Relatório I Haskell

Aluno: Jéssica Regina dos Santos
Matrícula: 22100626
Disciplina: INE5416 - Paradigmas de Programação
Professor: Maicon Rafael Zatelli
Data: 05/12/2024

---

1. Introdução

Este projeto implementa um resolvedor de Sudoku de Comparação usando Haskell.

Além das regras clássicas do Sudoku são consideradas condições extras que relacionam valores de células (como "a célula A deve conter um número menor que a célula B).

O algoritmo utiliza backtracking para encontrar soluções válidas, explorando todas as possibilidades de preenchimento de células de forma eficiente.

---

2. Estrutura do Código

2.1 Tipos de Dados

- Tabuleiro representa o tabuleiro como uma lista de listas de inteiros, onde 0 indica células vazias.
- Condicao representa restrições entre células definidas como pares de coordenadas que impõem uma relação (ex.: menor que).

```
type Tabuleiro = [[Int]] -- Define o tipo
type Condicao = ((Int, Int), (Int, Int))
```

2.2 Funções Principais

1. **ehValido**
   Verifica se um número pode ser inserido em uma célula específica, considerando as regras clássicas do Sudoku (ausência do número na linha, coluna e bloco) e restrições adicionais especificadas em Condicoes.

```haskell
ehValido :: Tabuleiro -> Int -> Int -> Int -> [Condicao] -> Bool
ehValido tabuleiro linha coluna numero condicoes =
  notElem numero (tabuleiro !! linha) &&                        -- Verifica se o número já está na li
  notElem numero [tabuleiro !! r !! coluna | r <- [0..n-1]] &&   -- Verifica se o número já está na c
  notElem numero (elementosDoBloco linha coluna) &&              -- Verifica se o número já está no b
  all condicaoValida condicoes                                   -- Verifica se todas as condições sã
  where
    n = length tabuleiro -- Tamanho do tabuleiro
    tamanhoBloco = round . sqrt . fromIntegral $ n -- Calcula o tamanho de um bloco
    inicioLinha = (linha `div` tamanhoBloco) * tamanhoBloco -- Linha inicial do bloco
    inicioColuna = (coluna `div` tamanhoBloco) * tamanhoBloco -- Coluna inicial do bloco
    elementosDoBloco l c = [tabuleiro !! i !! j | i <- [inicioLinha..inicioLinha+tamanhoBloco-1],
                                                  j <- [inicioColuna..inicioColuna+tamanhoBloco-1]]
    condicaoValida ((x1, y1), (x2, y2)) -- Verifica se a condição entre duas células é válida
      | (linha, coluna) == (x1, y1) = tabuleiro !! x2 !! y2 == 0 || numero < tabuleiro !! x2 !! y2
      | (linha, coluna) == (x2, y2) = tabuleiro !! x1 !! y1 == 0 || numero > tabuleiro !! x1 !! y1
      | otherwise = True -- Caso contrário, a condição não é relevante para esta célula
```

2. **resolveSudoku**

   Implementa a lógica de backtracking:

   - Encontra a próxima célula vazia.
   - Testa números possíveis (1 a n).
   - Valida cada tentativa usando ehValido.
   - Atualiza o tabuleiro recursivamente até encontrar uma solução ou concluir que nenhuma solução é possível.

```haskell
resolveSudoku :: Tabuleiro -> [Condicao] -> Maybe Tabuleiro
resolveSudoku tabuleiro condicoes
  | null celulasVazias = Just tabuleiro  -- Se não há células vazias, o tabuleiro está resolvid
  | otherwise = tentaValores tabuleiro (head celulasVazias) -- Tenta preencher a próxima célula
  where
    n = length tabuleiro -- Tamanho do tabuleiro
    celulasVazias = [(r, c) | r <- [0..n-1], c <- [0..n-1], tabuleiro !! r !! c == 0] -- Lista
    tentaValores t (r, c) = foldr (\num acc -> acc <|> tenta num) Nothing [1..n] -- Tenta cada
      where
        tenta numero
          | ehValido t r c numero condicoes = -- Se o número é válido para a célula
              resolveSudoku (atualizaTabuleiro t r c numero) condicoes -- Avança para a próxima
          | otherwise = Nothing -- Caso contrário, tenta outro número
    atualizaTabuleiro t r c numero = take r t ++
                                     [take c (t !! r) ++ [numero] ++ drop (c + 1) (t !! r)] ++
                                     drop (r + 1) t -- Atualiza o tabuleiro com o novo número
```

