`, la rama "si no" vuelve a llamar recursivamente a `pregunta`. Si introducimos `y`, la rama "si no" se encarga de finalizar la evaluación. La primitiva `read-char` es como `read` salvo que lee un carácter en lugar de un símbolo, número o lista. En el ejemplo, se pide que se pulse la barra espaciadora para evitar el "eco" del carácter tecleado, pero con cualquier otra tecla la evaluación también terminaría. Por último, aparece una llamada, sin ningún argumento, a la primitiva `values`. Esto es hecho para evitar que el carácter pulsado vuelva a aparecer en la pantalla, puesto que `(values)` … no devuelve ningún valor. Se usa cuando queremos que un subalgoritmo sea realmente un procedimiento (es decir, que no devuelva ningún valor) y habitualmente con operaciones de salida en las que no queremos que la parte de escritura del bucle de lectura-evaluación-escritura añada nada en la pantalla. En particular, puede ser interesante usarlo tras un `(format t …)` para evitar que el valor devuelto `nil` aparezca en pantalla.

La rama "si no" de `(if (eq entrada 'fin) …)` es la que realmente procesa la entrada. Se trata en esencia de un condicional (`case`) tras el que se vuelve a llamar recursivamente a `pregunta`. El primer argumento para el elemento especial `case` es un elemento Lisp que será evaluado. Su valor será empleado para compararlo con las *claves* que aparecen en las *cláusulas* *`case`*. El formato de una cláusula `case` es: una lista cuyo primer elemento es la clave, que no será evaluada (por ello es conveniente que las claves de los `case` sean símbolos) y a continuación tenemos el cuerpo de la cláusula, que será un `progn` implícito. El valor obtenido por evaluación del primer argumento de `case` será comparado (con `eq`) secuencialmente con cada una de las claves. Si es igual a alguna de ellas, se procederá a evaluar el cuerpo de esa cláusula y el valor devuelto (por el `progn` implícito) será el valor del `case`, terminando la evaluación del `case` (es similar al `case` de Pascal y diferente del `switch` de C). Si no hay igualdad con ninguna clave, el `case` devolverá `nil`. Para evitar esta situación se puede utilizar la *clave*

reservada `otherwise` que hace que el cuerpo de su cláusula siempre sea evaluado si se llega hasta ella. Esta posibilidad ha sido utilizada en el ejemplo.

Como último comentario al ejemplo, obsérvese que en algunos cuerpos de las cláusulas del `case` aparece la primitiva `print`. Se trata de un procedimiento de salida, pero sin formato. Evalúa su argumento y, como efecto lateral, imprime el valor en pantalla. Otra diferencia con `format` es que el valor impreso será también el valor devuelto por la llamada a `print`. Así, obtendremos resultados que pueden parecer extraños:

```
> (print 'agua)
  AGUA    ===> AGUA
```

Es decir, veremos en pantalla dos veces el símbolo `AGUA`: una vez escrito como efecto lateral del `print` y otra escrito como su valor por la parte de escritura del bucle de lectura-evaluación-escritura. Si quisiésemos tan sólo escribir una vez en pantalla sería mejor evaluar:

```
> (progn
    (print 'agua)
    (values))
AGUA  ===>     "nada"
```

O en este nivel, bastaría hacer :

```
>    'agua     ===> AGUA
```

Sin embargo, esta última solución no sería adecuada en el `case` de la función que estamos comentando (¿por qué?).

3.4. Texto del fichero SEARCH.LSP.

```
;;; Busqueda en anchura en arbol
;; Esquema generico de busqueda:
(defun busqueda ()
  (let ((ABIERTOS (list (crea-nodo-inicial))))
    (do ((el-nodo (first ABIERTOS) (first ABIERTOS)))
        ((or (endp ABIERTOS) (estado-objetivo? (estado-de-nodo el-nodo)))
         (if (endp ABIERTOS)
             (solucion-no-encontrada)
           (escribe-solucion el-nodo)))
      (setq ABIERTOS (reorganizar-nodos-a-expandir (expandir-nodo el-nodo)
                                                   (rest ABIERTOS)))
      (informacion-proceso-busqueda el-nodo))))

;; Las funciones nombradas en ese algoritmo generico pueden depender
```

```
;; del tipo de busqueda, del problema a resolver o de las decisiones
;; de diseño adoptadas (por ejemplo, el modo de representacion de los
;; nodos).

;; (defun crea-nodo-inicial () ...) --> nodo
; Construye el nodo inicial para comenzar la busqueda. Dependera de
; cual es la estructura de los nodos y tambien del problema, mas concretamente
; del estado inicial del problema.

;; (defun expandir-nodo (nodo) ...) --> lista de nodos
; Devuelve una lista con los nodos creados a partir del estado asociado al
; nodo que es su argumento. Si ninguna regla esta sensibilizada para ese
; estado, se devolvera la lista vacia. Esta funcion recoge el aplicador
; de reglas, el ciclo reconocimiento-actuacion. Cubre el bucle interno
; del algoritmo generico, el paso 6 y parte del paso 7. Depende del problema
; y de la estructura de los nodos.

;; (defun reorganizar-nodos-a-expandir (lista-nodos-nuevos lista-nodos) ...)
;;       --> lista de nodos
; Toma como argumentos dos listas de nodos (tal vez vacias) y las reorganiza
; para devolver una lista de nodos. Esta funcion recoge la parte mas importante
; del paso 7 del algoritmo y es la que determina el tipo de busqueda que
; estamos realizando. Por ejemplo, en la busqueda en anchura en arbol la
; funcion no depende de ninguna otra cosa y su codigo seria:

(defun reorganizar-nodos-a-expandir (lista-nodos-nuevos lista-nodos)
  (if (endp lista-nodos-nuevos)
      lista-nodos ; si no hay nodos nuevos la lista ABIERTOS es la de antes
    (append lista-nodos lista-nodos-nuevos)))
        ; en otro caso, los nuevos nodos se an~aden al final de la lista

;; Observese que en la actualizacion de ABIERTOS, se llama a
;; reorganizar-nodos-a-expandir con segundo argumento (rest ABIERTOS),
;; con lo que se consigue "eliminar" el primer nodo en ABIERTOS, que
;; ya habra sido examinado y expandido. Notese tambien que la organizacion
;; del bucle hace que si la busqueda termina sin exito, el valor que tenga
;; la variable el-nodo no sea realmente un nodo sino el simbolo NIL (el
;; "primer elemento" de la lista vacia).

;; (defun estado-objetivo? (estado) ...) --> T o Nil
; Este es el predicado que decide si un estado es un objetivo. Depende
; exclusivamente del problema.

;; (defun estado-de-nodo (nodo) ...) --> estado
; Es la funcion que permite acceder al estado asociado a un nodo. Depende
; exclusivamente de la eleccion de la estructura para los nodos.

;; El uso combinado que se hace de estas dos ultimas funciones y el
;; comportamiento del DO permite no utilizar una variable como RESUELTO
;; que aparecia algoritmo generico.

;; (defun solucion-no-encontrada () ...) --> "indefinido"
; Se trata de un procedimiento final que no es llamado para obtener ningun
; valor (por lo que su resultado queda indefinido), sino para conseguir un
; efecto lateral: la impresion en pantalla de un mensaje que indique que
; la busqueda no ha tenido exito. Depende de la eleccion de disen~o del
; programador (puede querer dar cierta informacion sobre el proceso de
; busqueda), pero un codigo generico podria ser:

(defun solucion-no-encontrada ()
  (format t "~%~%Busqueda terminada sin haber encontrado un estado objetivo.")
  (values))

;; (defun escribe-solucion (nodo) ...) --> "indefinido"
; Otro procedimiento final que informara del resultado de la busqueda,
; en el caso en que un estado objetivo ha sido encontrado. Se
; supone que su argumento tiene asociado un estado objetivo. Aunque el
; procedimiento en general dependera de ciertas decisiones de disen~o,
```

```
; se puede considerar una version generica que pregunta si solo se
; desea conocer cual ha sido el estado solucion alcanzado (problema de
; decision) o si tambien se quiere saber cual es el camino seguido hasta
; encontrarlo (problema de explicacion). En este caso un codigo puede ser:

(defun escribe-solucion (nodo)
  (format t "~% Responda si desea ver el camino solucion: ")
  (if (y-or-n-p)
      (escribe-camino-solucion nodo)
    (escribe-solucion-sin-camino nodo)))

;; (defun escribe-camino-solucion (nodo) ...) --> "indefinido"
; Procedimiento final que muestra en pantalla el camino solucion encontrado.
; Depende de ciertas decisiones de disen~o y, en particular, de la estructura
; de los nodos.

;; (defun escribe-solucion-sin-camino (nodo) ...) --> "indefinido"
; Depende de ciertas decisiones de disen~o, pero como codigo generico se
; podria tomar:

(defun escribe-solucion-sin-camino (nodo)
  (format t "~%~% El estado objetivo alcanzado ha sido:")
  (dibuja-estado (estado-de-nodo nodo))
  (values))

;; En ese procedimiento se usa:
;; (defun dibuja-estado (estado) ...) --> "indefinido"
; Se trata de un procedimiento auxiliar que muestra en pantalla una
; representacion grafica de un estado.

;; (defun informacion-proceso-busqueda (nodo) ...) --> "indefinido"
; Procedimiento auxiliar para ayudar a comprender el modo de funcionamiento
; de la busqueda. Puede ser complejo, pero una version simple generica
; podria ser:

(defun informacion-proceso-busqueda (nodo)
  (dibuja-estado (estado-de-nodo nodo))
  (read-char)) ; Espera que se pulse una tecla tras escribir el ultimo estado
               ; examinado.

;;; EJEMPLOS "ARTIFICIALES".
;; Los operadores estan "camuflados" (no se trata de verdaderos sistemas
;; de produccion)

;; Comenzamos decidiendo las componentes de los nodos y el modo
;; (cabeceras) de acceder a ellas, asi como de construir nodos.
;; (Recuerdese que 'estado-de-nodo' ya ha sido fijado mas arriba.)

;; Vamos a almacenar en un nodo la informacion acerca de su nodo padre
;; (el nodo inicial no tendra padre, pero completaremos ese campo con NIL)

;; (defun padre-de-nodo (nodo) ...) --> nodo (o NIL)

;; (defun crea-nodo (estado padre) ...) --> nodo
; Construye un nodo a partir de un estado y de su nodo padre (o de NIL,
; en el caso del nodo inicial)

;; Una vez determinada la estructura abstracta de los nodos, elegimos
;; una representacion para ellos. En este caso elegimos una representacion
;; con registros.

(defstruct nodo padre estado)

;; Los tres operadores quedan en este caso:

(defun estado-de-nodo (nodo)
  (nodo-estado nodo))
```

```
(defun padre-de-nodo (nodo)
  (nodo-padre nodo))

(defun crea-nodo (estado padre)
  (make-nodo :padre padre :estado estado))

;; Supondremos que en todos los ejemplos el estado inicial es
;; el simbolo i

(defun crea-nodo-inicial ()
  (crea-nodo 'i NIL))

(defun expandir-nodo (nodo)
  (do ((lista-estados (expandir-estado (estado-de-nodo nodo))
                      (rest lista-estados))
       (lista-nodos (list )
                    (cons (crea-nodo (first lista-estados) nodo)
                          lista-nodos)))
      ((endp lista-estados) lista-nodos)))

;; La siguiente funcion es la que define el espacio de estados.
;; En este caso se trata del arbol de estados:

;            i
;          /   \
;         b     a
;        / \   / \
;       f   e d   c
;           |
;           o


; Hemos escrito las adyacencias en orden inverso, porque expandir-nodo
; invierte el orden en la lista. De este modo, si leemos de izquierda a
; derecha y por niveles, veremos aparecer los estados en el mismo orden
; en que van siendo examinados por el algoritmo.

(defun expandir-estado (estado)
  (case estado
    (i '(a b))
    (a '(c d))
    (b '(e f))
    (e '(o))
    (otherwise (list ))))

(defun estado-objetivo? (estado)
  (eq estado 'o))

(defun escribe-camino-solucion (nodo)
  (reverse (construye-camino-inverso nodo)))

(defun construye-camino-inverso (nodo)
  (if (eq nil (padre-de-nodo nodo))
      (list (estado-de-nodo nodo))
    (cons (estado-de-nodo nodo)
          (construye-camino-inverso (padre-de-nodo nodo)))))

(defun dibuja-estado (estado)
  (print estado))

;; Otro ejemplo, en el que no se encuentra la solucion.

(defun expandir-estado (estado)
  (case estado
    (i '(a b))
    (a '(c d))
    (b '(e f))
    (e '(u))
```

```
    (otherwise (list ))))

;; Otro ejemplo mas, en el que se hay dos objetivos, en el mismo nivel:

(defun expandir-estado (estado)
  (case estado
    (i '(a b))
    (a '(c d))
    (b '(e f))
    (e '(o1))
    (c '(o2))
    (otherwise (list ))))

(defun estado-objetivo? (estado)
  (or (eq estado 'o1) (eq estado 'o2)))

;; El objetivo o1 es encontrado.

;; Si alteramos el orden de "disparo de las reglas" puede encontrar o2:

(defun expandir-estado (estado)
  (case estado
    (i '(b a))
    (a '(d c))
    (b '(f e))
    (e '(o1))
    (c '(o2))
    (otherwise (list ))))

;; En cambio si los objetivos estan en niveles diferentes, siempre se
;; encontrara el menos profundo, independientemente del orden de las
;; reglas.

(defun expandir-estado (estado)
  (case estado
    (i '(a b))
    (a '(c d))
    (b '(e f))
    (e '(g))
    (c '(o2))
    (g '(o1))
    (otherwise (list ))))

;; Asi tambien encuentra o2:

(defun expandir-estado (estado)
  (case estado
    (i '(b a))
    (a '(d c))
    (b '(f e))
    (e '(g))
    (c '(o2))
    (g '(o1))
    (otherwise (list ))))

;; Un ejemplo en el que el espacio de estados es un grafo dirigido aciclico,
;; pero no un arbol. Los estados son explorados varias veces.

(defun expandir-estado (estado)
  (case estado
    (i '(b a))
    (a '(d c))
    (b '(e d))
    (d '(e g))
    (g '(o1))
    (otherwise (list ))))

;; Un ejemplo en el que el espacio de estados es un grafo dirigido, pero
```

```
;; con ciclos dirigidos, y en el que existe una solucion. Observaremos
;; un efecto de redundancia muy similar al anterior, pero la solucion sera
;; encontrada sin problemas.

(defun expandir-estado (estado)
  (case estado
    (i '(b a))
    (a '(d c))
    (b '(f e))
    (e '(g))
    (c '(i))
    (g '(o1))
    (otherwise (list ))))
```

```
;; Sin embargo, si el grafo posee ciclos dirigidos y no hay solucion
;; el algoritmo de busqueda entrara en un bucle infinito.

;(defun expandir-estado (estado)
;  (case estado
;    (i '(b a))
;    (a '(d c))
;    (b '(f e))
;    (e '(g))
;    (c '(i))
;    (g '(f))
;    (otherwise (list ))))  ;; Atencion: si se evalua esto y a continuacion
                            ;; (busqueda) se colgara
```

3.5. Búsqueda en grafo.

El programa que aparece recogido en el apartado anterior puede ser particularizado para adaptarse sin dificultad a cualquier sistema de producción.

Ejercicios.

- Particularizar el programa anterior para que resuelva el problema de las n reinas, con la representación que hace que el espacio de estados sea un árbol.
- Particularizar el programa anterior para el problema de las garrafas de vino, con la representación con la que venimos trabajando.
- Particularizar el programa anterior para el problema del granjero, el lobo, el carnero y la lechuga. En este caso habrá que elegir en primer lugar una representación para el problema.

Si se ejecuta la búsqueda con el resultado del ejercicio (2) anterior y se mantiene la definición de `informacion-proceso-busqueda` dada en el fichero `search.lsp`, se comprobará que, pese a que la solución es alcanzada, el número de veces que un mismo estado es examinado (en particular, el estado inicial) es enorme. Para evitar este derroche se puede considerar una nueva versión del esquema genérico de búsqueda en el que los estados ya examinados son almacenados en una lista llamada (de estados) CERRADOS. Cuando se expande un estado sólo dan lugar a nuevos nodos aquellos estados que no *aparezcan* en ABIERTOS o que no *estén* en CERRADOS. Nótese que ABIERTOS es una lista de nodos (mientras que CERRADOS es una lista de estados), por lo que lo que habrá que comprobar es que un estado no es igual a ninguno de los estados asociados a los nodos ABIERTOS.

El esquema genérico que permite realizar búsquedas seguras y no redundantes en espacios de estados que sean grafos (dirigidos o no, acíclicos o no) aparece más abajo.

Vemos que las únicas diferencias con el esquema que hemos denominado búsqueda en árbol son:

- En la línea 1' se inicializa CERRADOS con la lista vacía.
- En la línea 5' se añade el estado que ya ha sido examinado a CERRADOS.
- En las líneas 7' y 7" se comprueba si el estado generado es nuevo.

PRINCIPIO

1 ABIERTOS := (nodo-inicial)

1' *CERRADOS := ()*

RESUELTO := FALSO

2 **mientras que** ABIERTOS no es vacía **Y NO** RESUELTO **hacer**

3 N := quitar primer elemento de ABIERTOS; E := estado asociado a N

4 **si** E es un estado objetivo

5 **entonces** RESUELTO := verdad

5' **si no** *Añadir E a CERRADOS*

para cada operador O **hacer**

6 **si** O se puede aplicar a E

7 **entonces** crear un nodo correspondiente al estado *E'*

obtenido por aplicación de O a E siempre que:

7' – *E' no aparezca en ABIERTOS*

7" – *E' no esté en CERRADOS*

y añadir ese nodo a ABIERTOS

si RESUELTO

8 **entonces** devuelve el estado objetivo (y si se requiere una explicación, el camino por el que hemos llegado a él)

9 **si no** informa de que el objetivo no puede ser alcanzado

FIN

Se podrían considerar variantes en las que las líneas 7' o 7" no son incluidas. Particularizando a la búsqueda en anchura las consecuencias serían las que siguen. Si no incluimos la línea 7" tendremos una versión que no cometerá cierto tipo de redundancias (en concreto, nunca re-examinará dos estados que aparerecieran en la versión anterior simultáneamente en ABIERTOS), pero no tiene en cuenta otras redundancias (cuando vuelve a aparecer un estado que ya ha sido examinado; nótese que esto puede darse tanto en grafos dirigidos acíclicos como en otros que tengan ciclos dirigidos) y, sobre todo, puede entrar en bucles infinitos si los estados objetivos no son alcanzables y hay ciclos dirigidos. Si eliminamos la línea 7' (manteniendo la 7") este último problema no puede darse, pero un estado puede ser examinado varias veces (puesto que puede aparecer en dos nodos distintos de ABIERTOS). En conclusión, la versión con las dos líneas tiene las siguientes propiedades en el caso de la búsqueda en anchura:

- Cada estado del espacio de estados es examinado, a lo más, una vez.
- Si existe solución, encuentra una de longitud mínima, independientemente del tipo de espacio de estados de que se trate (árbol, grafo dirigido acíclico, grafo dirigido o no-dirigido y cada uno de ellos finito o infinito).
- Si no existe solución, siempre terminará la ejecución salvo en el caso (inevitable para las estrategias exhaustivas) en el que el espacio de estados es infinito.

Como se ve las propiedades teóricas de la búsqueda en anchura en grafo son bastante buenas. Si embargo, no hay que olvidar que dicha mejora tiene un precio en cuanto al coste en tiempo (gestión de CERRADOS y algoritmos de búsqueda en ABIERTOS y CERRADOS) y, sobre todo, en espacio (la lista CERRADOS siempre aumenta, nunca se eliminan elementos de ella), por lo que si sabemos que el espacio de estados es un árbol será preferible utilizar la versión inicial. Por ejemplo, sería un derroche utilizar esta versión que acabamos de dar con la representación elegida para el problema de las n reinas.

Este algoritmo de búsqueda en grafo permite clasificar a los estados en cuatro clases. En un instante cualquiera del proceso de búsqueda, distinguimos 4 tipos de estados:

- Los estados todavía no generados (que, por tanto, no aparecen asociados a ningún nodo de ABIERTOS ni están en la lista CERRADOS). Estos estados podemos considerar que tienen una existencia "potencial", puesto que hasta que no es generado un estado no aparece en ABIERTOS ni forma parte de CERRADOS. En general, muchos estados del espacio de estados no llegarán a ser nunca generados en el proceso de búsqueda.

- Los estados ya generados, pero todavía no examinados. Son aquéllos que aparecen en nodos de ABIERTOS.

- Los estados examinados, pero no expandidos. En realidad se trata de una clase con, a lo más, un elemento que es el estado que aparece ligado al nodo valor de la variable N.

- Los estados expandidos, que son aquellos que constituyen la lista CERRADOS.

