Usando un caballo para recorrer el tablero de ajedrez

Por N. Aguilera

Mayo de 2019

Un problema de larga tradición es tratar de encontrar un recorrido de un tablero de ajedrez, visitando todas las casillas exactamente una vez, mediante movimientos de un caballo.

Por supuesto, el problema es un caso particular de los caminos y ciclos de Hamilton: visitar todos los vértices de un grafo exactamente una vez mediante un recorrido admisible.

Según Wikipedia, una de las primeras referencias al problema que se conocen es del siglo IX, en una obra poética escrita en sánscrito.

Quien primero parece haber abordado el problema matemáticamente fue Euler (¡cuándo no!), quien hacia 1759 Euler había encontrado recorridos tanto abiertos como cerrados (ciclos) en el tablero tradicional de 8×8. [(1)] Legendre y Vandermonde son otros matemáticos famosos que estudiaron el problema.

Se sabe que hay soluciones para tableros cuadrados de $n \times n$ para $n \geq 5$ (Conrad y otros, 1994), y hay recorridos que son ciclos para $n \geq 6$ (De Curtins y Cull, 1978). Es claro que no puede haber un ciclo en un tablero de $m \times n$ cuando m y n son impares, pues debe haber igual número de casillas «blancas» y «negras», ya que en un paso el caballo pasa de una casilla a otra con distinto color. Una descripción muy mencionada es la de Rouse Ball y Coxeter (1974), quienes cuentan parte de la historia y técnicas para su resolución, y un estudio sencillo de los casos es la dada por Schwenk (1991).

En el caso tradicional del tablero de 8×8, Löbbing y Wegener (1996) mencionan que hay 33 439 123 484 294 ciclos posibles (¡pero me perdí al contarlas!).

En los libros de introducción a la programación, es común ver algoritmos de "rastreo inverso" (*backtracking*), usando una variante de recorrido en profundidad. Personalmente conocí el problema muchos años atrás al leer el libro de Wirth (1987, p. 158).

Sin embargo, estos algoritmos en general no son muy eficientes, y tardan demasiado tiempo en resolver problemas de tamaño reducido.

Una heurística efectiva para acelerar el procedimiento es usar la *regla de Warnsdorf*, desarrollada hacia 1823 por H. C. von Warnsdorf: elegir el vecino que ingresa a la cola que tenga menor número de lugares disponibles.

Otra heurística efectiva, especialmente cuando se busca un ciclo, es comenzar desde una casilla cercana al centro.

Hay versiones más "científicas" que consisten en trabajar con tableros menores y después ensamblarlos. Por ejemplo, Parberry (1997) menciona algoritmos que usan $O(n^2)$ pasos.

---

(1) Ver http://eulerarchive.maa.org/hedi/HEDI-2006-04.pdf.

Al final de esta nota presentamos una versión en Python usando la heurística de Warnsdorf:

- Recordar que *en Python, los subíndices de las listas empiezan en* 0.
- Dependiendo del lector de pdf (Adobe Reader no tiene problemas), es posible extraer *caballo.py* como archivo de texto (cambiar la extensión de .txt a .py para su ejecución con Python).
  Alternativamente, se puede copiar de `http://oma.org.ar/invydoc/python`.
- Por defecto, el programa da el ejemplo en el tablero de 8×8 empezando desde un punto central y formando un ciclo.
- Python es bueno para hacer programas sencillos pero no es muy rápido. Programas equivalentes en C son mucho más rápidos.

Referencias

A. Conrad, T. Hindrichs, H. Morsy y I. Wegener, 1994. Solution of the knight's hamiltonian path problem on chessboards. *Discrete Applied Mathematics*, 50:125–134.

J. De Curtins y P. Cull, 1978. Knight's tour revisited. *The Fibonacci Quarterly*, 16(3):276–285.

M. Löbbing y I. Wegener, 1996. The number of knight's tours equals 33,439,123,484,294 — counting with binary decision diagrams. *The Electronic Journal of Combinatorics*, 3.

I. Parberry, 1997. An efficient algorithm for the knight's tour problem. *Discrete Applied Mathematics*, 73:251–260.

W. W. Rouse Ball y H. S. M. Coxeter, 1974. *Mathematical Recreations and Essays*. University of Toronto Press. Reimpreso por Dover, 1987.

