LÓGICA Y COMPUTABILIDAD

EL ESQUEMA ALGORÍTMICO DEL BACKTRACKING

INTRODUCCIÓN A ESTE MÉTODO DE RESOLUCIÓN DE PROBLEMAS, ESTUDIO DE VARIANTES POSIBLES Y EJEMPLOS DE IMPLEMENTACIÓN

CARLOS VECINO DE CASAS
MIGUEL ÁNGEL MELÓN PÉREZ
JOSÉ LUIS REDONDO GARCÍA

CONTENIDO

1 Introducción

El backtracking es una estrategia usada para encontrar soluciones a problemas que tienen una solución completa, en los que el orden de los elementos no importa, y en los que existen una serie de variables a cada una de las cuales debemos asignarle un valor teniendo en cuenta unas restricciones dadas. El término fue acuñado por el matemático D. H. Lehmer e la década de los cincuenta y formalizado por Walker, Golom y Baumert en el siguiente decenio. El NIST (U.S. National Institute of Standards and Technology) lo define en su *Diccionario de Algoritmos y Estructuras de Datos* [1] de la siguiente manera:

> *Find a solution by trying one of several choices. If the choice proves incorrect, computation backtracks or restarts at the point of choice and tries another choice.*
> Traducción:
> *Encontrar una solución intentándolo con una de varias opciones. Si la elección es incorrecta, el cómputo retrocede o vuelve a comenzar desde el punto de decisión anterior y lo intenta con otra opción.*
> Definición 1. Backtracking

Este esquema algorítmico es utilizado para resolver problemas en los que la solución consta de una serie de decisiones adecuadas hacia nuestro objetivo. El backtracking está muy relacionado con la búsqueda combinatoria. De manera más exacta podemos indicar que los problemas a considerar son lo siguientes:

1. *Problemas de Decisión:* Búsqueda de las soluciones que satisfacen ciertas restricciones.
2. *Problemas de Optimización*: Búsqueda de la mejor solución en base a una función objetivo.

De forma general, el método del backtracking, concebido como tal, genera todas las secuencias de forma sistemática y organizada, de manera que prueba todas las posibles combinaciones de un problema hasta que encuentra la correcta. En general, la forma de actuar consiste en elegir una alternativa del conjunto de opciones en cada etapa del proceso de resolución, y si esta elección no funciona (no nos lleva a ninguna solución), la búsqueda vuelve al punto donde se realizó esa elección, e intenta con otro valor. Cuando se han agotado todos los posibles valores en ese punto, la búsqueda vuelve a la anterior fase en la que se hizo otra elección entre valores. Si no hay más puntos de elección, la búsqueda finaliza.

Para implementar algoritmos con Backtracking, se usan llamadas a funciones recursivas, en la que en cada llamada la función asigna valores a un determinado punto de la posible solución, probando con todos los valores posibles, y manteniendo aquella que ha tenido éxito en las posteriores llamadas recursivas a las siguientes partes de la solución.

2 Aplicaciones del Método Algorítmico.

La técnica de Backtracking es usada en muchos ámbitos de la programación, por ejemplo, para el cálculo de expresiones regulares o para tareas de reconocimiento de texto y de sintaxis de lenguajes regulares. También es usado incluso en la implementación de algunos lenguajes de programación, tales como Planner o Prolog y da soporte a muchos algoritmos en inteligencia artificial.

3 Esquema general.

A continuación vamos a mostrar un esquema general para este tipo de algoritmos, comentando posteriormente cada una de sus partes y funciones a las que llama. Como ya hemos adelantado con anterioridad, se trata de una función recursiva, que por lo general tiene una forma como la que sigue. Además hemos de señalar que este esquema sirve para resolver aquellos problemas en los que estamos buscando una solución, es decir, cuando encontremos la primera que cumpla todas las restricciones, finalizaremos, sin importarnos si en realidad existen algunas más o si se trata de la mejor de las posibles. Estos otros casos son muy similares y se derivan de este esquema sin ninguna complicación. El ejemplo de los cuadrados mágicos listado posteriormente usa la técnica de búsqueda de todas las soluciones.

```
Funcion Backtracking (Etapai) devuelve: boolean
Inicio
    Éxito = falso;
    IniciarOpciones(i, GrupoOpciones o);
    Repetir
          SeleccionarnuevaOpcion(o, Opcion n);
          Si (Aceptable(n)) entonces
                AnotarOpcion(i, n);
                SiSolucionCompleta(i) entonces
                      Éxito  = verdadero;
                Sino
```

Cuando queremos encontrar la mejor solución, eliminaremos esta instrucción. Y en su lugar compararemos la solución obtenida con una de referencia, para ver si tenemos que actualizar la mejor solución obtenida hasta el momento.

```
                        Éxito = Backtracking(i+1);
                        Si Éxito = false  entonces
                              cancelamosAnotacion(i, n);
                        finsi;
                  Finsi;
            Finsi;
      Hasta (éxito = verdadero) o (NoQuedanOpciones(o));
      Retorna Éxito;
Fin;
```

Si queremos todas las soluciones la instrucción cambiaria por NoQuedanOpciones(o) para conseguir así todas las soluciones posibles. También habría que hacer esto para la mejor solución.

Seguidamente, explicaremos brevemente cada una de las funciones que intervienen en el cuerpo del algoritmo principal y que vienen resaltadas en verde:

1. *IniciarOpciones*(Etapa i, GrupoOpciones o);

En base a las alternativas elegidas en la solución parcial i que nos llega como parámetro, introducimos en "O" el grupo de opciones posibles que podemos probar en la etapa actual. Si ninguno de estos valores condujese a una solución válida, la llamada a la función finalizaría y volveríamos a una etapa posterior.

