Campus de Florianópolis

Relatório do Trabalho II de Paradigmas de Programação - INE5416

João Paulo A. Bonomo
Gabriel Lima Jacinto
Rodrigo Santos de Carvalho

Florianópolis

2023

Relatório do Trabalho II de Paradigmas de Programação - INE5416

Professor Maicon Rafael Zatelli

Resumo

Este relatório é referente ao segundo trabalho da disciplina de **Paradigmas de Programação - INE5416**, do trio João Paulo Bonomo, Gabriel Lima Jacinto e Rodrigo Santos de Carvalho. O trabalho em questão é um resolvedor do puzzle Vergleichssudoku, usando a linguagem de programação funcional Elixir.

Sumário

1 Introdução ao Problema Proposto

1.1 Introdução

No arquivo de descrição do Trabalho II , o professor deixou a cargo dos grupos que escolhessem dentre três opções de puzzle:

1. Kojun;
2. Makaro;
3. **Vergleichssudoku**.

Nosso grupo optou pela terceira opção por uma questão de maior proximidade com o sudoku tradicional, e também pela maior simplicidade das regras (o que julgamos que simplificaria nossa implementação em Elixir, uma linguagem que ambos os 3 tínhamos pouco domínio). Além disso, a familiaridade com a solução desenvolvida no Trabalho I ajudaria na implementação dessa atividade.

1.2 Regras e Descrição do Jogo

Sucintamente, as regras do **Vergleichssudoku** são praticamente as mesmas do Sudoku tradicional, sendo acrescentada a presença de sinais de comparação ('>' e '<') entre as células do tabuleiro, o que torna o **Vergleichssudoku** mais fácil do que o próprio Sudoku, uma vez que diminui os números possíveis para cada uma das células no tabuleiro.

Um exemplo de tabuleiro 9x9 inicial de jogo pode ser visualizado na Figura 1, e a solução para esse exemplo pode ser vista na Figura 2.

Todos os tabuleiros que usamos para aprender a jogar, e também como "inputs"para testar nosso código foram retirados do site janko.at. Portanto, nosso trabalho consiste em resolver tabuleiros de tamanho 4x4, 6x6 e 9x9.

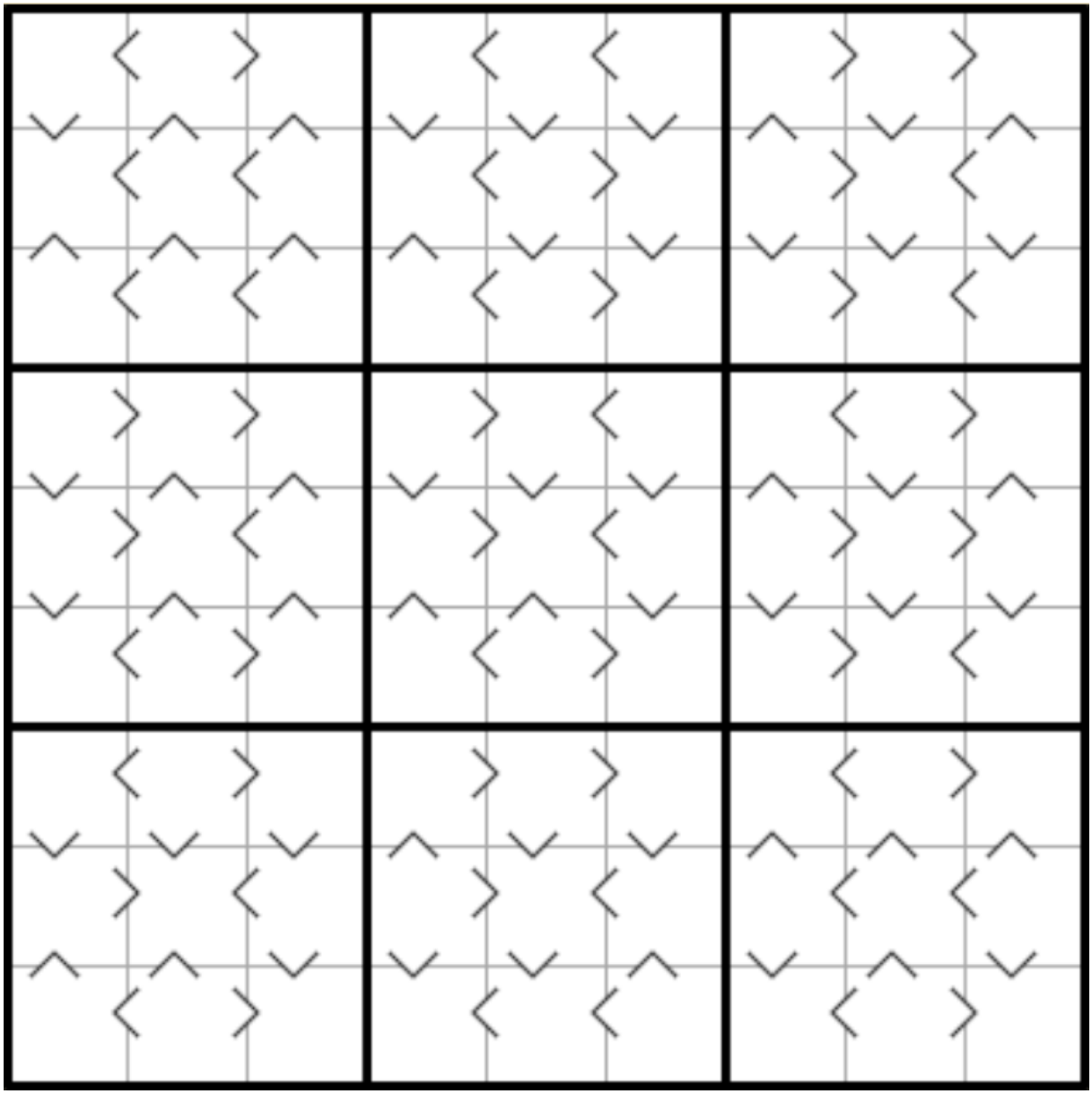


Figura 1 – Exemplo de tabuleiro 9x9 - retirado de janko.at/Raetsel/Sudoku/Vergleich/011.

