Hoo-Doo Solver

Daniel Mendonça e José Pedro Moreira

FEUP-PLOG, Turma 3MIEIC9, Grupo 123

Abstract. Este projecto consiste na implementação de um *solver* para o jogo de tabuleiro *Hoo-Doo*, jogo esse que consiste em colocar numa matriz de lado n, vários exemplares de n tipos de peças por forma a que não existam duas peças iguais na mesma linha,coluna ou diagonal. O solver que nos propomos a desenvolver funciona para uma dimensão arbitrária do tabuleiro. A implementação foi feita usando Prolog, mas concretamente a plataforma *Sicstus Prolog* tendo sido usados para tal os módulos desta mesma ferramenta para Programação em Lógica com Restrições sobre domínios finitos. Com a execução do mesmo, e olhando aos seus resultados verificámos os benefícios de usar PLR para resolução de problemas semelhantes a este.

1 Introdução

O Hoo-Doo é um jogo de tabuleiro criado nos anos 50 e que deriva do popular N Queens. Dependendo da sua dimensão, pode ter várias, uma ou até nenhuma resolução se não forem usadas *pegs* transparentes. Quando foi lançado o jogo de tabuleiro Hoo-doo, os seus criadores ofereciam 1000$ à primeira pessoa que conseguisse resolver um tabuleiro de 8x8 sem recurso a *pegs* transparentes, e, os elementos deste grupo acharam que seria interessante e até certo ponto divertido verificar como a implementação deste jogo em Prolog, usando restrições seria uma mais valia para vencer o prémio que era na altura oferecido. Apesar das poucas regras do jogo, a resolução dos puzzles pode ser penosa. Num tabuleiro de oito posições encontram-se 1.817.1052 combinações possíveis, e nem sempre, noutros tamanhos há solução possível.

Apesar de ter sido realizada uma pesquisa sobre outras implementações deste jogo de tabuleiro, nenhuma foi encontrada. A informação sobre o jogo e soluções também é praticamente inexistente, pelo que não nos foi possível comparar a nossa abordagem com outras realizadas. Na nossa implementação, o problema é resolvido com a aplicação de restrições a linhas, colunas e ambas as diagonais possíveis para cada posição do tabuleiro .

Para testar o programa criado, deve ser chamado o predicado start. Na implementação realizada do jogo, também temos a possibilidade de procurar soluções para tabuleiros rectângulares, situação não prevista no jogo original.

O trabalho desenvolvido pelos elementos do grupo tem três partes distintas:

- Parte de visualização do jogo, em muito semelhante ao já desenvolvido num outro trabalho para a disciplina;
- Parte do *solver*, que resolve o tabuleiro que lhe é passado;
- Parte de conversão de tabuleiro *flat* para tabuleiro bi-dimensional e vice-versa.

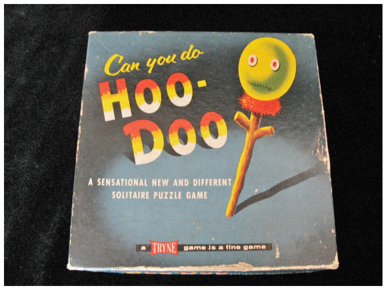
Na implementação do jogo é suportado o funcionamento em dois modos, um que não usa *pegs* transparentes, e outro que usa, tentando no entanto minimizar a sua utilização.

2 Descrição

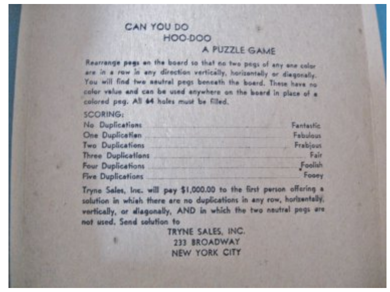
O jogo Hoo-Doo foi criado pela empresa Tryne Sales Inc., New York City, lançado na década de 50. Hoo-Doo é um jogo de tabuleiro, para um único jogador, normalmente quadrado e que tem pelo menos tantas cores quanto o número de colunas no tabuleiro, e o numero de pegs de cada cor é também o número de colunas do tabuleiro. Os tamanhos de tabuleiro mais frequentes são os de 4x4, 6x6 e 8x8.

O jogo tem como início um tabuleiro vazio, e o objectivo é preencher todas as posições do tabuleiro com as *pegs* disponíveis, sem nunca repetir peças da mesma cor na mesma linha, coluna, ou qualquer uma das diagonais. Para auxiliar na resolução do tabuleiro, existem as denominadas *pegs* transparentes, cuja característica é preencher uma posição sem lhe atribuir uma cor.

O uso de *pegs* transparentes é absolutamente necessário para a resolução de tabuleiros com determinados tamanhos (como exemplo temos um tabuleiro de 6x6, que é impossível resolver mesmo com duas *pegs* transparentes). É sempre considerada como a melhor resolução, aquela que usar menos *pegs* transparentes. O nível de dificuldade deste jogo de tabuleiro é considerável, e os próprios criadores no manual de instruções, escreveram na primeira linha *HOO-DOO is the puzzle game guaranteed not only to stump the experts but also to drive you personally and pleasantly nuts.* (HOO-DOO é um jogo de tabuleiro que garante não só desafiar os peritos, mas também levá-los à loucura).



(a) Capa original



CAN YOU DO
HOO-DOO
A PUZZLE GAME

Rearrange pegs on the board so that no two pegs of any one color are in a row in any direction vertically, horizontally or diagonally. You will find two neutral pegs beneath the board. These have no color value and can be used anywhere on the board in place of a colored peg. All 64 holes must be filled.

