Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Puzzle 2D

Yin Yang

Relatório Final

Professores:

Rui Carlos Camacho de Sousa Ferreira da Silva
Henrique Daniel de Avelar Lopes Cardoso
Daniel Augusto Gama de Castro Silva

Grupo Yin_Yang_2:

Catarina Alexandra Teixeira Ramos, up201406219
Inês Isabel Correia Gomes, up201405778
Turma 5

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

23 de dezembro 2016

Resumo

Resolução do problema de decisão com geração e solução do jogo de tabuleiro puzzle 2D Yin Yang utilizando programação em lógica com restrições. O objetivo é obter a maior eficiência possível, sendo para isso testadas várias opções de labeling e sendo elaboradas as melhores restrições possíveis. No final o resultado é visualizado através das diversas complexidades temporais apresentadas.

1. Introdução

O seguinte projeto foi realizado no âmbito da cadeira de *Programação em Lógica* do 3º ano 1º semestre do *Mestrado Integrado em Engenharia Informática e Computação* da Faculdade de Engenharia da Faculdade do Porto.

Este trabalho tem como principal objetivo a apresentação da geração e solução de um puzzle 2D com diferentes tamanhos e número de peças. Este é um problema de decisão com programação lógica com restrições (PLR). O jogo é implementado no ambiente SICStus Prolog que inclui um módulo de resolução de restrições sobre domínios finitos, clp (FD) .

O relatório aqui apresentado está estruturado em sete partes, sendo a primeira a introdução e a segunda a descrição do problema onde é apresentado o jogo. A terceira e quarta partes referem a abordagem à geração e solução do problema. Aqui descritas as variáveis de decisão, restrições, função de avaliação e estratégia de pesquisa. No quinto ponto é explica a visualização e no sexto ponto são descritos resultados textuais e gráficos. Por último teremos as conclusões do trabalho.

2. Descrição do problema

O jogo de tabuleiro Yin Yang é um puzzle 2D que tem como objetivo a separação do tabuleiro em duas regiões de peças pretas e brancas colocando uma peça preta ou branca por cada célula vazia. Para tal, é necessário seguir algumas regras:

- todas as peças da mesma cor devem estar conectadas umas às outras verticalmente ou horizontalmente;
- não deve existir um grupo de células 2x2 que contenha todos os círculos da mesma cor.

No nosso caso, a regra da conectividade NÃO verifica se TODAS as peças estão conectadas, apenas verifica que não há peças isoladas. Este problema será abordado mais adiante.

3. Abordagem da geração

A abordagem para a geração de puzzles aleatórios válidos é um pouco diferente da abordagem da solução dos mesmos. No entanto, devemos ter na mesma consideração à sua eficiência.

O algoritmo usado encontra-se no ficheiro 'Generator.pl' sobre o predicado "**generator(+Nc,+Nr,+Np,-Bf)**". Este predicado recebe o número de linhas, colunas e peças a colocar no tabuleiro e devolve o tabuleiro aleatório.

Para gerar este tabuleiro é usado o seguinte raciocínio:

- criar um tabuleiro vazio com o número de linhas e colunas recebido com o predicado "**create_empty_board(+Nc,+Nr,-B)**";
- usar o predicado "**solve_yin_yang**" sobre o tabuleiro vazio de forma a criar um novo tabuleiro aleatório. Este predicado pertence à solução do problema e usa programação em lógica com restrições. A diferença deste *solve* para o *solve* da solução encontra-se

no seu *labeling.* Neste caso é usada a seleção customizada de valores aleatórios através dos predicados "**mySelValores(Var, _Rest, BB, BB1)**" e "**select_rand_value(Set, Value)**";

- retirar peças da solução apresentada:
    - calcular quantas peças serão retiradas;
    - usar o predicado "**remove_pieces(+Nrem,+Nc,+Nr,+B,-Bf)**" que remove o número de peças recebidas aleatoriamente.
        - Para cada peça calcula uma posição aleatória com o predicado "**random_coords(+B,+Nr,+Nc,-R,-C,-Elem)**" ;
        - verifica se essa posição tem uma peça, se não tiver procura novas coordenadas;
        - coloca a casa calculada como vazia com o predicado "**remove_piece(+B,+R,+C,-NewB)**"

Esta solução garante tabuleiros aleatórios e válidos e a sua eficiência depende apenas do algoritmo usado para a solução do tabuleiro.

