UNIVERSIDADE FEDERAL DE SANTA CATARINA

Paradigmas de Programação - INE5416

Professor: Maicon Rafael Zatelli

Alunos:

Allan Soares Silva	(19200410)
Antonio Silverio Montagner	(19203742)
Eduardo Vinicius Betim	(19203161)

Trabalho I - Programação Funcional - Haskell

Puzzle Vergleichssudoku

Florianópolis

19 de Outubro de 2022

1. PROBLEMA ESCOLHIDO

O problema abordado neste trabalho foi o puzzle Vergleichssudoku. O puzzle possui sinais de relação de maior e menor entre duas células, considerando que para a célula qual a seta aponta é possui um número menor que o na outra célula, assim estabelecendo uma relação entre elas. Os símbolos de relação entre células são “<“, “>”, “^”, “v”.

Cada uma das células do puzzle possuem de duas a quatro setas nas divisas entre si, assim representando a comparação que deve ser feita.

O puzzle tratado possui o tamanho de 9x9, que é 9 linhas por 9 colunas, possuindo no total 6 regiões de dimensões 3x3. Escreva números de 1 a N nas células da grade de tamanho NxN, de modo que cada número ocorre exatamente uma vez em cada linha, em cada coluna e em cada região.

Além de respeitar a comparação apresentada, um número não pode se repetir numa mesma linha, coluna ou bloco.

2. SOLUÇÃO ESCOLHIDA

Para resolver o problema apresentado, o algoritmo trata o tabuleiro como uma lista de 81 células e percorre ela célula a célula, validando os números presentes na célula acima e à esquerda da célula atual. Também existe uma matriz 9x9 das possibilidades de cada célula. Quando o algoritmo chega em uma célula, ele calcula as possibilidades de números que podem preenchê-la e guarda-as em uma lista na matriz de possibilidades, no índice equivalente a célula atual.

A ideia dessa solução é baseada em um algoritmo regular para solucionar um tabuleiro de sudoku, mas também validando as células adjacentes enquanto o tabuleiro é explorado, seguindo a regra de maior/menor presente na variação Vergleich.

2.1 ENTRADA DE DADOS

Ao executar o programa, é pedido o '*Board id:*' do qual é o identificador do tabuleiro (sem sua extensão '.txt') do qual será solucionado. Para isso, é necessário ter o arquivo do tabuleiro devidamente salvo na pasta 'boards'.

A entrada é por meio da leitura e *parsing* de um arquivo de texto no formato da Figura 1, que representa o tabuleiro de um puzzle solucionável. O arquivo contém 9 linhas, cada uma com uma sequência de 9 pares de caracteres que representam as operações à direita e abaixo, respectivamente, de cada célula. Uma célula com a dupla **+\-**, por exemplo, indica que a mesma deve possuir um valor maior que a célula à direita e menor que a célula abaixo.

Existe também o identificador **.** que indica que a célula não possui uma condição em relação à célula da direita, caso seja o primeiro caractere da dupla, ou em relação à célula abaixo, caso seja o segundo caractere da dupla.

Por fim, será retornada uma matriz, parecida com o da Figura 3, com a solução do puzzle passado. A solução é retornada diretamente na interface de execução.

```
++ ++ .- +- ++ .- ++ +- .-
-- -+ .- ++ -+ .+ ++ +- .-
+. -. .. +. -. .. -. -. ..
++ -- .- +- -- .+ ++ +- .+
+- -- .- -+ ++ .- -+ ++ .-
-. +. .. -. +. .. +. -. ..
-- -+ .+ -- +- .- ++ -- .+
-- +- .- -- ++ .+ -+ ++ .+
-. +. .. +. +. .. -. -. ..
```

Figura 1: Formato dos dados de entrada.

