Universidade De Évora

Departamento de Informática

Programação Declarativa

Ano letivo 2019 - 2020

Nonogram Solver

Alunos:

Luís Ressonha - 35003

Rúben Teimas - 39868

Docentes:

Salvador Abreu

25 de Janeiro de 2020

1 Introdução

Este trabalho consiste no desenvolvimento de um programa que resolve Nonogramas usando uma linguagem de Programação em Lógica (*Prolog*) ou linguagem de Programação Funcional (*Ocaml*).

Para o desenvolvimento do trabalho pensámos usar *GNU Prolog*.

2 Desenvolvimento

2.1 Ideia Inicial

Quando começámos a estudar o problema, rapidamente nos apercebemos da complexidade temporal exponencial do mesmo, devido ao número de possibilidades que poderiam ser geradas a partir das restrições de cada linha e coluna.

Como tal, o nosso primeiro pensamento foi fazer uma abordagem matemática, i.e:

2.1.1 Sobreposições

Verificar se existem sobreposições nas possibilidades e pintar os locais de sobreposições.

```
        1 1 5 1 1
1       . . . . .
3       . . X . .
1 1 1   . . . . .
1       . . . . .
1       . . . . .
```

Após esta primeira verificação, podemos constatar que na segunda linha é possivel marcar a 3ª posição como "pintada", isto porque independentemente do ponto de início, a 3ª posição será sempre "pintada". Tal pode ser comfirmado pelas expressões seguintes:

(size/2) - pista < 0

size - pista = nº casas "não pintadas" ou "desconhecidas"

No exemplo acima verifica-se esta situação porque [2.5 - 3 < 0] o que significa que iremos ter pelo menos uma casa que poderá ser considerada como "pintada". Fazendo a segunda conta, verificamos o nº de casas ainda "desconhecidas"a contar pela primeira posição válida à direita e pela primeira posição válida à esquerda na matriz: [5 - 3 = 2].
Temos então a seguinte certeza:

[3] - - X - -

nota:
-> [n] = pista da linha
-> - = casas deconhecidas
-> X = casas pintadas

2.1.2 Linha / Coluna Completa

Verificar se é possivel preencher completamente alguma(s) linha(s) ou coluna(s) apenas com a restrição dessa linha ou coluna, repetivamente.
(utilizando o mesmo puzzle)

```
        1 1 5 1 1
1         . . X . .
3         . . X . .
1 1 1   X . X . X
1         . . X . .
1         . . X . .
```

Nesta verificação completamos as linhas / colunas que ficam preenchidas apenas com as intruções dadas para linha / coluna (repetivamente). Esta verificação pode ser comprovada pela expressão seguinte:

size - (pista + espaços) = 0

Quando o resultado é igual a 0 (zero) temos uma única solução para aquela linha / coluna.

Olhando então para dois exemplos práticos:

Na 3ª linha temos uma sequência de 3 casa "pintadas"divididas com, pelo menos, uma casa vazia [1 1 1]
Fazendo a conta acima apresentada temos:

1 + 1 + 1 + 2 = 5 -> soma das restrições (1, 1, 1) com o nº mínimo de casas vazias,

temos então:
[5 - 5 = 0] o que significa que existe apenas uma combinação possível nesta linha.

[1 1 1] X . X . X
(linha)

Na 3ª coluna, fazendo a mesma verificação, obtemos:

[5 - 5 = 0] -> soma das restrições, neste caso apenas uma (5), com o nº mínimo de casas vazias, neste caso 0 (zero)

```
[ 5 ]
  X
  X
  X
  X
  X
```
(coluna)

nota:
-> [n] = pista da linha em estudo
-> - = casas deconhecidas
-> X = casas pintadas
-> . = espaços não pintados

Este tipo de abordagem matemática iria reduzir bastante as combinações existentes e, consquentemente, diminuir a complexidade do problema. Infelizente, não conseguimos implementar esta solução.

2.2 Trabalho Desenvolvido

Dado não conseguirmos implementar a nossa ideia inicial, acabámos por seguir uma ideia que não nos agradava tanto: *Brute Force.*

Recebemos, como pedido, uma única lista que contem uma lista com as restrições das linhas e outra com as restrições das colunas. Usando o predicado predifinido *length/2* obtem-se o número de colunas e linhas aplicando-o a cada uma das listas. O número de colunas será equivalente ao tamanho de uma linha e o número de linhas ao tamanho de uma coluna, e é este o raciocinio que seguiremos.
Multiplicando o tamanho de uma coluna pelo número de colunas, equivalente ao tamanho da coluna * tamanho da linha, obtem-se o tamanho total do tabuleiro, i.e, quantas posições existem. Tendo o número de posições utiliza-se novamente o predicado *length/2* para criar uma lista *Board* com as posições do tabuleiro.

