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

08/11/2023

Aluno:

André Amaral Rocco

Relatório - *Kojun Solver* - Scala

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.

Solução adotada

A solução proposta segue uma lógica similar ao que foi desenvolvido para o Trabalho I em Haskell. Entretanto, foi realizada uma modificação com o objetivo de melhorar a eficiência do algoritmo: foi adotado o uso de um hashmap que armazena as posições de cada uma das regiões e que é consultado durante o processo de conferir se um valor pode ser inserido em uma posição do tabuleiro.

Modelagem do tabuleiro

A representação foi mantida da adotada no Trabalho I:

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 em Scala, 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.

Otimizações realizadas

Como mencionado anteriormente, foi implementada a otimização mencionada (mas não adotada) no Trabalho I: “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*”. Nesse caso, o atributo *regionMap* que contém um hashmap cuja chave é

a letra da região e o valor é a lista de posições pertencentes a aquela região é gerado quando *KojunSolver* é instanciado. Isso é feito pelo método *getRegionMap*.

Além de otimizar as consultas às regiões, essa otimização nos permite executar *tryValues* apenas para valores de 1 ao tamanho da região a qual a posição atual pertence, fator que não pôde ser implementado na versão em Haskell.

Funcionamento do programa principal

O programa principal inicia lendo e extraindo as matrizes do arquivos .txt cujo caminho é definido em *filePath*. A partir das linhas do arquivo de texto, inicializa instâncias *Matrix* que correspondem à matriz de valores e à matriz de regiões. Em seguida, é instanciado um objeto de *KojunSolver* e chamado o método *solve()* que aplica o algoritmo de backtracking desenvolvido sobre o tabuleiro (representado pelas duas matrizes).

```
object Main {
  def main(args: Array[String]): Unit = {
    // 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)
    val filePath = "src/inputs/10x10/kojun_10.txt" // Relative path to the file

    // Abre e lê o arquivo
    val fileContents = openFile(filePath)
    var valueGrid: List[List[Int]] = List()
    var regionGrid: List[List[String]] = List()

    fileContents match {
      case Some(contents) =>
        val matrixSize = contents.head.toInt
        // Lê a matriz de valores (inteiros) e regiões (caracteres)
        valueGrid = contents.slice(1, matrixSize + 1).map(line => line.split( regex = " ").map(_.toInt).toList)
        regionGrid = contents.slice(matrixSize + 1, matrixSize * 2 + 1).map(line => line.split( regex = " ").map(filterAlphabetic).toList)

      case None =>
        println("An error occurred while opening the file.")
        return
    }

    // Cria uma matriz de valores e regiões
    val valueMatrix = new Matrix[Int](valueGrid)
    val regionMatrix = new Matrix[String](regionGrid)

    // Imprime a matriz de valores e regiões
    println("Value matrix:")
    valueMatrix.printMatrix()
    println("\nRegion matrix:")
    regionMatrix.printMatrix()
```

```
    // Resolve o quebra-cabeça
    val solver = new KojunSolver(valueMatrix, regionMatrix)
    val solution = solver.solve()

    // Imprime a solução
    solution match {
      case Some(solutionMatrix) =>
        println("\nSolution:")
        solutionMatrix.printMatrix()
      case None =>
        println("No solution found.")
    }
  }

  // Função para filtrar caracteres alfabéticos
  private def filterAlphabetic(str: String): String = {
    str.filter(_.isLetter)
  }

  private def openFile(filePath: String): Option[List[String]] = {
    try {
      val source = Source.fromFile(filePath)
      val lines = source.getLines.toList
      source.close()
      Some(lines)
    } catch {
      case e: Exception =>
        println(s"An error occurred: ${e.getMessage}")
        None
    }
  }
}
```

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.

A classe *KojunSolver* contém as funções responsáveis por aplicar o algoritmo de backtracking. Como mencionado, a classe é instanciada recebendo as duas matrizes que representam o tabuleiro e é aplicado o algoritmo a partir da função *solve()* a qual inicia a cadeia de chamada recursivas sobre a função s*olveKojun(valueGrid, regionGrid, (0, 0), regionMap)*. Essa última 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).

```scala
/**
 * Resolve o Kojun a partir da uma matriz de valores (Int) e da matriz de regiões (Char).
 * Por ser uma função recursiva, recebe um ponto de partida (linha e coluna) para aplicar o backtracking. A função
 * normalmente é chamada externamente com (0, 0).
 * O retorno é um Option que pode ser None caso não exista solução ou Some caso exista.
 */
@tailrec
private def solveKojun(
  valueGrid: Matrix[Int],
  regionGrid: Matrix[String],
  position: Position,
  regionMap: mutable.HashMap[String, List[(Int, Int)]]
): Option[Matrix[Int]] = {
  val (row, col) = position
  if (row == valueGrid.numRows) {
    Some(valueGrid)
  } else if (col == valueGrid.numCols) {
    solveKojun(valueGrid, regionGrid, (row + 1, 0), regionMap)
  } else if (valueGrid.isValidPosition(position) && valueGrid.getMatrixValue(position) != 0) {
    solveKojun(valueGrid, regionGrid, (row, col + 1), regionMap)
  } else {
    val maxRegionSize = regionMap(regionGrid.getMatrixValue(position)).length
    tryValues(row, col, (1 ≤ to ≤ maxRegionSize).toList, valueGrid, regionGrid, regionMap)
  }
}

def solve(): Option[Matrix[Int]] = {
  solveKojun(valueGrid, regionGrid, (0, 0), regionMap)
}
}
```