A. J. Schwenk, 1991. Which rectangular chessboards have a knight's tour? *Mathematics Magazine*, 64:325–332.

N. Wirth, 1987. *Algoritmos y Estructuras de Datos*. Prentice-Hall Hispanoamericana.

El archivo caballo.py

```
"""Recorrido del caballo en un tablero de ajedrez de m x n.

- Buscamos un recorrido que empieza y termina en el mismo
    lugar, pasando por todos los cuadrados y sin repetir
    ninguno salvo el primero y el último, o sea, buscamos un
    ciclo de Hamilton.

- Trabajamos en un tablero de m x n, con casillas (i, j), 0 <=
    i < m, 0 <= j < n. m es el ancho y n la altura.

- Pedimos m, n >= 6, y alguno de ellos par para asegurar la
    existencia del ciclo.

- El algoritmo empieza su búsqueda desde un punto "raíz", y
    usa una técnica de "rastreo inverso" (backtracking).

- Usamos la heurística de Warnsdorf que acelera enormemente el
    algoritmo. No hay problemas para un tablero de 12 x 12.

- Por defecto la raíz está cerca del centro del tablero para
    que funcione la heurística, pero se puede cambiar.

"""

#-----------------------------------------------
# Dimensiones: para que exista un ciclo pedimos que ninguno sea
# par, ambos mayores que 5. Hay otros casos donde existe ciclo.

m = 8
n = 8
ciclo = True    # buscar recorrido cerrado (ciclo)

#-----------------------------------------------
# - El vértice correspondiente a (i, j) es k = 1 + i + j * m,
#         i, j = divmod(k - 1, m)
# - Los vecinos de (i, j) son de la forma
#         (i ± 2, j ± 1)   y   (i ± 1, j ± 2)
#     siempre que queden en el tablero.

def kdeij(i, j):
    return 1 + i + m * j

def ijdek(k):
    return divmod(k - 1, m)

#-----------------------------------------------
# ponemos la raíz en el centro para que sea rápido
```

```
raiz = kdeij(m // 2, n // 2)

#-------------------------------------------
# Construcción de vecinos

mn = m * n      # cantidad de vértices
mn1 = mn + 1    # para range

vecinos = [[] for k in range(mn1)]
vecinos[0] = None

# No hay vecinos con un mismo j,
# Ponemos sólo los de arriba (los de abajo se dan por simetría).

vecinos = [[] for v in range(mn1)]
vecinos[0] = None

for i in range(m):
    for j in range(n):
        k = kdeij(i, j)
        ii = i - 2
        if ii >= 0:
            jj = j + 1
            if jj < n:
                kk = kdeij(ii, jj)
                vecinos[k].append(kk)
                vecinos[kk].append(k)
        ii = i - 1
        if ii >= 0:
            jj = j + 2
            if jj < n:
                kk = kdeij(ii, jj)
                vecinos[k].append(kk)
                vecinos[kk].append(k)
        ii = i + 1
        if ii < m:
            jj = j + 2
            if jj < n:
                kk = kdeij(ii, jj)
                vecinos[k].append(kk)
                vecinos[kk].append(k)
        ii = i + 2
        if ii < m:
            jj = j + 1
            if jj < n:
                kk = kdeij(ii, jj)
                vecinos[k].append(kk)
                vecinos[kk].append(k)

#-------------------------------------------
```

```
# Función a usar
#-----------------------------------------------
def hamilton(m, n, vecinos, raiz=1, ciclo=True):
    """Retorna un recorrido de caballo cerrado en un tablero
        de m x n empezando desde "raiz".

    - Si no existe el recorrido pedido retorna la lista vacía.

    - Usa la heurística de Warnsdorf.

    """

    def grado(v):
        a = [u for u in vecinos[v] if padre[u] == None]
        return len(a)

    def visitar(u):
        """Recorrer el grafo desde la raíz."""
        nonlocal cuenta, llegamos
        cuenta = cuenta + 1

        if llegamos:
            return
        if len(ciclov) < mn:            # todavía no llegamos
            a = [v for v in vecinos[u] if padre[v] == None]
            if a != []:
                a.sort(key=grado)
                for v in a:
                    padre[v] = u
                    ciclov.append(v)
                    visitar(v)
                    if llegamos:
                        return
                    padre[v] = None
                    ciclov.pop()
        elif (len(ciclov) == mn):
            if not ciclo:
                llegamos = True
            if (u in vecinosraiz):
                llegamos = True
                ciclov.append(raiz)
        # en otro caso len(ciclov) == mn1

    mn = m * n      # cantidad de vértices
    mn1 = mn + 1    # para vértices entre 1 y mn

    vecinosraiz = vecinos[raiz]

    ciclov = [raiz]     # empezamos desde la raíz
    padre = [None for v in range(mn1)]
```

```
    padre[raiz] = raiz

    cuenta = 0
    llegamos = False

    visitar(raiz)

    print(40 * '-')
    print('Cantidad de entradas a visitar:', cuenta)
    if llegamos:
        return ciclov
    print('No hay ciclo de hamilton')
    return []

#-----------------------------------------------
# fin de función
#-----------------------------------------------

ciclo = hamilton(m, n, vecinos, raiz, True)

#-----------------------------------------------
print(40 * '-')
print('Ciclo resultante:')
print(ciclo)
print('Longitud:', len(ciclo))

#-----------------------------------------------
# Representación como tablero
#-----------------------------------------------

# cantidad de espacios para que no se superpongan los números
s = len(str(mn1)) + 1
formato = '{:' + str(s) + '}'

# buscamos la inversa del ciclo
pos = [None for v in range(mn1)]
for p in range(mn):
    pos[ciclo[p]] = p + 1

print(40 * '-')
print('Recorrido del tablero (posiciones en cada movimiento):')
for j in range(n-1, -1, -1):
    k = 1 + m * j
    for i in range(m):
        print(formato.format(pos[k + i]), end='')
    print()
```