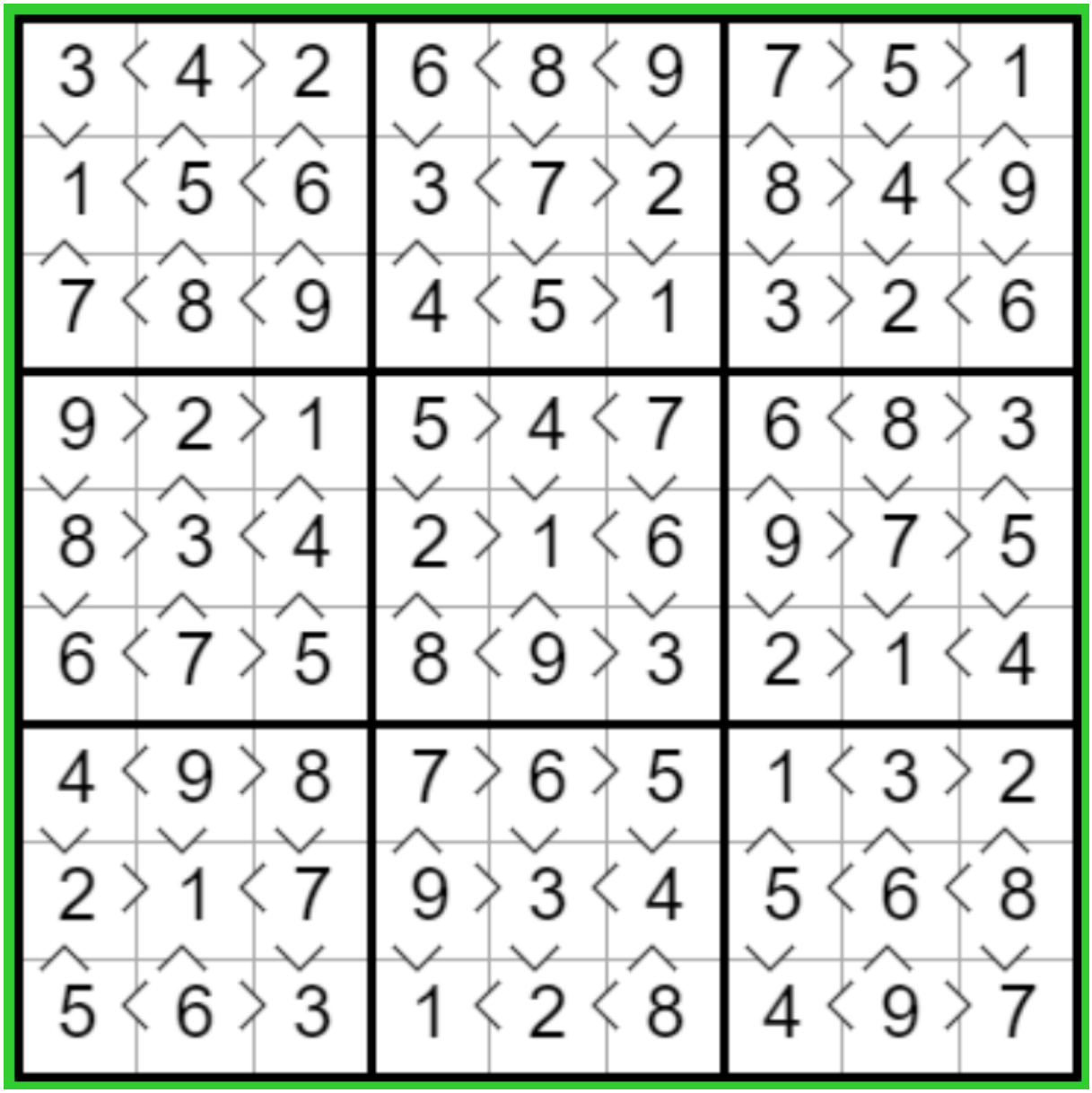


Figura 2 – Resolução da Figura 1 - retirado de janko.at/Raetsel/Sudoku/Vergleich/011.

2 Descrição Detalhada do Programa

2.1 Desenvolvimento da solução

Para o desenvolvimento de nossa solução, nos baseamos na solução do Trabalho I (feito em Haskell), que por sua vez foi baseado no post **Solving 'Greater-than sudoku' with python and z3**. O post traz uma solução em *Python* para o **Vergleichssudoku**, então nos bastou adaptar parte do código para *Elixir* e fazer algumas alterações na lógica para que funcionasse com tabuleiros de tamanhos diferentes de 9x9.

2.2 Técnica de programação escolhida

Após algumas tentativas frustradas de replicar técnicas baseadas em heurísticas (como exemplificado pela solução de P. Norvig em **Solving Every Sudoku Puzzle**), optamos por utilizar apenas a técnica de *backtracking* ("tentativa e erro") no desenvolvimento do projeto.

2.3 Modelagem do tabuleiro

A modelagem do tabuleiro pode ser vista de forma mais detalhada mais adiante na seção 3.1, onde tratamos do "input"do usuário no programa.

2.4 Aplicação

Ao arquivo **vergleichs_sudoku.ex** cabe a função de resolver o puzzle. Tratemos abaixo mais detalhadamente de cada uma das funções desse módulo.

