

**Universidade Federal de Santa Catarina (UFSC)**

Campus Reitor João David Ferreira Lima

Departamento de Informática e Estatística

Bacharelado em Ciência da Computação

**Disciplina INE5416 - Paradigmas de Programação - 2023.2**

Prof. Maicon Rafael Zatelli

26/09/2023

**Aluno:**

André Amaral Rocco

# Relatório - *Kojun Solver*

## Análise de problema

O jogo Kojun é um jogo de lógica onde, a partir de um tabuleiro dividido em regiões de tamanho  $N$ , deve-se inserir um número em cada célula da grade para que cada região contenha cada número de 1 a  $N$  exatamente uma vez. Além disso, os números em células ortogonalmente adjacentes devem ser diferentes e, se duas células são adjacentes verticalmente na mesma região, o número na célula superior deve ser maior que o número na célula inferior.

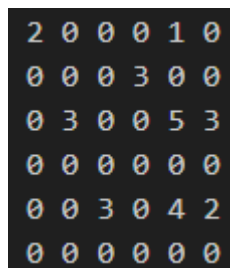
O problema proposto é a criação de um programa que, dado um tabuleiro de Kojun, encontre uma solução para o mesmo. Existem diferentes abordagens para a resolução desse problema, algumas mais performáticas e outras mais simples de implementar. Neste trabalho, foi escolhida uma abordagem que, apesar de não ser a mais performática, é relativamente simples de ser entendida e implementada: a técnica de backtracking.

## Análise de problema

### Modelagem do tabuleiro

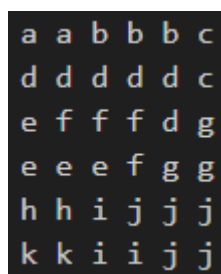
Um tabuleiro Kojun consiste em uma grade (onde podem ser colocados valores) dividida em diferentes regiões. Para representar computacionalmente um tabuleiro, foram criadas duas matrizes:

1. Uma matriz de valores (inteiros), onde cada célula representa uma posição do tabuleiro e o valor da célula representa o valor que está naquela posição. Em seu estado inicial, o tabuleiro pode conter valores ou não. Caso não contenha valores (células que devem ser preenchidas pelo jogador para completar o jogo), o valor da célula é 0.



2	0	0	0	1	0
0	0	0	3	0	0
0	3	0	0	5	3
0	0	0	0	0	0
0	0	3	0	4	2
0	0	0	0	0	0

2. Uma matriz regiões (representadas por caracteres), onde cada célula representa uma posição do tabuleiro e o valor da célula representa a região daquela posição. Note que células da mesma região precisam sempre estar conectadas, ou seja, não podem existir células da mesma região que não sejam ortogonalmente adjacentes.



a	a	b	b	b	c
d	d	d	d	d	c
e	f	f	f	d	g
e	e	e	f	g	g
h	h	i	j	j	j
k	k	i	i	j	j

Para serem utilizados pelo programa, os tabuleiros foram escritos em arquivos .txt onde a primeira linha do arquivo indica o tamanho  $N$  do tabuleiro ( $N$  linhas por  $N$  colunas). As próximas  $N$  linhas se referem à matriz de valores e as  $N$  linhas após a matriz de valores se referem à matriz de regiões. Não há restrições quanto ao tamanho do tabuleiro: o algoritmo é capaz de resolver tabuleiros Kujon de qualquer tamanho. Entretanto, como na implementação realizada cada região deve ser representada com apenas um caractere, gera-se então uma limitação quanto à quantidade de regiões.

Para o suporte à estrutura de matriz no Haskell, foi optado pela criação de um módulo *Matrix* que inicializa a estrutura e implementa algumas das operações básicas que seriam necessárias para a resolução do problema.

