UNIVERSIDADE FEDERAL DE SANTA CATARINA

CAMPUS TRINDADE

INE-DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

INE5416 - PARADIGMAS DE PROGRAMAÇÃO

Alunos:

João Victor Cabral Machado — **22204113**

Pedro Costa Casotti — **22200374**

Relatório do Trabalho 3 - Puzzle Kojun

Florianópolis

2024

1 - Introdução

O puzzle Kojun é um desafio de lógica que requer preencher um tabuleiro com números, garantindo que cada número apareça apenas uma vez por região e siga certas restrições de proximidade. O tabuleiro é composto por várias regiões não retangulares, cada uma contendo células preenchidas com números de 1 ao total de células na região. A resolução desse puzzle é complexa e fascinante, especialmente para algoritmos de resolução.

2 - Descrição da Solução

2.1 - Modelagem do Tabuleiro

O tabuleiro do puzzle Kojun é representado por uma lista de listas em Prolog. Cada célula pode conter um número de 1 a N, onde N é o tamanho máximo de células em uma região específica.

```
% Implementação do tabuleiro e das regiões
tabuleiro([
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 3, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 3, 0, 0],
    [0, 0, 3, 0, 0, 0, 0, 0],
    [0, 5, 0, 3, 0, 0, 0, 0],
    [0, 2, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 3, 0],
    [0, 0, 5, 3, 0, 0, 0, 0]
]).

regiões([
    (1, (0, 0)), (1, (0, 1)), (1, (1, 0)), (1, (1, 1)),
    (2, (0, 2)), (2, (0, 3)), (2, (1, 3)),
    (3, (0, 4)),
    (4, (0, 5)), (4, (1, 5)), (4, (1, 6)),
    (5, (0, 6)), (5, (0, 7)), (5, (1, 7)),
    (6, (1, 2)), (6, (2, 0)), (6, (2, 1)), (6, (2, 2)),
    (7, (1, 4)), (7, (2, 4)), (7, (3, 4)),
    (8, (2, 3)), (8, (3, 3)), (8, (4, 1)), (8, (4, 2)), (8, (4,
3)), (8, (4, 4)),
    (9, (2, 5)), (9, (3, 5)), (9, (3, 6)), (9, (4, 5)), (9, (4,
6)),
    (10, (2, 6)), (10, (2, 7)), (10, (3, 7)), (10, (4, 7)),
(10, (5, 7)),
    (11, (3, 0)), (11, (3, 1)), (11, (3, 2)),
    (12, (4, 0)), (12, (5, 0)),
    (13, (5, 1)), (13, (6, 0)), (13, (6, 1)), (13, (6, 2)),
(13, (6, 3)),
```

```
    (14, (5, 2)), (14, (5, 3)), (14, (5, 4)),
    (15, (5, 5)), (15, (6, 5)), (15, (6, 6)), (15, (6, 7)),
(15, (7, 5)),
    (16, (5, 6)),
    (17, (6, 4)), (17, (7, 1)), (17, (7, 2)), (17, (7, 3)),
(17, (7, 4)),
    (18, (7, 0)),
    (19, (7, 6)), (19, (7, 7))
]).
```

2.2 - Algoritmo do Backtracking

A solução utiliza um algoritmo de backtracking para preencher recursivamente o tabuleiro. A função *resolve* tenta preencher cada célula vazia com um número válido, respeitando as restrições do puzzle.

A função resolve é a função principal que inicia o processo de resolução do tabuleiro. Ela recebe como entrada o tabuleiro atual, a lista de regiões e retorna o tabuleiro final resolvido ou indica que não há solução possível.

```
resolve(Tabuleiro, Regioes, TabuleiroFinal) :-
    ( celula_vazia(Tabuleiro, CelulaVazia) ->
          resolve_celula(Tabuleiro, Regioes, CelulaVazia,
TabuleiroFinal)
    ; TabuleiroFinal = Tabuleiro
    ).
```

celula_vazia: Verifica se há alguma célula no tabuleiro que ainda não foi preenchida (contém 0).

resolve_celula: Se há uma célula vazia, resolve_celula é chamada para tentar preencher essa célula com um número válido.

A função *resolve_celula* é responsável por tentar atribuir um número válido a uma célula vazia do tabuleiro. Ela utiliza o predicado numero_valido/4 para verificar se um número pode ser atribuído à célula sem violar as regras do Kojun.

```
resolve_celula(Tabuleiro, Regioes, (I, J), TabuleiroFinal)
:-
    length(Tabuleiro, N),
    between(1, N, Num),
    numero_valido(Tabuleiro, Regioes, (I, J), Num),
    substitui(Tabuleiro, (I, J), Num, NovoTabuleiro),
    resolve(NovoTabuleiro, Regioes, TabuleiroFinal).
```

numero_valido: Verifica se um número pode ser colocado em uma determinada célula sem violar as regras do Kojun, incluindo regras de região e adjacências.

substitui: Substitui o valor na célula do tabuleiro, criando um novo tabuleiro com a célula atualizada.

3 - Entrada e Saída

A entrada do puzzle é definida estaticamente no código. A saída é apresentada no console, mostrando o tabuleiro completamente resolvido ou uma mensagem indicando que não foi encontrada uma solução.

4 - Desafios e Soluções

Desafio 1: Eficiência do Backtracking

Problema: Backtracking pode ser extremamente lento se o espaço de busca não for adequadamente restrito, especialmente em puzzles complexos com muitas possibilidades.

Solução: Para melhorar a eficiência, foram implementadas verificações para validar se um número pode ser colocado em uma célula, considerando as restrições de região e adjacências.

Desafio 2: Complexidade de Implementação

Problema: Gerenciar o estado mutável e a complexidade das regras de validação pode tornar o código difícil de entender e manter.

Solução 2: A complexidade da validação das regras do Kojun foi gerenciada com estruturas de dados claras e funções modulares, facilitando a manutenção e entendimento do código.

5 - Conclusões

A implementação em Prolog do resolvedor para o puzzle Kojun demonstrou a aplicação efetiva de técnicas de lógica e restrição. As otimizações realizadas permitiram resolver eficientemente puzzles dentro das limitações estipuladas. Este trabalho também destacou a importância da colaboração em equipe, onde diferentes contribuições foram integradas para alcançar uma solução robusta.

O trabalho foi feito de maneira distribuída entre os membros do grupo, Pedro teve foco na solução inicial enquanto João teve foco na otimização da solução do problema.