1. A Figura 3 trata das seguintes funções:

    - A função *getXY* recebe uma matriz de elementos, um valor x e um valor y, e retorna o valor na posição (x, y) da matriz;
    - A função *setXY* recebe uma matriz de elementos, um valor r, um valor c e um novo valor val, e retorna uma nova matriz que é a mesma que a matriz original, exceto que o valor na posição (r, c) é atualizado para val;
    - A função *getVergleichssudokuGrid* recebe um tamanho de tabuleiro como entrada e retorna uma matriz de tamanho *sizeBoard*X*sizeBoard* inicializada com todos os valores como 0. Isso é usado para criar o tabuleiro inicial do Sudoku;

- A função *getRow* recebe uma matriz de elementos, um valor x e um valor y, e retorna a linha na posição x da matriz;
- A função *getCol* recebe uma matriz de elementos, um valor x e um valor y, e retorna a coluna na posição y da matriz;
- A função *getRegion* recebe uma matriz de elementos, um valor x, um valor y e o tamanho do tabuleiro como entrada, e retorna a região (submatriz) correspondente à posição (x, y) na matriz. O tamanho da região é calculado com base nas configurações incluídas pelo usuário no arquivo **size_config.ex**.

2. Já a Figura 4 trata das seguintes funções:

   - *compareBigger* e *compareSmaller* são um conjunto de funções auxiliares que utilizam de *pattern matching* com o intuito de comparar o valor em uma determinada posição com o valor em uma posição vizinha, de acordo com a direção fornecida como argumento. Essas funções retornam *True* se a comparação for verdadeira ou se a posição vizinha estiver vazia (valor = 0);
   - Já a função *executeComparison* é responsável por executar uma comparação específica com base no símbolo de comparação fornecido. Se o símbolo for '', a função retorna *True*. Se for '>', a função chama *compareBigger*. Se for '<', a função chama *compareSmaller*;
   - Por fim, *getCompare* é responsável por obter todas as possíveis comparações que podem ser aplicadas em uma determinada célula/posição do tabuleiro. Ele obtém o comparador da posição correspondente no tabuleiro de comparações e, em seguida, para cada valor possível, executa todas as comparações possíveis chamando *executeComparison*. E então, a função retorna uma lista dos valores que podem ser colocados na célula correspondente.

3. A Figura 5, por sua vez, diz respeito às três seguintes funções:

   - A função *getPossibleOptions* recebe uma matriz de **Vergleichssudoku** representada por uma lista de listas, uma matriz de caracteres que representa a mesma matriz de **Vergleichssudoku**, e as coordenadas de uma posição na matriz. A função retorna uma lista com os números possíveis para serem colocados nessa posição;
   - Já a *getValueInList* recebe dois argumentos: uma lista e um inteiro. Ela retorna o valor do elemento i-ésimo da lista. Se o valor do inteiro for maior ou igual ao comprimento da lista, um erro é retornado;
   - Por último, a função *getListLenght* apenas retorna o tamanho da lista de entrada.

4. Na Figura 6 temos uma das funções mais importantes do código: *solveVergleichsSudoku*.

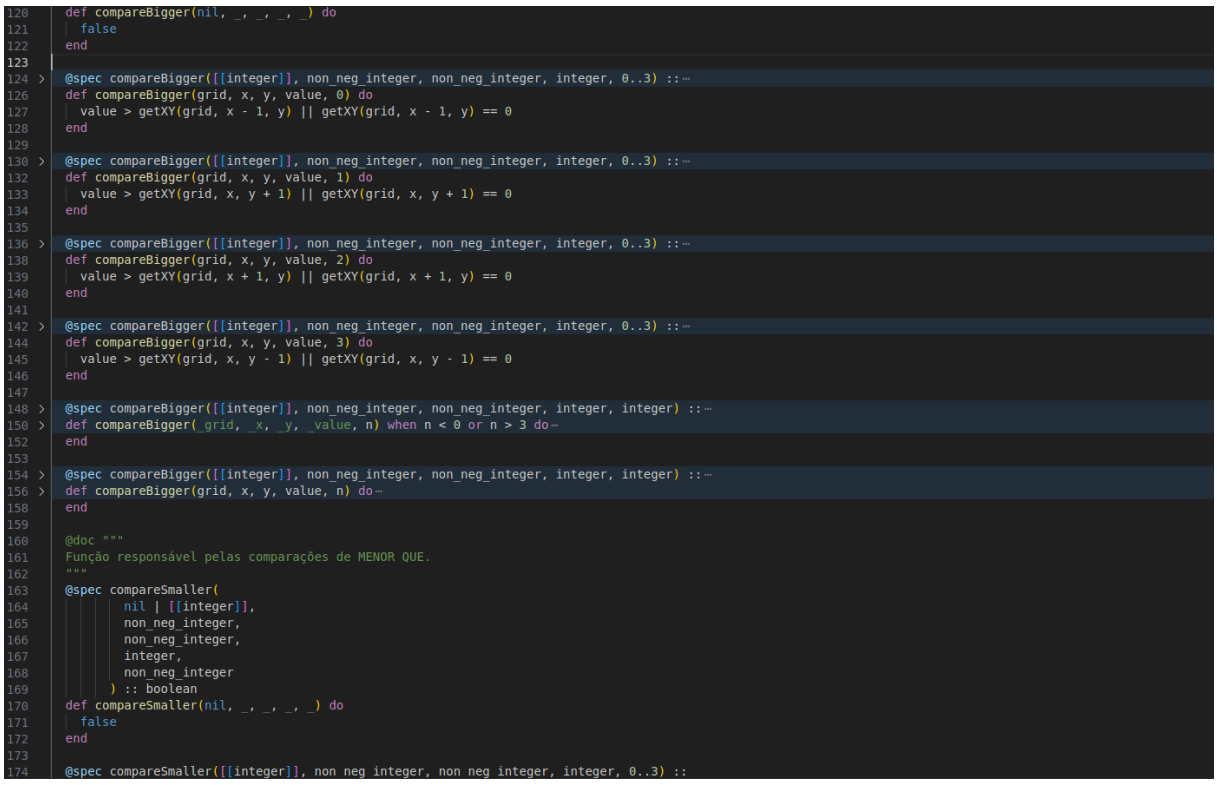
    - A função é responsável por resolver o tabuleiro do puzzle. Ela recebe como entrada a grade do jogo *vergleichssudokuGrid* e a grade de comparadores *comparatorsGrid*, a linha e a coluna atual;
    - A função começa verificando se a célula atual é a última do tabuleiro. Se for, a solução é encontrada e a grade completa é retornada. Caso contrário, ela verifica se chegou ao final da linha e passa para a próxima linha. Em seguida, verifica se o valor da célula já foi definido e passa para a próxima célula, caso contrário, ela chama a função *solveVergleichssudokuWithValues* (tratada no tópico abaixo e visualizada na Figura 7, que testa os valores possíveis para a célula atual.

