Algorítmica: práctica 4

El recorrido del caballo

Grupo 2

Sofía Almeida Bruno Antonio Coín Castro María Victoria Granados Pozo
Miguel Lentisco Ballesteros José María Martín Luque

1 de junio de 2017

Análisis del problema

Dado un tablero de ajedrez de tamaño N×N y un caballo colocado en una posición inicial, nuestro problema consiste en pasar por todas las casillas del tablero una sola vez realizando los movimientos permitidos para el caballo. Coloquialmente, se conoce como el **problema del recorrido del caballo.**

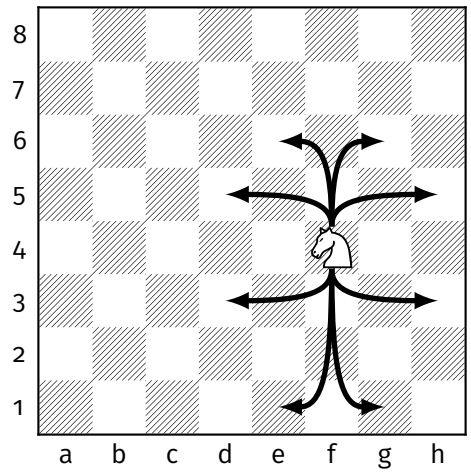

Figura 1: Movimientos posibles del caballo

Antes de comenzar con el diseño de un algoritmo que resuelva el problema, hemos consultado algunos artículos donde se discute el mismo. Por ejemplo, *Cull et al.* [1] probaron el siguiente resultado:

Teorema. *Todo tablero de tamaño* $N \times M$ *con* $\min(N, M) \geq 5$ *tiene un recorrido válido del caballo.*

Además, consultando la secuencia [2], observamos que el problema no tiene solución en un tablero $N \times N$ si $N = 2, 3, 4$. El caso $N = 1$ se considera trivial, por lo que pondremos especial hincapié en el caso $N \geq 5$.

Para representar nuestro problema, hemos usado una clase `TableroAjedrez`. En `TableroAjedrez` se encapsula la matriz que simboliza un tablero de ajedrez de tamaño N×N, hasta un máximo fijado. Además, disponemos de un `struct Posicion` que representa una casilla del tablero, para poder gestionar los posibles desplazamientos del caballo. Se adjunta el código fuente de la clase.

La solución es una secuencia de pasos que reflejan el movimiento del caballo a lo largo del tablero. A la hora de representarlo, modificaremos directamente el tablero, considerando que el contenido de la posición `(i,j)` es el paso en el que el caballo se situó en la casilla que representa dicha posición. Inicialmente, todas las casillas del tablero contienen un 0.

Técnica utilizada

Es obvio que podemos representar nuestro problema utilizando un grafo, en particular, un árbol de estados, donde los nodos serían los tableros con las posibles posiciones del caballo en cada paso. La raíz sería

un tablero con la posición inicial, que en cada nivel se va ramificando para permitirnos considerar todas las opciones de desplazamiento disponibles. De las dos técnicas estudiadas para exploración de grafos hemos utilizado *BackTracking* porque consideramos que se adapta mejor al problema que nos enfrentamos.

Branch&Bound es un procedimiento enfocado a resolver problemas de optimización, ya que se aprovecha del uso de cotas. En este caso, como simplemente queremos llegar a una solución sin minimizar ni maximizar ninguna función objetivo, no necesitamos esta técnica.