Es importante hacer notar que la noción del árbol de búsqueda no varía (sigue tratándose de un árbol), aunque sí cambia su relación con el espacio de estados. En este nuevo algoritmo tenemos que una adyacencia del espacio de estados que sea examinada puede dar lugar a lo más a una adyacencia en el árbol de búsqueda, pero también puede no dar lugar a ninguna (en el caso en que el espacio de estados no sea un árbol). En el algoritmo de búsqueda en árbol, cada adyacencia examinada del espacio de estados daba lugar al menos a una adyacencia en el árbol de búsqueda, pero podía dar lugar a varias (en el caso en que el espacio de estados no sea un árbol).

Una última observación respecto a este algoritmo se refiere a que, contrariamente a lo que sucedía en la anterior version, la igualdad entre estados pasa a ser muy importante, puesto que es necesario realizar tareas de localización de estados en listas. Este aspecto es recogido en la siguiente serie de ejercicios.

Ejercicios.

1) Supóngase conocido un predicado:

```
(defun estados-iguales? (estado1 estado2) ...)
```

que decide si dos estados son iguales. Se pide:

a) Programar una función Lisp

```
(defun esta-estado? (estado lista-estados) ...)
```

que decida si un estado se encuentra en una lista de estados.

b) Programar una función Lisp

```
(defun aparece-estado? (estado lista-nodos) ...)
```

que decida si un estado está asociado a alguno de los nodos de una lista (habrá que utilizar la función `estado-de-nodo`).

2) Usar las funciones del ejercicio anterior, para redefinir la función `expandir-nodo` en el caso del problema de las garrafas, de modo que los pasos 7' y 7'' del anterior algoritmo sean incorporados. Obsérvese que la cabecera de la función debe cambiar, pues solamente darán lugar a nuevos nodos los estados que no aparezcan en ABIERTOS o estén en cerrados. Una posible cabecera es:

```
(defun expandir-nodo (nodo lista-nodos lista-estados) ...)
```

Programar también la función `estados-iguales?` para el problema de las garrafas.

3) Escribir en Common Lisp la función que correspondería al algoritmo de búsqueda en grafo (utilizará `expandir-nodo` con la nueva cabecera).

4) Usando la función del ejercicio (2) y el esquema del (3), compárese el proceso de búsqueda para el problema de las garrafas con el obtenido para ese mismo problema con la búsqueda en anchura en árbol.

Utilizando para el problema del 8-puzzle la estrategia de búsqueda en anchura en grafo es posible obtener un árbol de búsqueda como en el que aparece recogido en la figura 4.

3.6. Búsqueda en profundidad.

La estrategia de búsqueda en profundidad es una variante del esquema genérico de búsqueda. Comentamos en primer lugar la versión del punto 3.1, que hemos llamado búsqueda en árbol (después comentaremos la búsqueda en profundidad en grafo, que corresponde al esquema presentado en 3.5).

En este tipo de búsqueda, el paso 7 del algoritmo se particulariza del siguiente modo: los nuevos nodos son añadidos *al comienzo* de la lista ABIERTOS. Así se consigue que los nodos en ABIERTOS estén ordenados según su profundidad, en orden creciente: los más profundos al principio, los menos profundos al final. La lista ABIERTOS tiene de este modo una forma de acceso que la convierte en una *pila*: los datos que primero entran (paso 7) son los últimos en salir (paso 3). Al igual que sucedía en la búsqueda en anchura, los nodos de igual profundidad se ordenan arbitrariamente, según el orden de las reglas u operadores. Veremos más adelante que la búsqueda en profundidad es mucho más sensible a ese ordenamiento de las reglas que la búsqueda en anchura.

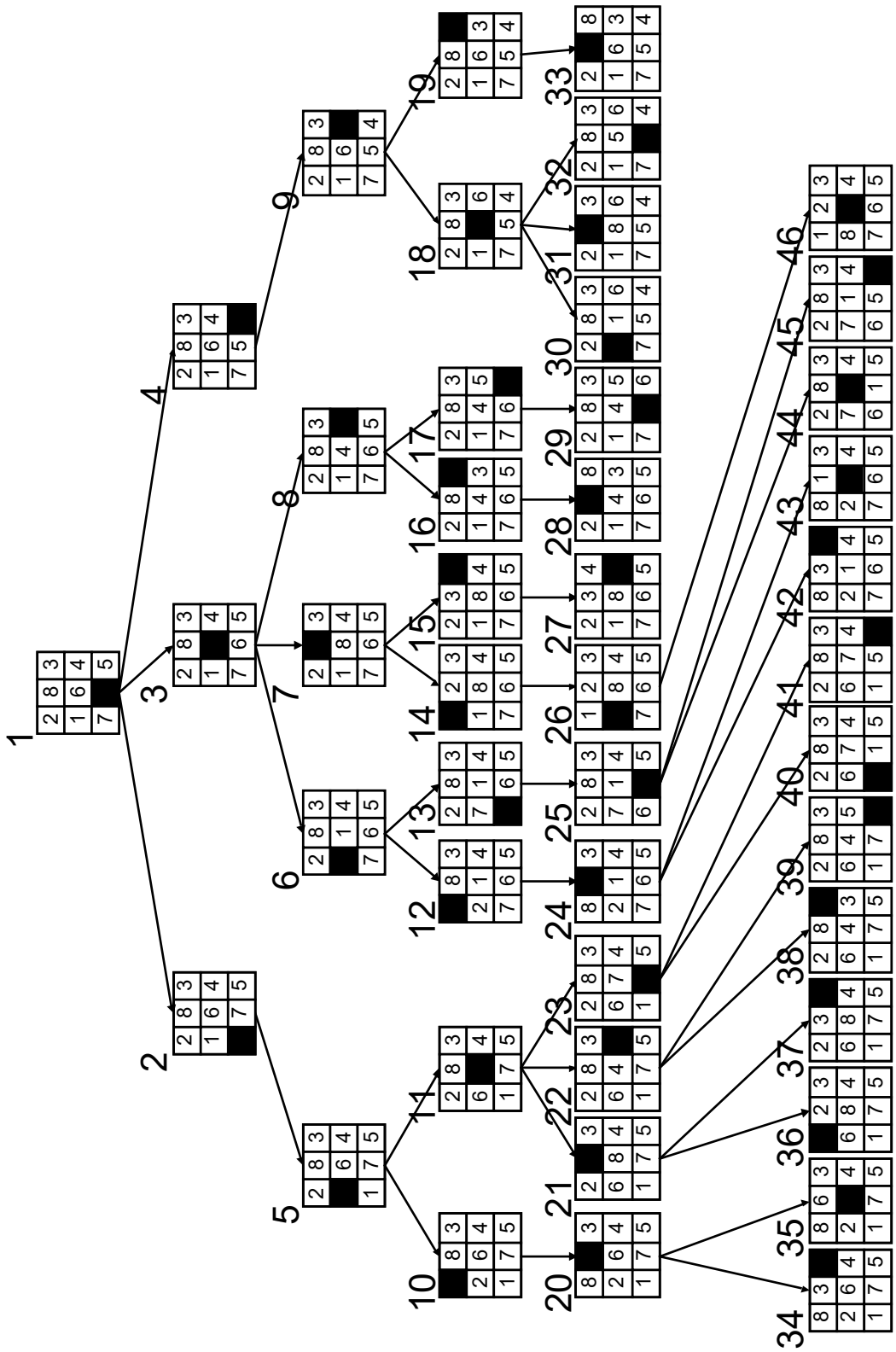


Figura 4

Mediante esta estrategia, se genera un camino hasta encontrar el objetivo o el límite de una rama; en el segundo caso se retrocede y se prueba con caminos alternativos inmediatos.

En cuanto a las modificaciones en la implementación son mínimas. En la búsqueda en anchura (en árbol) teníamos la definición:

```
(defun reorganizar-nodos-a-expandir
                              (lista-nodos-nuevos lista-nodos)
  (if (endp lista-nodos-nuevos)
      lista-nodos
     ; si no hay nodos nuevos la lista ABIERTOS es la de antes
      (append lista-nodos lista-nodos-nuevos)))
     ; en otro caso, nuevos nodos se an~aden al final de lista
```

mientras que en la búsqueda en profundidad el código es:

```
(defun reorganizar-nodos-a-expandir
                              (lista-nodos-nuevos lista-nodos)
  (if (endp lista-nodos-nuevos)
      lista-nodos
     ; si no hay nodos nuevos la lista ABIERTOS es la de antes
      (append lista-nodos-nuevos lista-nodos)))
     ; en otro caso, nuevos nodos se an~aden al comienzo de la
     ; lista
```

En cuanto a la comparación de los dos tipos de búsqueda, una primera observación es que la búsqueda en profundidad necesita, en general, menos espacio para ser realizada. En concreto, la complejidad en espacio será O(bd) frente a la complejidad exponencial $O(b^d)$ de la búsqueda en anchura. Recuérdese que b es el factor de ramificación (media del número de nodos generados desde un nodo). y d mide la profundidad del estado objetivo (mínimo del número de aplicaciones de reglas necesarias para llegar del estado inicial a un estado objetivo).

En cambio, la complejidad asintótica en tiempo es de orden exponencial $O(b^d)$, como era la de la búsqueda en anchura. Pese a este resultado teórico en ocasiones el número de estados examinados por la búsqueda en profundidad puede ser considerablemente menor que en la búsqueda en anchura. Compárese la Figura 5, que contiene un árbol de búsqueda en profundidad (en grafo y con límite de profundidad; ver luego) para el 8-puzzle con la Figura 4.

Ejercicio. Dibújese el árbol de búsqueda en profundidad para el problema de las 4 reinas con la representación que venimos trabajando. Compárese su tamaño con el del árbol de búsqueda en anchura para el mismo problema con la misma representación.

Otra característica que diferencia a las dos estrategias es que, como ya comentamos, la búsqueda en profundidad es mucho más sensible que la búsqueda en anchura en lo que se refiere al orden de aplicación de las reglas. Por ejemplo, en el problema de reescritura que hemos ido estudiando si el orden de las reglas es `[A --> AA, AB --> B]` la búsqueda en profundidad no termina, mientras que si el orden es `[AB --> B, A --> AA]` la búsqueda termina, encontrando un objetivo.

Ejercicio. Comprobar las afirmaciones anteriores. Comparar con los árboles de búsqueda en anchura primero con el orden `[A --> AA, AB --> B]` y después con el orden `[AB --> B, A --> AA]`.

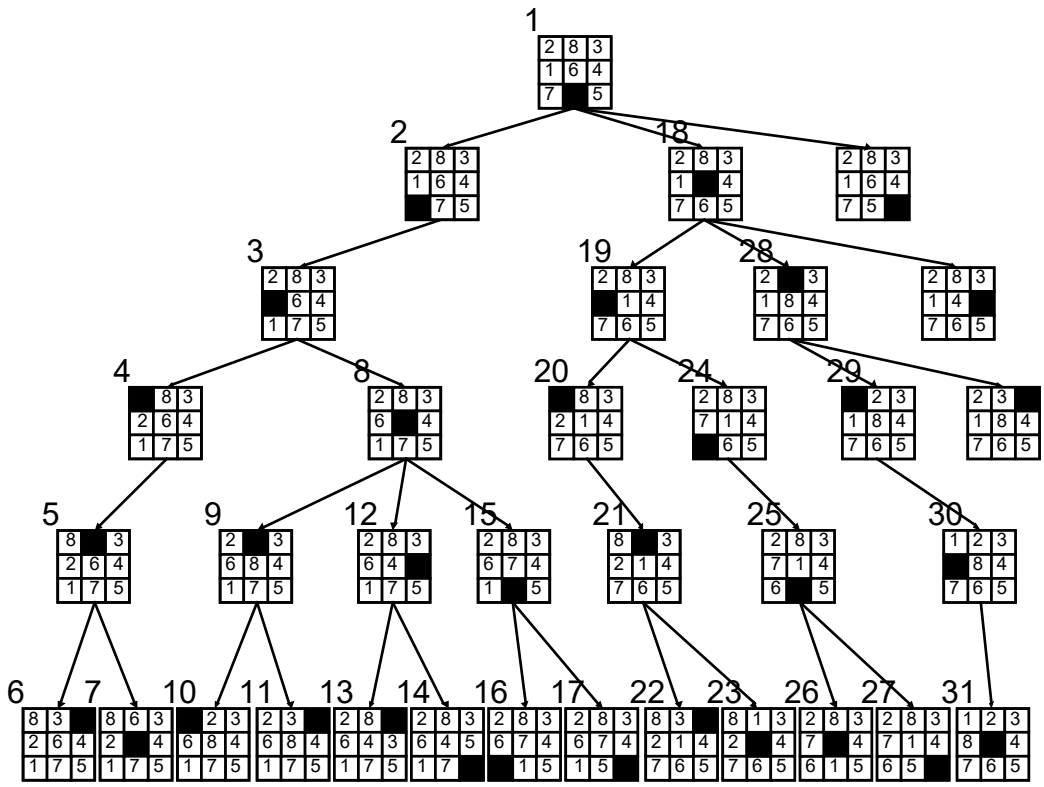


Figura 5

Otro caso en el que el orden de las reglas es importante es el de las n reinas. Aunque en los ejercicios previos no se precisó nada, lo más cómodo desde el punto

de vista de la programación es ordenar las reglas (casillas) de derecha a izquierda o de izquierda a derecha. Formalmente, si $R_{i,j}$ representa la regla "pon una reina en la casilla (i,j)" las dos ordenaciones que podríamos construir, sin mucha reflexión previa, serían $[R_{i,1}, R_{i,2}, \ldots, R_{i,n}]$ o $[R_{i,n}, R_{i,n-1}, \ldots, R_{i,1}]$. Nótese que sólo es relevante el orden de las reglas en una misma fila, pues, por el modo de representación, son las únicas que pueden estar sensibilizadas simultáneamente. Sin embargo, esas ordenaciones no tienen en cuenta que no todas las casillas "cubren" el mismo espacio en el tablero. Por ejemplo, una dama en la diagonal principal (en una casilla (i,i)) limita mucho las posibilidades en las siguientes filas. Siguiendo esta idea podemos definir la siguiente función asociada a cada casilla: diag(i,j) es la máxima longitud de las dos diagonales que pasan por la casilla (i,j). Por ejemplo, en el caso de las 4 reinas, diag(3,2) = 4, ya que una de las diagonales cubre 3 casillas y la otra 4. Entonces ordenamos las reglas en una misma fila del siguiente modo: $R_{i,j}$ precede a $R_{i,m}$ si o bien (1) diag(i,j) < diag(i,m), o bien (2) diag(i,j) = diag(i,m) y j < m. Por ejemplo, en las 4 reinas el orden en la fila 3 sería: $[R_{3,1}, R_{3,4}, R_{3,2}, R_{3,3}]$.

Ejercicios.

1) Dibujar el árbol de búsqueda en profundidad para las 4 reinas con este nuevo orden en las reglas. Compárese con el árbol de búsqueda con las ordenaciones anteriores. Comprobar que en anchura no varía demasiado el árbol de búsqueda con cualquiera de las ordenaciones.

2) Implementar esta variante (basta con modificar la función `expandir-nodo`).

Estas últimas disquisiciones respecto al orden de las reglas en el problema de las n reinas podrían llevar a pensar que estamos aprovechando información sobre el problema concreto que vamos a resolver y, por tanto, que las estrategias de búsqueda (en anchura y en profundidad) no serían "ciegas" en este caso. Sin embargo, la diferencia entre estrategias no informadas y estrategias heurísticas se encuentra en el momento en el que el conocimiento particular sobre el problema es aplicado. En las estrategias no informadas el conocimiento es utilizado en la fase de diseño del sistema de producción (y de análisis: piénsese en como afecta el modo de representar el problema en el proceso de búsqueda). Aquí se incluye qué son las reglas, cómo están implementadas, etc. y, también, cuál es el orden de disparo de las reglas sensibilizadas. Por el contrario, en las estrategias heurísticas el conocimiento particular sobre el problema a resolver es utilizado en tiempo de ejecución, dirigiendo directamente la búsqueda.

Analicemos las propiedades de la búsqueda en profundidad. En primer lugar, hagamos notar que, en los casos en que la búsqueda en profundidad encuentre un camino solución, nada nos asegura que dicho camino solución sea de longitud mínima. Véase en el árbol de estados de la figura 6 (estado inicial: I, únicos estados objetivo: O1 y O2), cuyo árbol de búsqueda en profundidad (con expansión de izquierda a derecha) aparece en la figura 7.

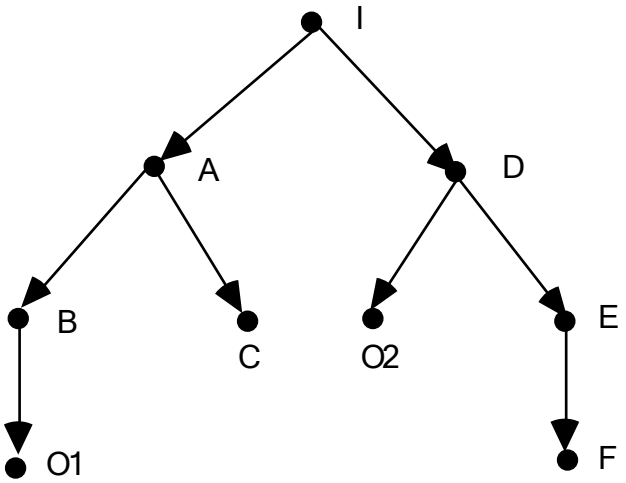


Figura 6

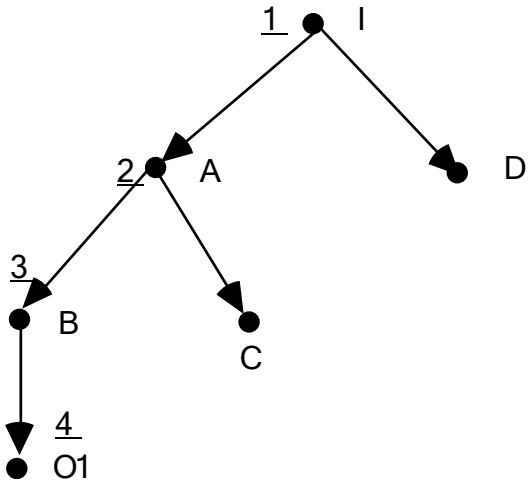


Figura 7

Ejercicio. Comprobar que la búsqueda en profundidad con expansión de derecha a izquierda aplicada a la figura 6, sí encuentra el camino solución de longitud mínima.

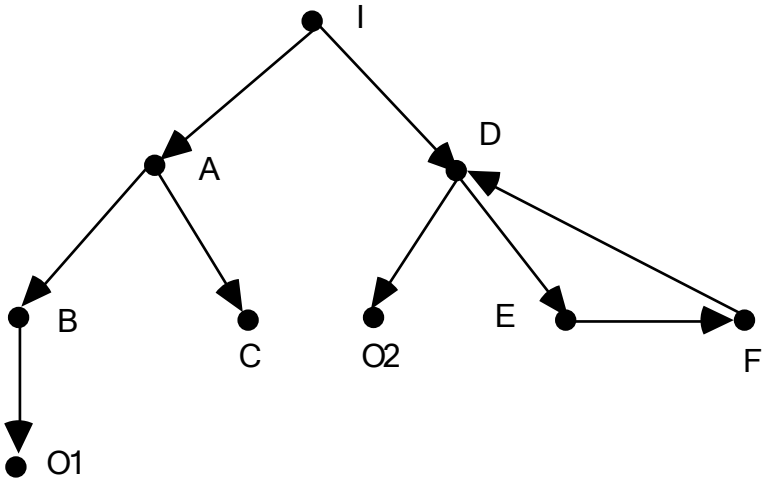


Figura 8

Estudiemos en primer lugar las propiedades de la versión en árbol (la que no almacena los estados que ya han sido examinados en una lista de CERRADOS) de la búsqueda en profundidad. Si el espacio de estados es finito y es un árbol o un grafo dirigido acíclico, la estrategia encontrará un estado objetivo, si existe. Si no existe, examinará todos los estados (es decir, se trata de una *estrategia exhaustiva*). Si el espacio de estados es un grafo dirigido con ciclos dirigidos, la ejecución no terminará si no existen estados objetivos (esto ya ocurría con la búsqueda en anchura en árbol), pero si existen estados objetivos la búsqueda puede terminar o no (compruébese con el problema de las garrafas). Aún peor, que la búsqueda termine (y encuentre un objetivo) o que no termine puede depender del orden del disparo de las reglas. Un ejemplo de esto es ilustrado con el grafo de estados de la figura 8. El árbol de búsqueda en profundidad (en árbol) con expansión de izquierda a derecha es igual al que aparece en la figura 7. Parte del árbol de búsqueda (infinito) si la expansión es de derecha a izquierda aparece en la figura 9. La misma situación se reproduce con los espacios de estados (árboles o grafos) que sean infinitos (recuérdese lo que ocurre con el ejemplo del sistema de reescritura).

