UNIVERSIDADE DE ÉVORA

Trabalho prático de Programação Declarativa

Nonogramas

Discente

Yaroslav Kolodiy nº 39859

Docentes

Salvador Abreu

Ano letivo 2019/2020

Descrição do problema

Nonograma é um puzzle japonês. Ao contrário do Sudoku, este é conhecido por vários nomes, incluindo Griddlers ou Paint by Numbers. Foi inventado em 1987 por Tetsuya Nishio. O objectivo do puzzle é descobrir, numa grelha, que células serão pintadas e quais ficarão vazias.

O que dita que células são pintadas são os números escritos ao lado da grelha. Neste puzzle, os números ditam quantos quadrados consecutivos são preenchidos numa dada linha ou coluna. Por exemplo, uma dica de "4 8 3" significa que existe um conjunto de 4, de 8 e de 3 quadrados pintados, por essa ordem, com pelo menos um espaço em branco a separar os blocos. É tão importante determinar que células são pintadas como que células ficam vazias.

Descrição do trabalho

O trabalho consistia na criação de um programa que resolvesse nonogramas, utilizando uma das duas linguagens aprendidas durante o semestre. Para tal este trabalho foi realizado na linguagem de programação prolog, neste caso gprolog.

Para que este resolva o nonograma irá receber como input uma lista com as restrições das linhas e das colunas que por sua vez também são listas, as quais contem sublistas as quais cada lista é a restrição de uma determinada linha/coluna.

Resolução do problema

Segundo algumas pesquisas realizadas, este problema é de uma complexidade elevada, e as principais maneiras de o resolver consistem em backtraking ou constraint programing. Deste modo, foi implementado um algoritmo utilizando constraint programing (constraint.pl), outro de backtraking (dfs.pl) e um terceiro no qual optei por juntar ambos os algoritmos (const_dfs.pl).

Em cada um destes três algoritmos a parte inicial, construção de uma matriz, e a parte final, print do resultado são bastante semelhantes. Na construção da matriz consiste em calcular-

se o tamanho da matriz, i.e., calcular o número de sublistas das restrições das linhas e das colunas. De seguida, cria-se uma matriz consoante os tamanhos pretendidos e obter-se uma lista de linhas e outra de colunas, deste modo pode-se aplicar as restrições de cada linha/coluna individualmente. O print consiste em percorrer os elementos da lista de linhas e dar print do elemento correto.

No primeiro algoritmo, constraint programing, durante a criação da matriz os elementos são restringidos a 0 ou 1. De seguida, é aplicada a constraint de que a soma da linha tem de ser igual a soma da restrição para essa linha/coluna. Após esta ser aplicada a todas as linhas e colunas haverá valores já conhecidos. De seguida, vai-se verificar cada linha para averiguar se esta respeita as restrições pretendidas. Após esta verificação o nonograma já esta resolvido com os valores corretos.

O segundo, backtraking, ou uma versão simplificada de um algoritmo de pesquisa 'Depth-first' no qual vamos tratar cada linha/coluna como uma no de um grafo, i.e., durante o algoritmo vamos construir uma lista com estruturas line (***Count***, ***Line***, ***Constr***), onde Count é o número de possibilidades de construir uma Line (linha/coluna) segundo uma Constr (restrição da linha). Ou seja, cada nó pode se ligar a outros. De momento a lista que temos não nos traz valor nenhum, para tal é aplicado um sort, o qual vai ser realizado sobre a Count, assim as linhas que tem menores números de possibilidades estarão mais próximas do início da lista, neste momento temos uma árvore para aplicar o algoritmo de pesquisa, o qual vai fixar uma lista e testar a sua conjugação com os próximos elementos da lista.

Alguns testes:

1) puzzle ([[[4],[1],[3,1],[2],[2]], [[1,1],[1,1],[3],[1,2],[3]]]).

```
X X X X .
. . X . .
X X X . X
. . . X X
. . . X X

true ?

(16 ms) yes
```

constraint.pl

```
X X X X .
. . X . .
X X X . X
. . . X X
. . . X X

true ?

(16 ms) yes
```

dfs.pl

2) puzzle([[[3],[2,1],[3,2],[2,2],[6],[1,5],[6],[1],[2]], [[1,2],[3,1],[1,5],[7,1],[5],[3],[4],[3]]]).

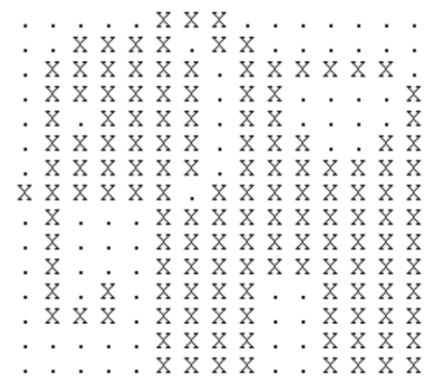
```
. X X X . . . .
X X . X . . . .
. X X X . . X X
. . X X . . X X
. . X X X X X X
X . X X X X X .
X X X X X X . .
. . . . X . . .
. . . X X . . .

true ?

(94 ms) yes
```

constraint.pl

```
. X X X . . . .
X X . X . . . .
. X X X . . X X
. . X X . . X X
. . X X X X X X
X . X X X X X .
X X X X X X . .
. . . . X . . .
. . . X X . . .

true ?

(16 ms) yes
```