2. *SeleccionarNuevaOpcion*(GrupoOpciones o, Opcion n);

De entre todas las alternativas posibles que tenemos en O, elegimos una. Aquí podemos realizar funciones de Poda, que pueden se r muy beneficiosas en ciertos algoritmos, proporcionando opciones que tienen una alta probabilidad de convertirse en solución y evitando muchas pruebas innecesarias con valores que según nuestra estimación no nos conducirán al éxito. Hay que señalar que, de alguna forma esta opción elegida tiene que ser desechada o marcada como usada en el grupo de opciones O, para que no volvamos a probar con el mismo valor repetidas veces, es decir, el conjunto vaya reduciéndose hasta que no queden mas alternativas por probar.

3. *Aceptable*(Opción n);

Los problemas que resuelve este tipo de esquemas algorítmicos tienen como característica que buscan un grupo de valores que cumplen entre si una serie de restricciones. Pues bien, el cumplimiento de estas restricciones se comprueba en esta función. Si la opción no es aceptable no será necesario expandir por esa rama las posibilidades porque en ningún caso nos llevarán a una solución. De esa manera evitamos tener que explorar zonas del espacio de alternativas que sólo nos harán perder tiempo de búsqueda sin proporcionar ningún resultado.

Esta función es por tanto de vital importancia, porque tiene funciones de poda que pueden convertir el algoritmo en mucho más eficiente. Para construirla, se suele llevar a cabo un análisis exhaustivo del problema a tratar, para identificar esas situaciones de la solución parcial en la que no debemos expandir más porque nunca llegaremos a una solución. Hay también que considerar que si el coste computacional es elevado, esta función puede incluso empeorar el tiempo de ejecución pues esta función que se ejecutará muchas veces, y si el coste computacional de la misma es elevado y los casos que evita son pocos debemos optar por otra más sencilla y menos difícil de calcular.

4. *AnotarOpcion*(Etapa i, Opción n);

Esta función anota la opción n elegida en las funciones anteriores dentro de la solución parcial que llevamos construida hasta la etapa i. Esta opción fue elegida en *SeleccionarNuevaAcción* y comprobada como válida en *Aceptable*, de manera que tenemos asegurado que es una alternativa que aún no hemos probado, y que no incumple ninguna de las reglas de poda que hayamos introducido en nuestro programa.

5. *cancelamosAnotacion*(Etapa i, Opcion n);

Realiza la acción contraria. Debido a que esa opción n que habíamos anotado no ha llevado a una solución, o estamos buscando soluciones alternativas, la eliminamos de la solución parcial en la etapa i, para que podamos probar con otra si es que existe.

6. *NoQuedanOpciones*(o);

Indica si todavía existe en el grupo inicial que calculamos en cada función recursiva, alguna alternativa más que probar. Si no queda ninguna opción, inevitablemente hemos terminado de buscar en la etapa actual, y debemos ir hacia atrás a la anterior llamada recursiva para buscar nuevas alternativas por donde expandir el árbol de posibilidades.

4 Variantes del esquema general

Hasta ahora hemos considerado que los problemas que se resuelven con esta técnica sólo encontraban una única solución, independientemente de que hubiese más. A continuación veremos que este esquema ofrece más posibilidades a la hora de buscar soluciones.

Básicamente, las variantes se dividen en tres grandes grupos que estudiaremos seguidamente: implementaciones que calculan una solución cualquiera, todas las soluciones y la mejor de todas las soluciones.

4.1 Una solución cualquiera

Es la variante más común y la que hemos utilizado hasta ahora. El algoritmo comienza y busca una solución en el espacio de búsqueda. En el momento de encontrarla para y la devuelve, sin considerar el resto de las ramas del árbol

de exploración. No importa que haya más soluciones, ni siquiera que la que hemos conseguido sea la mejor.

En el ejemplo del laberinto, que se presenta implementado junto a este informe, se usa esta variante.

4.1.1 Todas las soluciones

En este caso se explora todo el espacio de búsqueda. Cada vez que encontramos una *solución completa*, la anotamos, la guardamos en una estructura de datos y continuamos buscando hasta que ya no queden más posibilidades por explorar. De esta manera obtenemos el conjunto de todas las soluciones que satisfacen el problema.

El problema del laberinto se puede modificar para que adopte esta variante, aunque conlleva una serie de costes adicionales, tanto de tiempo de ejecución como de almacenamiento en memoria.

```
Función VueltaAtrásTodas(etapa,solucionesCompletas):
booleano
Inicio
   éxito = falso
   IniciarOpciones()
   repetir
      SeleccionarNuevaOpción()
      si Aceptable() entonces
         AnotarOpción()
         siSoluciónCompleta() entonces

   ApuntarSoluciónCompleta(actual,solucionesCompleta
s)
            éxito = verdadero
         si no
            éxito = VueltaAtrás(etapaSiguiente)
            si no éxito entonces
               CancelarAnotación()
            finsi
         finsi
      finsi
   hasta (ÚltimaOpción())
   devuelve éxito
Fin
```

Algoritmo de vuelta atrás para encontrar todas las soluciones [**¡Error! Marcador no definido.**]

En este fragmento de pseudocódigo correspondiente a esta variante se puede apreciar que el algoritmo únicamente acabará cuando se haya analizado la última opción. También podemos ver cómo se van anotando todas las soluciones completas para posteriormente devolverlas.