SCORING:

No Duplications	Fantastic
One Duplication	Fabulous
Two Duplications	Frabjous
Three Duplications	Fair
Four Duplications	Foolish
Five Duplications	Fooey

Tryne Sales, Inc. will pay $1,000.00 to the first person offering a solution in which there are no duplications in any row, horizontally, vertically, or diagonally, AND in which the two neutral pegs are not used. Send solution to

TRYNE SALES, INC.
233 BROADWAY
NEW YORK CITY

(b) Regras

3 Variáveis de Decisão

Para modelar o problema em causa, usamos uma variável por cada posição no tabuleiro, perfazendo para um tabuleiro $n*m$ (onde n é a altura do tabuleiro e m é a largura do mesmo), $n*m$ variáveis.O domínio dessas variáveis depende também ele do tamanho do tabuleiro, sendo que existem para cada tipo de tabuleiro dois domínios distintos dependendo de o jogo ser resolvido usando peças transparentes ou não. Para o caso de serem usadas peças transparentes o domínio de cada uma das variáveis é $[0, n]$ (assumindo que n é a maior dimensão do tabuleiro). No caso de se recorrer somente a peças com cor o domínio de cada uma das variáveis passa a ser $[1, n]$ (assumindo que n é a maior dimensão do tabuleiro).

No caso de o jogo ser resolvido recorrendo aos *pegs* transparentes é ainda utilizada uma variável extra, variável essa que conta o número de *pegs* transparentes no tabuleiro. O domímio desta variável vai desde zero até $n*m$, caso em que todas as peças do tabuleiro são *pegs* transparentes.

4 Restrições

Tratando-se de um jogo todas as restrições que incluímos são rígidas, à excepção de uma. As restrições rígidas são:

- A restrição que impede que duas peças da mesma linha tenham valores iguais;
- A restrição que impede que duas peças da mesma coluna tenham valores iguais;
- A restrição que impede que duas peças da mesma diagonal tenham valores iguais;

A restrição flexível, diz respeito á tentativa de minimizar o número de peças transparentes envolvidas na resolução do tabuleiro (que é por si também um dos objectivos do jogo).

A implementação das restrições rígidas é feita agrupando as variáveis em listas, ora por linha, ora por coluna, ora por diagonal , e chamando o predicado *all_distinct* sobre essas listas, obrigando assim que nunca haja duas peças que partilhem a mesma linha/coluna/diagonal que tenham o mesmo valor.

A implementação da restrição flexível, é feita pela introdução de uma variável antes do *labeling*, a qual se passa ao predicado *count* , com o intuito de contar o número de peças com valor zero no tabuleiro (número de pegs transparentes).

Esta última restrição, é implementada por meio da tentativa de minimização do valor da variável supra mencionada. Este efeito é conseguido passando como opção de *labeling* o predicado *minimize* passando-lhe a variável criada propositadamente para o efeito. Esta restrição permite assim tentar sempre obter a solução para o tabuleiro que usa menos *pegs* transparentes.

5 Estratégia de Pesquisa

Na estratégia de pesquisa utilizada, quanto o jogo é jogado com a possibilidade de introduzir *pegs* transparentes, optamos por incluir as seguintes opções para a etiquetarem das variáveis:

– *down* : permitindo assim que o domínio seja percorrido em ordem descendente;
– *minimize(X)* : para aplicar a restrição flexível ;
– *time_out(Time,Flags)* : para evitar que a execução se prolongue por tempo indeterminado;
– *ffc* : para tentar falhar o mais rapidamente possível,"cortando" a àrvore de pesquisa mais cedo e eliminando assim mais nós;

O *down* foi utilizado por razões obvias: como as *pegs* transparentes são codificadas com o número zero, e como o domínio das variáveis vai tipicamente de zero até n, em que n é o lado do tabuleiro, ao percorrer o domínio de valores maiores até zero obtemos melhores resultados mais rápidamente em geral. Isto tendo em conta que uma melhor solução é aquela que usa menos *pegs* transparentes.

O *minimize(X)* foi também ele utilizado por razões já explicadas anteriormente, que se prendem com a necessidade de tentar encontrar a solução que minimiza o número de *pegs* transparentes. Nesse sentido a variável que é passada ao predicado *minimize(X)* é a variável que conta o número de zeros no tabuleiro.

O predicado *time_out(Time,Flags)* é usado para que a execução não decorra indeterminadamente. Com esse fim é passado ao predicado o valor de *120000* que faz com que o *labeling* decorra durante no máximo 2 minutos e retorne então, senão antes, a melhor solução até ao momento.

A opção *ffc* foi utilizada na tentativa de tentar que fossem atribuídos primeiro valores as variáveis mais restritas, tentando assim "cortar" mais ramos da árvore de pesquisa, ao falhar mais cedo.

Como é referido na secção própria, verificamos que a utilização de outras opções como por exemplo *bisect* , resulta em resultados ligeiramente menos satisfatórios.

Quando é usado o *bisect* a performance é sensivelmente pior do que no caso de se usar a seleção de valores por defeito, provavelmente porque o beneficio que trás é quase nulo, e o custo do seu calculo é naturalmente algum, ainda que diminuto, causando então uma pequena degradação da performance.

