UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO
PARADIGMAS DA PROGRAMAÇÃO

Trabalho I - Programação Funcional - Haskell
Código solucionador de Kojun

ALEX DAVIS NEUWIEM DA SILVA (21202103)
LUAN DINIZ MORAES (21204000)
LUCAS CASTRO TRUPPEL MACHADO (22100632)

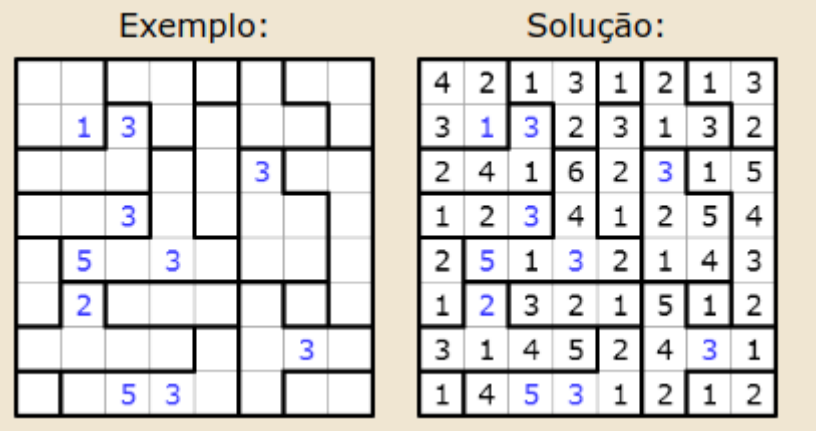
FLORIANÓPOLIS, 26 DE ABRIL DE 2023

1. Introdução

O puzzle escolhido para ser resolvido é o Kojun. O código foi feito na linguagem haskell utilizando backtracking como método de resolução.

2. Análise do problema

Kojun é um puzzle semelhante ao jogo sudoku. Inicialmente o jogador recebe um tabuleiro com alguns números preenchidos e deve completar os espaços vazios com números de acordo com algumas regras. Além disso, o tabuleiro é dividido em algumas regiões, indicadas por traços mais escuros e espessos, que são relevantes para resolução. A seguir um exemplo do tabuleiro inicial e de sua solução.



Para preencher os espaços em brancos temos apenas três regras:

1) Uma região precisa ter todos os números de 1 a N sem repetição, sendo N a quantidade de células na região.
2) Números em células ortogonalmente adjacentes devem ser diferentes.
3) Se duas células estiverem adjacentes verticalmente na mesma região, o número na célula superior deve ser maior que o número na célula inferior.

3. Entrada e saída

O programa precisa de duas matrizes como entrada, que devem ser especificadas diretamente no código. Uma matriz com os números iniciais das

células (com 0 nas casas vazias) e a outra matriz determina as diferentes regiões do tabuleiro. As regiões devem ser representadas por inteiros de 0 até n, sendo “n” a quantidade de regiões menos um. O arquivo “docs/exemploEntrada.txt” possui 3 exemplos de entrada que podem ser copiados em “haskell/kojun.hs”. Abaixo um exemplo de entrada.

```haskell
-- Matriz com os números inicias do puzzle kojun
matrizNumerosInicial :: [[Int]]
matrizNumerosInicial = [[0, 4, 3, 0, 2, 5, 0, 0, 0, 0],
                        [0, 2, 0, 0, 0, 4, 2, 0, 3, 0],
                        [0, 0, 0, 1, 4, 0, 0, 1, 0, 0],
                        [5, 6, 0, 2, 3, 0, 5, 0, 0, 0],
                        [0, 3, 5, 0, 0, 0, 3, 0, 0, 0],
                        [0, 0, 0, 7, 0, 7, 0, 5, 0, 4],
                        [0, 0, 5, 3, 0, 2, 0, 4, 0, 0],
                        [0, 0, 1, 5, 0, 0, 0, 5, 3, 0],
                        [1, 3, 7, 0, 0, 0, 6, 0, 0, 5],
                        [2, 1, 0, 0, 3, 0, 1, 0, 3, 4]]

-- Matriz que define as regiões do quebra-cabeça. As regiões devem ser representas por inteiros de 0 até n,
-- sendo n a quantidade de regiões - 1. Esses inteiros podem ser considerados o id de cada região.
matrizRegioes :: [[Int]]
matrizRegioes = [[0 , 1 , 1 , 1 , 1 , 2 , 3 , 4 , 4 ],
                 [0 , 0 , 1 , 1 , 2 , 2 , 2 , 3 , 3 , 5 ],
                 [0 , 0 , 6 , 6 , 6 , 7 , 8 , 3 , 3 , 5 ],
                 [6 , 6 , 6 , 12, 12, 7 , 8 , 9 , 10, 11],
                 [6 , 13, 13, 12, 14, 7 , 8 , 9 , 10, 11],
                 [13, 13, 14, 14, 14, 15, 8 , 10, 10, 10],
                 [13, 13, 14, 14, 15, 15, 8 , 8 , 8 , 16],
                 [17, 17, 14, 18, 15, 15, 15, 15, 16, 16],
                 [17, 17, 18, 18, 18, 18, 18, 16, 16, 16],
                 [19, 19, 19, 19, 18, 20, 20, 20, 20, 16]]
```