4.1.2 La mejor de todas las soluciones

Siguiendo esta opción, se escoge la mejor de las soluciones posibles. Para ello se tiene que procesar todo el espacio de búsqueda, luego esta variante no deja de ser un caso particular de la anterior. La diferencia radica en que a medida que se van obteniendo las soluciones comparamos la nueva con la que ya tenemos de antes, quedándonos en cada caso con la mejor hasta que se termina de explorar el árbol de búsqueda.

Para poder llevar a cabo esta técnica tenemos que partir de un caso base. De esta manera podemos tener un punto de referencia a la hora de hacer las comparaciones para obtener la mejor solución.

```
Función VueltaAtrásTodas(etapa,soluciónMejor):
booleano
Inicio
   éxito = falso
   IniciarOpciones()
   repetir
      SeleccionarNuevaOpción()
      si Aceptable() entonces
         AnotarOpción()
         siSoluciónCompleta() entonces
            éxito = verdadero

   siEsMejorSolución(soluciónActual,soluciónMejor)
         entonces
               ApuntarSoluciónActual()
         finsi
         si no
            éxito = VueltaAtrás(etapaSiguiente)
            si no éxito entonces
               CancelarAnotación()
            finsi
         finsi
      finsi
```

```
        hasta (ÚltimaOpción())
        devuelve éxito
    Fin
```

Algoritmo de vuelta atrás para encontrar la mejor solución [**¡Error! Marcador no definido.**]

El pseudocódigo muestra la manera en la que se va comparando cada solución completa con la más óptima obtenida hasta el momento. En caso de que sea mejor la actual, se actualiza el valor. Nuevamente vemos cómo el algoritmo no termina hasta que se procesa la última opción de todas las posibles.

5 Eficiencia y costes

Como ya hemos dicho anteriormente en algunas ocasiones los algoritmos de backtracking no son eficientes, ya que se basan en el método de prueba y error, y en muchas ocasiones para conseguir solución se tiene que recorrer todo el árbol de búsqueda o gran parte de él. También tendremos que achacar que la recursividad contribuye a su ineficiencia, por la memoria que gasta durante las diferentes llamadas recursivas.

Ahora bien hay que decir que la eficiencia depende de:

- El número de nodos del árbol de búsqueda que se visitan para conseguir la solución $\rightarrow v(n)$.
- El trabajo realizado en cada nodo, esto es, el coste de la función de solución completa o ver si la solución es aceptable hasta el momento. Este coste lo podemos expresar como $\rightarrow p(n)$, ya que generalmente será polinómico.
- El coste en general será: $O(p(n)v(n))$, este coste será exponencial en el peor caso.

Para conseguir mejoras en los costes se suele recurrir a la poda del árbol de búsqueda, lo cuál se hace marcando los caminos que ya se han estudiado y los no prometedores como cerrados con lo cuál el algoritmo no perderá tiempo con los nodos que estén dentro de estos caminos. También se podrían crear predicados acotadores reduzcan mucho el número de nodos generados, si el predicado acotador sólo dejara un nodo el algoritmo se convertiría en voraz y su coste bajaría a $O(p(n)n)$. Aunque normalmente los costes más bajos que se pueden conseguir son el orden de $O(p(n)2^n)$.

En conclusión podemos decir que debido al coste creado en tiempo y memoria (por la pila recursiva) los algoritmos de vuelta atrás no son todo lo eficientes que deberían, y debemos dejarlos para resolver parte de otros problemas o problemas reducidos. Aún así la gran ventaja que tienen es que si hay solución la encontrarán.

Ventajas	Inconvenientes
• Si existe una solución, la calcula. • Es un esquema sencillo de implementar. • Adaptable a las características específicas de cada problema.	• Coste exponencial $O(j^i)$ en la mayoría de los casos. • Si el Espacio de Búsqueda es infinito, la solución, aunque exista, no se encontrará nunca. • Por término medio consume mucha memoria al tener que almacenar las llamadas recursivas.

Figura 1: Ventajas e inconvenientes del backtracking

6 Árboles de búsqueda.

Cómo ya hemos comentado anteriormente el algoritmo de vuelta atrás proporciona una manera sistemática de generar las todas las posibles soluciones siempre que se puedan resolver por etapas, lo que se asemeja mucho a una búsqueda combinatoria (probar todas la posibles combinaciones).

Para conseguir este estudio tan exhaustivo del problema, se considera que se trabaja con un árbol (figura 1) que cuya existencia es sólo implícita, para nosotros cada nodo del nivel k representa una parte de la solución y nuestro árbol estará formados por las k etapas que se considerarán ya realizadas.

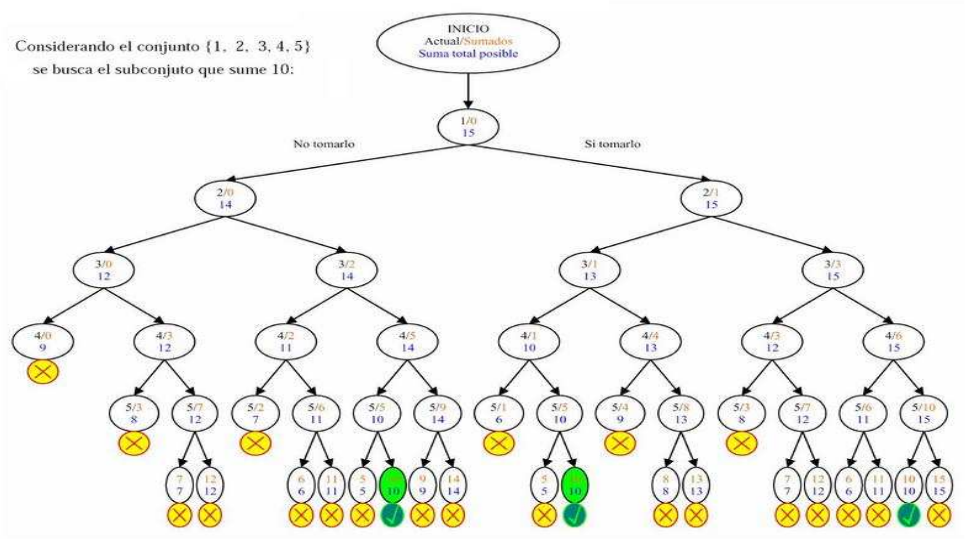


Figura 1: Árbol de Búsqueda.

