Aluno: Gilson Trombetta Magro
Matrícula: 18202192
Disciplina: INE5416 - Paradigmas de Programação

RELATÓRIO: TRABALHO III

Novembro de 2020

1 ANÁLISE DO PROBLEMA

O problema escolhido para o desenvolvimento deste trabalho foi o Straights (ou Str8s). Este puzzle é um jogo similar ao Sudoku, e é composto de um tabuleiro quadrado (usualmente de tamanho 6 x 6 ou 9 x 9) que possui casas brancas e negras. O tabuleiro deve ser preenchido de acordo com as seguintes regras:

- Todas as casas brancas devem ser preenchidas com números de 1 a N (considerando um tabuleiro NxN), e nenhuma linha/coluna pode conter números repetidos;
- O tabuleiro pode ser separado em "regiões": uma região é composta apenas por casas brancas conectadas vertical ou horizontalmente e é delimitada por casas negras e/ou pelas bordas do tabuleiro;
- Toda região (vertical ou horizontal) deve conter um conjunto de números consecutivos, porém não necessariamente em ordem;
- Um número em uma casa negra serve para remover a possibilidade de uso daquele número em sua respectiva linha/coluna;
- Casas negras não fazem parte de nenhuma região.

2 SOLUÇÃO PROPOSTA

2.1 ORGANIZAÇÃO

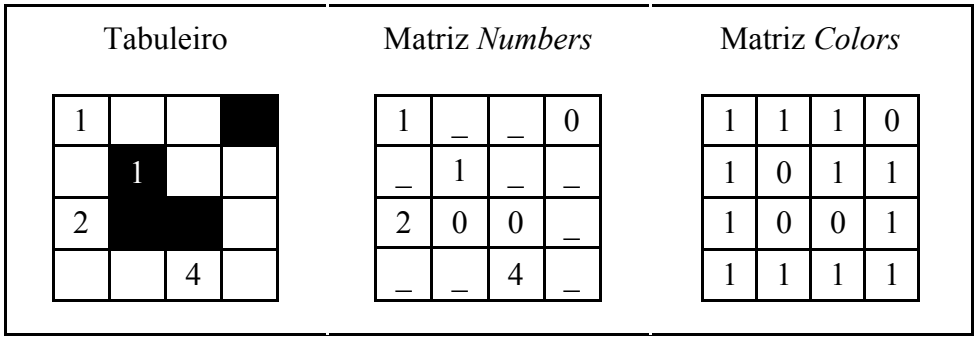
Neste trabalho, a linguagem de programação utilizada foi o Prolog. Devido à natureza descritiva do paradigma de programação lógico, o número de linhas de código necessárias para realizar o trabalho foi relativamente pequeno. Nesse sentido, o código foi disposto em apenas dois arquivos: *main.pl* e *examples.pl*. O primeiro contém a implementação do

resolvedor propriamente dita, e no segundo tem-se apenas alguns exemplos de entradas para o programa.

2.2 REPRESENTAÇÃO

O tabuleiro é representado por duas matrizes de tamanho NxN: *Numbers* e *Colors*. A primeira é uma matriz de inteiros ou variáveis de domínio, e representa os valores de cada casa do tabuleiro; casas negras vazias são preenchidas com um zero, enquanto casas vazias brancas são preenchidas com um *underscore*, para simbolizar que queremos que o programa atribua uma valor àquela casa. A segunda é uma matriz de zeros e uns, e representa a cor das casas do tabuleiro, onde 1 representa uma casa branca, e 0 representa uma casa negra.

Figura 1 - Representação do tabuleiro



2.3 SOLUÇÃO

Para construir as regras que solucionam o *puzzle*, utilizou-se a biblioteca CLPFD (do inglês *Constraint Logic Programming over Finite Domains*) do Prolog. Como o nome já indica, essa biblioteca permite definir restrições lógicas e aritméticas sobre variáveis de domínios finitos. Isto é útil no caso de um jogo como o *Straights*, onde cada casa precisa obedecer certas restrições, de acordo com sua posição no tabuleiro.