6 Visualização

Existem seis predicados utilizados para a construção visual do tabuleiro em modo de texto. O primeiro predicado a ser executado é o *print_tab(+board)* que recebe como argumento um tabuleiro representado por uma lista de listas de inteiros. Este predicado calcula o comprimento da lista, e das sub-listas. Obtidos os números de linhas e colunas do tabuleiro em questão, procede-se à impressão. Enquanto a cauda da lista não for vazia, isto é, ainda existirem mais linhas, segue-se um procedimento básico, que consiste em imprimir primeiro uma linha horizontal, numa nova linha verificar o tamanho do índice, para calcular o número de caracteres vazios a escrever entre o índice e início do tabuleiro(este processo de escalonamento do espaço para alinha a impressão também é usado para os valores das pegs no interior do tabuleiro), e só de seguida é impressa a linha de valores, que é a primeira lista. Este ciclo repete-se até chegar então à situação onde a cauda da lista é vazia.

Quando a cauda da lista é vazia, a primeira parte do processo mantém-se(impressão da linha horizontal, e também dos valores com o respectivo alinhamento), mas após a impressão dos valores da lista, é de imediato impressa outra linha horizontal e seguida do índice das colunas com valores Alfa-numéricos. Vemos em baixo um exemplo de tabuleiro impresso na consola:

```
   ---------------------------------
8 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |
   ---------------------------------
7 |  6 |  5 |  8 |  7 |  2 |  1 |  4 |  3 |
   ---------------------------------
6 |  7 |  4 |  3 |  6 |  8 |  0 |  5 |  2 |
   ---------------------------------
5 |  3 |  8 |  5 |  0 |  0 |  4 |  7 |  6 |
   ---------------------------------
4 |  2 |  1 |  4 |  0 |  6 |  5 |  3 |  8 |
   ---------------------------------
3 |  4 |  6 |  2 |  1 |  3 |  7 |  0 |  5 |
   ---------------------------------
2 |  0 |  3 |  7 |  8 |  5 |  2 |  1 |  4 |
    ---------------------------------
1 |  5 |  2 |  1 |  4 |  7 |  6 |  8 |  0 |
   ---------------------------------
     A    B    C    D    E    F    G    H
```

Fig. 1: *Tabuleiro 8x8 resolvido*

7 Resultados

No processo de desenvolvimento do solver fizemos vários testes utilizando várias opções de etiquetarem das variáveis, por forma a tentar obter o melhor resultado possível, o mais rapidamente possível, na maioria dos casos. O estudo que fizemos, centrou-se principalmente no caso em que é possível usar *pegs* transparentes, tentando no entanto minimizar o seu uso, por este ser o caso mais interessante. Dividimos o estudo em duas partes, na primeira parte analisamos o comportamento das várias opções tendo em conta parâmetros como o número de *backtracks* ou o tempo de execução, sendo este estudo feito para aqueles tabuleiros com que conseguimos garantidamente alcançar uma solução optima. A segunda parte do estudo consistiu na comparação do número de *pegs* transparentes utilizado na resolução de alguns dos problemas para os quais não temos uma solução optima, isto utilizando também diferentes opções de etiquetagem. Esta divisão deveu-se a que, em problemas em que a solução encontrada não é optima, de pouco interessa saber métricas como o número de *backtracks* ou o tempo de execução (que será sempre o mesmo), interessa ao invés disso saber qual a melhor solução de todas as encontradas com as diversas opções.

No que diz respeito aos problemas para os quais se tem uma resolução optima, verificou-se que o *down* leva à mais rápida resolução do problema na maior parte dos casos. Facto que pode ser verificado comparando os dados das tabelas 1, 2, 3 e 4 (em anexo). Isto no entanto não se verifica para tamanhos de tabuleiros maiores, nomeadamente, para tamanhos superiores a 6x6, onde a etiquetagem com as opções *down* e *ffc* revela-se a mais rápida e a que tem consistentemente melhores resultados. Esse facto começa-se a observar no tabuleiro 7x7 onde a etiquetagem com as opções supra referidas começa já a ser melhor e mais rápida que as outras, resultando em que o tempo de execução seja menor, isto embora o número de *prunings* seja menor e o número de *backtracks* maior. Esta tendência torna-se evidente para os tabuleiros de maior tamanho onde nunca se chega a alcançar uma solução optima mas, onde os resultados da etiquetarem com as opções *down* e *ffc* dão soluções francamente melhores. Esses factos podem ser observados principalmente nas tabelas 5, 6, 7, 8 e 9, tabelas essas onde se mostram o número de *pegs* transparentes utilizadas em resoluções com alguns tempos de duração máximos selecionados. Nessas tabelas é notório o beneficio de usar a combinação de opções supra mencionadas pois produzem consistentemente resoluções melhores (com menos *pegs* transparentes) e mais cedo do que as outras combinações. Esse facto é também aludido nos gráficos apresentados, (2, 3 e 4), onde se pode ver que claramente a pior opção de entre as testadas é utilizar os parâmetros por defeito, que retornam consistentemente piores resultados que qualquer uma das outras opções. De entre as outras opções na grande maioria dos casos, a alternativa que usa as opções *down* juntamente com *ffc* obtem ligeiramente melhores resultados. A opção de usar a combinação de opções *down* com *ffc* e *bisect* em simultâneo (não incluída em nenhum dos gráficos por uma questão de clareza dos mesmo) foi estudada com a ajuda da tabela 9, no entanto os resultados que se conseguem obter são em tudo semelhantes aos em que se obtem quando se utiliza somente *down* e *ffc*, sendo que em alguns casos tem resultados ligeiramente piores, e noutros ligeiramente melhores, pelo que não é uma opção claramente vantajosa.

