**Tic Tac Logic**

**Resolução de Problemas de Decisão**

**Usando Programação em Lógica com**

**Restrições**

**Autores**: André Barros e Edgar Ramos

**Resumo:** Este artigo complementa o segundo projecto da unidade curricular de PLOG, do MIEIC na FEUP. O projecto, escrito em Prolog, visa resolver qualquer tabuleiro de Tic Tac Logic.

**1 Introdução**

O objectivo deste segundo projeto era implementar a resolução de um problema de optimização ou decisão, na linguagem Prolog usando restrições. Foi dada a escolha de um dos dois tipos de problemas ao grupo, tal como o puzzle/problema a resolver.

A escolha acabou por ser o puzzle Tic Tac Logic, um problema de decisão. Este puzzle consiste num tabuleiro quadrado com lado de tamanho par, onde o jogador tem de colocar quer X’s quer O’s.

Este artigo descreverá detalhadamente uma resolução capaz de resolver qualquer tabuleiro apresentado.

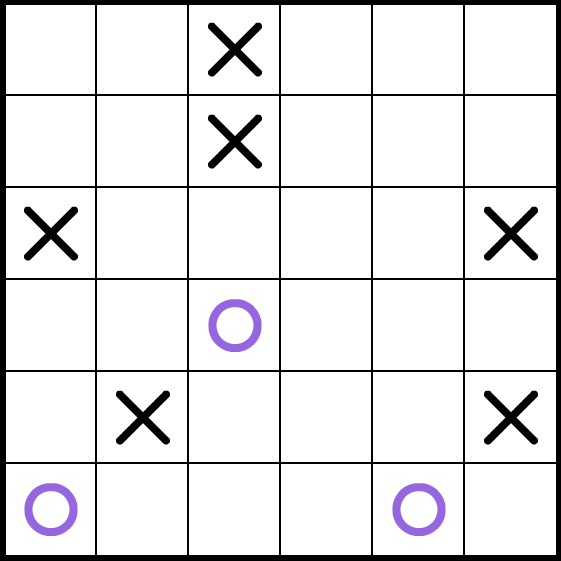
**2 Descrição do Problema**

O puzzle Tic Tac Logic consiste num tabuleiro NxN em que N tem de ser um número par, sendo este preenchido por X’s e O’s de maneira a que:

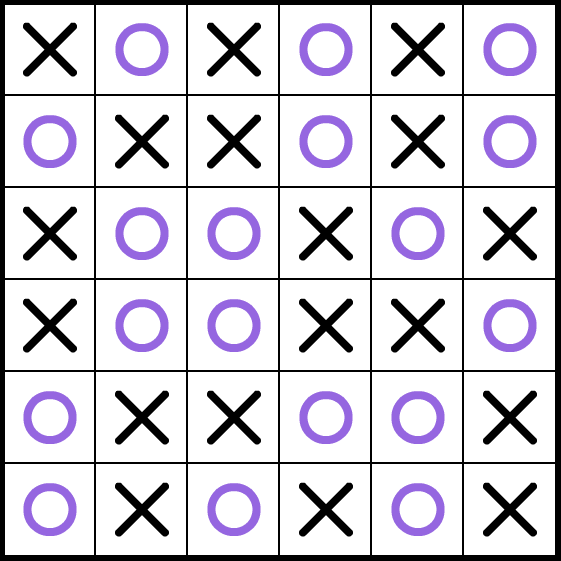
- Não pode haver mais do que dois X’s ou O’s consecutivos numa linha ou coluna.

*- O número de X’s e O’s de cada coluna e linha tem de ser identico.*

-Todas as linhas e colunas são unicas.



**Figura 1.** Exemplo de um tabuleiro inicial de Tic Tac Logic 6x6



**Figura 2.** Exemplo de um tabuleiro final de Tic Tac Logic 6x6

**3 Abordagem**

Na implementação do puzzle e Prolog, foi decidido representar o tabuleiro como uma lista de listas, onde cada elemento pode ser um 1 (X) ou 0 (O).

starting\_state6x6([[ C1, C2, 1,C4, C5, C6],

[ C7, C8, 1,C10,C11,C12],

[ 1,C14,C15,C16,C17, 1],

[C19,C20, 0,C22,C23,C24],

[C25, 1,C27,C28,C29, 1],

[ 0,C32,C33,C34, 0,C36] ]).

Este é arepresentação em código referente à Figura 1.

**3.1 Variavéis de Decisão**

Tendo em conta a representação anterior, a solução pretendida toma a forma de uma lista de tamanho identico ao numero de casas no tabuleiro, obtida ao executar um *flatten* do tabuleiro.

O domínio das variavéis de solução é [0,1].

De seguida é apresentado um excerto do predicado de resoluçõ de u tabuleiro, correspondendo à parte de *flatten* e definição do domínio.

principal(Board, N):-

flatten(Board, Vars),

domain(Vars, 0,1),

(...)

**3.2 Restrições**

Relembrando as regras do jogo, estas são:

- Não pode haver mais do que dois X’s ou O’s consecutivos numa linha ou coluna.

*- O número de X’s e O’s de cada coluna e linha tem de ser identico.*

-Todas as linhas e colunas são unicas.

Ora, como é necessário operar sobre quer linhas quer colunas, a abordagem do grupo foi em obter a matrix transposta do tabuleiro inicial, colocando, assim, as colunas como linhas. Como tal, apartir de agora e até ao fim cada vez que este relatório se referir a linhas, referir-se-á a colunas também.

