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 I - PROGRAMAÇÃO FUNCIONAL - HASKELL

RESOLVEDOR *PUZZLE STR8TS*

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*, neste jogo, 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, em que os valores deverão ser inseridos para formar um *straight*.

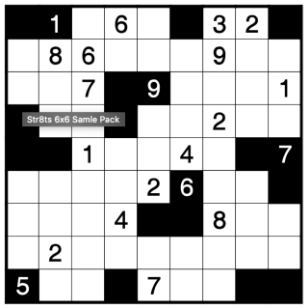

Figura 1. Exemplo de puzzle 9x9 do Str8ts.

[1] Disponível em: <https://www.str8ts.com/STR8TS_9x9_Sample_Pack.pdf>. Acesso em: 28 set. 2020.

1.2 Objetivos do Trabalho

O presente relatório tem como objetivo apresentar um programa, desenvolvido em linguagem de programação Haskell, 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 na forma de um vetor de inteiros. Para fins de demonstração do funcionamento do programa, alguns tabuleiros são pré-definidos, como exemplo de um tabuleiro 6x6 ilustrado na Figura 2, podendo o jogador alterá-los, conforme será detalhado na seção de entrada de dados.

```
str8tsBoard_6 :: [Int]
str8tsBoard_6 = [-1, 0, 1, 0, -1, 4,
                 -1, 0, 0, 3, 0, 1,
                 0, 5, 3, -1, 0, 0,
                 5, 0, -1, -1, 0, 3,
                 6, 0, 4, 0, 0, -1,
                 -1, 1, 0, 6, 0, -1]
```

Figura 2. Exemplo de modelagem de tabuleiro 6x6.

A seguir, implementou-se a função **indexToCoord** (Figura 3) que calcula para cada índice do vetor uma representação em coordenadas *x* e *y*, com o intuito de representar o vetor em forma de uma matriz.

```
indexToCoord :: Int -> Int -> (Int, Int)
indexToCoord i l= (calcX i l, calcY i l)
  where calcX i l  = i - l * (i `div` l)
        calcY i l  = i `div` l
```

*Figura 3. Função **indexToCoord** que representa o vetor em forma de matriz*

2.2 Entrada de Dados

O usuário realiza a entrada de dados diretamente pelo código do programa. Inicialmente, alguns tabuleiros de diferentes tamanhos foram pré-definidos, a título de demonstração (Figura 4a). O jogador pode alterar os valores, criando novos tabuleiros, em qualquer tamanho desejado NxN, desde que siga o formato dos tabuleiros exemplificados no código e as convenções adotadas durante o desenvolvimento do programa.

a)
```
str8tsBoard_8 :: [Int]
str8tsBoard_8 = [3, 4, 0, 2, -1, 8, 5, 0,
                 0, 1, 2, 0, 6, 0, 0, 0,
                 1, 0, 0, 4, 0, 6, 0, 0,
                 0, 3, -1, -1, 5, 0, 0, 7,
                 -1, 0, 0, 7, -1, 3, 2, 1,
                 5, 7, 4, 0, -1, -1, 0, 0,
                 0, 0, 7, -1, 0, 2, 1, 0,
                 8, 0, 0, -1, 2, 0, 4, 3]
```

b)
```
gapsBoard_8 :: [Int]
gapsBoard_8 = [1, 1, 1, 1, 0, 0, 1, 1,
               1, 1, 1, 1, 1, 1, 1, 1,
               1, 1, 1, 0, 1, 1, 1, 1,
               1, 1, 0, 0, 1, 1, 1, 1,
               0, 1, 1, 1, 0, 1, 1, 1,
               1, 1, 1, 1, 0, 0, 1, 1,
               1, 1, 1, 0, 1, 1, 1, 1,
               0, 1, 1, 0, 1, 1, 1, 1]
```

*Figura 4. Exemplo de tabuleiro 8x8. Em **a)** o jogador deve inserir os valores inicialmente pré-estabelecidos pelo puzzle de Str8ts, observando as convenções adotadas: 0 (zero) para quadrado branco vazio e -1 para quadrado preto vazio; em **b)** o jogador deve inserir valores 0 (zero) para quadrados pretos e 1 para quadrados brancos, referentes à matriz adota em **a)**.*

Adotou-se como convenção o **0** (zero) para sinalizar que o quadrado ou a célula respectiva é do tipo branca e editável e que, portanto, o programa deve alterar para a solução encontrada.

