Universidade Federal de Santa Catarina
Departamento de Informática e Estatística - INE
Curso de Ciências da Computação

Departamento de Informática e Estatística
CTC • UFSC

Alunos: Isaque Beirith e Luís Fernando Mendonça Junior
Disciplina: Paradigmas de Programação

Relatório

Análise do problema

Kojun é um quebra-cabeça lógico japonês, no qual o objetivo é completar as regiões do tabuleiro com número de 1 até N, sendo N o número de células da região. As regras definem que números em células ortogonais vizinhas devem ser diferentes e também que duas células adjacentes verticalmente na mesma região devem sempre conter o número maior na célula superior e o menor na inferior.

O problema consiste na implementação de um resolvedor de Kojun na linguagem Haskell, que recebe como entrada um tabuleiro qualquer do quebra-cabeça.

Solução implementada

O código implementa uma abordagem de busca com backtracking para encontrar uma solução válida para o tabuleiro.

```haskell
-- Matriz

-- Definição de tipos

type Valor = Int
type Linha i = [i]
type Matriz a = [Linha a]
type Tabuleiro = Matriz Valor
```

```haskell
-- Kojun

-- Definição de tipos
type Escolhas = [Valor]
```

O código define tipos personalizados para representar os valores, as células do tabuleiro, os grupos e as escolhas possíveis.

```haskell
-- Retorno de informações da matriz
linhas :: Matriz m -> [Linha m]
linhas i = i
colunas :: Matriz m -> [Linha m]
colunas j = transpose j
dimensao :: Matriz m -> Int
dimensao d = length(d!!0)
```

Ele também define funções para manipular matrizes, como obter linhas, colunas e dividir matrizes.

```haskell
--Divide a matriz com valores de acordo com a matriz de grupos
gruposMatriz :: Eq m => Matriz m -> Tabuleiro -> [Linha m]
gruposMatriz valores grupos = [grupoFiltro grupo valoresTupla | grupo <- gruposMapa]
    where
        valoresTupla = foldl1 (++) (zipWith zip valores grupos)
        gruposMapa = nub (map snd valoresTupla)
        grupoFiltro grupo list = map fst $ filter ((== grupo) . snd) list
```

Então o tabuleiro é dividido em grupos, e valores iniciais são fornecidos para algumas células. Os grupos são representados em uma matriz separada, onde cada célula pertence a um grupo específico.

```haskell
-- Define as escolhas possíveis para cada espaço no tabuleiro
escolhas :: Tabuleiro -> Tabuleiro -> Matriz Escolhas
escolhas valores grupos = map (map choice) (zipWith zip valores grupos)
    where
        choice (v, p) = if v == 0 then [1..(tamanhoGrupo p grupos)] `minus` (valorGrupo valores grupos p) else [v]
```

A função **'escolhas'** é responsável por calcular as escolhas possíveis para cada célula do tabuleiro com base nos grupos, por exemplo, para um grupo com três células, as escolhas possíveis podem ser os números de 1 a 3.

```haskell
-- Define o valor para um espaço onde há somente uma solução possível
reduzirEscolhasLista :: Linha Escolhas -> Linha Escolhas
reduzirEscolhasLista xss = [xs `minus` singles | xs <- xss]
    where
        singles = concat (filter elementoUnico xss)
```

Já a função **'reduzirEscolhas'** reduz o número de escolhas possíveis nas células onde há apenas uma solução clara, como quando há uma célula em um grupo com um valor já atribuído, essa escolha é removida das outras células do mesmo grupo.

```haskell
-- Faz a busca de todas as soluções possíveis por casa, informando se tabuleiro
-- possui soluções ou não e se é necessário expandir a busca
buscaSolucao :: Matriz Escolhas -> Tabuleiro -> [Tabuleiro]
buscaSolucao valores grupos
    | semSolucao valores grupos = []
    | all (all elementoUnico) valores = [map concat valores]
    | otherwise = [g | valores' <- expandirBusca valores, g <- buscaSolucao (reduzirEscolhas valores' grupos) grupos]
```

A função **'buscaSolucao'** é a parte central do código. Ela é responsável por encontrar a solução para o tabuleiro utilizando a lógica de backtracking. O algoritmo primeiramente verifica se o tabuleiro já não tem solução, verificando se uma célula não tem escolhas possíveis, se todas as células tiverem escolhas únicas, o tabuleiro é considerado resolvido; caso contrário, expande as escolhas possíveis em uma célula e chama recursivamente **'buscaSolucao'** para cada possibilidade, a recursão continua até que todas as células tenham escolhas únicas ou até que uma solução seja encontrada.

```haskell
-- Define se matriz é válida
valida :: Matriz Escolhas -> Tabuleiro -> Bool
valida valores grupos = all adjacenteValido (colunas valores) &&
                        all adjacenteValido (linhas valores) &&
                        all linhaValida (gruposMatriz valores grupos) &&
                        all linhaDecrescente (grupoColunas valores grupos)
```

Por fim é feita a verificação para confirmar se a solução encontrada é válida de acordo com as regras do Kojun, como garantir que as células adjacentes tenham valores diferentes, que não haja repetições de valores nas linhas, que as células estejam em ordem decrescente, entre outros.

```haskell
-- Tabuleiros:

-- Tabuleiro 6x6 - https://www.janko.at/Raetsel/Kojun/001.a.htm
valores6x6 :: Tabuleiro
valores6x6 = [[2,0,0,0,1,0],
              [0,0,0,3,0,0],
              [0,3,0,0,5,3],
              [0,0,0,0,0,0],
              [0,0,3,0,4,2],
              [0,0,0,0,0,0]]
-- Grupos do Tabuleiro 6x6
grupos6x6 :: Tabuleiro
grupos6x6 = [[1,1,2,2,2,3],
             [4,4,4,4,4,3],
             [5,6,6,6,4,7],
             [5,5,5,6,7,7],
             [8,8,10,0,0,0],
             [9,9,10,10,0,0]]
```

A solução implementada utiliza de tabuleiros (adquiridos no próprio site janko) em formato de matrizes presentes no próprio código, dividido em dois tipos, o de valores, que apresenta o valor presente em cada célula do tabuleiro; e o de grupos, que delimita os grupos do tabuleiro.

Organização do trabalho

O grupo inicialmente seguiu a recomendação do professor e pesquisou por implementações de soluções do quebra-cabeça Sudoku na linguagem Haskell para utilizar como base, visto que ambos os jogos possuem um funcionamento semelhante. Também foi pesquisado acerca da técnica de “tentativa e erro” em Haskell.

O grupo realizou reuniões por meio de chamadas de voz para implementar o trabalho. Foi utilizada a extensão liveshare do aplicativo VSCode, que possibilita a edição de arquivos em conjunto, para que os membros pudessem escrever e editar o código de maneira síncrona.

Dificuldades encontradas

As maiores dificuldades encontradas foram relacionadas ao percorrimento das matrizes, junto da manipulação, verificação e validação dos dados na linguagem Haskell, pois é um procedimento um tanto quanto confuso e diferente do qual estávamos

acostumados. O grupo precisou pesquisar vídeos e conteúdos relacionados sobre o tema para realizar a implementação.

Também tivemos problemas com a própria sintaxe da linguagem, e foi preciso rever os conteúdos apresentados na disciplina.