

Weight Puzzle

Daniel Ferreira Brandão^[up201705812]

e

Pedro Miguel Braga Barbosa Lopes Moás^[up201705208]

Faculdade de Engenharia Universidade do Porto
Mestrado Integrado em Engenharia Informática e Computação
FEUP-PLOG, Turma 3MIEIC4, Grupo Weight_5

Resumo: Este artigo tem como objetivo demonstrar o processo de desenvolvimento do segundo projeto da Unidade Curricular de Programação em Lógica, assim como os resultados que se obtiveram. O projeto consiste num programa escrito em Prolog, capaz de resolver qualquer instância do problema de decisão Weight, um puzzle cuja descrição poderá ser encontrada em <https://www2.stetson.edu/~efriedma/puzzle/weight/>. Este foi modelado como um PSR (Problema de Satisfação de Restrições) e resolvido utilizando a biblioteca ‘clpfd’ do SICStus Prolog, que permite implementação de PLR (Programação em Lógica com Restrições). O programa também possui a capacidade de gerar novos puzzles dinamicamente, com qualquer tamanho. Após análise de eficiência, concluiu-se que o *solver* executa de forma quase sempre instantânea para problemas de dimensão até 30, e em alguns segundos até cerca de 35, subindo sempre de modo exponencial. O gerador de problemas consegue quase sempre concluir, em poucos segundos, a criação de novos puzzles até dimensão 40.

Keywords: weight puzzle, clp, prolog, plog, feup

1 Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular Programação em Lógica, 3º ano do Mestrado Integrado em Engenharia Informática e Computação, tendo como objetivo aprofundar o conhecimento teórico e prático da matéria lecionada referente à resolução de problemas com restrições em Prolog, utilizando a biblioteca ‘clpfd’.

O tema escolhido pelo grupo foi o problema de decisão Weight Puzzle. Este consiste em distribuir pesos com valores consecutivos a partir de 1 em posições designadas numa estrutura em árvore, de modo a tudo ficar balanceado.

2 Weight Puzzle Solver

Este artigo descreve detalhadamente a abordagem seguida na resolução do problema, os resultados obtidos e conclusões, e está estruturado da seguinte forma:

Descrição do Problema

Abordagem

Variáveis de Decisão

Restrições

Estratégia de Pesquisa

Geração de Puzzles

Visualização da solução

Resultados Obtidos

Conclusões

Anexo

2 Descrição do problema

Weight é um problema de decisão que tem como objetivo distribuir um conjunto de pesos em posições designadas num diagrama representando uma balança, de modo a que haja equilíbrio total. Os valores dos pesos são consecutivos desde 1 ao número de posições destinadas a pesos do puzzle. De modo a que haja equilíbrio em cada ramo do diagrama, é necessário que o Torque total em relação à origem do ramo seja 0, ou seja, que a soma dos produtos entre os Pesos de cada elemento e respectivas distâncias à origem seja igual de cada lado. Cada puzzle tem apenas uma solução possível.

Para efeitos de resolução, o peso das cordas e varas é insignificante..

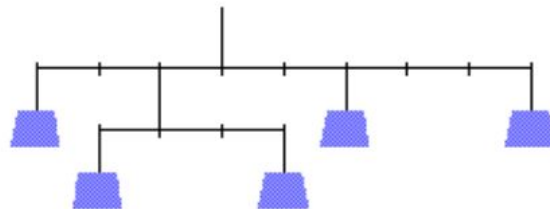


Fig. 1. Exemplo de um Weight Puzzle com 5 posições designadas para pesos, consequentemente os valores dos pesos para distribuir são: 1, 2, 3, 4 e 5.

3 Weight Puzzle Solver

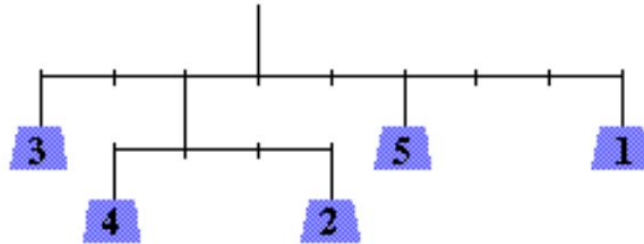


Fig. 2. Solução do puzzle representado na figura anterior. Podemos ver que $1 \times 4 = 2 \times 2$, e que $3 \times 3 + 6 \times 1 = 2 \times 5 + 5 \times 1$, logo o torque é nulo em ambos os ramos.

3 Abordagem

Para representar uma instância do puzzle em Prolog, criou-se uma estrutura em árvore, sendo cada **elemento** desta *weight(Distância, Peso)*, que corresponde a uma folha, ou *branch(Distância, SubÁrvore)*, que corresponde a um ramo. Os campos 'Distância' correspondem à posição X do **elemento** em relação ao nível superior.

Nas secções abaixo, o tamanho do problema (número de elementos *weight* que a árvore contém) será referido como **Size**. Considere-se também que o **Torque** de um ramo corresponde à soma dos produtos entre o **Peso** e a **Distância** de cada filho, sendo o **Peso** de um *branch* a soma do **Peso** dos seus filhos.

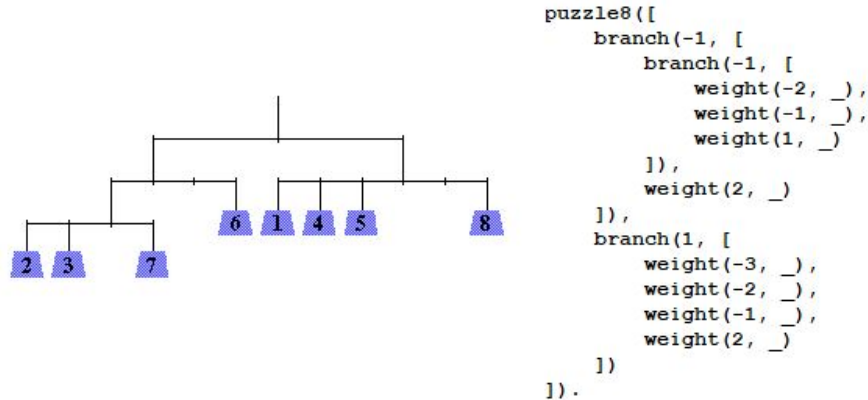


Fig. 3. Representação gráfica da solução de um puzzle de **Size** 8, assim como a representação em Prolog do mesmo por resolver

3.1 Variáveis de Decisão

A solução corresponde a uma lista com os **Pesos** de cada elemento *weight* da árvore, pela ordem que aparecem nesta. Por exemplo, no exemplo acima, a solução seria:

[2, 3, 7, 6, 1, 4, 5, 8]

O domínio de cada variável será, então, $[1, \text{Size}]$.