A saída é impressa no terminal. No caso de sucesso, é a matriz resolvida. Caso não encontre solução, é "Nao ha solucao".

```
lucas@lucas-Nitro-AN515-51:~/Documentos/Faculdade/2023.1/paradigmas_da_programacao/trabalhos/trabalho1/kojun$ ./a
 5 4 3 7 2 5 3 5 2 1
 4 2 1 6 1 4 2 4 3 2
 3 1 7 1 4 3 7 1 2 1
 5 6 3 2 3 2 5 2 3 2
 2 3 5 1 4 1 3 1 2 1
 6 2 6 7 2 7 2 5 1 4
 4 1 5 3 6 2 1 4 6 7
 2 4 1 5 3 1 4 5 3 6
 1 3 7 1 4 2 6 1 2 5
 2 1 3 4 3 4 1 2 3 4
```

4. Descrição da solução

A estratégia utilizada para a solução é o backtracking. A função principal da solução foi nomeada de “kojun” e ela é chamada pelo menos uma vez para cada posição da matriz. Esse método verifica se a posição é válida, ou seja, se é uma

posição que existe na matriz, e, caso ela esteja vazia, chama a função “avaliarNumeros”, que busca um número para colocar na posição analisada.

```haskell
kojun :: Int -> Int -> [[Int]] -> [[Int]] -> [[(Int, Int)]] -> (Bool, [[Int]])
kojun i j numerosMatriz regioesMatriz regioes =

    -- Percorreu a matriz inteira sem achar erros
    if (i == tamanhoMatriz - 1) && (j == tamanhoMatriz) then
        (True, numerosMatriz)

    -- Terminou a linha
    else if (j == tamanhoMatriz) then
        kojun (i+1) 0 numerosMatriz regioesMatriz regioes

    -- Posição já está ocupada
    else if ((numerosMatriz !! i !! j) > 0) then
        kojun i (j+1) numerosMatriz regioesMatriz regioes

    -- Posição válida e vazia, procura um número para ocupá-la
    else
        let maxNum = tamanhoRegiao regioes (regioesMatriz !! i !! j)
        -- Começa avaliando a partir do número 1 (segundo parâmetro)
        in avaliarNumeros 1 maxNum i j numerosMatriz regioesMatriz regioes
```

A função “avaliarNumeros” recebe uma posição “i j” e um número “num” para preenchê-la. Ela utiliza a função “numeroEhPossivel”, que verifica se o número atende às 3 regras do jogo. Caso seja um número válido, atualiza a matriz com o método “atualizarMatriz” e chama “kojun” para a próxima posição. Caso seja um número inválido, chama a função “avaliarNumeros” novamente com o próximo número.

O algoritmo começa testando o maior número possível da região e, em caso erro, subtrai 1 e repete o teste. Essa estratégia foi escolhida pois, segundo a regra 3, dois números verticalmente adjacentes da mesma região precisam estar em ordem decrescente de cima para baixo. Então, como os testes começam pela parte de cima da matriz, a probabilidade de acertar um número é maior começando do maior número.