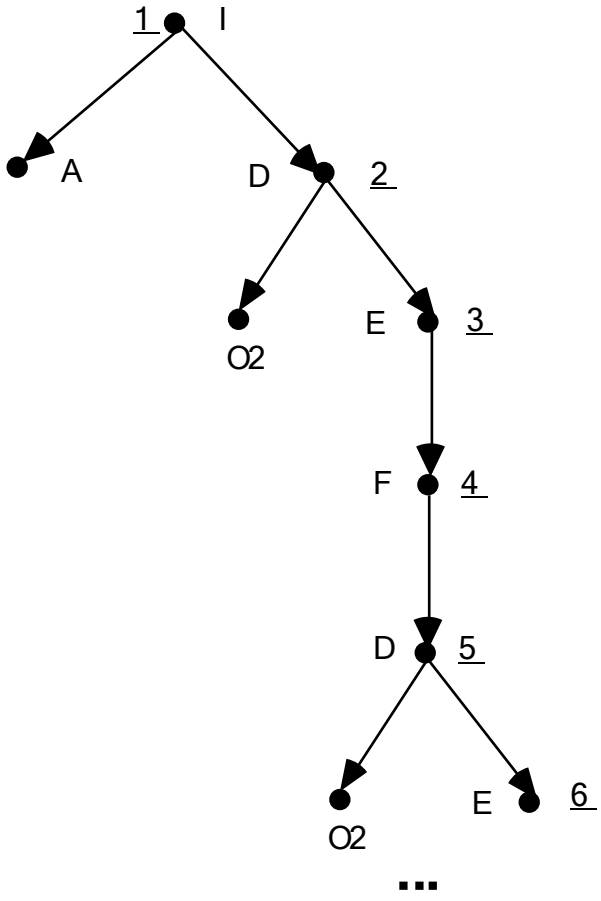


Figura 9

En cuanto a la versión en grafo (la que mantiene la lista CERRADOS), evitará la no terminación en el caso de grafos dirigidos (finitos) con ciclos dirigidos (compruébese de nuevo en el caso del problema de las garrafas), lo que permitirá encontrar siempre un estado objetivo (si existe alguno) y hace que la dependencia del orden de las reglas sea mucho menor. Sin embargo, en un espacio de estados infinito, pese a que haya caminos solución, esta versión tampoco asegura que la búsqueda en profundidad los encuentre ni que termine la ejecución (de nuevo el ejemplo del sistema de reescritura). Si queremos asegurar que la búsqueda en profundidad termine siempre su ejecución en el caso en que haya caminos solución e independientemente de cómo sea el espacio de estados (finito o infinito, grafo o árbol), propiedad que tiene la búsqueda en anchura, estamos obligados a perder el carácter exhaustivo de la búsqueda. También veremos que esto implica necesariamente que, en ocasiones, aunque existan caminos solución la búsqueda no encuentrará ninguno.

Un modo de conseguir que la búsqueda en profundidad termine siempre es el siguiente. Se fija un límite máximo de profundidad (por medio de un parámetro global, por ejemplo) y no se exploran aquellos nodos tales que su profundidad en el

árbol de búsqueda es mayor que dicho límite. Véase en la figura 10 el árbol de búsqueda correspondiente al grafo de la figura 8, con expansión de derecha a izquierda, búsqueda en profundidad en árbol y límite de profundidad 3. La profundidad de cada nodo ha sido escrita al lado de cada estado, separada con una coma. Nótese que el nivel de profundidad está asociado al nodo y no al estado, pues un mismo estado puede reaparecer en varios niveles (compruébese en el ejemplo si el límite de profundidad es 4 o mayor). Obsérvese también que si el límite de profundidad es 1, en ese ejemplo, no se encontrará solución (esto sucederá siempre que el límite de profundidad sea estrictamente menor que la profundidad del estado objetivo menos profundo). En la figura 5, donde se recogía el árbol de búsqueda en profundidad (en grafo) para el 8-puzzle, el límite de profundidad fijado era 5.

Ejercicios. Estúdiese cuál es el estado objetivo encontrado en el espacio de la figura 8 por la búsqueda en profundidad en árbol, si la expansión se realiza de izquierda a derecha y el límite de profundidad es 3. ¿Y si el límite de profundidad es 2?

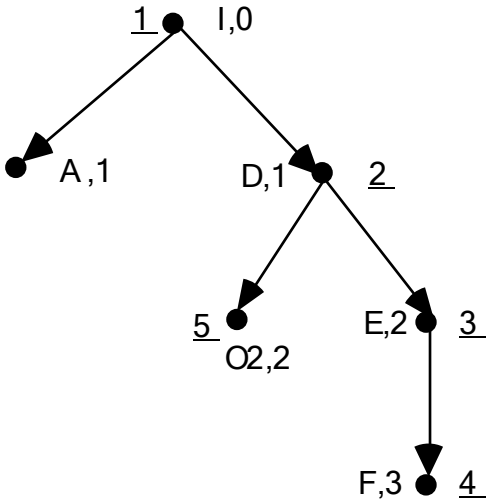


Figura 10

En cuanto a la puesta en práctica de la idea anteriormente expuesta es necesario disponer de una función `profundidad-de-nodo` que, tomando un nodo como argumento, nos indique en que nivel de profundidad ha sido generado. Una vez se dispone de una función tal, no es difícil modificar `expandir-nodo` para que, si hemos alcanzado el límite de profundidad, devuelva siempre la lista vacía, con lo que se consigue el efecto deseado.

Para poder implementar `profundidad-de-nodo` pueden emplearse diversas estrategias. Por ejemplo, si el nodo tiene información sobre su "padre" puede encontrarse su profundidad calculando la longitud del camino que lo une con el nodo

inicial. Otra idea análoga sería almacenar los operadores que han producido el estado del nodo a partir del estado inicial y contar su número. Estas dos posibilidades conllevan un coste en tiempo de ejecución cada vez que se llama a `profundidad-de-nodo`. Otra solución que no requiere tiempo de ejecución, pero que conlleva una necesidad mayor de memoria, consiste en almacenar en cada nodo explícitamente (por medio de un campo, por ejemplo) su nivel de profundidad. Por último, la solución más económica en tiempo y en espacio consiste en mantener en una variable global la profundidad en que nos encontramos en cada momento del proceso de búsqueda. Su inconveniente es que es necesario señalar de algún modo cuando hemos cambiado de nivel. La forma más natural de hacerlo es incluyendo algun tipo de marcas (elementos que no son nodos) en la lista ABIERTOS, lo que no es excesivamente elegante desde el punto de vista de la programación (ABIERTOS deja de ser una lista de nodos para convertirse en algo más general y de interpretación más difícil). Obsérvese que no es conveniente (aunque es mucho más claro) el almacenar el cambio de nivel (si existe) en los nodos, pues ello requeriría un coste en espacio equivalente al almacenaje explícito del nivel de profundidad.

Ejercicios.

1) Impleméntese la búsqueda con límite de profundidad. Utilícese una función genérica `(defun profundidad-de-nodo (nodo) …)`. Será necesario cambiar `expandir-nodo`.

2) Impleméntese la función `(defun profundidad-de-nodo (nodo) …)` de dos modos distintos: (a) calculando la longitud del camino al nodo inicial (lo que, en general, requiere información sobre el "padre" en cada nodo); (b) con un campo explícito para almacenar la profundidad (en este segundo caso será necesario alterar de nuevo `expandir-nodo`).

3) Verificar como funcionan las correspondientes búsquedas en el problema de las garrafas.

Como ya hemos señalado, la profundidad de la que se habla en la anterior variante es la profundidad *en el árbol de búsqueda*. Esto significa que si la variante es aplicada al algoritmo de búsqueda en árbol y el espacio de estados es un grafo, nada nos asegura que un estado no va a ser examinado más de una vez (compruébese en el problema de las garrafas y, como se indicó más arriba, en el espacio de la figura 8). Por tanto, sigue teniendo sentido el trabajar con la versión en grafo (variante del algoritmo de 3.5), aunque ahora sea sólo por economía y no por evitar los bucles infinitos.

Repetimos de nuevo que las versiones con límite de profundidad pierden el carácter exhaustivo, por lo que en ocasiones pueden "olvidarse" de la solución aunque ésta exista. Por ejemplo, en el problema de las n reinas siempre se terminará sin encontrar la solución si el límite de profundidad es estrictamente menor que n.

Ejercicio. Comprobar, dibujando el árbol de búsqueda, que en el sistema de reescritura no se encuentra solución si el límite de profundidad es uno, mientras que sí se encuentra si el límite es dos (o mayor que dos). Compruébese también que apenas varía el proceso si invertimos el orden en las reglas.

Para terminar el tema, resumimos brevemente la comparación entre las principales estrategias no informadas: la búsqueda en anchura y la búsqueda en profundidad.

Ventajas de la búsqueda en profundidad:

- necesita menos memoria (en profundidad sólo se almacenan, en esencia, los nodos del camino que se sigue, mientras que en anchura se almacena la mayor parte del árbol que se va generando).
- con "suerte" puede encontrar una solución sin tener que examinar gran parte del espacio de estados (depende en gran medida de que el orden de las reglas sea "adecuado").

Ventajas de la búsqueda en anchura:

- es más difícil que quede atrapada explorando callejones sin salida.
- si existe una solución garantiza que la encuentra (independientemente de la estructura combinatorial del espacio de estados) y además será una de las de longitud mínima.

TEMA 4. Estrategias de control heurísticas

4.1. Heurísticas.

Los métodos de búsqueda en anchura y en profundidad son denominados *métodos ciegos* de búsqueda. En general, serán muy ineficaces y sin aplicación práctica debido al crecimiento exponencial del tiempo y del espacio que requieren (*explosión combinatoria*).

Los métodos de búsqueda heurística disponen de alguna información sobre la proximidad de cada estado a un estado objetivo, lo que permite explorar en primer lugar los caminos más prometedores.

Son características de los métodos heurísticos:
- No garantizan que se encuentre una solución, aunque existan soluciones.
- Si encuentran una solución, no se asegura que ésta tenga buenas propiedades (que sea de longitud mínima o de coste óptimo).
- En *algunas ocasiones* (que, en general, no se podrán determinar a priori), encontrarán una solución (aceptablemente buena) en un tiempo razonable.

Podemos decir que los métodos heurísticos sacrifican la completitud (es posible que algunas soluciones se "pierdan") al incrementar la eficiencia. En general, los métodos heurísticos son preferibles a los métodos no informados en la solución de problemas difíciles para los que una búsqueda exhaustiva necesitaría un tiempo demasiado grande. Esto cubre prácticamente la totalidad de los problemas reales que interesan en Inteligencia Artificial.

En cuanto al aspecto algorítmico, el mismo esquema presentado en 3.1 (es la versión en árbol; ver 3.5 para la versión en grafo) sirve para las búsquedas heurísticas o informadas. La única diferencia es que los pasos 6 (sensibilización de las reglas) y 7 (reorganización de ABIERTOS) se realizan con criterios dependientes del dominio del problema, de modo que el algoritmo tiene cierto conocimiento, que utilizará en tiempo de ejecución, sobre el problema concreto que debe resolver.

En lo referente a la implementación, las búsquedas heurísticas pueden aprovechar la información en dos funciones diferentes:

```
> expandir-nodo
> reorganizar-nodos-a-expandir
```

(En el primer caso se trata de métodos de la familia "escalada" y en el segundo, que también puede aprovechar la información en `expandir-nodo`, se tiene la familia "primero el mejor".)

La información del problema concreto que estamos intentando resolver se suele expresar por medio de *heurísticas*. El concepto de heurística es difícil de aprehender. Newell, Shaw y Simon en 1963 dieron la siguiente definición: "Un proceso que puede resolver un problema dado, pero que no ofrece ninguna garantía de que lo hará, se llama una heurística para ese problema".

Si nos planteamos seguir concretando como aprovechar la información sobre el problema en sistemas de producción, la siguiente idea consiste en concentrar toda la información heurística en una única función que se denomina *función de evaluación heurística*. Se trata de una función que asocia a cada estado del espacio de estados una cierta cantidad numérica que evalúa de algún modo lo prometedor que es ese estado para acceder a un estado objetivo. Habitualmente, se denota esa función por h(e).

La función heurística puede tener dos interpretaciones. Por una parte, puede ser una estimación de la "calidad" de un estado. En este caso, los estados de mayor valor heurístico son los preferidos. Por otra parte, la función puede ser una estimación de lo próximo que se encuentra el estado de un estado objetivo. Bajo esta perspectiva, los estados de menor valor heurístico son los preferidos. Ambos puntos de vista son complementarios (de hecho, un cambio de signo en los valores permite pasar de una perspectiva a la otra).

Las funciones *heurísticas* más interesantes son aquéllas que son *bien fundadas*. La definición depende de los dos casos considerados en el anterior párrafo. Si la heurística asigna valores altos a los estados preferidos, se dirá que es bien fundada si todos los estados objetivos tienen valores mayores que el resto de estados (habitualmente el estado inicial tendrá valor 0 y los valores seán números naturales o reales positivos). Si la heurística estima la proximidad a un objetivo, se dirá bien fundada si a los estados objetivo les asocia el valor 0.

En todo lo que sigue (y mientras no se diga lo contrario)*, las funciones heurísticas serán siempre bien fundadas y supondremos que miden lo cerca que estamos de alcanzar un objetivo* (es decir, valores menores son preferibles).

Lo realmente difícil en el diseño de un sistema de producción con estrategia de control informada es encontrar la función heurística adecuada.

Veamos ejemplos de heurísticas para algunos problemas concretos. Para el problema del 8-puzzle tenemos la siguientes heurísticas:

a) La basada en la distancia Manhattan (o distancia taxi). Se asocia a cada casilla un número que es su distancia taxi con su posición en el tablero objetivo (esto es, la suma de diferencias de sus coordenadas x e y). La función heurística es la suma de las distancias de cada una de las casillas (excluyendo la que se encuentra vacía).

b) Otra heurística, mucho más simple, consiste en contar el número de casillas que están fuera de su sitio (respecto al tablero objetivo). Es una heurística más pobre que la anterior, puesto que no usa la información relativa al esfuerzo (número de movimientos) necesario para llevar una pieza a su lugar.

c) La distancia taxi no tiene en cuenta que si dos casillas adyacentes deben ser invertidas (por ejemplo, si tenemos a la izquierda del dígito 2 el dígito 1 y en el tablero objetivo el dígito 1 debe estar a la derecha del 2 y en la posición actualmente ocupada por éste) necesitaremos más de dos movimientos. Entonces, podríamos tomar como heurística el doble del número de pares de piezas que están en esa disposición. Esta heurística también es pobre, puesto que se concentra demasiado en un cierto tipo de dificultad. En particular, tendrán valor 0 muchos tableros que no son el objetivo (nótese que eso no sucedía en ninguna de las dos heurísticas anteriores).

d) La cuarta posibilidad que consideramos es definir como función heurística la suma de las funciones definidas en (a) y (c). Es la heurística más fina, pero también la que requiere un mayor esfuerzo de cálculo.

En la Figura 1 se recoge una ilustración de estas heurísticas en un caso concreto.

Ejercicio. Otra heurística posible para el 8-puzzle, si el estado objetivo es el recogido en la figura 1, es la siguiente: h(e) = 3 * seq(e), donde seq(e) cuenta 1 si hay un dígito central en e y 2 por cada dígito x no central que no es seguido (en el sentido de la agujas del reloj) por su sucesor x+1 (imponemos por convenio que 8 + 1 = 1). Calcular el valor de esta heurística para los estados que aparecen en la figura 1. Generalizar esta idea para estados objetivos cualesquiera.

Introducimos a continuación un nuevo problema:

- El problema del tren.
    - Entrada: un viajante se encuentra en una capital de provincia.
    - Salida: quiere viajar a otra capital.
    - Medios: hay un tren entre capitales de provincia colindantes; se dispone de un mapa con la disposición de las provincias y sus "coordenadas" en kilómetros respecto al "centro" (por ejemplo, Madrid, con coordenadas (0,0)).

Una función heurística para ese problema consiste en asignar a cada estado un valor que es la distancia en línea recta con el estado objetivo. Dicha distancia es la distancia euclídea entre las coordenadas de dos ciudades.

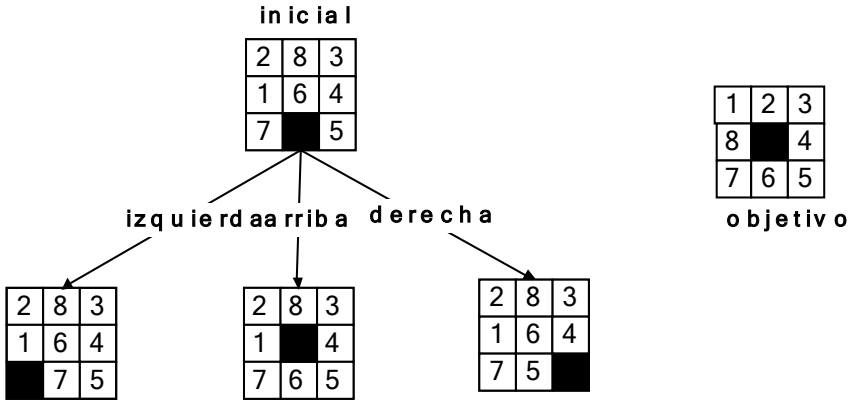
Ejercicios.

1) Escribir un mapa "pequeño".

2) Elegir una representación para el problema (al menos en el nivel conceptual). Dibujar el espacio de estados para el mapa de (1), con una elección de estado inicial y estado objetivo.

3) Calcular los valores heurísticos de cada una de las ciudades del mapa, con la elección de (2).

4) Dibujar los árboles de búsqueda en anchura y en profundidad (¿en árbol o en grafo?) para el espacio de estados anterior.



2 8 3 1 6 4 7 5	6	5	0	6
2 8 3 1 4 7 6 5	4	3	0	4
2 8 3 1 6 4 7 5	6	5	0	6
	h_a	h_b	h_c	h_d

Figura 1

4.2. Métodos de escalada.

Los métodos de escalada (o de ascensión a la colina) trabajan, en su versión más habitual, con funciones de evaluación en las que los valores superiores son preferibles. De ahí su nombre: se trata de elegir en cada paso un estado cuyo valor heurístico sea mayor que el del estado activo en ese momento. Los métodos de escalada se dividen en dos familias:

- Los métodos irrevocables (no preven la vuelta a un lugar del espacio de estados si algún camino elegido resulta inadecuado).

- La escalada en profundidad (es una variante de la búsqueda en profundidad, en la que la heurística dirige la búsqueda, pero que sigue manteniendo el carácter exhaustivo).

En los métodos irrevocables cada estado genera a lo más un nuevo estado, de modo que ABIERTOS nunca posee más de un elemento. Esto conlleva un gran ahorro en espacio, pero hace que sólo se mantenga en expectativa un único camino en la búsqueda de la solución.

El modo en que se determina qué estado es el que sigue a uno dado, da lugar a dos variantes de los esquemas en escalada irrevocables:

- *La escalada simple*. En este método, en el momento es que se encuentra un estado E que es más favorable que el que se está expandiendo, dicho estado E es devuelto sin generar el resto de estados hijos.
- *La escalada por la máxima pendiente*. En este caso, se generan todos los hijos de un estado, se calculan sus valores heurísticos y se determina uno de los estados de mejor valor heurístico; se compara dicho valor con el de el estado expandido y si es mejor, se devuelve ese estado como expansión.

En ambos casos, si ningún estado hijo es mejor que el estado que está siendo expandido, no se devuelve nada, lo que conllevará que la búsqueda termine sin haber encontrado un objetivo. Nótese que la escalada simple es mucho más dependiente que la escalada por la máxima pendiente del orden de disparo de las reglas (pese a que ésta última también lo es: el valor heurístico mejor puede ser alcanzado en varios hijos y, en ese caso, el método no dice nada sobre qué estado elegir).

Ejercicios.

1) Supóngase que se dispone de una función `(defun expandir-estado (estado) …)` que devuelve una lista de estados (tal vez vacía) con los estados generados a partir de un estado dado. Y también de otra función `(defun funcion-heuristica (estado) …)` que asocia a cada estado un número positivo o cero que mide lo cerca que está el estado argumento de un estado objetivo (esto es, a valores más pequeños les corresponden estados preferibles). Prográmese `(defun expandir-nodo (nodo) …)` de dos modos diferentes de manera que se implemente:

a) una escalada simple;

b) una escalada por la máxima pendiente.

Supóngase que se dispone de operaciones `(defun crea-nodo (estado padre) ...)`, `(defun estado-de-nodo (nodo) ...)` y `(defun padre-de-nodo (nodo) ...)` para manipular los nodos. (Nótese que la función `expandir-estado` desvirtúa ligeramente la idea de la escalada simple, puesto que se generan todos los hijos previamente a la selección; hemos preferido proponer de este modo el ejercicio para darle mayor genericidad.)