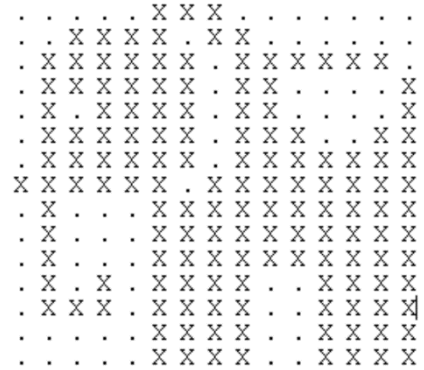
dfs.pl

3)puzzle([[[3],[4,2],[6,6],[6,2,1],[1,4,2,1],[6,3,2],[6,7],[6,8],[1,10],[1,10],[1,10],[1,1,4,4],[3,4,4],[4,4],[4,4]],[[1],[11],[3,3,1],[7,2],[7],[15],[1,5,7],[2,8],[14],[9],[1,6], [1,9],[1,9],[1,10],[12]]]).



```
true ?

(2094 ms) yes
```

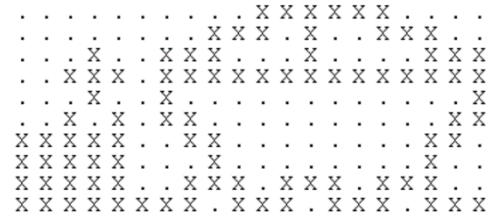
constraint.pl



```
true ?

(32 ms) yes
```

dfs.pl

4) puzzle([[[6], [3, 1, 3], [1, 3, 1, 3], [3, 14], [1, 1, 1], [1, 1, 2, 2],
[5, 2, 2], [5, 1, 1], [5, 3, 3, 3], [8, 3, 3, 3]], [[4], [4],
[1, 5], [3, 4], [1, 5], [1], [4, 1], [2, 2, 2], [3, 3], [1, 1, 2],
[2, 1, 1], [1, 1, 2], [4, 1], [1, 1, 2], [1, 1, 1], [2, 1, 2],
[1, 1, 1], [3, 4], [2, 2, 1], [4, 1]]]).



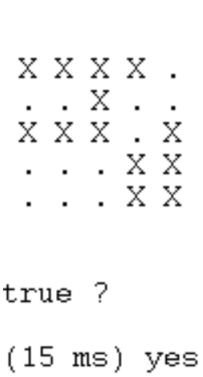
```
true ?

(344 ms) yes
```

dfs.pl

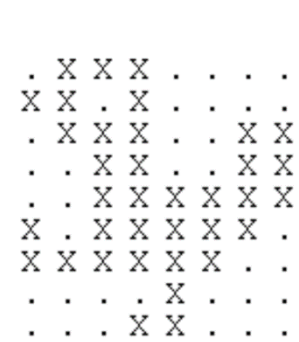
Neste exemplo o algoritmo de constraint, o tempo de execução torna-se bastante superior e sem resultado.

Para tal foi criado o terceiro algoritmo (const_dfs.pl), este consiste em fazer a primeira parte das restrições com constraint, ou seja, restringir as somas das linhas com que seja igual a soma das restrições e de seguida realizar a pesquisa igual ao algoritmo 2. Esta ordem foi a escolhida, porque ao realizarmos as constraints este fica com valores já definidos para as linhas mais preenchidas e de seguida a pesquisa ira ser facilitada porque o número de possibilidades já é menor.

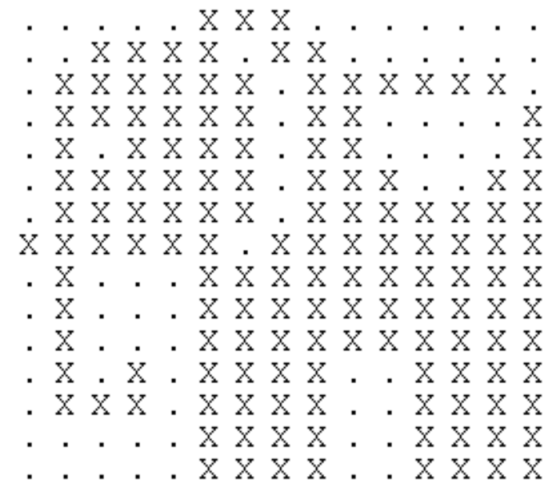
Resultados para os mesmos testes:



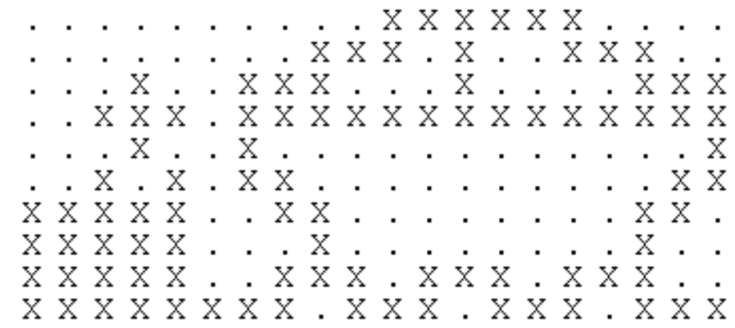
```
true ?

(15 ms) yes
```



```
true ?

(16 ms) yes
```



```
true ?

(31 ms) yes
```



```
true ?

(31 ms) yes
```

Como se pode concluir a eficiência deste algoritmo é bastante superior. Porém, nenhum destes algoritmos conseguiu resolver o segundo exemplo fornecido pelo professor.