Trabalho 3 - Resolvedor em *Prolog* para o jogo Suguru

INE5416 - Paradigmas de Programação

Matheus Dhanyel C. Roque e Pedro H. Aquino Silva

Abril 2021

1 Análise do problema

O trabalho consiste em desenvolver um programa em linguagem Prolog utilizando programação de restrições que encontre uma solução para um *puzzle* do tipo Suguru. No enunciado, indicou-se que o grupo utilizasse a biblioteca CLP(FD).

Este jogo é um quebra-cabeças com números, que tem como base um tabuleiro $n \times n$, dividido em regiões — ou *tectons*. Dentro de cada *tecton*, as posições livres devem ser preenchidas com cada número entre 1 e a quantidade de posições do *tecton*, sendo que dois números iguais não podem ser vizinhos ortogonais ou diagonais. Algumas posições já são preenchidas no tabuleiro de entrada.

2 Programação de restrições

Druante a fase de pesquisa do trabalho, o grupo estudou sobre programação de restrições, de forma que a mesma pudesse ser utilizada no trabalho. Utilizamos a biblioteca CLP(FD), como sugerido no enunciado.

Programação de restrições consiste em definir condições de consistência do problema em questão, de modo que o programador não manipule explicitamente os dados do programa para encontrar a solução, mas sim defina as condições para que a solução seja válida. A partir daí, o sistema ou módulo que permite o paradigma de programção de restrições pode encontrar os valores específicos que satisfaçam as restrições do problema.

No nosso trabalho, as restrições definidas foram duas: o teste de *tectons*, e o teste das posições adjacentes. Para as regiões, precisamos garantir que todos os valores sejam diferentes, e que eles se encontram entre 1 e o tamanho da região. As restrições de cada região são definidas no predicado `tecton_ok/1`:

```
% aplica restricoes aos valores do tecton
tecton_ok(Tec) :-
    all_distinct(Tec),
    length(Tec, L), Tec ins 1..L.
```

Já o teste de posições adjacentes teve suas restrições definidas no predicado `test_adj/3`. Nele, as restrições de que os vizinhos devem ser todos diferentes são definidas utilizando o operador `#\=` da biblioteca CLP(FD).

```
test_adj(R,C,Rows) :-
    % encontra o valor de todos os vizinhos
    get_up(R,C,Rows,Up),
    get_down(R,C,Rows,Down),
    get_left(R,C,Rows,Left),
```

```
    get_right(R,C,Rows,Right),
    get_uldiag(R,C,Rows,UL),
    get_urdiag(R,C,Rows,UR),
    get_lldiag(R,C,Rows,LL),
    get_lrdiag(R,C,Rows,LR),
    elt_at(R,C,Rows,_-Ctr),
    % define restricoes pra todos os vizinhos
    Ctr #\= Up, Ctr #\= Down, Ctr #\= Left, Ctr #\= Right,
    Ctr #\= UL, Ctr #\= UR, Ctr #\= LL, Ctr #\= LR.
```

3 Descrição da solução

A solução empregou a biblioteca CLP(FD) e a modelagem apresentada nas seções seguintes.

3.1 Estruturas e entrada de dados

Primeiramente foi decidido como seria feita a modelagem do *puzzle* Suguru em nosso programa. Analizando as regras do puzzle, decidimos por representar o jogo com uma matriz em que cada elemento vazio é representado por `_` e cada posição contém dois dados, na forma `ID-N`, em que ID é o identificador de cada tecton, e N é o valor inteiro do elemento na posição. Uma instância do jogo é inserida como segue:

```
% https://www.janko.at/Raetsel/Suguru/001.a.htm
problem(1,P) :-
    P = [
        [a-4,a-_,b-_,c-_,c-_,c-_],
        [a-_,b-_,b-_,b-_,c-_,d-_],
        [a-_,e-_,b-4,g-_,c-_,d-1],
        [e-_,e-_,g-_,g-2,g-_,d-_],
        [e-5,f-_,f-_,g-3,h-5,d-_],
        [e-_,h-_,h-_,h-_,h-_,d-_]].
```

O grupo decididiu que a instância de Suguru a ser solucionada deveria ser definida no arquivo `suguru.pl` do trabalho, de modo que a entrada de dados é *hard-coded* no programa Prolog.

