UNIVERSIDADE FEDERAL
DE SANTA CATARINA

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIA DA COMPUTAÇÃO
PARADIGMAS DE PROGRAMAÇÃO

Alan Djon Lüdke

Daniela Heckler Mello

Ricardo Giuliani

TRABALHO II - PROGRAMAÇÃO FUNCIONAL - SCALA

FLORIANÓPOLIS

2020

1 INTRODUÇÃO

Str8ts é um jogo de lógica inventado por Jeff Widderich, em 2008. O jogo apresenta algumas propriedades e regras semelhantes ao Sudoku, e tem seu nome derivado do poker straight (Wikipedia). *Straight*, em tradução livre, é definido como um conjunto de números que se apresentam em sequência.

1.1 REGRAS[1]

O jogo consiste em uma matriz de grandeza NxN, no qual linhas e colunas são divididos em quadrados ou células que podem ser brancos ou pretos (Figura 1).

O jogador deve completar os quadrados brancos vazios com valores entre 1 e N, observando as seguintes normas:

- os números em uma linha e uma coluna devem completar um *straight*, isto é, deve formar um conjunto de números sem espaços vazios e em qualquer ordem, mas que se ordenados formam uma sequência, por exemplo,
    - [7]-[6]-[4]-[5] forma um *straight*, porém [1]-[3]-[8]-[7] não;
- não pode haver números repetidos em uma linha e uma coluna;
- números em quadrados pretos não fazem parte do *straight*, e significam que não podem ser usados para formar um.
- as células pretas servem para delimitar as sequências independentes de células brancas, onde os valores deverão ser inseridos para formar um *straight*.

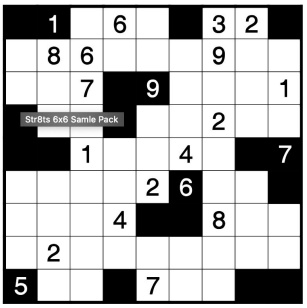

Figura 1. Exemplo de puzzle 9x9 do Str8ts.

[1] Disponível em: <https://www.str8ts.com/STR8TS_9x9_Sample_Pack.pdf>. Acesso em: 9 nov. 2020.

1.2 OBJETIVOS DO TRABALHO

O presente relatório tem como objetivo apresentar um programa, desenvolvido em linguagem de programação Scala, para a disciplina de Paradigmas de Programação, cuja finalidade é solucionar *puzzles* de Str8ts.

2 SOLUÇÃO

2.1 Modelagem do Tabuleiro

O tabuleiro (NxN) de Str8ts foi modelado em um formato similar a uma matriz, conforme ilustrado na Figura 2. Este tabuleiro deve ser armazenado em um arquivo simples de texto que será lido pelo programa principal. Na seção de entrada de dados será detalhado como um jogador pode alterar ou adicionar novos tabuleiros.

```
01-03
10300
--405
30020
43-12
```

Figura 2. Exemplo de modelagem de tabuleiro 5x5.

Quando o programa entra em execução, o arquivo é lido pela função **readFile()** (Figura 3) que transforma o arquivo em uma estrutura de dados do tipo *array*, sendo atribuída a uma variável do tipo *val*. Em seguida, a função **parseBoard()** (Figura 4) recebe como parâmetro este array gerado e é a responsável por construir a matriz que representa o tabuleiro de Str8ts que será solucionado.

```
def readFile(filename: String): Array[String] = {
  Source.fromFile(filename).getLines.toArray
}
```

*Figura 3. Função **readFile**. Lê arquivo de texto, processando-o em uma estrutura de dados do tipo array.*

```
def parseBoard(boardInput: Array[String]): Unit = {
  var row = 0
  var col = 0
  for (line <- boardInput) {
    for (c<- line) {
      board(row)(col) = c.asDigit
      col += 1
    }
    row += 1
    col = 0
  }
}
```

*Figura 4. Função **parseBoard**. Processa o array em uma matriz, gerando o tabuleiro de Str8ts.*

2.2 Estratégia

A estratégia adotada para a solução do jogo foi a técnica de *backtracking*. Esta é uma técnica algorítmica utilizada para resolver problemas recursivamente, tentando construir uma solução de forma incremental, um passo de cada vez, removendo soluções que falhem em satisfazer as regras e restrições do jogo a qualquer momento.

