Programação em Lógica com Restrições

Resolução de um problema “Doppelblock” em Prolog

FEUP-PLOG, Turma 3MIEIC01, Grupo Doppelblock\_4

Bruno Piedade - up201505668

Danny Soares - up201505509

Universidade do Porto, 2017/2018

**Abstract.** O objetivo deste trabalho era resolver um problema de decisão relacionado com a geração e solução do problema “Doppelblock”, utilizando programação com restrições em Prolog, da forma mais eficiente possível, evitando backtracking. Para isso desenvolvemos uma aplicação que permite resolver o problema, visualizar a resolução e verificar a sua complexidade temporal.

1. Introdução

Este trabalho foi proposto na unidade curricular “Programação em Lógica” do 3º ano do Mestrado Integrado em Engenharia Informática e Computação da Faculdade de Engenharia da Universidade do Porto, com o objetivo de desenvolver um programa em Prolog, com restrições, para a resolução de problemas de decisão ou de otimização sugeridos.

Neste caso, tratámos o problema “Doppelblock”, que é um puzzle numa grelha quadrada. Como tal, o nosso programa consiste num conjunto de predicados que permite gerar e resolver puzzles “Doppelblock” com tamanho variável, bem como a visualização da complexidade temporal da execução em função das dimensões da grelha.

Neste relatório, será feita uma descrição do problema e da nossa abordagem para o resolver. Além disso, será descrita a forma de visualização da solução e serão mostrados os resultados obtidos. No final, serão apresentadas as nossas conclusões em relação ao projeto desenvolvido.

1. Descrição do Problema

O problema *“*Doppelblock” consiste num puzzle resolvido numa grelha quadrada, com N linhas e colunas. A grelha começa completamente vazia, com um número correspondente a cada linha e a cada coluna, no exterior da grelha. Para resolver o problema, colocam-se os números de 1 até N-2 e 2 quadrados pretos em cada linha e em cada coluna, por forma a que a soma dos números entre os 2 quadrados pretos seja igual ao número que se encontra no exterior da grelha. No final a grelha fica completamente preenchida de acordo com as seguintes regras:

1. Numa mesma linha ou coluna não devem haver números repetidos.
2. Em cada linha e em cada coluna devem haver dois quadrados pretos.
3. A soma dos números entre 2 quadrados pretos da mesma linha ou coluna deve ser igual ao número no exterior da grelha correspondente a essa linha ou coluna.
4. Abordagem para solução do puzzle

A abordagem para resolver o problema, de uma forma eficiente, consistiu na determinação das variáveis de decisão, restrições e estratégia de pesquisa mais adequadas.

* 1. Variáveis de Decisão

As variáveis de decisão, neste caso, são as células da grelha quadrada. Portanto, para uma grelha de N linhas e colunas, a lista das variáveis de decisão corresponde às posições das células na grelha. Para a representação em Prolog da grelha utiliza-se uma lista de listas, em que cada uma das listas representa uma linha do tabuleiro e cada elemento dessas listas representa uma célula da grelha. Assim, a grelha é representada por uma lista com *N* listas de comprimento *N*. No estado inicial, a grelha encontra-se vazia, portanto as células da grelha aparecem “vazias”, não tendo nenhum símbolo. No estado final, as células são ocupadas por “0”, que representam os quadrados pretos, ou números de 1 até N-2, que representam os números que estão nas células. Como tal, o domínio das variáveis de decisão vai ser [0,N-2].

* 1. Restrições

As restrições descritas são aplicadas a cada linha e coluna da matriz através dos predicados *restrictRows(+Matrix, +Rows, +DiffValues, +DomainMax, +Cardinality)* e *restrictColumns(+Matrix,+Columns,+DiffValues,+DomainMax,+Cardinality)* respetivamente em que *Matrix* corresponde à matriz gerada, *Rows*/*Columns* aos índices que definem as somas, *DiffValues* ao número de valores distintos (*N*-1), *DomainMax*. ao valor máximo do domínio (*N*-2) e *Cardinality* à lista com a cardinalidade dos elementos.

