Algorítmica (2016-2017)

Grado en Ingeniería Informática

Universidad de Granada

Práctica 4

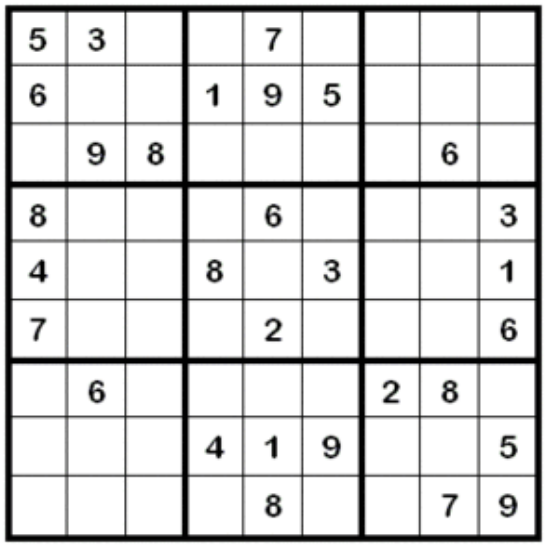
Algoritmos Backtracking y Branch&Bound: Problema de resolución de un sudoku

Gregorio Carvajal Expósito
Gema Correa Fernández
Jonathan Fernández Mertanen
Eila Gómez Hidalgo
Elías Méndez García
Alex Enrique Tipán Párraga

Contenidos

Análisis del Problema

Un **sudoku** consiste en rellenar con números del 1 al 9 una tabla de 9x9, dividida en subcuadrículas de 3x3, con casillas ya rellenadas anteriormente.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Debemos rellenar las casillas que estén vacías ateniéndonos a una serie de normas:

- ❑ No puede haber dos casillas en la misma fila con el mismo número.
- ❑ No puede haber dos casillas en la misma columna con el mismo número.
- ❑ No puede haber dos casillas en la misma subcuadrícula 3x3 con el mismo número.

Para solucionar un sudoku, es necesario escribir números en todas las casillas que quedan vacías de modo que cumplan las 3 reglas anteriores. Debemos tener en cuenta, que sólo pueden modificarse las casillas que estén vacías. Como solución final, daremos una tabla totalmente rellena de números como solución, cumpliendo con las anteriores reglas.

Elección de Técnica y Justificación

Para elegir la técnica a usar hemos descartado Branch&Bound, ya que con este método necesitamos conocer la solución de antemano para saber que ramas podar en función de las que sean más óptimas, lo cual en nuestro caso carece de sentido ya que el objetivo de nuestro problema en hallar la solución al sudoku. Por lo dicho anteriormente, hemos decidido utilizar **Backtracking** para la resolución de nuestro problema.

Diseño de la Solución

Tipo de árbol

Dado que en nuestro árbol se generan todas las combinaciones posibles y que no está balanceado y los nodos tienen más de 2 hijos, podemos decir que es un árbol combinatorio o permutacional.

Restricciones Explícitas

- ❑ Las componentes de la matriz solución contienen valores del 1 al 9.

Restricciones Implícitas

- ❑ No pueden existir dos números iguales en la misma fila.
- ❑ No pueden existir dos números iguales en la misma columna.
- ❑ No pueden existir dos números iguales en la misma subcuadrícula.

Solución Parcial

Las componentes preasignadas por el problema, serán números negativos del 1 al 9 y las componentes sin asignar tendrán un valor de 0.

Función de Poda

La función de poda consiste en, para una casilla libre, obtener los posibles números a introducir en ella. Cuando no obtenemos ninguna opción significa que no hay ningún camino más que seguir por esa rama.

Fichero de entrada

Simplemente es un fichero con 9 líneas que contiene 9 dígitos separados por espacios. Donde se desee insertar una casilla vacía, se colocará un 0. Ejemplo:

```
0 8 0 5 7 6 2 0 0
0 0 0 4 0 2 0 0 0
0 0 0 0 3 9 5 4 8
6 3 0 9 0 0 8 5 2
0 9 0 2 0 0 3 7 0
8 0 0 0 5 0 6 9 4
2 5 7 6 0 3 4 8 9
3 0 8 7 0 0 0 2 5
0 4 0 0 0 0 0 0 6
```

