ALGORITMOS KENKEN

Jonathan Nebot
Toni Palacios
Moises Diaz
Ali Muhammad
Grupo 44

Algoritmo createSudoku()

Version 1

```
private boolean _createSudoku(
        int rfind,
        int value,
        int cont_values,
        int cont_pos,
        boolean bpos[],
        int imat[][],
        boolean rmat[][],
        boolean cmat[][],
        int x,
        int y
) {
        if(freeCoordenate(imat,x,y) && validValue(value, x, y, rmat, cmat)) {
                Utilities u = new Utilities();

                rmat[x][value-1] = cmat[y][value-1] = true; //setBoolValues
                imat[x][y] = value;                                     //setValue

                if(rfind==0) {
                        setResultValues(imat);
                        return true;
                }

                int original_imat[][] = new int[tamany][tamany];  //Saving the Sudoku state
                for(int i = 0; i < tamany; i++) for(int j = 0; j < tamany; j++)
                        original_imat[i][j] = imat[i][j];

                while(cont_values<tamany) {

                        while(cont_pos<rfind) {

                                int nx = u.newRandom(tamany);
                                //Searching the next validate position
                                int ny = u.newRandom(tamany);

                                int c_imat[][] = new int[tamany][tamany];  //Initial State's copy
                                for(int i = 0; i < tamany; i++) for(int j = 0; j < tamany; j++)
                                        c_imat[i][j] = original_imat[i][j];
```

```
                        if(_createSudoku(
                                (rfind-1),
                                value,
                                0,
                                0,
                                new boolean[tamany],
                                c_imat,
                                rmat,
                                cmat,
                                nx,
                                ny)) return true;

                        if(freeCoordenate(imat,nx,ny)) {
                                cont_pos++;
                                imat[nx][ny] = -1;
                        }
                }

                for(int i = 0; i < tamany; i++) for(int j = 0; j < tamany; j++)
                        imat[i][j] = original_imat[i][j];       //Reset to initial state

                bpos[value-1] = true;

                while(bpos[value-1] && !u.allTrue(bpos))
                        value = u.newRandom(tamany)+1;
                        //Searching not use value if there is anyone

                cont_pos = 0;
                cont_values++;

        }
    }
    return false;
}

private void createSudoku() {
    Utilities u = new Utilities();
    int rfind = (tamany*tamany)-1;
    int imat[][] = new int[tamany][tamany];
    boolean rmat[][] = new boolean[tamany][tamany];
    boolean cmat[][] = new boolean[tamany][tamany];
    boolean bpos[] = new boolean[tamany];
    int x = u.newRandom(tamany);
    int y = u.newRandom(tamany);
    int value = u.newRandom(tamany)+1;
    int cont_pos = 0;
    int cont_values = 0;
```

```
        while(!_createSudoku(rfind, value, cont_pos, cont_values, bpos, imat, rmat, cmat, x, y)) {
                rfind = (tamany*tamany)-1;
                imat = new int[tamany][tamany];
                rmat = new boolean[tamany][tamany];
                cmat = new boolean[tamany][tamany];
                bpos = new boolean[tamany];
                x = u.newRandom(tamany);
                y = u.newRandom(tamany);
                value = u.newRandom(tamany)+1;
                cont_pos = 0;
                cont_values = 0;
        }
}
```

Este algoritmo fue el primero en funcionar con los mínimos definidos.

Se inicializaba toda una serie de estructuras necesarias para el backtraking junto con una posición (**int x, int y**) mayor que (-1,-1) y menor que (**tamany**-1,**tamany**-1) siendo **tamany** un entero de la clase que indica lo que mide la matriz contenedora de los valores a introducir en el sudoku y un valor **int value** mayor que 0 y menor que **tamany**+1 totalmente random y a partir de aquí se iniciaba la recursividad.

imat[4][4] {

0	0	3	0
2	0	4	0
3	0	1	0
0	0	0	2

}

//Un estado cualquiera del backtraking con los valores representados.