4. Abordagem da solução

Para tratar da solução do jogo de tabuleiro é utilizado o predicado ***solve_yin_yang(Incial_board, Final_board, Options)*** que recebe o tabuleiro inicial e, conforme as opções de labelling introduzidas, devolve o tabuleiro final com a solução.

As células do tabuleiro inicial podem tomar os valores:

- 0 - peça desconhecida;
- 1 - Peça preta;
- 2 - Peça branca.

4.1. Variáveis de decisão

As variáveis de decisão usados para a solução são as peças do tabuleiro cujo valor é desconhecido. Como cada peça pode ser preta ou branca, o domínio será entre 1 (peça preta) e 2 (peça branca).

Podendo o tabuleiro a resolver já se encontrar parcialmente preenchido, utiliza-se o predicado ***load_vars(Board_matrix,T ,Board_with_vars,T,All_vars)*** que analisa o board_matrix, substituindo as peças a 0 (desconhecidas) por variáveis (_) e adicionando as mesma a all_vars. Posto isto, a matriz board_with_vars será a mesma que board_matrix mas com as variáveis que depois serão passadas no labelling pela lista all_vars com todas as variáveis.

4.2. Restrições

As restrições implementadas para a solução deste jogo tem a haver com as regras do mesmo, nomeadamente, as já descritas no ponto 2 sobre a descrição do problema.

4.2.1. Connected

Esta regra implica que o valor de todas as células do tabuleiro seja igual ao valor da célula imediatamente acima, abaixo, da direita ou da esquerda. É implementada através do predicado ***connected(+Board)*** e está subdividida em 3 partes principais:

- ***(1) connected_first_row(Row1, Row2) -*** usado para restringir a primeira linha em que só é necessário analisar a primeira linha e a segunda.
- ***(2) connected_other_rows([RowX, RowY, RowZ | Others]) -*** usado para restringir as linhas intermédias em que só é necessário analisar três linhas: a linha intermédia, a linha imediatamente antes e a linha depois.
- ***(3) connected_last_row(Rown-1, Rown) -*** usado para restringir a última linha que, tal como a primeira, apenas analisa a penúltima e última linha.

Cada célula pode ser interpretada de 3 maneiras diferentes:

- ***(1) connected_corner(Corner,A,B) -*** utilizado para interpretar células dos cantos. Nestes casos, o valor da célula de canto apenas pode ser equivalente às duas células adjacentes.
- ***(2) connected_limit(A,B,C,D) -*** utilizado para interpretar células nos limites do tabuleiro, nomeadamente, todas as células: da primeira coluna, última coluna, primeira linha e última linha. Nestes casos, o valor da célula limite apenas pode assumir os valores das três células adjacentes.
- ***(3) connected_center(Center,A,B,C,D) -*** utilizado para interpretar células centrais, ou seja, todas as células que são rodeadas por quatro células.

4.2.2. No_2x2

Esta regra implica que em submatrizes de 2x2 do tabuleiro existam pelo menos uma célula de valor diferente. Para implementar esta regra são usados os predicados:

- **(1) *no_2x2(Board)***
- **(2) *no_2x2_aux(Row1,Row2).***
- ***(3) check_values(Elem1,Elem2,Elem3,Elem4)***

O predicado (1) analisa todas as linhas do tabuleiro e invoca o predicado (2) que faz a análise de duas colunas consecutivas. O predicado (2) analisa todas as colunas das duas linhas e obtém as quatro células necessárias para a verificação da regra com o predicado (3).

4.2.3. Regions

Esta regra implica que existam duas regiões muito bem definidas, uma composta por peças pretas e outras por peças brancas. Após várias abordagens a este problema em particular não nos foi possível concluir a restrição de forma eficiente. No entanto, nos pontos abaixo, será descrita as estratégias postas em prática nas várias tentativas executadas.

