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

Módulos: foram criados dois módulos. O primeiro “Matrix” para envolver as funções referentes a matriz e o segundo “Solver” para funções de maior complexidade que resolvem os problemas.

```
defmodule Matrix do
  @moduledoc """
    Módulo com métodos auxiliares para problemas com matrizes.
    Inclui a criação de tipos de dados que encapsulam listas e valores inteiros.
  """
```

```
defmodule Solver do
  @moduledoc """
    Módulo com funções responsáveis pela solução de tabuleiros do jogo Kojun
  """
```

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

```elixir
@typedoc """
  Tipo valor de cada posição (encapsula inteiros)
"""
@type value_t :: integer

@typedoc """
  Linhas das matrizes são listas de valores
"""
@type row_t :: [value_t]

@typedoc """
  Define o tipo Matrix, com linhas de um determinado tipo
"""
@type matrix_t :: [row_t]

@typedoc """
  Define um Table, que é matriz de int
"""
@type table_t :: matrix_t

@typedoc """
  Lista de valores possiveis para uma determinada célula
"""
@type choices_t :: [value_t]
```

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.

```elixir
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]
]
```

```elixir
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]
]
```

A saída é dada pela primeira matriz encontrada no método principal.

```elixir
[
  [2, 1, 3, 2, 1, 2],
  [1, 4, 2, 3, 6, 1],
  [4, 3, 4, 2, 5, 3],
  [3, 1, 2, 1, 2, 1],
  [1, 2, 3, 5, 4, 2],
  [2, 1, 2, 1, 3, 1]
]
```

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.

```elixir
@doc """
  Encontra a solução para o tabuleiro (devolve a primeira encontrada)
"""
@spec getSolution(Matrix.table_t(), Matrix.table_t()) :: Matrix.table_t()
def getSolution(values, groups) do
  Enum.at(searchForSolution(reduceChoices(choices(values, groups), groups), groups), 0)
end
```

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.

```elixir
@doc """
  Retorna a matriz de escolhas a partir dos valores e grupos
"""
@spec choices(Matrix.table_t(), Matrix.table_t()) :: Matrix.matrix_t()
def choices(values, groups) do
  Enum.map(Enum.zip(values, groups), fn {value, group} ->
    Enum.map(Enum.zip(value, group), fn {v, g} ->
      if v == 0 do
        Enum.to_list(1..groupSize(g, groups)) -- getValuesInGroup(values, groups, g)
      else
        [v]
      end
    end)
  end)
end
```

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.

```elixir
@doc """
  Cria uma lista com tabuleiros com possíveis soluções para o tabuleiro e por meio do backtracking filtra as escolhas
"""
@spec searchForSolution(Matrix.matrix_t(), Matrix.table_t()) :: [Matrix.table_t()]
def searchForSolution(values, groups) do
  cond do
    Solver.notPossible(values, groups) ->
      []

    Enum.all?(values, fn sublist -> Enum.all?(sublist, &singleElementInList/1) end) ->
      [values]

    true ->
      for values1 <- expandChoices(values),
          g <- searchForSolution(reduceChoices(values1, groups), groups),
          do: g
  end
end
```

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

```elixir
@doc """
  Testa todas regras do jogo
"""
@spec valid(Matrix.matrix_t(), Matrix.table_t()) :: bool
def valid(values, groups) do
  Enum.reduce(Matrix.cols(values), true, fn col, acc -> acc && validNeighbour(col) end) &&
    Enum.reduce(Matrix.rows(values), true, fn row, acc -> acc && validNeighbour(row) end) &&
    Enum.reduce(Matrix.matrixByGroup(values, groups), true, fn matrixGroup, acc ->
      acc && validRow(matrixGroup)
    end) &&
    Enum.reduce(Matrix.groupsByColumn(values, groups), true, fn groupColumn, acc ->
      acc && descendingRow(groupColumn)
    end)
end
```

Vantagens e desvantagens

A linguagem Elixir se mostrou mais intuitiva quanto a forma que as funções e operações funcionam, bem como de mais fácil resolução para o problema. Entretanto, para tratar os erros, as mensagens de erro obtidas não são claras e por vezes insuficientes para compreender o que precisa ser corrigido, não contendo muitas vezes traceback das funções chamadas.

Organização do grupo

O grupo se reuniu por meio do discord para a comunicação e entendimento do jogo. Nos baseamos na solução implementada em Haskell da entrega 1 para a construção do jogo em Elixir. As funções foram sendo implementadas gradualmente pelos membros do grupo e avaliadas e corrigidas pelos outros, quando necessário.

Dificuldades e resolução

Encontramos dificuldades em traduzir algumas funções nativas do Haskell para o Elixir, já que algumas delas não eram nativamente implementadas nessa linguagem. Nesses casos, verificamos o funcionamento das mesmas e as implementamos dentro das funções necessárias.