### Restrição 1 – restringir a cardinalidade

Para garantir que por cada linha existiam dois 0’s e que os restantes elementos eram distintos e estavam entre a gama 1 a *N*-2 foi utilizado o predicado *global\_cardinality* cujos argumentos são gerados de acordo com o tamanho da matriz a partir do predicado *createCardinalityRestraints(+DomainMax, -Cardinality)* em que o *DomainMax* corresponde ao valor máximo do domínio (*N*-2) e a *Cardinality* é a lista de retorno que contém a cardinalidade de cada valor.

### Restrição 2 – soma entre blocos é igual a índice

Para garantir que a soma entre os “blocos” é igual ao índice indicado foi utilizado um autómato com recurso a um contador aplicando o predicado *automaton*. O autómato é definido por 3 estados - q0, q1, q2 - em que q0 corresponde ao estado inicial e q2 ao estado de aceitação. Para o estado q0 as transições possíveis são δ(q0,0,q1) para indicar que um bloco foi encontrado e δ(q0,t,q0), t ∈ [1*,N*-2] que corresponde a aceitar todas as transições que não sejam blocos. Para o estado q1 as transições são semelhantes exceto que em cada é incrementado ao contador (com valor inicial 0) o valor da transição contabilizando assim a soma entre os dois “blocos”. Desta forma, as transições são δ(q1,0,q2,C) e δ(q1,t,q1,C+t), t ∈ [1,*N*-2] em que C corresponde ao contador. Por fim, para o estado q2 as transições são δ(q2,t,q2), t ∈ [1,*N*-2] que corresponde a aceitar todos os valores diferentes de 0.

Todos os arcos (transições) são geradas de acordo com o tamanho da matriz através do predicado *createAllArcs(+DomainMax, +C, -Arcs)* em que o *DomainMax* corresponde ao valor máximo do domínio (*N*-2), *C* ao contador e *Arcs* à lista de retorno que contém todos os arcos gerados.

* 1. Estratégia de Pesquisa

Para resolver os puzzles, criámos o predicado *doppelblock(+N,+Rows,+Columns,-Res)*, que recebe o tamanho da grelha, uma lista com as somas das linhas, uma lista com as somas das colunas e retorna o puzzle resolvido.

O *labeling* da grelha é feito linha a linha, utilizando as opções *bisect* e *down* do *labeling*. A combinação destas duas opções foi a que apresentou melhores resultados em termos de tempo de execução.

1. Abordagem para geração do puzzle

A geração eficiente de puzzles é um problema diferente da resolução dos puzzles.

Para gerar puzzles válidos, a nossa abordagem foi escolher 1 puzzle aleatório de uma lista com puzles válidos. Para criar a lista com puzzles válidos, o predicado *getRandomDoppel/2* recorre ao predicado *find\_n/4*, que vai criar a lista e chamar o nosso predicado de resolução do puzzle, *doppelblock/4*, sem as linhas e colunas instanciadas, para dar puzzles com solução. Depois, da lista são recolhidos os 10 primeiros resultados e desses 10 é escolhido 1 aleatoriamente para ser apresentado ao utilizador.

1. Visualização da Solução

A visualização da grelha em modo de texto é feita com recurso ao predicado *printBoard/2*, que utiliza predicados auxiliares para exibir a grelha.

O predicado *printBoard/2* imprime a grelha com o aspeto de uma grelha real, com as células delimitadas lateralmente por ‘|’ e verticalmente por ‘\_’. Como o tamanho da grelha não é fixo, temos um predicado *printBorder(+Init,+Separator,+Times)* que imprime a string “Init” uma vez e depois imprime a string “Separator” “Times” vezes, o que permite desenhar os limites da grelha para qualquer tamanho.

As linhas da grelha são impressas com o predicado *p\_m(+Matriz,+Counter)* que usa o predicado *p\_l(+Linha,+Length)* para imprimir todas as “Linhas” da “Matriz” que representa a grelha.

Os valores diferentes de “0” nas células representam o próprio número, enquanto o valor “0” representa o quadrado preto.