Apesar da tentativa de contacto com colegas de outros grupos, com vista á troca de resultados, obtivemos apenas uma resposta, de um colega que utilizou uma abordagem muito diferente da nossa que não permite portanto fazer uma comparação dos resultados.

8 Conclusões

Com a realização do presente trabalho pudémos verificar a simplicidade e elegância com que, a programação em lógica com restrições, nos permite implementar soluções para problemas como o $Hoo-Doo$. A implementação é não só compacta, facilmente legível, como é também de uma elegância e simplicidade que não seria facilmente alcançavel com o uso de uma linguagem imperativa como *C* ou *Java*. Parece-nos no entanto, que essa simplicidade acarreta alguns custos a nível de performance, uma vez que nos parece, isto no entanto sem termos dados concretos que sustentem esta hipótese, que uma implementação numa linguagem imperativa do tipo *C*, utilizando um algoritmo adequado, teria uma performance bastante melhor. É no entanto indiscutível, a vantagem de utilizar PLR nesta situação, uma vez que a implementação do *solver* é muito mais simples e clara. Tivemos ainda oportunidade de atestar a versatilidade que as opções do predicado *labeling* trazem á resolução deste tipo de problemas, dotando o programador da possibilidade de fazer pequenos ajustes que permitem melhorar dramaticamente a performance do programa. Caso tivéssemos tido mais tempo poderíamos eventualmente fazer uma rápida implementação de um solver numa linguagem imperativa, para termos uma noção mais exacta de quais as diferenças em termos de performance que existem entre estes dois tipos de abordagens, atestando assim a aplicabilidade e utilidade da PLR em situações semelhantes a esta.

Bibliografia

1. Jaap. Hoo Doo. `http://www.jaapsch.net/puzzles/hoodoo.htm`. [Online; accessed 15-Dezembro-2013].
2. Sicstus. *Sicstus User Manual*. Sicstus, SE 164 29 Kista, Sweden, 4.2.1 edition, February 2012.