4.2.3.1. Tentativa 1

Esta estratégia está implementada no ficheiro RegionsRule1.pl que está junta com o restante código. Esta tentativa obteve resultados até boards de dimensões 4x4 mas mostrou-se muito pouco eficiente. A estratégia incidiu-se em:

1. Transformar a nossa matriz que representa o tabuleiro numa lista com o predicado ***flat_board***;
2. Obter os índices de uma peça branca e peça preta;
3. Criar uma lista com o mesmo tamanho do FlatBoard com todos os elementos a 0 utilizando o predicado ***create_list***. Esta lista serve para verificar se um determinado elemento do tabuleiro já foi processado, caso sim, o valor na posição desta lista será 1.
4. Fazer uma pesquisa em largura com o predicado ***check_conectivity*** primeiro começando no índice de valor 1 obtido no ponto 2. e depois para o índice de valor 2.

a. Processar um determinado índice com o predicado ***process_index*** conforme se o seu valor no tabuleiro, a validez do índice e se já foi processado ou não.
    b. Marcar como processado na lista de processados (valor igual a 1).
    c. Calcular os índices adjacentes e a sua validez com ***calc_index***.
    d. Processar ou não os adjacentes individualmente com ***check_conectivity_aux*** tendo em conta a sua validez e o veu valor no tabuleiro
    e. Se for para processar volta a a).
    f.

A condição de paragem verifica-se quando não houverem mais índices por processar.

Predicados acima referidos:

- ***flat_board(Board,FlatBoard)***
- ***create_list(+NCells,+Value, [ ], -List)***
- ***check_conectivity(+FlatBoard, +Index, +Value, +NumberRows, +NumberColumns, +ProcessedList, -FinalProcessedList)***
- ***process_index(FlatBoard, Index, Value, CellR, CellC, NR, NC, ProcessedList, ProcessedListF, Process)***
- ***calc_index(Index, Row, Column, NumberRows, NumberColumns, Valid)***
- ***check_conectivity_aux(ValidIndex, Index, Board, Value, IndexValue, NumberRows, NumberColumns, ProcessedList,ProcessedListF)***

4.2.3.2. Tentativa 2

Esta estratégia está implementada no ficheiro RegionsRule2.pl que está junta com o restante código. Esta tentativa não mostrou resultados.

A estratégia a implementar nesta tentativa seria ter duas listas em que cada uma representaria uma região. A cada uma destas listas pertenceria os valores dos índices no tabuleiro cujo valor correspondesse à região.

Para este efeito foram seguidos os seguintes passos:

1. Transformar a nossa matriz que representa o tabuleiro numa lista com o predicado ***flat_board***.
2. Invocar o predicado ***check_region*** em que T1 e T2 começam a [ ] e depois são retornadas as listas Region1 e Region2. Este predicado tem o seguinte efeito:
    a. Pega num elemento do FlatBoard na posição Index que ainda não se encontre em nenhuma das listas das regiões temporárias.
    b. Adiciona o Index à lista de região correta conforme o seu valor no tabuleiro utilizando o predicado ***verify_connection***. Para isto, verifica se existe um elemento na lista temporária, correspondente ao Value a processar, que é adjacente a Index. A verificação da adjacência seria feita com o predicado ***is_adj.***
    c. Repete a partir do ponto a).

A condição de paragem verifica-se quando a soma dos tamanhos das duas listas de regiões é igual ao tamanho do FlatBoard.

Predicados referidos acima:

- ***flat_board(Board,FlatBoard)***
- ***check_region(Board, NumberRows, NumberColumns, T1, T2, Region1, Region2)***

- ***verify_connection(FlatBoard, Index, NumberRows, NumberColumns, T1, T2, Region1, Region2, Value)***
- ***is_adj(A, B, NumberRows, NumberColumns)***

4.3.3.3. Tentativa 3

Esta estratégia está implementada no ficheiro RegionsRule3.pl que está junta com o restante código. Esta tentativa obteve resultados até boards de tamanho 4x4 mas mostrou-se pouco eficiente.

É uma estratégia semelhante à tomada na tentativa 1 (RegionRule1.pl), mas em vez de fazer a pesquisa em largura dos índices com valores 1 e 2 separadamente, nesta tentativa, temos a possibilidade de a fazer alternadamente.

