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

30 de mayo de 2017

Análisis del problema

Dado un tablero de ajedrez de tamaño $N \times N$ 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.**

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 lema:

Lema 1. *Todo tablero de tamaño* $5 \times M$ *con* $M \geq 5$ *tiene un recorrido válido del caballo.* Además, es conocido que dicho recorrido comienza en la posición más baja y más a la izquierda del tablero.

Además, consultando la secuencia [2], observamos que el problema no tiene solución 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 $N \times N$, 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.

Técnica utilizada

Es obvio que podemos representar nuestro problema utilizando un grafo, donde los nodos serían las casillas del tablero, y las aristas conectarían casillas adyacentes. 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 este tipo de algoritmo.

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, cada una de ellos en profundidad, finalizando cuando en alguno nos devuelva una solución.
Vamos a mostrar las componentes *BackTracking* de este problema:

- *Representación:* una matriz de enteros 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 $N \times N$.

- *Restricciones explícitas:* el caballo no puede salirse del tablero ni situarse en una casilla por la que ya se ha pasado.
- *Representación del árbol implícito:* en cada nivel i del árbol asignamos la posición del caballo en el paso i-ésimo.
- *Función objetivo:* encontrar un tablero en el que el caballo se ha desplazado por todas las casillas.
- *Función de poda:* al añadir un movimiento a la solución debe cumplir: el caballo no se sale del tablero (la posición no supera el tamaño de la matriz para ninguna de sus componentes), 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).

Una vez que hemos visto las componentes, pasemos a ver un esqueleto o *pseudocódigo* del algoritmo:
N ≡ número de nodos del grafo
M ≡ número de nodos del recubrimiento
L ≡ matriz de adyacencia
T[1, . . . , M] ≡ vector de nodos del recubrimiento
V[1, . . . , N] ≡ vector de incidencias

```
function MovimientosCaballoBT(Tablero[1, ..., n][1, ..., n], pos(i, j), k)
    Tablero[pos] ⟵ k                                          ▷ Casilla elegida
    if k == n * n then                                        ▷ Hay solución
        return true
    end if
    for mov = Movimiento de Caballo do
        pos ⟵ pos + mov
        correcto ⟵ true
        if mov valido then
            correcto ⟵ MovimientosCaballoBT(Tablero, pos, k + 1)   ▷ Llamada recursiva
            if correcto then
                return true
            else
                Tablero[pos] ⟵ 0                              ▷ Movimiento sin solución
            end if
        end if
    end for
    return false
end function
```

Como se puede observar, algunas de las componentes *greedy* listadas anteriormente se han omitido o modificado en la implementación, pues o bien no son necesarias, o bien es más sencillo implementarlas de otra manera.

- La *lista de candidatos* y la *lista de candidatos utilizados* están representadas como un vector de incidencias V, donde si el nodo i está utilizado, entonces V[i] = 0.

- El *criterio de factibilidad*, como ya dijimos, no es necesario, pues todos los nodos de la *lista de candidatos* son factibles. Implícitamente, estamos considerando como factibles los nodos i tales que V[i] > 0.
- La *función solución* es que el vector de incidencias V sea distinto del vector 0, o lo que es lo mismo, que el máximo de las componentes de V no sea 0. Esto significaría que ya se han considerado todas las aristas que inciden en cada uno de los nodos.
- La *función de selección* consiste en seleccionar, en cada paso, la componente máxima del vector V, es decir, el nodo con mayor grado.

En el diseño del algoritmo nos aprovechamos del vector V para identificar **cada nodo del grafo con cada posición de** V. Es decir, V[i] representa el número de incidencias del nodo i.

Algoritmo BackTracking

Veamos ahora el algoritmo implementado en el lenguaje C++. En primer lugar, hemos separado la implementación de la *función de selección*, para que se aprecie con claridad:

Ahora pasemos a la función principal. El algoritmo comienza construyendo un vector de incidencias, que contiene en cada posición i el número de aristas que inciden en el nodo i–ésimo. Después, seleccionamos el nodo con un número mayor de incidencias, lo añadimos a la solución y eliminamos las aristas que inciden en dicho nodo. A efectos prácticos, esto consiste en poner a 0 el número de incidencias del nodo en V, y en decrementar en 1 el número de incidencias de los nodos conectados con él.

El algoritmo continúa repitiendo el paso anterior hasta que no quedan aristas sin considerar (notemos que no volverá a seleccionar nunca un nodo ya considerado). Cuando finalice el algoritmo, tendremos como solución un vector con los nodos utilizados en el recubrimiento, y el coste será el número de nodos del mismo.

La solución `sol` utilizada es una instacia de la clase `Solucion`, que contiene un vector con los nodos y su tamaño. Esta función `RecubrimientoGrafoGreedy` recibe como parámetro un objeto `p` de la clase `Problema` que encapsula la matriz de adyacencia del grafo en cuestión, donde se ven reflejadas indirectamente las incidencias de cada nodo.

Ejemplo paso a paso

Eficiencia teórica

Queremos hallar la eficiencia de nuestro algoritmo en el peor de los casos. Esto ocurrirá en el caso de que para llegar a la solución tengamos que recorrer el árbol implícito por completo. Dado que hay 8 movimientos del caballo posibles, en cada nivel se multiplica por 8 el número de nodos del 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:

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

La eficiencia es, por tanto, $O\left(8^{N^2}\right)$. Esta eficiencia exponencial la hemos comprobado en múltiples ocasiones a la hora de ejecutar el algoritmo, ya que aunque el algoritmo funciona correctamente, no da una solución en un tiempo razonable para todas las posiciones iniciales.

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.