UNIVERSIDADE FEDERAL DE SANTA CATARINA
Disciplina: Paradigmas de Programação
Professor: Maicon Rafael Zatelli
Alunas: Gabriela Furtado da Silveira e Samantha Costa de Sousa

Relatório Trabalho III - Programação Lógica - Prolog

Análise do problema

O problema resolvido foi o jogo Makaro, que consiste em um tabuleiro nxn que contém células brancas a serem preenchidas com números (sendo que alguns números já começam preenchidos), células pretas que são apenas regiões vazias ignoradas e células que apresentam setas apontando na direção vertical ou horizontal. Há também regiões de células brancas, delimitadas pelas linhas mais grossas.

Células de uma mesma região devem ser preenchidas com números de 1 a N (tamanho da região), sem repetições. Além disso, uma célula não pode conter o mesmo número que nenhuma de suas vizinhas ortogonais (estejam dentro ou fora da região). Por fim, a célula apontada por uma seta deve conter o maior número entre os vizinhos ortogonais da seta.

Programação de restrições

A programação de restrições consiste em utilizar ferramentas do paradigma lógico, porém com a adição de restrições (regras) que permitem reduzir o conjunto de valores que as variáveis podem assumir. Assim, em vez de realizar uma busca exaustiva, são aplicadas regras que permitem diminuir o domínio válido das variáveis, sendo possível otimizar a resolução do problema.

Uma das principais vantagens na programação de restrições é poder escrever o código apenas indicando quais são as regras e restrições que precisam ser seguidas para a resolução do problema, sem a necessidade explicitar maneiras para fazê-lo. Ao contrário do que foi necessário para obter a solução dos puzzles nos trabalhos anteriores, nos quais precisamos gastar bastante tempo com as estruturas e a maneira que o programa utilizaria para chegar à resposta, com restrições isso não é necessário, e podemos focar em explicitar de forma efetiva as regras do jogo, tal que o programa chegue à resposta.

Por outro lado, uma das desvantagens de se trabalhar com restrições é a falta de conhecimento que tínhamos, bem como a necessidade de pensar de um jeito completamente diferente na solução a ser implementada. Além disso, como não sabíamos exatamente de que forma o programa estava chegando à resposta, isso tornava mais difícil depurar o código e encontrar erros na lógica.

Exemplos de uso da programação de restrições na solução

Várias restrições são aplicadas em cima das estruturas adquiridas da entrada do puzzle com o auxílio do predicado *maplist* do Prolog para a resolução do puzzle. A seguir, são ilustradas essas restrições no código.

```
% Retorna o domínio de um elemento na região baseado no tamanho da região
get_domain(RegionsSizes, Value, N) :-
    string_to_int(Value, IntValue),
    IntValue > 0,
    nth1(IntValue, RegionsSizes, Lim),
    N in 1..Lim.
get_domain(_, Value, 0) :-
    string_to_int(Value, IntValue),
    IntValue =:= 0.
```

Exemplo no código da restrição dos domínios para uma posição de uma região no código (simbolizada por um número "*Value*", lido da matriz do puzzle mapeado por regiões), baseado no tamanho da região que foi calculado previamente para *RegionsSize*.

```
% Condição para que os elementos de uma mesma região sejam distintos
apply_region_rule(DomainsList, Regions, Reg) :-
    indexes(Regions, Reg, PositionsReg),
    maplist(nth(DomainsList), ElemReg, PositionsReg),
    all_distinct(ElemReg).
```

Exemplo no código da regra de que todos os elementos de uma região devem ser diferentes entre si. Essa regra é aplicada para cada região, capturando os elementos da região e aplicando *all_distinct*.

```
% Condição que os elementos adjacentes de uma lista sejam distintos sem contar com o 0
check_adjacent_distinct([]).
check_adjacent_distinct([_]).
check_adjacent_distinct([X, Y | Rest]) :-
    (X #= 0 ; Y #= 0),
    check_adjacent_distinct([Y | Rest]).
check_adjacent_distinct([X, Y | Rest]) :-
    X #\= Y,
    check_adjacent_distinct([Y | Rest]).
```