Para a implementação desta estratégia foram tomados os seguintes passos:

1. Transformar a nossa matriz que representa o tabuleiro numa lista com o predicado ***flat_board.***
2. Criar uma lista com o mesmo tamanho do FlatBoard com todos os elementos a 0 utilizando o predicado ***create_list***. Esta lista serve para verificar se um determinado elemento do tabuleiro já foi processado, caso sim, o valor na posição desta lista será 1.
3. Verificar a conectividade entre as duas regiões com ***check_conectivity*** onde começa por dar prioridade a um dos índices tendo em conta o Value passado. Por sua vez, este predicado:
    a. Verifica se o index1 e index2 podem ser processados com o predicado ***check_process***.
    b. Processa um dos índices tendo em conta o Value do índice que devia processar e se este pode ser processado. Para isto é utilizado o predicado ***process_index.***
    c. Calcula e processa as células adjacentes com o predicado ***process_adj*** e para cada adjacência invoca o predicado ***check_conectivity_aux.*** Este último, invoca o ***check_conectivity*** passando o índice adjacente e o outro índice que não tinha sido escolhido para ser processado no ponto b. Desta vez, dá prioridade ao índice por processar mais antigo.

A condição de paragem verifica-se quando no ponto b) nenhum dos índices a processar deviam ser processados.

Predicados referidos acima:

- ***flat_board(Board,FlatBoard)***;
- ***create_list(-NCells, -Value, T, +List***
- ***check_conectivity(FlatBoard, Value, Index1, Index2, NumberRows, NumberColumns, ProcessedList, FinalProcessedList)***
- ***check_process(FlatBoard, Value, Index, ProcessedList, ProcessedListF***
- ***process_index(FlatBoard, Process1, Process2, Value, Index1, Index2, NumberRows, NumberColumns, ProcessedList, ProcessedListF)***
- ***process_adj(FlatBoard, Value, Index, Next_index, NumberRows, NumberColumns, ProcessedList, ProcessedListF)***
- ***check_conectivity_aux(Value, Next_index, Valid, Index, Board, NumberRows, NumberColumns, ProcessedList, ProcessedListF)***

4.4. Estratégia de pesquisa

Como já foi referido no ponto 4.1, existe a lista all_vars com todas as variáveis correspondentes às células do tabuleiro a resolver. É esta lista que é passada no labeling de forma, depois, a procurar a solução.

No caso da tentativa 2 da regra das regiões também seriam adicionadas as variáveis das listas de regiões.

5. Visualização da Solução

A visualização da solução está dividida em três partes : menu de seleção, tabuleiro e estatísticas. Para uma melhor visualização do tabuleiro é **recomendado** que utilize a fonte Consolas.

O menu de seleção inicial encontra-se no ficheiro "Display.pl" e tem como objetivo a seleção por parte do utilizador dos parâmetros número de colunas, número de linhas e número de peças. Os valores escolhidos são testados para mínimos e máximos. O tabuleiro tem um mínimo de 3x3 e máximo de 15x15. O mínimo advém das regras do jogo e o máximo apenas evita longas esperas para a solução e geração do tabuleiro. Já o número de peças pode ser escolhido pelo utilizador ou aleatório. No caso de ser escolhido pelo utilizador tem o mínimo de uma peça e o máximo metade do número máximo de peças. Se o utilizador escolher o modo automático o número de peças é um terço do seu máximo.

A segunda parte da visualização da solução está relacionada com o display do tabuleiro. Este implica predicados mais complexos que podem ser visualizados no ficheiro "Display.pl". É também de notar que para a visualização do tabuleiro a fonte do SICStus Prolog deve ser Consolas. O predicado principal é "**display_board(+B)**" que apenas recebe um tabuleiro em forma de lista de listas e que será repartido em vários outros predicados. A lista de listas recebida contém números entre 0 e 2, sendo zero uma célula vazia, 1 uma célula preta e 2 uma célula branca. Os predicados são invocados pela seguinte sequência dentro do predicado "**display_board(+B)**" :

- display_first_line(+Nc, +C)
- display_mid_lines(+List, +Nc)
    - display_line_1(+R)
        - display_cell(+C)
    - display_limits(+Nc, +C)
- display_last_line(+Nc, +C)

Estes predicados usam ainda o predicado **display_symbol(+N)** para o display de caracteres especiais.

Por último, a visualização das estatísticas é muito simples, apenas usa o predicado de Prolog "**fd_statistics"** e um pequeno cálculo de tempo pelo predicado "**print_time**".