Um possível estado da grelha será então:

\_\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_   
| 1 | 0 | 4 | 0 | 2 | 3 |  
|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|  
| 0 | 3 | 1 | 4 | 0 | 2 |  
|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|  
| 2 | 4 | 0 | 1 | 3 | 0 |  
|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|  
| 4 | 0 | 2 | 3 | 0 | 1 |  
|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|  
| 3 | 1 | 0 | 2 | 4 | 0 |  
|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|  
| 0 | 2 | 3 | 0 | 1 | 4 |  
|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|\_\_\_|

1. Resultados

A aplicação desenvolvida permite gerar grelhas aleatórias e válidas e permite solucionar estas mesmas, ou outras sugeridas pelo utilizador, que sejam válidas.

* 1. Geração de tabuleiros

Como foi referido na secção 4, foram utilizados dois algoritmos para geração de tabuleiro: um que garante a existência de apenas uma solução possível (nesta secção referido como “*slow*”; outro que garante que o tabuleiro gerado contém um número de peças proporcional ao tamanho do tabuleiro em si (nesta secção referido como “*fast*”). Para analisar o desempenho de ambos os algoritmos foram realizadas 10 execuções para tabuleiros de tamanho “6x6”, “8x8” e “10x10” cujos tempos de execução foram medidos com recurso ao predicado “*statistics*” do SICStus Prolog. Os resultados obtidos serão discutidos com base nos gráficos apresentados abaixo.

**Fig.** 1**.** Médias e desvios padrão dos tempos de execução dos algoritmos “*fast*” e “*slow*” no curso de 10 execuções

Através do gráfico presente na figura 1 é possível perceber que para tabuleiros pequenos (“6x6”) a diferença entre os dois algoritmos é quase inexistente. Contudo, com aumentos ligeiros no tamanho (“8x8”) a diferença torna-se bastante mais percetível, embora seja relativamente pequena. O grande impacto da melhor eficiência do algoritmo “*fast*” faz-se notar particularmente no tabuleiro “10x10” onde, embora o tabuleiro por si só não seja demasiado grande, o algoritmo “*slow*” demora em média 12 vezes mais do que o “*fast*”.

Para além disso, é ainda percetível outro efeito do aumento do tamanho do tabuleiro. Em ambos os algoritmos aumenta, de forma semelhante ao tempo médio, o desvio dos tempos de execução obtidos, o que se traduz numa maior irregularidade nos valores observados. Ou seja, por exemplo, para um tabuleiro “10x10”, é igualmente provável que o algoritmo “*slow*” demore 2 ou 22 segundos (aproximadamente).

**Fig.** 1**.** Médias dos tempos de execução de ambos os algoritmos de geração ao longo de 10 execuções

Pelo gráfico da figura 2 é visível a evolução do tempo de execução de ambos os algoritmos em função do tamanho do tabuleiro. Para aumentos ligeiros do tamanho do tabuleiro, o algoritmo “*slow*” apresenta aumentos muito mais significativos do que no algoritmo “*fast*”, mostrando uma evolução aparentemente exponencial face a uma evolução mais linear do outro algoritmo.

É importante salientar que não foram acrescentados os dados sobre tempos de execução dos algoritmos para tabuleiros ainda maiores (por exemplo “12x12”) uma vez que não acrescentariam mais informação àquela que foi apresentada e porque dificultaria a visualização dos resultados nos gráficos por questões de escala.

Apesar da clara desvantagem do algoritmo “*slow*” para tamanhos de tabuleiros superiores a “8x8”, deve-se notar a sua vantagem em termos de qualidade da solução. Dependendo do contexto a que o tabuleiro seria aplicado, a garantia de apenas existir uma solução pode traduzir-se numa grande vantagem. Assim, de uma forma geral, tendo em conta o tempo de execução e a qualidade da solução, o algoritmo “*slow*” é melhor para tabuleiros pequenos (no máximo “8x8”), mas nos restantes tabuleiros “*fast*” é superior tendo em conta a sua rapidez.