Caso nenhum número seja válido, pelo menos um número que não faz parte da solução correta foi colocado em alguma posição da matriz de números, então a função retornará “False”. Assim, como o algoritmo é recursivo, voltará para a posição anterior, que tentará outro número e, se não conseguir também, retornará False, voltando para posição anterior e assim o processo se repete até que sejam encontrados números válidos para todas casas. Desse modo, utilizando o backtracking, que testa números e volta para posições anteriores caso um erro seja

descoberto, o algoritmo consegue eliminar muitas possibilidades inválidas, sendo muito mais eficiente do que uma solução de força bruta pura.

```haskell
avaliarNumeros :: Int -> Int -> Int -> Int -> [[Int]] -> [[Int]] -> [[(Int, Int)]] -> (Bool, [[Int]])
avaliarNumeros num maxNum i j numerosMatriz regioesMatriz regioes =
    -- Tentou todos números e não encontrou número válido
    if (num > maxNum) then
        (False, numerosMatriz)
    -- Avaliará o número "num"
    else
        -- Número é válido para posição i j
        if (numeroEhPossivel num i j numerosMatriz regioesMatriz regioes) then
            let matrizAtualizada = atualizarMatriz num i j numerosMatriz
                -- Tenta preencher a próxima posição da matriz, chamando a função "kojun"
                (resultado, matriz) = kojun i (j+1) (matrizAtualizada) regioesMatriz regioes
            in
                -- Se o resultado do teste da próxima posição é válido, a posição atual também é válida
                if (resultado) then
                    (resultado, matriz)
                -- Teste da próxima posição retornou inválido, tentar outro número para posição atual
                else
                    avaliarNumeros (num + 1) maxNum i j numerosMatriz regioesMatriz regioes
        -- Número não é válido, tentará o próximo número
        else
            avaliarNumeros (num + 1) maxNum i j numerosMatriz regioesMatriz regioes
```

Além disso, o código usa uma estrutura para facilitar a verificação das células pertencentes a uma região. Essa estrutura é uma lista de listas em que cada lista interna representa uma região. O index de cada lista interna é o identificador da região, determinado na matriz de regiões. Cada lista de região é composta por tuplas (i, j) que representam as posições que pertencem a aquela região.

Por exemplo, a seguinte matriz de regiões:

```haskell
matrizRegioes = [[0 ,0 ,1 ,1 ,1 ,2],
                 [3 ,3 ,3 ,3 ,3 ,2],
                 [4 ,5 ,5 ,5 ,3 ,6],
                 [4 ,4 ,4 ,5 ,6 ,6],
                 [7 ,7 ,8 ,9 ,9 ,9],
                 [10,10,8 ,8 ,9 ,9]]
```

Resulta na seguinte de lista de regiões:

```
[
[(0,0),(0,1)],
[(0,2),(0,3),(0,4)],
[(0,5),(1,5)],
[(1,0),(1,1),(1,2),(1,3),(1,4),(2,4)],
[(2,0),(3,0),(3,1),(3,2)],
[(2,1),(2,2),(2,3),(3,3)],
[(2,5),(3,4),(3,5)],
[(4,0),(4,1)],
[(4,2),(5,2),(5,3)],
[(4,3),(4,4),(4,5),(5,4),(5,5)],
[(5,0),(5,1)]
]
```

Essa estrutura é criada pela função “definirRegioes”, que cria uma lista com listas vazias e, com o método “atualizarRegioes”, as preenche com posições (i, j). O método “atualizarRegioes” utiliza a função do haskell “foldr” com a função “AtualizarRegiao” para colocar cada coordenada (i,j) da lista “coordenadas” na lista de região adequada dentro da lista “regioes”.

```haskell
definirRegioes :: [[Int]] -> Int -> Int -> [[(Int, Int)]]
definirRegioes regioesMatriz quantidadeRegioes tamanhoMatriz =
    let regioes = replicate quantidadeRegioes []
    in atualizarRegioes regioesMatriz regioes tamanhoMatriz

-- Cria uma lista com todas coordenadas (i, j) e, com a função "foldr" do haskell, pega cada posição (i, j)
-- e aplica a função "atualizarRegiao", usando a lista de regioes como valor inicial.
atualizarRegioes :: [[Int]] -> [[(Int, Int)]] -> Int -> [[(Int, Int)]]
atualizarRegioes regioesMatriz regioes tamanhoMatriz =
    let coordenadas = [(i, j) | i <- [0..tamanhoMatriz-1], j <- [0..tamanhoMatriz-1]]
    in foldr (atualizarRegiao regioesMatriz) regioes coordenadas

-- Adiciona uma posição (i, j) na lista de sua região. Depois adiciona a lista da região atualizada na lista
-- de regioes.
atualizarRegiao :: [[Int]] -> (Int, Int) -> [[(Int, Int)]] -> [[(Int, Int)]]
atualizarRegiao regioesMatriz (i, j) regioes =
    let idRegiao = regioesMatriz !! i !! j
        regiaoAtualizada = (i, j) : (regioes !! idRegiao)
    in take idRegiao regioes ++ [regiaoAtualizada] ++ drop (idRegiao + 1) regioes
```