3.2 Restrições

As restrições rígidas que o programa teve de impor são:

1. Não existem dois pesos com o mesmo número
2. Qualquer ramo, incluindo a raiz, possui **Torque** nulo

Para a restrição 1, foi utilizado o predicado *all_distinct/1*, que garante que os elementos da lista fornecida são todos diferentes entre si.

Impor a restrição 2 já exigiu a criação de um predicado, *validPuzzle/1*. Para cada ramo da árvore, obtém os **Pesos** dos filhos e consequentemente o produto deste com a **Distância** (que será negativa para filhos à esquerda da raiz). Somando todos esses produtos obtém-se o **Torque**, restringindo-o a 0. Efetuando uma pesquisa em profundidade (DFS) aplica-se esta restrição aos descendentes da raiz, calculando os seus **Pesos** e **Torques**, simultaneamente, de baixo para cima, colocando para todos os ramos a restrição **Torque = 0**.

3.3 Estratégia de Pesquisa

Para a pesquisa, foi utilizada a heurística ffc (menor domínio) para escolha de variável, pois é a que aparenta levar a uma maior eficiência do programa, como descrito na secção 5. Esta heurística escolhe a variável com menor domínio, ou a que tem mais restrições suspensas, caso ocorra empate.

Relativamente à escolha e ordenação do valor, as heurísticas step/up foram selecionadas (opções por omissão), já que as outras hipóteses não aparentam um impacto significativo na eficiência do programa.

3.4 Geração aleatória de puzzles

A criação dinâmica de puzzles consiste em duas etapas. Primeiro, constrói-se uma árvore de forma aleatória, sem **Distâncias** nem **Pesos** preenchidos. De seguida, utilizando uma abordagem semelhante à descrita na secção anterior, calculam-se campos em falta de forma a que o puzzle seja válido.

Para a criação da árvore de forma aleatória, utiliza-se o seguinte método:

- Adicionam-se entre $\text{Size} / 4$ e $\text{Size} / 3$ ramos a um puzzle vazio, sendo mais provável estes serem colocados próximo do topo do que no fundo da árvore.
- Preenchem-se todos os ramos com elementos *weight* vazios, de forma a que cada **elemento** tenha pelo menos dois descendentes diretos.
- Os elementos *weight* que restam são colocados aleatoriamente, da mesma forma que os ramos foram colocados no passo 1.

Para a segunda etapa da geração do puzzle, utilizam-se as mesmas restrições e domínios do que os descritos em 3.2. Para as **Distâncias**, que neste caso são também

variáveis, estas têm domínio $[-\text{Size}/2, \text{Size}/2] \setminus \{0\}$, tendo os limites do intervalo valor absoluto mínimo de 3 e máximo de 5. Também foi imposta uma restrição que impede que não existam dois elementos com a mesma **Distância** no mesmo ramo.

Como heurística de escolha de variável, utilizou-se leftmost, a opção por omissão, pois leva a que os **Pesos** sejam as primeiras variáveis a serem escolhidas, calculando-se as distâncias baseado na estrutura que se obtém, sendo o comportamento igualmente aleatório, mas muito mais rápido, pois a probabilidade de ocorrer retrocesso **Distância** é muito mais pequena. A heurística de escolha de valor selecionada foi value(selRandom), sendo selRandom um predicado que escolhe um valor aleatório do domínio, o que não tem grande impacto na eficiência, mas aumenta a aleatoriedade do puzzle criado.

4 Visualização da solução

De modo a visualizar os puzzles e as suas soluções foram criados 2 conjuntos de predicados. O primeiro conjunto trata de criar e preencher uma matriz *printMatrix*, que é utilizada como estrutura para inserir os ramos da árvore, tendo em atenção interseções e colisões entre eles. O segundo conjunto imprime, linha a linha, essa matriz, tendo cada elemento com um bloco de impressão correspondente.

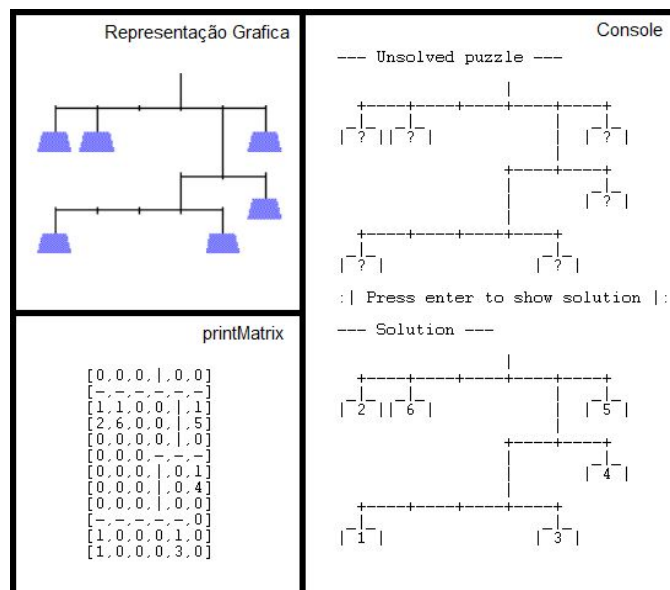


Fig. 4. Representação gráfica, printMatrix e representação na consola do puzzle 6

Depois de ser criada uma *printMatrix* vazia, tendo em conta as dimensões do puzzle, é chamado o predicado *insertbranchinmatrix/6*. Este recebe o ramo a inserir na matriz, a sua posição de origem e final. Após ser obtida a coordenada mínima e o

seu tamanho (*getbranchinfo/3*) é usado *checkifits/4* de modo a verificar se há colisões entre o ramo a ser inserido e os elementos já inseridos na matriz. Caso haja colisões, a posição final é aumentada até passar com sucesso o teste de colisões.

De seguida, são inseridos na matriz a corda vertical, horizontal e os elementos do ramo. Ao inserir a corda vertical utilizando *drawverticalstring/5*, são tomadas em atenção colisões verticais, que são resolvidas utilizando “portais”. Ao desenhar os elementos do ramo, caso o elemento seja um sub-ramo, é chamado novamente o predicado *insertbranchinmatrix/6*.