* 1. Solução de tabuleiros

No que diz respeito à solução de tabuleiros, foi feita uma comparação dos tempos de execução do predicado “*solver*” ao longo de 10 execuções, para os tamanhos “8x8”, “10x10”, “12x12” e “14x14”. Para além disso, foi testada a diferença de se colocar a opção “*enum*” na tentativa de melhorar os resultados obtidos.

**Fig.** 1**.** Tempo médio de execução do predicado “*solver*” com e sem a opção “*enum*”, ao longo de 10 execuções para tamanhos variáveis

Pelos dados da figura 3 é possível estabelecer conclusões quanto ao comportamento do predicado “*solver*” para tamanhos diferentes de tabuleiros, bem como a diferença causada pela opção “*enum*”.

Em primeiro lugar, o predicado é extremamente rápido para tabuleiros pequenos (no máximo “12x12”) uma vez que consegue resolvê-los em menos de 0,25 segundos. Para além disso, a resolução é praticamente instantânea para tabuleiros menores que 10x10. Contudo, o crescimento mostra-se exponencial em função do tamanho do tabuleiro a resolver, justificando o crescimento elevado para o tabuleiro “14x14”. Assim, para tabuleiros ainda maiores a resolução poderia durar desde alguns minutos a várias horas.

Por fim, foi observado que a presença de “*enum*” causou um pequeno aumento no tempo de execução do predicado “*solver*”, pelo que não será benéfica para acelerar a solução de tabuleiros.

1. Conclusões e Trabalho Futuro

Os resultados obtidos para a geração de um tabuleiro de jogo permitem concluir que a utilização do algoritmo implementado em “*generate\_board\_fast*” é vantajosa para tabuleiros de tamanho “10x10” ou maior. Contudo, para tabuleiros de dimensão menor o algoritmo implementado por “*generate\_board\_slow*” será preferível para serem obtidas soluções de maior qualidade.

No que diz respeito à solução de tabuleiros, o algoritmo implementado mostrou-se especialmente eficiente para tabuleiros de dimensão inferior ou igual a “12x12”. Para tabuleiros maiores o desempenho do algoritmo não é o desejável para situações em que o cálculo necessite de rapidez ou de várias execuções seguidas.

Bibliografia

Anexos

* Ficheiro “visualization.pl”, responsável pela implementação dos predicados de visualização de tabuleiros:

piece\_to\_ascii(**P**, ' ') **:-** var(P), !**.**

piece\_to\_ascii(*0*, ' ')**.**

piece\_to\_ascii(*1*, 'X')**.**

piece\_to\_ascii(*2*, 'O')**.**

print\_board([**Line**]) **:-**

print\_line(Line),

length(Line, **Size**),

print\_separating\_line(Size)**.**

print\_board([**Line1**, **Line2** |**Rest**]) **:-**

print\_line(Line1),

print\_board([Line2 | Rest])**.**

print\_board([])**.**

print\_line(**Line**) **:-**

length(Line, **Size**),

print\_separating\_line(Size),

print\_line\_cells(Line)**.**

print\_separating\_line(**Size**) **:-**

Size > *0*, !,

print\_separating\_line\_aux,

**S1** is Size - *1*,

print\_separating\_line(S1)**.**

print\_separating\_line(*0*) **:-** write('-'), nl**.**

print\_separating\_line\_aux **:-** write('----')**.**

print\_line\_cells([**Cell**|**Rest**]) **:-**

write('|'),

print\_cell(Cell),

print\_line\_cells(Rest)**.**

print\_line\_cells([]) **:-** write('|'), nl**.**

print\_cell(**Cell**) **:-**

piece\_to\_ascii(Cell, **P**),

write(' '), write(P), write(' ')**.**

write\_time(**T**) **:-** T > *1000*,

**S** is T/*1000*,

write(S), write('s')**.**

write\_time(**T**) **:-** write(T), write('ms')**.**

* Ficheiro “tictaclogic.pl”, responsável pela implementação dos predicados de geração e solução de tabuleiros:

