UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CIÊNCIA DA COMPUTAÇÃO

Artur Luiz Rizzato Toru Soda

Relatório - Trabalho 1 - Programação Funcional - Haskell
Kojum

Florianópolis
2024

1. INTRODUÇÃO

Este trabalho tem como objetivo aplicar os conhecimentos aprendidos na disciplina Paradigmas de Programação, criando um resolvedor para uns dos três puzzles mostrado no enunciado, na linguagem Haskell.

Dessa forma, o problema escolhido neste trabalho foi o Kojum, cujo o funcionamento é da seguinte forma: Dado um tabuleiro dividido em diversas regiões, um número deve-se ser inserido em cada campo, de modo que: Cada região de tamanho N contenha números de 1 à N exatamente uma vez; Campo adjacentes ortogonalmente devem ter valores diferentes; E por fim, se dois campo são verticalmente adjacentes na mesma região, o valor de cima deve ser maior que o de baixo.

O puzzle é resolvido quando o tabuleiro é completo, respeitando as três regras supracitadas.

2. DESENVOLVIMENTO

2.1. BACKTRACKING

Tendo noção das regras do puzzle, a estratégia utilizada para a resolução foi o backtracking, um algoritmo para resolução de problemas que envolve encontrar uma solução tentando diferentes opções e desfazendo-as se levarem a uma conclusão incorreta, comumente usado em situações em que é preciso explorar múltiplas possibilidades. Assim o algoritmo volta ao ponto de decisão anterior e explora um caminho diferente até que uma solução seja encontrada ou todas as possibilidades tenham sido esgotadas.

2.2. PYTHON

Como sugerido no enunciado, primeiramente, o desafio foi feito utilizando uma linguagem em que possuo um maior domínio, sendo assim python foi utilizado.

Dessa forma, sabendo a estratégia a ser utilizada, esse passo do desenvolvimento auxiliou para saber como o problema deveria ser modelado, facilitando então, na implementação na linguagem Haskell.

2.3. HASKELL - SOLUÇÃO

No começo de sua execução, o programa recebe como parâmetros qual tabuleiro deve ser resolvido, carregando então a matriz de valores e de regiões.

```haskell
-- Le matrizes e retorna tabuleiro com valores e tabuleiro de regioes
readBoard :: String -> IO ([[Int]], [[Char]])
readBoard path = do
    valuesContent <- readFile ("Entradas/" ++ path ++ "/numbers.txt")
    let vContWithoutSpace = Prelude.map (Prelude.filter isNotSpace) (lines valuesContent)
    let valuesMatriz = Prelude.map (Prelude.map charToInt) vContWithoutSpace

    regionsContent <- readFile ("Entradas/" ++ path ++ "/regions.txt")
    let regionsMatriz = Prelude.map (Prelude.filter isNotSpace) (lines regionsContent)

    return (valuesMatriz, regionsMatriz)

-- Indica se caracter nao eh espaco
isNotSpace :: Char -> Bool
isNotSpace a = a /= ' '

-- Transforma caracter em inteiro
charToInt :: Char -> Int
charToInt a | a == '*' = -1
            | otherwise = digitToInt a
```

Em seguida o tabuleiro é montado, onde cada campo é representado por uma *Posição*: Uma estrutura que guarda informações como, seu valor, bloqueio de cima, baixo, esquerda e direita, e a região em que pertence.

```haskell
-- Estrutura das posicoes do tabuleiro
data Position = Position {
    value :: Int,
    upBorderBlock :: Bool,
    downBorderBlock :: Bool,
    leftBorderBlock :: Bool,
    rightBorderBlock :: Bool,
    region :: Char
} deriving (Show, Eq)

-- Instancia posicao, definindo seu valor, bloqueios e regiao
makePosition :: Int -> Int -> Int -> [[Char]] -> Int -> Position
makePosition i j v rm l = let r = (rm!!i)!!j in
    Position {
        value = v,
        upBorderBlock = if (i == 0) then True else ((rm!!(i-1)!!j) /= r),
        downBorderBlock = if (i == (l-1)) then True else ((rm!!(i+1)!!j) /= r),
        leftBorderBlock = if (j == 0) then True else ((rm!!i)!!(j-1) /= r),
        rightBorderBlock = if (j == (l-1)) then True else ((rm!!i)!!(j+1) /= r),
        region = r
    }
```

Em seguida, com o tabuleiro inicial montado, a função *solve* é chamada, que recebendo o tabuleiro como parâmetro, é retornado um *Maybe Board*, ou seja, caso não for possível resolver, a função retorna *Nothing*, mas caso contrário, o tabuleiro resolvido é retornado. Assim tendo conhecimento do backtracking, sua lógica é da seguinte forma: a função percorre por todos os campos do tabuleiro, e nas casas em que ainda não possuem um valor, um número candidato é atribuído na célula, e então é feita a verificação se a atribuição foi válida. Caso seja, passamos para o próximo campo, se não, tentamos atribuir outro valor ao campo, e se caso as opções de candidatos acabarem, voltamos uma casa para trás.

