

Universidade Federal de Santa Catarina

Departamento de Informática e Estatística

Ciência da Computação

INE5416 – Paradigmas de programação

Professor: Maicon Rafael Zatelli

Equipe: Anthony Bernardo Kamers e Eduardo Willwock Lussi

### **Relatório - Trabalho III**

Resolvedor de puzzle Kojun na linguagem Prolog

Florianópolis, setembro de 2021

## Descrição do trabalho

Para o trabalho, precisamos fazer um programa na linguagem lógica Prolog de modo a resolver um puzzle japonês chamado Kojun. Neste jogo, há um tabuleiro de tamanho  $N \times N$  com vários grupos (dependendo do tabuleiro). Tal como no famoso jogo Sudoku, não se pode repetir números no mesmo grupo, nem de maneira adjacente. Além disso, se, dentro de um grupo, os campos estiverem de maneira verticalmente adjacentes, o maior valor deve ser posto em cima. Um exemplo de tabuleiro de jogo e solução para o mesmo estão nas imagens a seguir:

	1	3					
					3		
		3					
	5		3				
	2						
						3	
		5	3				

Exemplo de tabuleiro do jogo

4	2	1	3	1	2	1	3
3	1	3	2	3	1	3	2
2	4	1	6	2	3	1	5
1	2	3	4	1	2	5	4
2	5	1	3	2	1	4	3
1	2	3	2	1	5	1	2
3	1	4	5	2	4	3	1
1	4	5	3	1	2	1	2

Exemplo de solução do jogo

## Programação de restrições

Programação de restrições em linguagens lógicas, como o Prolog, são maneiras eficientes de resolver problemas lógicos. De forma simplificada, são informadas diretrizes para o programa das quais são interpretadas como restrições para a solução final (dada uma gama de opções) e, por meio de um backtracking automático, são geradas uma ou mais soluções para o problema informado.

Para fazer o resolvidor de Kojun, são informadas, basicamente, as regras do jogo em funções lógicas, seguindo as diretrizes da biblioteca *clpfd*, específica de programação de restrições. Desta forma, o programa encontra a única solução

possível. Um exemplo a ser citado no nosso resolvidor, de modo a ilustrar a programação de restrições, são as funções *orthogonalAdjacent* e *orthogonalAdjacentLine*, em que informamos que os elementos nas ortogonais adjacentes devem ser todas distintas (de acordo com as regras do jogo), como na imagem abaixo.

```
142 % Filtragem pela regra dos ortogonais adjacentes
143 orthogonalAdjacentLine([]).
144 orthogonalAdjacentLine([_ | []]).
145 orthogonalAdjacentLine([E0, E1 | RestEl]) :- all_distinct([E0, E1]), orthogonalAdjacentLine([E1 | RestEl]).
146
147 orthogonalAdjacent([]).
148 orthogonalAdjacent([MatrixLine | RestMatrix]) :- orthogonalAdjacentLine(MatrixLine), orthogonalAdjacent(RestMatrix).
```

Exemplo de trecho de código usando a programação orientada a restrições

## Vantagens e desvantagens programação de restrições

Primeiramente, deve-se salientar que, por se tratar de uma linguagem lógica, não estamos acostumados a pensar conforme a programação de restrições porque estamos mais habituados com linguagens de programação estruturada e orientada a objetos, tornando a tarefa de estruturar as soluções um pouco mais complicada. Entretanto, conforme evolui o aprendizado da linguagem e de como as restrições devem ser postas no código, naturalmente se torna mais fácil.

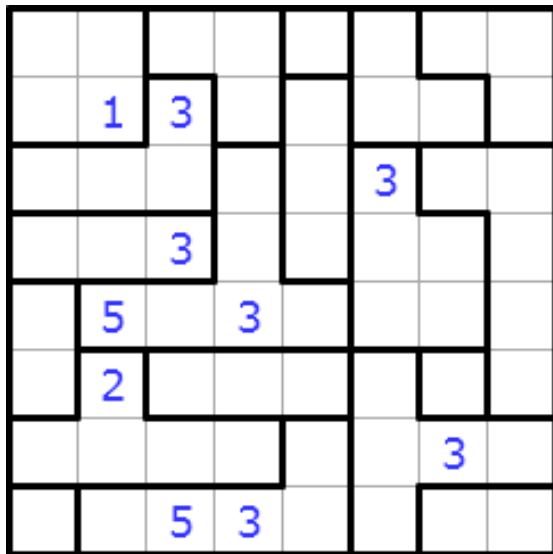
Outra dificuldade identificada é sobre a quantidade de filtros (restrições) que deveríamos fazer para que o problema fosse resolvido, pois não sabíamos se apenas colocando as regras do jogo, o problema já seria resolvido automaticamente.

Mas também encontramos várias comodidades, como a redução considerável de código e a facilidade em ter que informar apenas quais restrições o programa deve considerar para o puzzle, sem a necessidade de criar métodos para a realização do *backtracking*, *overlap*, *collapse* e verificador de solução.