**:-** use\_module(library(clpfd))**.**

**:-** use\_module(library(random))**.**

**:-** use\_module(library(system))**.**

**:-** use\_module(library(lists))**.**

**:-** now(**Timestamp**),

setrand(Timestamp)**.**

**:-** ensure\_loaded('visualization.pl')**.**

cross(*1*)**.**

circle(*2*)**.**

tictaclogic(**Width**, **Height**, **BoardGenerator**) **:-**

**Mw** is Width mod *2*,

Mw = *0*, *% Width must be even*

**Mh** is Height mod *2*,

Mh = *0*, *% Height must be even*

write('Generating board...'), nl,

**G** =.. [BoardGenerator, **B**, Width, Height],

statistics(walltime, [**InitTime**|\_]),

G,

statistics(walltime, [**GenTime**|\_]),

**Delta1** is GenTime - InitTime,

write('Board to be solved: '), nl, print\_board(B), nl,

write('Board generated in '), write\_time(Delta1), nl,

statistics(walltime, [**InitTime2**|\_]),

solver(B, Width, Height, [enum]),

statistics(walltime, [**SolveTime**|\_]),

write('Solution: '), nl, print\_board(B),

**Delta2** is SolveTime-InitTime2,

write('Board solved in '), write\_time(Delta2), nl**.**

generate\_board\_fast(**B**, **Width**, **Height**) **:-**

solver(**B1**, Width, Height, [variable(sel), enum]),

**Size** is Width \* Height,

**N** is Size - (Size div *7*),

board\_nonempty\_coords(B1, Width, Height, **NonEmpty**),

generate\_board\_fast\_aux(B1, B, Width, Height, NonEmpty, \_, N)**.**

generate\_board\_fast\_aux(**B**, B, \_, \_, \_, \_, *0*) **:-** !**.**

generate\_board\_fast\_aux(**B**, **NewBoard**, **Width**, **Height**, **NonEmpty**, **NonEmptyNew**, **N**) **:-**

print\_board(B),

random\_select(**Coords**, NonEmpty, **NonEmpty1**),

board\_remove\_piece(B, Coords, **B1**),

**N1** is N - *1*,

generate\_board\_fast\_aux(B1, NewBoard, Width, Height, NonEmpty1, NonEmptyNew, N1)**.**

generate\_board\_slow(**B**, **Width**, **Height**) **:-**

solver(**B1**, Width, Height, [variable(sel)]),

board\_remove\_pieces(B1, Width, Height, B)**.**

board\_remove\_pieces(**B**, **Width**, **Height**, **Result**) **:-**

print\_board(B),

board\_nonempty\_coords(B, Width, Height, **NonEmpty**),

random\_select(**Coords**, NonEmpty, **Rest**),

board\_remove\_piece(B, Coords, **B1**),

board\_copy(B1, **B2**),

findall(B2, solver(B2, Width, Height, []), **L**),

board\_remove\_pieces\_aux(B, B1, L, Width, Height, Rest, Result)**.**

board\_remove\_pieces\_aux(\_, **B1**, **L**, **Width**, **Height**, \_, **Result**) **:-**

length(L, *1*), !,

board\_remove\_pieces(B1, Width, Height, Result)**.**

board\_remove\_pieces\_aux(**B**, \_, \_, \_, \_, [], B) **:-** !**.**

board\_remove\_pieces\_aux(**B**, \_, \_, **Width**, **Height**, **NonEmpty**, **Result**) **:-**

random\_select(**Coords**, NonEmpty, **Rest**),

board\_remove\_piece(B, Coords, **B1**),

board\_copy(B1, **B2**),

findall(B2, solver(B2, Width, Height, []), **L**),

board\_remove\_pieces\_aux\_aux(B, B1, L, Width, Height, Rest, Result)**.**

board\_remove\_pieces\_aux\_aux(\_, **B1**, **L**, **Width**, **Height**, \_, **Result**) **:-**

length(L, *1*), !,

board\_remove\_pieces(B1, Width, Height, Result)**.**

board\_remove\_pieces\_aux\_aux(**B**, **B1**, **L**, **Width**, **Height**, [\_ | **T**], **Result**) **:-** board\_remove\_pieces\_aux(B, B1, L, Width, Height, T, Result)**.**

