UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Universidade Federal de Santa Catarina- UFSC
Departamento de Informática e Estatística (INE)
Ciência da Computação
INE5416 - Paradigmas de Programação

Leonardo Lima Appio (21101963)
Fillipi Mangrich Costa de Souza (21202110)
Arthur Scarpatto Rodrigues (21200068)

Análise do problema

Kojun é um quebra-cabeça lógico solucionável por meio da técnica de backtracking, que envolve a experimentação metódica de possibilidades. As regras do jogo exigem que cada célula seja preenchida com um número, garantindo que números iguais não sejam adjacentes. Adicionalmente, é proibido ter valores repetidos dentro de um mesmo grupo e cada grupo deve organizar seus números em ordem decrescente na direção vertical, posicionando o maior número no topo e o menor na base.

Solução e estratégia da implementação

Tipos de dados: foram criados usando type para especificação do problema, sendo eles:

```haskell
-- definindo os tipos
type Value = Int
type Row a = [a]
type Matrix a = [Row a]
type Table = Matrix Value
type Choices = [Value]
```

O tabuleiro do jogo é estruturado como um tipo de dado específico e é representado na forma de uma matriz de valores do tipo *Value*. Cada célula do tabuleiro contém um valor, onde o zero indica uma célula vazia. Além disso, existe uma matriz separada para identificar os grupos dentro do tabuleiro.

```haskell
--8x8
values :: Table
values = [[0,0,0,6,5,3,0,0],
          [0,0,6,0,0,0,0,0],
          [2,4,0,0,3,0,0,3],
          [1,0,6,0,0,0,4,0],
          [0,3,0,6,0,0,0,4],
          [0,2,0,0,0,0,1,3],
          [0,0,5,0,0,1,0,0],
          [0,2,0,0,0,0,2,4]]
```

```haskell
groups :: Table
groups = [[1,2,3,4,4,4,4,4],
          [1,3,3,5,4,6,4,7],
          [3,3,3,5,8,6,9,9],
          [3,10,10,8,8,9,9,9],
          [10,10,11,13,17,17,18,18],
          [12,10,10,13,13,16,16,18],
          [12,10,13,13,15,15,18,18],
          [12,14,14,13,19,19,19,19]]
```

O algoritmo principal da solução se baseia em tentativa e erro (backtracking), inicialmente calculando estimativas com força bruta, mas usando análises lógicas para melhorar o desempenho e busca por uma solução do tabuleiro dado. A função principal da solução desenvolvida é *getSolution*, a qual recebe ambos os tabuleiros e retorna o tabuleiro com a solução encontrada, caso encontre.

```haskell
-- Recebe dois table e retorna um table
getSolution :: Table -> Table -> Table

-- função 'choices' vai gerar uma matriz de possíveis escolhas para cada célula do tabuleiro com base nos valores e nos grupos.
-- função 'reduceChoices' pega a matriz gerada e tenta reduzir as possibilidades em cada célula,
-- utilizando a lógica para evitar escolhas que já não são possíveis com base nas outras células na mesma coluna ou grupo.
-- funcao 'searchForSolution' vai aplicar o backtracking para verificar se a matriz gerada é uma solução válida, se não
-- gera novas matrizes de forma recursiva e sempre reduzindo as possibilidades.
getSolution values groups = head $ searchForSolution (reduceChoices (choices values groups) groups) groups
```

Inicialmente, são estabelecidas as opções possíveis para cada posição da matriz, formando listas que variam de 1 até o tamanho do grupo, que corresponde ao valor máximo que uma posição em um grupo pode assumir. Uma redução imediata das possibilidades ocorre já nesta fase, eliminando-se os valores que já estão presentes no grupo, uma vez que estes não podem se repetir.

```haskell
-- Gera uma matriz de escolhas possíveis para cada célula no tabuleiro.
-- Para células vazias (valor 0), as escolhas são todos os números de 1 até o tamanho do grupo,
-- excluindo números que já estão presentes no grupo.
choices :: Table -> Table -> Matrix Choices
choices values groups = map (map choice) (zipWith zip values groups)
    where choice (v, p) = if v == 0 then [1..(groupSize p groups)] `minus` (getValuesInGroup values groups p) else [v]
```