A parte mais crucial do programa é implementada na regra *solve*, pois é ela que define quando um tabuleiro é uma solução válida ou não. Essa regra pode ser utilizada para gerar todas as configurações válidas de tabuleiros de tamanho NxN, para todo N natural maior ou igual a zero. Contudo, no caso específico deste trabalho, ela é utilizada como um resolvedor que recebe um tabuleiro incompleto e imprime todas as soluções válidas para aquele tabuleiro. A definição da regra *solve* é apresentada abaixo:

Regra *solve*

```
1  solve(NumberRows, ColourRows) :-
2      length(NumberRows, Length),
3      length(ColourRows, Length),
4      maplist(same_length(NumberRows), NumberRows),
5      maplist(same_length(ColourRows), ColourRows),
6      append(NumberRows, FlatNumbers),
7      append(ColourRows, FlatColours),
8      FlatColours ins 0..1,
9      colours(FlatNumbers, FlatColours, Blacks, Whites),
10     Blacks ins 0..Length,
11     Whites ins 1..Length,
12     transpose(NumberRows, NumberCols),
13     transpose(ColourRows, ColourCols),
14     maplist(no_repeats, NumberRows),
15     maplist(no_repeats, NumberCols),
16     maplist(regions(Length), NumberRows, ColourRows),
17     maplist(regions(Length), NumberCols, ColourCols).
18
19
20
```

A ideia por trás da regra *solve* pode ser resumida ao seguinte conjunto de restrições, que garantem que um tabuleiro seja uma configuração válida para o problema:

I. As matrizes *NumberRows* e *ColourRows* devem ser quadradas e de mesmo tamanho;

II. Todas os elementos na matriz *ColourRows* devem pertencer ao intervalo discreto [0,1].

III. Seja *Length* o tamanho do lado do tabuleiro, os valores de todas as casas brancas do tabuleiro devem pertencer ao intervalo discreto [1, *Length*]. Similarmente, os valores de todas as casas negras do tabuleiro devem pertencer ao intervalo discreto [0, *Length*], pois o valor zero é usado para representar casas negras vazias que nunca serão preenchidas.

IV. Não deve haver repetições de números nas linhas e colunas do tabuleiro, exceto pela eventual repetição de zeros em casas negras.

V. Ademais, a consecutividade das regiões horizontais e verticais do tabuleiro deve ser respeitada.

Note que as restrições I e II são triviais, e a restrição III torna-se trivial se notarmos que, para dividir as casas em brancas e negras, pode-se utilizar a própria matriz de cores, que descreve a cor de cada casa. Portanto, resta resolver as restrições IV e V.

Ora, apesar da peculiaridade de que zeros podem ser repetidos nas linhas e colunas, a restrição IV também é bastante simples: basta dividir as casas da matriz de números em dois grupos (nulos e não-nulos), e então aplicar a restrição somente sobre as casas que não são nulas.

Desse modo, resta resolver a restrição V. Esta última é certamente a restrição mais complexa, porque devemos garantir que dentro de uma linha ou coluna, todas as regiões de casas brancas conectadas formem conjuntos de números consecutivos. Nesse sentido, devemos dividir esta restrição em duas partes: a primeira consiste em encontrar cada região, e a segunda, em garantir que cada região encontrada forma, de fato, um conjunto de valores consecutivos. A primeira é implementada pela regra *regions*, e a segunda, pela regra *consecutive*. Ambas são apresentadas abaixo:

Regra *regions*

```
1  regions(L, Ns, Cs)                 :- regions(L, Ns, Cs, []).
2  regions(L, [], [], Rs)             :- consecutive(Rs, L).
3  regions(L, [N|Ns], [1|Cs], Rs) :- regions(L, Ns, Cs, [N|Rs]).
4  regions(L, [_|Ns],  [0|Cs], Rs) :- consecutive(Rs, L), regions(L, Ns, Cs, []).
```

Regra *consecutive*

```
1  consecutive([], _) :- !.
2  consecutive([_], _) :- !.
3  consecutive(Set, Limit) :-
4        length(Set, Length),
5        Range #= Limit - Length + 1,
6        LowerBound in 1..Range,
7        indomain(LowerBound),
8        UpperBound #= LowerBound + Length - 1,
9        Set ins LowerBound..UpperBound.
```

Note que a regra *regions* percorre recursivamente sobre a linha (ou coluna) do tabuleiro, e carrega consigo uma lista da região que está sendo percorrida naquele momento. Desse modo, sempre que uma casa branca é encontrada, ela é adicionada à lista; e sempre que

uma casa negra é encontrada, utiliza-se a regra *consecutive* para adequar a região atual à regra de consecutividade. Então, a lista é zerada e parte-se para a próxima região.

Já a regra *consecutive* recebe uma lista que representa uma região, e infere se esta região forma um conjunto consecutivo, ou não. Neste caso, dado o intervalo discreto [1, *Limit*], onde *Limit* é o tamanho do tabuleiro, definir "janelas" de tamanho *Length* (que corresponde ao tamanho da região, ou *Set*), e verificar se a região se enquadra em alguma destas janelas. Para fins deste trabalho, podemos assumir que os valores do conjunto recebido pela regra *consecutive* são necessariamente distintos. Deste modo, se a região enquadrar-se em alguma dos sub-intervalos de [1, *Limit*], então os valores da região certamente serão consecutivos.