*%replace(+N, +X, +L1, -L2)*

*% Replaces the Nth member of L1 by X and instanciates L2 to the result.*

replace(**N**, **X**, **L1**, **L2**) **:-**

length(**L3**, N),

append(L3, [\_ | **T**], L1),

append(L3, [X | T], L2)**.**

board\_xy(**Board**, [**X**, **Y**], **Cell**) **:-**

nth0(Y, Board, **Line**),

nth0(X, Line, Cell)**.**

board\_copy([], []) **:-** !**.**

board\_copy([**H1** | **T1**], [**H2** | **T2**]) **:-**

board\_copy\_aux(H1, H2),

board\_copy(T1, T2)**.**

board\_copy\_aux([], []) **:-** !**.**

board\_copy\_aux([**H1** | **T1**], [\_ | **T2**]) **:-**

var(H1), !,

board\_copy\_aux(T1, T2)**.**

board\_copy\_aux([**H1** | **T1**], [H1 | **T2**]) **:-** board\_copy\_aux(T1, T2)**.**

board\_nonempty\_coords(**Board**, **Width**, **Height**, **NonEmpty**) **:-** board\_nonempty\_coords\_aux(Board, NonEmpty, Width, Height, [*0*, *0*])**.**

board\_nonempty\_coords\_aux([], \_, \_, **Height**, [*0*, Height])**.**

board\_nonempty\_coords\_aux([**Bh** | **Bt**], **NonEmpty**, **Width**, **Height**, [*0*, **Y**]) **:-**

board\_nonempty\_coords\_aux\_aux(Bh, **NonEmpty1**, Width, Height, [*0*, Y]),

append(NonEmpty1, **NonEmpty2**, NonEmpty),

**Y1** is Y + *1*,

board\_nonempty\_coords\_aux(Bt, NonEmpty2, Width, Height, [*0*, Y1])**.**

board\_nonempty\_coords\_aux\_aux([], [], **Width**, \_, [Width, \_]) **:-** !**.**

board\_nonempty\_coords\_aux\_aux([**Rh** | **Rt**], [[**X**, **Y**] | **Et**], **Width**, **Height**, [X, Y]) **:-**

nonvar(Rh), !,

**X1** is X + *1*,

board\_nonempty\_coords\_aux\_aux(Rt, Et, Width, Height, [X1, Y])**.**

board\_nonempty\_coords\_aux\_aux([\_ | **Rt**], **NonEmpty**, **Width**, **Height**, [**X**, **Y**]) **:-**

**X1** is X + *1*,

board\_nonempty\_coords\_aux\_aux(Rt, NonEmpty, Width, Height, [X1, Y])**.**

board\_random\_coords(**Width**, **Height**, [**X**, **Y**]) **:-**

**Total\_size** is Width \* Height,

random(*0*, Total\_size, **R**),

**Div** is R div Width,

**Mod** is R mod Width,

X = Div,

Y = Mod**.**

board\_remove\_piece(**Board**, [**X**, **Y**], **New\_board**) **:-**

nth0(Y, Board, **Line**),

replace(X, \_, Line, **New\_line**),

replace(Y, New\_line, Board, New\_board)**.**

solver(**B**, **Width**, **Height**, **LabelingParams**) **:-**

empty\_board(B, Width, Height),

list\_board\_vars(B, **L**),

domain(L, *1*, *2*),

*% Restrictions*

**Nw** is Width div *2*,

**Nh** is Height div *2*,

transpose(B, **B1**),

cross(**X**),

circle(**O**),

no\_more\_than\_two\_consecutive(B),

no\_more\_than\_two\_consecutive(B1),

same\_number(B, Nw, X, O),

same\_number(B1, Nh, X, O),

all\_different\_lists(B),

all\_different\_lists(B1),

*% Labeling*

labeling(LabelingParams, L)**.**

same\_number([], \_, \_, \_)**.**

same\_number([**H** | **T**], **N**, **X**, **O**) **:-**

global\_cardinality(H, [X-N, O-N]),

same\_number(T, N, X, O)**.**

all\_different\_lists([])**.**

all\_different\_lists([**H** | **T**]) **:-**

all\_different\_lists(H, T),

all\_different\_lists(T)**.**

all\_different\_lists(\_, [])**.**

all\_different\_lists(**L**, [**H** | **T**]) **:-**

different\_lists(L, H, **Bs**),

sum(Bs, #\=, *0*),

all\_different\_lists(L, T)**.**

different\_lists([], [], [])**.**

different\_lists([**L1h** | **L1t**], [**L2h** | **L2t**], [**B** | **Bs**]) **:-**

