

# Universidade Federal de Santa Catarina

Centro Tecnológico  
Departamento de Informação e Estatística  
Ciências da Computação

Enzo da Rosa Brum

Relatório de paradigmas de programação

Florianópolis  
2023

# 1 Problema

Neste trabalho, foi decidido criar um resolvidor de sudoku comparativo (vergleichssudoku). As regras do jogo são:

1. Células na mesma linha não podem ter o mesmo valor
2. Células na mesma coluna não podem ter o mesmo valor
3. Células na mesma região não podem ter o mesmo valor
4. Os sinais »”entre duas células indicam que o número na célula para a qual a seta aponta é menor do que o número na outra célula.

## 2 Solução

### 2.1 Linguagem utilizada

Neste trabalho foi utilizada a versão 3 da linguagem Scala.

### 2.2 Algoritmo

#### 2.2.1 Estruturas utilizadas

Para resolver o sudoku comparativo, foi utilizado o backtracking junto com 4 estruturas de dados:

- Um array de Sets onde cada Set contém os valores escolhidos para uma determinada linha até o momento.
- Um array de Sets onde cada Set contém os valores escolhidos para uma determinada coluna até o momento.
- Um array de Sets onde cada Set contém os valores escolhidos para uma determinada região até o momento.
- Um array de Maps. Cada elemento do array é um Map representando os possíveis valores para cada número de uma determinada região. Nesse sentido, o Map mapeia o índice de um número para um Set contendo seus possíveis valores

```
1 val lines = Array.fill(N){ HashSet.empty[Int] } // Cada elemento i corresponde à um set contendo os valores na linha i
2 val cols = Array.fill(N){ HashSet.empty[Int] } // Cada elemento i corresponde à um set contendo os valores na coluna i
3 val regions = Array.fill(N){ HashSet.empty[Int] } // Cada elemento i corresponde à um set contendo os valores na região i
4
5 // mapeia o número de uma região para um set contendo os índices presentes nessa região
6 val indexes_per_region = HashMap[Int, HashSet[Int]](
7   (0 to N-1).map (key => key -> HashSet.empty[Int]): _*
8 )
9
10
11 for (i <- 0 until N) {
12   for (j <- 0 until N) {
13     val region_index = getRegionNum(i,j)
14     indexes_per_region(region_index).add(i*N + j)
15   }
16 }
17
18 val create_map = (sp: immutable.HashMap[Int, immutable.HashSet[Int]], v: Int) => mp + (v -> immutable.HashSet.range(0, N))
19
20 // Cria um array cujo cada elemento é um Map representando uma região.
21 // O map de uma região R mapeia os índices das células dentro da região R à Sets contendo os valores possíveis de cada elemento em R.
22 val possible_values = Array.tabulate(N) { region_index =>
23   indexes_per_region(region_index).foldLeft(immutable.HashMap.empty[Int, immutable.HashSet[Int]])(create_map)
24 }
25
```

Figura 1: Estruturas de dados

### 2.2.2 Algoritmo

A cada iteração do backtrack, é criado um conjunto contendo os possíveis valores para uma determinada célula. Tal conjunto corresponde à diferença entre o conjunto contendo os possíveis da célula atual (calculado com base nas restrições de maior/menor nas iterações anteriores) com os conjuntos que representam linha, coluna e região atual. Depois, para cada possível valor, é realizada a atualização das estruturas de dados definidas na seção anterior e a chamada recursiva para continuar o backtrack.

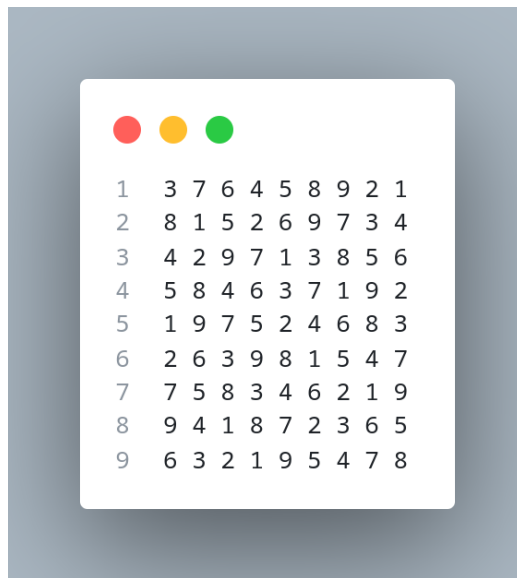
```
1 // A matriz é percorrida da esquerda para a direita.
2 // Se i for N-1, a solução foi encontrada
3 if (param_i == N-1 && param_j == N)
4   return true
5
6 val (i,j) =
7   if (param_i == N) (param_i-1,0)
8   else (param_i,param_j)
9
10
11 val index = i*N + j
12 val region_index = getRegionNum(i,j)
13 val next_right_index = index + 1
14 val next_down_index = index + N
15
16 // Possíveis valores para a célula atual
17 val cell_possible_values = region_possible_values(region_index)(index)
18   .diff(lines(i))
19   .diff(cols(j))
20   .diff(regions(region_index))
21
22
23 val old_region_map = region_possible_values(region_index)
24 var solution_found = false
25 boundary:
26   for (value <- cell_possible_values) {
27     val new_region_map = updateRegionMap(
28       updateRegionMap(old_region_map, hcmp, value, index, next_right_index),
29       vcmp, value, index, next_down_index
30     )
31
32     region_possible_values(region_index) = new_region_map
33
34     lines(i) += value
35     cols(j) += value
36     regions(region_index) += value
37
38     matrix(index) = value-1
39
40     val no_possible_value_right = (j != N-1 && getRegionNum(i,j+1) == region_index && region_possible_values(region_index)(i*N + j + 1).isEmpty)
41     val no_possible_value_down = (i != N-1 && getRegionNum(i+1,j) == region_index && region_possible_values(region_index)((i+1)*N + j).isEmpty)
42     if (!no_possible_value_right && !no_possible_value_down && backtrack(hcmp, vcmp, matrix, region_possible_values, lines, cols, regions, i, j+1)) {
43       solution_found = true
44       break(true)
45     }
46
47     regions(region_index) -= value
48     cols(j) -= value
49     lines(i) -= value
50
51     region_possible_values(region_index) = old_region_map
52   }
53
54
55 solution_found
```

Figura 2: Algoritmo de backtracking

## 3 Rodando o programa

Para rodar o programa, use o comando `make all ARGS=;arquivo-de-input.t`. É necessário que um compilador de Scala3 esteja instalado.

Quanto a saída do programa, o resultado do resolvidor é impresso no terminal em formato semelhante ao arquivo de input e, logo após isso, uma linha é impressa



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

Figura 3: Arquivo de input.

indicando se a solução encontrada pelo programa é a mesma que está presente no arquivo de input.

## 4 Mudanças em relação ao trabalho 1

Escolheu-se implementar um algoritmo muito mais simples que "Dancing Links".

## 5 Dificuldades encontradas

Eu diria que não houveram dificuldades nesse trabalho. Como usei um algoritmo mais simples e a linguagem Scala é bem parecida com Java (i.e: parecida com uma linguagem imperativa), não tive dificuldades na implementação.