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

Caio Prá Silva (21203773)
Lívia Corazza Ferrão (21202119)
Pedro Nack Martins (21200081)

Análise do problema

Kojun é um quebra-cabeça de lógica, o qual pode ser resolvido utilizando a técnica do backtracking, ou seja, de tentativa e erro. Entre as regras do jogo, constata-se a inserção de um número em cada campo, onde dois números idênticos não podem ser adjacentes, nenhum grupo pode possuir valores duplicados e cada grupo ordena verticalmente seus valores em ordem decrescente (com o maior no topo e o menor na base).

Solução e estratégia da implementação

Dados: foram criados tipos usando *type* para especificação do problema, sendo eles:

```
-- definições dos tipos
type Value = Int
type Row a = [a]
type Matrix a = [Row a]
type Table = Matrix Value
type Choices = [Value]
```

O tabuleiro foi também criado como um tipo de dado e é representado como uma matriz, nesse caso, uma matriz de valores do tipo *Value*. Para os valores em cada entrada no tabuleiro do jogo, há uma matriz, com zero representando que está em branco, bem como uma matriz para os grupos. Abaixo, segue exemplo para um tabuleiro seis por seis.

```
values6x6 :: Table
values6x6 = [[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]
            ]
```

```
groups6x6 :: Table
groups6x6 = [[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]
            ]
```

Algoritmo: 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. O método principal da solução desenvolvida é *getSolution*, o qual recebe ambos os tabuleiros e retorna o tabuleiro com a solução encontrada, caso encontre.

```haskell
-- encontra a solução para o tabuleiro (devolve a primeira encontrada)
getSolution :: Table -> Table -> Table
getSolution values groups = head (searchForSolution (reduceChoices (choices values groups) groups) groups)
```

Inicialmente são montadas as escolhas possíveis, em cada posição da matriz, são formadas listas que vão de 1 até o tamanho do grupo, sendo esse o valor máximo de uma posição em um grupo. Uma diminuição das possibilidades que já se pode fazer nessa etapa é remover os valores que estão no grupo, pois não irão se repetir.

```haskell
-- cria escolhas possíveis em cada célula
-- células sem valor possuem valores possíveis de 1 até tamanho do grupo
-- remove os valores que já estão naquele grupo (para não haver repetição)
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]
```

O método *reduceChoices*, com a matriz já com os valores das escolhas em lista, percorre os grupos por coluna e, onde encontrar apenas um valor na lista, fixa aquele valor na posição. Outro método importante é *searchForSolution*, que é responsável pelo backtracking. Recebe já a menor matriz possível sem assumir nenhum valor arbitrário dentro da lista de escolhas e, com isso, começa a fazer escolhas aleatórias para buscar uma solução, voltando para etapas anteriores caso o valor que tentou fixar gere uma inconsistência com as regras do jogo.

```haskell
-- filtra soluções baseada nas escolhas, percorrendo as células
-- cria uma lista com tabuleiros com possíveis soluções para o tabuleiro
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]
```

A verificação de validade de uma matriz, feita por *valid*, garante que as regras definidas para o jogo foram seguidas na construção daquela matriz.

```haskell
-- verifica se uma matriz de escolhas é valida
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
```

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

Uma das maiores 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.