La búsqueda que realizada sobre el árbol es una búsqueda en profundidad. En el transcurso de la búsqueda si se encuentra un estado incorrecto, se ha de retroceder hasta la decisión anterior y si existe uno o más caminos aún no explorados que puedan conducir a la solución, el recorrido del árbol continúa por uno de ellos (hijos si nos referimos a un árbol). Si no quedasen más alternativas la búsqueda fallaría y el problema no tendría solución. En este caso en el que consideramos el problema en forma de árbol, la solución sería un camino que llevara desde el nodo raíz hasta uno nodo hoja, y las soluciones parciales llevarían desde el nodo raíz a los nodos interiores del árbol.

6.1 Ejemplo de espacio de búsqueda: las N- Reinas.

Como podemos ver, esta forma de actuar, concebida de forma estricta, realiza una búsqueda exhaustiva del espacio de direcciones. El proceso puede ser por lo tanto muy lento y tedioso, porque muchos de los problemas se pueden hacer enormes a medida que aumenta el número de pasos para encontrar una solución, y sobre todo, el número de alternativas que podemos elegir en cada etapa. Este crecimiento exponencial puede producir que muchos problemas sean inasumibles desde el punto de vista temporal.

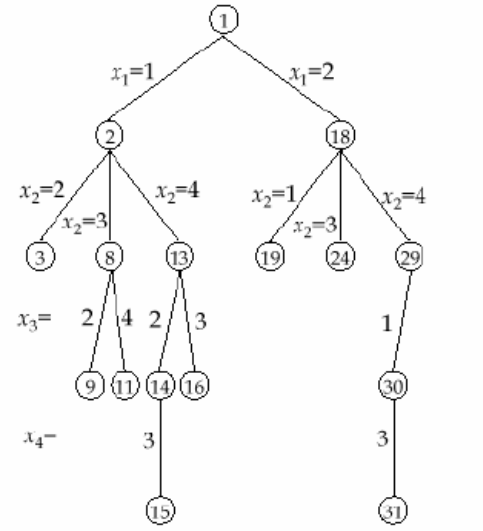
Sin embargo, para la mayoría de los problemas, se puede refinar más la búsqueda. Esto es debido a que muchos planteamientos de resolución admiten acotar el espacio de alternativas posibles.

Un ejemplo muy claro es el de las N-reinas, en este juego, el objetivo consiste en colocar las N reinas en un tablero de ajedrez sin que se den jaque. Si comprobamos todas las posibles combinaciones, el número de posibilidades es enorme, del orden de N al cuadrado sobre N.

Sin embargo, si analizamos el problema, podemos acotar mucho el espacio de posibles situaciones: en primer lugar, dos reinas no pueden estar en la misma fila, por lo q solo tenemos decidir en qué columna colocar cada reina. Además, dos reinas no pueden estar en la misma columna, por lo que la as soluciones son permutaciones de la tupla (1, 2 …. N). Ahora el tiempo de computación es mucho menor, hemos reducido el número de posibilidades a factorial de N.

Otros métodos también pueden acelerar el proceso de búsqueda. Debido a que no importa el orden en que procesemos las variables, suele ser más eficiente elegir valores en aquellas partes de la posible solución donde existen menos alternativas para explorar, es decir, aquella donde hay aplicadas más restricciones. También es aplicable la técnica de *eliminación de simetrías.* En muchos problemas reales existen simetrías que hacen que varias soluciones sean esencialmente la misma (rotaciones unas de otras, o proyecciones…). Encontrar la misma solución varias veces es ineficiente, por lo que se añaden nuevas restricciones que no aparecían al principio y que prohíben que haya soluciones repetidas.

Además, muchas implementaciones, antes de elegir el valor que van a asignar a la etapa actual, realizan una comprobación hacia delante (que consiste básicamente, en adelantarse unas cuantas etapas más), para ver cuál de los valores nos puede llevar con mayor probabilidad a una solución.



Árbol de búsqueda reducido para
N = 4, (16 nodos frente a 65 iniciales)

Otra técnica implementada consiste en usar, dentro de la fase de elección de una de las alternativas, una función de poda. Esta función tratará de predecir si es posible obtener una solución en base a la solución parcial formada por los valores ya elegidos más el valor en cuestión que estamos probando. Debido a que esta función se ejecutará muy frecuentemente, lo más seguro, en cada uno de los pasos o etapas, el coste computacional de la misma debe ser el menor posible, porque sino la eficiencia del algoritmo, lejos de aumentar, se hará mucho peor.

6.2 Ejemplo de espacio de búsqueda: el problema del laberinto

Como hemos visto anteriormente, cuando se resuelve un problema, se busca la mejor solución entre un conjunto de posibles soluciones. Cada punto en el espacio de búsqueda representa una posible solución, parcial en el caso de estar en un nodo interior o completa si estamos en un nodo hoja del árbol. Igual que en los casos anteriores, este árbol se irá explorando en profundidad.