### Funcionamento do programa principal

O programa principal inicia lendo e extraindo as matrizes do arquivos .txt especificado em *contents*. A partir das linhas do arquivo de texto, inicializa as estruturas do tipo *Matrix* que correspondem à matriz de valores e à matriz de regiões. Em seguida, é chamada a função *solveKojun* que aplica o algoritmo de backtracking desenvolvido sobre o tabuleiro (representado pelas duas matrizes).

```
1 import Utils.Matrix
2 import KojunSolver
3
4 import Data.Char (isAlpha)
5
6 -- Função para filtrar caracteres alfabéticos
7 filterAlphabetic :: String -> String
8 filterAlphabetic = filter isAlpha
9
10 main :: IO ()
11 main = do
12     -- Lê arquivos que contém
13     -- O tamanho N da matriz (N x N) na primeira linha
14     -- A matriz de valores (ocupando N linhas)
15     -- A matriz de regiões (ocupando N linhas)
16     contents <- readFile "./inputs/8x8/kojun_4.txt"
17
18     -- A partir do arquivo, extrai o tamanho da matriz e as matrizes de valores e regiões
19     let linesRead = lines contents
20     let matrixSize = read (head linesRead) :: Int
21     let valueGrid = listToMatrix . map (map read) . take matrixSize . map words . drop 1 $ linesRead
22     let regionGrid = listToMatrix . map (map head) . take matrixSize . map words . drop 1 . drop matrixSize $ linesRead
23
24     putStrLn "\n> Matriz de valores:"
25     printMatrixFormatted valueGrid
26
27     putStrLn "\n> Matriz de regiões:"
28     printMatrixFormatted regionGrid
29
30     let maxRegionSize = matrixSize
31     -- Valor chute para o tamanho da maior região
32     -- Se for menor que o tamanho da matriz, a solução não será encontrada
33     -- Se for muito maior que o tamanho da matriz, a solução será encontrada, mas o desempenho será degradado
34
35     -- Resolve o Kojun
36     let solution = solveKojun valueGrid regionGrid (0, 0) maxRegionSize
37
38     -- Imprime a solução (ou uma mensagem indicando que não há solução)
39     case solution of
40     Just solutionGrid -> do
41         putStrLn "\n> Solução encontrada:"
42         printMatrixFormatted solutionGrid
43     Nothing -> print "Não há solução possível."
```

## Algoritmo de backtracking

O algoritmo de backtracking é uma técnica de busca recursiva que tenta encontrar uma solução para um problema através de tentativa e erro, retrocedendo (backtrack) quando uma tentativa falha. Começando de uma configuração inicial, o algoritmo faz uma escolha dentre várias possíveis e avança na solução. Se chegar a um ponto onde não é possível continuar sem violar uma condição ou restrição, ele retrocede para a escolha anterior e tenta outra opção. Esse processo recursivo continua até que uma solução seja encontrada ou todas as possibilidades tenham sido exploradas.

O módulo *KojunSolver* contém as funções responsáveis por aplicar o algoritmo de backtracking. Seu ponto de entrada é a função recursiva *solveKojun*. Ela testa as condições de parada ou continuidade do algoritmo de backtracking e, no caso de continuidade, faz uma chamada recursiva para si mesma com a próxima posição do tabuleiro a ser resolvida (possivelmente).

```
1 {-
2  Resolve o Kojun a partir de uma matriz de valores (Int) e uma matriz de regiões (Char).
3  Por ser uma função recursiva, recebe também um ponto de partida (linha e coluna) para aplicar o backtracking. A função
4  normalmente é chamada externamente com (0, 0).
5  O retorno é um monad Maybe, que pode ser Nothing (caso não haja solução) ou Just Matrix Int (caso haja solução) onde a
6  matriz de inteiros representa a solução que foi encontrada para o Kojun.
7 -}
8 solveKojun :: Matrix Int -> Matrix Char -> Position -> Int -> Maybe (Matrix Int)
9 solveKojun valueGrid regionGrid (row, col) maxRegionSize
10 | row == numRows valueGrid = Just valueGrid -- Caso chegou ao final da matriz
11 | col == numCols valueGrid = solveKojun valueGrid regionGrid (row + 1, 0) maxRegionSize -- Caso chegou ao final da linha, pula para a próxima
12 | getMatrixValue valueGrid (row, col) /= 0 = solveKojun valueGrid regionGrid (row, col + 1) maxRegionSize -- Caso a posição já esteja ocupada, pula para a próxima
13 | otherwise = tryValues valueGrid regionGrid row col [1..maxRegionSize] -- Caso nenhuma das condições acima seja satisfeita, chama tryValues com os valores de 1 a N,
14 where
15 {-
16  Tenta colocar valores válidos na posição atual.
17  Input: Recebe a matriz de valores, a matriz de regiões, a linha e coluna atual e uma lista de valores a serem testados.
18  Output:
19  Caso nenhum dos valores da lista seja válido (conferido por isValidPlacement), retorna Nothing.
20  Caso algum valor da lista seja válido, chama recursivamente solveKojun com a matriz de valores atualizada.
21 -}
22 tryValues :: Matrix Int -> Matrix Char -> Int -> Int -> [Int] -> Maybe (Matrix Int)
23 tryValues _ _ _ [] = Nothing -- Caso não há mais valores para testar, retorna Nothing
24 tryValues vg rg r c (value:rest) -- Desconstrói a lista de valores em head (value) e tail (rest)
25 | isValidPlacement vg rg (r, c) value =
26 | case solveKojun (setMatrixValue vg (r, c) value) rg (r, c + 1) maxRegionSize of
27 | Just result -> Just result
28 | Nothing -> tryValues vg rg r c rest -- Nova chamada de tryValue com o restante da lista
29 | otherwise = tryValues vg rg r c rest
```