Para imprimir a *printMatrix*, é utilizado o predicado *printPuzzleMatrix/1*, que percorre todas as linhas desta e chama o predicado *printPuzzleRow/1*, que, por sua vez, imprime cada elemento da matriz em blocos de impressão para cada elemento.

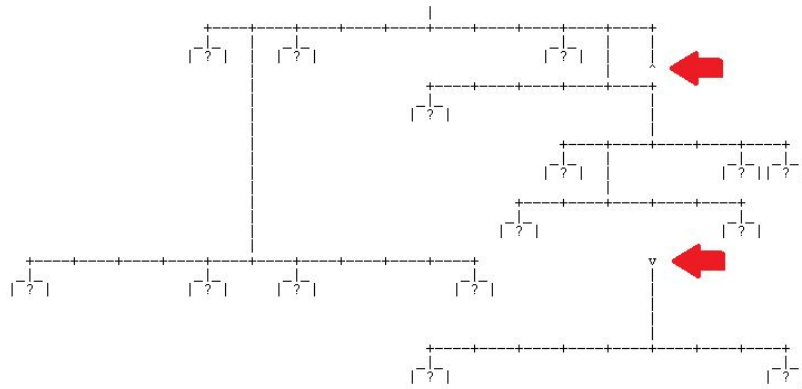


Fig. 5. Representação na consola de um puzzle com 15 pesos, estando presente um “portal” para evitar colisão da corda vertical com outros elementos do puzzle.

5 Resultados Obtidos

Com o intuito de determinar as melhores estratégias de pesquisa, foi medido o tempo de execução do programa para algumas heurísticas de escolha de variável e valor. As medições completas encontram-se no Anexo III.

5.1 Eficiência do gerador de puzzles

Para melhorar a eficiência do programa, foi analisada a rapidez com que este corre com diferentes heurísticas de escolha de valor: aleatória (random) e step (default).

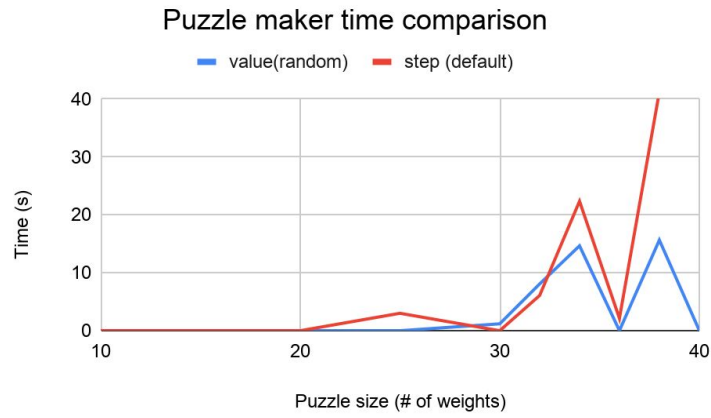


Fig. 6. Variação do tempo de execução do gerador de puzzles com o aumento do tamanho, com diferentes heurísticas

Pelos resultados obtidos, conclui-se que não há grande alteração no tempo de execução, a não ser com puzzles de tamanho superior a 30, para os quais o crescimento é muito mais acentuado. Por essa razão, e como a aleatoriedade aumenta o número de puzzles que é possível gerar, foi escolhida a heurística aleatória.

5.2 Eficiência do solver

De seguida, foi analisado o tempo que o programa demora a calcular uma solução, com diferentes heurísticas de escolha de variável, valor e ordenação: menor domínio (ffc)/mais à esquerda (leftmost - default), step (default)/bisect, up (default)/down.

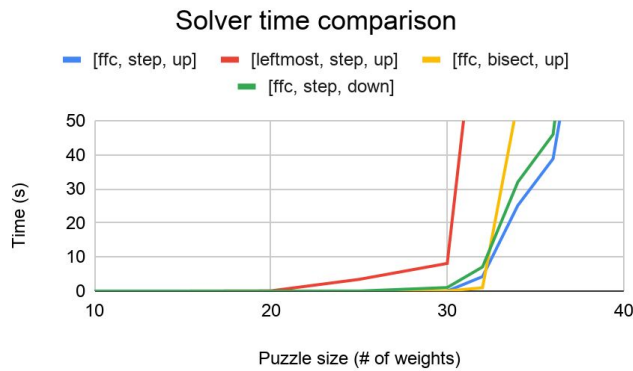


Fig. 6. Variação do tempo de execução do gerador de puzzles com o aumento do tamanho, com diferentes heurísticas

Observando o gráfico, determina-se que a heurística *ffc* tem um enorme impacto na eficiência de resolução do problema, diminuindo significativamente o tempo de execução. As outras heurísticas têm um impacto mais negligível, logo opta-se pelas opções [*ffc*, *step*, *up*].

Numa perspectiva mais geral, conclui-se que é possível criar e resolver, em tempo útil, problemas de tamanho inferior a 30.

6 Conclusões

Terminando o desenvolvimento do projeto, asseguramo-nos que Prolog é uma linguagem de programação muito poderosa, sendo que o uso de restrições torna o desenvolvimento de programas de resolução de problemas de decisão e otimização mais simples, com uma resolução bastante eficiente.

Ao longo do desenvolvimento deste projeto, foram encontradas algumas dificuldades, particularmente a geração de novos puzzles, assim como a apresentação de resultados, que foram mais complexos e demoradas do que inicialmente previsto. No entanto, foi possível criar um programa que gera e resolve, em tempo útil, problemas de tamanho inferior a 30, e puzzles até tamanho 40 em tempo razoável.

De forma geral, achamos que realizamos com sucesso todos os objetivos do projeto e o seu desenvolvimento contribuirá para uma melhor compreensão do funcionamento do uso de restrições em Prolog.

Bibliografia

1. Weight Puzzles, <https://www2.stetson.edu/~cfriedma/puzzle/weight/>, acedido em 2020/01/03
2. SICStus Prolog, <https://sicstus.sics.se/>, acedido em 2020/01/03.
3. SWI-Prolog, <https://www.swi-prolog.org/>, acedido em 2020/01/03.