A função *reduceChoices* trabalha com uma matriz já preenchida com listas de valores possíveis. Esta função percorre os grupos verticalmente e, ao identificar listas que contenham apenas um valor, fixa esse valor na respectiva posição do tabuleiro. Outra função crucial é *searchForSolution*, que implementa o backtracking. Esta função inicia com a matriz mais reduzida possível, sem predefinir valores arbitrários na lista de escolhas. A partir daí, procede fazendo seleções aleatórias para encontrar uma solução, revertendo para passos anteriores sempre que a escolha feita resulta em uma incongruência com as regras do jogo.

```haskell
-- Reduz as escolhas disponíveis em cada célula com base nas escolhas nas colunas e grupos correspondentes.
-- Aplica uma redução coluna por coluna, ajustando as escolhas disponíveis para evitar conflitos.
-- A ideia é simplificar o espaço de busca antes do backtracking.
reduceChoices :: Matrix Choices -> Table -> Matrix Choices
reduceChoices values groups = cols $ originalCols (map reduceChoicesByList (groupsByColumn values groups)) (size values)
```

```haskell
-- Função principal de backtracking que busca por soluções válidas.
-- Avalia se o estado atual das escolhas é viável; se não for, retorna uma lista vazia.
-- Se todas as células têm uma única escolha, uma solução completa foi encontrada.
-- Se ainda existem múltiplas escolhas, a função continua a busca recursivamente com novas configurações geradas por `expandChoices`.
searchForSolution :: Matrix Choices -> Table -> [Table]
searchForSolution values groups
    | notPossible values groups = []  -- caso não tenha solução, retorna lista vazia
    | all (all singleElementInList) values = [map concat values]  -- não precisa de mais reduções
    | otherwise = [g | values' <- expandChoices values, g <- searchForSolution (reduceChoices values' groups) groups]  -- ainda precisa de redução.
```

O método *valid* é responsável por verificar a validade de uma matriz, assegurando que todas as regras estabelecidas para o jogo foram garantidas.

```haskell
-- Verifica se uma matriz de escolhas é válida com base em várias regras.
-- Validações incluem a verificação de vizinhos (nenhum par adjacente pode ter o mesmo valor),
-- valores dentro de um grupo (não devem repetir), e ordenamento nas colunas dentro de um grupo (devem ser descrescentes).
valid :: Matrix Choices -> Table -> Bool
valid values groups = all validNeighbour (cols values) &&
                      all validNeighbour (rows values) &&  -- nenhuma célula tem vizinhos com valor igual (em linha e coluna)
                      all validRow (matrixByGroup values groups) &&  --compara dentro do grupo para ver se não há valor igual
                      all descendingRow (groupsByColumn values groups)  -- deve haver ordenamento descrescente de cima para baixo nas colunas dentro de um grupo
```

Também implementamos um makefile a fim de facilitar a compilação e a remoção dos arquivos gerados.

```
all: build

build:
    ghc -o main main.hs matrixHelper.hs types.hs kojunSolver.hs

clean:
    find . \( -name "*.o" -o -name "*.hi" -o -name "main" \) -delete
```

Organização do grupo

O grupo se reuniu por meio do discord para a comunicação e entendimento do jogo. Foram encontradas soluções do jogo Sudoku em Haskell e nos baseamos nelas para a construção da solução por conta da similaridade entre os dois jogos (regras parecidas e podem ser resolvidos com a técnica de backtracking).

Dificuldades e resolução

As principais dificuldades que encontramos foram em relação a linguagem Haskell, que é muito diferente de todas as outras linguagens que já utilizamos.

Uma das dificuldades foi encontrar mecanismos para percorrer matrizes utilizando a linguagem Haskell. A adaptação dos algoritmos de Sudoku encontrados para um que proporcionasse a solução do Kojun podia se tornar complicada ao mudar a maneira como as matrizes eram iteradas.

A falta de estruturas de repetição com ‘while’ e ‘for’ dificultou bastante pois era complicado pensar em soluções sem essas estruturas.