**UNIVERSIDADE FEDERAL DE SANTA CATARINA**
**DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA**
**CURSO DE CIÊNCIAS DA COMPUTAÇÃO**

**ALEX DAVIS NEUWIEM DA SILVA**
**LUAN DINIZ MORAES**
**LUCAS CASTRO TRUPPEL MACHADO**

# Trabalho III - Programação de Restrições - Prolog
# Programa Solucionador do jogo Kojun

**Florianópolis, 2023.**

## 1. Introdução

O puzzle escolhido a ser resolvido é o Kojun. O código foi feito em Prolog, utilizando o backtracking próprio da linguagem para obter a solução.

Esse puzzle foi o que escolhemos para ser resolvido com a linguagem Haskell no Trabalho I e com a linguagem Elixir no Trabalho II.

## 2. Análise do Problema

No Kojun, o jogador recebe inicialmente um tabuleiro dividido em regiões com a maioria das posições vazias e algumas delas já preenchidas por números. O objetivo do jogo é completar esse tabuleiro com os números faltantes seguindo 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 requer duas matrizes como entrada, sendo uma delas a matriz de números iniciais, com 0 representando as posições vazias, e a outra a matriz de regiões, com um ID de região começando em 0. Já a saída é impressa no terminal, podendo ser tanto a matriz resolvida quanto “false”, caso não haja solução.

O arquivo “docs/exemploEntrada.txt” possui 3 exemplos de entrada que podem ser copiados em “prolog/kojun.pl”.

## 4. Descrição da Solução

A estratégia utilizada na solução foi utilizar o próprio backtracking do Prolog para criar uma solução genérica, que resolva qualquer jogo Kojun com base nas duas matrizes de entrada.

Para fazer isso, utilizou-se duas operações principais: “buscarMatriz()” e “atualizarMatriz()”. A imagem a seguir apresenta a implementação dessas duas operações.

```
/* A operação "nth0()" pode retornar o enésimo valor de uma lista */
/* Como as matrizes utilizadas nessa solução são listas de listas, "buscarMatriz()" nos retorna o valor desejado */
buscarMatriz(Matriz, Linha, Coluna, Valor) :- nth0(Linha, Matriz, Lista), nth0(Coluna, Lista, Valor).

/* Percorre a lista até Posicao - 1 (length), atualiza o valor e armazena em Lista2 */
atualizarPosicao(Posicao, ValorAntigo, ValorNovo, Lista1, Lista2) :-
    length(Pos, Posicao),
    append(Pos, [ValorAntigo|Resto], Lista1),
    append(Pos, [ValorNovo|Resto], Lista2).

/* Faz o mesmo que a função anterior, mas em uma lista de listas */
atualizarMatriz(Matriz, I, J, ValorNovo, NovaMatriz) :-
    atualizarPosicao(I, Antigo, Novo, Matriz, NovaMatriz),
    atualizarPosicao(J, _Antigo, ValorNovo, Antigo, Novo).
```

“buscarMatriz()” nos permite tanto obter um valor em uma determinada posição da matriz quanto varrer uma matriz percorrendo os seus valores. Já “atualizarMatriz()” retorna uma matriz com o valor atualizado na posição indicada.

Utilizando “buscarMatriz()”, foi possível percorrer uma determinada região e obter todos os seus valores. Esse foi um passo importante, pois tendo os números, podemos saber quais são os valores que faltam para preencher uma região.