Anexo

I. Código Fonte

src.pl

```
:- use_module(library(clpfd)).
:- use_module(library(samsort)).
:- use_module(library(random)).
:- use_module(library(lists)).
:-consult('display.pl').
:-consult('statistics.pl').
:-consult('puzzles.pl').

/* Menu */

menu:-
    repeat,
        write('+-----+'), nl,
        write('| Weight solver |'), nl,
        write('+-----+'), nl,
        write('1 - Solve Puzzle 1 (Size: 5)'), nl,
        write('2 - Solve Puzzle 2 (Size: 6)'), nl,
        write('3 - Solve Puzzle 3 (Size: 8)'), nl,
        write('4 - Solve Puzzle 4 (Size: 17)'), nl,
        write('5 - Solve Puzzle 5 (Size: 20)'), nl,
        write('-X - Solve Random Puzzle (Size: X, range:
1-20)'), nl,
        write('0 - Quit'), nl,
        read_option(Option),
        choose_option(Option),
        Option = 0, !.

choose_option(0).
choose_option(Option):-
    Option > 0,
    get_option_puzzle(Option, Puzzle),
    solvePuzzle(Puzzle),
    hidePuzzleSolution(Puzzle, HiddenPuzzle),
```

```

    printPuzzleAndSolution(HiddenPuzzle, Puzzle).

choose_option(Option):-
    Option < -1,
    PuzzleSize is abs(Option),
    makePuzzle(PuzzleSize, SolvedPuzzle),
    hidePuzzleSolution(SolvedPuzzle, HiddenPuzzle),
    printPuzzleAndSolution(HiddenPuzzle, SolvedPuzzle).

% Translates each option into respective puzzle
get_option_puzzle(0, _).
get_option_puzzle(1, Puzzle):- puzzleDefault(Puzzle).
get_option_puzzle(2, Puzzle):- puzzle6(Puzzle).
get_option_puzzle(3, Puzzle):- puzzle8(Puzzle).
get_option_puzzle(4, Puzzle):- puzzle17(Puzzle).
get_option_puzzle(5, Puzzle):- puzzle20(Puzzle).

% Prints unsolved, then solved version
printPuzzleAndSolution(Puzzle, Solution):-
    nl, print('--- Unsolved puzzle ---'), nl, nl,
    printPuzzle(Puzzle),
    print(':| Press enter to show solution '), get_code(X),
    get_code(X),
    nl, print('--- Solution ---'), nl, nl,
    printPuzzle(Solution).

% Read an option from standard input
read_option(Input):-
    repeat,
        write('Option: '),
        catch(read(Input), _Error, bad_input_format),
        validate_choice(Input),
    !.

% Validate the menu option input
validate_choice(Input):- integer(Input), Input >= -40, Input

```

```

=< 5, Input \= -1.
validate_choice(_):- bad_input_format.

% Printing error message for read_option
bad_input_format:- write('Invalid option.'), nl, fail.

/* Solver */

/*
    validPuzzle(+Puzzle)
    validPuzzle(+Puzzle, -Torque, -TotalWeight)

    Recursively enforces constraints on Puzzle, according to
    Weight rules. Unifies Torque and Weight with respective
    values.
    Can be called with only the first argument if Torque must be
    0 and Weight can be ignored.
*/
validPuzzle([], 0, 0).
validPuzzle([weight(Distance, Weight) | Puzzle], Torque,
Total):-
    NewTorque #= Torque + Distance * Weight,
    validPuzzle(Puzzle, NewTorque, Subtotal),
    Total #= Subtotal + Weight.

validPuzzle([branch(Distance, Weights) | Puzzle], Torque,
Total):-
    validPuzzle(Weights, 0, BranchWeight),
    NewTorque #= Torque + Distance * BranchWeight,
    validPuzzle(Puzzle, NewTorque, Subtotal),
    Total #= Subtotal + BranchWeight.

validPuzzle(Puzzle):- validPuzzle(Puzzle, 0, _).

/* getPuzzleVars(+Puzzle, -Weights, -Distances)

    Unifies Weights with a list containing all puzzle weights.

```

```

Unifies Distances with a list containing all puzzle
distances.
*/
getPuzzleVars([], [], []).
getPuzzleVars([weight(Distance, Weight) | Puzzle], [Weight |
Weights], [Distance | Distances]):-
    getPuzzleVars(Puzzle, Weights, Distances).
getPuzzleVars([branch(Distance, SubBranch) | Puzzle],
Weights, [Distance | Distances]):-
    getPuzzleVars(SubBranch, BranchWeights,
BranchDistances),
    append(BranchWeights, SubWeights, Weights),
    append(BranchDistances, SubDistances, Distances),
    getPuzzleVars(Puzzle, SubWeights, SubDistances).

/* solvePuzzle(+Puzzle, -Solution)
Solves a Weight puzzle. Solution will be a list containing
the puzzle weight values (DFS order)
*/
solvePuzzle(Puzzle):-
    getPuzzleVars(Puzzle, Solution, _Distances),
    length(Solution, PuzzleSize),
    domain(Solution, 1, PuzzleSize),
    all_distinct(Solution),
    validPuzzle(Puzzle),
    reset_timer,
    labeling([ffc, time_out(5000, Flag)], Solution),
    checkTimeOutFlag(Flag),
    print_time,
    fd_statistics.

checkTimeOutFlag(success).
checkTimeOutFlag(time_out):- write('Puzzle solving timed
out.'), nl, fail.

% testtest
solver:-
    puzzle8(Puzzle8),

```

```

    puzzle17(Puzzle17),
    puzzle20(Puzzle20),
    solvePuzzle(Puzzle8),
    solvePuzzle(Puzzle17),
    solvePuzzle(Puzzle20),
    write('Puzzle 8 solution: '), write(Puzzle8), nl,
    write('Puzzle 17 solution: '), write(Puzzle17), nl,
    write('Puzzle 20 solution: '), write(Puzzle20), nl.

/* Maker */

/* addNBranches(+N, +Puzzle, -NewPuzzle)
Adds N branches to Puzzle, unifying result with NewPuzzle.
*/
addNBranches(0, Puzzle, Puzzle).
addNBranches(N, Puzzle, NewPuzzle):-
    N > 0,
    N1 is N - 1,
    addBranch(Puzzle, Puzzle2),
    addNBranches(N1, Puzzle2, NewPuzzle).

/*
    addBranch(+Puzzle, -NewPuzzle)
    addBranch(+R, +Puzzle, -NewPuzzle)

Adds a branch to Puzzle, unifying with NewPuzzle.
If called with 2 arguments, has a 50% chance of adding a
branch to the root, 50% of adding to a random subbranch (if
no subbranches, adds to root)
If called with 3 arguments, R must be either 0 -> add to
root, or 1 -> add to random subbranch
*/
addBranch(Puzzle, NewPuzzle):-
    random(0, 2, R),
    addBranch(R, Puzzle, NewPuzzle).

addBranch(0, Puzzle, [branch(_Dist, []) | Puzzle]). % 0 ->