O número **-1** 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 notar a existência de uma matriz auxiliar (Figura 4b), denominada de *gapsBoard,* contendo apenas valores de 0 e 1, que representa as cores dos quadrados do tabuleiro, isto é, se é branco (1) ou preto (0). Importante destacar que esta matriz também deve ser alterada, para o funcionamento correto do programa. 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*.

O resultado obtido após a execução do programa é mostrado no terminal do usuário (Figura 5). Cabe ressaltar que a solução do jogo pode apresentar valores **-1** que deverá ser interpretado como quadrados pretos. 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 diferentes de -1, que pertencem a células pretas e não fazem parte do *straight*.

```
*Str8ts> table (solveIt str8tsBoard_6 gapsBoard_6)
-1      3       1       2       -1      4
-1      4       2       3       5       1
4       5       3       -1      1       2
5       6       -1      -1      2       3
6       2       4       5       3       -1
-1      1       5       6       4       -1
*Str8ts>
```

Figura 5. Exemplo de saída do resultado esperado para um Tabuleiro 6x6. Note que esta é a solução do tabuleiro exemplificado na Figura 2.

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

2.3 Estratégia

A estratégia adotada para 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. Esta função tenta recursivamente e por força bruta resolver o tabuleiro, que é passado como argumento ao parâmetro *s*, iniciando no índice *p*, com o conjunto de possíveis soluções para este índice *p* passado ao parâmetro h. Uma segunda matriz, denominada de *gapsBoard* é passada como argumento ao parâmetro *g*.

```haskell
solve :: Int -> [Int] -> [Int] -> [Int] -> [Int]
solve p s g h  | h == [] = []
               | (p == ((boardSize s) - 1)) && (tail h /= []) = []
               | (p == ((boardSize s) - 1)) && (tail h == []) = tryWith ((boardSize s)-1) s g (head h)
               | solvedNext == [] = solve p s g (tail h)
               | otherwise        = solvedNext
  where solveNext p s g = solve (nextBlank p s g) s g (solutionsAt (nextBlank p s g) s)
        solvedNext    = solveNext p (tryWith p s g (head h)) g
```

*Figura 6. Função **solve** que implementa a técnica de backtracking responsável por resolver o puzzle Str8ts*

Esta função requer para seu funcionamento o uso de algumas funções auxiliares, como a *solutionsAt, nextBlank* e *tryWith*.

A função **solutionsAt** (Figura 7) é a responsável por determinar uma lista de possíveis valores de solução para um determinado índice. O seu funcionamento consiste em remover números já presentes na linha e na coluna deste índice. Para isto, outros três métodos são utilizados (Figura 8): **columnAt** e **rowAt** que retornam, respectivamente, os valores da coluna e da linha deste índice, e que são concatenados para formar um vetor com todos os números presentes; e **remove** que elimina números presentes nesta lista concatenada anteriormente a partir de outra que contém números no intervalo de 1 a N (que representa a dimensão da matriz).

```haskell
solutionsAt :: Int -> [Int] -> [Int]
solutionsAt p s | p > length s  = []
                | ((s !! p) == 0) = [1..(lineWidth s)] `remove` (columnAt p s ++ rowAt p s)
                | otherwise     = [s !! p]
```

*Figura 7. Função **solutionAt**. Responsável por encontrar os possíveis valores de solução para um determinado índice da matriz.*

```haskell
columnAt :: Int -> [Int] -> [Int]
columnAt p s = helperColumnAt (indexToCoord p (lineWidth s)) s
  where helperColumnAt (x, _) s = map (\y -> s !! coordToIndex (x, y) (lineWidth s)) [0..((lineWidth s)-1)]

rowAt :: Int -> [Int] -> [Int]
rowAt p s = helperRowAt (indexToCoord p (lineWidth s)) s
  where helperRowAt (_, y) s = map (\x -> s !! coordToIndex (x, y) (lineWidth s)) [0..((lineWidth s)-1)]

remove :: [Int] -> [Int] -> [Int]
remove [] _       = []
remove xs []      = xs
remove xs (y:ys)  = remove (removeAll y xs) ys
```