2) Particularícense los ejercicios anteriores para el problema del tren. Prográmese y háganse comparaciones con las dos búsquedas irrevocables.

3) Lo mismo que en el ejercicio anterior, pero para el problema del 8-puzzle. Compárense además los resultados con las distintas heurísticas presentadas en el apartado 4.1.

4) Como hemos indicado antes, se puede mantener para las escaladas irrevocables la misma estructura del programa `busqueda` que hemos utilizado hasta ahora. Sin embargo, es más natural modificar la estructura del programa para que no se utilice la lista ABIERTOS (no es necesaria, puesto que a lo más contiene a un elemento). Diséñese el algoritmo correspondiente y prográmese en Common Lisp. Obsérvese que sigue teniendo sentido la distinción "búsqueda en árbol" / "búsqueda en grafo". Prográmense pues las dos versiones.

Las propiedades de los algoritmos en escalada irrevocables son las siguientes:
- Tienen una complejidad constante en espacio.
- Su complejidad en tiempo, en el peor caso, continúa siendo exponencial.
- No garantizan que se encuentre el camino más corto (ni, en general, el de menor coste).
- No son métodos completos: pueden no encontrar la solución aunque exista.

Las dificultades principales que pueden aparecer y que conllevarán la "pérdida" de las soluciones son las siguientes:
- Máximo local (en nuestro contexto, sería preferible hablar de *mínimo* local, pero entonces se perdería el símil con el "alpinismo"): todos los hijos de un estado son menos prometedores que él, pero no se trata de un estado objetivo. Por ejemplo, supongamos que en el 8-puzzle elegimos la heurística que cuenta el número de dígitos mal situados. La experiencia (¡heurística!) nos muestra que en ocasiones es necesario descolocar algún dígito para alcanzar el objetivo; las escaladas irrevocables no recogerán dicho conocimiento.
- Mesetas: todos los estados hijos (y tal vez los descendientes) tienen el mismo valor heurístico. Si tomamos la desigualdad estricta en la escalada, eso

significará que el algoritmo terminará sin encontrar solución. Si a igual valor heurístico permitimos seguir, una meseta significa que el valor heurístico no nos proporciona ninguna información y nos encontramos con una búsqueda ciega no exhaustiva que, por supuesto, es bastante inoperante en general (si se trata de una búsqueda en árbol y el espacio de estados es un grafo es posible que se caiga en bucles infinitos).

- Crestas: existe una única dirección en la que el valor heurístico va siendo cada vez mejor (o igual), pero en esa dirección no se encuentra ningún estado objetivo. Puede terminar en un máximo local, o tener un efecto muy similar al de la meseta.

Podemos decir que los métodos de escalada irrevocables limitan sensiblemente el espacio de búsqueda, pero no ofrecen garantías de resultar eficaces.

Una variante que permite recuperar las características de completitud y exhaustividad consiste en mezclar las ideas de la escalada con la búsqueda en profundidad. Recordemos que en la búsqueda en profundidad (y en la búsqueda en anchura) los nodos de igual profundidad se ordenan arbitrariamente. En el método de escalada en profundidad los nodos de igual profundidad son ordenados poniendo al comienzo los más prometedores (eso significa que, siguiendo nuestras convenciones, los nuevos nodos serán ordenados crecientemente).

Ejercicios.

1) Siguiendo los mismos convenios de los últimos ejercicios enunciados, defínase `expandir-nodo` de modo que se implemente la búsqueda de escalada en profundidad. Escríbanse las versiones en árbol y en grafo.

2) Compárese el proceso de búsqueda en profundidad, en anchura, en escalada simple, en escalada por la máxima pendiente y en escalada en profundidad en el problema del tren, con un ejemplo concreto.

3) Mismo ejercicio que (2), pero con el 8-puzzle y con las distintas heurísticas.

Con esta nueva versión, la búsqueda heurística vuelve a ser exhaustiva y completa: si existe solución encuentra una, aunque no necesariamente una de longitud mínima. Por otra parte, vuelve a requerir tanto espacio, en el peor caso, como la búsqueda en profundidad. La única diferencia con ella es que la dirección de la búsqueda es regida por el valor heurístico de los *nuevos* nodos. En cuanto a la comparación con las escaladas irrevocables, la versión de escalada en profundidad siempre supera los máximos locales, las mesetas y las crestas. Sin embargo, al

igual que todos los métodos de escalada, se aprovecha la información heurística de un modo local (se dice que son métodos de visión corta). En particular, puede suceder que todos los descendientes de un estado prometedor sean nefastos, con lo que en ABIERTOS puede haber nodos mucho mejores que los que están al comienzo de la lista. Para evitar estas situaciones se introducen la siguiente familia de búsquedas heurísticas.

4.3. Métodos "primero el mejor".

Los métodos primero el mejor (o en primer lugar el mejor) mantienen la lista de nodos ABIERTOS ordenada de modo que se puedan considerar siempre la alternativa más prometedora de entre todas las que están siendo consideradas. Sus características combinan las de la búsqueda en profundidad (puede encontrar una solución sin expandir todos los nodos), la búsqueda en anchura (no queda atrapada en caminos sin salida) y los métodos de escalada (puesto que tiene en cuenta los valores heurísticos de los estados).

Concretamente, la estrategia de control primero el mejor consiste en darle a la lista ABIERTOS una estructura de cola de prioridad, ordenada (de menor a mayor) por los valores heurísticos de los estados asociados a los nodos. Habrá versiones en árbol y en grafo (es decir, en ésta última se mantiene una lista de *estados* CERRADOS).

En el árbol de estados de la figura 2 hemos escrito entre paréntesis al lado de cada estado el (supuesto) valor heurístico del estado. Como viene siendo habitual, el estado inicial es I y el único estado objetivo es O. En las figuras 3, 4, 5 y 6 recogemos, respectivamente, los árboles de búsqueda en escalada simple (con expansión de izquierda a derecha), escalada por la máxima pendiente, escalada en profundidad y primer lugar el mejor.

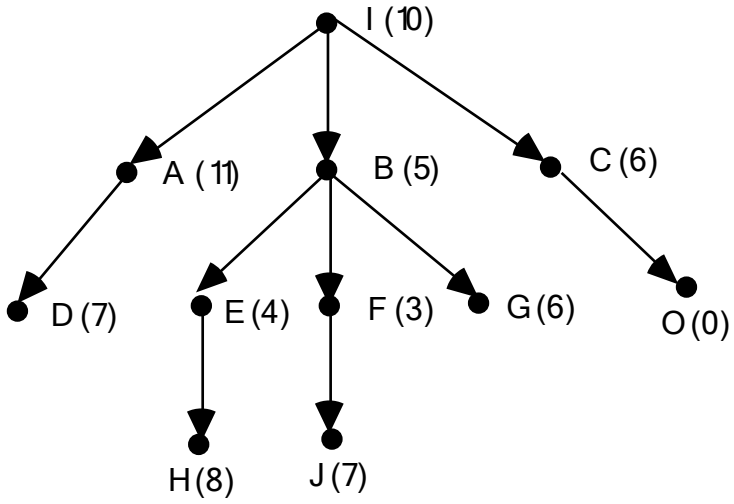


Figura 2

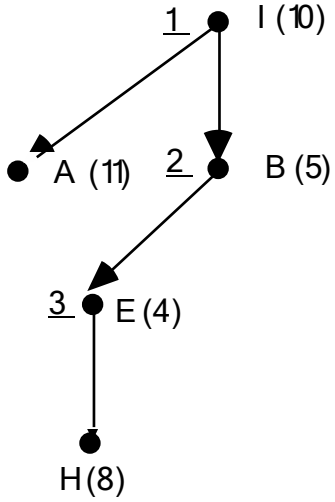


Figura 3

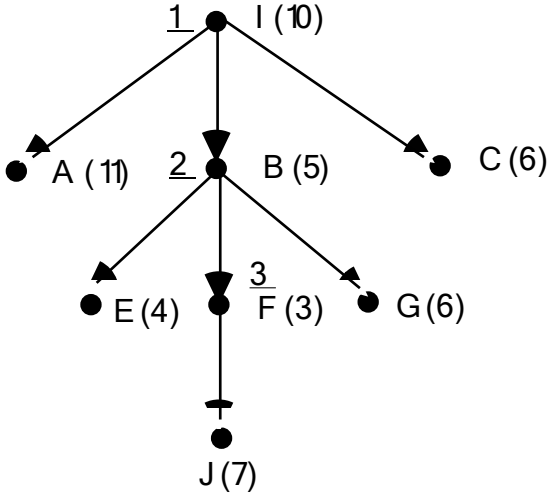


Figura 4

Las flechas punteadas en las figuras 3 y 4 significan estados que han sido generados y de los que se ha calculado su valor heurístico, pero que han sido desechados (no han llegado a ser incluidos en ABIERTOS).

En el árbol de búsqueda de la figura 6, se produce un empate entre los estados G y C. Hemos elegido (arbitrariamente) el estado más profundo (el que fue generado el último), decisión que, en este ejemplo concreto, ha retrasado ligeramente el hallazgo del estado objetivo.

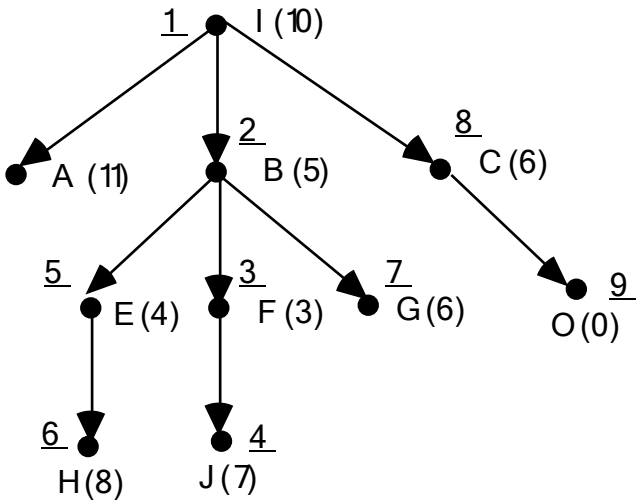


Figura 5

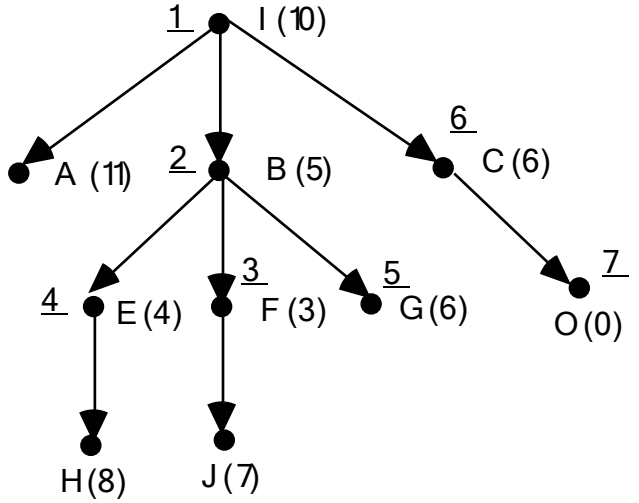


Figura 6

Ejercicios.

1) Dibújense los árboles de búsqueda en anchura y en profundidad, con expansión de izquierda a derecha, para el espacio de la figura 2 (sin tener en cuenta, por tanto, los números que aparecen entre paréntesis). Compárense, en cuanto al número de estados generados y el número de examinados, los distintos árboles de búsqueda. ¿Qué sucede si se expande, en las 6 estrategias, de derecha a izquierda?

2) Existe otra estrategia intermedia entre las escaladas irrevocables y la escalada en profundidad. Consiste en añadir a ABIERTOS como nuevos nodos sólo aquellos que corresponden a estados cuyo valor heurístico mejora el del estado que está siendo expandido. Dibujar el árbol de búsqueda con el espacio de la figura 2. Estudiar su implementación (basta con modificar `expandir-nodo`). ¿Puede aplicarse una idea de ese tipo para encontrar una variante de primero el mejor?

En cuanto a la complejidad algorítmica, su comportamiento en el peor caso es el mismo que el de la búsqueda en profundidad, tanto en tiempo como en espacio. Es un método exhaustivo y completo; esto es, encontrará una solución si existe alguna, aunque no garantiza que dicha solución sea de longitud mínima.

En lo referente al aprovechamiento de la heurística, es claro que el método primero el mejor es mucho más conveniente que los métodos de escalada (usa la información de un modo global y no de un modo local), pero hay que tener el cuenta el coste en tiempo que requiere el mantener ordenada la lista de ABIERTOS (o, equivalentemente, el coste de acceder a ella calculando el mínimo de los valores heurísticos; ver más abajo).

En cuanto a la implementación, varias soluciones son posibles:

- modificar sólo `reorganizar-nodos-a-expandir` para que ordene en cada ocasión la lista ABIERTOS;
- modificar también `expandir-nodo` de manera que devuelva ya ordenados los nuevos nodos (en este caso, sería exactamente la misma función que la necesaria para la búsqueda de escalada en profundidad); aprovechando que ABIERTOS ya está ordenada, se puede implementar `reorganizar-nodos-a-expandir` con un algoritmo de mezcla, en lugar de ordenación, con el consiguiente ahorro en tiempo;
- se podría modificar el acceso a ABIERTOS, de modo que en lugar de extraer siempre el primer elemento de la lista, se extraiga el nodo cuyo estado tenga el valor heurístico más favorable; si se adopta esta solución no sería necesario

modificar los subalgoritmos `reorganizar-nodos-a-expandir` y `expandir-nodo`, pero nos veríamos obligados a variar el programa genérico `busqueda`.

Ejercicios. Implementar las tres ideas anteriores. Estúdiese qué sucede con las versiones en árbol y en grafo.

Para terminar este tema comentamos una familia de variantes de los métodos de búsqueda heurística vistos hasta ahora. Se trata de los métodos de *búsqueda en haz* o en rayo (el símil proviene la búsqueda con una linterna, en la que el haz de luz sólo ilumina una parte del espacio de búsqueda). En esta colección de métodos, se fija un límite, sea n, que va a ser utilizado para podar de una manera rígida el espacio de estados. Según en qué momento se utilice el límite n tenemos variantes diferentes de la búsqueda en haz.

En primer lugar, el límite n puede ser interpretado como un *límite en anchura*. Esto significa que al expandir un nodo, nos quedamos tan solo con los n nuevos nodos más favorables (o con todos los que haya, si se generan menos de n). En este caso, todavía podemos elegir entre situar esos elementos al comienzo de ABIERTOS (con lo que tenemos una variante de la búsqueda de escalada en profundidad) o reordenar toda la lista de ABIERTOS (variante de la búsqueda primero el mejor). En ambos casos, si n = 1 reencontramos la búsqueda irrevocable de escalada por la máxima pendiente.

Otra variante en la misma línea consiste en interpretar el límite n como la longitud máxima en la lista ABIERTOS, descartando los nodos menos prometedores. En esta versión, se incluyen, en primer lugar, todos los nuevos nodos y, en segundo lugar, dejamos en ABIERTOS sólo n nodos. Es también una variante de la búsqueda primero el mejor (pero distinta de la anterior). En este caso, sería apropiado utilizar un vector para almacenar ABIERTOS en lugar de una lista. Si la longitud máxima permitida es 1 no hay diferencia con las búsquedas anteriores y, por tanto, volvemos a encontrar la búsqueda irrevocable de escalada por la máxima pendiente.

La ventaja de los métodos en haz es que reducen drásticamente el espacio de búsqueda. Sin embargo, se pueden perder soluciones (se trata de métodos incompletos).

Ejercicios. Estúdiense los modos de implementación de las distintas búsquedas en haz y compárese su comportamiento en ejemplos concretos (problema del tren, 8-puzzle) para diferentes valores del límite n.

TEMA 5. Búsqueda de una solución óptima

5.1. Estrategias no informadas para la búsqueda de una solución óptima.

Comenzamos planteando el problema a resolver. Este tipo de estrategias aparece cuando en el sistema de producción el disparo de las reglas tiene asociado un *coste*, que, en lo que sigue, será un número (real o natural) estrictamente mayor que cero. Esta condición es equivale a que en el espacio de estados cada arista tenga asociado como peso un coste numérico mayor que cero. Llamaremos *coste de un camino* en el espacio de estados a la suma de los costes de las aristas que lo constituyen. Como es habitual, la longitud de un camino es su número de aristas. Si el coste de cada arista es 1, coste y longitud coinciden. Este es el caso en sistemas de producción para los que ninguna noción de coste es adecuada, pero para los que queramos aplicar las estrategias de búsqueda de una solución óptima.

El problema consiste pues en encontrar una *solución óptima*, es decir un camino solución cuyo coste sea mínimo entre los de los caminos solución. A un tal camino, se le denomina *camino solución óptimo* y a su coste, *coste óptimo*.

Por ejemplo, en el problema del tren introducido en el apartado 4.1 podríamos estar interesados no sólo en encontrar un trayecto entre la ciudad de partida y la de llegada, sino en encontrar uno de duración mínima. En ese caso el peso de cada arista debería ser la duración del viaje. Si estuviésemos interesados en minimizar el número de kilómetros, entonces deberíamos disponer para cada arista del número de kilómetros. Por último, si nos interesase tan solo encontrar el trayecto con el mínimo número de trasbordos, daríamos a cada arista el peso 1.

Una primera estrategia para resolver el problema planteado consiste en mantener la lista de ABIERTOS ordenada, pero en lugar de hacerlo respecto a los valores heurísticos como sucedía en "primero el mejor", se ordena respecto al coste de los caminos definidos por los nodos (apoyándonos en la información relativa al `padre`). Denominaremos a las estrategias que llevan a cabo esta idea "primero el menos costoso". Se tratará de estrategias que permitirán calcular soluciones de coste

óptimo, pero que van a ser tan ineficaces (en tiempo y en espacio) como la búsqueda en anchura. De hecho, si el coste de cada arista es una constante fija k (antes hemos comentado el caso k = 1), tendremos que estas estrategias son equivalentes (se examinan y expanden los mismos estados y en el mismo orden) a la búsqueda en anchura. Por otra parte, si el coste es constante y ordenamos ABIERTOS de mayor a menor, obtendremos un comportamiento equivalente al de la búsqueda en profundidad.

En cuanto a la implementación, supondremos conocida una función

```
(defun coste-arista (estado1 estado2) …)
```

que calcula el coste de la arista que se supone que existe de `estado1` a `estado2`. En estas condiciones, si los nodos tienen al menos información relativa al `padre` (la información relativa al `estado` es obligatoria), se puede calcular el coste del camino "contenido" en un nodo a partir exclusivamente del nodo. Sin embargo, para evitar el recálculo es mucho mejor añadir una nueva información a los nodos que almacene el coste del camino de ese nodo, calculando su valor conforme se va desarrollando el proceso de búsqueda (Esta misma idea podía haber ser utilizada en "primer lugar el mejor", almacenando en los nodos el valor heurístico del estado correspondiente; esta idea es explotada en los algoritmos A del siguiente apartado). Habitualmente se denomina `g` a esta nueva información (como se llamaba `h` al valor heurístico). En general, el valor de `g` dependerá del nodo (es decir, del modo de llegar a un estado) y no del estado en sí. En el caso de que el espacio de estados sea un árbol, sólo hay (a lo más) un camino que une el estado inicial con cualquier otro estado y así el valor de `g` depende del estado únicamente. Esta diferencia es esencial y, por tanto, va a haber dos estrategias diferentes según se considere que el espacio de estados es un árbol o un grafo.

En el caso de la estrategia "primero el menos costoso" en árbol, las modificaciones a realizar en la implementación son las siguientes:

1) Al crear el nodo inicial se actualiza la información `g` poniendo el valor 0.

2) La función `reorganizar-nodos-a-expandir` ordena ABIERTOS de menor a mayor valor de `g` (aquí se pueden aplicar tres estrategias diferentes, como sucedía en "primer lugar el mejor").

3) La función `expandir-nodo` puede ser expresada en notación algorítmica del siguiente modo:

   Expandir-nodo (n) --> lista nodos

e := estado-de-nodo (n)

(e_1, ..., e_r) := expandir-estado (e)

lista-nuevos-nodos := ()

Desde i = 1 hasta r hacer

Construir un nodo n_i con informaciones:

estado de n_i : e_i

g, coste de n_i : coste-arista (e,e_i) + valor de g en el nodo n

padre de n_i : n

Añadir el nodo así construido a la lista-nuevos-nodos

devolver (lista-nuevos-nodos)

Veamos un ejemplo artificial del funcionamiento de la estrategia "primero el menos costoso" en árbol. Consideremos el árbol de estados de la figura 1, en el que los estados son letras, el estado inicial es I y los dos únicos estados objetivos son O1 y O2. El coste de las aristas ha sido anotado al lado de cada una de ellas.