3 ENTRADA E SAÍDA

3.1 ENTRADA

Visto que o programa pode ser carregado e testado via o próprio interpretador do Prolog, e que ao realizar as consultas, certas posições do tabuleiro devem ser deixadas indefinidas para que o próprio resolvedor preencha-as, optei por não implementar funções que leiam o tabuleiro do teclado. Invés disso, a entrada para o problema deverá ser feita editando o próprio código, ou definindo tabuleiros na linha de comando. Exemplos de entradas são fornecidas no arquivo *examples.pl*. É importante lembrar que um tabuleiro sempre é definido por duas matrizes: uma de números, e outra de cores (veja a seção 2.2 deste documento).

Exemplo de entrada

```
1  board(2,
2        [[1,_,_,0],   % numbers
3         [_,1,_,_],
4         [2,0,0,_],
5         [_,_,4,_]],
6        [[1,1,1,0],   % colors
7         [1,0,1,1],
8         [1,0,0,1],
9         [1,1,1,1]]).
```

3.2 SAÍDA

A função *solve_all* serve de interface com o usuário. Ao receber um tabuleiro (isto é, as duas matrizes que juntas representam um tabuleiro), a função *solve_all* imprimirá na tela as soluções válidas para aquela configuração inicial, uma após a outra. Note que este utilitário só funcionará quando ambas as matrizes de números e cores forem bem definidas e já tiverem sido inicializadas. Se o objetivo for enumerar todas as soluções possíveis para todos os tamanhos de tabuleiros, então deve-se utilizar a função *solve* diretamente. Abaixo é apresentado um exemplo de saída do programa.

Figura 3 - Exemplo de saída do programa

```
?- ['main.pl'], ['examples.pl'].
true.

?- board(2, Numbers, Colours), solve_all(Numbers, Colours).

 1  3  2 [0]
 4 [1] 3  2
 2 [0][0] 3
 3  2  4  1

Numbers = [[1, 3, 2, 0], [4, 1, 3, 2], [2, 0, 0, 3], [3, 2, 4, 1]],
Colours = [[1, 1, 1, 0], [1, 0, 1, 1], [1, 0, 0, 1], [1, 1, 1, 1]] ;
false.
```

4 CONSIDERAÇÕES

4.1 PARADIGMAS FUNCIONAL E LÓGICO

Considerando o primeiro trabalho, implementado em Haskell, o segundo trabalho, implementado em LISP, e agora este, é possível destacar algumas das principais diferenças entre os paradigmas funcional e lógico.

É evidente que os paradigmas são bastante diferentes. No paradigma funcional, cabe ao programador implementar funções que resolvam determinado problema. Já no paradigma lógico, o programador deve apenas descrever o problema utilizando as ferramentas fornecidas pela linguagem, e a própria linguagem encarrega-se de encontrar as soluções que satisfaçam as restrições impostas pelo programador.

Nesse sentido, o paradigma lógico parece mais adequado para certos tipos de problemas, especialmente os que envolvem equações aritméticas e sentenças lógicas. Por outro lado, o paradigma funcional assemelha-se mais à programação usual (com o paradigma procedural ou orientado a objetos).

4.2 DIFICULDADES ENCONTRADAS

Acredito que, durante a implementação do trabalho, o aspecto mais desafiador foi justamente encontrar uma maneira de aliar as restrições aritméticas da biblioteca CLPFD com a estrutura do *puzzle*. Em particular, eu demorei para entender como eu poderia montar as regiões do tabuleiro sem que a cor de cada casa estivesse definida (no caso em que a matriz de cores não está completamente inicializada).

Por fim, entretanto, compreendi que muito do que se faz sob o paradigma lógico tem a ver com "restringir", invés de "verificar". Nesse sentido, ao contrário de checar se uma casa é branca ou negra, o algoritmo na verdade restringe a cor da casa para uma das duas cores e procede a partir desta afirmação.

5 REFERÊNCIAS

COMPUTERPHILE. **Python Sudoku Solver - Computerphile**. 2020. (10m52s). Disponível em: <https://youtu.be/G_UYXzGuqvM>. Acesso em: 07 out. 2020.

SWI PROLOG. **Example: Sudoku.** Disponível em: <https://www.swi-prolog.org/pldoc/man?section=clpfd-sudoku>. Acesso em: 21 nov. 2020.