5. Por último, temos a função *solveVergleichsSudokuWithValues*, que está representada na Figura 7.

    - A função é responsável por testar os valores possíveis para a célula atual;
    - Se a lista de possíveis valores estiver vazia, não há solução possível para o tabuleiro de entrada. Se o índice ultrapassar o tamanho da lista de possíveis valores, não há solução possível para a célula atual. Em seguida, a função seta o valor encontrado na lista de possibilidades no índice recebido e chama a função solveVergleichssudoku para testar a próxima célula;
    - Se *solveVergleichssudoku* retornar *nil*, isso significa que não houve solução para a célula atual com o valor encontrado, então a função reseta o valor da célula atual e chama a si mesma para testar o próximo valor na lista de possíveis valores. Se *solveVergleichssudoku* retornar uma solução, ela é retornada pela função *solveVergleichssudokuWithValues*.

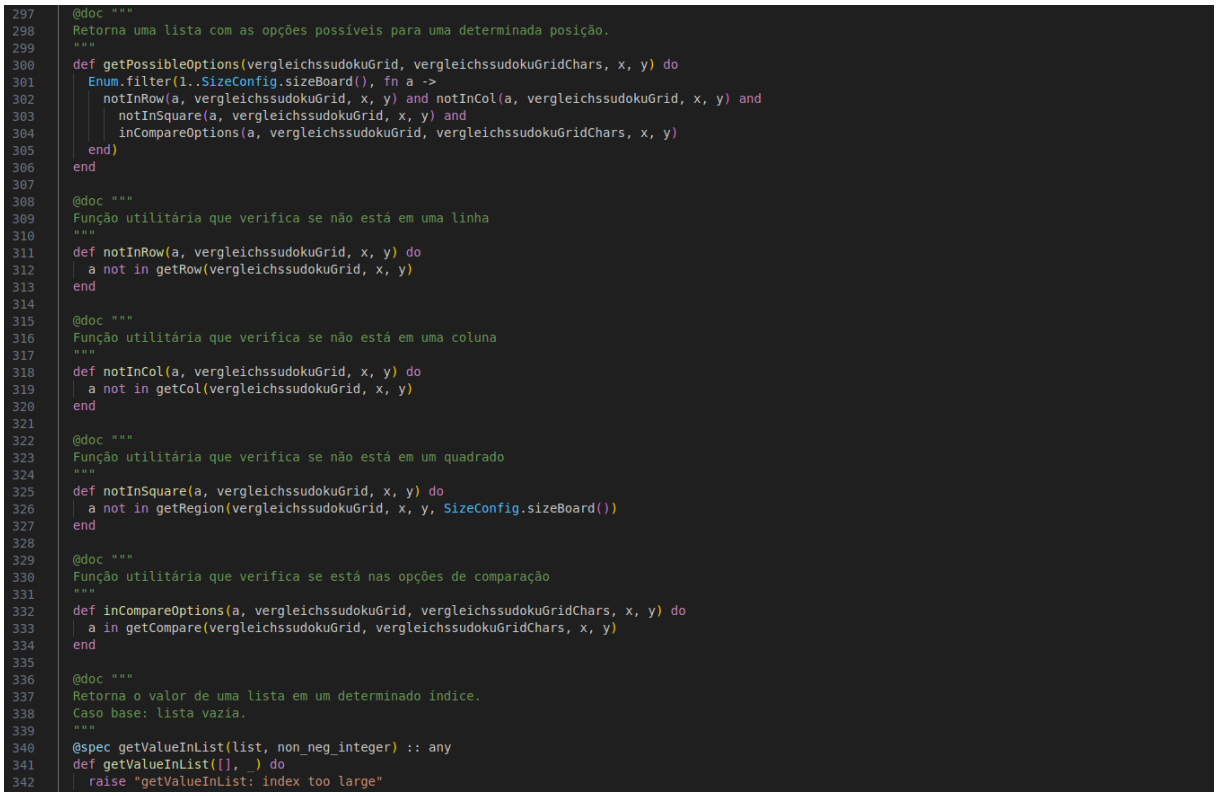
```elixir
def getXY(nil, _, _) do
  raise "getXY: Nothing"
end

@doc """
Retorna o valor de uma matriz em uma determinada posição.
Caso recursivo: retorna o valor da posição (x, y) da matriz.
"""
def getXY(grid, x, y) do
  Enum.at(Enum.at(grid, x), y)
end

@doc """
Altera o valor de uma matriz em uma determinada posição (x, y).
Caso base: matriz vazia.
"""
def setXY(nil, _, _, _, _) do
  raise "setXY: Nothing"
end

@doc """
Altera o valor de uma matriz em uma determinada posição (x, y).
"""
def setXY(grid, r, c, val) do
  new_grid = List.replace_at(grid, r, List.replace_at(Enum.at(grid, r), c, val))
  new_grid
end

@doc """
Retorna a matriz inicial do tabuleiro, com o tamanho configurado pelo usuário em SizeConfig.hs.
"""
@spec getVergleichsSudokuGrid(non_neg_integer) :: nil | [[integer]]
def getVergleichsSudokuGrid(sizeBoard) when sizeBoard > 0 do
  grid = List.duplicate(List.duplicate(0, sizeBoard), sizeBoard)
  grid
end

def getVergleichsSudokuGrid(_sizeBoard) do
  nil
end
```