A função *tryValues* faz a tentativa de cada um dos valores de 1 a *maxRegionSize* (valor que, nessa implementação, é passado como argumento vindo na chamada de *solveKojun*) para uma célula do tabuleiro. A validação se um valor é válido ou não em uma célula do tabuleiro é feita por *isValidPlacement*. Essa função, em sua essência, consiste implementação a validação de jogadas baseada nas regras do jogo:

1. Inserir um número em cada célula da grade para que cada região de tamanho N contenha cada número de 1 a N exatamente uma vez.
2. Os números em células ortogonalmente adjacentes devem ser diferentes.
3. Se duas células são adjacentes verticalmente na mesma região, o número na célula superior deve ser maior que o número na célula inferior.

Note que para validar corretamente se uma jogada é correta ou não precisamos ter como informação os valores das posições adjacentes (os quais podem ser facilmente obtidos utilizando a função auxiliar *getMatrixValue* implementada pelo módulo *Matrix*) e precisamos também das outras posições do tabuleiro que estão na mesma região. Para obtermos a segunda, foi desenvolvida a função *getRegionFromPosition*.

Veja nas imagens a seguir a implementação das funções *isValidPlacement* e *getRegionFromPosition*.

```

1 isValidPlacement :: Matrix Int -> Matrix Char -> Position -> Int -> Bool
2 isValidPlacement valueGrid regionGrid (row, col) value
3   | getMatrixValue valueGrid (row, col) /= 0 = False -- Verifica se a posição já está ocupada por algum valor diferente de 0
4   | value < 1 || value > regionSize = False           -- Verifica se o número está entre 1 e N, onde N é o tamanho da região
5   | hasAdjacentValue = False                         -- Verifica se algum dos valores adjacentes é igual ao valor que queremos inserir
6   | hasValueInRegion = False                         -- Verifica se o valor já está na região
7   | isTopInvalid = False                             -- Se a posição acima for inválida (for da mesma região e menor em valor), retorna False
8   | isBottomInvalid = False                          -- Se a posição abaixo for inválida (for da mesma região e maior em valor), retorna False
9   | otherwise = True
10  where
11    numRows' = numRows valueGrid
12    numCols' = numCols valueGrid
13    region = getMatrixValue regionGrid (row, col)
14
15    -- Filtra as posições adjacentes que são válidas (dentro da matriz) e obtém os valores das posições adjacentes
16    adjacentPositions =
17      filter (\(r, c) -> r >= 0 && r < numRows' && c >= 0 && c < numCols')
18        [(row - 1, col), (row + 1, col), (row, col - 1), (row, col + 1)]
19    adjacentValues = map (\(r, c) -> getMatrixValue valueGrid (r, c)) adjacentPositions
20    -- Obtém as posições da região que contém a posição (row, col)
21    regionPositions = getRegionFromPosition regionGrid region row col []
22    regionSize = length regionPositions
23
24    -- (1) Verifica se o valor já está presente na região que contém a posição
25    hasValueInRegion = value `elem` map (\(r, c) -> getMatrixValue valueGrid (r, c)) regionPositions
26    -- (2) Verifica se algum dos valores adjacentes é igual ao valor que queremos inserir
27    hasAdjacentValue = any (== value) adjacentValues
28    -- (3) Verifica se a posição acima de (row, col) é da mesma região e se o valor é maior
29    isTopSameRegion = (row - 1, col) `elem` regionPositions
30    isTopInvalid = isTopSameRegion && getMatrixValue valueGrid (row - 1, col) <= value
31    -- (3) Verifica se a posição abaixo de (row, col) é da mesma região e se o valor é menor
32    isBottomSameRegion = (row + 1, col) `elem` regionPositions
33    isBottomInvalid = isBottomSameRegion && getMatrixValue valueGrid (row + 1, col) >= value

```

```

1 getRegionFromPosition :: Eq a => Matrix a -> a -> Int -> Int -> [(Int, Int)] -> [(Int, Int)]
2 getRegionFromPosition regionGrid region row col visited
3   | row < 0 || row >= numRows regionGrid ||
4   | col < 0 || col >= numCols regionGrid ||
5   | getMatrixValue regionGrid (row, col) /= region ||
6   | (row, col) `elem` visited = visited
7   | otherwise =
8     let visited' = (row, col) : visited
9     in foldr
10      (\(r, c) accVisited -> getRegionFromPosition regionGrid region r c accVisited)
11      visited'
12      [(row - 1, col), (row + 1, col), (row, col - 1), (row, col + 1)]

```

A função *getRegionFromPosition* gera, recursivamente, uma lista de posições (representadas por tuplas) de uma região. Ela funciona como uma espécie de *flood fill*, realizando visitas que se espalham a partir de um ponto de origem e passando apenas uma vez por cada posição da região.

