Trabalho I – Resolvedor de Suguru em Haskell

Arthur Machado Capaverde – 17207215

Implementação

O código é dividido em 6 partes: a configuração da entrada (feita diretamente no código fonte), 3 testes, a busca da solução utilizando backtracking com os testes previamente implementados e a parte final da impressão do resultado na tela.

Como configurar a entrada

```haskell
-- O tabuleiro que queremos resolver
tabuleiro :: [Int]
tabuleiro = [0, 3, 0, 2, 0, 3,
             4, 0, 4, 0, 1, 0,
             0, 0, 0, 5, 0, 4,
             3, 5, 0, 0, 0, 0,
             0, 0, 0, 2, 0, 0,
             5, 2, 0, 0, 4, 0]

-- O tabuleiro com as areas definidas atraves de tuplas
tabuleiroDiv :: [(Int,Int)]
tabuleiroDiv = [(1,0), (1,3), (1,0), (2,2), (2,0), (2,3),
                (3,4), (4,0), (1,4), (5,0), (2,1), (2,0),
                (3,0), (4,0), (4,0), (5,5), (5,0), (5,4),
                (3,3), (3,5), (4,0), (6,0), (6,0), (5,0),
                (7,0), (3,0), (4,0), (6,2), (6,0), (8,0),
                (7,5), (7,2), (7,0), (7,0), (6,4), (8,0)]
```

```haskell
--n é a dimensao do tabuleiro(tabuleiro tera tamanho nxn)
n :: Int
n = 6
```

Os elementos do tabuleiro devem ser colocados na matriz *tabuleiro* e os elementos do tabuleiro junto com identificadores da área a que pertencem devem ser inseridos em forma de

tupla (identificador, elemento) na matriz *tabuleiroDiv*. A dimensão do tabuleiro também deve ser inserida em *n*.

Primeiro teste: lados e diagonais

```
--testa se o quadrado ao sudeste tem um número igual
seOk :: Int -> [Int] -> Bool
seOk i s = if i >= (n*(n-1)) || ((i+1) `mod` n) == 0 then True
  else if s!!(i+n+1) == 0 then True
    else if s!!i == s!!(i+n+1) then False
      else True

--testa se o quadrado ao sudoeste tem um número igual
swOk :: Int -> [Int] -> Bool
swOk i s = if i >= (n*(n-1)) || (i `mod` n) == 0 then True
  else if s!!(i+n-1) == 0 then True
    else if s!!i == s!!(i+n-1) then False
      else True

--testa se algum dos quadrados em volta tem um número igual
sidesOk :: Int -> [Int] -> Bool
sidesOk i s = (downOk i s) && (upOk i s) && (leftOk i s) && (ri
```

Este teste checa a regra de que o número inserido não pode ser igual a um número nos seus lados ou diagonais, cada função checa para uma direção específica e sidesOk utiliza todas elas para checar todas as direções.

Segundo teste: números na mesma área

```
--testa se duas tuplas são iguais e retorna 1 se sim e 0 se não
isEqual :: (Int, Int) -> (Int, Int) -> Int
isEqual (x, y) (x', y') = if x == x' && y == y' then 1
else 0

--aplica isEqual em todos os elementos de uma matriz
mapF :: (Int, Int) -> [(Int, Int)] -> [Int]
mapF (x', y') s = map (\(x, y) -> isEqual (x', y') (x, y)) s

--testa se tem dois números iguais na área do indice i
areaOk :: Int -> [(Int, Int)] -> Bool
areaOk i s = if (sum (mapF (s!!i) s)) == 2 then False
else True
```

Este teste checa a regra de que o número inserido não pode ser igual a um número na sua área/seção, aqui temos a primeira ocorrência de precisar invocar a função map com mais de um parâmetro, para isso foi criada a funçao lambda mapF, ela aplica isEqual em todos os elementos da matriz, isEqual retorna 1 caso os elementos testados sejam iguais e 0 caso não seja. areaOk vai somar todos os elementos da matriz retornada por mapF e caso a soma seja igual a 2 irá retornar False. O número 2 foi utilizado pois significa que mapF encontrou o próprio elemento sendo testado (+1) e um elemento igual a ele (+1=2).