A Codigo

A.1 HooDoo.pro

```
/* -*- Mode:Prolog; coding:iso-8859-1; -*- */



%%%%%3x3 [[a,b,c],[d,e,f],[g,h,i]]
%%%%%4X4 [[1,a,c,v],[,e,3,4,5],[,a,s,d,y],[l,k,j,h]]

:-include('flat_2D_convert.pro').
:-include('tabPrint.pro').
:-include('custom_all_distinct.pro').
:-use_module(library(timeout)).
:-use_module(library(clpfd)).





%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Input validation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

checkValidSize(0, _, _):-!, halt.
checkValidSize(_, 0, _):-!, halt.
checkValidSize(NLines, NColumns, Status):-
       number(NLines),
       number(NColumns),
       NLines > 0,
       NLines < 25,
       NColumns > 0,
       NColumns < 25,
       Status=ok.




executeMenuCommand('1',ok).
executeMenuCommand('2',ok):-!, halt.
executeMenuCommand(Input,Output):- !,Input\=2,Input\=1,Output=bad.


menu:-
       write('1- Choose board sizes\n'),
       write('2- Exit\n').
```

```
getSize(NLines, NColumns):-
       write('Write the number of lines you want followed by ".",\nWrite
           0 to exit'), nl,
       read(NLines), nl,
       write('Write the number of columns you want followed by
           "."\nWrite 0 to exit'), nl,
       read(NColumns), nl.


isValidTransparency('0', 0).
isValidTransparency('1', 1).
isValidTransparency('2', _):-!, halt.
isValidTransparency(_, bad).

getTransparency(Transparency):-
       write('Do you wish to solve with transparent pegs?\n'),
       write('0- No Transparent pegs\n1-With transparent
           pegs\n2-Exit\n'),
       get_char(_),
       get_char(Option),
       get_char(_),
       isValidTransparency(Option, Transparency),
       Option \=bad;
       !,
       write('Invalid Option, try again\n'),
       getTransparency(Transparency).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Game Start
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%




start:-
       write('###########Hoo-Doo##########\n'), nl,
       menu, !,
       get_char(Input),
       get_char(_), %consume enter key
       executeMenuCommand(Input,Output),
       Output=ok,
       getSize(NLines, NColumns),
       !,
       checkValidSize(NLines, NColumns, Is_ok),
       Is_ok= ok,
```

```
        getTransparency(Transparency),
        Transparency \= bad,
        solve(SolveBoard, NLines, NColumns , Transparency), !, nl,
        print_tab(SolveBoard);
        write('You chose an invalid option or board size\n\n\n'),
        !,
        start.


solve(SolvedBoard,NrLines,NrColumns,TransparentMode):-
        DesiredSize is NrLines*NrColumns,%plus one for the Transparent
        generateFlatList(Board,DesiredSize),
        applyConstraints(Board,NrLines,NrColumns,TransparentMode),
        count(0,Board,#=,Soma),
        append(Board,[Soma],TodasAsVars),
        statistics,
        !,
        labeling([minimize(Soma),down,ffc,bisect,time_out(120000,Flag)],TodasAsVars),
        statistics(runtime,Value),
        nl,
        write('Tempo decorrido :'),
        write(Value),
        nl,
        write('Pegs Transparentes Utilizadas:'),
        write(Soma),
        nl,
        fd_statistics,
        inflate(SolvedBoard, Board, NrLines, NrColumns),
        nl,
        write(Flag),
        nl.


test(Bi):-
        generateFlatList(Board,36),
        applyConstraints(Board,6,6,1),
        %sum(Board,#=,Soma),
        count(0,Board,#=,Soma),
        append(Board,[Soma],TodasAsVars),
        statistics,
        labeling([minimize(Soma),down,ffc,bisect,time_out(120000,Flag)],TodasAsVars),
        statistics(runtime,Value),
        nl,
        write('Tempo decorrido :'),
        write(Value),
        nl,
        write('Transparentes :'),
        write(Soma),
        nl,
        fd_statistics,
```

```
        inflate(Bi,Board,6,6),
        nl,
        print_tab(Bi), nl,
        write(Flag),
        nl.



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Generate List
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
generateFlatList(Board,DesiredSize):-
        Index is 0,
        generateFlatListAux(DesiredSize, Index, [], Board).


generateFlatListAux(DesiredSize, DesiredSize, Board, Board).

generateFlatListAux(DesiredSize, Index, Progress, Board):-
                append(Progress, [_], Nprogress),
                Next_index is Index+1,
                generateFlatListAux(DesiredSize,Next_index, Nprogress,
                    Board).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Apply Hoo-Doo Constraints
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

applyConstraints(FlatBoard,NrLines,NrColumns,UseTransparent):-
        max_member(UpperValue,[NrColumns,NrLines]),
        %apply domain constraints
        (UseTransparent =
            0,!,domain(FlatBoard,1,UpperValue);domain(FlatBoard,0,UpperValue)),
        inflate(Inflated, FlatBoard,NrLines,NrColumns),
        applyLineConstraints(Inflated,UseTransparent),
        applyColumnConstraints(Inflated,NrColumns,UseTransparent),
        applyDiagonalConstraints(Inflated,NrLines,NrColumns,UseTransparent).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Apply Line Constraints
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
applyLineConstraints([BoardHead|BoardTail],UseTransparent):-
        custom_all_distinct(BoardHead,UseTransparent,0),
        !,
        applyLineConstraints(BoardTail,UseTransparent).
```

```
applyLineConstraints([],_).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Get Columns
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getColumn(Board,Col,ColNr):-
	getColumn(Board,Col,ColNr,[]).

getColumn([BoardHead|BoardTail],Col,ColNr,Tmp):-
	nth0(ColNr,BoardHead,Elem),
	append(Tmp,[Elem],NewTmp),
	!,
	getColumn(BoardTail,Col,ColNr,NewTmp).
getColumn([],Col,_,Tmp):-
	Col = Tmp.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Apply Column Constraints
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