A função *tryValues* faz a tentativa de cada um dos valores de 1 a *maxRegionSize* (valor que é definido pelo tamanho da região obtido através do hashmap de valores) para uma célula do tabuleiro.

```scala
/**
 * Tenta colocar valores válidos na posição atual.
 * Input: Recebe a matriz de valores, de regiões, a linha e coluna atual e uma lista de valores a serem testados.
 * Output:
 * Caso a lista de valores a serem testados esteja vazia, retorna None.
 * Caso o valor possa ser inserido na posição atual, retorna a matriz de valores com o valor inserido.
 * Caso o valor não possa ser inserido na posição atual, chama a função recursivamente com a lista de valores
 * restantes.
 */
@tailrec
private def tryValues(
  row: Int,
  col: Int,
  valuesToTry: List[Int],
  valueGrid: Matrix[Int],
  regionGrid: Matrix[String],
  regionMap: mutable.HashMap[String, List[(Int, Int)]]
): Option[Matrix[Int]] = {
  if (valuesToTry.isEmpty) {
    None
  } else {
    val value = valuesToTry.head
    if (canInsertValue((row, col), value, valueGrid, regionGrid, regionMap)) {
      val updatedValueGrid = valueGrid.setMatrixValue((row, col), value)
      val nextPosition = if (col == valueGrid.numCols - 1) (row + 1, 0) else (row, col + 1)
      solveKojun(updatedValueGrid, regionGrid, nextPosition, regionMap) match {
        case Some(solution) => Some(solution)
        case None => tryValues(row, col, valuesToTry.tail, valueGrid, regionGrid, regionMap)
      }
    } else {
      tryValues(row, col, valuesToTry.tail, valueGrid, regionGrid, regionMap)
    }
  }
}
```

A validação se um valor é válido ou não em uma célula do tabuleiro é feita por *canInsertValue*. 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.

```scala
private def canInsertValue(
  position: Position,
  value: Int,
  valueGrid: Matrix[Int],
  regionGrid: Matrix[String],
  regionMap: mutable.HashMap[String, List[(Int, Int)]]
): Boolean = {
  val (row, col) = position
  val regionValue = regionGrid.getMatrixValue(position)
  val regionPositions = regionMap(regionValue)

  val adjacentValues = List(
    (row - 1, col),
    (row + 1, col),
    (row, col - 1),
    (row, col + 1)
  ).filter(valueGrid.isValidPosition).map(valueGrid.getMatrixValue)

  val isTopValid = if (valueGrid.isValidPosition((row - 1, col)) && regionGrid.getMatrixValue((row - 1, col)) == regionValue) {
    value < valueGrid.getMatrixValue((row - 1, col))
  } else true

  val isBottomValid = if (valueGrid.isValidPosition((row + 1, col)) && regionGrid.getMatrixValue((row + 1, col)) == regionValue) {
    value > valueGrid.getMatrixValue((row + 1, col))
  } else true

  !(
    valueGrid.getMatrixValue(position) != 0 ||
    value < 1 || value > regionPositions.length ||
    adjacentValues.contains(value) ||
    regionPositions.map(valueGrid.getMatrixValue).contains(value) ||
    !isTopValid ||
    !isBottomValid
  )
}
```

Entrada e saída

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

```
def main(args: Array[String]): Unit = {
  // 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)
  val filePath = "src/inputs/10x10/kojun_10.txt" // Relative path to the file
```

O resultado da execução do programa será impresso no terminal como segue:

```
Value matrix:
5 0 2 0 2 0 3 1 3 1
0 4 0 1 0 5 0 5 0 4
7 5 1 7 0 0 3 1 3 0
0 4 0 0 0 0 0 0 0 3
2 0 3 4 0 2 0 0 4 0
5 0 2 0 6 0 0 0 0 0
0 1 3 0 1 0 0 4 0 3
6 7 0 3 0 1 4 0 0 1
4 0 3 0 4 0 0 0 0 3
0 1 0 2 0 6 2 0 2 1

Region matrix:
a b b b c c c c d d
a a a b e e f f d f
g g a e e h i f f f
g g e e j h i i i k
g g g e j j l k k k
m m n n n j o o p p
m m m n n q r s p p
t t m n q q r s u u
t t v v v v r s s w
t t t v v v r r w w

Solution:
5 3 2 4 2 4 3 1 3 1
2 4 3 1 3 5 6 5 2 4
7 5 1 7 1 2 3 1 3 2
6 4 2 6 4 1 2 4 1 3
2 1 3 4 3 2 1 2 4 1
5 6 2 5 6 1 2 1 2 4
4 1 3 4 1 3 5 4 1 3
6 7 2 3 2 1 4 3 2 1
4 2 3 5 4 7 3 2 1 3
3 1 5 2 1 6 2 1 2 1
```

Diferenças e vantagens do uso de Scala em relação ao uso de Haskell

Uma das principais vantagens da escolha do Scala foi a disponibilidade de diversos módulos de utilidades pré-implementados nas bibliotecas padrões. Além disso, Scala possui funcionalidades de linguagens mais modernas quando comparada ao Haskell. A sintaxe da linguagem também evidencia esse fator.

O uso das listas em Scala se mostrou muito mais simples do que o uso da estrutura equivalente em Haskell. Isso gerou uma das principais diferenças entre a implementação realizada e a implementação em Haskell: foi possível abstrair algumas das complexidades da manipulação de listas utilizando os módulos List e ListBuffer da biblioteca padrão da Scala.

Dificuldades encontradas

A maior dificuldade encontrada foi a adaptação do funcionamento das guardas que foram utilizadas na versão em Haskell para um estrutura condicional em Scala e na garantia do funcionamento da lógica de backtracking. Isso foi feito com o uso de estruturas condicionais *if* e *else* mas demandou algumas tentativas até que a lógica tivesse sido convertida corretamente.