Programação em Lógica com Restrições

Geração e Resolução de um puzzle “Tic Tac Logic” em Prolog

FEUP-PLOG, Turma 3MIEIC06, Grupo TicTacLogic\_3

André Sousa Lago ([up201303313@fe.up.pt](mailto:up201303313@fe.up.pt))

Gustavo Rocha da Silva (up201304143@fe.up.pt)

Universidade do Porto, 2015/2016

**Abstract.** Resolução do problema de decisão inerente à geração e solução um puzzle “Tic Tac Logic” utilizando programação em lógica com restrições em SICStus Prolog da forma mais eficiente possível, evitando *backtracking* e instanciação de variáveis antes do seu *labeling*. Foi elaborado um conjunto de predicados para o SICStus Prolog que permite a resolução do problema enunciado, bem como a visualização da sua complexidade temporal e do impacto das opções de *labeling*.

**Keywords:** Programação em lógica com restrições **·** Prolog **·** SICStus Prolog **·** Problemas de decisão.

1. Introdução

Este projeto foi proposto no âmbito da unidade curricular “Programação em Lógica” do 3º ano do Mestrado Integrado em Engenharia Informática e Computação da Faculdade de Engenharia da Universidade do Porto, com o objetivo de construir um programa em Programação em Lógica com Restrições para a resolução de problemas de decisão combinatória ou de otimização sugeridos.

Neste caso, o problema em análise trata-se de um problema de decisão combinatória inerente ao puzzle lógico “*Tic Tac Logic*”. Tratando-se de um puzzle de tabuleiro, o nosso programa consiste num conjunto de predicados de *SICStus Prolog* que permite a geração e solução de puzzles “*Tic Tac Logic*” com altura e largura variáveis e independentes, bem como a visualização dos estados iniciais, intermédios e finais dos tabuleiros e da variação da complexidade temporal da execução em função das dimensões do tabuleiro.

Assim, neste relatório, será feita uma descrição do problema em análise, ou seja, as regras do “*Tic Tac Logic*”. De seguida, será apresentada a abordagem aplicada à resolução do problema no que diz respeito a variáveis de decisão, restrições, função de avaliação e estratégia de pesquisa. Seguir-se-á a demonstração da visualização da solução bem como os resultados obtidos. Por fim, serão discutidas as conclusões atingidas com a realização do projeto, terminando com a bibliografia e os anexos existentes.

1. Descrição do Problema

O problema do “*Tic Tac Logic*” apresenta algumas semelhanças com o tradicional “jogo do galo” no que diz respeito ao tabuleiro e possibilidades de jogo. O problema consiste num puzzle que se constrói sobre um tabuleiro retangular com *N* células de largura e *M* células de altura. Cada célula deve ser preenchida com uma de duas possibilidades, ‘*X*’ ou ‘*O*’. O estado inicial do tabuleiro pode ser vazio ou com algumas células preenchidas com uma das duas possibilidades referidas. Assim, o tabuleiro deve ser preenchido de tal forma que se verifiquem as seguintes condições:

1. Numa mesma linha ou coluna não devem haver mais do que duas posições com ‘*X*’ ou ‘*O*’ consecutivas.
2. Para cada linha e coluna o número de posições com ‘*X*’ deve ser igual ao número de posições com ‘*O*’.
3. Cada linha e cada coluna devem ser únicas.
4. Abordagem para solução do puzzle

A abordagem ao problema em questão consistiu na determinação das variáveis de decisão, restrições e estratégia de pesquisa que permitem a solução eficiente do puzzle “*Tic Tac Logic*”.

* 1. Variáveis de Decisão

As variáveis de decisão correspondem às células do tabuleiro de jogo. Assim, dado um tabuleiro *N*\**M*, a lista de variáveis de decisão corresponde à ordenação linear de todas as células do tabuleiro, linha a linha. Para a representação em Prolog do tabuleiro utiliza-se uma lista de listas, sendo que cada uma das listas “internas” representa uma linha do tabuleiro, sendo por sua vez composta pelas células do tabuleiro. Assim, o jogo deve ser representado por uma lista composta por *M* listas de comprimento *N*. Por sua vez, cada célula é representada por um número de 0 a 2, que representa uma célula vazia (0), uma célula com ‘*X’* (1) ou uma célula com '*O’* (2). Uma vez que o tabuleiro se deve encontrar totalmente preenchido no final da resolução do puzzle, o domínio das variáveis presente nessa lista será [1, 2].