Figura 3 – Primeiras funções de /lib/**vergleichs_sudoku.ex**.



```elixir
def compareBigger(nil, _, _, _, _) do
  false
end

@spec compareBigger([[integer]], non_neg_integer, non_neg_integer, integer, 0..3) ::…
def compareBigger(grid, x, y, value, 0) do
  value > getXY(grid, x - 1, y) || getXY(grid, x - 1, y) == 0
end

@spec compareBigger([[integer]], non_neg_integer, non_neg_integer, integer, 0..3) ::…
def compareBigger(grid, x, y, value, 1) do
  value > getXY(grid, x, y + 1) || getXY(grid, x, y + 1) == 0
end

@spec compareBigger([[integer]], non_neg_integer, non_neg_integer, integer, 0..3) ::…
def compareBigger(grid, x, y, value, 2) do
  value > getXY(grid, x + 1, y) || getXY(grid, x + 1, y) == 0
end

@spec compareBigger([[integer]], non_neg_integer, non_neg_integer, integer, 0..3) ::…
def compareBigger(grid, x, y, value, 3) do
  value > getXY(grid, x, y - 1) || getXY(grid, x, y - 1) == 0
end

@spec compareBigger([[integer]], non_neg_integer, non_neg_integer, integer, integer) ::…
def compareBigger(_grid, _x, _y, _value, n) when n < 0 or n > 3 do…
end

@spec compareBigger([[integer]], non_neg_integer, non_neg_integer, integer, integer) ::…
def compareBigger(grid, x, y, value, n) do…
end

@doc """
Função responsável pelas comparações de MENOR QUE.
"""
@spec compareSmaller(
        nil | [[integer]],
        non_neg_integer,
        non_neg_integer,
        integer,
        non_neg_integer
      ) :: boolean
def compareSmaller(nil, _, _, _, _) do
  false
end

@spec compareSmaller([[integer]], non_neg_integer, non_neg_integer, integer, 0..3) ::
```

Figura 4 – Funções que tratam das comparações entre as células do tabuleiro.

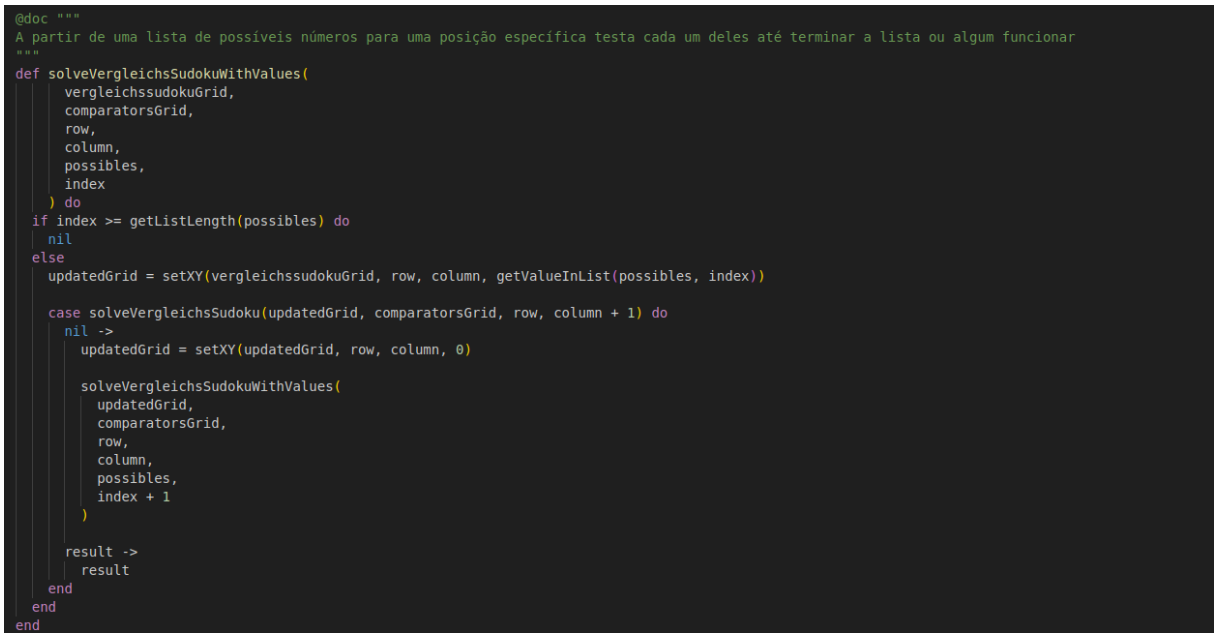


```elixir
@doc """
Retorna uma lista com as opções possíveis para uma determinada posição.
"""
def getPossibleOptions(vergleichssudokuGrid, vergleichssudokuGridChars, x, y) do
  Enum.filter(1..SizeConfig.sizeBoard(), fn a ->
    notInRow(a, vergleichssudokuGrid, x, y) and notInCol(a, vergleichssudokuGrid, x, y) and
      notInSquare(a, vergleichssudokuGrid, x, y) and
      inCompareOptions(a, vergleichssudokuGrid, vergleichssudokuGridChars, x, y)
  end)
end

@doc """
Função utilitária que verifica se não está em uma linha
"""
def notInRow(a, vergleichssudokuGrid, x, y) do
  a not in getRow(vergleichssudokuGrid, x, y)
end

@doc """
Função utilitária que verifica se não está em uma coluna
"""
def notInCol(a, vergleichssudokuGrid, x, y) do
  a not in getCol(vergleichssudokuGrid, x, y)
end

@doc """
Função utilitária que verifica se não está em um quadrado
"""
def notInSquare(a, vergleichssudokuGrid, x, y) do
  a not in getRegion(vergleichssudokuGrid, x, y, SizeConfig.sizeBoard())
end

@doc """
Função utilitária que verifica se está nas opções de comparação
"""
def inCompareOptions(a, vergleichssudokuGrid, vergleichssudokuGridChars, x, y) do
  a in getCompare(vergleichssudokuGrid, vergleichssudokuGridChars, x, y)
end

@doc """
Retorna o valor de uma lista em um determinado índice.
Caso base: lista vazia.
"""
@spec getValueInList(list, non_neg_integer) :: any
def getValueInList([], _) do
  raise "getValueInList: index too large"
```