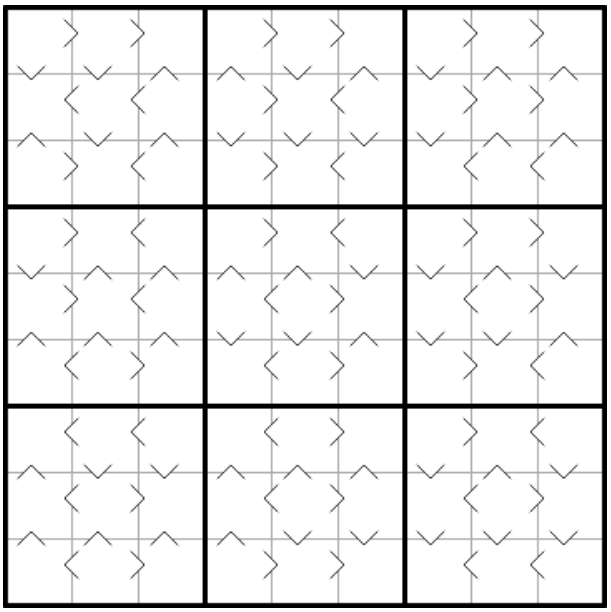

Figura 2: Imagem do puzzle número 111.

```
9++ 7++ 5.-   6+- 4++ 3.-   8++ 2+- 1.-
1-- 4-+ 6.-   8++ 2-+ 9.+   7++ 5+- 3.-
3+. 2-. 8..   5+. 1-. 7..   4-. 6-. 9..

6++ 1-- 2.-   4+- 3-- 8.+   9++ 7+- 5.+
5+- 3-- 4.-   7-+ 9++ 1.-   6-+ 8++ 2.-
8-. 9+. 7..   2-. 6+. 5..   3+. 1-. 4..

2-- 6-+ 9.+   1-- 7+- 4.-   5++ 3-- 8.+
4-- 5+- 1.-   3-- 8++ 6.+   2-+ 9++ 7.+
7-. 8+. 3..   9+. 5+. 2..   1-. 4-. 6..
```

Figura 3: Imagem da solução para o puzzle número 111 retornado ao usuário.

2.2 ELABORAÇÃO

2.2.1 Estruturas e tipos de dados utilizadas

O tipo de dado Cell representa uma célula e possui os campos value, right e bottom que representam o valor, o char correspondente a operação na parte de baixo da célula e o char correspondente a operação do lado direito da célula.

```haskell
data Cell = Cell {value :: Int, right :: Char, bottom :: Char}
```

2.2.2 Manipulação dos dados

Ao longo do código o tabuleiro é tratado às vezes como uma matriz de ordem 9, e às vezes como uma lista de 81 elementos. Isso é feito apenas por conveniência, já que algumas operações são mais fáceis de serem implementadas como lista. As funções a seguir são responsáveis pela conversão entre uma estrutura e outra:

```haskell
boardToList :: Board -> [Cell]
boardToList = concat

listToBoard :: [Cell] -> Board
listToBoard [] = []
listToBoard b = take 9 b : listToBoard (drop 9 b)

itop :: Int -> (Int, Int)
itop i = swap (i `divMod` 9)
```

Para transformar os caracteres right e bottom da *Cell* em operações válidas, foi implementada a função *partialOp*, que recebe um *Char* e um Int e retorna uma função (equivalente ao *Char* fornecido) parcialmente aplicada com o Inteiro.

```haskell
partialOp :: Char -> Int -> (Int -> Bool)
partialOp '+' val = (>) val
partialOp '-' val = (<) val
partialOp _ _  = const True
```

O *fallback* de *partialOp* é uma função parcial que, independente do segundo valor aplicado, sempre retornará verdadeiro. Ele será utilizado em casos onde a célula não possui uma operação em algum dos lados.

As outras funções principais utilizadas na manipulação de dados são responsáveis por retornar as células presentes na linha, coluna e região de uma dada coordenada.

```haskell
cellAt :: Board -> (Int, Int) -> Cell
cellAt b (x, y) = (b !! y) !! x

rowAt :: Board -> Int -> Line
rowAt m i
    | i < 0 || i >= length m = []
    | otherwise = m !! i

columnAt :: Board -> Int -> Line
columnAt [] _ = []
columnAt (x:xs) i
    | i < 0 || i >= length x = []
    | otherwise = x !! i : columnAt xs i

regionAt :: Board -> (Int, Int) -> Line
regionAt b (x, y) = let
        y' = (y `div` 3 * 3)
        x' = (x `div` 3 * 3)
        rows = take 3 (drop y' b)
    in
    concatMap (take 3 . drop x') rows
```

2.2.3 Elaboração do solucionador do problema

Percorre o tabuleiro e determina os números que devem preencher cada célula, usando a função possibilities para determinar os valores possíveis em cada iteração.

As possibilidades são calculadas gerando uma lista no intervalo [1, 9] e removendo os valores já utilizados na linha, coluna e região em que a célula está presente. Após isso, a lista é filtrada novamente com base na aplicação da operação *bottom* da célula acima da atual e na operação *right* da célula à esquerda da atual utilizando o valor da célula atual em ambas[1].