Las estructuras son:
La matriz antes mencionada de enteros imat[tamany][tamany].
Dos matrices de boleanos rmat[tamany][tamany] y cmat[tamany][tamany] las cuales indican los valores que ya se estan utilizando en las filas y/o columnas de forma que en el caso de las filas introducimos en la primera posición la fila **x** y en la segunda el **value**-1 y con las columnas de igual forma introducimos la columna **y** en la primera posición y el **value**-1 en la segunda de forma que la consulta resulta eficiente.

rmat[4][4] {

false	false	true	false
false	true	false	true
true	false	true	false
false	true	false	false

}

cmat[4][4] {

false	true	true	false
false	false	false	false
true	false	true	true
false	true	false	false

}

//Un estado cualquiera del backtraking, a la izquierda las filas y a la derecha las columnas, con los valores que aparecen en la tabla **imat** vista anteriormente.

Tres enteros que realizan la función de contadores, comenzando por **int rfind** que indica el número de valores que aún quedan por encontrar y se inicializa con el total de casillas del tablero, es decir, (**tamany*tamany**)-1, ya que el primer valor viene dado inicialmente sabiendo que no habrá ningún valor en el tablero y siempre será válido.
Le sigue **int cont_pos** el cual indica el número de posiciones que se han probado en el tablero, es decir, que por cada llamada recursiva que no cumpliese la condición de fila y columna válida, y no sea una posición que contiene ya un valor o que ya haya sido probada anteriormente sin éxito (**cada prueba sin éxito se marcan con un -1 en la matriz imat)** se incrementa este contador el cual siempre tendrá que ser menor a **rfind** ya que son las posiciones que aún no tienen valor, es decir, que quedan por probar.
Y finalmente **int cont_value,** que va de la mano de **boolean bpos[tamany],** indica el número de valores restantes que quedan por probar para una posición **(x,y)** concreta y se calcula de forma que cada vez que se prueba un valor válido sin éxito se indica en el array **bpos** poniéndo a **true** ese valor**,** por lo que este contador indica el número de **false** dentro de **bpos.**

Definitivamente en cada llamada del backtraking lo que se hace es probar aleatoriamente y una a una todas las posiciones del tablero con sus respectivos valores y se marcan como descartadas o se continua dependiendo del resultado condicional:

if(freeCoordenate(imat,x,y) && validValue(value, x, y, rmat, cmat))

Por un lado la primera función **freeCoordenate** lo que hace es mirar si ya se ha introducido un valor en la posición **(x,y)** de **imat** o si esa posición ya a sido probada y se a descartado. Si se dan uno de los dos casos se devuelve **false,** sino **true.**
Por otro lado tenemos la función **validValue** y lo que hace es utilizando las matrices mencionadas anteriormente, **rmat** y **cmat,** junto con la posición **(x,y)** y el **value** a comprobar es mirar si para esa fila o columna ya se ha utilizado ese valor, en este caso devuelve **false,** sino **true.**

En caso de cumplirse ambas condiciones se introduce el **value** en la posición correspondiente de **imat,** y en la casilla de la estructura de la clase **MatrixKenken** con la función **setResultValues,** y se actualizan **rmat** y **cmat.**
Por tal de garantizar el estado del Sudoku y que **no se vea afectado el resultado se copia imat** de forma local, se reserva para las siguientes llamadas recursivas, y se trabaja con las originales a la hora de marcar las posiciones descartadas para cada valor, de forma que probamos cada valor en cada posición del tablero hasta que ambos sean váidos y se repite el proceso.

El backtraking finaliza cuando **rfind** llega a 0, el cual se decrementa por cada llamada, devolviendo **true** hacia la anterior llamada recursiva y esta hacia su anterior y así hasta finalizar, teniendo el resultado guardado en **MatrixKenken.**

(Tabla)

Este algoritmo presentaba una eficiencia lamentable como se muestra en la tabla.