## Entrada e saída de dados

Para informar o tabuleiro, deve-se colocar instâncias de cada tabuleiro no código, com nome *problem*, seguindo a estrutura: id do tabuleiro, matriz do tabuleiro (informando os números que já vêm com o tabuleiro e com `_`, caso esteja vazio) e a

matriz dos grupos (com números identificados para cada grupo, à começar de 0). Segue abaixo exemplo de um tabuleiro e de como deve ser feita a entrada de dados no código do programa:



```
problem(1, [
    [_,_,_,_,_,_,_,_,_,_],
    [_,1,3,_,_,_,_,_,_],
    [_,_,_,_,3,_,_,_,_],
    [_,_,3,_,_,_,_,_,_],
    [_,5,_,3,_,_,_,_,_],
    [_,2,_,_,_,_,_,_,_],
    [_,_,_,_,_,3,_,_,_],
    [_,_,5,3,_,_,_,_,_],
    [0,0,1,1,2,3,4,4,4],
    [0,0,5,1,6,3,3,4,4],
    [5,5,5,7,6,8,9,9,9],
    [10,10,10,7,6,8,8,9,9],
    [11,7,7,7,7,8,8,9,9],
    [11,12,13,13,13,14,15,9,9],
    [12,12,12,12,16,14,14,14,14],
    [17,16,16,16,16,14,18,18,18]])
```

Para executar o tabuleiro de sua preferência, deve ser informada na linha 7, na função inicializadora *main*, apenas o id dado para a instância do puzzle, como na imagem abaixo:

```
7 main :- problem(1, Matrix, Groups),
8         kojun(Matrix, Groups),
9         print_matrix(Matrix).
```

Dessa forma, fica facilitado para o usuário, quando executar o *swipl* e importar o caminho do arquivo ou, simplesmente rodando o arquivo *.pl*, a resposta já é gerada automaticamente. Além disso, a resposta é dada em formato de matriz (número a número). Seguem abaixo imagens de como deveria ser a resposta de um puzzle e a nossa respectiva representação:

4	2	1	3	1	2	1	3
3	1	3	2	3	1	3	2
2	4	1	6	2	3	1	5
1	2	3	4	1	2	5	4
2	5	1	3	2	1	4	3
1	2	3	2	1	5	1	2
3	1	4	5	2	4	3	1
1	4	5	3	1	2	1	2

## Vantagens / Desvantagens: paradigma funcional e lógico

Algumas vantagens de trabalhar com linguagens de paradigma funcional como Haskell e Lisp, são, principalmente, a facilidade de trabalhar com algo que já é feito em outras linguagens procedurais. Dessa forma, fica mais fácil de pensar na solução. Além disso, há vários atalhos para trabalhar com listas e matrizes, sendo que a linguagem Haskell, por exemplo, foi feita especificamente para trabalhar com esse propósito. Uma desvantagem para fazer um resolvidor de puzzle, por exemplo, é ter que programar manualmente mecanismos de *backtracking*.

Ao se tratar de paradigma lógico e, mais especificamente com programação de restrições, temos como vantagem a quantidade de código reduzida, pois são apenas colocadas as “afirmações” sobre o programa e é gerada a(s) solução(ões). Com isso, só são testadas as alternativas que satisfazem as condições impostas. Como desvantagem, vale salientar a dificuldade de se acostumar com a ideia do paradigma, algumas vezes é difícil transmitir ao código uma ideia que está clara para o programador, além da necessidade de criar algumas funções complementares para fazer coisas “elementares” em outras linguagens, mas essa pode ser uma avaliação injusta porque o paradigma lógico nos livra da necessidade de codificar vários outros mecanismos.

## Organização do grupo

Como havia várias tarefas para fazer o desenvolvimento do trabalho, foi criado um repositório privado no github com os integrantes, para colaborarem simultaneamente no projeto. Cada um dos conjuntos de funções foi dividido de tal forma que ambos tivessem um entendimento total do código. Houveram várias dúvidas durante o projeto, sendo que o grupo sempre entrava em contato, para sanar da melhor maneira possível, por mensagem de texto ou mesmo por ligação de vídeo e voz.

## Dificuldades encontradas

A parte que mais precisou atenção foram os filtros (para computar em tempo hábil a solução adequada), pois são eles que contém as afirmações que a programação de restrições usa para chegar a uma solução. Além disso, aprender sobre a biblioteca *clpfd* também foi bastante complicado, tal como encontrar qual é a maneira correta de se implementar cada regra do jogo como restrições.

Quanto à linguagem Prolog, além do fato de se tratar de uma linguagem lógica, onde a maneira de pensar e de fazer as funções podem ser mais difíceis do que em linguagens funcionais, por exemplo, a sua sintaxe é bastante distinta do que estamos acostumados com relação outras linguagens mais corriqueiras, como o python, Java e C.