```

```

puts branch here
addBranch(1, [], [branch(_Dist, [])]). % 1 but no branches
-> puts branch here
addBranch(1, Puzzle, [branch(_Dist, NewBranch) |
OtherBranches]):- % 1 -> picks random branch, puts branch
there
    random_select(branch(_Dist, SubBranch), Puzzle,
OtherBranches),
    addBranch(SubBranch, NewBranch).

/* fillBranch(+Branch, -FilledBranch, -NumWeights)

Fills Branch and all of its Subbranches, so that every
branch has at least 2 elements.
The added elements are weight(_, _). Unifies NumWeights with
the total number of elements added.

Branch must not contain any element other than branch(_, _)
*/
fillBranch([], [weight(_, _), weight(_, _)], 2).
fillBranch([branch(_Distance, SubBranch)], [weight(_, _),
branch(_Distance, FilledSubBranch)], NumWeights):-
    fillBranch(SubBranch, FilledSubBranch,
SubBranchNumWeights),
    NumWeights is SubBranchNumWeights + 1.
fillBranch(Branch, FilledBranch, NumWeights):-
    length(Branch, BranchLength), BranchLength > 1,
    fillSubBranches(Branch, FilledBranch, NumWeights).

/* fillSubBranches(+Branch, -FilledBranch, -NumWeights)

Same as fillBranch/3, but ignores number of elements of the
root branch. (Meant for filling a list of subbranches)
*/
fillSubBranches([], [], 0).
fillSubBranches([branch(_Distance1, SubBranch) |
OtherBranches], [branch(_Distance1, FilledSubBranch) |
OtherFilledBranches], NumWeights):-
    fillSubBranches(OtherBranches, OtherFilledBranches,

```

```

OtherNumWeights),
    fillBranch(SubBranch, FilledSubBranch,
SubBranchNumWeights),
    NumWeights is OtherNumWeights + SubBranchNumWeights.

/* addNWeights(+N, +Puzzle, -NewPuzzle)
Adds N weights to Puzzle, unifying result with NewPuzzle.
*/
addWeight(Puzzle, NewPuzzle):-
    random(0, 2, R),
    addWeight(R, Puzzle, NewPuzzle).

addNWeights(0, Puzzle, Puzzle).
addNWeights(N, Puzzle, NewPuzzle):-
    N > 0,
    N1 is N - 1,
    addWeight(Puzzle, Puzzle2),
    addNWeights(N1, Puzzle2, NewPuzzle).

/*
    addWeight(+Puzzle, -NewPuzzle)
    addWeight(+R, +Puzzle, -NewPuzzle)

Adds a weight to Puzzle, unifying with NewPuzzle.
If called with 2 arguments, has a 50% chance of adding a
weight to the root, 50% of adding to a random subbranch (if
no subbranches, adds to root)
If called with 3 arguments, R must be either 0 -> add to
root, or 1 -> add to random subbranch
*/
addWeight(0, Puzzle, [weight(_Dist, _Weight) | Puzzle]). % 0
-> puts weight here
addWeight(1, Puzzle, [weight(_Dist, _Weight) | Puzzle]):- %
1 but no branches -> puts weight here
    getWeightsAndBranches(Puzzle, _Weights, []).
addWeight(1, Puzzle, [branch(_Dist, NewBranch) | Rest]):- %
1 -> picks random branch, puts weight there
    getWeightsAndBranches(Puzzle, Weights, Branches),
    length(Branches, NumSubBranches), NumSubBranches > 0,

```

```

        random_select(branch(_D, SubBranch), Branches,
OtherBranches),
        addWeight(SubBranch, NewBranch),
        append(OtherBranches, Weights, Rest).

/* getWeightsAndBranches(+Puzzle, -Weights, -Branches)

Separates Weights from Branches at root level.
*/
getWeightsAndBranches([], [], []).
getWeightsAndBranches([branch(_D, _W) | Puzzle], Weights,
[branch(_D, _W) | Branches]):-
    getWeightsAndBranches(Puzzle, Weights, Branches).
getWeightsAndBranches([weight(_D, _W) | Puzzle], [weight(_D,
_W) | Weights], Branches):-
    getWeightsAndBranches(Puzzle, Weights, Branches).

/* makeEmptyPuzzle(+PuzzleSize, -Puzzle)

Randomly makes a puzzle of exactly PuzzleSize weights, by
combining the previous functions. Unifies result with
Puzzle.
The puzzle's distances and weights will not be unified.
Steps:
- Create all of the branches (amount: between Size/3 and
Size/3)
- Fill all the branches, so that each has at least two
children
- With the remaining weights to place, randomly add them to
the puzzle.
*/
makeEmptyPuzzle(Size, Puzzle):-
    NumBranches is div(Size, 3),
    addNBranches(NumBranches, [], PuzzleBranches),
    fillBranch(PuzzleBranches, FilledPuzzle, NumWeights),
    RemainingWeights is Size - NumWeights,
    addNWeights(RemainingWeights, FilledPuzzle, Puzzle).

/* noOverlappingDistances(+Puzzle)

```



```

Enforce restrictions, so that there are no Puzzle elements
in the same place.
(At the same level, two elements can't have the same
distance)
*/
noOverlappingDistances(Puzzle):-
    noOverlappingDistances(Puzzle, BranchDistances),
    all_distinct(BranchDistances).

noOverlappingDistances([], []).
noOverlappingDistances([weight(Distance, _Weight) | Puzzle],
[Distance | BranchDistances]):-
    noOverlappingDistances(Puzzle, BranchDistances).

noOverlappingDistances([branch(Distance, Weights) | Puzzle],
[Distance | BranchDistances]):-
    noOverlappingDistances(Weights, SubBranchDistances),
    all_distinct(SubBranchDistances),
    noOverlappingDistances(Puzzle, BranchDistances).

/* noZeros(+List)

Enforce restrictions, so that no element from List can be
zero.
*/
noZeros([]).
noZeros([Elem | List]):-
    Elem #\= 0,
    noZeros(List).

/* makePuzzle(+PuzzleSize, -Puzzle)

Makes and solves a random weight puzzle of size PuzzleSize.
Puzzle is unified with result.
*/
makePuzzle(PuzzleSize, Puzzle):-
    PuzzleSize > 1,
    reset_timer,