A seguir está a implementação de “listaNumerosRegiao()”, que retorna uma lista com todos os números da região especificada e “listaComplemento()”, que retorna uma lista com os valores que complementam uma lista de números.

```
/* Retornando todos os números presentes em uma região com base em um ID de entrada */
buscarNumerosRegiao(Matriz, IdRegiao, Valor) :-
    matrizRegioes(MatrizRegioes),
    buscarMatriz(MatrizRegioes, I, J, IdRegiao),
    buscarMatriz(Matriz, I, J, Valor).

/* Utilizando "findall()" para gerar uma lista que contenha todos os números de uma determinada região */
listaNumerosRegiao(Matriz, IdRegiao, ListaNumerosRegiao) :-
    findall(Valor, buscarNumerosRegiao(Matriz, IdRegiao, Valor), ListaNumerosRegiao).

/* Retornando uma lista com os números que completam uma região */
/* Exemplo: seja Lista = [1, 0, 5, 2, 0], o retorno será [3, 4] */
/* Em uma região completa, o retorno é [] */
listaComplemento(Lista, ListaComplemento) :-
    length(Lista, Tamanho),
    numlist(1, Tamanho, ListaTotal),
    delete(Lista, 0, ListaResto),
    subtract(ListaTotal, ListaResto, ListaComplemento).
```

Agora que temos uma lista com todos os números necessários para preencher uma região, precisamos saber quais as células livres da região que serão ocupadas por esses valores.

A seguir, temos a implementação de “listaCoordenadasLivres()”, que retorna uma lista que contém todas as posições vazias de uma determinada região. Esta implementação é semelhante à da operação “listaNumerosRegiao()”, porém retorna as coordenadas que possuem valor 0 na matriz inicial de números.

```
/* Retornando as coordenadas de uma região que possuem o valor 0 na matriz de números */
buscarCoordenadasLivres(Matriz, IdRegiao, [I, J]) :-
    matrizRegioes(MatrizRegioes),
    buscarMatriz(MatrizRegioes, I, J, IdRegiao),
    buscarMatriz(Matriz, I, J, 0).

/* Utilizando "findall()" para gerar uma lista que contenha todas as coordenadas livres de uma determinada região */
listaCoordenadasLivres(Matriz, IdRegiao, ListaCoordenadasLivres) :-
    findall([I, J], buscarCoordenadasLivres(Matriz, IdRegiao, [I, J]), ListaCoordenadasLivres).
```

Nesse ponto, temos os dois principais elementos de nossa solução: uma lista que contém todos os números que são necessários para preencher uma região e uma lista que contém todas as posições vazias de uma determinada região. Agora, podemos percorrer a matriz inicial inteira e preencher cada posição seguindo as regras do jogo.

A seguir, temos uma imagem da implementação da operação “preencherPosicao()”, que retorna uma matriz com um valor válido na posição especificada.

```
/* Dado uma matriz e uma coordenada, retorna um valor válido para essa posição segundo as regras do jogo */
/* Com o valor válido, gera uma nova matriz de números e a retorna */
/* Caso a posição já esteja preenchida, retorna a própria matriz de entrada */
preencherPosicao(Matriz, I, J, NovaMatriz) :-
    verificarCoordenadas(Matriz, I, J) ->
    verificarMembro(Matriz, I, J, Valor),
    verificarCima(Matriz, I, J, Valor),
    verificarBaixo(Matriz, I, J, Valor),
    verificarEsquerda(Matriz, I, J, Valor),
    verificarDireita(Matriz, I, J, Valor),
    atualizarMatriz(Matriz, I, J, Valor, NovaMatriz);
    NovaMatriz = Matriz.
```

