El problema del tour del caballo y una solución óptima

Estructura de Datos y Algoritmos

Ricardo Ivan González Franco
ivan.gonzalez@cimat.mx

Universidad de Guanajuato – 11 de junio de 2019

1. El problema

El tour del caballo consiste en que, dado que tu caballo inicia en una casilla específica de un tablero de $m \times n$, encontrar una secuencia de movimientos tal que visites todas las casillas del tablero sin visitar una casilla más de una vez y usando los movimientos permitidos de un caballo de ajedrez normal.

Este problema es conocido desde la edad media, donde Al-Adli en su obra «Kitab ash-shatranj» (Libro del ajedrez). Cull y Conrad probaron que en cualquier tablero rectangular cuya dimensión de al menos 5, existe un camino del caballo.

2. Implementación

Existen varias maneras de implementarlo:

- Fuerza bruta: intentando todos los posibles caminos, que resulta imprácticco ya que, por ejemplo, en un tablero de 8×8 existen $4 \cdot 10^{51}$ posibles secuencias de movimientos, sobrepasando la capacidad de las computadoras actuales.
- Usando la heurística de Warnsdorff's.

En este proyecto se utilizó una pila para llevar un registro de los movimientos anteriores y poder usar la técnica de «backtracking» que consiste en que al no existir un movimiento en un determinado tiempo, regresar a turnos

anteriores y buscar un camino en un tiempo anterior. Para este programa se crearon 2 clases, `Node` y `KnightsTour`.

La primera clase representa una casilla por lo que se guarda su posición en el tablero, el número de turno que corresponde a dicha casilla y un array de booleanos donde se guardan la dirección de los movimientos que ya se han llevado a cabo desde esa casilla.

La segunda clase representa un tour del caballo donde se inicializa con el tamaño del tablero; aquí se implementan las funciones que ayudan a encontrar un camino. Tenemos un array de los 8 movimientos posibles que puede hacer el caballo en orden contrario a las manecillas del reloj y empezando por la derecha superior, además de un array que representa el tablero de ajedrez donde se guarda la secuencia de pasos para después visualizar el camino. Las funciones `backtrack`, `getmove` y `nodegrade` son las principales de la heurística y se explicarán en la siguiente sección. Las funciones `isthereapath` y `countpaths` son las principales que inician la búsqueda del camino, ambas mueven la posición inicial por todo el tablero pero la primera regresa un booleano al encontrar el primer camino y la segunda cuenta todos los posibles caminos.

La función `findpath` encuentra un camino iniciando con el caballo en las coordenadas que se le pasan como parámetros, y recibe uno opcional que en vez de encontrar el primero, cuenta todos los posibles caminos; esto lo hace de la siguiente manera: mientras existan movimientos anteriores, si aún no estamos en el turno del tamaño del tablero, es decir, en el último turno, en la casilla no se han probado todas direcciones de movimientos o aún se pueda mover entonces obtén el siguiente movimiento, regístralo en la casilla actual y haz un push en el stack del nuevo movimiento; si cualquiera de las condiciones anteriores no se cumple entonces en este turno no podemos hacer nada por lo que haz un *backtracking* y desecha el turno actual. Nótese que esto último se logra haciendo un pop desde el principio pero nunca hacer un push del nodo actual si no continuando a la siguiente iteración.

Además de todo lo anterior, se incluye un archivo `Makefile` para compilarlo. Todo el código se subió a `https://github.com/scratchmex/knightstour`.

2.1. Heurística

Este algoritmo utiliza el *backtracking* para evitar caminos que no lleven a ningún lado por lo que esa función lo que hace es borrar de la casilla actual el turno para representar como si nunca hubiera pasado por ahi. Es de notar que esta técnica permite implementar fuerza bruta de manera sencilla pero no es una solución óptima.

También se utiliza la regla de Warnsdorff's que nos dice que para elegir

el siguiente movimiento te fijes en la casilla que hará que el caballo tenga la menor cantidad de siguientes posibles movimientos. En la función `getmove` se implementa este heurística de la siguiente manera: para cada movimiento posible desde la actual casilla se obtiene el grado de la casilla efectuando dicho movimiento siempre y cuando esta casilla no esté ya visitada, elige la de menor grado para regresarlo si este movimiento no se ha efectuado ya por el nodo actual; si ningún movimiento es posible entonces regresa -1.

3. Tiempos de ejecución

La complejidad únicamente usando *backtracking*, es decir, fuerza bruta, es de $\mathcal{O}(5^n)$ y al implementar la regla de Warnsdorff's baja considerablemente hasta $\mathcal{O}(n)$. A continuación se presenta una comparación de los tiempos de ejecución medidos con el comando de Linux `time <ejecutable>`.

Tamaño del tablero	Tiempo de ejecución
7x8	0m6.7s
8x8	>5m18s (no se acabó de ejecutar)

Tabla 1: Sin heurística

Tamaño del tablero	Tiempo de ejecución
7x8	0m0.015s
8x8	0m0.014s
9x9	0m0.014s
12x12	0m0.014s
119x120	0m0.014s
120x120	>6m5.8s (no se acabó de ejecutar)
150x160	0m0.074s
180x220	0m0.130s
200x220	0m0.124s
220x220	0m0.211s
300x400	0m0.319s
300x450	>5m1.3s (no se acabó de ejecutar)

Tabla 2: Con heurística