Version 2

```
private boolean _createSudoku(int rfind, int imat[][], boolean rmat[][], boolean cmat[][]) {

	if(rfind==0) {
		setResultValues(imat);
		return true;
	}

	Utilities u = new Utilities();

	int indexC, indexV, value;
	int c_imat[][];

	boolean c_rmat[][], c_cmat[][];

	Pair coordenates;

	ArrayList<Pair> candidates = getCandidates(imat, rmat, cmat);
	//Filtering the candidates to the next position
	ArrayList<ArrayList<Integer>> values = getValues(candidates, rmat, cmat);

	while(candidates.size()>0) {

		indexC = u.newRandom(candidates.size());
		coordenates = candidates.get(indexC);

		while(values.get(indexC).size()>0) {

			indexV = u.newRandom(values.get(indexC).size());
			value = values.get(indexC).get(indexV);

			c_imat = new int[tamany][tamany];  //Initial State's copy
			c_rmat = new boolean[tamany][tamany];
			c_cmat = new boolean[tamany][tamany];
			for(int i = 0; i < tamany; i++) for(int j = 0; j < tamany; j++) {
				c_imat[i][j] = imat[i][j];

				c_rmat[i][j] = rmat[i][j];
				c_cmat[i][j] = cmat[i][j];
			}

			c_rmat[coordenates.getFila()][value-1] =
				c_cmat[coordenates.getColumna()][value-1] = true;
				//setBoolValues
			c_imat[coordenates.getFila()][coordenates.getColumna()] = value;
				//setValue

			if(_createSudoku((rfind-1), c_imat, c_rmat, c_cmat)) return true;

				values.get(indexC).remove(indexV);
		}
```

```
                candidates.remove(indexC);

        }

        return false;
}

private void createSudoku(int iMat[][]) {
        int imat[][] = iMat;
        int rfind = calculateRfind(imat);
        boolean rmat[][] = calculateRows(imat);
        boolean cmat[][] = calculateColumns(imat);

        if(!allValidValues(imat) || !_createSudoku(rfind, imat, rmat, cmat)) this.tamany = -1;
}
```

Algoritmos de filtrado de candidatos

```
getCandidates(imat, rmat, cmat); //Filtering the candidates to the next position
valuesByZone = getValues(candidatesByZone, rmat, cmat);
filterByZoneCandidates(candidates, values, imat, izone, iop, ires);
filterByZoneValues(candidates, values, imat, izone, iop, ires);
filterByResultCandidates(candidatesByZone, valuesByZone);
filterByResultValues(candidatesByZone, valuesByZone);
```

Algoritmo defineZone()

```
public void defineZona(CasillaKenken ck[][], Pair p, ArrayList<Zona> zonas) {
        Utilities u = new Utilities();
        int quantity = 0;
        boolean first = true;

        if(this.operacion == '+' || this.operacion == 'x') quantity = u.newRandom(ck.length)+1;
                if(quantity<2) quantity = 2;

                Stack<Pair> s = new Stack<Pair>();
                s.push(p);

                this.resultado = ck[p.getFila()][p.getColumna()].getResultado();

                while(!s.empty() && quantity>0) {

                Pair ap = s.pop();

                if(!first) applyOperacion(ck[ap.getFila()][ap.getColumna()].getResultado());

                ck[ap.getFila()][ap.getColumna()].setIndexZona(zonas.size());

                for(int i = 0; i < directionsX.length; i++) {
                        int nx = ap.getFila()+directionsX[i];
                        int ny = ap.getColumna()+directionsY[i];
```

```
                        if(insideTaulell(nx,ny,ck.length) && ck[nx][ny].getIndexZona()==-1) {
                                Pair np = new Pair(nx,ny);
                                s.push(np);
                        }
                }

                first = false;
                quantity--;
        }

        //Overflow exception
        if(this.resultado<0) {
                this.operacion = '+';
                this.resultado = 0;

                setZonaToSuma(ck, zonas.size());
        }
}
```