Práctica 2

Mikel Barrenetxea, Amaia Lobato
Diseño de algoritmos

El problema

Se trata de mover un caballo dentro de un tablero de ajedrez $n \times n$ de forma que partiendo de una posición inicial dada (x_1, x_2) y realizando solamente los movimientos permitidos al caballo en el juego del ajedrez, recorra exactamente una sola vez todas las casillas del tablero. Se pide buscar todas las soluciones posibles partiendo de (x_1, x_2).

Propuestas algorítmicas y complejidades

Para resolver el problema usando la búsqueda en profundidad programaremos el siguiente algoritmo en el lenguaje Python:

Algoritmo 1: Búsqueda en profundidad

Entrada: pos_in : posición inicial del caballo, tablero: tablero de ajedrez, sol_parc: solución parcial
Salida: Todas las soluciones posibles en las que el caballo recorre todas las casillas del tablero una única vez, empezando desde la posición inicial que se le indica

```
    Función caballo_profundidad (pos_in, tablero, sol_parc)
    principio
 1: n ← longitud(tablero)
 2: soluciones ← ∅
 3: sol_parc ← sol_parc ∪ pos_in
 4: i, j ← pos_in
 5: tablero[i][j] ← False
 6: si longitud(sol_parc) = n^2 entonces
 7:    soluciones ← soluciones ∪ sol_parc
 8: si no
 9:    para movimiento en posiciones_factibles(pos_in, tablero) hacer
10:       soluciones ← soluciones ∪ caballo_profundidad(movimiento, tablero, sol_parc)
11:    fin para
12: fin si
13: borrar sol_parc[-1]
14: tablero[i][j] ← True
15: devolver soluciones
```

En cambio, para resolver el mismo problema con la búsqueda en anchura hemos diseñado el siguiente algoritmo:

Algoritmo 2: Búsqueda en anchura

Entrada: pos_in : posición inicial del caballo, tablero: tablero de ajedrez
Salida: Todas las soluciones posibles en las que el caballo recorre todas las casillas del tablero una única vez, empezando desde la posición que se le indica

```
    Función caballo_anchura (pos_in, tablero)
    principio
 1: n ← longitud(tablero)
 2: i, j ← pos_in
 3: tablero[i][j] ← False
 4: V ← [[pos_in, tablero, [pos_in]]]
 5: soluciones ← ∅
 6: mientras longitud(V) ≠ 0 hacer
 7:    si longitud(V[0][2]) ≠ n^2 entonces
 8:       t← V[0][1]
 9:       recorrido ← V[0][2]
10:       para move ∈ posiciones_factibles(V[0][0],t) hacer
11:          i, j← move
12:          fila ← t[i][:]
13:          fila[j] ← False
14:          t[i] ← fila
15:          nuevo recorrido ← recorrido + [move]
16:          V ← insertar (V, [move, t[:][:], nuevo_recorrido])
17:          fila ← t[i][:]
18:          fila[j] ← True
19:          t[i] ← fila
20:       fin para
21:       V ← borrar(V, V[0])
22:    si no
23:       soluciones ← insertar(soluciones, V[0][2])
24:       V ← borrar(V, V[0])
25:    fin si
26: fin mientras
27: devolver soluciones
```

La función "posiciones_factibles"que usamos en ambos algoritmos es la siguiente:

Algoritmo 3: Posiciones factibles

Entrada: casilla: posición en la que está el caballo, tablero: tablero de ajedrez
Salida: Una lista con todos los saltos posibles que puede realizar el caballo

```
    Función posiciones_factibles (casilla, tablero)
    principio
 1: n ← longitud(tablero)
 2: i, j ← casilla
 3: l ← ∅
 4: para di, dj ∈ [(1,2),(1,-2),(-1,2),(-1,-2),(2,1),(2,-1),(-2,1),(-2,-1)] hacer
 5:    si 0 ≤ i+di < n   y   0 ≤ j+dj < n   y   tablero[i+di][j+dj] entonces
 6:       l ← insertar(l, (i+di, j+dj))
 7:    fin si
 8: fin para
 9: devolver l
```

La complejidad temporal de la función posiciones_factibles es constante en cualquier caso. Por lo tanto, su complejidad temporal no va a afectar en la de los algoritmos que contienen a esta.

El algoritmo de búsqueda en profundidad es recursivo. Su complejidad temporal depende de cuantas ramas salgan de cada nodo.

