Aluno: Gilson Trombetta Magro
Matrícula: 18202192
Disciplina: INE5416 - Paradigmas de Programação

RELATÓRIO: TRABALHO I
Outubro de 2020

1 ANÁLISE DO PROBLEMA

O problema escolhido para o desenvolvimento deste trabalho foi o Straights (ou Str8s). Este puzzle é um jogo similar ao Sudoku, e é composto de um tabuleiro quadrado (usualmente de tamanho 6 x 6 ou 9 x 9) que possui casas brancas e negras. O tabuleiro deve ser preenchido de acordo com as seguintes regras:

- Todas as casas brancas devem ser preenchidas com números de 1 a n (considerando um tabuleiro n x n), e nenhuma linha/coluna pode conter números repetidos;
- O tabuleiro pode ser separado em "regiões": uma região é composta apenas por casas brancas conectadas vertical ou horizontalmente e é delimitada por casas negras e/ou pelas bordas do tabuleiro;
- Toda região (vertical ou horizontal) deve conter um conjunto de números consecutivos, porém não necessariamente em ordem;
- Um número em uma casa negra serve para remover a possibilidade de uso daquele número em sua respectiva linha/coluna;
- Casas negras não fazem parte de nenhuma região.

2 SOLUÇÃO PROPOSTA

2.1 ORGANIZAÇÃO

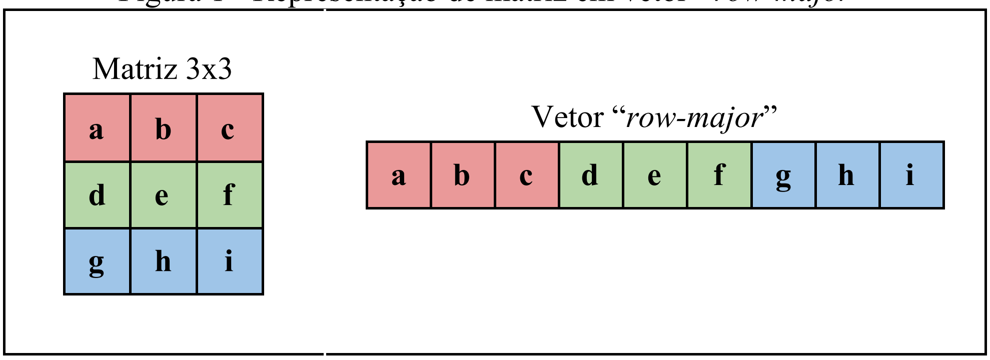
O código é organizado em quatro módulos: *Square*, *Board*, *Solver* e *Main*. O módulo *Square* é responsável por implementar as funções que manipulam as casas do tabuleiro. O módulo *Board* encapsula as funções que lidam com o tabuleiro em si. O módulo *Solver* encapsula as funções que realizam as computações necessárias para resolver de fato o problema. Já o módulo *Main* serve de entrada para o programa, e portanto cabe a ele interpretar a entrada e exibir a saída para o usuário.

2.2 REPRESENTAÇÃO

As casas (posições) do tabuleiro são representadas pelo tipo *Square*, que corresponde a uma tupla (*Int*, *Bool*) onde o primeiro valor representa o número da casa, e o segundo valor marca a cor da casa (*True* para casas brancas, *False* para casas negras). Para indicar que uma casa é vazia, define-se o primeiro valor da tupla como sendo zero.

Por outro lado, o tabuleiro é representado pelo tipo *Board* e equivale a uma lista de "*squares*", isto é, [*Square*]. Note que utiliza-se uma estrutura unidimensional para representar um tabuleiro bidimensional. Nesse sentido, convenciona-se o uso da ordenação *row-major*, em que as linhas do tabuleiro são "concatenadas" para formar o *Board*. Veja a figura 1.

Figura 1 - Representação de matriz em vetor *"row-major"*



Note que o acesso aos elementos do tabuleiro são completamente encapsulados pelas funções "readAt", "writeAt" e "emptyAt" do módulo *Board*. Essas funções, apresentadas abaixo, recebem uma tupla (y, x) como entrada, que corresponde à posição do elemento no tabuleiro. Portanto, na prática, o tabuleiro é de fato tratado como uma matriz bidimensional.

