Universidade Federal de Santa Catarina - UFSC
Departamento de Informatica e Estatistica (INE)
Disciplina: INE5416 - Paradigmas de Programação
Aluno: Lucas Broering dos Santos (22100909)

Relatório Kojun T2

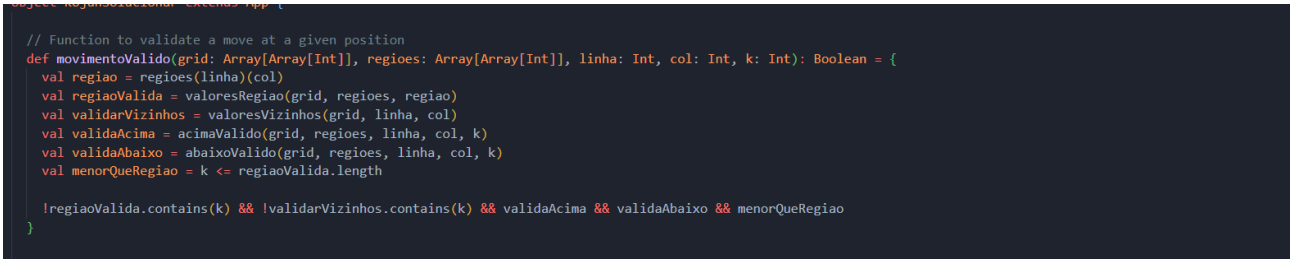
Problema:

O Kojun é um puzzle lógico onde o tabuleiro é dividido em regiões, cada região precisa ter suas casas preenchidas do 1 ao tamanho da região. Ex: Região de tamanho 5, em suas cédulas terão valores de 1 … 5. Seria uma variação muito distante do sudoku. Ele possui algumas regras básicas, como:

- Não pode haver casas com valores repetidos adjacentes.
- Dentro da região não pode haver números repetidos na linha.
- Os valores de uma coluna na mesma região precisam estar em ordem decrescente.

Estratégias de Solução:

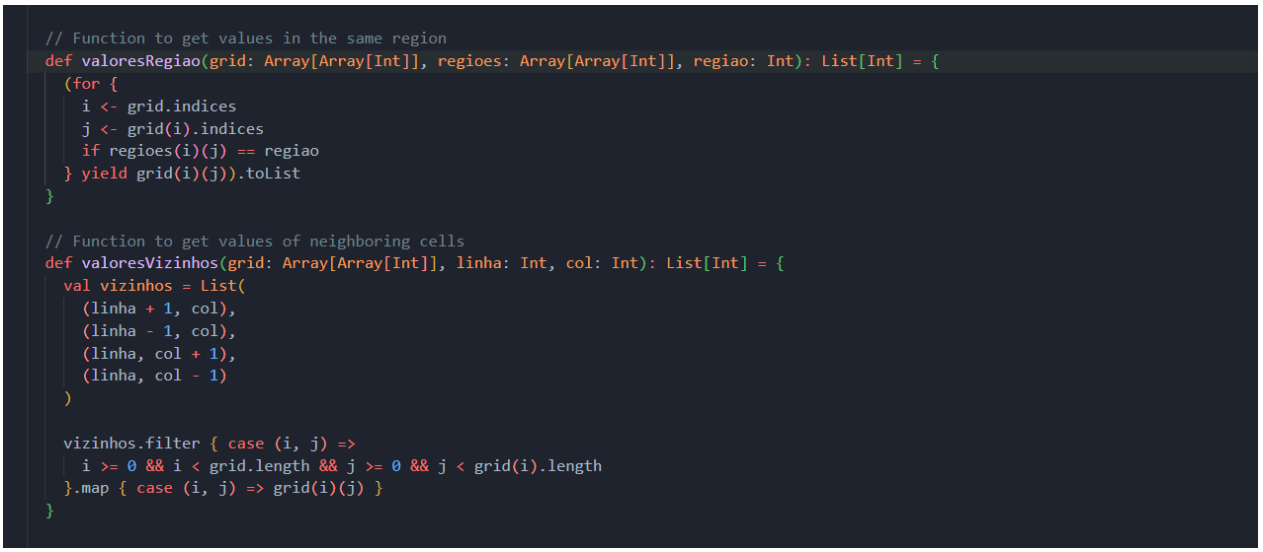
Desenvolvi a solução utilizando a linguagem Scala, abaixo vou descrever algumas funções utilizadas:



```scala
// Function to validate a move at a given position
def movimentoValido(grid: Array[Array[Int]], regioes: Array[Array[Int]], linha: Int, col: Int, k: Int): Boolean = {
  val regiao = regioes(linha)(col)
  val regiaoValida = valoresRegiao(grid, regioes, regiao)
  val validarVizinhos = valoresVizinhos(grid, linha, col)
  val validaAcima = acimaValido(grid, regioes, linha, col, k)
  val validaAbaixo = abaixoValido(grid, regioes, linha, col, k)
  val menorQueRegiao = k <= regiaoValida.length

  !regiaoValida.contains(k) && !validarVizinhos.contains(k) && validaAcima && validaAbaixo && menorQueRegiao
}
```

