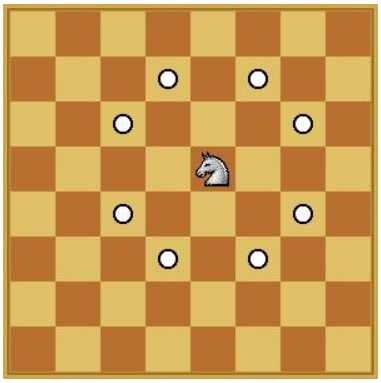
Análisis de Algoritmos

”Caballo Ávido”

Solución del problema Caballo Ávido



***Docente**: Roberto Oswaldo cruz Leija.*

***Alumna:** Paulina Medrano Hurtado.*

28/10/2019

Introducción

En el desarrollo de esta práctica se pretende resolver de forma ávida el problema del caballo, contemplando todas las restricciones que ya se dieron a conocer anteriormente y a la vez identificar o bien razonar correctamente la forma en la que se da la solución a dicho problema. Dejando claro un algoritmo voraz.

Términos

Los algoritmos voraces suelen ser bastante simples. Se emplean sobre todo para resolver problemas de optimización, como por ejemplo, encontrar la secuencia óptima para procesar un conjunto de tareas por un computador, hallar el camino mínimo de un grafo, etc. Habitualmente, los elementos que intervienen son:

- un conjunto o lista de ***candidatos*** (tareas a procesar, vértices del grafo, etc);
- un conjunto de ***decisiones*** ya tomadas (candidatos ya escogidos);
- una ***función*** que detemina si un conjunto de candidatos es una ***solución*** al problema (aunque no tiene por qué ser la óptima);
- una ***función*** que determina si un conjunto es ***completable***, es decir, si añadiendo a este conjunto nuevos candidatos es posible alcanzar una solución al problema, suponiendo que esta exista;
- una ***función*** de selección que escoge el candidato aún no seleccionado que es más ***prometedor***;
- una ***función objetivo*** que da el valor/coste de una solución (tiempo total del proceso, la longitud del camino, etc) y que es la que se pretende maximizar o minimizar;

Para resolver el problema de optimización hay que encontrar un conjunto de candidatos que optimiza la función objetivo. Los algoritmos voraces proceden por pasos. Inicialmente el conjunto de candidatos es vacío. A continuación, en cada paso, se intenta añadir al conjunto el mejor candidato de los aún no escogidos, utilizando la función de selección. Si el conjunto resultante no es completable, se rechaza el candidato y no se le vuelve a considerar en el futuro. En caso contrario, se incorpora al conjunto de candidatos escogidos y permanece siempre en él. Tras cada incorporación se comprueba si el conjunto resultante es una solución del problema. Un algoritmo voraz es correcto si la solución así encontrada es siempre óptima. El esquema genérico del algoritmo voraz.

Descripción del problema

"El Problema del Recorrido del Caballo de Ajedrez"

Pascual PEIRÓ CODINA

La peregrinación del caballo de ajedrez consiste en su paseo por todas las casillas del tablero sin pasar dos veces por la misma, utilizando sus movimientos: dos casillas horizontales y una vertical o a la inversa. Cuando desde la última casilla podamos pasar a la primera se trata de una "peregrinación cerrada".

A lo largo de los siglos, matemáticos de todo el mundo dedicaron un especial interés a este problema. Uno de ellos se destacó por sus ingeniosas e increíbles soluciones, Leonard Euler (Basilea - Suiza, 1707-1783). Una de sus soluciones es un recorrido en el que las filas y columnas suman 260. El desarrollo de los movimientos del caballo por las 64 casillas ya es, en sí, muy difícil de conseguir como para, además, lograr que filas y columnas sumen lo mismo.

Podemos imaginarnos cómo Euler, en el siglo XVIII, trabajó para resolver el acertijo, sus herramientas fueron una pluma, papel, mucha paciencia y una grandísima dosis de algo que demostró toda su vida: genialidad. Así, la técnica empleada sería, básicamente, la misma que he utilizado con ayuda de un programa informático diseñado en B.A.S.I.C. para este estudio, realizar movimientos del caballo de una a otra casilla hasta que, en el caso de que no queden casillas vacías, es necesario volver atrás, borrar el movimiento anterior y realizar un salto de caballo distinto. El proceso se repite hasta que se complete el recorrido.