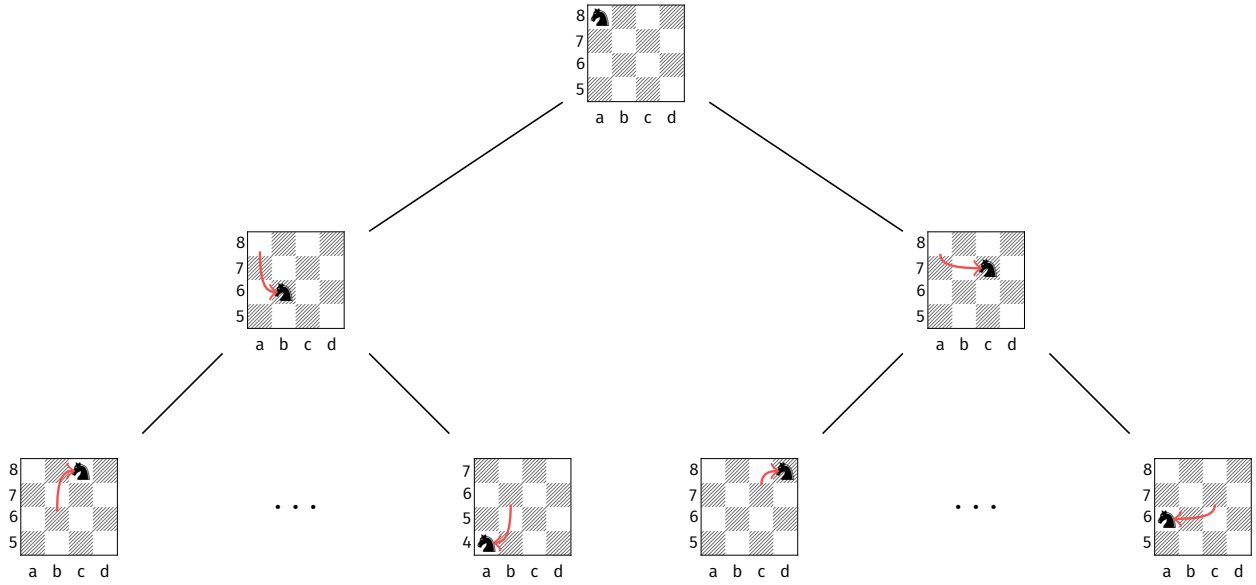

Figura 2: Representación del árbol de estados (sólo se muestran estados factibles)

Diseño de la solución

Empleando un enfoque basado en la técnica *BackTracking* para la resolución del problema, la idea es la siguiente: partiendo del tablero inicial exploraremos todos los tableros de movimientos posibles, y cada uno de ellos en profundidad. Finalizaremos cuando en alguno nos devuelva una solución.

Vamos a mostrar las componentes *BackTracking* de este problema:

- *Representación de la solución:* una matriz de enteros $M = (m_{ij})$ donde cada entrada representa el paso en el que el caballo se situó en la posición (i,j).
- *Restricciones implícitas:* el número de paso añadido tiene que ser menor o igual que N^2, donde N es la dimensión del tablero.
- *Restricciones explícitas:* el caballo no puede salirse del tablero ni situarse en una casilla por la que ya se ha pasado. Además, solo puede realizar movimientos válidos según las reglas del ajedrez.

- *Función objetivo:* encontrar un tablero en el que el caballo se ha desplazado por todas las casillas una única vez.
- *Función de poda:* un nodo será explorado si cumple: el caballo no se sale del tablero (la posición no supera el tamaño de la matriz para ninguna de sus componentes), y además no ocupa una casilla por la que se ha pasado (el valor de dicha casilla es diferente de 0, valor al que se inicializan todas las casillas del tablero).
- *Representación del árbol implícito:* en cada nivel i del árbol asignamos cada una de las posibles posiciones del caballo en el paso i-ésimo. Este árbol es el que muestra la Figura **??**.

Una vez que hemos visto las componentes, pasemos a ver un esqueleto o *pseudocódigo* del algoritmo:

Tablero[1, . . . , N][1, . . . , N] ≡ Representa el tablero. Tras finalizar, representa la solución
N ≡ Dimensión del tablero
pos ≡ Posición actual del caballo (i, j) en el tablero
k ≡ Paso actual

```
function MovimientosCaballoBT (Tablero[1, . . . , N][1, . . . , N], pos, k)
    Tablero[pos] ← k                                          ▷ Casilla elegida
    if k == n^2 then                                          ▷ Hay solución
        return true
    end if
    for mov in Movimientos válidos del caballo do
        pos ← pos + mov
        if esFactible(mov) then                               ▷ Función de poda
            correcto ← MovimientosCaballoBT(Tablero, pos, k + 1)   ▷ Explora siguiente nivel
            if correcto then
                return true
            else
                Tablero[pos] ← 0                              ▷ Movimiento sin solución
            end if
        end if
    end for
    return false                                              ▷ Sin solución en pasos siguientes
end function
```

Observamos que necesitamos una estructura de datos adicional, donde se almacenen los *movimientos válidos del caballo*. Dichos movimientos serán relativos a la posición que en ese momento ocupe el caballo, es decir, si la posición actual es (i, j), los movimientos válidos son alguno de (i ± 2, j ± 1) ó (i ± 1, j ± 2).

En la práctica, hemos implementado las *restricciones implícitas* como una comprobación para saber si se ha llegado a una solución, pues, por la construcción del algoritmo, cuando se llegue a N^2 pasos, se habrá conseguido la *función objetivo*. Además, la *función de poda* se ha separado y renombrado por simplicidad.