Figura 5 – Algumas das funções auxiliares de /lib/**vergleichs_sudoku.ex**.

```elixir
@doc """
Função responsável pela solução do tabuleiro.
Recebe o tabuleiro, o tabuleiro de comparações, a linha atual e a coluna atual.
"""
def solveVergleichsSudoku(vergleichssudokuGrid, comparatorsGrid, row, column) do
  case {row == SizeConfig.sizeBoard() - 1 && column == SizeConfig.sizeBoard(),
        column == SizeConfig.sizeBoard(), getXY(vergleichssudokuGrid, row, column) > 0} do
    # Retorna o tabuleiro caso tenha chegado na última célula.
    {true, _, _} ->
      IO.puts("Found the solution: ")
      vergleichssudokuGrid

    # Verifica se chegou ao fim de uma linha, caso tenha chegado, salta para a próxima.
    {_, true, _} ->
      solveVergleichsSudoku(vergleichssudokuGrid, comparatorsGrid, row + 1, 0)

    # Verifica se o valor da célula atual já foi definido, caso tenha sido, passa para a próxima célula.
    {_, _, true} ->
      IO.puts("Value already defined")
      solveVergleichsSudoku(vergleichssudokuGrid, comparatorsGrid, row, column + 1)

    _ ->
      # Checa os valores possíveis para a célula atual.
      possibles = getPossibleOptions(vergleichssudokuGrid, comparatorsGrid, row, column)

      # Após validar a posição e adquirir os possíveis números chama a função recursiva que testa cada um deles.
      solveVergleichsSudokuWithValues(
        vergleichssudokuGrid,
        comparatorsGrid,
        row,
        column,
        possibles,
        0
      )
  end
end
```

Figura 6 – Funções principais que resolvem o tabuleiro (sem valor).



```
@doc """
A partir de uma lista de possíveis números para uma posição específica testa cada um deles até terminar a lista ou algum funcionar
"""
def solveVergleichsSudokuWithValues(
      vergleichssudokuGrid,
      comparatorsGrid,
      row,
      column,
      possibles,
      index
    ) do
  if index >= getListLength(possibles) do
    nil
  else
    updatedGrid = setXY(vergleichssudokuGrid, row, column, getValueInList(possibles, index))

    case solveVergleichsSudoku(updatedGrid, comparatorsGrid, row, column + 1) do
      nil ->
        updatedGrid = setXY(updatedGrid, row, column, 0)

        solveVergleichsSudokuWithValues(
          updatedGrid,
          comparatorsGrid,
          row,
          column,
          possibles,
          index + 1
        )

      result ->
        result
    end
  end
end
```

Figura 7 – Funções principais que resolvem o tabuleiro (com valor).

3 *Input* e *Output*

3.1 *Input*

Optamos pela inserção dessas relações diretamente no código fonte.

Para inserir um *input*, primeiramente, o usuário precisa alterar as linhas 3, 6 e 9 do arquivo **sizelib_config.ex**, que está na pasta **/lib/**. Os comentários nas linhas, que podem ser vistos na Figura 8, são o suficiente para que o usuário não cometa nenhum engano.

```
defmodule SizeConfig do
  @moduledoc """
  Configura o tabuleiro
  """
  @sizeBoard 9
  @sizeRowRegion 3
  @sizeColumnRegion 3

  @doc """
  Variável referente ao tamanho do tabuleiro.
  Usuario pode escolher entre 4, 6 e 9.
  """
  @spec sizeBoard() :: integer
  def sizeBoard(), do: @sizeBoard

  @doc """
  Variável referente ao número de linhas de cada região do tabuleiro.
  Para um sizeBoard == 4, sizeRowRegion = 2; Para um sizeBoard == 6 ou sizeBoard == 9, sizeRowRegion = 3.
  """
  @spec sizeRowRegion() :: integer
  def sizeRowRegion(), do: @sizeRowRegion

  @doc """
  Variável referente ao número de colunas de cada região do tabuleiro.
  Para um sizeBoard == 4 ou sizeBoard == 6, sizeRowRegion = 2; Para um sizeBoard == 9, sizeRowRegion = 3.
  """
  @spec sizeColumnRegion() :: integer
  def sizeColumnRegion(), do: @sizeColumnRegion
end
```

Figura 8 – Arquivo /lib/size_config.ex.

