

Ejercicios resueltos de programación 3

Prácticas de la asignatura.

Hemos creado un documento aparte donde iremos guardando los ejercicios de las distintas prácticas hasta el momento propuestas por el equipo docente. Al ser un documento aparentemente sencillo, ya que en ocasiones dependerá de la solución lo que el alumno haya implementado en su práctica (es bastante subjetivo), no en todas las preguntas se pondrá la solución. La idea de hacer este documento es ver la tónica general a la hora de preguntar y además completar los ejercicios que faltan.

No tendremos un índice como tal, ya que estas cuestiones forman un conjunto, por tanto, empezaremos a ver estas preguntas.

Febrero 2000-1ª (ejercicio 2)

Enunciado: Comenta las condiciones de poda que has utilizado en la realización de la práctica del Nim

Respuesta: Este ejercicio es totalmente subjetivo como hemos comentado antes, pero podremos decir basándonos en el libro de Brassard, que las condiciones de poda podrán ser aquellas que no lleguen a ganar el juego o bien en tablas. Por ello, se tendría que hacer el árbol de expansión hasta llegar a solución. De todas maneras, para ello en el citado libro vendría más expandido, incluso más que en nuestro resumen de la asignatura.

Febrero 2006-1ª (ejercicio 2)

Enunciado: Declara en Java o en Modula-2 las clases y/o estructuras de datos que utilizarías en el problema del Su-doku (práctica de este año) para comprobar en un tiempo de ejecución constante respecto a la dimensión n del tablero ($n \times n$) que una casilla contiene un valor factible.

Respuesta: Este tipo de problemas creo que es preferible implementarlos en Java, antes que en Modula-2, ya que será el lenguaje que posteriormente se use. Este año fue especial, ya que al ser de transición se dejaron ambos lenguajes.

Existen varias alternativas, pero de nada sirve que verificar una casilla se realice en tiempo constante si luego utilizar su valor se realiza en tiempo lineal. En la siguiente solución se declaran tres tablas de booleanos que indican si ya hay o no un determinado valor en una determinada fila, columna o región (cuadrante), respectivamente. Si no es así, entonces es factible poner el valor en la casilla. Tanto la función de verificación como las de actualización tienen un coste computacional constante.

```
public class Tablero {
    int N = 9;
    int subN = (int)Math.sqrt(N);
    boolean val_en_fil [][] = new boolean [N] [N];
    boolean val_en_col [][] = new boolean [N] [N];
    boolean val_en_reg [][] = new boolean [N] [N];

    boolean valorFactible (int fil, int col, int val) {
        int reg = region (fil, col);
        return (!val_en_fil [fil][val] &&
                !val_en_col [col][val] &&
                !val_en_reg [reg][val]);
    }

    void poner (int fil, int col, int val) {
        int reg = region (fil, col);
        val_en_fil [fil][val] = true;
        val_en_col [col][val] = true;
        val_en_reg [reg][val] = true;
    }
}
```

```

void quitar (int fil, int col, int val) {
    int reg = region (fil, col);
    val_en_fil [fil][val] = false;
    val_en_col [col][val] = false;
    val_en_reg [reg][val] = false;
}

int región (int fil, int col) {
    return (col/subN) * subN + fil/subN;
}

```

Febrero 2006-2ª (ejercicio 1)

Enunciado: En la práctica obligatoria del presente curso 2005/2006 se ha tenido que diseñar y desarrollar un algoritmo para resolver el juego del Su-doku. Codifique en Java o en Modula-2 un algoritmo iterativo que recorra el cuadrado de 3 x 3 casillas que corresponda a una casilla que ocupa las posiciones i, j del tablero.

Respuesta: Se trataba de encontrar la relación entre las posiciones i, j del tablero y las posiciones de comienzo del cuadrado de 3 x 3 que les corresponde en el tablero del Su-doku. Supongamos que el tablero tiene como índices 0..8, 0..8 y que i y j son de tipo entero, la relación se puede establecer de la siguiente manera:

```

int coordFilaInicioCuadrado = (i/3) * 3;           // División entera
int coordColumnaInicioCuadrado = (j/3) * 3;        // División entera

```

Ahora sólo queda hacer el recorrido:

```

for (int k = coordFilaInicioCuadrado; k < coordFilaInicioCuadrado + 3; k++) {
    for (int l = coordColumnaInicioCuadrado; l < coordColumnaInicioCuadrado + 3; l++) {
        procesar (tab [k][l]);
    }
}

```

De nuevo y como curiosidad vemos que las variables tienen un nombre enorme, yo personalmente lo recortaría y pondría algo más adecuado para el examen, ya que como se ha insistido casi no hay tiempo material y perder el tiempo supone casi suspender. Queda, por tanto, como ejercicio el cambiar el nombre.