applyColumnConstraints(Board,NrColumns,UseTransparent):-
	applyColumnConstraints(Board,NrColumns,0,UseTransparent).

applyColumnConstraints(Board,NrColumns,Nr,UseTransparent):-
	Nr<NrColumns,
	getColumn(Board,Col,Nr),
	custom_all_distinct(Col,UseTransparent,0),
	NewNr is Nr+1,
	!,
	applyColumnConstraints(Board,NrColumns,NewNr,UseTransparent).
applyColumnConstraints(_,NrColumns,NrColumns,_).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Find l Diagonal
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getlDiagonalStartingPosition(Side,DiagonalNr,Line,Col):-
	(DiagonalNr>=Side, Pos is (DiagonalNr mod Side)+1;Pos is
	    (Side-DiagonalNr-1)*Side),
	Line is Pos // Side,
```

```
        Col is Pos mod Side,
        !.


getLDiagonal(NrLines,NrColumns,Board,Diagonal,DiagonalNr):-
        max_member(Max,[NrLines,NrColumns]),
        getlDiagonalStartingPosition(Max,DiagonalNr,Line,Col),%get
            position for the square containing the board
        !,
        getLDiagonal(NrLines,NrColumns,Board,Diagonal,DiagonalNr,[],Line,Col).


%case where the position is outside the board and the square containing
    it
getLDiagonal(NrLines,NrColumns,_,Diagonal,_,Tmp,CurrentCol,CurrentLine):-
        max_member(Max,[NrLines,NrColumns]),
        (CurrentCol>=Max;CurrentLine>=Max),
        !,
        Diagonal=Tmp.%just return what you have so far



getLDiagonal(NrLines,NrColumns,Board,Diagonal,DiagonalNr,Tmp,CurrentLine,CurrentCol):-
        CurrentLine<NrLines,CurrentCol<NrColumns,
        CurrentLine>=0,CurrentCol>=0,
        NewCol is CurrentCol+1,
        NewLine is CurrentLine+1,
        nth0(CurrentLine,Board,Line),
        nth0(CurrentCol,Line,Elem),
        append(Tmp,[Elem],NewTmp),
        !,
        getLDiagonal(NrLines,NrColumns,Board,Diagonal,DiagonalNr,NewTmp,NewLine,NewCol).


%case where it is outside the board but not outside the square that
    constains it probably
%because the initial point of the diagonal is in a position that is not
    present in this
%board but only on the square that contains it. Remember that
    getlDiagonalStartingPosition
%gets the position in the square that constains the board not the board
    itself
%being so it may happen that the initial position is outside the board

getLDiagonal(NrLines,NrColumns,Board,Diagonal,DiagonalNr,Tmp,CurrentCol,CurrentLine):-
        NewCol is CurrentCol+1,
        NewLine is CurrentLine+1,
        %just increment position hoping it will get inside the board
        !,
        getLDiagonal(NrLines,NrColumns,Board,Diagonal,DiagonalNr,Tmp,NewLine,NewCol).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Find r Diagonal
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



getrDiagonalStartingPosition(Side,DiagonalNr,Line,Col):-
       (DiagonalNr<Side,Pos=DiagonalNr;DiagonalNr>=Side,Pos is
           (DiagonalNr-Side+2)*Side-1),
       Line is Pos // Side,
       Col is Pos mod Side,
       !.


%refer to getLDiagonal for comments on the implementation (they are the
    same for both cases)

%both both predicates are similar, differing only on the increment
    function for NewLine and NewCol
%that defines the next position in the diagonal

getRDiagonal(NrLines,NrColumns,Board,Diagonal,DiagonalNr):-
       max_member(Max, [NrLines,NrColumns]),
       getrDiagonalStartingPosition(Max,DiagonalNr,StartingLine,StartingCol),
       getRDiagonal(NrLines,NrColumns,Board,Diagonal,DiagonalNr,[],StartingLine,StartingCol).

getRDiagonal(NrLines,NrColumns,_,Diagonal,_,Tmp,CurrentLine,CurrentCol):-
       max_member(Max,[NrLines,NrColumns]),
       (CurrentLine<0;CurrentCol<0;CurrentLine>=Max;CurrentCol>=Max),
       !,
       Diagonal=Tmp.

getRDiagonal(NrLines,NrColumns,Board,Diagonal,DiagonalNr,Tmp,CurrentLine,CurrentCol):-
       CurrentLine<NrLines,CurrentCol<NrColumns,
       CurrentLine>=0,CurrentCol>=0,
       NewLine is CurrentLine+1,
       NewCol is CurrentCol-1,
       nth0(CurrentLine,Board,Line),
       nth0(CurrentCol,Line,Elem),
       append(Tmp,[Elem],NewTmp),
       !,
       getRDiagonal(NrLines,NrColumns,Board,Diagonal,DiagonalNr,NewTmp,NewLine,NewCol).

getRDiagonal(NrLines,NrColumns,Board,Diagonal,DiagonalNr,Tmp,CurrentLine,CurrentCol):-
       NewLine is CurrentLine+1,
       NewCol is CurrentCol-1,
       !,
```

```
        getRDiagonal(NrLines,NrColumns,Board,Diagonal,DiagonalNr,Tmp,NewLine,NewCol).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Apply Diagonal Constraints
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


applyDiagonalConstraints(Board,NrLines,NrColumns,UseTransparent):-
        applyRDiagonalConstraints(Board,NrLines,NrColumns,1,UseTransparent),
        applyLDiagonalConstraints(Board,NrLines,NrColumns,1,UseTransparent).


applyRDiagonalConstraints(Board,NrLines,NrColumns,DiagonalNr,UseTransparent):-
        max_member(Max,[NrLines,NrColumns]),
        DiagonalNr<2*Max-2,
        NextDiagonalNr is DiagonalNr +1,
        getRDiagonal(NrLines,NrColumns,Board,Diagonal,DiagonalNr),
        custom_all_distinct(Diagonal,UseTransparent,0),
        !,
        applyRDiagonalConstraints(Board,NrLines,NrColumns,NextDiagonalNr,UseTransparent).
applyRDiagonalConstraints(_,NrLines,NrColumns,DiagonalNr,_):-
        max_member(Max,[NrLines,NrColumns]),
        DiagonalNr>=2*Max-2.


applyLDiagonalConstraints(Board,NrLines,NrColumns,DiagonalNr,UseTransparent):-
        max_member(Max,[NrLines,NrColumns]),
        DiagonalNr<2*Max-2,
        NextDiagonalNr is DiagonalNr +1,
        getLDiagonal(NrLines,NrColumns,Board,Diagonal,DiagonalNr),
        custom_all_distinct(Diagonal,UseTransparent,0),
        !,
        applyLDiagonalConstraints(Board,NrLines,NrColumns,NextDiagonalNr,UseTransparent).
applyLDiagonalConstraints(_,NrLines,NrColumns,DiagonalNr,_):-
        max_member(Max,[NrLines,NrColumns]),
        DiagonalNr>=2*Max-2.
```