```

```

repeat,
    % Make random puzzle
    makeEmptyPuzzle(PuzzleSize, Puzzle),
    getPuzzleVars(Puzzle, Weights, Distances),
    append(Weights, Distances, Vars),
    % Establish domain
    DistanceMax is min(5, max(3, div(PuzzleSize, 2))),
    DistanceMin is 0 - DistanceMax,
    domain(Weights, 1, PuzzleSize),
    domain(Distances, DistanceMin, DistanceMax),
    % Enforce restrictions
    all_distinct(Weights),
    noZeros(Distances),
    noOverlappingDistances(Puzzle),
    validPuzzle(Puzzle),
    % Calculate distance and solution
    % labeling( [bisect], Vars).
    labeling( [value(selRandom), time_out(1000,
success)], Vars),
    !,
    print_time,
    fd_statistics.

selRandom(Var, _Rest, BB0, BB1):- % seleciona valor de forma
aleatória
    fd_set(Var, Set), fdset_to_list(Set, List),
    random_member(Value, List), % da library(random)
    ( first_bound(BB0, BB1), Var #= Value ;
    later_bound(BB0, BB1), Var #\= Value ).

/* hidePuzzleSolution(+Puzzle, -NewPuzzle)

Replaces Puzzle's weights with non-unified variables.
*/
hidePuzzleSolution([], []).
hidePuzzleSolution([weight(Distance, _) | Puzzle],
[weight(Distance, _) | NewPuzzle]):-
    hidePuzzleSolution(Puzzle, NewPuzzle).
hidePuzzleSolution([branch(Distance, SubBranch) | Puzzle],

```

```
[branch(Distance, NewSubBranch) | NewPuzzle]):-
    hidePuzzleSolution(SubBranch, NewSubBranch),
    hidePuzzleSolution(Puzzle, NewPuzzle).
```

statistics.pl

```
reset_timer :- statistics(walltime,_).
print_time :-
    statistics(walltime,[_,T]),
    TS is ((T//10)*10)/1000,
    nl, write('Time: '), write(TS), write('s'), nl, nl.
```

display.pl

```
/* printPuzzle(+Puzzle)
   Prints the puzzle
*/
printPuzzle(Puzzle):-
    getPuzzlePrintMatrix(Puzzle,Matrix),
    printPuzzleMatrix(Matrix), !.

/* getPuzzlePrintMatrix(+Puzzle,-Matrix)
   Gets the puzzle print matrix, the print matrix is used to
   make
   a structure to how to print the tree. Elements are put in
   the print matrix so then they can be printed line by line
*/
getPuzzlePrintMatrix(Puzzle,Matrix):-
    getpuzzleinfo(Puzzle,Min,Max,NumBranches),
    Height is 7 * NumBranches + 1,
    Width is Max - Min + 1 ,
    make_empty_matrix(Width,Height,EmptyMatrix),
    insertbranchinmatrix(EmptyMatrix,Puzzle,Min,Matrix).

/* getpuzzleinfo(+Puzzle,-Min,-Max,-NumBranches)
   Gets the puzzle info: Minimum and maximum coords and num
   of branches
*/
```

```

getpuzzleinfo([], _, Min, Max, NumBranches, Min, Max,
NumBranches).
getpuzzleinfo([weight(Distance, _) | Puzzle], RelPos, Min,
Max, NumBranches, FMin, FMax, FNumBranches):-
    Pos is Distance + RelPos,
    NewMin is min(Pos, Min),
    NewMax is max(Pos, Max),
    getpuzzleinfo(Puzzle, RelPos, NewMin, NewMax,
NumBranches, FMin, FMax, FNumBranches).

getpuzzleinfo([branch(Distance, Weights) | Puzzle], RelPos,
Min, Max, NumBranches, FMin, FMax, FNumBranches):-
    NewRelPos is RelPos + Distance,
    NewNumBranches is NumBranches + 1,
    getpuzzleinfo(Weights, NewRelPos, Min, Max,
NewNumBranches, NewMin, NewMax, NewerNumBranches),
    getpuzzleinfo(Puzzle, RelPos, NewMin, NewMax,
NewerNumBranches, FMin, FMax, FNumBranches).

getpuzzleinfo(Puzzle, Min, Max, NumBranches):-
    getpuzzleinfo(Puzzle, 0, 99, -99, 1, Min, Max, NumBranches).

/*
insertbranchinmatrix(+Matrix,+PosX,+Origin,+Dest,+Branch,-FMatrix)
    Inserts a branch into the matrix. Its the main function and
    is called
    for every branch.
*/
insertbranchinmatrix(Matrix, PosX, Origin, Dest, Branch, FMatrix):
-
    getbranchinfo(Branch, Min, Max),
    X is PosX + Min,
    Size is Max - Min + 1,
    Y1 is Dest + 1,
    checkiffits(Matrix, X, Y1, Size),
    drawsverticalstring(Matrix, Origin, Dest, PosX, NewMatrix1),
    drawhorizontal(NewMatrix1, X, Y1, Size, NewMatrix2),
    Y2 is Dest + 2,

```

```

        drawbranchelements(NewMatrix2,Branch,PosX,Y2,FMatrix).

insertbranchinmatrix(Matrix,PosX,Origin,Dest,Branch,FMatrix):
-
    NewDest is Dest + 3,

insertbranchinmatrix(Matrix,PosX,Origin,NewDest,Branch,FMatrix).

insertbranchinmatrix(EmptyMatrix,Puzzle,Min,FinalMatrix):-
    Zero is -Min,

insertbranchinmatrix(EmptyMatrix,Zero,0,0,Puzzle,FinalMatrix)
.

/* checkiffits(+Matrix,+X,+Y,+Size)
   Check if a branch with certain dimensions doesnt colide
   with
   previous inserted elements.
*/
checkiffits(Matrix,X,Y,Size):-
    X1 is X-1,Y1 is Y-1,
    XN is max(X1,0),YN is max(Y1,0),
    NSize is Size+2,
    checkiffits(Matrix,XN,YN,NSize,4).

checkiffits(_,_,_,_,0).
checkiffits(Matrix,X,Y,Size,Loops):-
    checkrow(Matrix,X,Y,Size),
    NLoops is Loops - 1,
    Y1 is Y+1,
    checkiffits(Matrix,X,Y1,Size,NLoops).

checkrow(_,_,_,-1).
checkrow(Matrix,X,Y,Size):-
    Size >= 0,
    get_elem_at(X-Y,Matrix,A),
    !,A = 0,
    NX is X+1,NSize is Size-1,