Board.hs

```haskell
-- | "readAt" retorna o Square na posição (y,x) do tabuleiro
readAt :: Board -> (Int, Int) -> Square
readAt b (y,x) = b !! (index b (y,x))

-- | "writeAt" "escreve" um valor na posição (y,x) e retorna um novo tabuleiro
writeAt :: Board -> (Int, Int) -> Int -> Board
```

```haskell
writeAt b (y,x) v = write' b (index b (y,x)) v
   where
      write' (a:b) i v
         | i == 0 = (setValue a v):b
         | otherwise = a:(write' b (i-1) v)

-- | "emptyAt" retorna verdadeiro se a posição (y,x) do tabuleiro for branca e vazia
emptyAt :: Board -> (Int, Int) -> Bool
emptyAt b (y,x) = isEmpty (readAt b (y,x))
```

2.3 SOLUÇÃO

A solução para o problema baseia-se na técnica de *backtracking*, conforme sugerido no enunciado. A código referente às funções *solve* e *_solve* é apresentado abaixo. Note que a função *solve* apenas encapsula a função *_solve*.

Solver.hs

```haskell
1  -- | "solve" apenas encapsula a função "_solve"
2  solve :: Board -> [Board]
3  solve b = _solve b (0,0) []
4  
5  _solve :: Board -> (Int, Int) -> [Board] -> [Board]
6  _solve b (y,x) l
7     | x == (side b)   = _solve b (y+1, 0) l
8     | y == (side b)   = b:l
9     | emptyAt b (y,x) = _solve' b (y,x) l (getAvailable b (y,x))
10    | otherwise       = _solve b (y,x+1) l
11    where
12       -- | "_solve' " lida com o backtracking
13       _solve' :: Board -> (Int, Int) -> [Board] -> [Int] -> [Board]
14       _solve' b (y,x) l []    = l
15       _solve' b (y,x) l (h:t) = do
16                                    l1 <- [_solve (writeAt b (y,x) h) (y,x+1) l]
17                                    _solve' b (y,x) l1 t
```

É a função *_solve* (e sua sub rotina *_solve'*) que implementa o passo a passo geral da solução, descrito a seguir:

1. Itera-se recursivamente sobre todas as posições do tabuleiro até encontrar uma posição (branca) vazia (corresponde às linhas 7 a 10);

2. Se uma posição (y0,x0) vazia é encontrada, utiliza-se a função *getAvailable* para construir uma lista com todos os possíveis valores para aquela posição;
    a. Se houver pelo menos um valor possível, então, para cada valor válido, “escreve-se” o valor na posição (y0,x0) e a função *_solve* é chamada novamente, agora com a posição (y0,x0) atualizada.
    b. Se não houver nenhum valor válido para a posição (y0,x0), então atingiu-se uma solução inválida, e é preciso fazer o backtracking, ou seja, desfazer o passo anterior e tentar novamente com outros valores.
3. Por outro lado, se nenhuma posição vazia for encontrada, então todas as posições já foram preenchidas, e encontramos uma solução válida. Portanto, adiciona-se o tabuleiro atual à lista de soluções válidas e o programa retorna essa mesma lista.

Ao término da solução, a função *_solve* retornará uma lista contendo todas as soluções válidas que podem ser derivadas da configuração inicial do tabuleiro de entrada. Se a lista retornada for vazia, então o tabuleiro inicial não possui nenhuma solução válida.

Note que, apesar do aspecto um pouco obscuro do backtracking, a estrutura da função *_solve* é relativamente simples. A peça que garante o corretude da solução está, na verdade, na função *getAvailable*. O objetivo dessa função é retornar uma lista de inteiros que são valores válidos para uma dada posição (y0,x0) do tabuleiro. Segundo as regras do puzzle, um número só pode ser colocado na posição (y0,x0) se ele respeitar as seguintes condições:

1. o número não deve ser repetido na linha y0;
2. o número não deve ser repetido na coluna x0;
3. na solução final, as regiões vertical e horizontal que contém a casa (y0,x0) devem formar, cada uma, um conjunto de números consecutivos;

Nesse sentido, é fácil notar que as condições 1 e 2 são triviais: para encontrar os números que, devido a essas duas condições, são excluídos da posição (y0,x0), basta que se observe a linha y0 e a coluna x0. Essa operação é realizada pelas funções *getRow* e *getCol*, conforme exibidas abaixo:

Solver.hs

```haskell
-- | "getRow" retorna os números presentes na linha (y,_)
```

```haskell
getRow :: Board -> Int -> [Int]
getRow b y = [value (readAt b (y,x')) | x' <- [0..((side b)-1)]]

-- | "getCol" retorna os números presentes na coluna (_,x)
getCol :: Board -> Int -> [Int]
getCol b x = [value (readAt b (y',x)) | y' <- [0..((side b)-1)]]
```

A condição 3, contudo, precisa ser divida em duas etapas: encontrar as casas que compõem uma região (seja ela vertical ou horizontal); e então encontrar os números válidos para tal região.