É possível observar nas figuras 1 e 2 do Anexo A, alguns exemplos da representação do tabuleiro, menu e estatísticas.

6. Resultados

Para demonstrar melhor os algoritmos utilizados foram analisados, para diferentes complexidades, os resultados temporais obtidos. Os gráficos seguintes podem ser divididos em duas partes: algoritmo para o gerador e algoritmo para a solução. Isto acontece devido aos diferentes labelings utilizados, como já foi referido nos pontos 3 e 4 do presente relatório, pois o predicado é o mesmo.

Independentemente do algoritmo, a primeira análise focou-se nas opções de labeling. Para o tipo de jogo, chegamos à conclusão que apenas necessitaríamos das opções:

- Ordenação de variáveis: *leftmost, first_fail, anti_first_fail, occurrence, ffc;*
- Forma de seleção de valores: *step, enum, bisect, median, middle, value(Enum)*;

Para a ordenação de variáveis, não faz sentido os valores *min* e *max*, dado que não se adequa ao nosso problema, visto que não é um problema de optimização. A opção *max_regret* também não toma efeito dado que o domínio é [1,2], pelo que as sua subtração será sempre 0 ou 1, não se tornando eficiente. A opção *variable(Sel)* não foi tida em conta neste projeto.

Tendo em conta o domínio (1,2), não seria necessário a ordenação de valores, nem minimização ou maximização de valores (soluções a encontrar), e o esquema de pesquisa seria irrelevante, bem como as restante opções.

6.1 Gerador

Tendo em conta que o objetivo do *labeling* para o gerador é encontrar diferentes soluções aleatórias, este *labeling* precisa obrigatoriamente da opção **value(Enum)** para a forma de seleção de valores. Assim, tendo em conta a descrição acima efetuada, a única opção que pode ser alterada é a ordenação de variáveis.

Deste modo, foram analisados os tempos de execução para diferentes labelings dentro da opção Ordenação de variáveis. Para o gráfico representado na figura 3 do Anexo A, foram usadas 10 amostras (por cada variável do eixo xx) para um tabuleiro 6x6 com 12 peças.

As opções *first_fail* e *anti_first_fail* tendo em conta o domínio [1,2], verificam tempos semelhantes. Já a opção *occurence* demonstra-se pouco efetiva para a nossa solução. As opções *leftmost*, *ffc* e *first_fail*/*anti_fisrt_fail* apresentam-se como as mais eficientes, com tempos relativamente semelhantes.

Tendo em conta estes resultados, comparados com as suas avaliações teóricas, escolhemos a opção **ffc** para a ordenação das variáveis.

Após a escolha da opção de ordenação de variáveis, podemos então verificar os resultados temporais do algoritmo de geração aleatória do puzzle. O gráfico da figura 4 do anexo A mostra os tempos para o número de peças automático (1 terço do número máximo de peças de cada tabuleiro) e diferentes dimensões do tabuleiro, para 10 amostras (por cada variável do eixo xx).

Como é possível observar, o tempo médio aumenta, possivelmente, quadraticamente. No entanto o desvio padrão aumenta, possivelmente, exponencialmente com o aumento das dimensões. Estes dados demonstram que o algoritmo funciona dentro do expectável até dimensões máximas 8x8. No entanto, para dimensões maiores o desvio padrão aumenta substancialmente devido ao elevado número de restrições criadas associadas ao *backtracking* e outros métodos de *labeling* que, por vezes, causam tempos fora do normal, e mesmo um *crash* do programa. Este tempos são também influenciados pelo número de peças, pois quanto mais peças precisa de tirar, mais *randoms* terá de realizar.

6.2 Solução

A geração do puzzle aleatório, como já foi referido, usa o mesmo predicado de *solve* da sua solução. No entanto, o seu *labeling* é diferente.

Para a ordenação de variáveis foi efetuada uma experiência de 10 amostras (para cada elemento do eixo xx) para tabuleiros de dimensões 6x6 e 12 peças (com as restantes opções *default*).