```

```

    checkrow(Matrix,NX,Y,NSize).

/*
drawsvverticalstring(+Matrix,+Current,+Dest,+PosX,-FinalMatrix
)
    Draws a vertical line in the matrix from Current to Dest.
Without coliding
    with previous inserted elements
*/
drawsvverticalstring(FinalMatrix,Current,Dest,_PosX,FinalMatrix
x):- Current > Dest.

drawsvverticalstring(Matrix,Current,Dest,PosX,FinalMatrix):-
    get_elem_at(PosX-Current,Matrix,A),
    A == '-',
    C1 is Current - 1,
    replace_elem_at(PosX-C1,['|',Matrix,NewMatrix),
    C2 is Current + 4,
    drawstringcolision(NewMatrix,C2,Dest,PosX,FinalMatrix).

drawsvverticalstring(Matrix,Current,Dest,PosX,FinalMatrix):-
    replace_elem_at(PosX-Current,['|',Matrix,NewMatrix),
    New is Current +1,
    drawsvverticalstring(NewMatrix,New,Dest,PosX,FinalMatrix).

drawstringcolision(Matrix,Current,Dest,PosX,FinalMatrix):-
    get_elem_at(PosX-Current,Matrix,A),
    A \= 0,
    C1 is Current + 4,
    drawstringcolision(Matrix,C1,Dest,PosX,FinalMatrix).

drawstringcolision(Matrix,Current,Dest,PosX,FinalMatrix):-
    replace_elem_at(PosX-Current,['|',Matrix,NewMatrix),
    C1 is Current+1,
    drawsvverticalstring(NewMatrix,C1,Dest,PosX,FinalMatrix).

/* drawhorizontal(+Matrix,+X,+Y,+Size,-FinalMatrix)
    Draws a horizontal line in the matrix from X-Y position

```

```

with size Size
*/
drawhorizontal(FinalMatrix,_,_,0,FinalMatrix).
drawhorizontal(Matrix,X,Y,Size,FinalMatrix):-
    NSize is Size-1,
    replace_elem_at(X-Y,'-',Matrix,NewMatrix),
    Xn is X + 1,
    drawhorizontal(NewMatrix,Xn,Y,NSize,FinalMatrix).

/* drawbranchelements(+Matrix,+Branch,+PosX,+Y,-FMatrix)
   Draws the elements from a branch. If the element is a
   branch
   it will call insertbranchinmatrix.
*/
drawbranchelements(Matrix,Branch,PosX,Y,FMatrix):-
    drawbranchelements1st(Matrix,Branch,PosX,Y,NewMatrix),
    Y1 is Y + 1,
    drawbranchelements2nd(NewMatrix,Branch,PosX,Y1,FMatrix).

drawbranchelements1st(FinalMatrix,[],_,_,FinalMatrix).
drawbranchelements1st(Matrix,[weight(Distance, _) |
Branch],PosX,Y,FMatrix):-
    X is PosX + Distance,
    replace_elem_at(X-Y,'l',Matrix,NewMatrix),
    drawbranchelements1st(NewMatrix,Branch,PosX,Y,FMatrix).
drawbranchelements1st(Matrix,[branch(Distance, _) |
Branch],PosX,Y,FMatrix):-
    X is PosX + Distance,
    replace_elem_at(X-Y,'|',Matrix,NewMatrix),
    drawbranchelements1st(NewMatrix,Branch,PosX,Y,FMatrix).

drawbranchelements2nd(FinalMatrix,[],_,_,FinalMatrix).
drawbranchelements2nd(Matrix,[weight(Distance, Weight) |
Branch],PosX,Y,FMatrix):-
    integer(Weight),
    X is PosX + Distance,
    replace_elem_at(X-Y,Weight,Matrix,NewMatrix),
    drawbranchelements2nd(NewMatrix,Branch,PosX,Y,FMatrix).

```

```

drawbranchelements2nd(Matrix,[weight(Distance, _Weight) |
Branch],PosX,Y,FMatrix):-
    X is PosX + Distance,
    replace_elem_at(X-Y,'?',Matrix,NewMatrix),
    drawbranchelements2nd(NewMatrix,Branch,PosX,Y,FMatrix).
drawbranchelements2nd(Matrix,[branch(Distance, NewBranch) |
Branch],PosX,Y,FMatrix):-
    X is PosX + Distance,
    insertbranchinmatrix(Matrix,X,Y,Y+1,NewBranch,NewMatrix),
    drawbranchelements2nd(NewMatrix,Branch,PosX,Y,FMatrix).

/* getbranchinfo(+Puzzle,-Min,-Max)
   Gets branch information : min and maximum x values
*/
getbranchinfo(Puzzle,Min,Max):-
    getbranchinfo(Puzzle,99,-99,Min,Max).

getbranchinfo([],Min, Max, Min,Max).
getbranchinfo([weight(Distance, _) | Puzzle],Min, Max, FMin,
FMax):-
    NewMin is min(Distance,Min),
    NewMax is max(Distance,Max),
    getbranchinfo(Puzzle, NewMin, NewMax, FMin, FMax).
getbranchinfo([branch(Distance, _) | Puzzle],Min, Max, FMin,
FMax):-
    NewMin is min(Distance,Min),
    NewMax is max(Distance,Max),
    getbranchinfo(Puzzle, NewMin, NewMax, FMin, FMax).

/* make_empty_matrix(+Width, +Height, -Matrix)
   Makes an empty matrix a certain Width and Height
*/
make_empty_matrix(_,0,[]).
make_empty_matrix(Width, Height, [Row|Matrix]):-
    Height > 0,
    make_row(Width, Row),
    Height1 is Height - 1,
    make_empty_matrix(Width, Height1, Matrix).