Após definir as dimensões do tabuleiro, basta ao usuário definir as relações de comparação de cada uma das células do tabuleiro. Como isso pode ser um trabalho chato, deixamos alguns exemplos já formatados para o usuário na pasta /test/inputs/. Essa pasta está representada neste relatório na Figura 12. As relações devem ser redefinadas no arquivo **relations_board.ex** (que pode ter seu trecho inicial visto na Figura 9), que está na pasta **/lib/**. Caso o usuário tenha definido um tabuleiro 4x4 em **size_config.ex**, o usuário deve alterar as linhas de 6 a 9. Para um tabuleiro 6x6, as linhas de 12 a 17, e para um tableiro 9x9, as linhas de 20 a 28. Importante frisar que uma lista de caracteres (*string*), que é a forma que optamos por representar o tabuleiro e suas relações, é declarada com ” (aspas simples) em Elixir, enquanto na solução feita em Haskell era utilizado o

símbolo de aspas dupla.

Em relação à formatação dos *inputs*: cada linha representa uma linha do tabuleiro, contendo as 4 relações de cada célula daquela linha, separando as células por uma '|'. A ordem das relações para cada célula segue o sentido horário: **ACIMA, À DIREITA, ABAIXO, À ESQUERDA**. A Figura 10 demonstra como o exemplo da Figura 1 fica após a formatação do tabuleiro para o *input*.

```elixir
defmodule RelationsBoard do
  @moduledoc """
  Módulo que faz o parser do tabuleiro para extrair as informações para o solucionador
  """
  # INPUT 0001
  @row1size4 '.<>.|..<>|.<<.|..>>|'
  @row2size4 '<<..|>..>|>>..|<..<|'
  @row3size4 '.>>.|..<<|.<<.|..>>|'
  @row4size4 '<>..|>..<|>>..|<..<|'

  # INPUT 004
  @row1size6 '.<<.|..>>|.><.|..<<|.><.|..><|'
  @row2size6 '>>>.|<.><|>>>.|>.<<|>><.|<.<<|'
  @row3size6 '<<..|<..>|<<..|>..>|><..|>..>|'
  @row4size6 '.<<.|..<>|.<>.|..>>|.<<.|..>>|'
  @row5size6 '>>>.|>.><|<<<.|<.<>|><<.|<.>>|'
  @row6size6 '<>..|<..<|><..|>..>|>>..|<..<|'

  # INPUT 011
  @row1size9 '.<>.|.><>|..<<|.<>.|.<>>|..>>|.><.|.>><|..<<|'
  @row2size9 '<<<.|><<>|>.<>|<<<.|<>>>|<.><|>>>.|<<><|>.>>|'
  @row3size9 '><..|><.>|>..>|><..|<>.>|<..<|<>..|<<.<|<..>|'
  @row4size9 '.>>.|.><<|..<<|.>>.|.<><|..>>|.<<.|.>>>|..<<|'
  @row5size9 '<>>.|><<<|>.<>|<><.|<<<<|<.>>|>>>.|<>><|>.><|'
  @row6size9 '<<..|>>.>|>..<|><..|>>.>|<..<|<>..|<<.<|<..>|'
  @row7size9 '.<>.|.>>>|..><|.><.|.>><|..><|.<<.|.><>|..<<|'
  @row8size9 '<><.|<<<<|<.>>|>>>.|<<><|<.<>|><>.|><<>|>.>>|'
  @row9size9 '><..|>>.>|<..<|<<..|<<.>|>..>|<<..|>>.>|<..<|'

  @doc """
  Retorna as linhas do tabuleiro dado o tamanho dele.
  Pode ser 9, 6 ou 4
  """
  @spec allRows(integer) :: list(binary)
  def allRows(size) do
    case size do
      9 ->
        [
          @row1size9,
          @row2size9,
          @row3size9,
          @row4size9,
          @row5size9,
          @row6size9,
          @row7size9,
          @row8size9,
          @row9size9
        ]
```

Figura 9 – Trecho inicial do arquivo /lib/relations_board.ex.

3.2 *Output*

A Figura 11 demonstra a saída de nosso programa para o *input* da Figura 10. Como pode ser visto, ele é compatível com a Figura 2. Por simplificação de código, nossa saída contém apenas os números ordenados de acordo com sua posição no tabuleiro, sem a presença de divisórias verticais ou horizontais entre as células ou regiões do tabuleiro. Apesar de simples, o formato do *output* não atrapalha ou dificulta a sua visualização e interpretação.

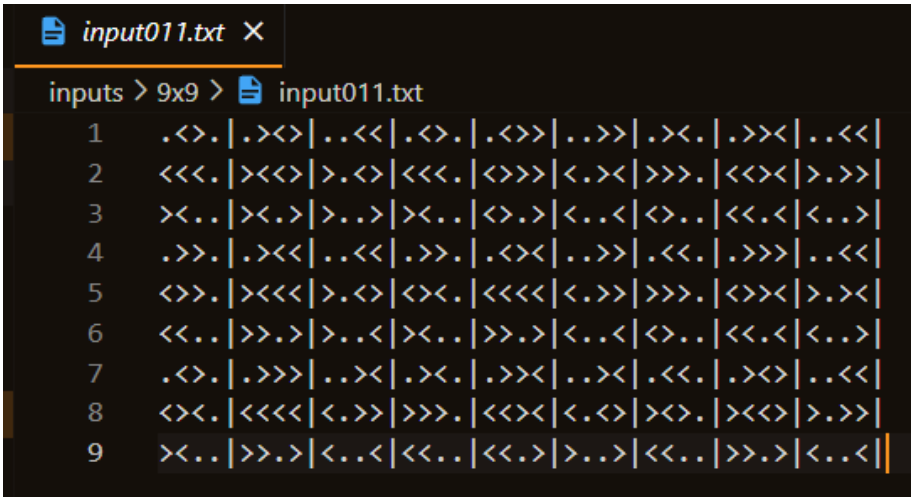


Figura 10 – Exemplo da Figura 2 formatado.