As estruturas de dados utilizadas são diferentes das utilizadas nos resolvedores de Makaro e Suguro desenvolvido em Haskell e Common Lisp para os Trabalho 1 e 2. Naquele momento, utilizamos um *array* 2-D para representar o tabuleiro, e listas de tuplas para representar cada *tecton*. Enquanto a abordagem atual simplifica a entrada de dados, faz-se necessário construir alguma estrutura de dados que contenha somente os valores nas posições de cada tecton, possibilitando o uso das funções da biblioteca CLP(FD).

Em comparação às soluções no paradigma funcional, a programção de restrições se mostra mais simples. Nas nossas implementações anteriores, gerávamos uma lista de valores válidos para cada posição e, então, utilizávamos um algoritmo de tentativa e erro para encontrar uma posição, isto é, encontrar valores válidos para todas as posições vazias. Aqui, não é necessário gerar valores válidos, mas sim programar as regras para que um dado valor o seja. Como o backtracking e a geração de valores válidos ficam por debaixo dos panos (a cargo da implementação do Prolog), é bastante mais simples implementar estas restrições.

Contudo, por ser bastante diferente do costume dos membros do grupo, a implementação deste resolvedor em Prolog enfrentou desafios de compreensão e raciocínio na linguagem. Isso se traduziu em maior dificuldade de implementação do trabalho em Prolog quando comparado ao paradigma funcional, principalmente pelo tempo de uso da linguagem (menor no caso do Prolog) e da novidade da programação de restrições.

3.2 Detalhes de otimização e implementação

A implementação da testagem de adjacentes é feita por posição não preenchida. Desta forma, precisamos encontrar algum jeito de buscar os valores da linha e coluna que cada posição se encontra. As posições abertas são indicadas com a variável anônima (_), e não encontramos uma maneira simples de buscá-las somente. O modo como essa busca foi efetivamente realizada só pode ser descrita como um *workaround*, ou ainda uma gambiarra:

```
1 check_adj(Rows) :-
2     % gambiarra para buscar os valores livres da matriz (tem o efeito
     colateral de pegar tambem os valores 1)
3     findall((R,C), elt_at(R,C,Rows,_-1),Vars),
4     check_adj_(Vars, Rows).
```

A solução empregada para esse problema consistiu em utilizar o predicado `findall/3`, buscando as linhas e colunas (R e C) de todos os valores `_-1`. Isso faz com que as variáveis anônimas que ainda podem ser avaliadas em 1 sejam buscadas. Portanto, precisamos executar a checagem de adjacentes antes da checagem de *tectons*; e ainda, essa abordagem também nos retorna as posições já preenchidas com 1, o que não é um problema em termos lógicos, mas é um desperdício de tempo de processamento. Por estas razões, é uma solução não ótima para a busca dos valores ainda não avaliados.

Outra otimização que poderia ter sido feita é ainda relativa a testagem de adjacentes. Poderíamos avaliar somente os valores de *tectons* diferentes, uma vez que os do mesmo *tecton* tem a garantia de serem diferentes, dada pelo predicado `check_tectons/1`.

4 Sobre a execução do trabalho

4.1 Divisão de tarefas

A carga de implementação foi dividida de forma que ambos pudessem trabalhar independentemente, um responsável pela checagem dos valores adjacentes e o outro pela checagem dos tectons e da geração e formatação da saída. Para facilitar o trabalho entre as duas partes foi criado um repositório no Github para colocar os arquivos referentes ao trabalho.

Contudo, devido às dificuldades com a linguagem e programação de restrições, assim como questões de tempo relativas ao fim do semestre, uma das partes do grupo trabalhou consideravelmente mais que a outra, inclusive reimplementando parte da solução realizada pelo outro membro. Nesse aspecto, o grupo poderia ter se organizado melhor, e inclusive conversado mais sobre as dificuldades encontradas com o paradigma de programação de restrições, uma vez que era um trabalho de pesquisa que não foi passado em sala de aula.

4.2 Dificuldades

No geral, não foram encontradas dificuldades relativas à modelagem do problema. O grupo pesquisou bastante sobre outras implementações de problemas parecidos com o Suguru, o que também ajudou bastante nas dificuldades encontradas.

A seção que mais apresentou alguma dificuldade foi a checagem de adjacentes, mas o problema também foi eventualmente resolvido.