A implementação do *backtracking* é feita pela função que foi nomeada como **solve()**, como ilustrado na Figura 6. A função **main()** (Figura 5) faz a chamada à **solve()** passando como parâmetros 0 (zero) para linha e 0 (zero) para coluna.

Essa função tenta recursivamente e por força bruta resolver o tabuleiro, com o auxílio de quatro funções: *next()*, *validate()*, *verifySequenceCol()* e *verifySequenceRow()*.

A função *solve()* verifica se o valor ocupado em uma determinada linha e coluna é igual a 0 (zero), isto é, verifica se a célula é um campo branco editável. Caso não seja, a função **next()**, ilustrada na Figura 7, é invocada, atuando no cômputo da linha e coluna da próxima célula e realizando uma nova chamada à *solve()*. Caso a verificação determine que o valor da célula é igual a 0 (zero), a função inicia um laço de repetição para tentar inserir um valor àquela célula entre 1 e a dimensão do tabuleiro. Para determinar se é possível inserir o número, a função

```
def main(args: Array[String]): Unit = {
  val board_source = readFile("input/board_"+size)
  val gaps_source = readFile("input/gaps_"+size)
  parseBoard(board_source)
  parseGaps(gaps_source)
  solve(0, 0)
}
```

*Figura 5. Função **main**. Responsável por ler os dados de entrada, processá-los em uma matriz conforme explicado na Modelagem do Tabuleiro e fazer a chamada ao método solve com argumentos 0(zero) tanto para linha quanto para coluna.*

```
def solve(row: Int, col: Int): Boolean = {
  if (row == size-1 && col == size-1 && boardSolved()) {
    if(verifySequenceCol() && verifySequenceRow()){
      display_board()
      return true
    }
  } else if (board(row)(col) != 0) {
    return next(row, col)
  } else {
    for (i <- 1 to size) {
      if (validate(row, col, i)) {
        board(row)(col) = i;
        if(next(row, col)) {
          return true
        }
        board(row)(col) = 0
      }
    }
  }
  false
}
```

*Figura 6 Função **solve**. É a função responsável por implementar a técnica de backtracking e resolver os tabuleiros de Str8ts.*

validate() é chamada. Esta função (Figura 8) verifica se o número a qual se quer inserir já se encontra presente na determinada linha e coluna. Em caso afirmativo, a função retorna *false* para respeitar a regra da não repetição de números, e uma nova tentativa deve ser feita, com um novo número.

```
def next(row: Int, col: Int): Boolean = {
  if (col >= (size-1) && (row >= (size-1))) {
    solve(row, col)
  } else if (col >= (size-1)) {
    solve(row + 1, 0)
  } else {
    solve(row, col + 1)
  }
}
```

*Figura 7. Função **next**. A função faz chamada à solve() com um novo valor para linha ou coluna.*

```
def validate(row: Int, col: Int, num: Int): Boolean = {
  for (i <- 0 until size) {
    if (board(row)(i) == num || board(i)(col) == num) {
      return false
    }
  }
  true
}
```

*Figura 8. Função **validate**. A função verifica se o número o qual se quer inserir já existe na linha e na coluna. Caso exista, retorna false; caso contrário, retorna true.*