Após ter o "tabuleiro"criado, serão obtidas as posições mapeadas em linhas e colunas. Para obter as colunas foi criado um predicado *rows/3* que recebe o tamanho de uma linha, o tabuleiro e a lista onde as linhas serão guardadas. O predicado irá percorrer o tabuleiro, adicionando à cabeça da lista de linhas cada posição, até que a cabeça tenha o mesmo tamanho que uma linha, i.e, a linha esteja completa. Isto será feito para o resto do tabuleiro até que se chegue ao fim.
Para obter as colunas foram criados 3 predicados, sendo 2 deles auxiliares. O predicado *oneCol/3* serve para obter uma única coluna, recebendo o índice da coluna que vai ser construída, i.e, posição na linha, a lista das linhas e a lista das colunas. Este predicado serve-se de um predicado predifinido *nth1/3*, que é verdade se o elemento for a posição I da lista das linhas. Para obter todas as colunas é usado o predicado *allCols/4* que faz uso do predicado anterior para obter todas as colunas. O predicado *columns/3* simplesmente dá o início ao index e chama os anteriores.

Após ter todo o "jogo"construído, ir-se-á finalmente começar a sua resolução. Tal como foi dito anteriormente, o algortimo de resolução deste problema é um algortimo *brute force*, em que poderá ser necessário calcular todas as possibilidades e, como tal, o seu desempenho será reduzido quando comparado com a nossa ideia inicial.

Para a resolução do mesmo foram definidos mais 3 predicados: *valid/2*, *followRules/2* e *sqtFill/3.* O predicado *valid/2* irá receber a lista das restrições e uma lista onde serão aplicadas/verificadas essas mesmas restrições. Esse predicado irá chamar o predicado *followRules/2* que receberá uma restrição e uma linha ou coluna na qual aplicar/verificar essa restrição.
Para "pintar"as posições irá ser chamado o predicado *sqtFill/3* que recebe a cabeça da lista de restrições dessa posição, a lista de posições a pintar e a lista de posições após serem pintadas. Este predicado irá "pintar"posições até que o valor da restrição seja 0, garantindo assim que a restrição, isolada de todas as restantes, esteja correta.

Após estarem "pintadas"as linhas, irá ser feita uma verificação das colunas usando de novo o predicado *valid/3.* Caso as restrições das colunas não estejam cumpridas, as posições serão novamente "pintadas"até que as restrições estejam cumpridas.

3 Conclusão

Na tentativa de resolver o problema proposto pelo docente da UC deparámo-nos com várias adversidades.

Em primeiro lugar, tivemos alguma incerteza sobre qual das liguagem sugeridas deveriamos usar (Ocaml ou Prolog).

Começámos por ponderar a utilização de Ocaml por ter uma sintaxe identica às liguagens que estamos mais habituados a trabalhar, mas em contrapartida, sabiamos que uma solução em Prolog seria mais simples (em realação ao código desenvolvido) e mais apropriada ao problema.

Esta incerteza levou-nos a outra adversidade que foi a má gestão do nosso tempo para a realização deste trabalho.

Com o prazo final a aproximar-se e a linguagem a utilizar estar ainda em dúvida, e por não sentirmos que tinhamos o conhecimento suficiente para realizar o trabalho no tempo restante, resolvemos então procurar uma solução já criada por outra pessoa que também nós conseguissemos perceber, fazendo assim as alterações necessárias para posteriormente apresentarmos.

O trabalho encontrado que decidimos apresentar, segundo o que constatámos, resolve apenas o 1^{o} exemplo (seta) em tempo aceitável. Isto pode ser porque esta solução baseia-se em "Brute Force" que testa todas as combinações possiveis até encontrar uma válida (dadas as restições).

Caso este trabalho não possa ser considerado, e tendo em conta que somos os dois alunos finalistas, gostariamos de propor uma apresentação presencial deste trabalho ou alargar o prazo de entrega para a época especial.

4 Referências

https://github.com/brstf/7languages7weeks/blob/master/3prolog/picross.pl