(L1h #\= L2h) #<=> B,

different\_lists(L1t, L2t, Bs)**.**

empty\_board([], \_, *0*) **:-** !**.**

empty\_board([**H** | **T**], **Width**, **Height**) **:-**

length(H, Width),

**Height1** is Height - *1*,

empty\_board(T, Width, Height1)**.**

list\_board\_vars([], [])**.**

list\_board\_vars([**Bh** | **Bt**], **L**) **:-**

append(Bh, **L1**, L),

list\_board\_vars(Bt, L1)**.**

no\_more\_than\_two\_consecutive([])**.**

no\_more\_than\_two\_consecutive([**H** | **T**]) **:-**

no\_more\_than\_two\_consecutive\_aux(H),

no\_more\_than\_two\_consecutive(T)**.**

no\_more\_than\_two\_consecutive\_aux([\_, \_])**.**

no\_more\_than\_two\_consecutive\_aux([**H**, **H2**, **H3** | **T**]) **:-**

H #\= H2 #\/ H #\= H3,

no\_more\_than\_two\_consecutive\_aux([H2, H3 | T])**.**

sel(**Vars**, **Selected**, **Rest**) **:-** random\_select(Selected, Vars, Rest), var(Selected)**.**

test\_board2(**B**, *6*, *6*) **:-**

cross(**X**),

circle(**O**),

B = [[\_, \_, \_, \_, \_, \_],

[X, \_, \_, \_, \_, X],

[\_, \_, \_, X, \_, \_],

[O, \_, \_, X, \_, \_],

[\_, O, \_, \_, \_, \_],

[\_, O, \_, \_, X, \_]]**.**

test\_board(**B**, *6*, *6*) **:-**

cross(**X**),

circle(**O**),

B = [[\_, \_, X, \_, \_, \_],

[\_, \_, X, \_, \_, \_],

[X, \_, \_, \_, \_, X],

[\_, \_, O, \_, \_, \_],

[\_, X, \_, \_, \_, X],

[O, \_, \_, \_, O, \_]]**.**

test\_board(**B**, *8*, *8*) **:-**

cross(**X**),

circle(**O**),

B = [[\_, \_, \_, X, \_, X, \_, \_],

[\_, X, \_, \_, \_, \_, \_, \_],

[\_, \_, \_, \_, X, X, \_, X],

[\_, \_, O, \_, \_, \_, O, \_],

[X, \_, \_, \_, X, \_, \_, \_],

[\_, \_, \_, O, \_, \_, X, X],

[\_, X, \_, \_, \_, \_, \_, \_],

[\_, \_, O, \_, \_, \_, O, \_]]**.**

test\_board(**B**, *10*, *10*) **:-**

cross(**X**),

circle(**O**),

B = [[\_, \_, X, \_, \_, \_, \_, X, \_, \_],

[\_, \_, \_, \_, \_, \_, \_, \_, X, \_],

[\_, O, \_, \_, \_, \_, \_, \_, \_, \_],

[\_, \_, \_, \_, \_, \_, O, \_, O, \_],

[\_, O, X, \_, \_, \_, \_, \_, \_, \_],

[\_, \_, \_, \_, \_, \_, \_, \_, \_, \_],

[\_, \_, \_, \_, O, \_, \_, \_, \_, \_],

[\_, \_, \_, \_, \_, \_, \_, \_, \_, \_],

[\_, \_, \_, \_, \_, \_, \_, \_, \_, \_],

[\_, \_, \_, \_, O, \_, \_, \_, \_, \_]]**.**