Algoritmo BackTracking

Veamos ahora el algoritmo implementado en el lenguaje C++. En primer lugar, hemos separado la implementación de la *función de poda*, para que se aprecie con claridad. Su funcionalidad es la que se describió anteriormente.

Código fuente 1: Criterio de Factibilidad

```
1 bool esFactible(const TableroAjedrez& tablero, const Posicion& pos) {
2   bool en_tablero = pos.i < tablero.getTam() && pos.i >= 0
3                  && pos.j < tablero.getTam() && pos.j >= 0;
4
5   return en_tablero && tablero[pos] == 0;
6 }
```

Ahora pasemos a la función principal. En primer lugar, se coloca en la casilla actual el número de paso que se recibe como argumento. A continuación, el algoritmo comprueba si está en el último paso, pues de ser así habrá encontrado solución y terminará, devolviendo **true**.

En caso contrario, el algoritmo se repetirá con cada una de las casillas a las que puede moverse el caballo desde la posición en la que se encuentra (cumpliendo las restricciones consideradas anteriormente), y aumentando en 1 el paso. Dichas posiciones se encuentran en un vector y se irán sumando a la posición actual para ir calculando las nuevas. Se comprueba si las posiciones son factibles con la función anteriormente mostrada.

Código fuente 2: Vector de movimientos válidos del caballo

```
1   Posicion movimientos[8] = { {2,1}, {1,2}, {-1,2}, {-2,1},
2                               {-2,-1}, {-1,-2}, {1,-2}, {2,-1} };
```

Al realizar la llamada recursiva, esta puede devolver *true* o *false*. En el primer caso, significaría que en el nivel siguiente se ha permitido seguir avanzando (o se ha llegado al final), por lo que el algoritmo finaliza devolviendo **true**. En el segundo caso, la razón sería que en el siguiente nivel no se ha encontrado ninguna posición válida para colocar el caballo. Entonces, tenemos que volver a poner a 0 la casilla por la que hemos intentado pasar, y continuar con el siguiente movimiento. En caso de que ningún movimiento sea factible, devolveremos **false**.

El algoritmo continúa repitiendo el paso anterior hasta que se haya pasado por todas las posiciones del tablero (notemos que no volverá a seleccionar nunca una casilla ya considerada). Cuando finalice el algoritmo, tendremos como solución una matriz donde cada casilla contiene el número de paso del caballo.

Código fuente 3: Algoritmo BackTracking

```
1 bool MovimientosCaballoBT(TableroAjedrez& tablero, const Posicion& pos_actual, int paso) {
2   tablero[pos_actual] = paso;
3
4   // Si estamos en el último paso
```

```
5   if (paso == tablero.getTam()*tablero.getTam())
6     return true;
7
8   for (int k = 0; k < 8; ++k) {
9     bool ok;
10    Posicion pos_nueva = pos_actual + movimientos[k];
11
12    if (esFactible(tablero, pos_nueva)) {
13      ok = MovimientosCaballoBT(tablero, pos_nueva, paso + 1);
14
15      if (!ok)
16        tablero[pos_nueva] = 0;
17      else
18        return true;
19    }
20  }
21
22  // Si sale del bucle, es que no puede colocar en ninguna casilla.
23  return false;
24 }
```

Por último, comentaremos el programa principal `main`, donde diferenciamos varios casos:

- Si no se pasa como parámetro la posición inicial generaremos una posición aleatoria hasta que encontremos una solución al problema. Este proceso, en general, puede ser muy costoso, pues como veremos la eficiencia del algoritmo para una posición dada es ya exponencial. La primera posición a comprobar será la (0, 0), pues es más probable que empezando en una esquina encontremos una solución.

- Si el tamaño del tablero menor que 5x5 nunca se obtendrá una solución, por tanto solo se ejecutará el algoritmo si $N \geq 5$. Sin embargo, si ejecutásemos el algoritmo, nos devolvería correctamente que no hay solución. Si el tamaño es 1, la solución (trivial) será la única casilla del tablero.