Esqueleto del Algoritmo

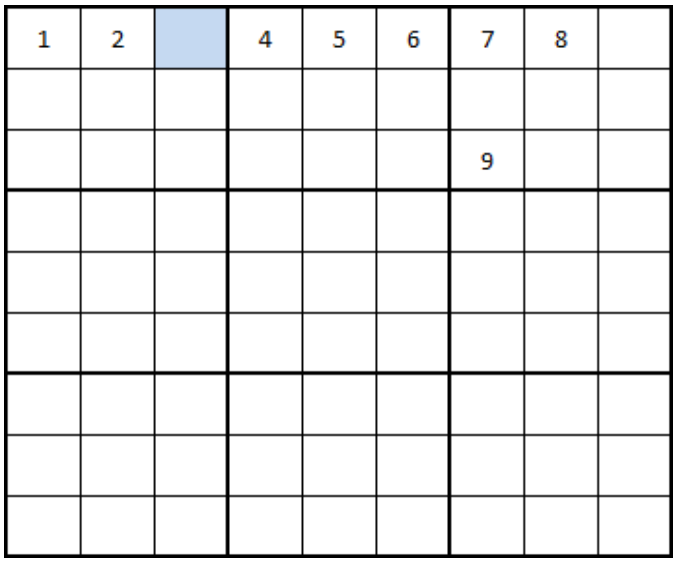
```
b:bool resolverSudoku (S:Sudoku)

      siguiente_casilla = getSiguienteCasilla()

      Si sudoku lleno
            Devolver true

      numerosDisponibles = getNumerosDisponibles()
      disponible = false

      Para cada numeroDisponible
            disponible = numerosDisponibles

      Si disponible
            setCasilla(numero)
            Si resolverSudoku(Sudoku)
                  Devolver true

            setCasilla(vacia)

      Devolver false
```

Funcionamiento del Algoritmo

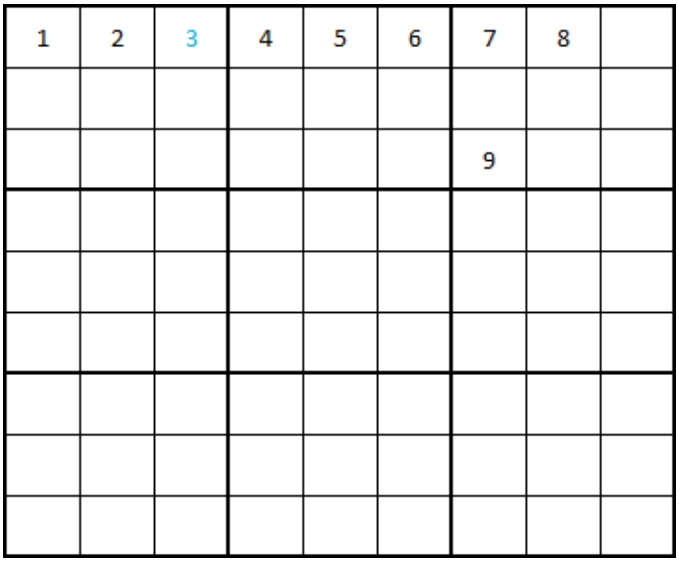
Partimos del siguiente sudoku como ejemplo:

1	2		4	5	6	7	8	
						9		

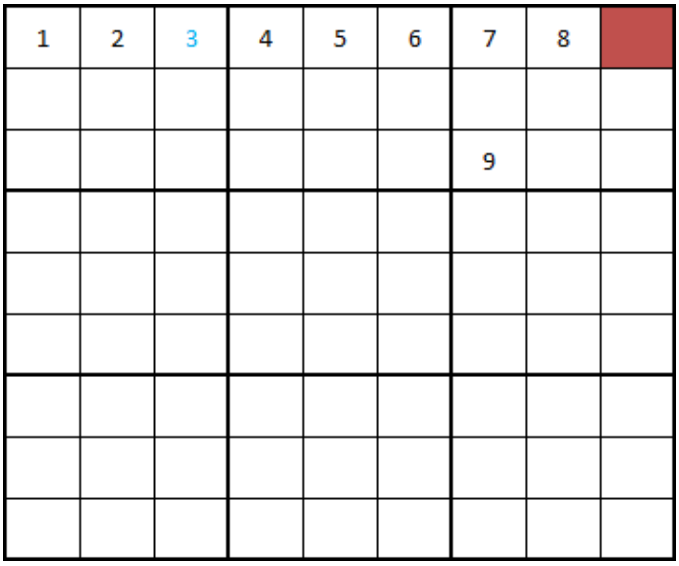
Nuestro algoritmo comienza comprobando las casillas por filas hacia la derecha. Entonces la casilla a rellenar sería la marcada en azul:



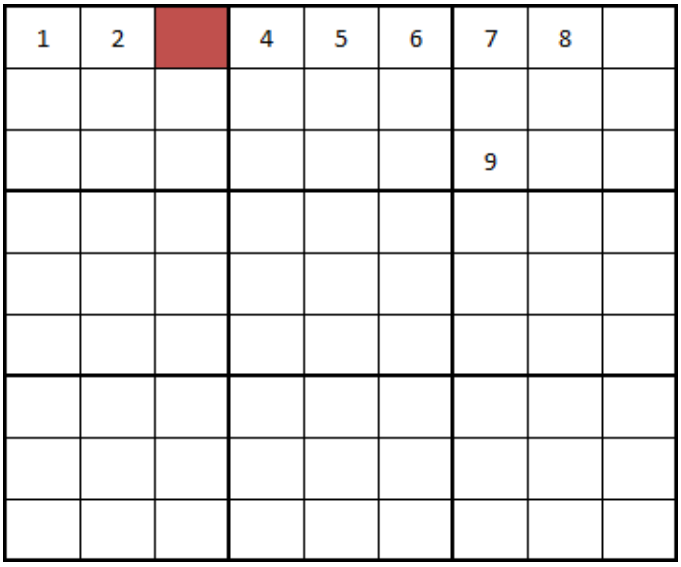
Según las reglas del sudoku, en dicha casilla solo se podría poner un 3 o un 9, estos serían los dos nodos generados. Dado que exploramos en profundidad, primero probaremos con el 3.



En la siguiente casilla a comprobar, no es posible colocar ningún número, puesto que todos ya han sido usados en la fila, columna o subcuadrícula de esa casilla:



Dada esta situación, no se generarán nodos hijos y por tanto, estamos ante un nodo hoja. Como todavía hay casillas vacías en el sudoku, el nodo hoja en cuestión no es un nodo solución, con lo cual hacemos Backtraking y volvemos a la casilla anterior:



Ahora, probamos con el otro nodo, que era el 9. De este modo, en la siguiente casilla solo se generaría un nodo, que es el 3:

1	2	9	4	5	6	7	8	3
						9		

Y así sucesivamente hasta conseguir rellenar completamente el sudoku. Cuando una casilla libre no tiene ninguna posibilidad, volvemos a la casilla anterior y probamos con el siguiente nodo.

Caso de un Problema Real (aplicando Backtracking)

La mayoría de sus usos del backtracking es la resolución de puzzles, como el solitario, crucigramas o el mismo ejemplo que se ha usado para este ejercicio, el sudoku.

El backtracking, también se usa en los editores de texto o entornos de programación para realizar análisis sintáctico o de sintaxis. Gracias a este sistema tenemos compiladores o intérpretes para los lenguajes de programación más usados. Por ejemplo, se puede realizar una técnica de backtracking en un navegador web para interpretar el contenido de un html y mostrarlo al usuario.

Cálculo de la Eficiencia Teórica

Tomaremos n como el número total de casillas del sudoku.

```
bool resolverSudoku(Sudoku &sudoku) {
  // Obtener la siguiente casilla libre O(n)
  Sudoku::Casilla siguiente_casilla = sudoku.getSiguienteCasillaVacia();

  if (siguiente_casilla == Sudoku::NO_LIBRES) { return true; }

  // Obtener los movimientos disponibles para la siguente casilla. O(√n)
  Sudoku::NumerosDisponibles numeros = sudoku.getNumerosDisponibles();

  // Para cada número disponible probamos a introducirlo ver si nos
  // lleva a la solución
  for (int i = 1; i < numeros.size(); ++i) {
    disponible = numeros[i];

    if (disponible) {
      // Introducir el número en la casilla libre. O(1)
      if (resolverSudoku(sudoku)) {
        // Devolver que se ha completado O(1)
      }
      // Vaciar la casilla para probar otra O(1)
    }
  }
  return false;
}
```

Vamos a obtener la eficiencia del anterior código. Para ello, planteamos la siguiente ecuación recurrente:

$$\begin{aligned} T(0) &= 0 \\ T(n) &= 9T(n-1)+n \end{aligned}$$

$$t_n - 9t_{n-1} = n$$

$$P.\ Característico:\ (x-9)(x-1)^2$$

$$t_n = A\cdot 9^n + B\cdot 1^n + C\cdot n\cdot 1^n$$

$$t_n = A\cdot 9^n + B + C\cdot n$$

$$n=0 \longrightarrow A+B=0$$

$$n=1 \longrightarrow 9A+B+C=1$$

$$n=2 \longrightarrow 81A+B+2C=11$$

Resolviendo el sistema de tres ecuaciones, llegamos a la conclusión de que el orden de eficiencia es:

$$O(9^n)$$

Instrucciones para Compilar y Ejecutar

Se ha creado un fichero Makefile para que sea más cómodo y sencillo compilar y ejecutar el algoritmo. Ejecutamos la orden `make` y se creará el ejecutable `sudoku_main`.

Después simplemente hay que ejecutar el programa pasándole un fichero como parámetro:
`./sudoku_main datosudoku.dat`