Algorítmica

BACKTRACKING

Entrega 4:

Resolución de un Sudoku mediante la técnica de backTracking

Javier Martín Gómez
Alfonso Soto López
Julián Torices Hernández
Antonio David Mota Martínez
Alberto Peinado Santana
Enrique Moreno Carmona

24 de mayo de 2017

Índice

Lista de Algoritmos

Resumen

En esta memoria se expone la resolución del pasatiempo/rompecabezas llamado **Sudoku**.

Se realizara mediante el uso de la técnica **BackTracking** (Vuelta atrás) analizando la **eficiencia** de esta implementación y las decisiones tomadas.1

1. Análisis del problema

2. Justificación de la técnica implementada

3. Diseño de la solución

La solución consiste en **recorrer el sudoku generado** y rellenar los huecos vacíos (marcados con un 0) con un valor comprobando posteriormente que utilizando ese valor se obtiene una solución válida. Utilizamos **llamadas recursivas** para recorrer el árbol.

Hacemos un **recorrido en profundidad**. Hasta encontrar un valor que no cuadre, en tal caso se vuelve al **nodo** anterior y se comprueba con otro número. Así se llega a una solución correcta.

En nuestro caso la **lista de nodos vivos** es nuestra matriz de sudoku

4. Implementación

Algoritmo 1: Pseudo-Codigo de un algoritmo backtracking para la resolución del juego Sudoku. Se presenta el núcleo de la función ignorando funciones obvias

```
1
2       ResolverSudoku(sudoku &s, int x, int y ){
3
4           while[x][y] != 0
5               if( ! incrementar(x,y) )
6                   // Si no conseguimos incrementar las coordenadas, significa que hemos
                        llegado a la ultima posicion del sudoku.
7                   return true;
8
9               else
10                  // Continuamos
11
12
13          for( i desde 1 hasta 9 ){
14
15              s[x][y] = i
16
17              if( checkRow() && checCol() && checkGroup)
18                  if(ResolverSudoku(s,x,y))
19                      return true;
20
21              s[x][y] = 0;
22
23          }
24
25      }
```

Algoritmo 2: Representación de la función checkRow

```
1  for(int i=0; i < 9; i++)
2   if (i != x)
3    if (s[x][y] == s[i][y])
4     return false;
5  return true;
```

Algoritmo 3: Representación de la función checkCol

```
1  for(int i=0; i < 9; i++)
2   if (i != y)
3    if (s[x][y] == s[x][i])
4     return false;
5  return true;
```

Algoritmo 4: Representación de la función checkGroup

```
1  int groupX = x/3;
2  int groupY = y/3;
3  int iniX = groupX*3;
4  int iniY = groupY*3;
5
6  for(int i = iniX; i < iniX+3; i++)
7   for(int j = iniY; j < iniY+3; j++)
8    if(i != x && j != y)
9     if(s[i][j] == s[x][y])
10     return false;
11 return true;
```

5. Explicación del funcionamiento del algoritmo, sobre un caso de ejemplo

Para la explicación vamos a usar de ejemplo, como no puede ser de otra manera, el juego del sudoku[2].

				4				
		2	6		7	1		
8	7	1				6	9	4
	6						4	
2		5	9		6	7		8
	8						2	
6	5	8				4	7	1
		9	4		8	5		
				7				

Figura 1: Ejemplo de un sudoku

La idea principal es desarrollar las **restricciones** en las que se basa este juego matemático, teniendo en cuenta que solo disponemos de los elementos del 1 al 9.

Columnas: Comprobar que en cada una de las columnas no aparece ningún elemento repetido mas de una vez.

Filas: Comprobar que en cada una de las fila no aparece ningún elemento repetido mas de una vez.

Sector: Comprobar que en cada sector (Sub-sector de 3x3 en los que se divide la cuadricula completa de 9x9 [2]) no aparece ningún elemento repetido mas de una vez.

Estas restricciones se representan en el pseudo-código [c1] linea 17 como 'checkRow'2, 'checkCol'3, 'checkGroup'4 respectivamente.

Para aplicar el concepto del **backTracking** sobre este problema, iteraremos sobre cada una de las casillas, colocando un valor/elemento de los disponibles y comprobando si es esta asignación satisface las restricciones ya planteadas.

Bastará con avanzar de casilla cada vez que encontremos un valor que cumpla con las condiciones o se trate de un valor diferente de 0 (nos indica un valor inicial) y retroceder de casilla cuando no se cumplan y hayamos comprobado todos valores en esa casilla.

Al combinar todo esto sobre una función recursiva obtenemos un algoritmo **backtracking**.

Hay que tener destacar que se ha implementado un algoritmo que **no requiere de una mascara** de ningún tipo sobre los elementos iniciales.

6. Uso en situaciones reales

7. Calculo teórico de la eficiencia

I. El método leerSudoku tiene 3 bucles for homogéneos, de los cuales 2 son anidados por lo que la eficiencia, aplicando la regla de la suma es el $max(n^2, n)$ que da $O(n^2)$

II. El método printSudoku tiene 2 bucles for homogéneos anidados, la eficiencia es $O(n^2)$

III. El método incrXY son solo condicionales cuya eficiencia es $O(1)$

IV. El método checkRow tiene un bucle for homogéneo cuya eficiencia es $O(n)$

V. El método checkCol tiene un bucle for homogéneo cuya eficiencia es $O(n)$

VI. El método chechGroup tiene 2 bucles for homogéneos anidados, la eficiencia es $O(n^2)$

VII. El método resolverSudoku tiene un bucle While y for homogéneos pero no están anidados por lo que la eficiencia es $O(n)$

VIII. El Main solo tiene salidas por pantalla y llamadas a métodos cuya eficiencia es $O(1)$

IX. La eficiencia del algoritmo, aplicando la regla de la suma, es el $max(n^2, n)$ por lo que da como resultado que la eficiencia del algoritmo es $O(n^2)$

8. Instrucciones de compilación y ejecución

Se incluye con el código un archivo 'data.txt' que contiene 10.000 sudokus categorizados como *muy dificiles* extraídos de la web: Printable 9x9 Sudoku Puzzles.

Para seleccionar cualquiera de los sudokus base proporcionados basta con pasar como argumento un valor $0 < valor < 10001$.

```
soto@soto-PC $ make
        g++ -std=c++11 sudoku.cpp -o sudoku

soto@soto-PC $ ./sudoku
```

Algoritmo 5: Ejemplo de compilación y ejecución del algoritmo

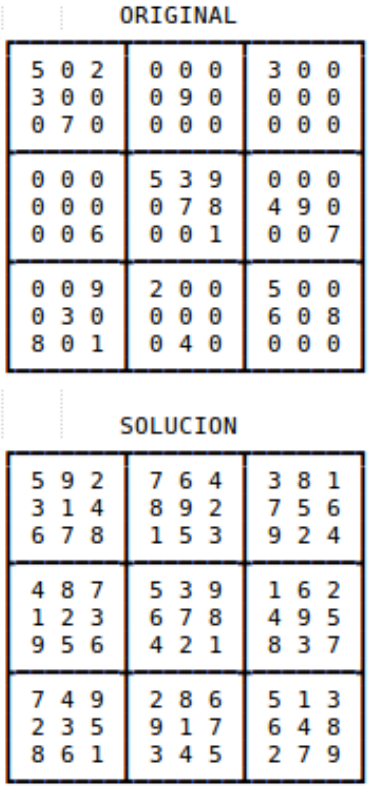

Figura 2: Salida de la ejecución del algoritmo