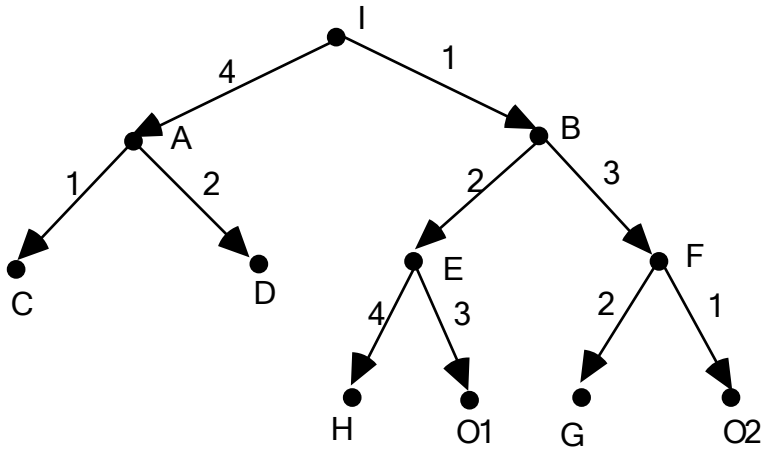


Figura 1

El árbol de búsqueda ha sido recogido en la figura 2, en la que aparecen todos los estados generados (todos en este ejemplo), la información `g` aparece al lado de los estados (la información `padre` está implícita por tratarse de un *árbol* de búsqueda) y los nodos han sido numerados (con números subrayados) según van siendo explorados (así, un estado no numerado quedará en un nodo de ABIERTOS y un estado numerado, y no objetivo, quedará en CERRADOS). En caso de conflicto (nodos con igual valor del campo `g`), elegimos un nodo cualquiera (en el ejemplo hemos elegido el que hace terminar más rápidamente el proceso de búsqueda).

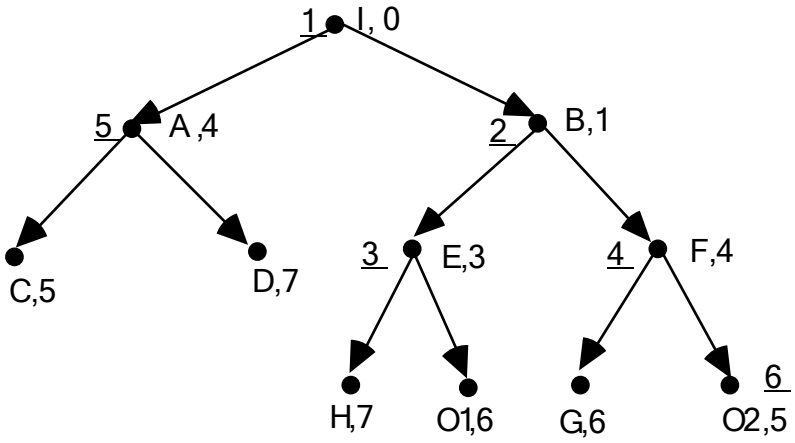


Figura 2

Como es habitual, esta estrategia de control también funcionará correctamente para espacios de estados con ciclos, pero se perderá tiempo y espacio al re-examinar y re-expandir ciertos estados. En el ejemplo de la figura 3 puede verse que existen 4 caminos solución: uno de I a O1 (coste 7) y tres de I a O2 (de costes 7, 6 y 7). En el árbol de búsqueda para "primero el menos costoso" en árbol que aparece en la figura 4 puede verse que ciertos estados son re-expandidos (notar que en este caso, el árbol de búsqueda no puede ser un subgrafo del espacio de estados).

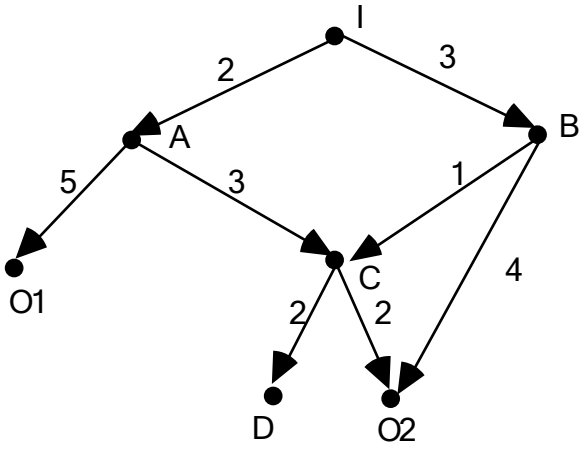


Figura 3

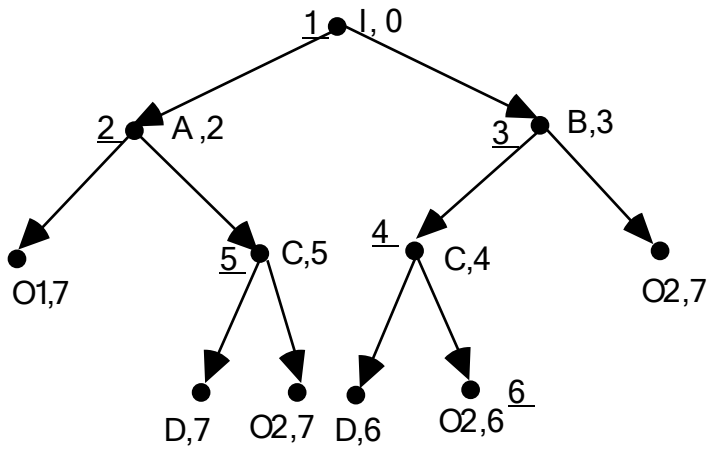


Figura 4

Una primera idea para evitar ese recálculo sería hacer exactamente la misma adaptación que se realizó para pasar de árbol a grafo en estrategias como profundidad, anchura, etc. Es decir, solamente añadimos la gestión de una lista de estados CERRADOS y no expandimos los estados que ya aparecen en ABIERTOS o que están en CERRADOS. En la figura 5 aparece el árbol de búsqueda que correspondería con la estrategia así esbozada al espacio de estados de la figura 3 (el árbol de búsqueda vuelve a ser un subgrafo del espacio de estados).

Como puede observarse, sea cual sea el nodo de coste 7 que elijamos, habremos perdido el camino solución óptimo. La razón es que las aristas que aparecen punteadas no llegan a ser consideradas. La solución consiste en considerarlas, pero no para dar lugar a nuevos nodos (como sucedía en la estrategia en árbol), sino para revisar los valores `g` que ya aparecían y modificar la información `padre` en caso de que el nuevo camino sea menos costoso que el que había sido elegido hasta ese momento.

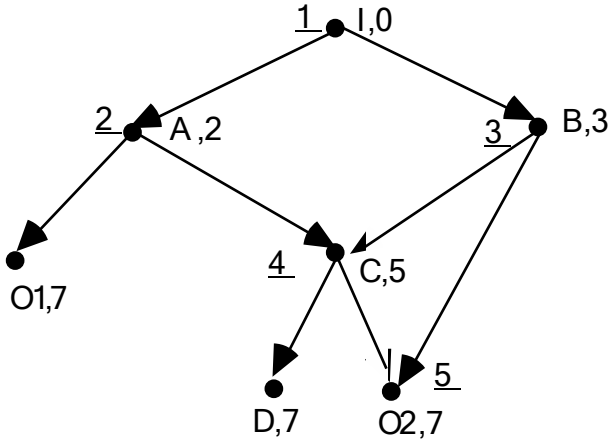


Figura 5

Así, para obtener una versión de "primero el menos costoso" en grafo que asegure el hallazgo de un camino solución óptimo debemos modificar `expandir-nodo` del siguiente modo:

Expandir-nodo (n,ABIERTOS,CERRADOS) --> lista nodos
e := estado-de-nodo (n)
(e_1, ..., e_r) := expandir-estado (e)
lista-nuevos-nodos := ()
Desde i = 1 hasta r hacer
    coste-auxiliar := coste-arista (e,e_i) + valor de g en el nodo n
    Si e_i aparece en un nodo n' de ABIERTOS
        entonces Si coste-auxiliar < valor de g en el nodo n'
                entonces modificar (destructivamente) en n' las informac.
                        padre : n
                        g : coste-auxiliar
                    (estas modificaciones alteran ABIERTOS)
                si no : no hacer nada
        si no Si e_i no está en CERRADOS
                entonces Construir un nodo n_i con informaciones:
                        estado de n_i : e_i
                        g, coste de n_i : coste-auxiliar
                        padre de n_i : n
                    Añadir el nodo así construido a la lista-nuevos-nodos
devolver (lista-nuevos-nodos)

Es muy importante notar que `expandir-nodo` pasa de ser una función a ser un procedimiento funcional, puesto que los resultados de su evaluación pueden dar lugar a dos procesos: modificación (en ocasiones) del parámetro ABIERTOS y (siempre) devolución de una lista de nodos nuevos (eventualmente vacía). Las características de Common Lisp hacen que al modificar (destructivamente) alguna variable generalizada interna de un elemento que sea miembro de una lista, ésta quede automáticamente actualizada, por lo que implementando de modo natural el anterior algoritmo obtendremos el resultado buscado. (Con "modificar destructivamente" nos referimos a un proceso que no reconstruya un nuevo nodo, sino que modifique la estructura interna, las variables generalizadas, del nodo que

está siendo modificado.) Una consecuencia de esta característica es que ABIERTOS puede quedar desordenada tras una evaluación de `expandir-nodo`. Por tanto, la idea de programar `reorganizar-nodos-a-expandir` como un algoritmo de mezcla de los nuevos nodos y ABIERTOS no es aplicable (al menos directamente).

Observemos también que siempre se revisa en ABIERTOS y nunca en CERRADOS, de modo que esta segunda lista puede seguir siendo una lista de estados (y no de nodos). La razón es que una vez que se expande un nodo, ya se ha encontrado la mejor manera (camino óptimo) de llegar al estado de ese nodo (esto es un resultado teórico que señalamos sin demostración).

Para el espacio de estados de la figura 3, el árbol de búsqueda "primero el menos costoso" en grafo queda recogido en la figura 6, en la que las flechas punteadas no significan aristas no consideradas, sino aristas que han formado parte del árbol de búsqueda en algún momento y que después han sido desechadas al encontrarse otros caminos mejores. Del mismo modo, los costes tachados corresponden a costes que han sido mejorados.

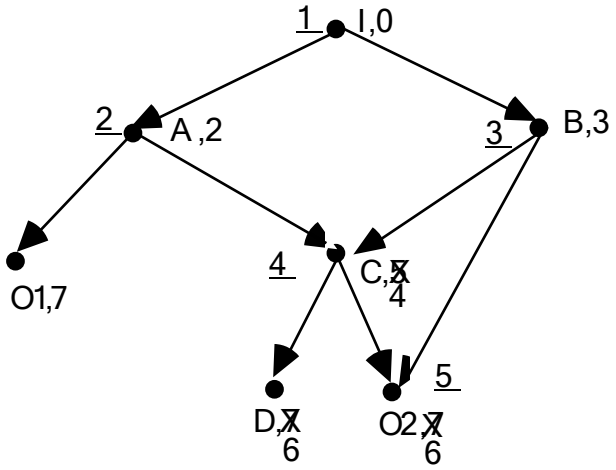


Figura 6

Ejercicios. Escribir programas Common Lisp que correspondan a las dos estrategias explicadas en este apartado. Escribir programas para resolver con ellos el problema del tren, con las tres posibilidades para los costes explicadas más arriba.

5.2. Estrategias heurísticas: algoritmos A.

Consideremos de nuevo el problema del tren. Supongamos que disponemos de las "coordenadas" de cada una de las ciudades y, para cada trayecto entre dos ciudades adyacentes, del número de kilómetros que realiza el tren en ese trayecto. Situémonos en un momento intermedio del proceso de búsqueda. Para decidir cuál es el siguiente paso a dar, es decir cuál es la siguiente ciudad a la que viajar, podríamos utilizar varios criterios:

a) Elegir una cualquiera, sin tener en cuenta el conocimiento adicional del que disponemos (coordenadas, costes). Es esencialmente lo que hacen las búsquedas no informadas en anchura o profundidad. No se garantiza que se vaya a obtener un camino solución óptimo ni tampoco que la búsqueda esté dirigida por ninguna otra consideración.

b) Tener en cuenta la estimación (heurística) de la cercanía dada por la distancia en línea recta a la ciudad objetivo. Esto es lo que hacen las escaladas o "primero el mejor". En estas aproximaciones no se considera la distancia previamente recorrida (es decir, la información acerca del coste de las aristas). La búsqueda estará dirigida por cierto conocimiento específico, aunque no totalmente seguro, pero no se garantiza el hallazgo de una solución óptima.

c) Ir calculando el coste de los caminos que van siendo construidos, eligiendo en cada paso el menos costoso. Se trata de las estrategias explicadas en el apartado anterior. Aseguran el hallazgo de una solución óptima, pero, al no tener en cuenta el conocimiento heurístico, pueden resultar tan ineficaces como la búsqueda en anchura.

En este apartado, estudiamos la posibilidad de integrar las aproximaciones (b) y (c) anteriores, para obtener estrategias que encuentren soluciones óptimas, pero de un modo más eficaz que "primero el menos costoso". Es lo que se conoce en la literatura con el nombre de Algoritmos A, nominación ciertamente poco descriptiva.

Para esta familia de algoritmos los nodos han de disponer (al menos) de cuatro informaciones: `estado`, `padre`, `g` (con el mismo significado y modo de cálculo que en "primero el menos costoso") y `h` (valor heurístico, que sólo depende del estado y que puede ser calculado por una `funcion-heuristica` como en "primero el mejor").

Más concretamente, vamos a suponer que interactuaremos con los nodos por medio de las siguientes operaciones.

Una operación de construcción:

```
(defun crea-nodo (estado padre g h) ...) --> nodo
```

Cuatro operaciones de acceso:

```
(defun estado-de-nodo (nodo) ...) --> estado
(defun padre-de-nodo (nodo) ...) --> nodo o Nil
(defun g-de-nodo (nodo) ...) --> número >= 0
(defun h-de-nodo (nodo) ...) --> número >= 0
```

Estas cinco operaciones tienen el significado evidente. Además necesitaremos dos operaciones de modificación (que ya eran necesarias en la estrategia "primero el menos costoso" en grafo del apartado 5.1):

`(defun modifica-padre (nodo nuevo-padre) ...)` --> *indefinido*

`(defun modifica-g (nodo nuevo-g) ...)` --> *indefinido*

Hemos especificado que los valores devueltos por las dos operaciones de modificación quedan "indefinidos", porque estas dos operaciones deben ser utilizadas para obtener un efecto lateral: la modificación destructiva de la información `padre` (`o g`, respectivamente) que estaba asociada al nodo que aparece como primer argumento. Obsérvese que no tiene sentido incluir operaciones de modificación para el estado de un nodo (en esencia, el estado es lo que caracteriza al nodo) ni para la información `h` (puesto que estamos suponiendo que la función heurística sólo depende del estado y no del modo de llegar a él).

Con esta estructura para los nodos, los cambios en los algoritmos son los siguientes:

a) En el esquema genérico de búsqueda (en grafo), CERRADOS tiene que ser una lista de nodos (y no de estados).

b) En `reorganizar-nodos-a-expandir`, ABIERTOS se ordena por el valor de `g` + `h` (= `f`, según la terminología habitual). Esta es la razón por la que CERRADOS debe ser una lista de nodos: puesto que ABIERTOS es ordenada apoyándose en una información que no es segura (la heurística `h`), podría suceder que un nodo fuese extraído de ABIERTOS antes de haber encontrado el mejor modo de llegar al estado correspondiente y, por tanto, que los costes deban ser revisados en CERRADOS además de en ABIERTOS.

c) Por lo anterior, el algoritmo de expansión de un nodo queda como sigue:

```
Expandir-nodo (n,ABIERTOS,CERRADOS) --> lista nodos
e := estado-de-nodo (n)
(e1, ..., er) := expandir-estado (e)
lista-nuevos-nodos := ()
Desde i = 1 hasta r hacer
   coste-auxiliar := coste-arista (e,ei) + valor de g en el nodo n
   Si ei aparece en un nodo n' de ABIERTOS
      entonces  Si coste-auxiliar < valor de g en el nodo n'
                   entonces modificar en n' las informaciones
                                padre : n
                                g : coste-auxiliar
                             (estas modificaciones alteran ABIERTOS)
                   (si no : no hacer nada)
      si no Si ei aparece en un nodo n' de CERRADOS
                entonces  Si coste-auxiliar < valor de g en el nodo n'
                             entonces modificar en n' las informaciones
                                          padre : n
                                          g : coste-auxiliar
                                       eliminar n' en CERRADOS
                                       añadir n' en lista-nuevos-nodos
                             (si no : no hacer nada)
                si no Construir un nodo ni con informaciones:
                             estado de ni : ei
                             padre de ni : n
                             g, coste de ni : coste-auxiliar
                             h: valor heurístico de ei
                          Añadir el nodo así construido a la lista-nuevos-nodos
devolver (lista-nuevos-nodos)
```

El anterior algoritmo no es difícil de implementar en Common Lisp, salvo por el punto en el que es necesario "eliminar n' en CERRADOS", puesto que se trata de una modificación destructiva de un parámetro lista (y no de la modificación interna de uno de sus miembros como sucede en el caso de ABIERTOS). Pese a que este paso puede ser programado de muy diversas maneras, preferimos alterar la estructura general de los programas de búsqueda, para tener un conjunto de

funciones adaptadas a los esquemas A (y, con modificaciones menores, a las búsquedas en anchura, profundidad, primero el mejor y primero el menos costoso; tan sólo los esquemas de escalada necesitarían alguna modificación ligeramente mayor para adaptarse a la nueva estructura).

El cambio más importante consiste en mantener una única lista, llamada NODOS, que almacene todos los nodos que van siendo generados en el proceso de búsqueda. Los nodos permitirán ahora acceder al menos a 5 informaciones: `estado`, `padre`, `g`, `h` y `ubicacion`. La nueva información `ubicacion` sólo puede tener como valor los símbolos `ABIERTOS` o `CERRADOS`. Podemos decir que en la lista NODOS van a convivir las dos listas de ABIERTOS y CERRADOS, distinguiéndose sus respectivos miembros por el valor de `ubicacion`.

Será por tanto necesario añadir para los nodos dos nuevas operaciones, una de acceso:

```
(defun ubicacion-de-nodo (nodo) ...) --> ABIERTOS o CERRADOS
```

y otra de modificación:

```
(defun modifica-ubicacion (nodo nueva-ubicacion) ...)
--> indefinido
```

además de cambiar la cabecera de `crea-nodo`:

```
(defun crea-nodo (estado padre g h ubicacion) ...)
--> nodo
```

Una vez comprendida la nueva estructura de almacenamiento, no es difícil modificar el esquema de búsqueda para implementar los algoritmos A. Trascribimos a continuación los fragmentos Lisp fundamentales.

```
;; Esquema generico A:

(defun busqueda ()
  (let ((NODOS (list (crea-nodo-inicial))))
    (do ((el-nodo (selecciona NODOS) (selecciona NODOS)))
        ((or (eq el-nodo NIL) (estado-objetivo? (estado-de-nodo el-nodo)))
         (if (eq el-nodo NIL)
             (solucion-no-encontrada)
           (escribe-solucion el-nodo)))
      (modifica-ubicacion el-nodo 'CERRADOS)
      (setq NODOS (append (expandir-nodo el-nodo NODOS) NODOS))
      (informacion-proceso-busqueda el-nodo))))

; Representacion de los nodos e implementacion de las operaciones:

(defstruct nodo estado padre g h ubicacion)

(defun crea-nodo (estado padre g h ubicacion)
  (make-nodo :estado estado
             :padre padre
             :g g
```

```
             :h h
             :ubicacion ubicacion))

(defun estado-de-nodo (nodo)
  (nodo-estado nodo))

(defun padre-de-nodo (nodo)
  (nodo-padre nodo))

(defun g-de-nodo (nodo)
  (nodo-g nodo))

(defun h-de-nodo (nodo)
  (nodo-h nodo))


(defun ubicacion-de-nodo (nodo)
  (nodo-ubicacion nodo))

(defun modifica-padre (nodo nuevo-padre)
  (setf (nodo-padre nodo) nuevo-padre))

(defun modifica-g (nodo nuevo-g)
  (setf (nodo-g nodo) nuevo-g))

(defun modifica-ubicacion (nodo nueva-ubicacion)
  (setf (nodo-ubicacion nodo) nueva-ubicacion))

; Crea el nodo inicial (las funciones estado-inicial
; y funcion-heuristica dependen del problema a resolver):

(defun crea-nodo-inicial ()
  (let ((estado-inicial (estado-inicial)))
    (crea-nodo estado-inicial  ; estado
               NIL             ; padre
               0               ; g
               (funcion-heuristica estado-inicial)  ; h
               'ABIERTOS)))    ; ubicacion

; Selecciona elige el nodo de menor valor f = g + h de entre los
; que estan en ABIERTOS; si no hay ninguno se devuelve NIL (lo que
; hace que en ocasiones el-nodo no sea tal, pero facilita el control
; del bucle).

(defun selecciona (lista-nodos)
  (do ((nodos lista-nodos (rest nodos))
       (el-nodo NIL))
      ((endp nodos) el-nodo)
    (let ((nodo (first nodos)))
      (if (eq 'ABIERTOS (ubicacion-de-nodo nodo))
          (if (or (eq el-nodo NIL)
                  (< (+ (g-de-nodo nodo) (h-de-nodo nodo))
                     (+ (g-de-nodo el-nodo) (h-de-nodo el-nodo))))
              (setq el-nodo nodo))))))

; Expansion de un nodo (funciones que dependen del problema: expandir-estado,
; coste-arista, estados-iguales?)

(defun expandir-nodo (nodo lista-nodos)
  (let ((estado (estado-de-nodo nodo))
        (lista-nuevos-nodos (list )))
    (do ((estados (expandir-estado estado) (rest estados)))
        ((endp estados) lista-nuevos-nodos)
      (let ((estado-nuevo (first estados)))
        (let ((nodo-viejo (encuentra estado-nuevo lista-nodos))
              (coste-auxiliar (+ (coste-arista estado estado-nuevo)
                                 (g-de-nodo nodo))))
          (if (eq nodo-viejo NIL)
```

```
              (setq lista-nuevos-nodos
                    (cons (crea-nodo estado-nuevo     ; estado
                                     nodo             ; padre
                                     coste-auxiliar   ; g
                                     (funcion-heuristica estado-nuevo) ; h
                                     'ABIERTOS)       ; ubicacion
                          lista-nuevos-nodos))
            (if (< coste-auxiliar (g-de-nodo nodo-viejo))
                (progn (modifica-padre nodo-viejo nodo)
                       (modifica-g nodo-viejo coste-auxiliar)
                       (modifica-ubicacion nodo-viejo 'ABIERTOS)))))))))
                         ; esto ultimo a veces supone cambio y otras no

(defun encuentra (estado lista-nodos)
  (do ((nodos lista-nodos (rest nodos)))
      ((or (endp nodos) (estados-iguales? (estado-de-nodo (first nodos))
                                          estado))
       (if (endp nodos) nil (first nodos)))))
```

Veamos algunos ejemplos del comportamiento del algoritmo A. Consideremos el espacio de estados que aparece en la figura 7, en el que los valores heurísticos de los estados están escritos entre paréntesis.