A restrição das adjacências é aplicada para todas as linhas do tabuleiro, e então para todas as linhas do tabuleiro transposto (para aplicar nas colunas). O 0 não é contabilizado pois ele simboliza uma posição vazia no tabuleiro, assim como as posições com setas (na matriz de números da solução).

```
% Aplica regra onde a maior posição deve ser a apontada pela seta
apply_arrow_rule(N, MatResult, StrRegions, (Row, Col)) :-
    nth0(Row, StrRegions, RowList),
    nth0(Col, RowList, Arrow),
    RowAbove is Row - 1,
    RowBelow is Row + 1,
    ColLeft is Col - 1,
    ColRight is Col + 1,
    element_within_bounds(N, RowAbove, Col, MatResult, ValueAbove),
    element_within_bounds(N, RowBelow, Col, MatResult, ValueBelow),
    element_within_bounds(N, Row, ColLeft, MatResult, ValueLeft),
    element_within_bounds(N, Row, ColRight, MatResult, ValueRight),
```

```
(
    Arrow = 'D' ->
        ValueBelow #> ValueAbove, ValueBelow #> ValueLeft, ValueBelow #> ValueRight
    ;
    Arrow = 'U' ->
        ValueAbove #> ValueBelow, ValueAbove #> ValueLeft, ValueAbove #> ValueRight
    ;
    Arrow = 'L' ->
        ValueLeft #> ValueAbove, ValueLeft #> ValueBelow, ValueLeft #> ValueRight
    ;
    Arrow = 'R' ->
        ValueRight #> ValueAbove, ValueRight #> ValueBelow, ValueRight #> ValueLeft
    ;
    true
).
```

Exemplo da restrição da “regra das setas” do jogo Makaro. Para aplicar essa restrição no código, dada uma seta, é adquirido os números ao redor dela e então verificada a regra a depender de onde a seta aponta (simbolizado por ‘D’, ‘U’, ‘L, ‘R’), pois o número apontado deverá ser o maior número em relação aos outros ao redor da seta. Essa regra é aplicada para todas as setas do puzzle.

```
% Resolve o puzzle
solve_makaro(Board, RegionsMatrix, RegionsSizes, MatResult) :-
    length(Board, L),
    append(Board, FlatBoard),
    append(RegionsMatrix, FlatRegions),

    maplist(get_domain(RegionsSizes), FlatRegions, DomainsList),

    length(RegionsSizes, Max), % Max = número de regiões
    numlist(1, Max, Regs), % Lista de 1 a Max
    maplist(number_string, Regs, RegsStr),
    maplist(apply_region_rule(DomainsList, FlatRegions), RegsStr),

    DomainsList = FlatBoard,

    % Aplica regra das adjacências linha por linha da matriz, e então transpõe e aplica nas colunas
    list_to_matrix(DomainsList, L, MatResult),
    maplist(check_adjacent_distinct, MatResult),
    transpose(MatResult, TransposedBoard),
    maplist(check_adjacent_distinct, TransposedBoard),

    % Posições das setas e aplicação da regra
    maplist(atom_string, StrFlatRegions, FlatRegions), % Transforma em string pois não encontrava 'D', etc
    list_to_matrix(StrFlatRegions, L, StrRegions),
    arrow_positions(StrRegions, PositionsArrows),
    maplist(apply_arrow_rule(L, MatResult, StrRegions), PositionsArrows).

    maplist(label, MatResult).
```

Todas essas restrições são utilizadas na resolução do puzzle, como visto acima em *solve_makaro*, utilizando também predicados auxiliares para isso.

Entrada e resultado

O programa recebe como entrada um tabuleiro do jogo Makaro na forma de um arquivo de texto com extensão “.txt”, cujo nome é passado na main obedecendo o seguinte formato:

- Na primeira linha há apenas um número n que corresponde ao número de linhas do puzzle (ex: para um tabuleiro 4x4, n = 4).