A Euler le hubiera encantado disponer de la capacidad de cálculo de un ordenador. Al programa en B.A.S.I.C., en cambio, me encantaría añadirle la genialidad que demostró este gran matemático. A diferencia de los programas para jugar a ajedrez, este no da prioridad a los movimientos, el caballo puede mover, como máximo, a ocho casillas y, como mínimo, a dos realizándose los ocho posibles movimientos por orden. Así, intenta el primer movimiento, y si este no es posible porque la casilla está ocupada, intenta el segundo y así, sucesivamente, hasta que completa todas las posibilidades.

Teniendo en cuenta que disponemos de 64 casillas y en cada movimiento de dos a ocho posibles casillas para mover el caballo, podemos hacernos una idea del gran número de posibilidades. Sin entrar en cálculos con grandes números, traducido a tiempo real, el algoritmo podría resultar “infinito”. Por supuesto, se puede encontrar una solución en unos minutos o segundos si el ordenador realiza los movimientos adecuados. Esto me indujo a pensar en realizar recorridos con menos casillas que se pueden completar en pocos segundos, por ejemplo en un tablero de 4x4, de 5x5, de 3x8.

Código

```
/*
 * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Caballo;

/**
 *
 * @author HP
 */
public class caballoA {

    private final int nFilas;
    private final int nCol;
    private final int[][] tablero;
    private int     cont;

    public caballoA(int n) {
        cont =0;
        nFilas = n;
        nCol = n;
        //creamos un tablero de n*n
        tablero     = new int[n][n];
    }

    @Override
    public String toString() {
        for (int[] fila : tablero) {
            for (int j = 0; j < fila.length; j++) {
                System.out.printf(" %2d ", fila[j]);
            }
            System.out.println();
            for (int j = 0; j < fila.length; j++) {

            }
            System.out.println();
        }
        return "cont "+this.cont ;
    }


 public boolean Solucion(int f, int c) {
```

