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

INE5416 – PARADIGMAS DE PROGRAMAÇÃO

PEDRO AUGUSTO DA FONTOURA (22215098)
ERIK ORSOLIN DE PAULA (22102195)
RAFAEL CORREA BITENCOURT (22203673)

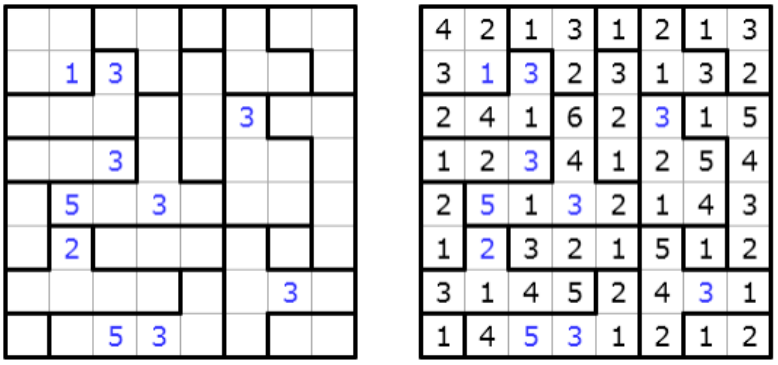
TRABALHO 3 DE PARADIGMAS DE PROGRAMAÇÃO

FLORIANÓPOLIS

2024

KOJUN

O jogo consiste em preencher todas as regiões de um tabuleiro, que pode variar de 6x6 até 17x17, com números de 1 a n, sem repetição, onde n é o tamanho da região a ser preenchida. Números adjacentes não podem ser iguais e, se estiverem na mesma região, o número diretamente acima deve ser maior do que o abaixo dele.



RESUMO DO ALGORITMO

O algoritmo em questão emprega a técnica de retrocesso (backtracking), onde valores são atribuídos a cada célula, e as regras do jogo são verificadas para garantir a conformidade. Para tal propósito, foram utilizadas as funções valor_maximo_regiao e vizinhos_diferentes. Essas funções são responsáveis pelas restrições do problema nas posições dos números.

```
:- use_module(library(clpfd)).

% Definição do tabuleiro com valores iniciais e regiões
tabuleiro([[[1,2],[1,_],[2,_],[2,_],[2,1],[3,_]],
           [[4,_],[4,_],[4,_],[4,3],[4,_],[3,_]],
           [[5,_],[6,3],[6,_],[6,_],[4,5],[7,3]],
           [[5,_],[5,_],[5,_],[6,_],[7,_],[7,_]],
           [[8,_],[8,_],[10,3],[0,_],[0,4],[0,2]],
           [[9,_],[9,_],[10,_],[10,_],[0,_],[0,_]]]).


tamanho_regiao(0,5).
tamanho_regiao(1,2).
tamanho_regiao(2,3).
tamanho_regiao(3,2).
tamanho_regiao(4,6).
tamanho_regiao(5,4).
tamanho_regiao(6,4).
tamanho_regiao(7,3).
tamanho_regiao(8,2).
tamanho_regiao(9,2).
tamanho_regiao(10,3).

% Definir o valor máximo que os valores de cada região podem assumir
valor_maximo_regiao([R,X]) :-
    tamanho_regiao(R,T),  % Obtém o tamanho da região
    X in 1..T.  % Define o domínio de X entre 1 e o tamanho da região
```

```
% Verificar se o vizinho à direita é diferente
vizinhos_diferentes([[_,_]]).
vizinhos_diferentes([[_,X1],[R2,X2]|T]) :-
    X1 #\= X2,  % Garante que os valores vizinhos sejam diferentes
    append([[R2,X2]],T,L),  % Cria uma nova lista com o vizinho atual e o restante
    vizinhos_diferentes(L).  % Chama recursivamente para o restante da lista

% Verificar se o valor acima de outro é maior, se eles fizerem parte do mesmo grupo
maior_acima([[_,_]]).
maior_acima([[R1,X1],[R2,X2]|T]) :-
    ((R1 #\= R2);  % Se os grupos são diferentes
    (X1 #> X2)),  % Ou se o valor de cima é maior que o valor de baixo
    append([[R2,X2]],T,L),  % Cria uma nova lista com o valor atual e o restante
    maior_acima(L).  % Chama recursivamente para o restante da lista

% Agrupar os elementos do mesmo grupo (recebe o grupo)
agrupar(_, [], []).
agrupar(R, [[R1, X1] | T], [X1 | L]) :-
    R #= R1,  % Se o grupo atual é igual ao grupo R
    agrupar(R, T, L).  % Chama recursivamente para o restante da lista
agrupar(R, [[R1, _] | T], L) :-
    R #\= R1,  % Se o grupo atual é diferente do grupo R
    agrupar(R, T, L).  % Chama recursivamente para o restante da lista

% Verificar se os membros de uma lista são diferentes
todas_regioes_distintas([H]) :-
    all_distinct(H).  % Garante que todos os elementos da lista são distintos
todas_regioes_distintas([H|T]) :-
    all_distinct(H),  % Garante que todos os elementos da lista são distintos
    todas_regioes_distintas(T).  % Chama recursivamente para o restante das listas

% Obter o segundo elemento de cada matriz
extrair([], []).
extrair([[_, Segundo] | Resto], [Segundo | ValoresSegundo]) :-
    extrair(Resto, ValoresSegundo).  % Chama recursivamente para o restante da lista
```

```
% Extrair os valores de cada linha da matriz
extrair_valores([], []).
extrair_valores([Sublista | Resto], [ValoresSegundo | Resultado]) :-
    extrair(Sublista, ValoresSegundo),  % Extrai os valores da sublista
    extrair_valores(Resto, Resultado).  % Chama recursivamente para o restante da lista

% Solucionador do problema
solucionador(Problema) :-
    append(Problema, Lista),  % Cria uma lista com todos os elementos do problema
    maplist(valor_maximo_regiao, Lista),  % Define os valores máximos para cada região
    maplist(vizinhos_diferentes, Problema),  % Verifica vizinhos na linha
    transpose(Problema, Colunas),  % Transpõe a matriz para verificar colunas
    maplist(vizinhos_diferentes, Colunas),  % Verifica vizinhos na coluna
    maplist(maior_acima, Colunas),  % Verifica se os valores acima são maiores
    agrupar(0, Lista, Grupo0),
    agrupar(1, Lista, Grupo1),
    agrupar(2, Lista, Grupo2),
    agrupar(3, Lista, Grupo3),
    agrupar(4, Lista, Grupo4),
    agrupar(5, Lista, Grupo5),
    agrupar(6, Lista, Grupo6),
    agrupar(7, Lista, Grupo7),
    agrupar(8, Lista, Grupo8),
    agrupar(9, Lista, Grupo9),
    agrupar(10, Lista, Grupo10),
    Grupos = [Grupo0,Grupo1,Grupo2,Grupo3,Grupo4,Grupo5,Grupo6,Grupo7,Grupo8,Grupo9,Grupo10],
    todas_regioes_distintas(Grupos),  % Verifica se todos os grupos têm valores distintos
    !.

% Solução do problema
solucao(Resposta) :-
    tabuleiro(ProblemaTabuleiro),  % Define o tabuleiro inicial
    solucionador(ProblemaTabuleiro),  % Soluciona o problema
    extrair_valores(ProblemaTabuleiro, Resposta).  % Extrai os valores do tabuleiro resolvido
```

