# Análisis y resolución del problema del caballo en Java

Héctor Abraham Galván García

***Abstract*—En el presente reporte se examina el procedimiento para resolver el problema del caballo de ajedrez mediante un algoritmo ávido, usando una estrategia propia y adjuntando los resultados obtenidos en un programa de Java.**

**Palabras clave: Ajedrez, Algoritmo Ávido**

## I. Introducción

Los algoritmos ávidos son métodos sencillos que ayudan a resolver problemas de optimización. Para solucionar un problema ávidamente éste debe tener n entradas de las cuales obtener un subconjunto que satisfaga alguna restricción definida por el sistema. Aquellos subconjuntos que cumplan con las restriciones se les llama soluciones prometedoras.

Si un problema es susceptible a resolverse mediante un algoritmo ávido debe tener las siguientes características:

1. N entradas.
2. Función de selección que determine una entrada idónea para formar una solución.
3. Función que compruebe si alguna entrada es prometedora.
4. Una función objetivo que determine el valor de la solución hallada.
5. Una función que compruebe si un subconjunto es solución o no.

En este reporte se examina y soluciona el problema del caballo de ajedrez mediante un algoritmo ávido.

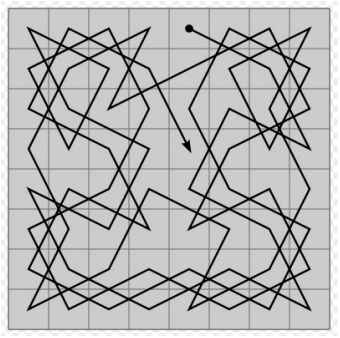

Fig. 1. Ejemplo de la solución del problema

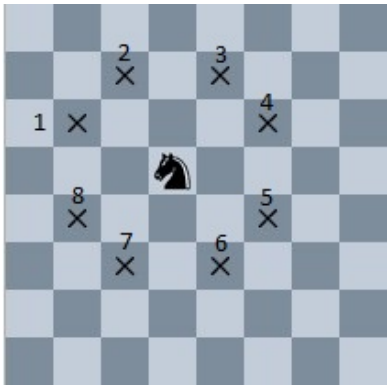


Fig. 2. Estrategia elegida

## II. Problema del caballo

El problema del caballo es un problema de matemáticas antiguo que consiste en que, dada una cuadrícula de $nxn$ y un caballo de ajedrez colocado en una posición cualquiera $(x, y)$, el caballo pase por todas las casillas una sola vez. Matemáticos han encontrado muchas maneras de resolverlas, aquí se propondrá una propia que tal vez se parezca. En teoría de grafos, el problema del caballo se puede ver como un camino hamiltoniano. Se ejemplifica la solución de este problema con la siguiente imagen:

Como hay muchas maneras de resolverlo, la que se usará es la siguiente: dada una posición $(x, y)$, se empezará a mover el caballo al movimiento con menos movimientos posibles siguientes, siempre en el sentido de las manecillas del reloj, respetando el movimiento del caballo, que no puede pasar por la misma casilla dos veces y que no se puede salir del tablero. Se muestra en la imagen el movimiento:

En la programación, se maneja una clase *Coordenada* de donde se obtiene la posición del caballo y siguiente posible posición. El tablero se inicializa en ceros y cada vez que pase por alguna posición, esa posición se setea en 1. Hay una función *posibleSiguienteMovimiento* que, como su nombre lo indica, calcula los movimientos posibles en un switch, puesto que el movimiento del caballo siempre será el mismo esté en la posición que esté.

Una función *validarSiguienteMovimiento* verifica que el siguiente movimiento esté dentro de los límites y que no vaya a ser una posición ya visitada(con un 1) y otra función llamada *recorrido* que va aumentando el número de movimientos que fue posible realizar mediante la ejecución de todas las otras funciones dentro de ésta. Esta función tambien valida que se salga del ciclo si ya no encuentra otro movimiento posible.

## III. Pruebas

Se probó para varias instancias de posición del caballo en un tablero de 8x8 completando siempre el camino con 64 movimientos.

Se probó para varias instancias de posición del caballo en un tablero de 5x5 completando el camino aunque no para todos los casos, se adjunta la imagen de una prueba con éxito iniciando en $(0, 0)$ y un fallo inciando en $(4, 3)$

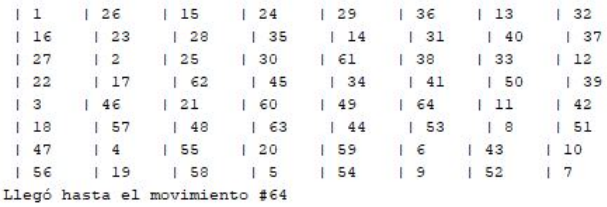


Fig. 3. Prueba en tablero de 8x8 iniciando en (0,0)

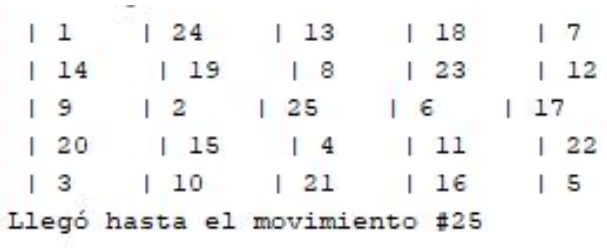


Fig. 4. Éxito de 5x5 en (0,0)

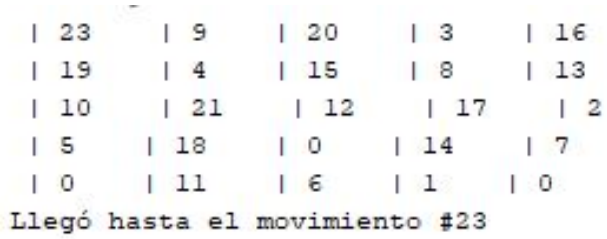


Fig. 5. Fallo de 5x5 en (4,3)

En un tablero de 4x4 (el mínimo posible para que se pueda mover el caballo) no se encontró un camino que visitara todas las casillas.

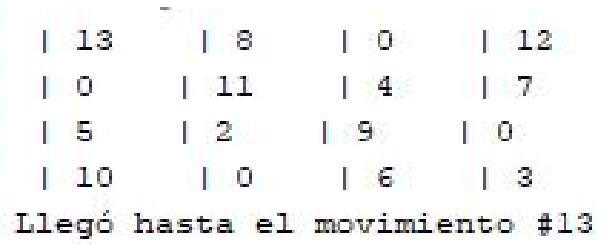


Fig. 6. Fallo de 4x4 en (0,0)

## IV. Conclusiones

A pesar de que existen muchas soluciones para este problema, se pudo llegar a una solución bastante aceptable para el problema. Son muy pocos los casos para los que falla y no se puede terminar el camino hamiltoniano. Eso hace pensar que programarlo para un ciclo hamiltoniano es más difícil y tal vez solo con solución para n muy grande.

Además, la teoría de los algoritmos ávidos vista en clase se entendió con el desarrollo de este software, puesto que se pudo identificar cada uno de los pasos expuestos en la introducción en las funciones implementadas en el código, cumpliendo los requisitos para ser voraz.