Febrero 2007-1ª (ejercicio 1) (igual al ejercicio 3 de Septiembre 2007-reserva)

Enunciado: Implementa en Java o Modula-2 la función de estimación de la cota en el problema del n-puzzle (práctica de este curso).

Respuesta: En este caso el ejercicio no tiene solución oficial, por lo que dependerá en cada caso del alumno. En estos tipos de ejercicios aunque sea más complicado preferiría implementarlo en Java que en Modula-2, por la facilidad del lenguaje. En cuanto a la estimación de la cota podremos decir que es la suma de la cota realizada hasta el momento más la distancia de Manhattan, recordemos que sería esta la suma del número de posiciones que hay desde su disposición inicial al final. Para implementarlo en Java (preferiblemente) se debería realizar dos funciones, la de la distancia de Manhattan y la de la cota superior propiamente dicha.

Febrero 2007-2ª (ejercicio 2)

Enunciado: Escribe en Java el algoritmo principal del esquema de ramificación y poda utilizado en la práctica del presente curso (no es necesario implementar las funciones auxiliares).

Respuesta: La implementación más sencilla es dejar los nodos solución dentro de la cola de prioridad. Cuando el primero de la cola sea solución hemos encontrado la solución óptima, porque el resto de nodos proporcionan una estimación que aún en el mejor de los casos sería peor:

```
public class MejorPrimero implements SearchAlgorithm {
    public solution SearchSolution (Node nodoInicial) {
        Solution solución;
        Queue<Node> cola = new PriorityQueue<Node> ();
        cola.add (nodoInicial);
        while (!cola.isEmpty()) {
            Node nodo = cola.poll();
            If (nodo.isSolution () ) {
                return nodo.getSolution () ;
            } else {
                for (Node nodoHijo: nodo.generatePossibleChildren()) {
                    cola.offer (nodoHijo);
                }
            }
        }
        return null;
    }
}
```

Sin embargo, la solución anterior tiene el problema de que todos los nodos (incluidas algunas soluciones) se guardan en memoria (en la **cola de prioridad**) aún sabiendo que muchos no pueden proporcionar la solución óptima. Por tanto, es posible refinar el algoritmo anterior para ir eliminando de la cola de prioridad todos los nodos que podamos ir descartando. Para descartar nodos, primero tenemos que encontrar una solución que se convierta en la solución de referencia. Todos los nodos ya existentes con una estimación peor ya no se meten en la cola. Esta versión necesita menos memoria, pero puede tener mayor coste computacional, puesto que hay que recorrer la cola cada vez que encontramos una nueva solución. Por otro lado, mantener la cola tiene menos coste porque en general contendrá menor número de nodos.

```

public class RamificacionYPoda implements SearchAlgorithm {
    public solution SearchSolution (Node nodoInicial) {
        Queue<Node> cola = new PriorityQueue<Node> ();
        cola.add (nodoInicial);
        int costeMejor = Integer.MAX_VALUE;    // Solución mejor
        while (!cola.isEmpty()) {
            Node nodo = cola.poll();
            if (nodo.isSolution () ) {
                int costeReal = nodo.getCost ();
                if (costeReal < costeMejor) {
                    costeMejor = costeReal;
                    solucion = nodo.getSolution ();
                    depurar (cola, costeMejor);
                }
            } else {
                for (Node nodoHijo: nodo.generatePossibleChildren()) {
                    int cota = nodoHijo.calculateEstimatedCost();
                    if (cota < costeMejor) {
                        cola.offer (nodoHijo);
                    }
                }
            }
        }
        return solucion;
    }

    private void depurar (Queue<Node> cola, int costeMejor) {
        Iterator it = cola.iterator ();
        List<Node> depurationList = new ArrayList<Node> ();
        while (it.hasNext () ) {
            Node nodo = (Node) it.next ();
            if (nodo.calculateEstimatedCost() > costeMejor) {
                depurationList.add (nodo);
            }
        }
        cola.removeAll (depurationList);
    }
}