Como já foi concluído na ponto 6.1 do presente relatório, a opção de ordenação de variáveis *anti_first_fail,* neste caso, é bastante semelhante à opção *first_fail* pelo que não precisa de ser novamente testada. Como a opção *occurence* se mostrou tão ineficiente na geração, também foi ignorada para esta experiência. Como pode ser observado pelo gráfico da figura 5 do Anexo A, as restantes 3 opções mantêm-se bastante próximas, pelo que a opção mais viável será a **ffc,** que devido às suas vantagens teóricas, se destaca.

Para a análise da forma de seleção de valores, é usada a opção *ffc* para otimizar os resultados. As características da experiência são semelhantes à experiência anterior: 12 peças, dimensão 6x6 e 10 amostras (para cada elemento do eixo xx).

Após a análise do gráfico da figura 6 do Anexo A, podemos verificar que os tempos médios poucos diferem de solução para solução, pelo que concluímos que neste caso a forma de seleção de valores é irrelevante, pelo que usaremos apenas a opção **step** (*default*).

Verificado o *labeling*, é necessário analisar a complexidade temporal do algoritmo para as opções que o otimizam. Para tal é efetuada uma experiência com 10 amostras (por cada variável no eixo xx), para o número de peças automático para cada dimensão.

O gráfico da figura 7 do Anexo A, permite verificar que a média temporal inicialmente se mantém estável, pois o seu desvio padrão é relativamente baixo e os tempos aumentam gradualmente. No entanto a partir das dimensões 10x10, o desvio padrão aumenta substancialmente, o que significa que não há uma variação temporal estável. Para o valor máximo estabelecido (15x15), os valores temporais já não se encontram razoáveis. Para estas dimensões, já foram necessárias bastantes amostras visto que, muitas vezes, o programa *crashava*.

Por fim, iremos analisar o impacto do número de peças nos tempos médios de desempenho do algoritmo. Para tal, a nossa experiência contará com 10 amostras (por variável no eixo xx), todas com dimensões 10x10, e as opções de labeling que otimizam o algoritmo.

Analisando o gráfico da figura 8 do Anexo A, podemos concluir que o algoritmo é ótimo para um número de peças entre um quarto e metade do número máximo de células (neste caso 25-50). Também é possível verificar que é muito ineficiente para um número de peças muito baixo.

7. Conclusões e Trabalho Futuro

Este projeto é muito interessante na medida em que nos impulsiona a procurar novas soluções cada vez mais eficientes para o problema apresentado. No nosso caso, o jogo Yin Yang, trouxe desafios e dificuldades o que também permitiu a busca mais aprofundada de conhecimentos na área da programação em lógica com restrições.

No final, podemos verificar que os algoritmos elaborados são ótimos para a opção de labeling ffc. No caso do gerador essa otimização é alcançada pela opção value(mySelValores) que garante a aleatoriedade do puzzle. Após a análise minuciosa dos gráficos realizados a partir dos dados recolhidos, conseguimos concluir que os algoritmos são ótimos para dimensões entre 3x3 e 8x8, e que o número de peças ótimas é entre um quarto e metade do número máximo de células.

Os algoritmos elaborados têm a vantagem de possibilitar o utilizador de escolher o seu número de peças e número de linhas e colunas. Também possui a vantagem de poder escolher um número de linhas e colunas superior ao original (6x6), ainda com elevada eficiência. No entanto apresenta a limitação de não conseguir conectar as peças de modo a ter 2 regiões, e não conseguir construir puzzles com dimensões maiores que 15x15.

Uma forma de melhorar o trabalho seria conseguir implementar a regra das regiões com sucesso e, se fosse possível, implementar da maneira mais eficiente possível. Entre outros aspetos, poderia-se também melhorar alguns pequenos pormenores com o objetivo de melhorar a eficiência na procura da solução em geral.