- As próximas n linhas devem conter as linhas do puzzle representadas de acordo com as regiões: todas as células pertencentes a uma mesma região devem ser representadas pelo mesmo número, sendo que os números correspondentes a cada região são atribuídos em ordem crescente (1, 2,…), da esquerda para a direita e de cima para baixo conforme a primeira aparição de uma célula da região. Para células pretas (vazias) deve ser colocado um “X” na posição e para as setas, devem ser usadas letras que representem o sentido para o qual apontam: L - esquerda (left), R - direita (right), U - cima (up), D - baixo (down).
- As n linhas seguintes devem conter os números de cada célula, utilizando 0 para células vazias (brancas ou pretas) e para setas.

Exemplo de entrada para um tabuleiro 8x8:
8
1 D 2 3 3 4 4 4
5 5 2 6 D U 4 U
R 5 5 6 7 7 8 8
9 5 U 10 7 11 L 12
9 10 10 10 11 11 11 12
9 X D 13 U 11 X 12
14 14 13 13 15 15 15 D
16 16 16 X 17 17 15 15
0 0 0 2 0 0 3 4
0 5 2 0 0 0 0 0
0 0 3 0 3 0 0 0
0 0 0 1 2 0 0 3
0 3 2 4 0 5 0 0
3 0 0 0 0 0 0 1
0 2 0 0 0 4 0 0
2 3 0 0 2 0 3 0
(puzzle disponível em: https://www.janko.at/Raetsel/Makaro/001.a.htm)

O puzzle resolvido é apresentado na forma da impressão no terminal da matriz nxn contendo os números da solução.

Paradigma Funcional X Paradigma Lógico

No paradigma funcional, descrevemos exatamente como as coisas são executadas e todo o processo de resolução do puzzle incluindo o processo de backtracking. Isso torna o código bem mais extenso e complexo em relação ao paradigma lógico, mas também menos abstraído, isto é, conseguimos visualizar exatamente pelo código como a máquina executa cada função sem muitas dúvidas.

O paradigma lógico é mais próximo do que pensamos de um resolvedor de puzzle, porque nos preocupamos essencialmente com os fatos e lógica, e podemos juntar isso com a programação de restrições. O código nesse segundo paradigma se tornou bem menor e com tempo equivalente de execução se comparado aos códigos gerados do paradigma funcional, no qual para conseguirmos esse tempo, foram necessárias muito mais linhas de código. Porém, pode ser difícil entender como as coisas estão sendo feitas por “debaixo dos panos” justamente por ser um paradigma baseado em inferência lógica, que é um método muito

diferente dos paradigmas de costume, em que passamos o “passo-a-passo” da execução do programa.

Organização

Gabriela implementou a parte do código que permitia que o programa preenchesse o tabuleiro considerando os valores válidos para cada região. Samantha implementou as demais regras do jogo. Além disso, ambas trabalharam juntas na parte de leitura do arquivo e montagem das matrizes.

Dificuldades e soluções

Uma das principais dificuldades encontradas foi entender a sintaxe e o funcionamento, tanto da linguagem Prolog como da biblioteca específica para programação de restrições. Além disso, notamos que é bastante difícil encontrar erros nos códigos dessa linguagem, o que gerou muitos problemas durante o desenvolvimento da solução do puzzle, já que o programa muitas vezes não indicava erros de nenhum tipo, mas apresentava um comportamento diferente do esperado. Dessa forma, muito tempo foi gasto tentando entender a causa dos erros e onde estavam os problemas, sendo que geralmente eram equívocos simples de sintaxe ou confusão entre os subtipos do Prolog (atom, number, string).

Os erros foram superados com testes, análises de código e mudanças de tipos nas estruturas antes de aplicar as restrições (utilização de predicados como *number_string* e *atom_string*).

Todos os puzzles solucionados nas versões em paradigma funcional (de tamanho 8x8, 12x12 e 15x15) continuam sendo solucionados em tempo hábil nesta versão.