Terceiro teste: tamanho da área

```
--testa se duas tuplas tem o mesmo primeiro elemento
isEqual' :: (Int, Int) -> (Int, Int) -> Int
isEqual' (x, y) (x', y') = if x == x' then 1
else 0

--aplica isEqual' em todos os elementos de uma matriz
mapS :: (Int, Int) -> [(Int, Int)] -> [Int]
mapS (x', y') s = map (\(x, y) -> isEqual' (x', y') (x, y)) s

--retorna quantos quadrados tem a área do indice i
areaSize :: Int -> [(Int, Int)] ->Int
areaSize i s = (sum (mapS (s!!i) s))

--testa se o elemento e é maior do que a quantidade de quadrados da área do indice i
possibilitiesOk :: Int -> Int -> [(Int, Int)] -> Bool
possibilitiesOk e i s = if e < ((areaSize i s) + 1) then True
else False
```

Este teste checa se o número inserido está no intervalo (1:N) onde N é o tamanho da área/seção do índice onde o número foi inserido. A função mapS é utilizada com isEqual' para retornar uma matriz com 1s nos indíces que pertencem a área/seção e 0s nos outros. AreaSize então soma todos os elementos desta matriz para encontrar N e possibilitiesOk checa se o número inserido é menor do que N, não é necessário testar se o número é maior do que 0 pois 0 não está na matriz *possibilities* declarada no começo do código que contém todos os elementos inseríveis [1, 2, 3, 4, 5, 6, 7, 8, 9].

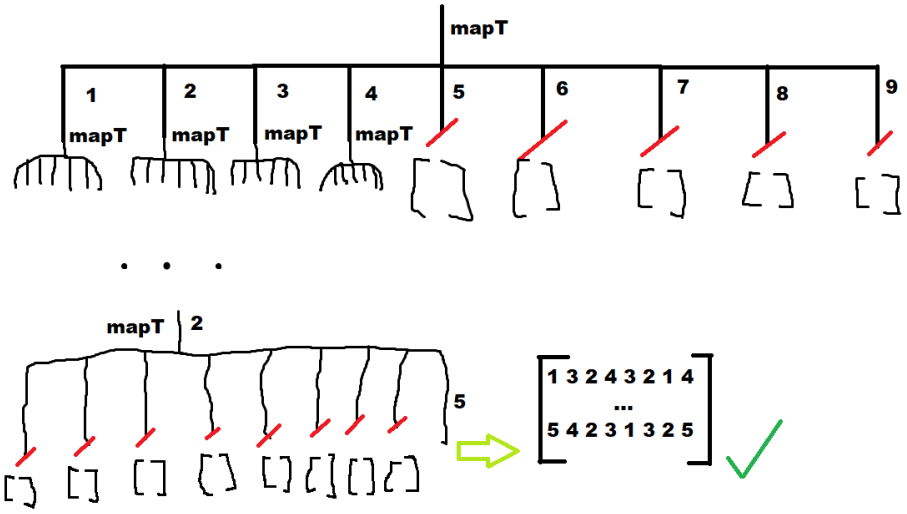
Encontrando a solução

```
--aplica todos os testes previamente implementados
tester :: Int -> [Int] -> [(Int, Int)] -> Bool
tester i s s' = (sidesOk i s) && (areaOk i s') && (possibilitiesOk (s!!i) i s')

--testa se o tabuleiro foi completamente preenchido
isEnd :: Int -> [Int] -> Bool
isEnd i s = if (i == f) && ((s!!i) /= 0) then True
else False
```

```haskell
--retorna um novo tabuleiro com o elemento x inserido no indice i
getTesterMap :: Int -> Int -> [Int] -> [Int]
getTesterMap x i s  = take i s ++ [x] ++ drop (i + 1) s

--retorna um novo tabuleiroDiv com o elemento x inserido no indice i
getTesterMap' :: Int -> Int -> [(Int, Int)] -> [(Int, Int)]
getTesterMap' x i s' = take i s' ++ [(fst(s'!!i), x)] ++ drop (i + 1) s'

--aplica a função solver com todos os elementos da matriz x(possibilities)
mapT :: [Int] -> Int -> [Int] -> [(Int, Int)] -> [[Int]]
mapT x i s s' = map (\x -> solver x i s s') x

--funcao recursiva que encontra a solução do suguru
solver :: Int -> Int -> [Int]-> [(Int, Int)] -> [Int]
solver x i s s' = if (tester i (getTesterMap x i s ) (getTesterMap' x i s')) then
  if (isEnd i (getTesterMap x i s)) then (getTesterMap x i s)
  else concat (mapT possibilities (nextBlank i s) (getTesterMap x i s) (getTesterMap' x i s'))
  else []
```