A matriz tabuleiro contém dois números, sendo o primeiro indicando a região e o segundo o valor em si.

VANTAGENS E DESVANTAGENS DE PROGRAMAÇÃO DE RESTRIÇÕES PARA RESOLVER O PROBLEMA

Usar programação de restrições para resolver problemas como o Kojun tem várias vantagens. Ela permite expressar regras e restrições de forma direta e intuitiva, facilitando a compreensão e modificação do código. Motores de programação de restrições exploram automaticamente o espaço de soluções, utilizando técnicas eficientes de busca e poda, o que pode aumentar a confiança na correção da solução. Além disso, é possível incorporar heurísticas que aceleram a

resolução do problema. No entanto, a programação de restrições pode ser menos performática que algoritmos especializados para problemas muito grandes ou complexos. Há uma curva de aprendizado inicial para programadores não familiarizados com a técnica, e depurar programas de restrições pode ser desafiador devido às interações complexas entre as restrições. Embora flexível e poderosa, essa abordagem pode não aproveitar todas as otimizações possíveis para um problema específico e pode enfrentar limitações em termos de ferramentas e suporte.

INSTRUÇÕES DE USO

A entrada será dada através da matriz tabuleiro. O resultado será apresentado através do terminal do SWI-Prolog. Para rodar o programa, utilizamos o compilador online https://swish.swi-prolog.org/

ORGANIZAÇÃO DO GRUPO

O grupo se reuniu para aplicar os conhecimentos adquiridos na matéria de Paradigmas de Programação. Após discussões e estudos, desenvolvemos o trabalho de forma colaborativa, utilizando ideias conjuntas para encontrar soluções eficazes.

VANTAGENS E DESVANTAGENS ENTRE O PARADIGMAS FUNCIONAL E O PARADIGMA LÓGICO

Ao implementar um resolvedor de puzzles, o paradigma funcional oferece vantagens como imutabilidade, composição clara de funções e boas abstrações para recursão, mas apresenta uma curva de aprendizado íngreme e pode ter desempenho inferior devido à sobrecarga de memória. Já o paradigma lógico, como em Prolog, permite expressar regras e restrições diretamente, aproveitando a busca e poda automáticas, o que simplifica a lógica do programa. No entanto, pode ser menos eficiente para problemas grandes, difícil de depurar devido às interações complexas entre restrições, e também tem uma curva de aprendizado elevada. A escolha entre os paradigmas depende das necessidades específicas do problema e da familiaridade do desenvolvedor.

DIFICULDADES ENCONTRADAS

Resolver o problema do Kojun em Prolog apresentou diversas dificuldades. A sintaxe e a lógica declarativa de Prolog foram desafiadoras para membros não

familiarizados com programação de restrições. Implementar corretamente todas as restrições, especialmente a unicidade e a ordem dos valores nas regiões, exigiu várias iterações e refinamentos.