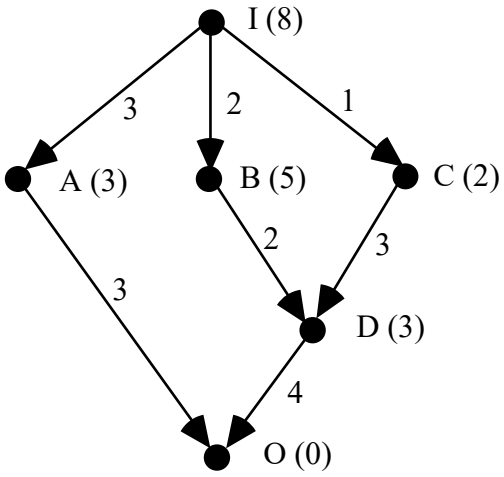


Figura 7

El árbol de búsqueda queda recogido en la figura 8, en la que los números a continuación de los estados corresponden a los valores de `g` y `h`, respectivamente.

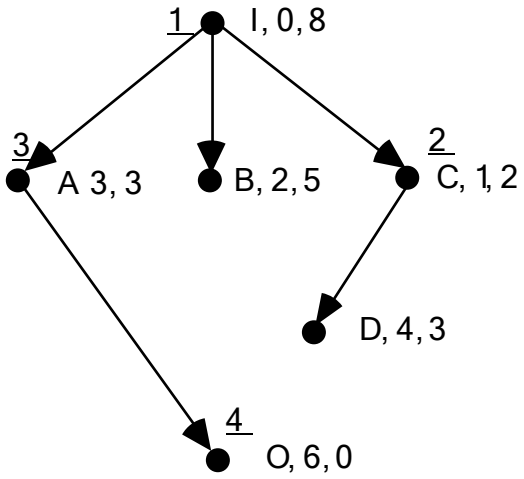


Figura 8

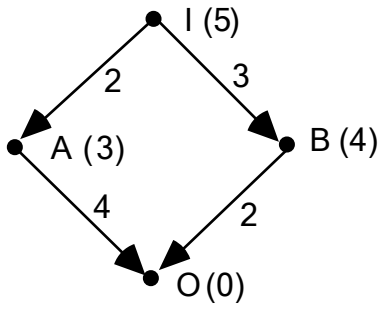


Figura 9

En este ejemplo, la estrategia A ha encontrado un camino solución óptimo, pero, en general, no tiene por qué ser así. Con el espacio de estados de la figura 9 se obtiene el árbol de búsqueda de la figura 10, donde puede comprobarse que el camino solución encontrado no es el óptimo.

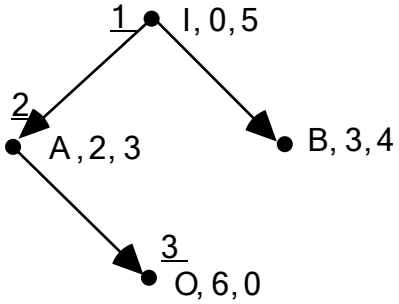


Figura 10

Si nos preguntamos cuál es la razón por la que en este segundo ejemplo la solución óptima no es encontrada, veremos que la causa es que el valor heurístico de B, h(B) = 4, es demasiado grande (se puede ir hasta O con coste 2). Ello hace que la ordenación respecto a f = g + h en ABIERTOS resulte engañosa y el nodo de estado A pase por delante del nodo de estado B, perdiéndose el camino óptimo.

Para generalizar este análisis, introduzcamos algunas nociones teóricas. Si n es un nodo, el valor de g para n, denotado g(n), es el coste del camino "contenido en n" desde el estado inicial e_0 hasta e = estado(n). Puesto que en el algoritmo A se revisan los nodos respecto a g, resulta que g(n) es el coste mínimo entre todos los caminos desde e_0 hasta e *que hayan aparecido hasta ese momento*.

Si e es un estado tal que existe un camino desde e_0 hasta él, definimos:

$$g^*(e) = \text{mínimo } \{\text{coste}(C);\ C \text{ es un camino de } e_0 \text{ a } e\}.$$

Según la observación anterior, se verificará que g(n) >= g*(e) para todo nodo n tal que estado(n) = e.

Por otra parte, si n es un nodo, el valor de `h` sólo depende del estado e = estado(n), por lo que lo denotaremos h(e). Se le denomina *término heurístico*. Se interpreta como una estimación del mínimo coste entre los de los caminos que van desde e hasta algún estado objetivo. Se trata de una función de evaluación estática (que en la implementación Lisp denominamos `funcion-heuristica`) que depende del conocimiento que tengamos sobre el problema a resolver.

Si e es un estado desde el que podemos encontrar algún camino hasta alguno de los estados objetivo, definimos:

h*(e) = mínimo {coste(C); C es un camino de e hasta alguno de los estados objetivo}.

En general, h y h* no son comparables. Es decir, no se verifica que h(e) <= h*(e) para todo e, ni tampoco h(e) >= h*(e) para todo e.

Si e es un estado tal que existe un camino solución que pasa por él, definimos: f*(e) = g*(e) + h*(e), que es el mínimo de los costes de todos los caminos solución que pasan por e. El mínimo de f*(e) para todos los estados e (para los que está definido f*) será el coste de cada una de las soluciones óptimas. En general, f*(e) y f(n) = g(n) + h(e), con e = estado(n), no serán comparables, por no serlo h y h*.

En la mayoría de las ocasiones, las funciones g*, h* y f* serán muy costosas de calcular (en general, requerirán búsqueda), así que deben ser interpretadas como definiciones teóricas que, en principio, no serán calculadas. De hecho, se puede interpretar que la estrategia "primero el menos costoso" va calculando los valores g*(e) hasta encontrar un camino solución óptimo (se calcula por tanto también el mínimo de los f*(e), e implícitamente el valor h* de todos los estados que forman parte del camino solución). Veremos a continuación que esta interpretación no siempre es cierta para los algoritmos A.

Volvamos al ejemplo estudiado en las figuras 9 y 10. Podemos reinterpretar los números que allí aparecen diciendo que h(B) = 4 es mayor que h*(B) = 2, lo que implica que el valor de f = g + h para el nodo que corresponde a B (3 + 4) es mayor que f*(B) = 5. De hecho, el valor de f para el nodo de B es también mayor que el valor de f para el primer nodo que aparece de estado O (6 + 0), por lo que al ordenar ABIERTOS este último nodo pasa por delante del de B (incorrectamente: f*(B) es menor que f del nodo que contiene en ese momento a O).

La situación anterior no podrá producirse si h es una subestimación de h^*. Si la heurística h verifica que h(e) <= h^*(e) para todo estado e (para el que exista h^*) se dice que h es una *heurística admisible*.

La importancia de la anterior noción viene determinada por el siguiente resultado: si un algoritmo A utiliza una heurística admisible, entonces, si existe solución, encuentra una solución óptima.

Una instancia del esquema A en la que la heurística sea admisible se denomina algoritmo A^*. Así, un algoritmo A^* siempre encontrará una solución óptima (si existe alguna).

Obsérvese que si tomamos como heurística la función identicamente nula (h = 0), siempre subestima h^*, es decir, siempre es admisible y, por tanto, A es A^* y siempre encontrará una solución óptima. Pero éste no es un resultado nuevo. Si h = 0, entonces f = g; es decir, la búsqueda está controlada totalmente por la información `g` y en este caso el algoritmo A se comporta como "primero el menos costoso". Veremos más adelante que éste es, en cierto sentido, el caso menos interesante entre los A^*.

Pese a que, evidentemente, la admisibilidad de una heurística tiene importantes repercusiones en el comportamiento del algoritmo A, puede parecer un concepto meramente teórico, puesto que, como ya hemos indicado, el cálculo de h^*(e) no tiene por qué ser sencillo. Sin embargo, para determinar la admisibilidad de una heurística no es necesario calcular con exactitud los valores h^*(e), basta con demostrar que esos números acotan superiormente al valor heurístico y, en ocasiones, esa demostración no es difícil.

Por ejemplo, consideremos el caso del 8-puzzle, con coste de cada arista igual a 1 y con heurística el número de dígitos mal situados. Puesto que para situar un dígito mal situado se requiere al menos un movimiento (más de uno en la mayoría de las ocasiones), resulta que esta heurística es admisible.

Si consideramos el problema del tren, con coste de cada arista igual al número de kilómetros del trayecto y con heurística la distancia (en kilómetros) en línea recta a la ciudad objetivo, es claro que la heurística es admisible (la distancia más corta entre dos puntos es la determinada por la recta que los une). En cambio, si el coste de cada arista mide la duración en horas del trayecto resulta que la heurística no tiene por qué ser, en general, admisible (aunque sigue siendo una heurística ...).

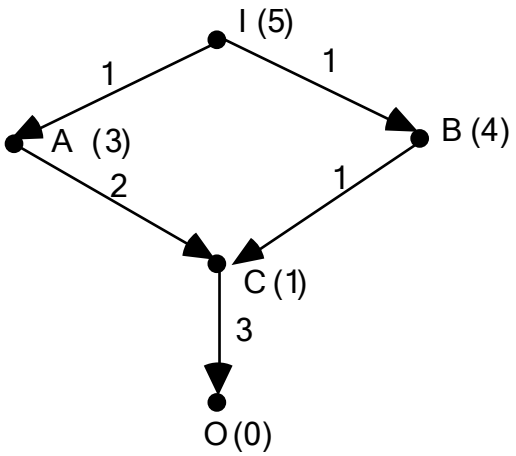


Figura 11

La primera diferencia de comportamiento entre "primero el menos costoso" y el algoritmo A es que éste puede no encontrar una solución óptima (la condición de admisibilidad suprime esta diferencia). La segunda es que en un algoritmo A un estado expandido puede ser re-expandido más adelante. Esto obliga a que CERRADOS sea una lista de nodos y no de estados, como sucedía en todas las estrategias de control anteriores. Veamos en un ejemplo artificial que esta situación puede efectivamente producirse. Al espacio de estados de la figura 11 le corresponde el árbol de búsqueda A de la figura 12. Nótese que, aunque el espacio de estado tiene tan sólo 5 estados, en el árbol de búsqueda son examinados 6 estados: C es examinado (¡y expandido!) dos veces.

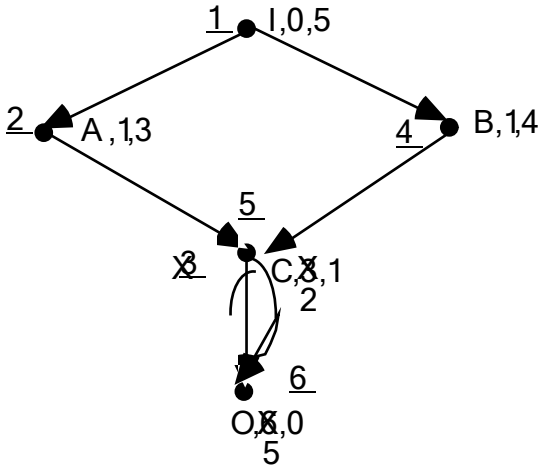


Figura 12

La importancia del paso de un nodo n de CERRADOS a ABIERTOS reside en que *todos* los nodos producidos a partir de n, ya estén en ABIERTOS o en CERRADOS, deberán *obligatoriamente* ser revisados y para aquellos que estén en CERRADOS se lanza un proceso recurrente de revisión y re-expansión, que perjudica

enormemente el número de operaciones a realizar por el algoritmo. Por ello es de gran interés encontrar una propiedad teórica (el equivalente de la "admisibilidad" para este problema) que garantice que los estados nunca serán expandidos más de una vez.

Si nos preguntamos en la figura 11, qué anomalías o incoherencias han podido producir este comportamiento del algoritmo A, veremos que no pueden tener que ver con la admisibilidad de la heurística (en el ejemplo es admisible). En cambio, si observamos localmente la relación entre los costes de cada arista y los valores heurísticos de sus extremos, notaremos ciertas inconsistencias. Por ejemplo, la arista BC tiene coste 1, mientras que el valor heurístico de B es 4 y el de C, 1. Parece claro que o bien B está sobrevalorado o C está infravalorado. La situación inconsistente es recogida en la desigualdad: h(B) > h(C) + coste-arista(B,C).

Si esta situación no se produce diremos que se verifica la *condición de monotonía*. Es decir, h es *monótona* si para cada arista (e_i,e_j) del espacio de estados se verifica que

$$h(e_i) <= h(e_j) + \text{coste-arista}(e_i,e_j).$$

Para comprender mejor porque se denomina condición de monotonía a la desigualdad anterior, observémos que si n_j es un nodo obtenido por expansión de otro n_i y h es monótona, entonces $f(n_i) <= f(n_j)$. En efecto, $f(n_i) = h(\text{estado}(n_i)) + g(n_i) <= h(\text{estado}(n_j)) + \text{coste-arista}(\text{estado}(n_i),\text{estado}(n_j)) + g(n_i) = h(\text{estado}(n_j)) + g(n_j) = f(n_j)$. De hecho, no es difícil comprobar también que si para todo nodo n_i y todo hijo suyo n_j se verifica $f(n_i) <= f(n_j)$, entonces la heurística es monótona.

Nótese que la propiedad de monotonía es *local* a cada una de las aristas, mientras que la condición de admisibilidad se apoya en h^* que requiere un conocimiento más *global* del espacio de estados.

Un resultado teórico asegura que si la heurística es monótona, en la ejecución del algoritmo A ningún estado volverá de CERRADOS a ABIERTOS. Por tanto, si h es monótona no es necesario revisar los nodos que ya están en CERRADOS y el condicional correspondiente en la versión de `expandir-nodo` que hemos dado en seudo-código puede ser eliminado. De lo anterior se sigue que CERRADOS ya no tiene por qué ser una lista de nodos y puede volver a ser una lista de estados. En conclusión, si la heurística es monótona el código que implementa el algoritmo A puede ser exactamente el mismo que el de "primero el menos costoso", salvo que `reorganizar-nodos-a-expandir` ordena ABIERTOS respecto a f = g + h, en lugar de respecto a g.

Obsérvese que la heurística idénticamente nula es monótona, puesto que hemos supuesto que los costes de las aristas son positivos.

En general, no es necesario comprobar arista a arista si se verifica en cada una de ellas la condición de monotonía (difícil tarea en espacios de estados infinitos …). Por ejemplo, en el caso del 8-puzzle, con coste uno en cada arista y con heurística el número de dígitos mal situados, la heurística es monótona. La razón es que un movimiento puede colocar en su posición a lo más un dígito. En el problema del tren, con coste el número de kilómetros y con heurística la distancia en línea recta a la ciudad objetivo, ésta es monótona: se sigue de la propiedad triangular de la distancia (d(a,b) <= d(a,c) + d (b,c) para cualquier a, b y c) y de que la distancia en línea recta es menor o igual que el número de kilómetros que se realizan. En cambio, si el coste es la duración en horas, la heurística no tiene por qué ser monótona.

En cuanto a las relaciones entre admisibilidad y monotonía, ya se ha comentado en el ejemplo de la figura 11 que una heurística puede ser admisible y en cambio no ser monótona. Sin embargo, la implicación contraria sí que es cierta.

Teorema. Si una heurística es monótona, entonces es admisible.

Demostración. Sea h una heurística monótona y sea e_0 un estado cualquiera que puede ser unido por algún camino con algún estado objetivo. Sea C = (e_0, e_1, …, e_n) un tal camino (es decir, e_n es un estado objetivo, lo que implica que $h(e_n)$ =0). Puesto que h es monótona, se tiene $h(e_0)$ <= $h(e_1)$ + coste-arista(e_0,e_1) <= … <= $h(e_n)$ + coste-arista(e_0, e_1) + …+ coste-arista(e_{n-1},e_n) = coste(C). Y lo anterior es cierto para cualquier camino de e_0 a un objetivo, luego $h(e_0)$ <= $h^*(e_0)$ y esto para cualquier estado e_0 para el que $h^*(e_0)$ esté definido, lo que termina la demostración.

Corolario. Si la heurística es monótona, el algoritmo A es un A*. En particular, si h es monótona, el algoritmo A encuentra una solución óptima (si existe alguna).

Un mismo algoritmo A puede ser parametrizado por distintas heurísticas.¿Cuál es mejor? Llamemos A_1 a un esquema que use h_1 y A_2 al mismo pero que use h_2. Supongamos que h_1 y h_2 son admisibles (o, con otras palabras, que A_1, A_2 son algoritmos A*). Diremos que A_2 *está mejor informado* que A_1 si $h_1(e)$ <= $h_2(e)$ para cada estado e. La razón de ser de esta definición es la siguiente: si A_1, A_2 son dos versiones del mismo esquema A* y A_2 está mejor informada que A_2, entonces todo estado expandido por A_2 será también expandido por A_1. En particular, A_2 expande como mucho tantos estados como A_1.

Según el criterio anterior, la heurística idénticamente nula define el algoritmo A* peor informado. O dicho de otro modo, "primero el menos costoso" es el algoritmo que más estados expande entre los algoritmos A que garantizan que si se encuentra una solución, ésta será óptima.

En cualquier caso, es conveniente no extraer consecuencias precipitadas. El criterio anterior sólo se refiere a la cantidad de cálculo empleada para la expansión de estados, pero no hace referencia al número de operaciones necesario para calcular los valores h(e). Es un fenómeno bastante común que heurísticas mejor informadas requieran también más tiempo para su cálculo. Los dos casos extremos son h = 0, con coste de cálculo mínimo, pero ninguna potencia heurística, y h = h^*, de máxima potencia heurística (de hecho es un conocimiento seguro), pero muy difícil de calcular (en el caso general, para calcular h^* es necesario realizar una búsqueda, lo que, evidentemente, limita su interés como heurística).

Ejercicios. Hacia el final del apartado 4.1 se introdujeron cuatro heurísticas para el 8-puzzle y en un ejercicio una quinta. Supondremos que el coste de cada movimiento es 1. Se pide estudiar la admisibilidad y monotonía de las cinco heurísticas (una ya ha sido someramente comentada en el texto), así como cuáles están mejor informadas.

El final del apartado y del tema está dedicado a un problema clásico en Inteligencia Artificial y también en optimización: el problema del viajante de comercio. Este problema fue brevemente introducido en un ejercicio del apartado 2.1. De él existen muchas variantes. La que nos interesa aquí responde al siguiente enunciado informal: "Un vendedor tiene una lista de ciudades, cada una de las cuales debe visitar exáctamente una vez. Existen carreteras directas entre cada pareja de ciudades de la lista. Encontrar la ruta más corta posible que debe seguir el vendedor que empiece y termine en alguna de las ciudades".