[1] Existe uma pequena optimização feita no código que leva em consideração as operações nos 4 lados da célula, e pode reduzir o intervalo [1, 9] e ocasionar em uma quantidade consideravelmente menor de passos durante a solução. Essa otimização será explicada mais detalhadamente adiante.

```haskell
possibilities :: [Cell] -> Int -> [Int]
possibilities b' index =
    let
        coord@(x, y) = itop index
        b = listToBoard b'
        currentCell = cellAt b coord
        upperCell = if y > 0 then cellAt b (x, y - 1) else Cell 0 '.' '.'
        leftCell = if x > 0 then cellAt b (x - 1, y) else Cell 0 '.' '.'

        assert v = all (\cond -> cond v) [rightOp leftCell, bottomOp upperCell]

        usedValues = map value (rowAt b y ++ columnAt b x ++ regionAt b coord)

        operations = [
            inverseOf (right leftCell),
            inverseOf (bottom upperCell),
            right currentCell,
            bottom currentCell]
        minValue = 1 + length (filter (=='+') operations)
        maxValue = 9 - length (filter (=='-') operations)

        possibilities = filter (`notElem` usedValues) [minValue .. maxValue]
    in
    filter assert possibilities
```

Os passos para solucionar são:

0. O índice inicia em 0 e a matriz de possibilidades inicia vazia
1. Verifica o valor do índice
    1.1 O valor é -1 ou 81
    1.1.1 Passo 4
2. O algoritmo está incrementando ou decrementando?
    2.1 Incrementando
    2.1.1 Calcula as possibilidades da célula atual
    2.1.2 Passo 3
    2.2 Decrementando
    2.2.1 Busca as possibilidades restantes disponível na matriz para a célula atual
3. Verifica as possibilidades
    3.1. Há possibilidades

3.1.1. Substitui o valor da célula no tabuleiro pela primeira possibilidade calculada

3.1.2. Guarda o resto das possibilidades na matriz de possibilidades

3.1.3. Vai para a próxima célula

3.1.4. Passo 1

3.2. Não há possibilidades

3.2.1 Zera o valor da célula atual no tabuleiro

3.2.2 Retorna à célula anterior

3.2.3 Passo 1

4. Fim do algoritmo.

```haskell
solve :: Board -> Board
solve b = let
        board = boardToList b
        -- Cria uma matriz de possibilidades para cada célula do tabuleiro
        possMatrix =  map (const []) board
    in
    go board possMatrix 0 True where
        go b _ (-1) _ = listToBoard b
        go b _ 81 _ = listToBoard b
        go b b' i forward = let
                poss = if forward then possibilities b i else b' !! i
            in
            case poss of
                [] -> let
                        currCell = b !! i
                        emptyCell = Cell 0 (right currCell) (bottom currCell)
                    in
                    go (replace b emptyCell i) b' (i - 1) False
                _  -> let
                        (x:xs) = poss
                        currCell = b !! i
                        newCell = Cell x (right currCell) (bottom currCell)
                    in
                    go (replace b newCell i) (replace b' xs i) (i + 1) True
```

2.2.4 Otimização

A otimização implementada no cálculo de possibilidades leva em conta todas as quatro operações às quais uma célula está relacionada. O menor valor possível para uma célula é a quantidade + 1 de operações “maior que” dentre quatro,

enquanto o menor valor possível é 9 - a quantidade de operações “menor que” dentre essas quatro.

Executando o algoritmo para o tabuleiro 111 através do GHCi, a solução demora em média 1 minuto e 34 segundos sem a otimização. Com a otimização aplicada nas mesmas pré-condições, a execução demora aproximadamente 45 segundos para finalizar. Fora do ambiente do GHCi e suas limitações de recursos, ambas versões demoram menos de 10 segundos sob as mesmas pré-condições.

3. PROBLEMAS ENCONTRADOS

As maiores dificuldades da implementação foram relacionadas à operações das células, tanto na forma como representá-las, quanto na verificação de valores.

Inicialmente a ideia era que os campos *right* e *bottom* das células fossem realmente funções parciais do tipo *Int -> Bool* em vez de *Char*, que posteriormente são mapeadas para funções, mas isso dificultaria a utilização do construtor *Cell* em outros lugares e tornaria o código desnecessariamente complexo. Outra tentativa de melhoria foi utilizar *enumerations* em vez de *Char* para representar as funções

```
data Operation = Noop | GreaterThan | LessThan
```

porém, *enumerations* não suportam a otimização de *unpacking* feita pelo GHC, e isso impactou fortemente a performance do algoritmo tornando-o inutilizável.