* 1. Restrições

Como foi descrito na secção 2 deste artigo, a resolução do puzzle é limitada por três restrições. A sua implementação em *SICStus Prolog* foi a seguinte:

### Restrição 1 – células iguais consecutivas

Para garantir o cumprimento desta restrição foi utilizado o predicado “*no\_more\_than\_two\_consecutives(+Board)*”. Este predicado sucede se em nenhuma das linhas do tabuleiro fornecido ocorrer o mesmo símbolo (*X* ou *O*) mais do que duas vezes seguidas. Contudo, isso não garante o cumprimento da restrição nas colunas do tabuleiro. Para isso, é efetuada a transposição do tabuleiro inicial com recurso ao predicado “*transpose(+OriginalMatrix, -TransposedMatrix)*”, que transpõe a matrix fornecida, fornecendo-se o seu resultado ao predicado “*no\_more\_than\_two\_consecutives*” para analisar o cumprimento da restrição.

### Restrição 2 – igual número de posições com *X* e *O*

Uma vez mais, para facilitar a validação desta restrição, pode ser utilizada a técnica de transposição referida no parágrafo anterior uma vez que dessa forma, a verificação linha a linha e coluna a coluna é igual.

Assim, utiliza-se o predicado “*same\_number()*” tanto no tabuleiro original como no tabuleiro transposto para garantir que em cada linha e coluna o número de posições com *X* e com *O* é igual.

### Restrição 3 – linhas e colunas únicas

Também na verificação desta restrição foi utilizada a técnica de transposição já referida pelos motivos mencionados na secção anterior.

Deste modo, é aplicado o predicado “*all\_different\_lists(+ListOfLists)*” que sucede apenas se todas as listas em “*ListOfLists*” forem distintas umas das outras. O predicado é aplicado com o tabuleiro original e o tabuleiro transposto para, assim, garantir a unicidade de cada linha (em relação às restantes linhas) e de cada coluna (em relação às restantes colunas).

* 1. Estratégia de Pesquisa

Para solucionar um tabuleiro foi desenvolvido o predicado “*solver(+Board, +Width, +Height, +LabelingParams)*” que permite alterar os parâmetros de “*labeling*”. Neste predicado, se a variável “*Board*” não estiver instanciada, o predicado coloca nela um tabuleiro vazio. Se, porém, se pretender que o predicado resolva um tabuleiro parcialmente preenchido, as posições vazias deste não devem estar instanciadas. Para a solução de um tabuleiro já existente não são necessários parâmetros de “*labeling*” pois as opções por defeito permitem uma resolução eficiente do problema.

Contudo, quando se pretende gerar um tabuleiro de jogo a partir de um tabuleiro vazio, é necessário garantir que o “*labeling*” faz uma escolha aleatória das variáveis para evitar que o tabuleiro gerado seja sempre igual de umas vezes para as outras. Neste caso, o “*labeling*” deve ter ativada a opção “*variable(sel)*” que torna o predicado “*sel(+Variables, -Selected, -Rest)*” responsável por escolher a variável seguinte a instanciar. Neste caso, o predicado “*sel*” faz uma escolha aleatória das variáveis disponíveis.

1. Abordagem para geração do puzzle

A geração eficiente de um puzzle de jogo válido traduz-se em problemas diferentes daqueles que são inerentes à solução de um puzzle. Contudo, um algoritmo de solução eficiente é essencial para uma geração não só eficiente como válida. Por outras palavras, sem um algoritmo de solução de um puzzle seria impossível de garantir que o tabuleiro gerado fosse válido sem possuir tabuleiros pré-definidos.

Foram implementados dois algoritmos de geração distintos uma vez que cada um apresenta as suas vantagens e desvantagens:

* 1. Geração com garantia de uma e só uma solução possível

O primeiro algoritmo, implementado pelo predicado “*generate\_board\_slow(-Board, +Width, +Height)*”, usa uma técnica de geração menos eficiente mas que garante que existe apenas uma solução possível tabuleiro gerado. Esta técnica consiste nos seguintes passos:

1. Criação de um tabuleiro válido com recurso ao predicado “*solver*”, utilizando como opção de “*labeling*” o predicado “*variable(sel)*” de forma a tornar aleatório o tabuleiro gerado.
2. Construção de uma lista com todas as posições preenchidas to tabuleiro (“*board\_nonempty\_coords(+Board, +Width, +Height, -NonEmpty)*”).
3. Seleção aleatória de uma posição (“*-Coords, +NonEmpty, -Rest*”).
4. Se a remoção da posição escolhida mantiver a restrição de apenas existir uma solução possível para o tabuleiro resultante, a posição é removida e retorna-se ao passo 2 (“*board\_remove\_pieces\_aux*”);
5. Em caso contrário, faz-se uma nova escolha aleatória no resto da lista obtida em 2
6. No caso de não haver nenhuma posição a remover que garanta a restrição referida, termina-se a execução do predicado, sendo retornado o tabuleiro gerado.
   1. Geração com número definido de peças preenchidas