Verifica se o movimento é válido, ou seja, faz todas as verificações do kojun, descritas acima e retorna um boolean.



```scala
// Function to get values in the same region
def valoresRegiao(grid: Array[Array[Int]], regioes: Array[Array[Int]], regiao: Int): List[Int] = {
  (for {
    i <- grid.indices
    j <- grid(i).indices
    if regioes(i)(j) == regiao
  } yield grid(i)(j)).toList
}

// Function to get values of neighboring cells
def valoresVizinhos(grid: Array[Array[Int]], linha: Int, col: Int): List[Int] = {
  val vizinhos = List(
    (linha + 1, col),
    (linha - 1, col),
    (linha, col + 1),
    (linha, col - 1)
  )

  vizinhos.filter { case (i, j) =>
    i >= 0 && i < grid.length && j >= 0 && j < grid(i).length
  }.map { case (i, j) => grid(i)(j) }
}
```

Exemplo de duas funções utilizadas na função movimentoValido. Essas duas em questão validam os valores possíveis para tal região e valores vizinhos, respectivamente. O valoresRegiao retorna uma lista dos valores da região. A valores vizinhos retorna uma lista dos valores vizinhos.

```scala
// Function to try placing values at a cell
def tentarPosicionar(grid: Array[Array[Int]], regioes: Array[Array[Int]], linha: Int, col: Int, ks: List[Int]): Option[Array[Array[Int]]] = {
  val novoGrid = grid.map(_.clone) // Create a new copy of the grid
  if (ks.isEmpty) None // Base case: no more values to try, return None
  else {
    if (movimentoValido(grid, regioes, linha, col, ks.head)) { // Check if placing the value is valid
      novoGrid(linha)(col) = ks.head // Place the value
      solucionar(novoGrid, regioes, linha, col + 1) match { // Recur for next cell
        case Some(solution) => Some(solution) // If solution found, return it
        case None => tentarPosicionar(grid, regioes, linha, col, ks.tail) // Otherwise, try next value
      }
    } else {
      tentarPosicionar(grid, regioes, linha, col, ks.tail) // If move not valid, try next value
    }
  }
}
```

Essa função tenta posicionar um valor, fazendo todas as verificações para ver se é válido, Ela faz o backtracking, chamando a função recursivamente até que seja solucionado, de modo exaustivo, tentando todos os valores possíveis.

```scala
// Main solving function
def solucionar(grid: Array[Array[Int]], regioes: Array[Array[Int]], linha: Int, col: Int): Option[Array[Array[Int]]] = {
  if (linha == grid.length) Some(grid) // Base case: if reached end of linhas, solution found
  else if (col == grid(linha).length) solucionar(grid, regioes, linha + 1, 0) // Move to next linha
  else if (grid(linha)(col) != 0) solucionar(grid, regioes, linha, col + 1) // Cell already filled, move to next cell
  else {
    val kValues = (1 to grid.length).toList // List of possible values to try
    tentarPosicionar(grid, regioes, linha, col, kValues) // Try placing values at current cell
  }
}
```

A função solucionar testa todos os valores até que exista uma solução para o grid, faz algumas verificações dentro da função, como se a solução foi encontrada, se a posição já está ocupada e por último vai chamando tentarPosicionar até que o grid seja solucionado.

```scala
}

// Function to print the grid
def printGrid(grid: Array[Array[Int]]): Unit = {
  grid.foreach(linha => println(linha.mkString(" ")))
}

// Try to solve the grid and print the solution or a message if no solution found
solucionar(valores10x10, regioes10x10, 0, 0) match {
  case Some(gridSolucionado) =>
    println("Solution found:")
    printGrid(gridSolucionado) // Print the solucionard grid
  case None =>
    println("No solution found.") // No solution found
}
}
```

Por último a função printGrid, imprime o grid no terminal e no final estou chamando o solucionar, caso haja solução ela é imprimida no terminal caso não uma mensagem é imprimida em seu lugar. Os grids, de região e valores estão representados por arrays como no t1. Utilizei o mesmo puzzle, com valores e regiões.

Conclusão

Novamente a maior barreira para mim vou se adaptar ao paradigma de programação. Aprender a linguagem também foi um pouco complicado, não encontrei muitos tutoriais de scala, somente em seu site oficial. Tentar adaptar as funções feitas em haskell para essa linguagem foi interessante..Foi de grande aprendizado realizar esse trabalho utilizando outra linguagem com o mesmo paradigma.