Por fim, as funções **verifySequenceCol()** e **verifySequenceRow()** descritas na Figura 9, verificam se os números nas colunas e nas linhas, respectivamente, formam uma sequência, para satisfazer uma das regras do jogo. O funcionamento para ambas funções é semelhante, alterando apenas a direção o qual a iteração ocorre. Para cada linha ou coluna, dependendo da função, criam-se listas temporárias delimitadas pelas células pretas. À cada lista é chamada a função *isConsecutive* (Figura 10) que fará a determinação da existência ou não da sequência. O tabuleiro é resolvido quando o retorno destas funções é verdadeiro e completou-se o tabuleiro até a última célula, que é determinado pela função *boardSolved()*, Figura 11.

```
def verifySequenceCol(): Boolean = {
    var temp = new ListBuffer[String]()
    for (j <- 0 until (size)) {
      for (i <- 0 until (size)) {
        if (gaps(i)(j) == 1) {
            temp += (board(i)(j)).toString
            if (i == size-1) {
                val tempList = (temp.toList).sorted
                val seque = isConsecutive(tempList.map(_.toString.toInt).toArray)
                if (seque == false) {
                    return false
                }
                temp = new ListBuffer[String]()
            }
        } else {
            val tempList = (temp.toList).sorted
            if(tempList.size > 0){
                val seque = isConsecutive(tempList.map(_.toString.toInt).toArray)
                if (seque == false){
                    return false
                }
            }
            temp = new ListBuffer[String]()
        }
      }
    }
    return true
  }
```

```
  def verifySequenceRow(): Boolean = {
    var temp = new ListBuffer[String]()
    for (i <- 0 until (size)) {
      for (j <- 0 until (size)) {
        if (gaps(i)(j) == 1) {
            temp += (board(i)(j)).toString
            if (j == size-1) {
                val tempList = (temp.toList).sorted
                val seque = isConsecutive(tempList.map(_.toString.toInt).toArray)
                if (seque == false) {
                    return false
                }
                temp = new ListBuffer[String]()
            }
        } else {
            val tempList = (temp.toList).sorted
            if(tempList.size > 0){
                val seque = isConsecutive(tempList.map(_.toString.toInt).toArray)
                if (seque == false){
                    return false
                }
            }
            temp = new ListBuffer[String]()
        }
      }
    }
    return true
  }
```

*Figura 9. Funções **verifySequenceCol** e **verifySequenceRow**. Ambas funções particionam a matriz em listas temporárias para verificar se as colunas e a linhas formam sequências, através da chamada à função isConsecutive (Figura 9).*

```
def isConsecutive(tempList: Array[Int]): Boolean = {
  for (i <- 0 until (tempList.size - 1)) {
    if (tempList(i+1) != tempList(i)+1) {
      return false
    }
  }
  return true
}
```

*Figura 10. Função **isConsecutive**. A função verifica se os números de uma lista são consecutivos.*

```
def boardSolved(): Boolean = {
  for (i <- 0 until size) {
    for (j <- 0 until size) {
      if (board(i)(j) == 0) {
        return false
      }
    }
  }
  true
}
```

*Figura 11. Função **boardSolved**. Determina se o tabuleiro foi resolvido, verificando todos os valores da matriz.*

2.3 ENTRADA DE DADOS

A entrada de dados para execução do programa é feita via arquivo simples de texto. O jogador deve descrever o tabuleiro em um formato de matriz (Figura 12a), dispondo os valores de uma mesma linha sem espaços ou tabulações e os valores de linhas diferentes.

A elaboração da matriz deve seguir a convenção adotada, no qual o 0 (zero) representa um quadrado ou célula branca e, portanto, um campo editável, o qual o programa irá alterar para a solução encontrada. O símbolo de hífen (-) é utilizado para sinalizar que o quadrado é do tipo preto e não apresenta nenhum valor, logo não é um campo editável.

O usuário deve construir uma matriz auxiliar (Figura 12b), similar ao tabuleiro, em um segundo arquivo, contendo apenas valores de 0 e 1, que representa as cores dos quadrados do tabuleiro, isto é, se é branco (1) ou preto (0). A utilização desta matriz auxiliar é importante para quadrados pretos que apresentam números. Como mostrado na seção de regras, este quadrado elimina o número como possível solução para aquela linha e coluna, e não faz parte do *straight*.