```

NOTA DEL AUTOR: De nuevo, vemos que los nombres de los métodos de java son grandísimos, por lo que los reduciría a 8 caracteres o 10 a lo sumo. Por otro lado, se observa que es un refinamiento. Como he comentado en ejercicios anterior, en todo caso el código lo haría de una manera más fácil y más general, aun así es interesante ver el uso de las clases, métodos e incluso interfaces (SearchAlgorithm) o eso creo en Java.

Septiembre 2007 (ejercicio 1)

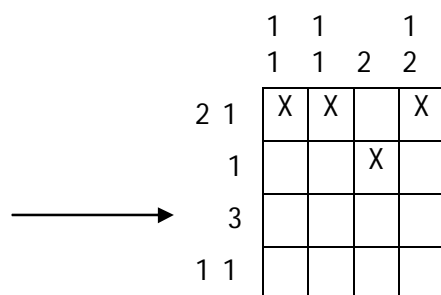
Enunciado: Describe el grafo asociado al espacio de solución del problema del n -puzzle (práctica de este curso).

Respuesta: En estos ejercicios al pedir el grafo asociado personalmente lo dibujaría y lo explicaría. Al ser personal este grafo no lo haríamos, en este caso, pero pondremos la solución aportada del ejercicio:

- Cada nodo se bifurca como máximo en los cuatro movimientos posibles del hueco (arriba, abajo, izquierda y derecha), siempre que sea posible.
- La profundidad del árbol está limitada por el número de distintos tableros posibles $((n * n)!)^2$, siendo n el tamaño del lado del tablero), ya que no se deben repetir.

Febrero 2008-1ª (ejercicio 1)

Enunciado: Escribe el grafo asociado al espacio de soluciones del problema del nonograma (práctica de este año), a partir del nodo que se presenta a continuación. Se supone que a partir de dicho nodo se van a comenzar a explorar la fila 3 (no puntúa la exploración por fuerza bruta de coste 2^{n^2}).



Respuesta: En TODOS los exámenes de este mismo curso va a ser la tónica general. Por ello, con hacer un solo ejercicio nos es suficiente para los demás.

Recordemos y así lo hice yo es que tenemos dos vuelta atrás, una que es la que vamos a hacer para hallar los descendientes de las filas según las restricciones dadas al nonograma y otra para verificar si en las columnas es correcto y cumple las restricciones de las columnas. Suponemos que inicialmente están vacías las casillas y que conforme se completan tienen valor 0 ó 1. Pondremos un pequeño ejemplo y se verá como se hace.

Para la fila 3, que es donde empezaremos tendremos estos descendientes:

Desc. 1	X	X	X	
Desc. 2		X	X	X

Empezaremos colocando el primer descendiente al grafo anterior:

		1	1		1
		1	1	2	2
2	1	X	X		X
	1			X	
	3	X	X	X	
1	1				

Verificamos que las columnas cumplan las condiciones, es decir, que al intentar probar con la restricción 1 1 (recordemos que tiene, al menos, un espacio en blanco) realmente pueda llegar a solución. En este caso, sería positiva la verificación por lo que probaríamos con la siguiente fila de igual manera:

Desc. 1	X		X	
Desc. 2	X			X
Desc. 3		X		X

Probaríamos ahora con el primer descendiente, aunque no acabaremos de resolver el nonograma lo dejaremos marcado para que se vea cuando hay que hacer vuelta atrás, ya que el procedimiento es similar.

		1	1		1
		1	1	2	2
2	1	X	X		X
	1			X	
	3	X	X	X	
1	1	X			X

Al verificar las restricciones de la primera columna ya vemos que no las cumple, por lo que se tendría que probar con el siguiente descendiente, hasta verlos todos. Si no se subiría de nivel y se cambiaría de descendiente consecutivamente. Gráficamente, es más un árbol donde se podría al llegar a este nivel con este descendiente. Se deja, por tanto, la finalización pendiente de hacer.