Al estudiar con mayor precisión el problema, observamos que más que una lista de ciudades necesitamos un mapa con las ciudades y las distancias de las carreteras que las unen. Un tal mapa puede ser representado matemáticamente por medio de un grafo no dirigido (sin direcciones en las aristas) y simple (es decir, sin aristas que unan un vértice a sí mismo y con a lo más una arista entre dos vértices).

Un *grafo completo* es un grafo como los anteriores y en el que entre dos vértices distintos siempre existe una arista. Estos grafos quedan determinados por su número de vértices. En grafo completo con n vértices suele ser denotado por K_n. En la figura 13 aparece el grafos K_4.

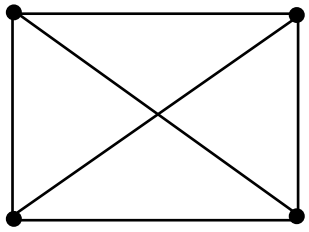


Figura 13

En un grafo completo, llamaremos *ciclo total basado en un vértice v* a un camino que, comenzando en v pasa una única vez por cada uno de los vértices del grafo y vuelve a v.

En todo grafo completo existen ciclos totales. En K_4 hay 6 ciclos totales, basados en cada uno de los vértices, que definen tan solo tres subgrafos, los que aparecen en la figura 14. Fijado uno de los vértices, cada uno de los subgrafos de esa figura representa dos ciclos totales, según la dirección en que se recorran las aristas.

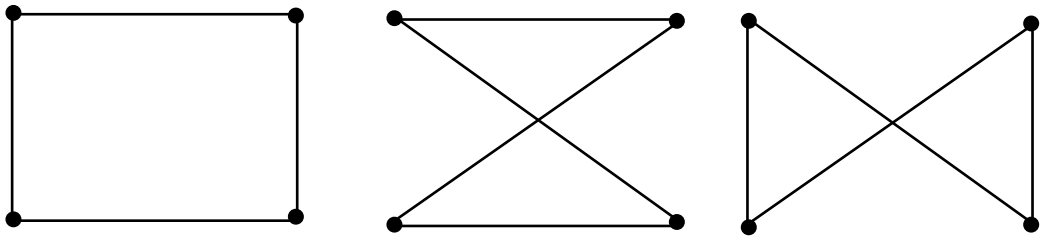


Figura 14

Si el grafo completo, tiene pesos en sus aristas, llamaremos *ciclo total óptimo* (basado en un vértice v) a uno de los que tengan coste mínimo entre los ciclos totales (basados en v). Para el "mapa" recogido en la figura 15, los costes de los ciclos totales aparecen en la figura 16. Como se ve, el único ciclo total (único salvo la dirección de recorrido de las aristas) es el que aparece en la posición central en la figura 16. En general, puede existir más de un ciclo total óptimo.

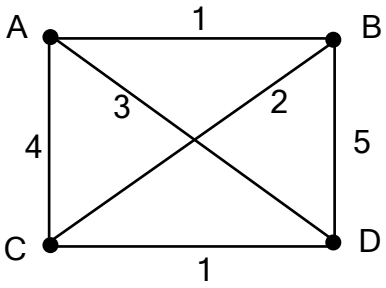


Figura 15

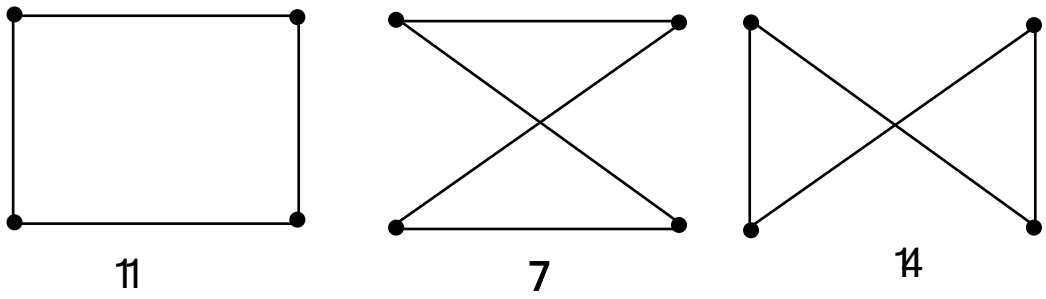


Figura 16

Con la terminología introducida, el problema del viajante puede ser re-enunciado de modo mucho más conciso: "Dado un grafo completo con pesos en sus aristas (y un vértice v en él), encontrar un ciclo total óptimo (basado en v)".

Las ventajas de esta redefinición del problema son múltiples. Aparte de la concisión ya mencionada, se obtiene una descripción mucho más precisa del problema, en la que las circunstancias irrelevantes han sido eliminadas (abstracción). Por último, este proceso de matematización o formalización del enunciado es un primer paso para la tarea de encontrar las estructuras de datos que van a permitir representar en el computador el sistema de producción que resolverá el problema. En lo que sigue, utilizaremos indistintamente la terminología que proviene del problema inicial (ciudades, mapa, etc.) y la de su formalización (vértices, grafo, ciclo, etc.).

Utilizando el mismo estilo que en el apartado 2.1, el planteamiento definitivo del problemas sería el siguiente.

- El problema del viajante del comercio.
    - Entrada: un nombre, asociado a un vértice de un grafo completo.
    - Salida: un ciclo total (óptimo), basado en el vértice con ese nombre.
    - Medios: se dispone del grafo completo con pesos en las aristas; se puede pasar entre cualquier par de vértices distintos por medio de la arista correspondiente.

Obsérvese que la palabra "óptimo" ha sido escrita entre paréntesis al describir la salida del problema. Esto es debido a que el problema del viajante puede ser interpretado como un problema de optimización (en este sentido, sólamente "primero el menos costoso" y los algoritmos A* garantizan la solución del problema) o bien como un problema de búsqueda de un ciclo total (para el que podríamos aplicar

cualquiera de las estrategias explicadas en el texto) en el que, después, se pide que la solución encontrada tenga una cierta calidad (optimalidad, en este caso).

El siguiente paso consiste en elegir una representación para el problema. Esto conlleva buscar nociones adecuadas para los estados y los operadores (en los tres niveles conceptual, lógico y físico).

Por similitud con el problema del tren y viendo que la entrada para el problema puede ser interpretada como una ciudad, podría suponerse que la noción de "estado" adecuada puede identificarse con "ciudad". Sin embargo, un análisis un poco más detallado nos mostrará que eso no es cierto. ¿Basta una ciudad para saber si hemos alcanzado el objetivo? Evidentemente no. La situación es muy diferente en el problema del tren: allí una ciudad basta como estado, puesto que sólo nos interesa llegar a una ciudad fijada. En el problema del viajante saber dónde nos encontramos en cada momento no es suficiente para describir el estado del proceso. La razón es que no sólo buscamos un camino, sino un camino *con restricciones*. Las restricciones se refieren al camino ya recorrido, luego éste debe formar parte del estado, debe *ser* el estado (nótese que en un grafo completo un camino queda definido por cualquier secuencia de vértices en la que no haya dos vértices consecutivos repetidos). La situación, pese a lo que pudiese parecer en una primera observación, es mucho parecida a la del problema de las n reinas (el estado no puede identificarse con una posición con el tablero, sino que debe ser la configuración del tablero, lo que puede ser interpretado como un camino con restricciones) que a la del problema del tren.

Podría pensarse que con la información relativa al "padre" en los nodos, puede definirse como estado sólo la ciudad y recuperar el camino a partir del nodo. Esto sería mezclar, inapropidamente, la noción de nodo y la de estado. Sí puede emplearse esta idea para definir una representación en el nivel lógico (o físico) para los estados (ver más adelante).

En conclusión, los estados serán secuencias de nombres de vértices, comenzando con un mismo nombre, digamos A (que representa a la ciudad de partida), tal que en la secuencia no hay nombres repetidos, salvo en el caso de que la secuencia defina un ciclo total (es decir, en la secuencia aparecen todos los vértices del grafo), caso en el que el último nombre de la secuencia vuelve a ser A. Al imponer que, salvo en el caso de ciclo total, no haya repetidos (es natural hacerlo, puesto que en el objetivo no debe haberlos) estamos ya fijando una precondición para las reglas a definir. Tendremos una regla por cada uno de los pares (ordenados) de vértices (v,w).

– Regla (v,w) en el estado [A, …, B]:

– precondiciones: B = v y, o bien w = A y en [A, …, B] aparecen todas las ciudades del mapa, o bien w no aparece en [A, …, B];

– acción: construir [A, …, B, w]

Una vez fijado el nivel conceptual de la representación ya tenemos definido, implícitamente, el espacio de estados y podemos plantearnos el análisis de su estructura combinatorial. La representación elegida determina que el espacio de estados será siempre un árbol.

Ejercicio. Dibújese el árbol de estados correspondiente al mapa de la figura 15, escribiendo los costes al lado de las aristas.

El hecho de que el espacio de estados sea un árbol conlleva, como ya sabemos, importantes simplificaciones en el proceso de búsqueda, que pueden ser recogidas en las estrategias de control.

Ejercicios.

1) Prográmese el algoritmo A en árbol, adaptando "primero el menos costoso" en árbol (basta ordenar ABIERTOS respecto a g + h, en lugar de respecto a g).

2) El algoritmo anterior puede ser adaptado en el problema del viajante, con la representacion fijada, con distintas variantes:

a) Podemos identificar los nodos con los estados. Esto implicará que cada vez que se requiera la información g, ésta deberá ser recalculada.

b) Para evitar el recálculo citado en (a), podemos modificar la noción de estado, incluyendo en él los costes acumulados de las aristas. Por ejemplo, en el ejemplo de la figura 15, tendríamos como estado [A,0,B,1,D,5]. Esto puede ser fácilmente implementado en Common Lisp, apoyándonos en que sus estructuras de datos no tienen que ser homogéneas en cuanto al tipo de sus componentes (no tienen por qué ser todo números o todo símbolos, etc.).

c) Si representamos en el nivel físico los estados de (b) por medio de listas (o de vectores) lo más probable es que malgastemos mucho espacio de memoria (los caminos van aumentando incrementalmente). Para evitar ese derroche, se pueden representar los estados por medio de estructuras enlazadas con la ayuda de un registro con campos: ciudad, padre y coste. (Esto recuerda mucho a la estructura

de los nodos, pero debe ser entendido como una representación lógica y física de los estados definidos conceptualmente en (b)).

Puesto que el espacio de estados es un árbol, se sigue que en el algoritmo A nunca se revisarán los costes (ni los padres) en ABIERTOS ni en CERRADOS (de hecho, CERRADOS no juega ningún papel en el algoritmo A en árbol). Por ello, la característica de monotonía de las heurísticas no es especialmente importante para el problema del viajante representado de ese modo. Sin embargo, las aproximaciones heurísticas (y su admisibilidad) siguen siendo muy importantes, pues el árbol de estados crece enormemente con el número de ciudades, por lo que no generar todos los estados es imprescindible para poder resolver el problema en la práctica. Por otra parte, si la heurística no es admisible no podremos decir que el problema ha sido *realmente* resuelto (tal vez encontremos en primer lugar un ciclo total que no sea óptimo).

Vamos a considerar seis heurísticas para el problema del viajante. Supongamos que estamos en un estado e que consta de r aristas. Si el número de ciudades en el mapa es n, resultará que para completar un ciclo total a partir de e faltan todavía p = n - r aristas. Las heurísticas son las siguientes:

1) $h_1(e) = 0$.

2) $h_2(e) = p * $ coste mínimo entre los de las aristas que no aparecen en el estado e.

3) $h_3(e)$ = suma de los p menores costes entre los de las aristas que no aparecen en el estado e.

4) $h_4(e)$ = suma del coste de p aristas, una elegida por cada uno de los vértices de los que todavía se tiene que salir para completar un ciclo total: para cada vértice se elige el coste mínimo entre los de las aristas que inciden en él y que no aparecen en el estado e.

5) $h_5(e)$ = suma del coste de p aristas, una elegida por cada uno de los vértices a los que todavía se tiene que llegar para completar un ciclo total: para cada vértice se elige el coste mínimo entre los de las aristas que inciden en él y que no aparecen en el estado e.

6) $h_6(e)$ = coste de la arista que une el origen con el otro extremo del estado.

La primera heurística señala, en realidad, la ausencia de heurística (el algoritmo A se comportará como "primero el menos costoso"). Las heurísticas 2 y 3 se

diferencian de las 4 y 5 en que en las primeras los mínimos son calculados *globalmente* sobre el conjunto de todas las aristas que no aparecen en el estado e, mientras que en las segundas los mínimos son calculados *localmente* sobre cada uno de los vértices involucrados. La heurística 4 se diferencia de la 5 en que en la primera se considera el extremo derecho del estado e, pero no el vértice origen y en la 5 sucede justo al contrario. Las cuatro heurísticas 2, 3, 4 y 5 pueden ser mejoradas fácilmente, pero su cálculo comenzaría a complicarse, siendo necesario hacer cierto tipo (limitado) de búsqueda. En cuanto a la heurística 6, es bastante pobre y rompe la línea de las anteriores. Tendría más sentido si en el mapa dispusiésemos de las coordenadas de las ciudades (como ya se hizo en el problema del tren) y definiésemos como valor heurístico de un estado la distancia *en línea recta* entre el origen y el otro extremo del estado.

Por ejemplo, con el mapa de la figura 15 y para el estado e = [A,B,C], tendremos p = 2 y los siguientes valores heurísticos: $h_1(e) = 0$, $h_2(e) = 2 * 1 = 2$, $h_3(e) = 1 + 3 = 4$ (las aristas son respectivamente CD y AD), $h_4(e) = 1 + 1 = 2$ (las aristas corresponden respectivamente a los vértices C y D; de hecho, los dos números corresponden a la misma arista; éste es uno de los aspectos que podría ser mejorado en la heurística), $h_5(e) = 3 + 1 = 4$ (las aristas corresponden respectivamente a los vértices A y D) y $h_6(e) = 4$.

Ejercicios.

1) Con el mapa de la figura 15, dibújese el árbol de búsqueda A con cada una de las seis heurísticas anteriores.

2) Estúdiese para cada una de las 6 heurísticas anteriores si es monótona y si es admisible. Entre aquellas que sean admisibles, analícese cuáles son las mejor y peor informadas (comparándolas por pares).

TEMA 6. Estrategias para juegos

6.1. Arboles de juego.

Vamos a ocuparnos tan solo de juegos de dos jugadores, con información perfecta (es decir, en los que no interviene el azar y en los que cada jugador "ve" toda la información del otro) y de suma cero (sólo se puede GANAR, PERDER o EMPATAR). Ejemplos son las damas, el ajedrez, las tres en raya.

Vamos a introducir un jueguecito de esas características (que podemos denominar juego de los *autos de choque*) que tiene un pequeño factor de ramificación. Se juega con dos fichas blancas y dos fichas negras en un tablero 3 x 3. La configuración inicial del tablero aparece recogida en la figura 1. Sólo una única ficha puede ocupar una casilla en cada momento. Los movimientos de las fichas están restringidos a las nueve casillas del tablero, excepto las fichas negras que pueden salir del tablero a partir de la columna de más a la derecha (y no pueden volver al tablero), mientras las blancas pueden salir por la fila superior. Uno de los jugadores mueve las negras y puede, en un movimiento, empujar una de ellas a una casilla adyacente arriba, abajo o a la derecha. Su adversario juega con blancas y puede mover arriba, a la derecha o a la izquierda. Gana quien saca en primer lugar sus dos fichas fuera del tablero. Si uno de los jugadores no puede mover en su turno, se producen tablas (un empate). En la figura 2 se recoge una situación de empate si juegan negras.

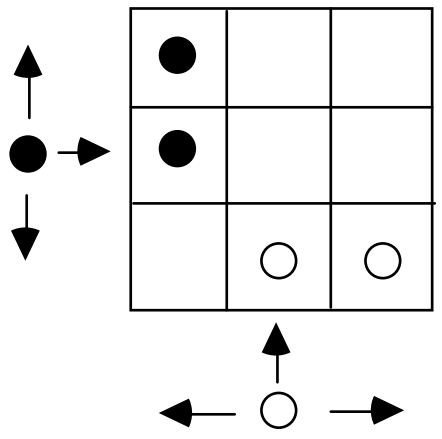


Figura 1

Un árbol de juego es una representación explícita de todas las jugadas posibles en una partida. Las hojas, es decir los vértices terminales, corresponden a posiciones GANAR, PERDER o EMPATAR. Cada camino desde la raíz (la posición inicial del juego) hasta una hoja representa una partida completa.

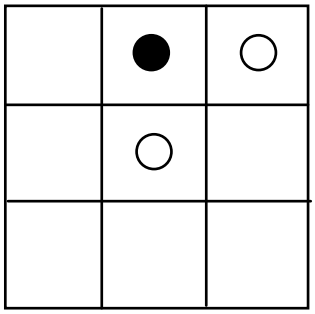


Figura 2

Habitualmente, se denomina MAX al jugador que abre el juego y al segundo, MIN. Son posiciones MAX (MIN) aquéllas en las que tiene que jugar MAX (MIN, respectivamente). Estas posiciones se distinguen por los niveles en que aparecen. Si identificamos la raíz con el nivel 0, y comienza jugando MAX, las posiciones de nivel par corresponden a MAX y las de nivel impar a MIN.

En la figura 3 aparece recogido el comienzo de un árbol de juego para el problema de los autos de choque (comienza MAX con negras).

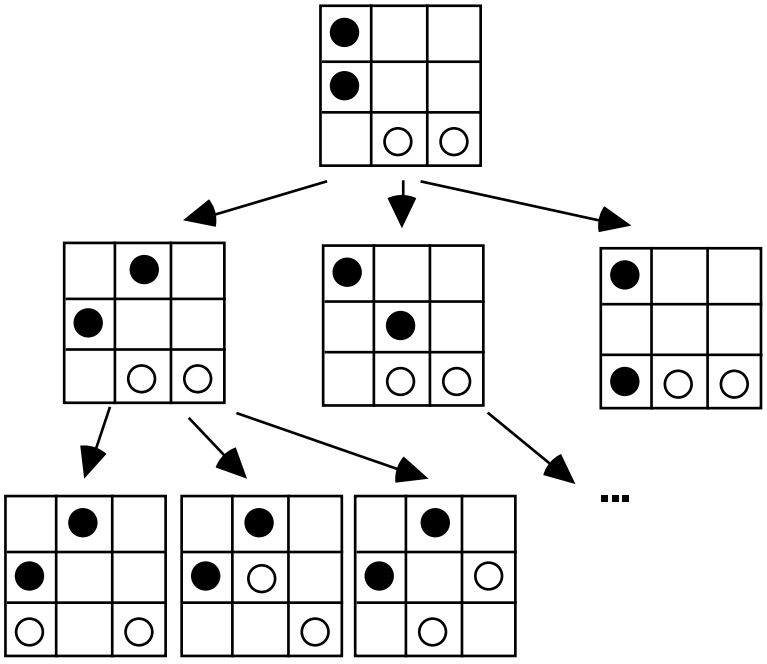


Figura 3

En la figura 4 aparece reflejado un árbol de juego, con las posiciones MAX representadas por cuadrados y las MIN por círculos. Las posiciones terminales tienen etiquetas G, P o E que significan, respectivamente, GANAR, PERDER o EMPATAR *para MAX*.

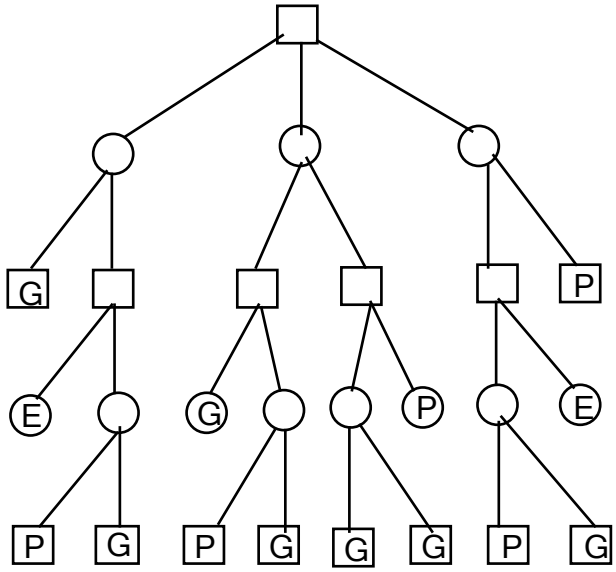


Figura 4

Una vez conocido el árbol de juego podemos propagar hacia arriba las etiquetas P, G, E de las posiciones terminales, según el siguiente *método de etiquetado*. Si N es una posición MAX no terminal, entonces Etiqueta (N) = G, si algún sucesor de N tiene etiqueta G; Etiqueta (N) = P, si todos los sucesores de N tienen etiqueta P; y Etiqueta (N) = E, en otro caso. Si N es una posición MIN no terminal, entonces Etiqueta (N) = G, si todos los sucesores de N tienen etiqueta G; Etiqueta (N) = P, si algún sucesor tiene etiqueta P; y Etiqueta (N) = E, en otro caso.