A.2 custom_all_distinct.pro

```
/* -*- Mode:Prolog; coding:iso-8859-1; -*- */


:-use_module(library(clpfd)).


%second parameter is to use dont care or not to use it
custom_all_distinct(List,0,_):-
        all_distinct(List).
```

```
custom_all_distinct([ListHead|Tail],1,DontCareValue):-
        apply_all_distinct_constraints([ListHead|Tail],1,DontCareValue),
        !,
        custom_all_distinct(Tail,1,DontCareValue).

custom_all_distinct([],1,_).



apply_all_distinct_constraints([ListHead,ListTailHead|ListTail],1,DontCareValue):-
        ((ListHead #\= ListTailHead) #\/ (ListHead#=DontCareValue)),
        !,
        apply_all_distinct_constraints([ListHead|ListTail],1,DontCareValue).

apply_all_distinct_constraints([_],1,_).
```

A.3 flat_2D_convert.pro

```
/* -*- Mode:Prolog; coding:iso-8859-1; -*- */

:-use_module(library(clpfd)).
:-use_module(library(lists)).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Creates a 2D board from a flat list %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

inflate(Bidimentional,Flatten,NrLines,NrColumns):-
        inflate(Bidimentional,Flatten,[],NrLines,NrColumns,0).


inflate(B,_,Tmp,NrLines,_,LineNr):-
        LineNr = NrLines,
        B=Tmp.

inflate(B,F,Tmp,NrLines,NrColumns,LineNr):-
        LineNr<NrLines,
        generateLine(F,Line,LineNr,NrColumns),
        append(Tmp,[Line],NewTmp),
        NewLineNr is LineNr+1,
        !,
        inflate(B,F,NewTmp,NrLines,NrColumns,NewLineNr).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Makes a 2D list a Flat one%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

flat(Bidimentional,Flat,NrLines):-
       flat(Bidimentional,Flat,[],NrLines,0).

flat(_,F,Tmp,NrLines,LineNr):-
       LineNr=NrLines,
       F=Tmp.

flat(B,F,Tmp,NrLines,LineNr):-
       LineNr<NrLines,
       Pos is LineNr +1,
       nth1(Pos,B,Line),%not zero based
       append(Tmp,Line,NewTmp),
       NewLineNr is LineNr +1,
       !,flat(B,F,NewTmp,NrLines,NewLineNr).




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Extracts Line in a given postion from a flat board%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
generateLine(FlatBoard,Line,LineNr,LineLength):-
       generateLine(FlatBoard,Line,LineNr,LineLength,[],0).

generateLine(FlatBoard,Line,LineNr,LineLength,Tmp,Pos):-
       Pos<LineLength,
       PositionInBoard is LineLength*LineNr+Pos+1,%+1 because is not
           zero based
       element(PositionInBoard, FlatBoard, Elem),
       append(Tmp,[Elem],NewTmp),
       NewPos is Pos+1,
       !,generateLine(FlatBoard,Line,LineNr,LineLength,NewTmp,NewPos).

generateLine(_,Line,_,LineLength,Tmp,Pos):-
       Pos is LineLength,
       Line =Tmp.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.4 tabPrint.pro

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Print Solution Board
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



tab_map(NUM):-
       write(NUM).
%tab_map(_):- write('0').


print_line([X|[]]):-
       X < 10,
    write('| '),
    tab_map(X),
    write(' |').

print_line([X|[]]):-
  X >= 10,
    write('| '),
    tab_map(X),
    write(' |').



print_line([X|Xs]):-
  X < 10,
    write('| '),
    tab_map(X),
    write(' '),
    print_line(Xs).

print_line([X|Xs]):-
  X >= 10,
    write('| '),
    tab_map(X),
    write(' '),
    print_line(Xs).
```

```
print_empty_line(Len):-
     Len > 1,
     write('-----'),
     LenNext is Len - 1,
     print_empty_line(LenNext).

print_empty_line(1). %:-
     % write('-----').



print_column_index(_,0):- nl.
print_column_index(ASCIICode,I):-
     I > 0,

     char_code(C,ASCIICode),
     write(C), write('  '),

     ASCIICodeNext is ASCIICode + 1,
     Inext        is I - 1,
     print_column_index(ASCIICodeNext, Inext).



print_tab_aux([X|[]], 1, Ci):-
  write('   '),
  print_empty_line(Ci), nl,
  write(' '), write(1), write(' '),
  print_line(X), nl,
  write('  '),
  print_empty_line(Ci), nl,
  write('     '),

  char_code('A',ASCIICode),
  Ci1 is Ci-1,
  print_column_index(ASCIICode, Ci1).



print_tab_aux([X|Xs], Li, Ci):-
     Li > 9,

     write('   '),
     print_empty_line(Ci), nl,


     write(Li),
     write(' '),
```

```
     print_line(X), nl,


     LiNext is Li - 1,
     print_tab_aux(Xs, LiNext, Ci).

print_tab_aux([X|Xs], Li, Ci):-
     Li =< 9,
     write('  '),
     print_empty_line(Ci), nl,

     write(' '),
     write(Li),
     write(' '),

     print_line(X), nl,

     LiNext is Li - 1,
     print_tab_aux(Xs, LiNext, Ci).

print_tab([X|Xs]):-
     length([X|Xs],Vi),
     length(X,Hi),
     Hi1 is Hi+1,
     print_tab_aux([X|Xs], Vi, Hi1).
```

B Tabelas e Gráficos

Table 1: Problemas com solução optima : Opções por defeito

Métrica	2x2	3x3	4x4	5x5	7x7
Runtime	10	10	125	10	10900
Resumptions	304	9141	478328	16882	33054848
Entailments	161	4669	234053	9878	29865115
Prunings	205	5126	240508	10307	17858406
Backtracks	8	129	3194	194	346785
Constraints	27	103	261	531	1527

Table 2: Problemas com solução optima : Opção *down*

Métrica	2x2	3x3	4x4	5x5	7x7
Runtime	10	10	125	10	10900
Resumptions	304	9141	478328	16882	33054848
Entailments	161	4669	234053	9878	29865115
Prunings	205	5126	240508	10307	17858406
Backtracks	8	129	3194	194	346785
Constraints	27	103	261	531	1527

Table 3: Problemas com solução optima : Opções *down* e *ffc*

Métrica	2x2	3x3	4x4	5x5	7x7
Runtime	10	10	90	10	59180
Resumptions	286	7667	268519	14254	295555138
Entailments	134	3776	139331	6073	127518166
Prunings	177	4180	139466	6244	118181814
Backtracks	5	90	1748	107	1727869
Constraints	27	103	261	531	1527