Este algoritmo, implementado pelo predicado “*generate\_board\_fast(-Board, +Width, +Height)*”, revelou ser bastante mais rápido do que o anterior, mas tem uma desvantagem no que diz respeito ao número de soluções possíveis para o tabuleiro gerado. Enquanto o outro algoritmo garante que apenas há uma solução possível, este não garante esta restrição, sendo precisamente essa a causa do aumento da velocidade de execução. É, assim, impossível dizer se este algoritmo é melhor ou pior do que o anterior pois isso dependeria do contexto de aplicação.

A técnica obstante deste algoritmo é a seguinte:

1. Geração de um tabuleiro válido com recurso ao predicado “*solver*”, utilizando como opção de “*labeling*” o predicado “*variable(sel)*” de forma a tornar aleatório o tabuleiro gerado.
2. Sendo “*Size*” o número total de posições do tabuleiro gerado, remoção sucessiva de posições preenchidas aleatórias até que o número total de posições preenchidas no tabuleiro seja igual a *Size/7.*
3. Visualização da Solução

A visualização do tabuleiro em modo de texto é feita com recurso aos predicados implementados no ficheiro “*visualization.pl*”.

De uma forma geral, o predicado “*print\_board(+Board)*” imprime o tabuleiro de forma a ser percetível a organização das células. Para isso, desenha “linhas” que limitam as células do tabuleiro e coloca em cada célula um caractere representativo do valor dessa célula.

Para fazer a conversão dos valores das células para texto, o predicado “*piece\_to\_ascii(+Piece, -Char)*” determina o caractere correspondente a “*Piece*”. Se “*Piece*” for uma variável ou tiver o valor zero, o caractere correspondente é o espaço (célula vazia). Por sua vez, “1” corresponde ao caractere ‘*X*’ e “2” ao caractere “*O*”.

O predicado “*print\_board*” faz uso dos predicados “*print\_separating\_line(+Size)*” e “*print\_line(+Line)*” que fazem, respetivamente, a impressão de uma linha constituída por caracteres ‘-‘ que separa as linhas do tabuleiro e a impressão da linha em si do tabuleiro, separando as diferentes células por linhas verticais (‘|’).

Um exemplo de tabuleiro seria o seguinte:

---------------------------------

| | X | | | | O | | |

---------------------------------

| O | X | | | | | | |

---------------------------------

| | | O | | O | O | | |

---------------------------------

| O | | | | | | | X |

---------------------------------

| | X | | | X | | | |

---------------------------------

| | X | | | | | | |

---------------------------------

| | | | | O | | | O |

---------------------------------

| | | O | | | | | |

---------------------------------

1. Resultados

A implementação realizada pode ser usada tanto para a geração como para a solução de um tabuleiro. Assim, é importante perceber o desempenho das técnicas usadas para ambas as situações, discutindo as suas vantagens e desvantagens.

* 1. Geração de tabuleiros

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

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

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

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

**Fig. 2.** Médias dos tempos de execução de ambos os algoritmos de geração ao longo de 10 execuções

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

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

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

* 1. Solução de tabuleiros

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

**Fig. 3.** Tempo médio de execução do predicado “*solver*” com e sem a opção “*enum*”, ao longo de 10 execuções para tamanhos variáveis

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

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

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

1. Conclusões e Trabalho Futuro

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

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

Embora seja reconhecido que o desempenho de geração ou solução de tabuleiros de grande dimensão poderia ser melhorado por via de melhores restrições em Prolog ou melhores opções de “*labeling*” e que esse seria o foco de trabalho futuro, é também possível que essa limitação esteja também inerente ao SICStus Prolog pelo que melhorar os resultados num fator relevante pode ser impossível.

Bibliografia

1. Conceptis Ltd.: Conceptis Puzzles, Tic-Tac-Logic, <http://www.conceptispuzzles.com/index.aspx?uri=puzzle/tic-tac-logic> (página consultada pela última vez a 08/12/2015 pelas 10:30)

Anexos

Código fonte, ficheiros de dados e resultados, e outros elementos úteis que não sejam essenciais ao relatório (não são contabilizados para o limite de 6 a 8 páginas).