La etiqueta de una posición nos indica lo mejor que podría jugar MAX en caso de que se enfrentáse a un oponente perfecto.

Resolver un árbol de juego significa asignar, siguiendo el proceso anterior, una etiqueta G, P o E a la posición inicial. En al figura 5 aparece la resolución del árbol de juego de la figura 4. Un *árbol solución para MAX* es un subárbol del árbol de juego que contiene a la raíz y que contiene un sucesor de cada posición MAX no terminal que aparezca en él y todos los sucesores de cada nodo MIN que aparezca en él. (En la literatura, es más frecuente hablar de "estrategias" en lugar de "árboles solución"; hemos preferido esta terminología para evitar confusiones con la noción de "estrategia de control" en sistemas de producción.) Un árbol solución para MAX debe ser interpretado como un plan (una estrategia) de los movimientos que debe realizar MAX, ante cualquier movimiento posible de MIN. Conociendo un árbol solución para MAX, podemos programar a un computador para que juegue con un contrincante MIN, humano o máquina . Sin embargo, en general, un árbol solución para MAX no asegurará que éste gane (ni siquiera que juegue con un mínimo de habilidad). Un árbol solución para MAX se llamará *árbol ganador para MAX* (o estrategia ganadora para MAX) si todas las posiciones terminales del árbol solución tienen etiqueta G. Evidentemente, un árbol ganador para MAX asegura que el jugador MAX ganará, haga lo que haga MIN. En general, existirá un árbol ganador para MAX si y sólo si al resolver el árbol de juego la etiqueta de la posición inicial es G. En la figura 5, el subárbol definido por las aristas punteadas es un árbol ganador para MAX.

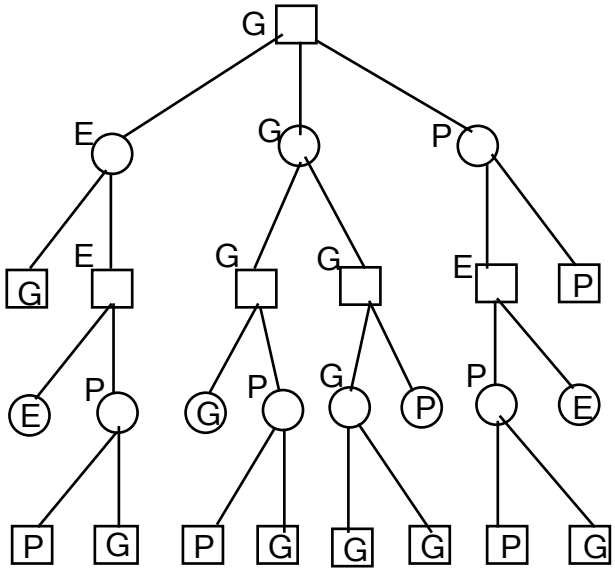


Figura 5

Ejercicios. Definir las nociones de árbol solución para MIN y de árbol ganador para MIN. ¿Existe un árbol ganador para MIN en el árbol de juego de la figura 4?

6.2. Algoritmos MiniMax.

El método de etiquetado descrito en el apartado anterior requiere un árbol de juego completo. Para la mayoría de los juegos, desarrollar todo el árbol de juego es una tarea impracticable, debido a que existen muchos posibles movimientos en cada paso, con lo que el árbol de juego crece enormemente. Un árbol de juego para las damas tiene aproximadamente 10^{40} posiciones no terminales. Generar el árbol completo requeriría más o menos 10^{21} siglos, incluso si en cada segundo se generasen 3 billones de posiciones. El tamaño de un árbol de juego para el ajedrez es todavía más astronómico: unas 10^{120} posiciones y 10^{101} siglos con la misma velocidad de generación. Incluso si un adivino nos proporcionase un árbol ganador (una estrategia ganadora) no tendríamos posibilidad de almacenarlo ni de recorrerlo.

Puesto que, por restricciones de tiempo y de espacio, no podemos evaluar la etiqueta exacta de cada posición sucesora, debemos pensar en una aproximación heurística. Podemos suponer que tenemos una *función de evaluación* (estática) h que nos indica cuál es el "mérito" de cada una de las posiciones. Al contrario que en los dos temas anteriores, se supone que h asigna valores grandes a las posiciones que son más favorables para MAX. Por ejemplo, en el juego de los autos de choque podemos suponer que una ficha negra situada en una casilla cuenta por el número

marcado en la figura 6 para esa casilla (suponemos que MAX juega con negras). Si una ficha negra está fuera del tablero cuenta por 50. Recíprocamente, una ficha blanca cuenta según la numeración que aparece en la figura 7 y -50 si está fuera del tablero.

10	25	40
5	20	35
0	15	30

Figura 6

-30	-35	-40
-15	-20	-25
0	-5	-10

Figura 7

Con esa heurística, la configuración recogida en la figura 8 tendría valor heuristíco -30.

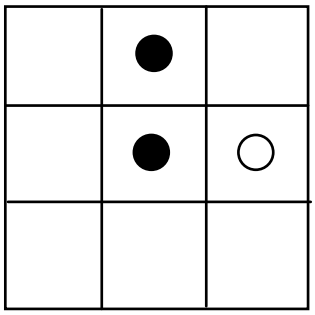


Figura 8

La heurística anterior puede ser mejorada si añadimos además una puntuación para los *bloqueos directos e indirectos*. En la figura 9 aparece un bloqueo directo de las fichas negras sobre las blancas y en la figura 10 un bloqueo indirecto. A los primeros podemos asignarles un peso 40 y a los segundos 30. Recíprocamente, un bloqueo directo de las blancas sobre las negras contará como -40 y uno indirecto como -30.

Con la heurística así enriquecida el valor asignado a la configuración de la figura 8 es -70.

Una vez definida la función heurística h, se podría utilizar una estrategia de control de tipo "escalada" para guiar el juego. Sin embargo, una estrategia como ésa no tiene en cuenta la incertidumbre introducida por la existencia del segundo jugador (no sabemos qué movimiento realizará) y depende demasiado de la calidad de la función h.

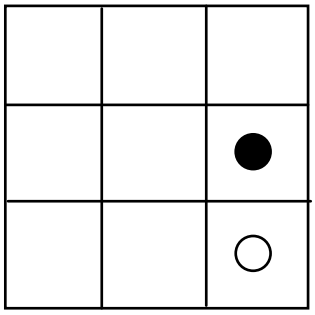


Figura 9

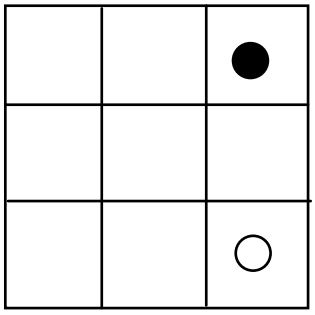


Figura 10

Lo más lógico es explorar el árbol de juego hasta un nivel fijado (que se denomina *frontera de búsqueda*), en las posiciones de ese nivel aplicar la función de evaluación estática y después propagar hacia arriba esos valores (de un modo parecido a como se ha hecho con las etiquetas en el apartado anterior). El nivel de la frontera de búsqueda se denomina *número de capas* del proceso. De este modo, asociamos un nuevo valor V(N) a cada posición N (en general, diferente al valor heurístico h(N)) y, si estamos ante una posición MAX (una posición en la que nos toca mover a nosotros, o … a nuestro computador) elegiremos mover a la posición N tal que V(N) es mayor que el de las otras posibles posiciones. Nótese que si aplicamos la anterior idea con una única capa, recuperamos la estrategia de escalada.

El modo más popular de llevar a la práctica esta idea es el *método MiniMax*:

1. Si N es una posición terminal o una posición en la frontera de búsqueda, el valor V(N) es igual al valor de la función de evaluación estática h(N); en otro caso:

2. Si N es una posición MAX, V(N) es igual al máximo de los valores de sus sucesores.

3. Si N es una posición MIN, V(N) es igual al mínimo de los valores de sus sucesores.

Si ha sido calculado así, V(N) se denomina *valor MiniMax* de la posición N.

Obsérvese que si asociamos con la etiqueta P(erder) el valor numérico -1, con E(mpatar) el valor 0, con G(anar) el valor 1 y tomamos como número de capas el nivel máximo del árbol de juego (o, dicho con otras palabras, no ponemos ningún límite de profundidad), entonces el método anterior es equivalente al procedimiento de etiquetado explicado en el apartado anterior.

En la figura 11 hemos recogido el cálculo del valor MiniMax de 4 capas para la raíz en un árbol de juego. Los números dentro de las posiciones terminales corresponden a unos supuestos valores heurísticos, mientras que el resto de números corresponden a valores encontrados por el método MiniMax.

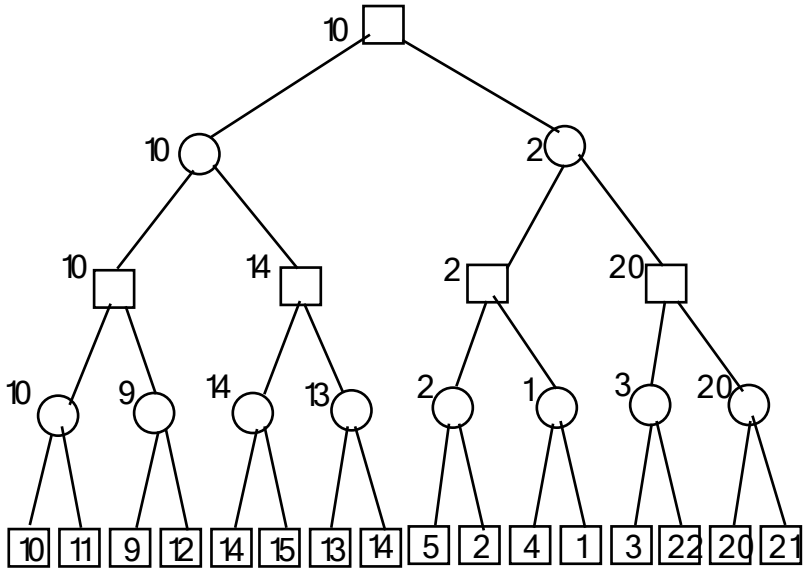


Figura 11

Ejercicios. Calcúlese el valor MiniMax para la posición inicial del jueego de los autos de choque con una capa y la primera heurística introducida (la que no tiene en

cuenta los bloqueos). Idem con 2 capas. Repítase el ejercicio con una y dos capas utilizando la segunda heurística.

Para calcular el valor MiniMax de una posición podemos emplear la siguiente función Common Lisp, que se comporta como una búsqueda en profundidad. Aquí el esquema MiniMax ha sido programado como un caso particular de sistema de producción, por lo que hemos denominado estados a las posiciones y hemos utilizado una función `expandir-estado` para calcular la lista de posiciones sucesoras a una dada.

```
; El metodo MiniMax "en profundidad"
; Una llamada al operador seria: (minimax estado <jugador> <numero-capas>)

(defun minimax (estado jugador n)
       (if (or (terminal? estado) (= n 0))
              (funcion-heuristica estado)
              (let    (     (s-j (siguiente-jugador jugador))
                                  (n-1 (- n 1))
                                  (lista-valores '())    )
                     (do    ((plista (expandir-estado estado) (rest plista)))
                                  ((endp plista) )
                            (setq  lista-valores
                                  (cons  (minimax (first plista) s-j n-1)
                                                                  lista-valores)))
                     (if    (eq jugador 'MAX)
                                  (maxi lista-valores)
                            (mini lista-valores)))))

; Funciones auxiliares:

; (defun terminal? (estado) ...)
;            predicado que determina si una posición es o no es terminal.

; (defun funcion-heuristica (estado) ...)
;            funcion que calcula el valor heuristico de una posicion.

(defun siguiente-jugador (jugador)
       (if    (eq jugador 'MAX)
       'MIN
       'MAX))

; (defun expandir-estado (estado) ...)
;            funcion que calcula la lista de posiciones sucesoras a una dada.

; (defun maxi (lista-numeros) ...)
;            funcion que calcula el maximo de una lista de numeros.

; (defun mini (lista-numeros) ...)
;            funcion que calcula el minimo de una lista de numeros.
```

Es importante notar que en este algoritmo la evaluación de una posición no está completa hasta que cada una de sus posiciones sucesoras en el espacio de búsqueda, hasta un cierto nivel, no ha sido evaluada. Se trata por tanto de una estrategia exhaustiva, aunque de profundidad limitada. Veremos en el último apartado que es posible "podar" el espacio de búsqueda sin perder información.

6.3. Poda Alfa/Beta.

Para ilustrar el hecho de que el algoritmo MiniMax anterior realiza demasiados esfuerzos y que no es necesario evaluar exhaustivamente todos los sucesores para obtener el valor MiniMax, vamos a estudiar el problema en una situación más sencilla. Se trata de establecer un procedimiento de etiquetado sin límite de profundidad, como en el apartado 6.1, pero en el que sólo son posibles dos etiquetas: G (=Ganar) y P (=Perder). En este caso, podemos establecer el siguiente esquema de etiquetado:

1 Si N es una posición terminal, devolver directamente P o G; en otro caso:

2 Si N es una posición MAX, devolver G en cuanto aparezca un sucesor con valor G (este valor habrá sido calculado por el mismo procedimiento que está siendo descrito); si todos los sucesores tienen valor P, devolver P.

3 Si N es una posición MIN, devolver P en cuanto aparezca un sucesor con valor P; si todos los sucesores tienen valor G, devolver G.

En la figura 12 aparece un árbol de juego en el que se ha aplicado este método de etiquetado (se ha explorado de izquierda a derecha). Las aristas punteadas señalan posiciones que no han tenido que ser evaluadas (con el mecanismo de etiquetado del apartado 6.1, todas las posiciones deben ser evaluadas).

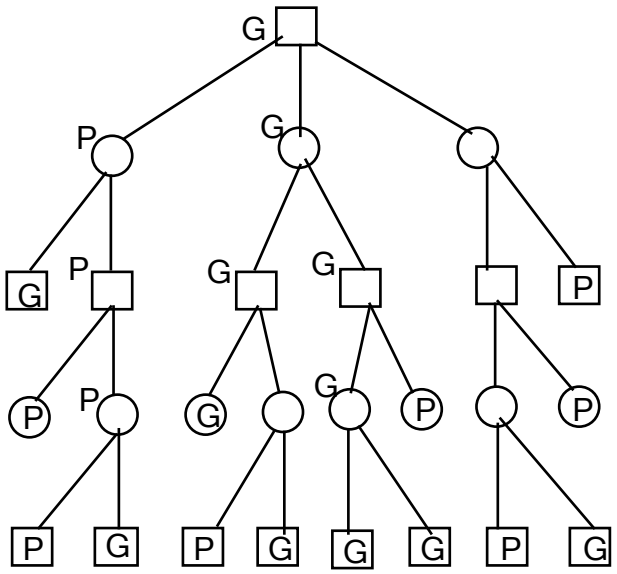


Figura 12

Ejercicio. Estudiar que sucede en el árbol de la figura 12 si expandimos de derecha a izquierda.

Ahora es necesario aplicar la idea contenida en el anterior procedimiento al cálculo del valor MiniMax. Este problema es más difícil por dos razones: la existencia de un número de capas (límite de profundidad) y porque el trabajo debe ser realizado con valores numéricos y no con un par de valores simbólicos.

La idea para llevar a la práctica esa generalización es la siguiente. En lugar de construir una lista con los valores de los sucesores de una posición y, después, calcular el máximo (o mínimo) de esa lista, procedemos a calcular el máximo (o mínimo) secuencialmente, pero si el valor máximo (o mínimo) provisional cruza una cierta cota, entonces se para el proceso de cálculo y se devuelve la cota como valor; si el máximo (o mínimo) no llega a cruzar la cota es el máximo (o mínimo) lo que es devuelto.

Tenemos el siguiente algoritmo V(N,alfa,beta,r), donde r es el número de capas y alfa, beta son dos parámetros numéricos tales que alfa <= beta. El algoritmo calcula el valor MiniMax de N con r capas si dicho valor se encuentra entre alfa y beta. Si el valor MiniMax de N es menor o igual que alfa, devuelve alfa. Y si el valor MiniMax de N es mayor o igual que beta, devuelve beta. Así pues si queremos calcular el valor MiniMax de r capas bastará hacer una llamada V(N,-infinito,+infinito,r). (Los valores -infinito y +infinito pueden ser reemplazados, respectivamente, por un número menor o mayor que el mínimo o máximo de los valores que puede tomar la función heurística, si disponemos de esa información.) Es importante observar que en el siguiente algoritmo recursivo los 4 parámetros pasan por valor y que el contador "i" utilizado, debe ser entendido como una variable local a cada una de las llamadas.

Función alfa-beta V(N,alfa,beta,r) devuelve un número;

1. Si N está en la frontera de búsqueda (es decir, r = 0) o es una posición terminal, entonces devuelve (h(N)); en otro caso: sean (N_1, …,N_i, …,N_k) las posiciones sucesoras de N.
2. i := 1
3. mientras i <= k hacer
4. Si N es una posición MAX

   entonces – alfa := máximo de alfa y V(N_i,alfa,beta,r-1)

   – Si alfa >= beta entonces devuelve (beta)

– Si i = k entonces devuelve (alfa)

si no – beta := mínimo de beta y $V(N_i,alfa,beta,r-1)$

– Si beta <= alfa entonces devuelve (alfa)

– Si i = k entonces devuelve (beta)

5. i : = i+1

En ese algoritmo, las cotas que determinan las podas (el "cortocircuitado" del cálculo del máximo o mínimo) son ajustadas dinámicamente y propagadas del siguiente modo:

1. *Cota alfa*. La cota para una posicion N que sea MIN es una cota inferior alfa, igual al mayor valor hasta ese momento de los antecesores MAX de N. La expansión de N puede ser terminada en el momento en que el mínimo provisional sea menor o igual que alfa.

2. *Cota beta*. La cota para una posicion N que sea MAX es una cota superior beta, igual al menor valor hasta ese momento de los antecesores MIN de N. La expansión de N puede ser terminada en el momento en que el máximo provisional sea mayor o igual que beta.

En la figura 13 se ilustra el funcionamiento de la poda alfa/beta en el árbol de juego de la figura 11. En la figura 13 aparecen punteadas las aristas que llegan a posiciones que no son evaluadas. La mejora respecto al método MiniMax del apartado anterior depende en gran medida del modo de exploración. En el ejemplo de la figura 13 se ha expandido de izquierda a derecha y solamente ha sido necesario evaluar 7 posiciones terminales. En cambio, si expandiésemos de derecha a izquierda, sería necesario evaluar las 16 posiciones terminales y, por tanto, todas las posiciones intermedias también (luego no habría ganancia respecto al método explicado en el apartado anterior).

Ejercicio. Analícese el funcionamiento de la poda alfa/beta con el árbol de juego de la figura 11 y expansión de derecha a izquierda.

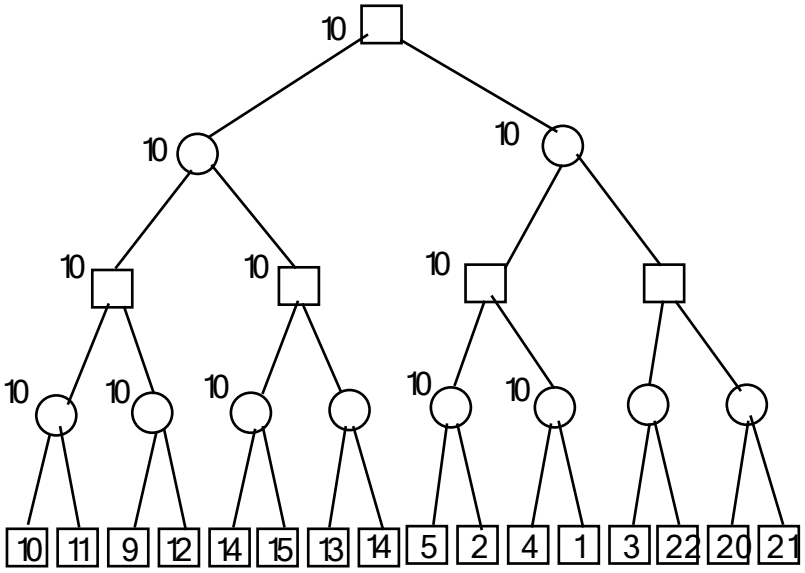

10
10
10
10
10
10
10
10
10
10
10
10
10
11
9
12
14
15
13
14
5
2
4
1
3
22
20
21

Figura 13