*Figura 8. **columAt** e **rowAt**. Funções que retornam a lista contendo os números da coluna e da linha de um determinado índice, respectivamente. **remove.** Função que remove valores presentes na linha e na coluna evitando a repetição.*

A função **nextBlank** (Figura 9) procura o próximo quadrado branco sem valor, isto é, aquele que apresenta o valor 0 editável, a partir do índice *p* passado como parâmetro.

```
nextBlank :: Int -> [Int] -> [Int] -> Int
nextBlank p s g | p == ((boardSize s)-1)           = ((boardSize s)-1)
                | (s !! (p + 1) == 0) && (g !! (p+1) == 1) = p + 1
                | otherwise          = nextBlank (p + 1) s g
```

*Figura 9. **nextBlank**. Função que procura o próximo quadrado editável (branco, com valor 0).*

O retorno destas duas últimas funções citadas é passado como parâmetro para **solveNext** que invoca a função **solve**. Note que, a função **solve** será chamada com um novo índice *p*. Assim, pode-se observar a recursividade da função, uma vez que quando **solve** é chamada com um índice *p*, o próximo índice *p* + 1 também chamará a função.

Por fim, a função **tryWith** (Figura 10) é usada para gerar uma nova versão do tabuleiro, passado como parâmetro *s*, no qual tenta-se inserir um novo valor *x* em um índice *p* específico. O valor no qual se tenta inserir no tabuleiro provém da lista de possíveis soluções retornadas pela função **solutionsAt.** Quando o valor é inserido, a lista gerada por **solutionsAt** para o índice *p* + *1* é atualizada. Novamente tenta-se inserir um novo valor, observando as regras do jogo. Quando não for possível inserir um número sem infringir as regras, o algoritmo volta atrás, fazendo tentativas com novos números.

```
tryWith :: Int -> [Int] -> [Int] -> Int -> [Int]
tryWith p s g x | (p == ((boardSize s) - 1)) && normalParse && transposedParse = take ((boardSize s) - 1) s ++ [x] ++ drop (boardSize s) s
                | (p /= ((boardSize s) - 1))                                   = take p s ++ [x] ++ drop (p + 1) s
                | otherwise                                 = []
  where normalParse =  ((parseList (splitManySizes (sizes (splitIn0and1 g)) s ) (splitIn0and1 g)) == True)
        transposedParse = ((parseList (splitManySizes (sizes (splitIn0and1Transposed g)) (joinTranspose (splitIn0and1Transposed s))) (splitIn0and1Transposed g)) == True)
```

*Figura 10. **tryWith**.*Função que gera nova versão de tabuleiro com valor novo *x* inserido.

3 ORGANIZAÇÃO DO GRUPO

A elaboração do trabalho foi realizada em grande parte em tempo real utilizando a plataforma Google Meet, facilitando a comunicação entre os membros do

grupo e a visualização e o andamento do trabalho por meio do compartilhamento de telas.

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.

4 DIFICULDADES

A dificuldade encontrada para a implementação do código residiu majoritariamente na linguagem de programação adotada. Para todos os integrantes do grupo, Haskell não era de conhecimento prévio, sendo o primeiro contato com esta linguagem na disciplina de Paradigmas de Programação.

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

Inicialmente, o trabalho foi implementado em uma linguagem dominada por todos os integrantes, o Python, com o intuito de entender como funciona na prática o backtracking e como implementar a lógica do jogo Str8ts, observando suas regras e restrições.

Para auxiliar a implementação do código em Haskell, buscou-se inspiração em códigos de resolvedores para o jogo Sudoku, jogo parecido com o proposto por este trabalho, encontrados no GitHub. Desta forma, conseguiu-se compreender melhor a técnica de *backtracking* o que permitiu implementarmos o resolvedor de *Str8ts* com maior facilidade.

5 CONSIDERAÇÕES FINAIS

O código do programa desenvolvido pode ser obtido no repositório GitHub acessando o link https://github.com/alanludke/str8ts.

O vídeo de apresentação do trabalho pode ser visualizado acessando o link https://drive.google.com/file/d/1Wob2mWoJnhw17k6kJAO0pFOJUXR3keUm/view?usp=sharing.

REFERÊNCIAS BIBLIOGRÁFICAS

STR8TS, disponível em: <https://en.wikipedia.org/wiki/Str8ts>. Acesso em: 28 set 2020