Assim, a resolução deste problema pode ser atingida com três restrições:

- Não podem existir 3 números iguais consecutivos

- A soma do elementos de uma linha é igual a metade do tamanho da linha

- Todas as linhas têm de ter pelo menos um elemento diferente.

**Não podem existir 3 números iguais consecutivos:**

Para cada linha seram analisados os seus elementos de maneira a que cada um é verificado com os dois que seguem, por exemplo, na lista [H1,H2,H3,H4,H5,H6], a primeira iteração será:

H1 != H2 ou H1 != H3

Assim, será garantido que nenhuma linha tem mais que dois números iguais seguidos.

De seguida é apresentado a parte principal do predicado encarregado de tratar dessa restrição:

check\_consecutive([H1, H2, H3 |T]):-

(H1 #\= H2) #\/ (H1 #\= H3),

check\_consecutive([H2, H3 | T]).

**A soma do elementos de uma linha é igual a metade do tamanho da linha:**

Esta é a restrição mais simples, tendo de se ter apenas em conta que para cada linha o número de X’s e 0’s têm de ser iguais, e que um X é representado por 1 e O por 0 a soma será sempre igual a metade do tamanho da linha.

De seguida é apresentado a parte principal do predicado encarregado de tratar dessa restrição, sendo N o tamanho da linha:

check\_sum(Line, N):-

D is N div 2,

sum(Line, #=, D).

**Todas as linhas têm de ter pelo menos um elemento diferente:**

O grupo decidiu abordar esta restrição criando para cada duas linhas, uma lista do mesmo tamanho que contém o resultado booleano da diferença de cada dos elementos das duas linhas, exemplificando:

[1 ,1 ,0 ,0 ,1 ,0]

!=

[0 ,0 ,1 ,0 ,1 ,1]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[1, 1, 1, 0, 0, 1]

Obtendo a lista com os resultados booleanos verifica-se de seguida se o resultado da soma da lista é maior do que 0, para garantir que tem pelo menos um elemento diferente.

De seguida é apresentado a parte principal dos predicados encarregados de tratar dessa restrição:

check\_consecutive([H1, H2, H3 |T]):-

(H1 #\= H2) #\/ (H1 #\= H3),

check\_consecutive([H2, H3 | T]).

check\_unique(Results):-

sum(Results, #> , 0).

**3.4 Estratégia de Pesquisa**

De modo a evitar que para o mesmo tabuleiro a resolução fosse sempre identica, o grupo decidiu tornar a estratégia de etiquetagem (labeling) aleatória de modo a gerar soluções diferentes para o mesmo problema.

O predicado encarregue de solucionar isso é:

choose\_var(Vars, Selected, Rest):-

length(Vars, Num),

random(0, Num, Ran),

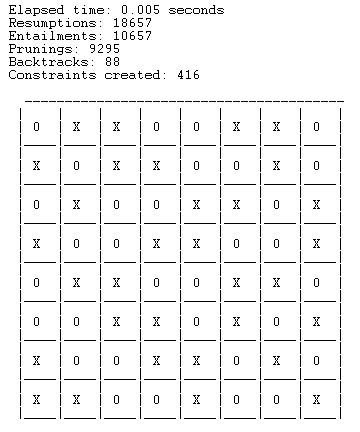
nth0(Ran, Vars, Selected, Rest),

var(Selected).

Este predicado irá gerar um número aleatorio de 0 a N-1, sendo N o tamanho da lista de variáveis Vars, de seguida, vai buscar o elemento no indice com esse número e Selected toma esse valor e Rest é uma lista identica a Vars com o elemento Selected retirado.

**4 Visualização da Solução**

A parte de visualização é praticamente igual à visualização do primeiro projeto, recebe uma matrix de qualquer tamanho e representa-o em modo texto.



**5 Resultados**

Os resultados obtidos, embora não sejam extensos, permitem perceber que a geração de um jogo é feita quase instantaneamente, sendo que raramente é ultrapassado o tempo de 1s para a resolução do jogo.

Assim sendo, passamos então a apresentar uma tabela em que foi tido em conta o tempo decorrido desde que se corre o jogo (apresentado no primeiro parâmetro na apresentação do tabuleiro, como visto na imagem anterior).

|  |  |
| --- | --- |
| Modo de jogo | Tempo |
| 6x6 | 0s |
| 8x8 | 0.004s |
| 10x10 | 0.071s |

É também de fácil perceção que quanto maior o tamanho do tabuleiro maior o tempo decorrido para que seja encontrada uma solução válida para o jogo.

O gráfico a seguir demonstra a informação também contida na tabela anterior.

**6 Conclusões**

Finalmente, o grupo pode afirmar que o uso de Prolog com restrições é útil e eficiente para determinadas situações. Esta linguagem dá uma autonomia ao programador que se baseia no facto de lhe dar a liberdade de escolher o quão complexos são os predicados criados pelo mesmo. Também é possível verificar que esta é uma linguagem extremamente eficaz, podendo isto ser verificado por exemplo no tempo de execução do nosso jogo.

Porém, é de salientar que esta é uma linguagem complexa na medida em que torna muitas situações de programação, que recorrem com frequência, algo complicadas, o que não acontece com outras linguagens.

Em suma, o grupo estabelece que Prolog é uma linguagem bastante capaz no que toca à resolução de problemas de elevada complexidade, com uma rapidez notável de execução.