Cada nodo del árbol representa una etapa y los hijos de dichos nodos serán las alternativas que se nos ofrecen en cada etapa. Como decisión tomaremos un hijo por el que seguir explorando y expandiendo el árbol, siempre en profundidad. A continuación se muestra un ejemplo de espacio de búsqueda junto con su árbol para el problema del laberinto:

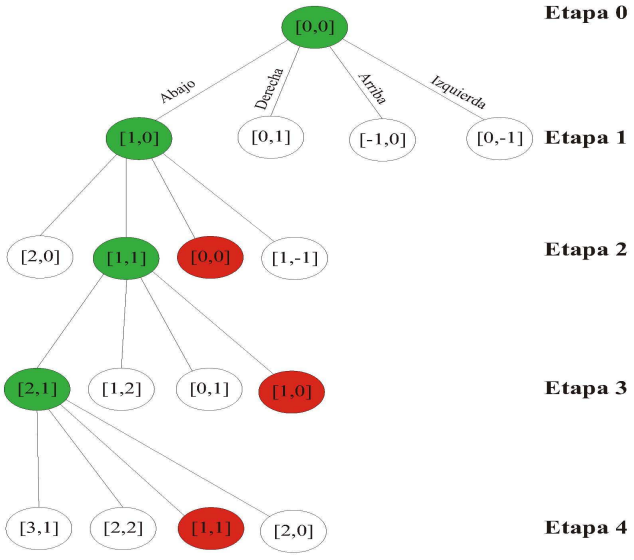


Figura 2. Árbol de exploración para las primeras etapas del problema del laberinto

Como se puede ver, partimos de la posición [0,0] (etapa 0) que hemos supuesto la entrada al laberinto y que va a ser el nodo raíz en el árbol. Expandiendo todas las alternativas que se nos ofrecen, optamos en primer lugar por ir hacia abajo en el mapa [1,0]. Más adelante veremos que las posiciones que se salen del laberinto ([-1,0], [0,-1], [x,y],

$\exists z \in \{x,y\} \setminus z < 0$) las descartamos antes de optar por ellas para incluirlas en el camino hasta la salida.

Una vez que estamos en la etapa 1 con nodo actual [1,0], volvemos a expandir las posibilidades que se nos ofrecen. Aquí nuevamente hacemos una criba para desechar las alternativas que nos sacan del mapa, así como las que ya hemos elegido (sería volver sobre nuestros pasos). Nuevamente tomamos una decisión, en este caso ir hacia la derecha y pasar a la posición [1,1]. Este proceso se repite sucesivamente hasta encontrar la salida (o no).

En el caso de encontrar un camino muerto, el propio algoritmo retrocede sobre las decisiones que ha tomado para, en un determinado punto, tomar otra alternativa y seguir explorando, tal y como se muestra en la *figura 2*. Se puede ver cómo desde la etapa *k* hemos expandido los nodos para obtener las distintas alternativas. En el siguiente nivel (*k+1*), las dos primeras ya han sido exploradas con resultados poco fructíferos. Sin embargo, en la tercera opción parece que podemos seguir explorando. Nuevamente expandimos los nodos hijos (etapa *k+2*), que analizándolos vemos que todos ellos son estados muertos. Entonces *retrocedemos* a la etapa anterior (*k+1*) para seguir explorando por alguna de las alternativas abiertas. Otra vez nos encontramos con un nodo muerto, y al ser la última de las opciones que se nos ofrecían en esta etapa tendremos que subir otro nivel (volvemos a la etapa *k*) en el árbol para buscar una alternativa abierta y seguir intentando encontrar la solución a partir de ella.

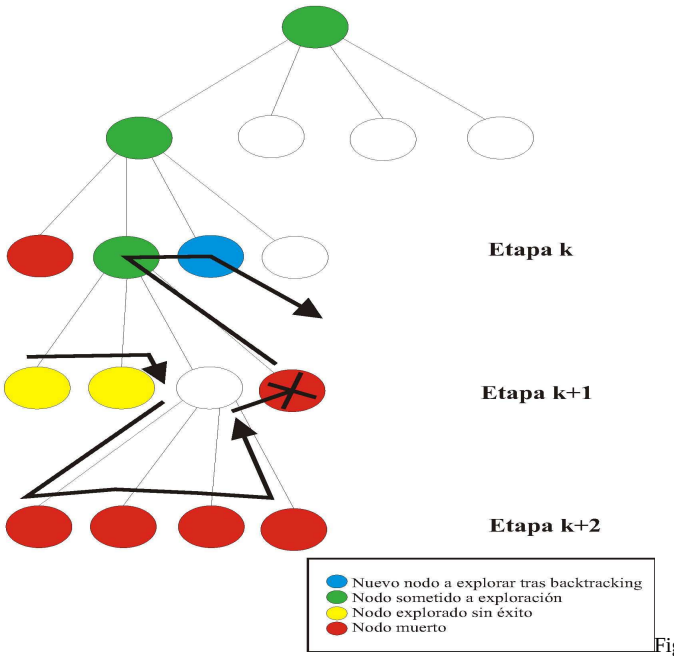


Figura 3. Ejemplo de funcionamiento del backtracking sobre el árbol de exploración

Como caso particular, en el problema del laberinto el árbol tendrá una altura infinita, pero como veremos en la implementación se aplican algunas condiciones para *podar* ramas por las que el algoritmo no avanzaría en su proceso, sólo retrocedería. Tras aplicar esta poda, el árbol contará, como mucho, con *n* niveles, siendo *n* el número de casillas del laberinto.