Ejemplo paso a paso

Supongamos que llamamos a nuestro programa pasándole como parámetro un 5, que indicará el número de filas y columnas.
Así, tendremos un tablero de tamaño 5×5 cuyos elementos están inicializado a 0. Como no pasamos ningún otro argumento al ejecutar el programa la posición inicial será la (0,0).
Invocamos el procedimiento BackTracking pasándole como parámetros el tablero ya inicializado, por referencia (para evitar el elevado coste que provocaría la copia, ya que llamaremos recursivamente a este procedimiento); la posición inicial; y 1 como número de paso.
Lo primero que se hace es actualizar el contenido de la posición pasada al número de paso correspondiente. Nuestro tablero tendría entonces un 1 en la posición (0,0) y el resto continuaría a 0. Se comprueba si estamos en el último paso y, como no lo estamos, continuamos con la ejecución de un bucle `for` en el que se irán explorando los movimientos que puede realizar el caballo. La posición nueva será (0, 0)+`movimientos[0]`= (0, 0) + (2, 1) (donde `movimientos` es el vector que almacena las posibles opciones de desplazamiento del caballo). Como esta posición se encuentra en el tablero y además no se ha pasado por ella, se cumple la condición para volver a llamar a la función BackTracking donde el paso será 2 e irá realizando el mismo algoritmo de forma recursiva. Una vez resuelta la recursividad, la función devolverá *true* o *false*.

Si devuelve *true* es porque ya ha finalizado, pero si devuelve *false* es porque a partir de la casilla actual no existe una solución, por lo que volvemos a poner a 0 su contenido (dicho contenido cambió al llamar a la función para esta casilla). Continúa con la ejecución del bucle actualizando la posición y ejecutando el mismo algoritmo otra vez para el paso 2.

Eficiencia teórica

Queremos hallar la eficiencia teórica de nuestro algoritmo en el peor de los casos. Esto ocurrirá cuando para llegar a la solución tengamos que recorrer el árbol implícito por completo. Dado que hay 8 movimientos del caballo posibles, en cada nivel el número de nodos se multiplica por 8 con respecto al nivel anterior. Seguiremos añadiendo niveles hasta que el tablero esté completo, es decir, hayamos recorrido las $N \times N$ casillas.

Por tanto, el número total de nodos que habría que recorrer sería la suma de los nodos de cada nivel, que podemos expresar como:

$$1 + 8 + 8^2 + \ldots + 8^{N^2} = \sum_{i=0}^{N^2} 8^i = \frac{8^{N^2} - 1}{8 - 1} \tag{1}$$

Como en cada nodo únicamente realizamos una serie de operaciones con eficiencia constante, afirmamos que la eficiencia del algoritmo es de orden $O\left(8^{N^2}\right)$. En particular, para un tablero estándar de ajedrez, el orden de eficiencia es:

$$O(8^{64}) = O(6.277 \cdot 10^{57})$$

Esta eficiencia exponencial la hemos comprobado en múltiples ocasiones a la hora de ejecutar el algoritmo, ya que aunque funciona correctamente, no da una solución en un tiempo razonable para todas las posiciones iniciales.

Otra forma de analizar la eficiencia teórica sería resolver la ecuación en recurrencia:

$$\begin{cases} T(1) = 8 \\ T(N) = 8T(N-1) \end{cases}$$

pues en cada paso se realizan, como mucho, 8 llamadas recursivas, donde habría una casilla menos por rellenar. Esta ecuación en recurrencia no es sino otra forma de expresar la progresión geométrica (1), ya que en cada paso se multiplica por un término constante, concretamente 8, que es la razón de la progresión.

Problema real

El problema del recorrido del caballo es un caso particular de un problema más general: hallar, si existe, un camino hamiltoniano [3] en un grafo. Si además imponemos que desde la casilla donde finaliza el caballo se pueda volver a la casilla de origen mediante un movimiento válido, el problema se transforma en hallar un camino cerrado o ciclo hamiltoniano [3].

No se nos ocurre ninguna situación real de aplicación del problema del recorrido del caballo, tal y como lo hemos enunciado. Sin embargo, si consideramos el problema de hallar un camino cerrado, existe una similitud con el problema del *viajante de comercio*, que consiste en hallar un ciclo hamiltoniano **minimal**

para un grafo completo, no dirigido y ponderado.

Compilación y ejecución del proyecto

Para compilar este proyecto simplemente hay que situarse en la carpeta `p4_backtracking` y ejecutar el comando `make`.

Una vez compilado, si estamos situados en la misma carpeta ejecutamos `./bin/main` pasando como parámetro el tamaño del tablero, y opcionalmente la posición inicial del caballo. Un ejemplo del formato completo sería el siguiente: `./bin/main 8 0 0`.

Si N es el tamaño del tablero, las posiciones del caballo comienzan en 0 y acaban en N – 1, tanto para las filas como para las columnas.

Referencias

[1] Paul Cull, Jeffery De Curtins. *Knight's tour revisited*. PDF.

[2] Sequence A165134 in the OEIS.

[3] Hamiltonian path problem on Wikipedia.