## Otimizações realizadas

Para cada célula vazia do tabuleiro, é chamada a função *isValidPlacement* é chamada para cada um dos valores possíveis (em ordem crescente). Dessa forma, a função *isValidPlacement* mostra-se de grande impacto no desempenho do algoritmo de backtracking.

A função *isValidPlacement* chama *getRegionFromPosition* toda vez que é executada. Na primeira implementação realizada, a função *getRegionFromPosition* position varria todas as células da matriz de regiões para gerar a lista de posições (linha, coluna) de uma região. A partir disso, a função foi otimizada

para realizar uma busca recursiva de posições da região a partir de um ponto (*flood fill*), visitando assim apenas  $K$  células (onde  $K$  é o número de elementos daquela região).

Seria possível otimizar mais ainda a execução de *isValidPlacement* gerando um hashmap com todas as posições de cada uma regiões apenas uma vez antes de sua execução e passando essa informação como argumento *isValidPlacement*. Entretanto, como essa otimização envolveria um aumento da complexidade da implementação e como o desempenho não era uma preocupação para essa proposta, essa otimização não foi realizada.

Gerando um hashmap no qual a região a chave e a lista de posições daquela região são os valores seria possível encadear outra otimização: poderíamos remover o argumento *maxRegionSize* de *solveKojun*. Como visto anteriormente, esse argumento define a quantidade total de valores que serão testados em uma posição antes de uma possível falha. Entretanto, sabemos previamente que o maior valor possível para uma posição pertencente a uma região  $X$  de tamanho  $Y$  é  $Y$ . Tendo uma lista de posições de uma região, seria possível fazer com que *maxRegionSize* fosse sempre o tamanho da região daquela posição e não um valor arbitrário (possivelmente muito maior que o tamanho da região). Isso reduziria drasticamente a quantidade de chamadas para *isValidPlacement*.

## Entrada e saída

Para definir qual arquivo .txt será a entrada do programa deve-se alterar o caminho para o arquivo na função *main*.

```
main :: IO ()
main = do
  -- Lê arquivos que contém
  -- O tamanho N da matriz (N x N) na primeira linha
  -- A matriz de valores (ocupando N linhas)
  -- A matriz de regiões (ocupando N linhas)
  contents <- readFile "./inputs/8x8/kojun_4.txt"
```

É possível compilar e executar o programa com o comando ``make run`` no terminal. A como saída, o programa imprime no terminal:

> Matriz de valores:	> Matriz de regiões:	> Solução encontrada:
2 5 0 0 3 0 0 0	'a' 'b' 'b' 'b' 'b' 'c' 'd' 'd'	2 5 1 4 3 4 1 2
0 0 6 0 0 0 0 0	'a' 'a' 'e' 'b' 'c' 'c' 'f' 'f'	1 3 6 2 6 3 2 1
0 0 5 0 5 2 0 0	'g' 'h' 'e' 'i' 'c' 'c' 'j' 'j'	3 1 5 1 5 2 5 2
0 0 0 2 0 0 0 0	'g' 'k' 'e' 'e' 'e' 'c' 'j' 'j'	2 3 4 2 3 1 4 1
0 0 1 0 4 0 0 0	'g' 'k' 'e' 'l' 'l' 'l' 'j' 'm'	1 2 1 3 4 2 3 5
3 0 2 0 0 4 0 0	'n' 'k' 'o' 'l' 'o' 'o' 'm' 'm'	3 1 2 1 5 4 1 4
0 0 0 6 0 0 0 0	'n' 'n' 'o' 'o' 'o' 'p' 'p' 'm'	2 5 1 6 3 2 5 3
0 0 0 0 4 0 3 2	'n' 'n' 'q' 'q' 'p' 'p' 'p' 'm'	1 4 2 1 4 1 3 2

## Dificuldades encontradas

A primeira dificuldade foi encontrada durante a implementação da estrutura de matriz que representa os tabuleiros. Por falta de familiaridade com o funcionamento da linguagem, a operação de escrita na matriz, principalmente, mostrou-se de difícil implementação.

Além disso, tive algumas dúvidas de como deveria ser feito o retorno da função *solveKojun* visto que, além de ser uma função recursiva, poderia ter que retornar um caso em que não foi possível encontrar uma solução. Para resolver esse problema, escolheu-se por utilizar o monad *Maybe* e retornar *Nothing* nos casos em que a solução não é encontrada e *Just Matrix Int* nos casos em que a solução é encontrada.