Matheus Henrique Schaly – 18200436

14-05-2020

Straights- Haskell

1. Análise do problema

O jogo consiste em um tabuleiro com campos brancos e pretos. O tabuleiro é iniciado já com alguns campos preenchidos, possui dimensões NxN, e os campos são preenchidos com valores de 1 a N.

Para resolver o jogo, todos os campos brancos devem ser preenchidos, nenhum número pode ocorrer mais de uma vez em uma linha ou em uma coluna (independentemente da cor do campo). Além disso, todas as tiras de campos brancos em uma linha ou coluna (que podem ser interrompidas por campos pretos) precisam formar uma sequência de números sem interrupção, porém, tal sequência não precisa estar em ordem, por exemplo 2-3-4 ou 5-4-7-6.

2. Solução

A solução adotada para resolver o jogo foi:

1. Usar backtracking e recursividade para resolver o problema, começando pelo índice 0 e indo até o tamanho do tabuleiro.
2. Durante o backtracking, cada posição do tabuleiro (que não possui valor prévio e não é preto) recebe um conjunto de possibilidade de valores. Tais possibilidades são calculadas da seguinte forma:
    a. Os valores não podem se repetir nas colunas.
    b. Os valores não podem se repetir nas linhas.
    c. Os valores das tiras brancas devem formar uma sequência válida nas linhas.
    d. Os valores das tiras brancas devem formar uma sequência válida nas colunas.
3. Fazendo-se a diferença de conjunto entre todas as possibilidades e essas 4 listas, obtém-se os valores possíveis de serem preenchidos em um determinado campo.
4. Para obter possibilidades de sequências válidas para uma tira branca:
    a. Encontra-se a tira branca e seus valores.
    b. Encontra-se o valor máximo da tira (max).
    c. Encontra-se o valor mínimo da tira (min).
    d. Cria-se uma lista que varia de *max – length (tira branca) + 1* até *min + length (tira branca) - 1)*.

```
-- Recursively tries the board given in values, starting at p, using colors vector (colors)
-- and the size of the board (n) with the set of possible solution at that point 1
solve :: Int -> Int -> [Int] -> [Int] -> [Int] -> [Int]
solve n p values colors x  | (p == n) && null x        = values
                           | (p == n) && null (tail x) = tryWith n values (head x)
                           | (p == n)                  = []
                           | null x                    = []
                           | otherwise                 = case () of
                                                           () | null solvedNext -> solve n p values colors (tail x)
                                                              | otherwise       -> solvedNext
  where blankCell          = nextBlank n p values colors
        solvedNext         = solveNext p newTry
        newTry             = tryWith p values (head x)
        solveNext p values = solve n blankCell values colors (solutionsAt blankCell values colors)
```

```haskell
-- The list of solutions at the index (p) of value vector (values) and color vector (colors) 2
solutionsAt :: Int -> [Int] -> [Int] -> [Int]
solutionsAt p values colors | p > length values  = []                                    3
                            | (values !! p) == 0 = [1..n] `setDifference` (columnAt n p values ++ 2a
                                                                           rowAt n p values ++ 2b
                                                                           getSeqInPoss n p values colors 0 ++ 2c
                                                                           getSeqInPoss n p values colors 1) 2d
                            | otherwise          = [values !! p]
  where n = isqrt (length values)

-- Retrieves the possibilities of a white strip (strip) with a vector size (allPossLen)
-- and the strip length (sLen)
getStripPoss :: [Int] -> Int -> Int -> [Int]
getStripPoss strip allPossLen sLen | (length strip == 0) = [1..allPossLen]
                                   | otherwise           = [minSeq..maxSeq] 4d
  where max    = maximum strip 4b
        min    = minimum strip 4c
        minSeq = getMinSeq strip max sLen
        maxSeq = getMaxSeq strip min sLen allPossLen

-- Retrieves the sequence of invalid possibilities considering a white strip, using the size
-- of the board (allPossLen) position (p) values vector (values) colors vector (colors) and
-- analyzing a row when rOrC is 0 and column when rOrC is 1 4
getSeqInPoss :: Int -> Int -> [Int] -> [Int] -> Int -> [Int]
getSeqInPoss allPossLen p values colors rOrC = [1..allPossLen] `setDifference` sequence
  where rowSeqAt    = getRowStripAt p values colors rOrC 4a
        preSequence = getStripPoss (tail rowSeqAt) allPossLen (head rowSeqAt)
        sequence    | (head preSequence == 0)   = []
                    | otherwise                 = preSequence
```

3. Como usar o programa

O tabuleiro utiliza dois vetores. O primeiro contém os valores do tabuleiro, onde 0 significa ausência de valor. O segundo contém as cores da matriz, onde 1 simboliza um campo preto e 0 um campo branco.

Há cinco exemplos de tabuleiros já criados no código fonte. Para fornecer um novo input ao programa o usuário precisa criar ou alterar dois novos vetores diretamente no código fonte.

Para executar, por exemplo, a matriz 9x9 fornecida, primeiramente carregue o Haskell *ghci*, em seguida carregue o programa *:l straights.hs* e execute a algoritmo *solveIt v9 c9*. Onde v9 e c9 é o vetor de valores e o vetor de cores respectivamente.

O output pode ser visualizado a partir do software que chamou o programa, como pode ser visto abaixo:

```
C:\Users\Matheus_Schaly\Desktop\PP_UFSC\Haskell>ghci
GHCi, version 8.10.1: https://www.haskell.org/ghc/  :? for help
Prelude> :l straights.hs
[1 of 1] Compiling Main             ( straights.hs, interpreted )
Ok, one module loaded.
*Main> solveIt v9 c9
0 0 6 5 0 1 2 3 0
6 9 5 3 8 2 1 7 4
7 8 1 2 4 3 0 6 5
5 0 0 4 3 0 9 8 6
0 4 3 6 5 7 8 9 0
8 3 4 0 7 6 0 0 2
2 1 0 7 9 8 5 4 3
3 2 9 8 6 4 7 5 1
0 7 8 9 0 5 6 0 0

*Main> _
```

4. Dificuldades

As dificuldades e soluções foram:

1. Como programar em Haskell: Nunca havia programado em Haskell, mas a linguagem é similar ao Scheme e também possui algumas similaridades com o Prolog, então não foi difícil aprender rapidamente.
2. Como depurar o programa: Haskell é bem exigente no momento da compilação, entrando em bastante detalhes sobre onde o erro está ocorrendo. Entretanto, no momento da execução, retornava erros muito genéricos. Não encontrei uma forma de depurar o programa linha por linha, isso acabou atrasando o desenvolvimento do programa.

Não houve grandes dificuldades para criar o código, eu já havia criado uma solução para o problema tanto em Prolog quanto em Scheme. Acabei utilizando a mesma lógica que utilizei no Scheme, mas de uma maneira mais eficiente, funcional e limpa.