A primeira etapa é realizada pelas funções *top* e *bottom*, que encontram as bordas superior e inferior de uma região vertical; e as funções *left* e *right* que encontram as bordas esquerda e direita de uma região horizontal. Todas essas funções são muito similares, e portanto apenas o código da função *top* é apresentado a seguir:

Solver.hs

```haskell
top :: Board -> (Int, Int) -> (Int, Int)
top b (y,x)
    | isBlack (readAt b (y,x)) = (y+1,x)
    | y == 0                   = (y,x)
    | otherwise                = top b (y-1,x)
```

Desse modo, sabendo quais casas compoẽm uma região (vertical ou horizontal), é fácil realizar a segunda etapa: imagine uma região com n casas; sabe-se que essa região deverá conter os números consecutivos: m, m+1, m+2, …, m+n-1; portanto, conhecendo o valor mínimo nessa região, um limite superior m+n-1 pode ser estabelecido. Essa lógica é simétrica para estabelecer um limite inferior a partir do valor máximo da região.

Assim, após calcular os limites superior e inferior para os valores de uma região, a função *getColRegion* (ou *getRowRegion*, no caso de uma região horizontal) retorna uma lista dos inteiros inválidos para aquela região. A função *getColRegion* é apresentada abaixo:

Solver.hs

```haskell
getColRegion :: Board -> (Int, Int) -> [Int]
getColRegion b (y,x) = let {top' = top b (y,x); bottom' = bottom b (y,x)}
                       in getColRegion' b (minMax(inColRegion b top' bottom')) top' bottom'
```

```haskell
  where
    getColRegion' :: Board -> (Int, Int) -> (Int, Int) -> (Int, Int) -> [Int]
    getColRegion' _ (0,0) _ _                    = []
    getColRegion' b (m0,m1) (y0,_) (y1,_) = [1..(side b)] \\ [(m1-y1+y0)..(m0+y1-y0)]
```

Desta maneira, completa-se por fim a função *getAvailable*, que subtrai da lista [1..n] (onde n é o tamanho do lado do tabuleiro, obtido através da função *side* do módulo *Board*), os valores que são inválidos para uma determinada posição do tabuleiro. A função *getAvailable* é descrita a seguir:

Solver.hs

```haskell
getAvailable :: Board -> (Int, Int) -> [Int]
getAvailable b (y,x) = [1..(side b)] \\ ((getRow b y) ++ (getCol b x) ++ (getRowRegion b (y,x)) ++ (getColRegion b (y,x)))
```

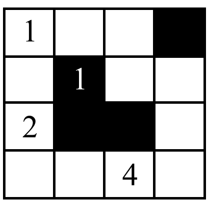
3 ENTRADA E SAÍDA

3.1 ENTRADA

A entrada para o programa se dá através de um arquivo de texto ou do próprio terminal. O formato de entrada consiste de um inteiro n >= 1 na primeira linha, que corresponde ao tamanho do lado tabuleiro n x n. As n linhas subsequentes deverão conter, cada uma, n números (∈ [0, n]) separados por espaços. Cada um desses números deve ser precedido de um "w", caso corresponda a uma casa branca, ou "b", caso corresponda a uma casa negra. O número zero corresponde a uma casa vazia. A figura 2 ilustra a entrada de um tabuleiro 4x4. Exemplos adicionais são fornecidos junto com o código-fonte, na pasta "examples".

Figura 2 - Exemplo de entrada do programa

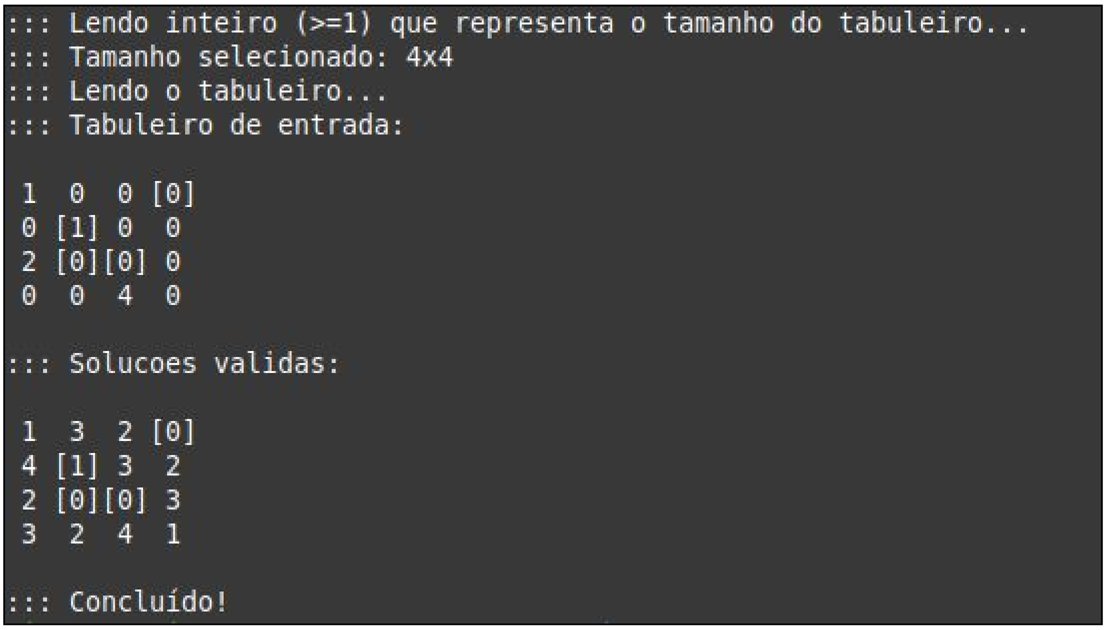
```
4
w1 w0 w0  b0
w0  b1 w0 w0
w2  b0  b0 w0
w0 w0 w4 w0
```