```



```

make_row(0, []).
make_row(Width, [0 | Row]):-
    Width > 0,
    Width1 is Width - 1,
    make_row(Width1, Row).

/* printPuzzleMatrix(+Matrix)
   Prints the print matrix
*/
printPuzzleMatrix([]):-nl.
printPuzzleMatrix([Row,Row2|_Matrix]):-
    checkiffinished(Row),
    checkiffinished(Row2),nl. %if there are 2 empty rows in a
row, the printing is finished
printPuzzleMatrix([Row|Matrix]):-
    printPuzzleRow(Row),
    printPuzzleMatrix(Matrix).

/* checkiffinished(+Row)
   Checks if a row is empty
*/
checkiffinished([]).
checkiffinished([0|Row]):-
    checkiffinished(Row).

/* printPuzzleRowLine(+Row)
   Prints a horizontal line
*/
printPuzzleRowLine(['-'|Row]):-
    write('----+'),
    printPuzzleRowLine(Row).
printPuzzleRowLine(Row):-
    write(' '),
    printPuzzleRow(Row).

/* printPuzzleRow(+Row)
   Prints elemets from the row
*/

```

```

printPuzzleRow([]):-nl.
printPuzzleRow([0|Row]):-
    write('      '),
    printPuzzleRow(Row).

printPuzzleRow(['-'|Row]):-
    write('  +'),
    printPuzzleRowLine(Row).
printPuzzleRow(['|'|Row]):-
    write(' | '),
    printPuzzleRow(Row).
printPuzzleRow(['['|Row]):-
    write(' ^ '),
    printPuzzleRow(Row).
printPuzzleRow([']|Row]):-
    write(' v '),
    printPuzzleRow(Row).
printPuzzleRow(['1'|Row]):-
    write(' _|_ '),
    printPuzzleRow(Row).
printPuzzleRow(['? '|Row]):-
    write(' | ? | '),
    printPuzzleRow(Row).
printPuzzleRow([Elem|Row]):-
    write(' | '),displayNum(Elem),write(' | '),
    printPuzzleRow(Row).

/** Utility functions **/

/* Replaces an element from a list
Arguments:
- List
- Index to replace
- New element
- Returned new list
*/
replace([_|T], 0, X, [X|T]).
replace([H | T], I, X, [H | R]):-

```

```

    I > 0,
    I1 is I - 1,
    replace(T, I1, X, R).

replace_elem_at(X-Y, Elem, Matrix, NewMatrix):-
    nth0(Y, Matrix, Row),
    replace(Row, X, Elem, NewRow),
    replace(Matrix, Y, NewRow, NewMatrix).

get_elem_at(X-Y, Matrix, Elem):-
    nth0(Y, Matrix, Row),
    nth0(X, Row, Elem).
get_elem_at(_-, _, 0).

print_matrix([]).
print_matrix([H|T]) :- write(H), nl, print_matrix(T).

displayNum(V):- %if the number has 2 digits doesnt print the
space so it can fit
    V > 9 , write(V).
displayNum(V):-
    write(V),write(' ').

```

puzzles.pl

```

puzzleDefault([
    weight(-3, _),
    weight(-1, _),
    branch(2, [
        weight(-2, _),
        weight(-1, _),
        weight(1, _)
    ])
]).

puzzle6([
    weight(-3, _),
    weight(-2, _),

```

```

    branch(1,[
        branch(-1,[
            weight(-3,_),
            weight(1,_),
        ]),
        weight(1,_),
    ]),
    weight(2,_),
]).

puzzle8([
    branch(-1, [
        branch(-1, [
            weight(-2, _),
            weight(-1, _),
            weight(1, _)
        ]),
        weight(2, _)
    ]),
    branch(1, [
        weight(-3, _),
        weight(-2, _),
        weight(-1, _),
        weight(2, _)
    ])
]).

puzzle17([
    weight(-3,_),
    weight(-2,_),
    branch(-1,[
        branch(-1,[
            weight(-1,_),
            weight(1,_),
            weight(2,_),
        ]),
        branch(1,[
            branch(-1,[
                weight(-2,_),

```

```

        branch(1,[
            weight(-1,_),
            weight(2,_)
        ])
    ],
    weight(2,_)
]),
weight(2,_)
]),
branch(1,[
    branch(-2,[
        weight(-1,_),
        weight(2,_)
    ]),
    weight(1,_),
    branch(2,[
        weight(-3,_),
        weight(1,_)
    ])
]),
weight(2,_,
weight(3,_)
]).

```

```

puzzle20([
    branch(-4, [
        weight(-3,_),
        weight(-2,_),
        weight(3,_)
    ]),
    branch(-2, [
        weight(-2,_),
        weight(-1,_),
        branch(1, [
            branch(-1, [
                weight(-1,_),
                weight(2,_)
                weight(3,_)
            ]),

```

```

        weight(1,_),
        weight(2,_)
    ])
  ],
  weight(1,_),
  branch(3, [
    branch(-1, [
      branch(-1, [
        weight(-1,_),
        branch(1, [
          weight(-2,_),
          weight(1,6)
        ]),
        weight(2,_),
        weight(3,_)
      ]),
      weight(1,_),
      weight(2,_)
    ]),
    weight(1,_),
    weight(2,_)
  ])
  ]).

```

II. Resultados de medições

Tabela 1: Medições do tempo de execução do gerador

Size	Making [leftmost, value(random)]		Making [leftmost, step]	
	Time(s)	Backtracks	Time(s)	Backtracks
5	0,01	27	0,01	7
10	0	74	0	328
15	0	331	0	24
20	0	126	3,01	431200
25	1,19	105396	0	251
30	8,07	717020	6,08	780816
32	14,66	1447790	22,36	2492098
34	0,01	288	2,11	231890
36	15,65	1274013	41,23	4796495
38	0,06	2322	110,39	11611138
40	9,49	6969879	Timeout	Timeout

Tabelas 2 e 3: Medições do tempo de execução do *solver*

Size	Solving [ffc, step, up]		Solving [leftmost, step, up]	
	Time(s)	Backtracks	Time(s)	Backtracks
5	0	8	0	29
10	0	7	0	102
15	0	400	0,07	5791
20	0	256134	3,44	271024
25	0	331	8,13	168133
30	4,25	934574	Timeout	Timeout
32	25,16	2248107	Timeout	Timeout
34	38,9	2368590	Timeout	Timeout
36	99	300311	Timeout	Timeout

32 Weight Puzzle Solver

38	Timeout	Timeout	Timeout	Timeout
40	Timeout	Timeout	Timeout	Timeout

Size	Solving [ffc, step, up]		Solving [leftmost, step, up]	
	Time(s)	Backtracks	Time(s)	Backtracks
5	0	29	0	9009
10	0	76	0	64
15	0	386	0	73170
20	0	144	0,01	192425
25	0,03	723080	1,04	136944
30	0,98	175349	7,09	249164
32	55,76	2312780	32,03	1259989
34	Timeout	Timeout	46,05	3514102
36	Timeout	Timeout	127,84	4271006
38	Timeout	Timeout	Timeout	Timeout
40	Timeout	Timeout	Timeout	Timeout