7 Ejemplos de implementación.

7.1 Problema del "salto de caballo"

El enunciado del problema que vamos a resolver es básicamente el recorrido que tiene que hacer un caballo por el tablero para poder pasar por todas y cada una de sus casillas. Nosotros podremos elegir donde empezará a moverse el caballo si queremos o no poner una mina en el tablero para que el caballo no pueda pasar por esa casilla. Hay que decir que debido a la poca eficiencia de la vuelta atrás el tamaño del tablero estará restringido a 5 casillas x 5 casillas, ya que con tableros más grandes los tiempos de resolución se disparaban por encima del minuto y medio en un ordenador con 2,8Ghz.

7.1.1 Etapas de la resolución

Para empezar identificaremos el tipo de algoritmo de vuelta atrás que debemos usar, y veremos que como no se nos piden todas la soluciones, y como no se puede saber a ciencia cierta si un recorrido del caballo es más costoso que otro, utilizaremos el algoritmo que para cuando obtiene la primera solución encontrada.

Para presentarle la solución al usuario y para ir guardando la solución tenemos que elegir cuál es la estructura de datos más adecuada. En nuestro caso elegiremos una matriz que nos servirá para ir marcando las casillas por las que va pasando el caballo y el orden en que lo hace. Para mostrarle la solución al usuario bastará con mostrar por pantalla la matriz o en caso de que no se encuentre la solución con poner el mensaje de que no hay solución.

El mayor problema que podemos tener es el de representar cuál será la siguiente posición que debe de tomar la ficha, pero es fácil de solucionar ya que podemos incrementar y valor de las coordenadas actuales de la ficha en función de la dirección en la que queramos hacer el movimiento. Para ayudarnos tendremos dos vectores que en cada una de sus posiciones tendrán cuanto hay que sumar a la posición actual para que el caballo se mueva, el vector tendrá ocho casillas ya que el número máximo de posibles movimientos del caballo son 8. A continuación mostraré una figura que ilustre lo comentado en este párrafo.

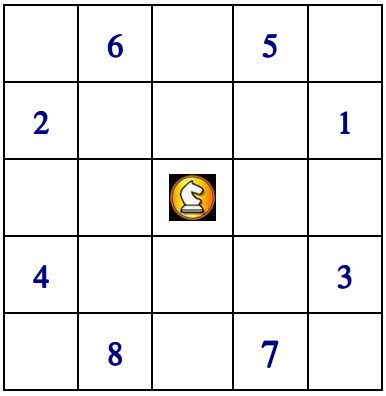


Figura4. Movimientos Caballo

constintmovX[8]={2,-2,2,-2,1,-1,1,-1};
constintmovY[8]={1,1,-1,-1,2,2,-2,-2};

En los dos vectores anteriores podemos ver como coinciden los incrementos con cada una de las posiciones indicadas en la Figura4.

Llegados este punto sólo queda por decidir cuando una etapa es válida y cuando se llega a la solución final del problema.

Una etapa es válida y vale como solución parcial del problema, cuando el caballo se encuentra en una casilla no visitada anteriormente y no se encuentre sobre ninguna mina. Si el estado al que se llega no es válido habrá que borrar la casilla, poniéndola a cero como si el caballo no hubiera pasado por allí, y se probará con otra posibilidad. Si las posibilidades se acaban y no se puede retroceder más el problema no tendrá solución y se mostrará un mensaje al usuario indicando el suceso.

Para llegar a la solución completa basta con mirar el número de movimientos realizados. En nuestro caso serán 25 (tablero de 5x5) y si ponemos una mina serán 24, por tanto al llegar a este número de movimientos tendremos una solución completa para nuestro problema y este podrá terminar.

A partir de este momento al tener ya claros cuales son las etapas y requisitos que tendrá que tener nuestro código procederemos a la implementación del mismo.

7.1.2 Estructuras de datos utilizadas para resolver el problema.

Las estructuras de datos que definiremos para la resolución del problema serán las siguientes:

```
constintTamMax= 5;
typedefintTablero[TamMax][TamMax];
typedefint mina[2];
constintmovX[8]={2,-2,2,-2,1,-1,1,-1};
const int movY[8]={1,1,-1,-1,2,2,-2,-2};
```

Ahora explicaré el significa y utilización de cada declaración:

`const int TamMax= 5;` → es una constante que indica el Tamaño de nuestro tablero.

`typedef int Tablero[TamMax][TamMax];` → es una matriz que representará a nuestro tablero ya que es la estructura más idónea para ir llevando la solución.

`typedef int mina[2];` → vector para guardar la posición de las minas.

Y por último dos vectores que nos servirán para mover la ficha por el tablero, sumando o restándole a la posición inicial según convenga:

```
constintmovX[8]={2,-2,2,-2,1,-1,1,-1};
constintmovY[8]={1,1,-1,-1,2,2,-2,-2};
```

7.1.3 Función de Vuelta Atrás.

En este apartado voy a mostrar como ha quedado la función más importante de nuestro programa. Para ello me ayudaré del código ya implementado e iré aclarando los asuntos importantes al lado de las diferentes instrucciones. Cabe destacar que la variante de solución que hemos escogido es la que nos da la primera solución que encuentre sobre el problema. También podremos observar la gran similitud que tiene con la plantilla genérica de resolución que mostrábamos en apartados anteriores.