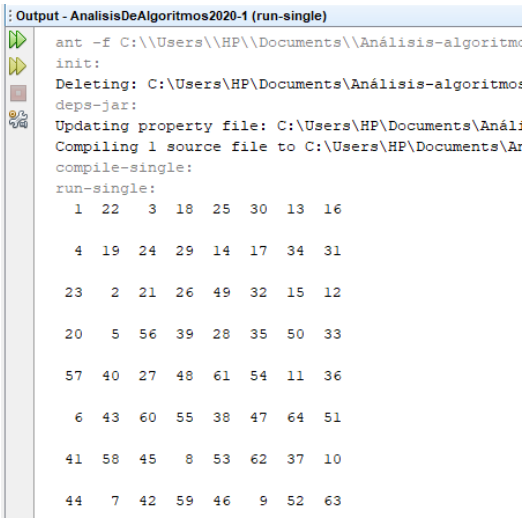
```
        cont++;
        tablero[f][c] =cont;
        //caso base
        if(cont==nFilas*nCol) return true;
        //caso especifico
        int[][] posibles = Casillas(f, c);
        //se buscan todos los movimientos posibles de la posicion
        if (posibles.length != 0 ){
            // Ordeno los datos por el menor num de movimiento de la
siguiente posicion
            Movimientos(posibles);
            // si la siguiente casilla tiene solucion, ese camino es
una solucion
            if (Solucion(posibles[0][0], posibles[0][1])) {
                return true;
            }
        }

        return false;
 }


// nuevo método: ordena el arreglo de casillas a saltar
void Movimientos(int[][] movimientos) {
    //Ordenamiento burbuja para cambiar el orden en el que se
encuentran los saltos
    for(int i=0; i<movimientos.length; i++) {
        for(int j=i+1; j<movimientos.length; j++) {
            //Vease como se calcularan los movimientos de cada casilla
            int cuantosPosiblesI = Casillas(movimientos[i][0],
movimientos[i][1]).length;
            int cuantosPosiblesJ = Casillas(movimientos[j][0],
movimientos[j][1]).length;
            //Orden por los menores movimientos posibles de la sig
posición
            if(cuantosPosiblesJ<cuantosPosiblesI) {
                int tmp0 = movimientos[i][0];
                movimientos[i][0] = movimientos[j][0];
                movimientos[j][0] = tmp0;
                int tmp1 = movimientos[i][1];
                movimientos[i][1] = movimientos[j][1];
                movimientos[j][1] = tmp1;
            }
        }
    }
}

    int[][] Casillas(int x, int y) {
        //cantidad maxima de casillas que pueden ser un movimieto
        int[][] resp = new int[8][2];
```

```
        // inicialización de casillas disponibles
        int     aux  = 0;


        //la coordenada a donde va a salta
        //Chequeo de posiciones validas
        if(pmh(x-2,y-1)){ resp[aux][0]=x-2; resp[aux++][1]=y-1; }
        if(pmh(x-2,y+1)){ resp[aux][0]=x-2; resp[aux++][1]=y+1; }
        if(pmh(x-1,y-2)){ resp[aux][0]=x-1; resp[aux++][1]=y-2; }
        if(pmh(x-1,y+2)){ resp[aux][0]=x-1; resp[aux++][1]=y+2; }
        if(pmh(x+2,y-1)){ resp[aux][0]=x+2; resp[aux++][1]=y-1; }
        if(pmh(x+2,y+1)){ resp[aux][0]=x+2; resp[aux++][1]=y+1; }
        if(pmh(x+1,y-2)){ resp[aux][0]=x+1; resp[aux++][1]=y-2; }
        if(pmh(x+1,y+2)){ resp[aux][0]=x+1; resp[aux++][1]=y+2; }
        int[][] tmp = new int[aux][2];
        for(int i=0; i<aux; i++) { tmp[i][0] = resp[i][0];
tmp[i][1]=resp[i][1]; }
        //retorna la matriz del tamaño del numero de casillas
disponibles [2] siendo el num de casillas en coordenadas
        return tmp;
    }

    boolean pmh(int f, int c) {
                return !(f<0 || f>nFilas-1 || c<0 || c>nCol-
1||tablero[f][c]!=0);
    }
}
```

Resultados

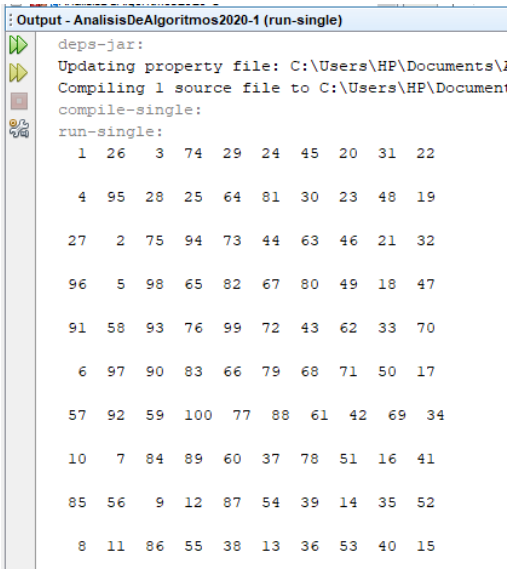
Tablero de 8*8

Iniciando en coordenada (0,0)



Tablero de 10*10

Iniciando en coordenada (0,0)



Conclusiones

En el resultado de la implementación del algoritmo ávido que da solución al problema planteado, el caballo se pudo obtener buenos resultados a lo que se estuvo comprobando. En un inicio se tenía una idea algo clara puesto que el problema ya había sido analizado y explicado en código. Lo cual fue muchísima ayuda para poder desarrollarlo e implementarlo.

Con los distintos tableros y coordenadas que estuve modificando los resultados eran favorables o bien correctos. En cada uno de los distintos casos me imprimía todo el tablero con los movimientos y también hasta donde llegó.

Bibliografía

http://casanchi.com/rec/caballo01.htm

https://thales.cica.es/rd/Recursos/rd99/ed99-0033-04/voraz_introd.html