O algoritmo utilizado utiliza *backtracking* para encontra a solução, primeiro ele procura o primeiro espaço vazio da matriz (=0) e então insere sucessivamente os dígitos de 1 a 9 começando pelo 1. ele então testa se o elemento inserido viola alguma das regras do suguru (lados, área, etc.), caso viole ele retorna uma matriz vazia [].

Caso o elemento inserido passe no teste o algoritmo checa se o tabuleiro está completamente preenchido, se sim, então encontramos a solução e retornamos a matriz construída até agora. Se não, ele procura o próximo espaço vazio e recomeça a operação de inserir elementos. Segue um desenho para tentar ilustrar este processo:



mapT retorna uma lista de listas [[Int]] e solver retorna uma lista [Int], portanto, concatenamos todos os elementos da lista [[Int]] com *concat* para termos uma lista [Int] com todos os elementos das listas em [[Int]], porém, como a imagem ilustra, apenas quando encontramos a solução retornamos uma matriz que não esteja vazia, então ao usar concat recursivamente terminaremos apenas com a matriz solução, que é impressa na próxima seção.

Imprimindo a solução

```haskell
--insere um elemento c entre todos os elementos de uma matriz
joinWith :: a -> [a] -> [a]
joinWith _ (x:[])  = [x]
joinWith c (x:xs)  = x : c : joinWith c xs

--altera o resultado obtido para um formato mais legível
pPrint [] = []
pPrint s  = spaceOut s ++ pPrint (drop n s)
  where showS s    = concatMap show s
        space      = ' '
        newline    = "\n"
        spaceOut s = joinWith space (take n (showS s) ++ newline)

--comando para imprimir na tela a solução
solve = putStrLn (pPrint (concat (mapT possibilities 0 tabuleiro tabuleiroDiv)))
```

Aqui utilizamos a função joinWith para inserir um espaço entre cada elemento da matriz solução, a função pPrint (Pretty Print) transforma a matriz solução com os espaços em uma matriz de chars com as devidas quebras de linha. O comando solve invoca pPrint com putStrLn (necessário para que as quebras de linha sejam adequadamente processadas), pPrint é invocado com a função concat como parametro para que a lista de listas [[Int]] retornada por mapT seja transformada em apenas uma lista [Int]. MapT é invocado com os parametros possibilities (matriz com os dígitos inseríveis), 0 (índice do primeiro elemento), tabuleiro (tabuleiro inicial sem a divisão em áreas/seções) e tabuleiroDiv (tabuleiro inicial com a divisão em áreas/seções, implementada usando tuplas).

Conclusão

Este trabalho foi difícil mas poder inserir a entrada diretamente no código fonte facilitou bastante, a dica do professor de reutilizar o código do resolvedor de sudoku também ajudou, foram alteradas e reutilizadas a função nextBlank, que encontra o próximo 0 na matriz e as funções joinWith e pPrint que são utilizadas na formatação da solução para ser impressa na tela. A implementação dos testes não foi muito difícil, mas a implementação do resolvedor de suguru foi complicada, com vários mini-obstáculos (map só aceita uma função e um parametro, mapT e solver terem retornos de tipos diferentes, etc.) e com diversas tentativas até entender como implementar *backtracking*. Tentei explicar bem o código neste relatório mas também tentei ser sucinto, então caso algo não tenha ficado claro com o relatório e o vídeo não hesite em me perguntar por moodle ou e-mail.