Para cada nodo se tienen k ramas nuevas, siendo k la cantidad de movimientos posibles desde la casilla donde estamos. Dependiendo de la casilla e iteración en la que esté el caballo, tendremos una cantidad diferente de movimientos. En general, $k \in [1, 8]$, excepto en la última iteración, donde k será 0. La profundidad máxima de una rama es n^2, es decir, recorrer el tablero de tamaño $n \times n$ una vez. Por lo tanto, $T(n) \in O(k^{n^2})$ donde $k \in [1, 8]$.

Para conseguir una buena aproximación podríamos estimar la cantidad media de ramas que suelen salir de cada nodo y usar esa cantidad como k para calcular la complejidad.

Para analizar la complejidad espacial del algoritmo de búsqueda en profundidad, tenemos que observar cuantas llamadas a la función guardamos en la pila y el tamaño de los elementos que guardamos. La profundidad de la rama nos dice cuantas llamadas a la función llevamos guardadas; por lo tanto, como máximo solo tendremos n^2 llamadas acumuladas en la lista. Pero el tamaño de la lista soluciones puede llegar a ser más grande que n^2; en el peor caso tendríamos k^{n^2-1} soluciones y cada solución es una lista de tamaño n^2. Por lo tanto, tenemos que $S(n) \in O(n^2 k^{n^2-1})$.

La complejidad temporal del algoritmo de búsqueda en anchura será cte$*\sum_{i=0}^{n^2-1} k^i = cte * (1 + k + k^2 + ... + k^{n^2-1})$, siendo $k \in \{1, .., 8\}$, ya que en el algoritmo solamente se hacen operaciones de coste 1 o constante (la función posiciones factibles). Por tanto, $T(n) \in O(k^{n^2-1})$, siendo $k \in \{1, ..., 8\}$.

En el caso del algoritmo de búsqueda en achura, no se realizan llamadas recursivas a la propia función. Para calcular la complejidad espacial tendremos en cuenta la longitud de la lista V. En el peor caso, la cantidad de elementos que tenemos guardado en V alcanza su máximo cuando estamos el último nivel del árbol. Entonces, V está formado por k^{n^2-1} listas (siendo $k \in \{1, .., 8\}$). Y los elementos de mayor tamaño de cada lista de V son de tamaño n^2 (tablero y solución parcial). Por tanto, tenemos que $S(n) \in O(n^2 k^{n^2-1})$.

Resultados

Una vez programados los algoritmos, hemos hecho varias pruebas para analizar sus complejidades temporales.

Primero, hemos analizado el algoritmo de búsqueda en profundidad. En primer lugar, hemos fijado la posición inicial en la casilla $(0, 0)$ y hemos analizado el tiempo de ejecución y la cantidad de nodos explorados para distintos tamaños de tablero. Analizar la cantidad de nodos explorados es interesante porque nos da una información más fiable de la complejidad temporal real.

	Tiempo (s)	Nodos explorados
$n = 3$	$6{,}953 \times 10^{-5}$	15
$n = 4$	0,009	2223
$n = 5$	5,563	1735079

Cuadro 1: Tiempos y nodos explorados partiendo desde la casilla $(0, 0)$ para diferentes tamaños de tablero para el algoritmo de búsqueda en profundidad.

En el Cuadro 1 podemos apreciar que a medida que el valor de n aumenta, los tiempos y los nodos explorados aumentan considerablemente.

Después de analizar la complejidad del algoritmo de búsqueda en profundidad, ahora vamos a analizar el algoritmo de búsqueda en anchura. Para ello, hemos seguido el mismo procedimiento que en el caso

anterior.

Primero, hemos fijado la posición inicial en la casilla $(0, 0)$ y hemos analizado el tiempo de ejecución y la cantidad de nodos explorados para distintos tamaños de tablero.

	Tiempo (s)	Nodos explorados
$n = 3$	$8{,}573 \times 10^{-5}$	15
$n = 4$	0,011	2223
$n = 5$	92,311	1735079

Cuadro 2: Tiempos y nodos explorados partiendo desde la casilla $(0, 0)$ para diferentes tamaños de tablero para el algoritmo de búsqueda en anchura.

En el Cuadro 2 podemos apreciar que a medida que el valor de n aumenta, los tiempos y los nodos explorados aumentan considerablemente.