- Código función Vuelta Atrás:

```
  voidvueltatras(Tablerot,intposX, intposY, bool&solucion, intmovimiento,int
minas){
      intpX=0;
      intpY=0;    (1)
      intpmov;
      pmov=0;
      solucion=false;
      do{
      pX=posX+movX[pmov];
      pY=posY+movY[pmov];          (2)
      if(valido(pX,pY,t)==true){
            t[pY][pX]=movimiento;
            if(movimiento<(25-minas)){
                  vueltatras(t,pX,pY,solucion,movimiento+1,minas);
            if(solucion==false) t[pY][pX]=0; (3)
            }
            else{
                solucion =true;
            }                          (4)
         }
         pmov++;
       }while(solucion==false && pmov<8);   (5)
  }
```

Ahora comentaremos ayudándonos de las llaves y los números con que hemos dividido el código, el significado que tienen los diferentes conjuntos de instrucciones:

- (1): declaración de las variables de posicionamiento (pX y pY) que nos indican la posición del caballo sobre nuestro tablero, y la variable pmov que será la que nos indique en que fase de movimiento estamos, es decir, cuantos movimientos nos quedan por probar y que posición de los vectores de posición hemos de tomar para obtener el siguiente intento de posición de nuestro caballo. Si pmov llega a ocho no habrá solución.
- (2): con estas instrucciones actualizaremos los valores de pX y pY para poner el caballo en una determinada posición ayudándonos de los vectores de posicionamiento (movX y movY). A continuación miraremos si la casilla es aceptable y marcaremos la casilla como visitada.
- (3): En estas instrucciones es donde se controla si se ha conseguido llegar a una solución Aceptable, y si no es así se realiza la llamada recursiva para la siguiente etapa. Si cuando se hace la llamada recursiva no tiene éxito el movimiento marcado en el anterior punto se borrará y dejará la casilla como no visitada.
- (4): aquí se hace el control para ver si se ha llegado a la solución, poniendo solución a true. También se aumenta el pmov para que pase a la siguiente alternativa de movimiento si no hemos llegado a ninguna solución con la anterior.
- (5): por último aquí se comprueba si el algoritmo ha terminado de ejecutarse, ya sea por haber encontrado una solución o por haber recorrido todo el árbol de búsqueda sin éxito.

7.2 Problema de los Cuadrados Mágicos.

A continuación se muestra un programa a modo de ejemplo que usa la vuelta atrás como método de resolución del problema. En este caso las soluciones serán cuadrados mágicos de orden N. A continuación pasamos a desarrollar algunos puntos del mismo para comprender mejor como funciona.

7.2.1 ¿Qué es un cuadrado mágico?

Un cuadrado mágico es la disposición de una serie de números enteros en un cuadrado o matriz de forma tal que la suma de los números por columnas, filas y diagonales sea la misma, la constante mágica. Usualmente los números empleados para rellenar las casillas son consecutivos, de 1 a n^2, siendo n el número de columnas y filas del cuadrado mágico. No siempre es así, es decir, pueden construirse cuadrados mágicos usando números no consecutivos, pero en este ejemplo hemos optado por la primera opción porque el rango de alternativas es menor, aunque, eso sí, hace que el problema sea más difícil de resolver.

4	***9***	***2***
3	***5***	***7***
8	***1***	***6***

17	***24***	***1***	***8***	***15***
23	***5***	***7***	***14***	***16***
4	***6***	***13***	***20***	***22***
10	***12***	***19***	***21***	***3***
11	***18***	***25***	***2***	***9***

7.2.2 Breve esbozo del proceso de construcción del cuadrado mágico.

La heurística de construcción que usará este programa para construir los cuadrados mágicos es muy simple. En general, la aplicación procederá como sigue:

Comenzando desde la casilla superior izquierda, iremos colocando números comprendidos entre 0 y N*N, siendo N el ancho del cuadrado. Cuando coloquemos uno en una celda, esté ya no estará disponible para ponerlo en las siguientes, pues los números no se pueden repetir.

De esta manera iremos rellenado el cuadrado, teniendo en cuenta lo siguiente: en ningún momento, estando en

una casilla, las sumas parciales de fila y columna, así como de las diagonales si estamos en una casilla de este tipo, debe superar la constante mágica. Además si estamos en la última casilla de cada fila o en la última de cada columna debemos comprobar que la suma total es la adecuada. Si estamos en la esquina inferior izquierda o en la inferior derecha, comprobaremos que las dos sumas diagonales coinciden exactamente con la constante mágica.

Si en un determinado estado ninguno de los números que podemos colocar permiten que se cumplan las restricciones explicadas, se vuelve atrás a la fase anterior y se prueba con otro número distinto al que habíamos colocado.

7.2.3 Aplicación del algoritmo general. Funciones claves implicadas.

1. *IniciarOpciones*(Etapa i, GrupoOpciones o);

En cada etapa nos dirá el conjunto de números que podemos colocar en esa casilla. Básicamente, corresponden con aquellos números que no hemos colocado en etapas anteriores..

2. *SeleccionarNuevaOpcion*(GrupoOpciones o, Opcion n);

Elige un número aleatorio de entre todos los que se encuentran en el grupo de opciones.

3. *Aceptable*(Opción n);

Es aceptable si en cada celda las sumas parciales de fila, columna y diagonal (si es el caso) no supera la constante mágica. Además si nos encontramos en el caso de que es la última celda de cada fila o de cada columna, comprobamos que la suma coincida exactamente con la constante, sin olvidar que lo mismo ocurre con llas diagonales cuando estemos en la esquina inferior derecha e izquierda.