Um exemplo de uso da estrutura é na função “numeroEhPossivel”, que avalia se o número “num” dado de parâmetro é válido na posição “i j”. A função acessa a lista de região “regiao” dentro da lista “regioes” a partir do identificador de região presente na posição “i j” da matriz “numerosMatriz”. Após isso, o método verificarRegiao analisa cada tupla (i, j) presente na lista “regiao” para verificar se existe algum número igual a “num” na região. Caso exista, “num” não é um número válido, pois viola a regra 1.

```haskell
-- Aplica as regras do kojun para verificar se "num" é válido na posição "i j". Caso todos os testes retornem
-- como True, a posição é válida, então realiza AND entre os resultados dos testes.
numeroEhPossivel :: Int -> Int -> Int -> [[Int]] -> [[Int]] -> [[(Int,Int)]] -> Bool
numeroEhPossivel num i j numerosMatriz regioesMatriz regioes =
    let regiao = regioes !! (regioesMatriz !! i !! j)
    in (verificarRegiao num regiao numerosMatriz) &&
       (verificarAdjacentes num i j numerosMatriz) &&
       (verificarCimaBaixo num i j numerosMatriz regioesMatriz)

-- Regra 1: uma região precisa ter todos os números de 1 a N sem repetição, sendo N a quantidade de células na região.
-- A função verifica se todos os números na casa são diferentes. Como o algoritmo já preenche as casas com números de
-- 1 a N, não é necessário verificcar se os números da casa estão nesse intervalo.
verificarRegiao :: Int  -> [(Int, Int)] -> [[Int]] -> Bool
verificarRegiao num regiao numerosMatriz =
    not (any (\(i, j) -> (numerosMatriz !! i !! j) == num) regiao)
```

5. Organização do grupo

O trabalho foi realizado de forma síncrona utilizando a ferramenta “LiveShare” do “Visual Studio Code”, que permite múltiplos usuários editarem o mesmo código em conjunto.

De modo geral, os integrantes escreveram, alteraram e discutiram a mesma função conjuntamente. No entanto algumas funções mais independentes foram feitas individualmente, como as funções que implementam cada uma das 3 regras do jogo: “verificarRegiao”, “verificarAdjacentes” e “verificarCimaBaixo”.

6. Dificuldades encontradas

A principal dificuldade encontrada foi manter o controle da recursividade da função principal “kojun” de modo que todas as possibilidades de execução fossem realizadas da maneira correta. O uso de recursão é muito efetivo, mas pode causar confusão, já que em alguns casos a ordem de execução não é muito intuitiva.

Além disso, os integrantes têm mais prática com programação imperativa do que funcional, então realizar algumas ações, que em outras linguagens podem ser feitas facilmente, causaram um pouco de dificuldade. Portanto, primeiro resolvemos o problema em python e depois transportamos a solução para haskell, que demandou várias alterações. Por exemplo, a solução em python utilizou vários loops “for”, que tiveram que ser adaptados no haskell. Em alguns casos foram criadas funções recursivas que recebem o iterador como argumento, já em outros casos foram utilizadas funções prontas do haskel como “foldr”.

7. Referências

- https://www.geeksforgeeks.org/introduction-to-backtracking-data-structure-and-algorithm-tutorials/
- https://www.janko.at/Raetsel/Kojun/index.htm
- https://www.askpython.com/python/examples/sudoku-solver-in-python