Por fim, só nos resta percorrer a matriz de números aplicando a operação “preencherPosicao()” em cada uma das posições. O código abaixo apresenta um exemplo da implementação em uma matriz 6x6.

```
kojun() :-

    matrizNumerosInicial(MNI),

    preencherPosicao(MNI, 0, 0, M00),
    preencherPosicao(M00, 0, 1, M01),
    preencherPosicao(M01, 0, 2, M02),
    preencherPosicao(M02, 0, 3, M03),
    preencherPosicao(M03, 0, 4, M04),
    preencherPosicao(M04, 0, 5, M05),

    preencherPosicao(M05, 1, 0, M10),
    preencherPosicao(M10, 1, 1, M11),
    preencherPosicao(M11, 1, 2, M12),
    preencherPosicao(M12, 1, 3, M13),
    preencherPosicao(M13, 1, 4, M14),
    preencherPosicao(M14, 1, 5, M15),
```

```
    preencherPosicao(M15, 2, 0, M20),
    preencherPosicao(M20, 2, 1, M21),
    preencherPosicao(M21, 2, 2, M22),
    preencherPosicao(M22, 2, 3, M23),
    preencherPosicao(M23, 2, 4, M24),
    preencherPosicao(M24, 2, 5, M25),

    preencherPosicao(M25, 3, 0, M30),
    preencherPosicao(M30, 3, 1, M31),
    preencherPosicao(M31, 3, 2, M32),
    preencherPosicao(M32, 3, 3, M33),
    preencherPosicao(M33, 3, 4, M34),
    preencherPosicao(M34, 3, 5, M35),
```

```
    preencherPosicao(M35, 4, 0, M40),
    preencherPosicao(M40, 4, 1, M41),
    preencherPosicao(M41, 4, 2, M42),
    preencherPosicao(M42, 4, 3, M43),
    preencherPosicao(M43, 4, 4, M44),
    preencherPosicao(M44, 4, 5, M45),

    preencherPosicao(M45, 5, 0, M50),
    preencherPosicao(M50, 5, 1, M51),
    preencherPosicao(M51, 5, 2, M52),
    preencherPosicao(M52, 5, 3, M53),
    preencherPosicao(M53, 5, 4, M54),
    preencherPosicao(M54, 5, 5, M55),

    nl, imprimirMatriz(M55), nl.
```

Note que, como o Prolog é uma linguagem de programação de restrições, é possível obter um resultado que satisfaça nossas regras impostas sem ter que “chutar” valores, pois todo o backtracking já é implementado na linguagem. Com isso, o resultado final é uma matriz solucionada completamente preenchida que será impressa na tela.

## 5. Prolog vs Elixir

Tanto o paradigma funcional, representado pelo Elixir, quanto o paradigma lógico, representado pelo Prolog, apresentaram vantagens e desvantagens na implementação de um resolvedor do jogo Kojun.

O Elixir, com sua abordagem funcional, permitiu uma escrita modular e flexível por meio da composição de funções. A velocidade de execução e a variedade de funções prontas contribuíram para o êxito do projeto. No entanto, há a necessidade de implementar o algoritmo de backtracking, o que não é preciso fazer quando usamos o Prolog.

Já o Prolog, com sua abordagem lógica baseada em regras e fatos, ofereceu um sistema de inferência poderoso e declarativo. A programação em Prolog permitiu expressar uma lógica complexa, sendo especialmente adequada para problemas que envolvam dedução e busca. No entanto, o desempenho pode ser um desafio, uma vez que a execução do Prolog se baseia na busca exaustiva de soluções e

acaba sendo lenta para problemas de grande escala, como é o caso da matriz de dimensões 17x17.

Portanto, a escolha entre Elixir e Prolog depende das características do puzzle em questão e das necessidades específicas do projeto. Em relação à implementação, o Prolog se provou incrivelmente eficiente com seu paradigma de restrições, porém o Elixir apresenta uma execução com o mais alto desempenho.

## 6. Organização do Grupo

O trabalho foi realizado tanto de forma síncrona, utilizando a extensão “LiveShare” do “Visual Studio Code”, quanto de forma assíncrona com o uso do GitHub.

De modo geral, todos os membros participaram do desenvolvimento do código: discutindo sobre uma mesma ideia em conjunto ou implementando algumas partes da solução separadamente.

## 7. Dificuldades Enfrentadas

A principal dificuldade encontrada pelo grupo foi fazer com que a parte final de percorrer a matriz aplicando “preencherPosicao()” em cada célula da matriz funcionasse. A dificuldade estava em tentar fazer uma função recursiva que fizesse isso, sem ter que escrever uma linha de código para cada elemento da matriz, porém a equipe não obteve êxito nessa tentativa de implementação.

Fora isso, houveram dúvidas recorrentes quanto às características da programação por restrições, mas elas foram sendo resolvidas por meio da colaboração entre os integrantes do grupo e do trabalho em equipe.