4. *AnotarOpcion*(Etapa i, Opción n);

Consiste en escribir el número elegido para esa celda dentro de la misma, y marcar el número como ya usado dentro del conjunto de posibles alternativas. De esta manera no volveremos a usarlo en esa misma etapa.

5. *cancelamosAnotacion*(Etapa i, Opcion n);

En este caso esta función no es necesaria,, simplemente los nuevos valores se escribirán encima del anterior haciendo las veces de cancelación.

6. *NoQuedanOpciones*(o);

Indica si todavía existe en el grupo inicial algún número con el que probar, o por el contrario se han examinado ya todas las alternativas y no es posible seguir desarrollando por esa rama..

7.2.4 Coste computacional y complejidad del problema.

Como reseña de cómo aumenta la complejidad de forma exponencial, podemos ver como los cuadrados de orden 4 tardan bastante en ser calculados, y los de 5 comienzan a convertirse en una cuestión de varias horas. Esto es debido a que el número de combinaciones posibles se dispara. En el caso de tres, tenemos que rellenar 9 casillas con una media de 3*3/2 = 4,5 alternativas… en el caso de cinco por 5, tenemos veinticinco casillas y 5*5/2 12,5 alternativas para cada una.

Probablemente la función de poda sea demasiado tediosa y pesada para ser ejecutada en cada iteración, pues realiza recorridos dentro de la matriz y muchas comprobaciones y sumas. Quizá incluso una función menos depurada y más simple hubiera proporcionado mejores resultados.

Cabe señalar q sería posible mejorar la construcción de cuadrados mágicos usando otra heurísticas y algunos teoremas matemáticos no tan triviales. Además, podríamos eliminar alternativas debido a que muchas situaciones corresponden al mismo cuadrado rotado.

7.3 Problema del laberinto

El problema del laberinto podemos plantearlo de la siguiente manera:

Tenemos un laberinto representado por una matriz cuadrada de casillas, las cuales pueden estar ocupadas o no. Desde cada casilla nos podremos mover a cualquiera de sus cuatro adyacentes, siempre y cuando no estén ocupadas. El problema consiste en encontrar un camino (si existe) desde la casilla origen, ubicada en la esquina superior izquierda, a la casilla destino, que a su vez estará en la esquina inferior derecha.

A continuación se lista un trozo de código del programa que se entrega junto a este informe. El fragmento corresponde a la función que utiliza el algoritmo de backtracking, que es la encargada de ir buscando la salida del laberinto. Como se ha implementado la aplicación siguiendo el paradigma de programación orientada a objetos, esta función será un método de la clase laberinto.

```
//Función de backtracking para encontrar una solución
voidprobarSolucion(intx,inty,bool&fin,puntovectorSolucion[N*N],int&ocupacion)
```

```
{
   if (fueraLaberinto(x,y)) fin = false; //Podamos el árbol de exploración
   else
   {
     if (mapa[x][y].isOcupado()) fin = false;
     else
     {
       mapa[x][y].setTipo(camino);
       vectorSolucion[ocupacion].setX(x); //Anotamos solución parcial
       vectorSolucion[ocupacion].setY(y);
       ocupacion++;
       if (x == N-1 && y == N-1) fin = true; /*Estado final. Solución completa*/
       else //Buscamos la solución completa
       {
         if (!haPasadoYa(x+1,y)) probarSolucion(x+1,y,fin,vectorSolucion,ocupacion);
         /*Decidimos buscar por abajo, aplicando previamente una nueva poda al árbol*/
         if (!fin) //Si no hemos encontrado la solución, seguimos buscando.
         {
           if (!haPasadoYa(x,y+1)) probarSolucion(x,y+1,fin,vectorSolucion,ocupacion);
         //Por la derecha
         }
         if (!fin)
         {
           if (!haPasadoYa(x-1,y)) probarSolucion(x-1,y,fin,vectorSolucion,ocupacion);
         //Por arriba
         }
         if (!fin)
         {
           if (!haPasadoYa(x,y-1)) probarSolucion(x,y-1,fin,vectorSolucion,ocupacion);
         //Por la izquierda
         }
         if (!fin) /*Si seguimos sin encontrar una solución al llegar a este punto es
porque por aquí no podemos salir del laberinto.*/
         {
           mapa[x][y].setTipo(imposible); //Marcamos la casilla como imposible
           ocupacion--; //Quitamos la casilla del camino
         }
       }
     }
   }
}
```

8 Conclusión

El backtracking es un esquema algorítmico que nos va a permitir resolver una gran variedad de tipos de problemas, pero, por término medio, es sumamente costoso (en cuanto a complejidades temporales se refiere) debido al tamaño que adoptan sus espacios de búsqueda. Aplicándole una serie de mejoras podremos conseguir que dicho espacio mengue para reducir en lo posible las altas complejidades que suelen tener.

9 Referencias

1]Black, Paul E. "Backtracking", in *DictionaryofAlgorithmsand Data Structures*, U.S. NationalInstituteof Standards andTechnology.
3]Romero, Santiago. Estrategias de programación.
2]UEx. Apuntes Estructuras de Datos y Algoritmos de 2º Ingeniería Informática. Tema 11: Esquemas algorítmicos.
4]Wikipedia – Problema del laberinto.
5] Apuntes de EDA 2004 -2005 UNEX
6] en.wikipedia.org/wiki/Backtracking
7] http://platea.pntic.mec.es/jescuder/c_magico.htm
8] http://students.ceid.upatras.gr/~papagel/project/kef5_8.htm
9] http://www.faust.fr.bw.schule.de/mhb/backtrack/mag5en.htm
10] http://www.lsi.upc.edu/~iea/TranspasJavier/BackTracking2.ppt