Dessa forma, esses passos são realizados, até que o tabuleiro seja resolvido, ou até que todas as possibilidades acabem.

```haskell
-- Logica para resolucao do tabuleiro
solveAux :: I -> J -> Int -> Board -> Int -> Map Char [Int] -> Maybe Board
solveAux i j guess board n regionDict
    -- Caso passe da ultima linha -> Problema resolvido
    | (i == n) = Just board
    -- Caso passe da ultima coluna -> Passa para a proxima linha
    | (j == n) = solveAux (i+1) 0 guess board n regionDict
    -- Caso posicao ja possui um valor -> Passe para o proximo da coluna
    | (getValue (getPosition i j board) /= (-1)) = solveAux i (j+1) guess board n regionDict
    -- Senao tenta atribuir um valor a posicao:
    | otherwise = do
        -- verifica o chute
        case guess of
            -- Se passar do ultimo chute, cancela e temos que voltar um passo
            10 -> Nothing
            -- Se ainda nao chegou no ultimo chute
            --   Tenta atribui um valor a posicao
            otherwise -> do
                -- Atribui um valor a posicao
                let pos = getPosition i j board
                let regPos = getRegion pos
                let regionList = findWithDefault [] regPos regionDict
                let newBoard = changeValueBoard board i j guess

                -- Verifica se valor atribuido a posicao eh valido
                case (validate i j guess regionList newBoard) of
                    -- Caso nao for -> Tenta atribuir proximo valor
                    False -> solveAux i j (guess+1) board n regionDict

                    -- Caso for valido
                    True ->
                        -- Tentamos resolver o novo tabuleiro
                        case (solve newBoard) of
                            -- Se nao for possivel -> Passamos para proximo valor
                            Nothing -> solveAux i j (guess+1) board n regionDict
                            -- Se for possivel -> retorna o tabuleiro pronto
                            Just bd -> Just bd
```

Para verificar se a casa possui um valor válido, respeitando as regras já citadas neste relatório, a função *validate* é chamada: recebendo como parâmetros, as coordenadas (I, J) da casa a ser verificada, o valor candidato a ser atribuído, e a lista de valores existentes na região, retorna um booleano indicando a validação.

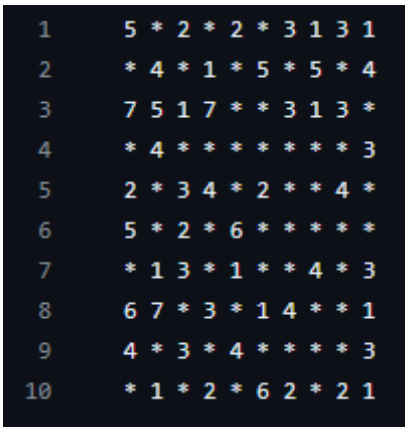
```haskell
-- Verifica se valor atribuido a posicao eh valido
validate :: I -> J -> Int -> RegionList -> Board -> Bool
validate i j guess rl board =
    -- Se valor ja esta presente na regiao
    notInArr guess rl &&
    -- Se o valor eh maior do que a regiao permite
    (guess <= regionLen) &&
    -- Verifica se adequa a posicao de cima
    verifyAbovePosition i j guess pos board &&
    -- Verifica se adequa a posicao de baixo
    verifyBellowPosition i j guess pos board n &&
    -- Verifica se adequa a posicao da esquerda
    verifyLeftPosition i j guess board &&
    -- Verifica se adequa a posicao da direita
    verifyRightPosition i j guess board n where
          pos = getPosition i j board
          n = getLen board
          regionLen = getLen rl
```

Por fim, caso o tabuleiro passado seja possível de resolver, é feita a impressão do puzzle resolvido.