3.2 SAÍDA

Como saída, o programa exibirá o tabuleiro inicial e uma lista com todas as soluções possíveis, derivadas a partir do tabuleiro inicial. Se nenhuma solução for imprimida na tela, então o tabuleiro inicial não tem solução. A figura 3 ilustra a saída do programa, considerando a entrada apresentada na figura 2. Note que, ao imprimir os tabuleiros, casas negras são representadas por colchetes em volta do número. Novamente utiliza-se o número zero para representar casas vazias.

Figura 3 - Exemplo de saída do programa



```
::: Lendo inteiro (>=1) que representa o tamanho do tabuleiro...
::: Tamanho selecionado: 4x4
::: Lendo o tabuleiro...
::: Tabuleiro de entrada:

 1  0  0 [0]
 0 [1] 0  0
 2 [0][0] 0
 0  0  4  0

::: Solucoes validas:

 1  3  2 [0]
 4 [1] 3  2
 2 [0][0] 3
 3  2  4  1

::: Concluído!
```

4 DIFICULDADES ENCONTRADAS

Dentre as dificuldades encontradas, destacaram-se três. A primeira delas foi a minha falta de familiaridade com o paradigma funcional e com a própria linguagem Haskell, coisa que prolongou consideravelmente o tempo que levei para desenvolver o trabalho. A maneira que o Haskell lida com conversões de tipos e tipagem de funções me confundiu profundamente em diversos momentos. Contudo, aprender a utilizar o *ghci* para acompanhar a execução do código me ajudou bastante a localizar e corrigir problemas, especialmente nas funções que utilizam de recursão.

A segunda dificuldade que quero destacar tem a ver com o uso de *monads*. Eu ainda não conhecia monads quando comecei a desenvolver o trabalho, e a forma como eu pretendia implementar o backtracking dependia de uma execução sequencial. Em determinado momento, cheguei a utilizar o monad *Maybe* como resultado da função *solve*, de modo a poder representar soluções inválidas com o valor *Nothing*. Posteriormente, porém, percebi que esse monad deixou de ser conveniente no momento em que passei a retornar uma lista de soluções, e então decidi removê-lo.

A terceira e principal dificuldade encontrada foi certamente a implementação do *backtracking* em Haskell. Eu já conhecia essa técnica porque vi ela sendo usada em um vídeo de um canal no YouTube chamado Computerphile (COMPUTERPHILE, 2020). Nele, utilizava-se *backtracking* em Python para resolver um tabuleiro de Sudoku. Como tenho certa experiência com Python, decidi partir daquela solução e implementar um algoritmo que resolvesse um tabuleiro de Straights (também em Python). O problema, então, resumiu-se a traduzir o código para algo que fizesse sentido em Haskell.

Por último, precisei ainda pesquisar maneiras de implementar *backtracking* para Sudoku em Haskell, como sugere o enunciado do trabalho. Assim, foi de uma dessas implementações (disponível na Haskell Wiki e destacada nas referências), que surgiram as principais ideias para o resolvedor de Straights que desenvolvi.

5 REFERÊNCIAS

COMPUTERPHILE. **Python Sudoku Solver - Computerphile**. 2020. (10m52s). Disponível em: <https://youtu.be/G_UYXzGuqvM>. Acesso em: 07 out. 2020.

HASKELL WIKI. **Sudoku:** Simple solver. Disponível em: <https://wiki.haskell.org/Sudoku#Simple_solver>. Acesso em: 07 out. 2020.