Table 4: Problemas com solução optima : Opções *down* e *bisect*

Métrica	2x2	3x3	4x4	5x5	7x7
Runtime	10	10	80	10	117930
Resumptions	286	8438	277571	9634	491394424
Entailments	134	4114	133215	3656	249165687
Prunings	182	4565	136734	3896	225597907
Backtracks	5	102	1611	51	1036895
Constraints	27	103	261	531	1527

Table 5: Número de *pegs* transparentes usadas apos timeout : Opções por defeito

Tempo(s)	6x6	8x8	9x9	10x10	11x11	20x20
2	7	11	18	25	32	265
10	7	8	16	22	30	248
120	6	8	13	21	24	185

Table 6: Número de *pegs* transparentes usadas apos timeout : Opções *down*

Tempo(s)	6x6	8x8	9x9	10x10	11x11	20x20
2	6	11	16	21	26	79
10	4	11	16	20	26	78
120	4	7	15	19	26	78

Table 7: Número de *pegs* transparentes usadas apos timeout : Opções *down* e *ffc*

Tempo(s)	6x6	8x8	9x9	10x10	11x11	20x20
2	5	11	12	17	21	68
10	4	11	11	17	21	68
120	4	11	11	16	21	67

Table 8: Número de *pegs* transparentes usadas apos timeout : Opções *down* e *bisect*

Tempo(s)	6x6	8x8	9x9	10x10	11x11	20x20
2	6	11	16	21	26	78
10	4	11	16	20	26	78
120	4	7	15	19	24	78

Table 9: Número de *pegs* transparentes usadas apos timeout : Opções *down*, *bisect* e *ffc*

Tempo(s)	6x6	8x8	9x9	10x10	11x11	20x20
2	5	11	12	18	21	68
10	4	11	11	18	21	67
120	4	11	11	17	21	67

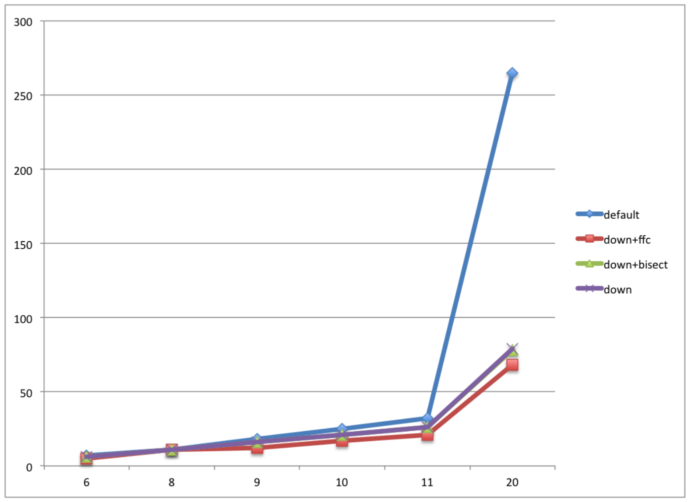


Fig. 2: Número de *pegs* transparentes ao fim de 1 segundo

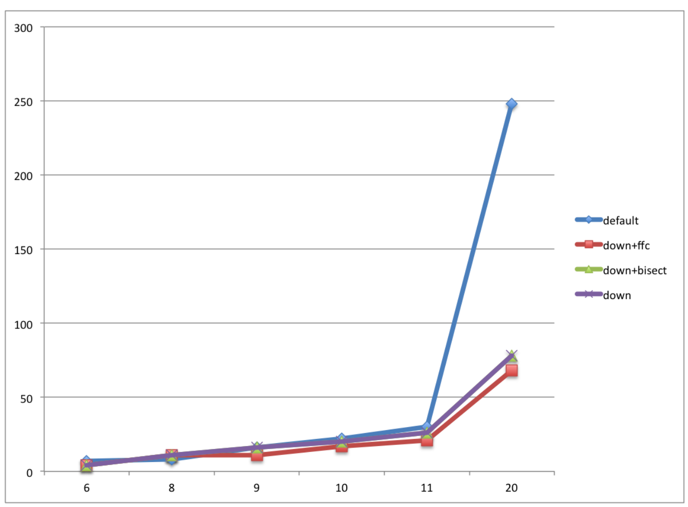


Fig. 3: Número de *pegs* transparentes ao fim de 2 segundos

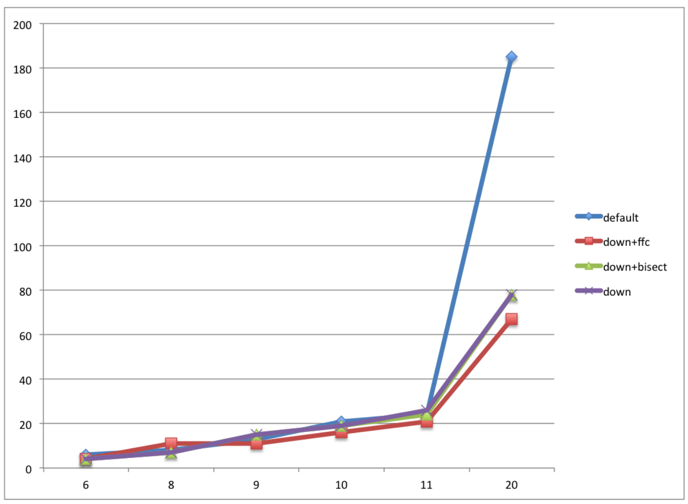


Fig. 4: Número de *pegs* transparentes ao fim de 2 minutos