3. **imprimeTabuleiro**

   Exibe o tabuleiro no console, com células vazias representadas por '.'.

```haskell
imprimeTabuleiro :: Tabuleiro -> IO ()
imprimeTabuleiro = mapM_ (putStrLn . unwords . map mostraCelula) --
  where
    mostraCelula 0 = "." -- Representa células vazias como ponto
    mostraCelula n = show n -- Exibe o número da célula
```

4. **main**

   Define o tabuleiro inicial, as condições de restrição e executa o resolvedor.

```haskell
-- Tabuleiro inicial e condições
main :: IO ()
main = do
  let n = 4 -- Tamanho do tabuleiro
      tabuleiro = replicate n (replicate n 0) -- Cria um tabulei
      condicoes = [((0, 0), (0, 1)), -- Ex. cond.: a célula (0,
                   ((0, 3), (0, 2)),
                   ((0, 0), (1, 0)),
                   ((1, 1), (1, 0)),
                   ((0, 1), (1, 1)),
                   ((1, 2), (0, 2)),
                   ((1, 2), (1, 3)),
                   ((1, 3), (0, 3)),
                   ((2, 0), (2, 1)),
                   ((2, 3), (2, 2)),
                   ((3, 0), (2, 0)),
                   ((3, 1), (2, 1)),
                   ((3, 1), (3, 0)),
                   ((2, 2), (3, 2)),
                   ((2, 3), (3, 3)),
                   ((3, 2), (3, 3))] -- Outras condições, seguir
  putStrLn "Tabuleiro inicial:" -- Mostra a mensagem inicial
  imprimeTabuleiro tabuleiro -- Mostra o tabuleiro vazio
  case resolveSudoku tabuleiro condicoes of
    Just solucao -> do -- Se uma solução for encontrada, mostra
      putStrLn "\nSolução encontrada:"
      imprimeTabuleiro solucao -- Então mostra o tabuleiro resol
    Nothing -> putStrLn "\nNenhuma solução encontrada." -- Caso
```

---

3. Algoritmo

3.1 Validação de Células

As condições adicionais são verificadas individualmente, considerando apenas aquelas relevantes para a célula atual.

O cálculo do bloco é baseado na raiz quadrada do tamanho do tabuleiro, garantindo generalização para tabuleiros n x n.

3.2 Estratégia de Backtracking

O algoritmo explora as possibilidades de forma exaustiva.

Quando uma tentativa falha, retrocede para tentar outra possibilidade.

Utiliza-se a operação <|> da biblioteca Control.Applicative para combinar tentativas de forma organizada.

---

4. Restrições

As restrições adicionais tornaram o problema mais desafiador, adicionando um nível de complexidade além do Sudoku clássico. Elas são implementadas por meio da função condicaoValida, que compara valores entre células dependendo da coordenada atual.

Exemplo de condição: ((0, 0), (0, 1)) significa que o valor em (0, 0) deve ser menor que o valor em (0, 1). Pensar como valores i (linha) e j (coluna) de uma matriz, começando pelo zero.

---

5. Análise de Complexidade

- **Complexidade Temporal**:
  Para um tabuleiro n x n, a complexidade no pior caso é aproximadamente O(n!), devido ao número de combinações possíveis de valores em células vazias. Adicionar condições extras pode aumentar o custo computacional, mas otimizações evitam verificações desnecessárias.

---

6. Resultados Esperados

O código é inicializado com um tabuleiro vazio de tamanho nxn e um conjunto de condições. Após a execução, se uma solução válida existir, o tabuleiro resolvido será exibido. Caso contrário, uma mensagem indicando "Nenhuma solução encontrada" será mostrada.

---

7. Potenciais Melhorias

1. **Otimização do Backtracking**
   Implementar heurísticas, como escolher a célula com menos opções de preenchimento primeiro.
2. **Generalização**
   Permitir tabuleiros maiores, como 9x9, ajustando as condições.

3. **Interface Gráfica**
   Adicionar uma visualização gráfica para melhorar a interação com o usuário.

---

8. Conclusão

O criado é funcional. Ele resolve tabuleiros de Sudoku de Comparação com condições adicionais, demonstrando a aplicação de técnicas de programação funcional e backtracking. Apesar de já ser eficiente para casos pequenos, há espaço para melhorias em termos de desempenho e usabilidade.