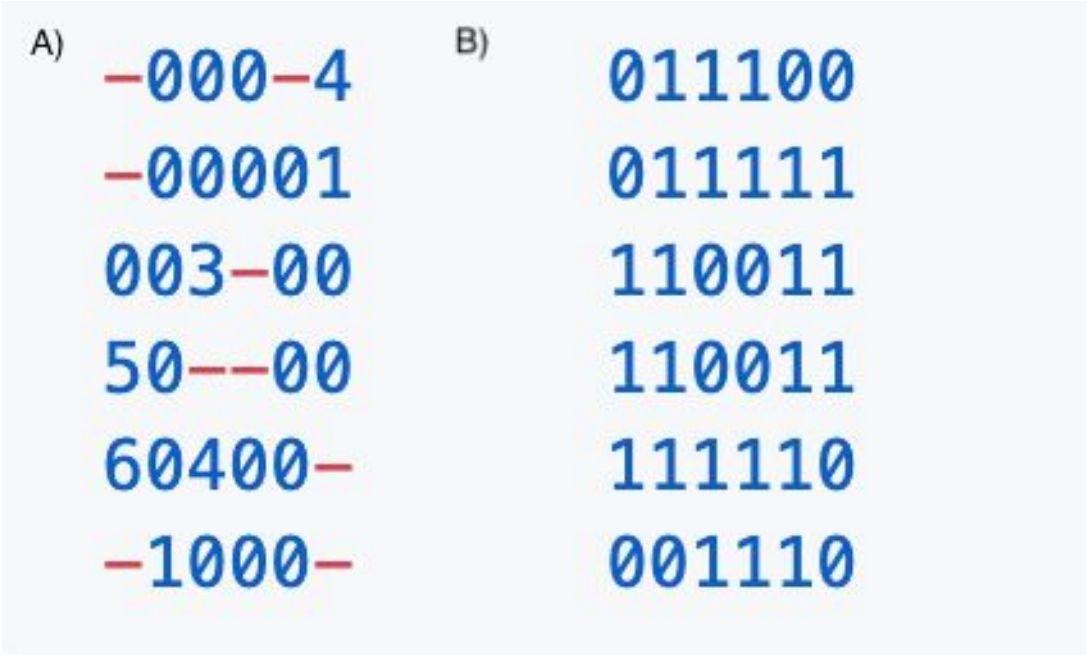

Figura 12. Exemplo de tabuleiro de Str8ts, de ordem 6x6. Em A) tabuleiro principal e em B) tabuleiro auxiliar, que representa as cores dos quadrados do tabuleiro.

O usuário deve observar que como o programa pode solucionar tabuleiros de tamanhos diversos, antes de iniciá-lo, deve verificar se a dimensão do tabuleiro foi atribuída corretamente a variável *size* no início do código, fazendo as mudanças necessárias caso não esteja.

Sugere-se que o jogador nomeie os arquivos da seguinte forma: para o tabuleiro principal "**board_X**" e para o tabuleiro auxiliar "**gaps_X**", no qual **X** deve ser substituído pelo número que representa a dimensão do tabuleiro. Por exemplo, se o tabuleiro é de dimensão 9x9, os arquivos nomeados são *board_9* e *gaps_9*. Dessa forma, o usuário não precisa fazer outras alterações no código. Caso nomeie os arquivos com outros nomes, deve ter o cuidado de alterar o código.

2.4 RESULTADO

O resultado obtido após a execução do programa é mostrado no terminal do usuário (Figura 13). As células que foram sinalizadas com o hífen (-) para determinar células pretas do tabuleiro são eliminadas e mostradas como um campo vazio. Com a ausência de uma interface gráfica que diferencia os quadrados brancos dos pretos, o jogador deve ter o cuidado de analisar o resultado, pois pode haver números presentes no resultado, que pertencem a células pretas e não fazem

parte do *straight*. Caso não haja solução para o *puzzle*, o programa não irá imprimir nada no terminal do usuário.