Anexo A

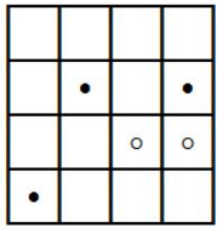
```
Number of columns and rows between 3 and 15 (inclusive)
Cols :4.

Rows :|: 4.

Number of pieces:
(1) automatic
(2) choosen by player
|: 1.
```

Figura 1 - Exemplo do menu de secção para o gerador

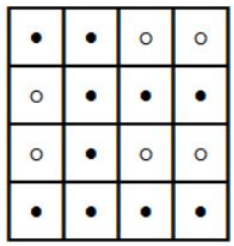
unresolved



```
Time: 0.08s

Resumptions: 259
Entailments: 90
Prunings: 166
Backtracks: 0
Constraints created: 90
```

resolved



```
Time: 0.07s

Resumptions: 310
Entailments: 156
Prunings: 205
Backtracks: 3
Constraints created: 85
```

Figura 2 - Exemplo da representação do tabuleiro e estatísticas antes e depois da resolução

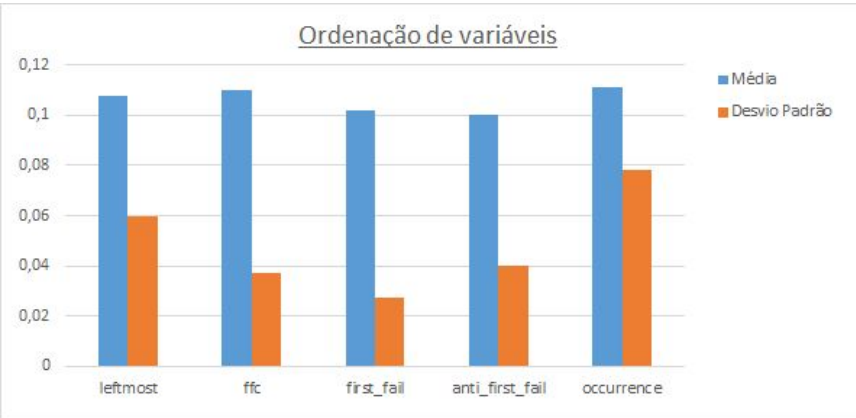


Figura 3 - ordenação de variáveis para o gerador

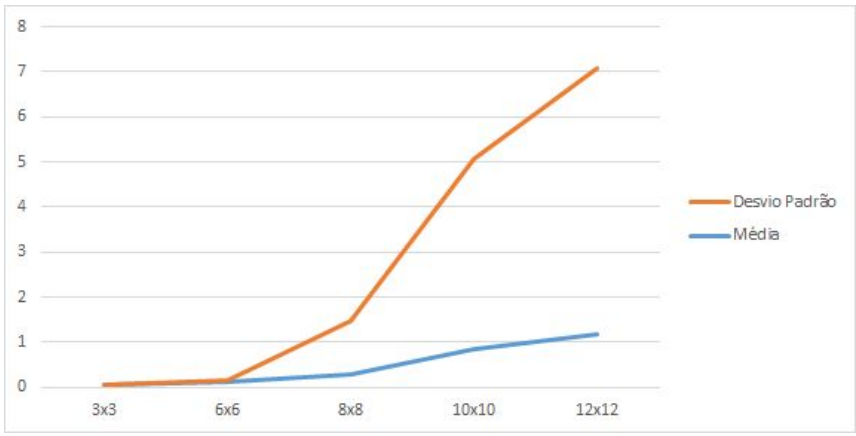


Figura 4 - tempos médios para diferentes dimensões

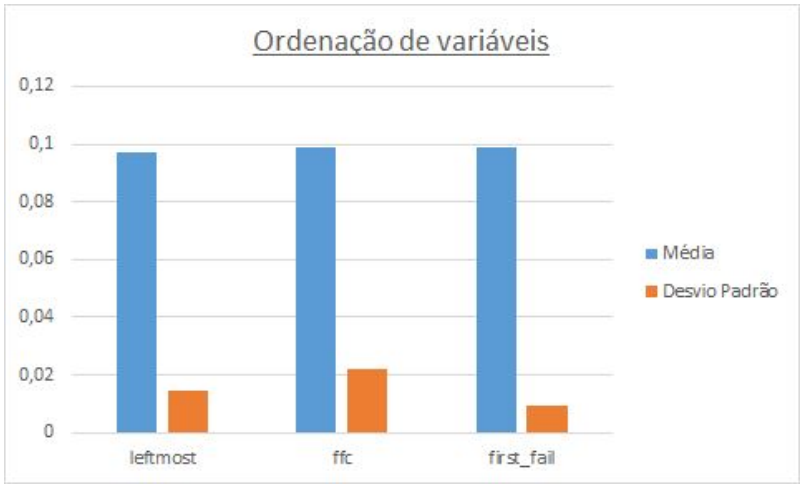


Figura 5 - Ordenação de variáveis para a solução

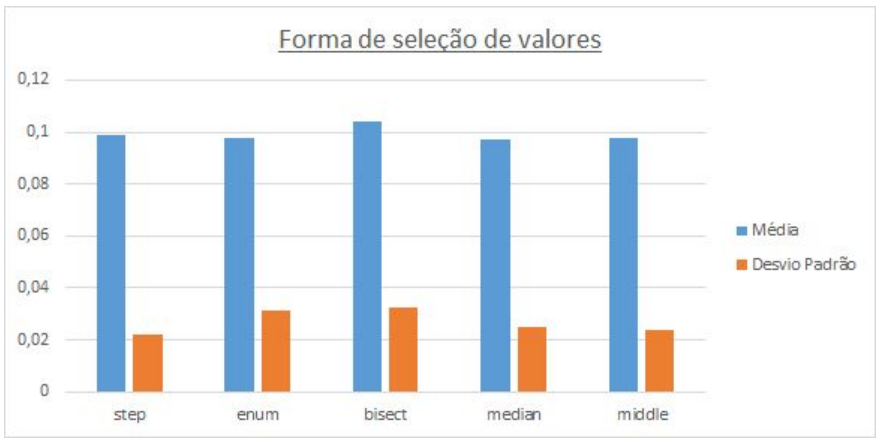


Figura 6 - forma de seleção de valores para a solução

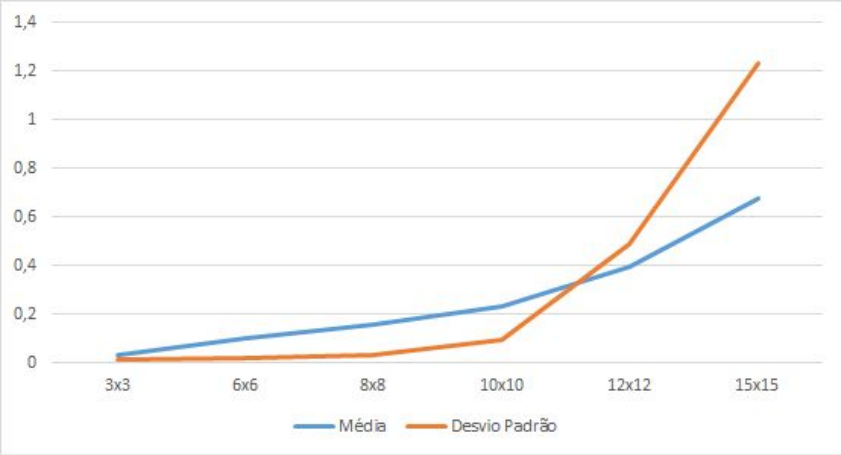


Figura 7 - Tempos médios para diferentes dimensões na solução

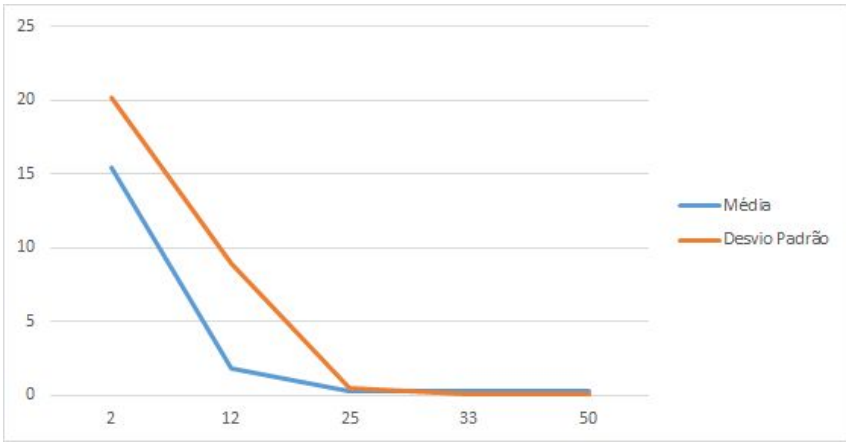


Figura 8 - Tempos médios para diferentes números de peças