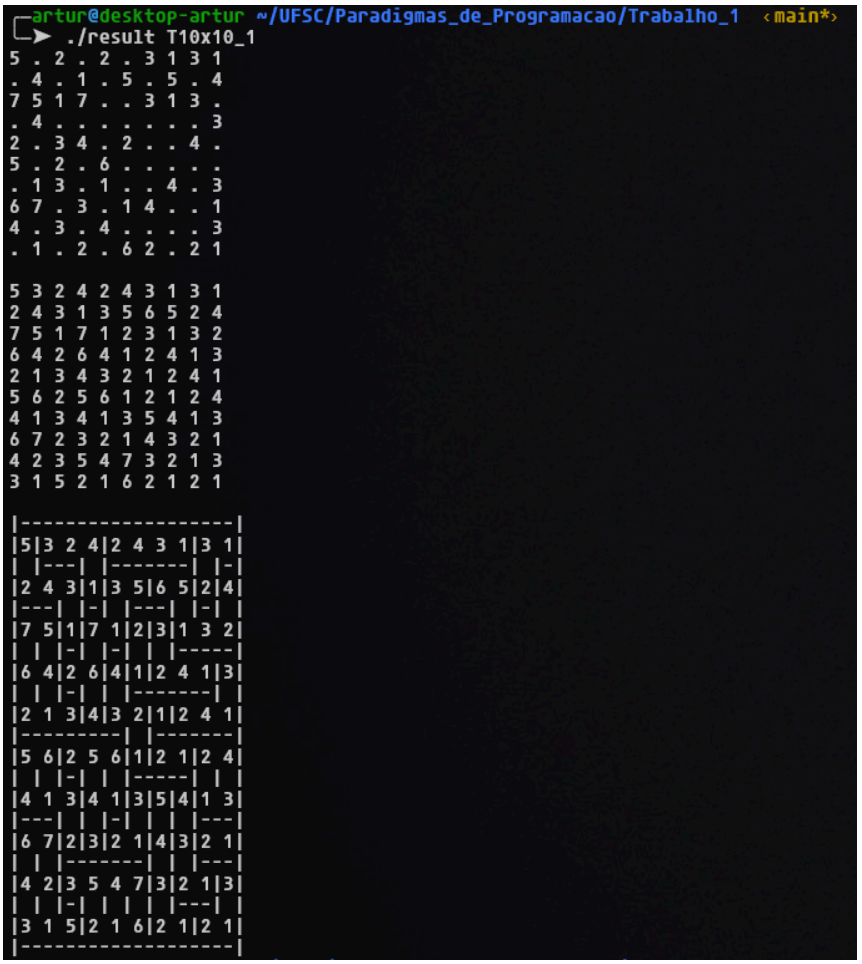
```haskell
main = do
    -- Define matrizes por argumento
    args <- getArgs
    let path = head args
    (valueMatriz, regionsMatriz) <- readBoard path

    -- Monta tabuleiro e dicionario das regioes
    let board = makeBoard valueMatriz regionsMatriz 0
    let regionsDict = defineRegions board

    printBoard board
    putStrLn ""

    let result = solve board
    case (result) of
        Nothing -> putStrLn "Tabuleiro nao tem solucao"
        Just board -> do
            printBoard board
            putStrLn ""
            prettyPrint board
```

3. EXECUÇÃO

Como já dito anteriormente, o problema a ser resolvido é passado como argumento do executável criado na compilação, que é o nome do diretório que possui dois arquivos textos, um com os valores de cada campo, e outro indicando as regiões (indicadas por caracteres).



Logo, com o tabuleiro resolvido é feito a impressão do estado inicial do quadro, depois uma impressão simplificada de cada campo, e por fim, uma impressão mais detalhada da solução, com suas regiões separadas.



4. CONCLUSÃO E DIFICULDADES

Considerando os tópicos supracitados, é possível concluir que o projeto proposto, foi uma ótima prática, para o um melhor conhecimento da linguagem funcional Haskell, nos permitindo pesquisar e ir atrás de diferentes soluções para o problema escolhido, e então analisarmos qual o melhor para o devido desafio, como por exemplo, a implementação do algoritmo de backtracking, que no meu caso, foi onde mais encontrei dificuldade.

Dessa forma, tive dificuldade de implementar o backtracking em Haskell, pois mesmo já tendo feito anteriormente na linguagem python, Haskell é uma linguagem puramente funcional, um paradigma que não estava acostumado.

Além disso, o trabalho permitiu o conhecimento de novos conceitos, como Monads, que de forma resumida, são estruturas que combinam funções e envolvem seus valores de retorno em um tipo de computação adicional, permitindo controlar a ordem de dependência de computação em linguagens funcionais.