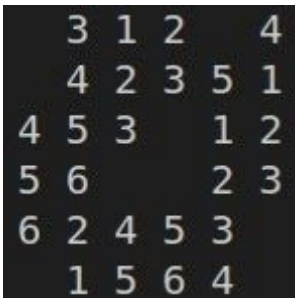

Figura 13. Exemplo de resultado do solucionador do jogo.

3 VANTAGENS/DESVANTAGENS EM RELAÇÃO A HASKELL

A linguagem de programação utilizada para a solução deste jogo foi o Scala. Esta linguagem provê certas vantagens em relação a Haskell, permitindo o uso de laços de iteração e provendo a construção de estrutura de dados, como matrizes, de forma mais facilitada.

A linguagem Scala apresenta grande similaridade com outras linguagens como Java e Python, as quais os integrantes do grupo são mais habituados. Dessa forma, conseguiu-se implementar o código mais facilmente.

Finalmente, o grupo já apresentava o entendimento das regras do jogo e do uso da técnica do backtracking, sendo terminantemente vantajoso a implementação em Scala quando comparado a Haskell.

4 ASPECTOS DE IMPLEMENTAÇÃO E ESTRATÉGIA QUE HOUVERAM MUDANÇAS

Este programa apresentou algumas mudanças em relação à implementação em Haskell.

A modelagem dos tabuleiros não é mais feita diretamente no código, sendo criado em arquivos de texto separados. Isto permite uma melhor organização e visualização do código.

Utilizou-se a estrutura de dados matriz nesta implementação. Anteriormente, havia sido utilizado um array de inteiros que era mapeado para simular matriz.

Finalmente, uma grande mudança foi a otimização do número de métodos utilizados, em que na linguagem Haskell utilizou-se mais em relação a Scala.

5 ORGANIZAÇÃO DO GRUPO

A elaboração do trabalho foi realizada em grande parte em tempo real utilizando o aplicativo Discord, facilitando a comunicação entre os membros do grupo e a visualização e o andamento do trabalho por meio do compartilhamento de telas.

O desenvolvimento do código do programa foi feito através do software Visual Studio Code, com a função de compartilhamento, no qual todos os membros puderam interagir, inserir e alterar o código simultaneamente.

Em outros momentos, a comunicação foi realizada através de aplicativo de troca de mensagens, o WhatsApp.

Em encontros síncronos, o grupo se reunia para discussão e o desenvolvimento do código do trabalho. Também era decidido, conjuntamente, a atribuição de algumas atividades para serem executadas assincronamente.

A elaboração deste relatório foi realizada pela plataforma Google Docs, que permitiu que todos os membros pudessem trabalhar e editar o documento ao mesmo tempo.

Por fim, para a apresentação do trabalho, optou-se pelo uso do Google Meet, com a opção de gravar reuniões.

6 DIFICULDADES

Algumas dificuldades foram contornadas na primeira resolução do jogo quando implementado com a linguagem Haskell, o que facilitou o desenvolvimento desta resolução com Scala.

Como sugestão de técnica de programação apresentada pela descrição do trabalho, o grupo buscou materiais na internet e vídeos explicativos sobre o *backtracking*.

Para auxiliar na implementação do código, buscou-se inspiração em códigos de resolvedores para o jogo Sudoku, jogo parecido com o proposto por este trabalho, encontrado no GitHub.

A parte que o grupo apresentou maior dificuldade foi na implementação da lógica para a verificação da existência ou não de sequência dos números nas colunas e nas linhas. Contudo, após o estudo da linguagem, o grupo conseguiu implementar a lógica de forma correta.

7 CONSIDERAÇÕES FINAIS

O código do programa desenvolvido pode ser obtido no repositório GitHub acessando o link https://github.com/alanludke/str8ts_scala.

O vídeo de apresentação do trabalho pode ser visualizado acessando o link: https://drive.google.com/file/d/1xenjWkS83UXpKAobK93RIIo7keZ6PhcZ/view?usp=sharing

REFERÊNCIAS BIBLIOGRÁFICAS

STR8TS, disponível em: <https://en.wikipedia.org/wiki/Str8ts>. Acesso em: 9 nov 2020