Figura 11 – Output gerado por nosso programa para o *input* da figura 10.

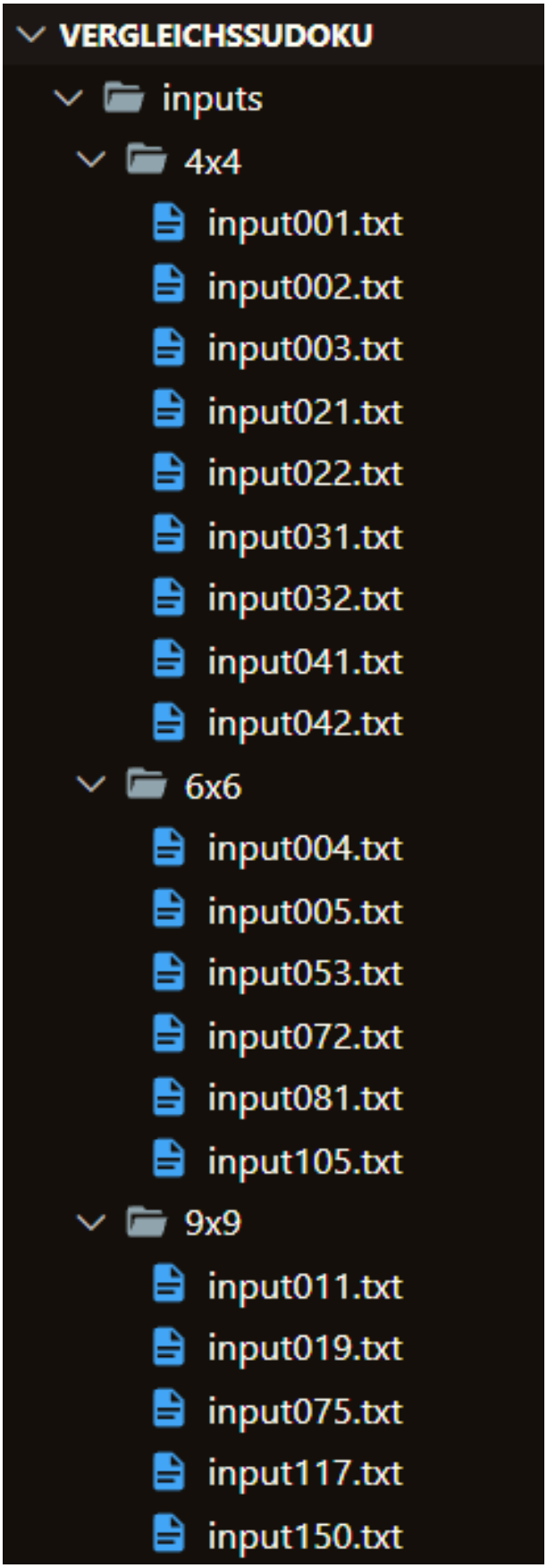


Figura 12 – Pasta de *inputs* de nosso repositório.

4 Organização do Grupo

4.1 Comunicação do Grupo

Para organização do trabalho, o grupo utilizou de algumas ferramentas de comunicação como:

- grupo de **Whatsapp**;
- aba de mensagens do **Moodle**.

4.2 Gerenciamento do Código

Para o gerenciamento do código do trabalho, utilizamos apenas o **GitHub**. Mantendo um repositório privado com apenas os 3 membros como colaboradores.

4.3 Ambiente de Desenvolvimento

A IDE (Integrated Development Environment) escolhida para o desenvolvimento do trabalho foi o **VSCode**, com o auxílio das extensões **Elixir**, **Elixir Syntax Highlighting**. Além disso, para organização e formatação das pastas e código do projeto foi utilizado a ferramenta de *build* automático do Elixir chamado *Mix*.

5 Dificuldades Encontradas

No desenvolvimento do projeto, algumas dificuldades foram encontradas, especialmente por nenhum de nós ter experiência com *Elixir*. Abaixo estão listadas algumas dessas negativas.

- **Tipagem de variáveis do *Elixir***: a tipagem de variáveis do *Elixir* é feita dinâmicamente durante o *runtime* da apicação. Dessa forma, a solução encontrada pelo grupo foi utilizar da diretiva *@spec* () para facilitar a leitura e documentação das funções;
- **Testes**: testar o código inicialmente era uma tarefa chata, tendo diversos problemas de compilação e, posteriormente, problemas com soluções erradas e/ou falta de solução para alguns dos tabuleiros (erros esses muitas vezes oriundos de um único comparador digitado incorretamente no arquivo de input). Por isso optamos por criar uma pasta específica para testes com os casos declarados na pasta /test/input/ e sua solução esperada.

Alguns outros problemas foram encontrados ao longo do projeto, mas foram menos marcantes/frustrantes, como importação de bibliotecas e outros módulos.

Para nosso segundo contato com uma linguagem funcional, foi uma experiência interessante e enriquecidora e que demandou uma curva de aprendizagem menor do que quando desenvolvemos a solução para o Trabalho I em *Haskell*. Uma vez que as dificuldades foram contornadas, foi muito prazeroso e divertivo programar em *Elixir* nesse projeto.