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 (‘|’).

1. Resultados

Demonstrar exemplos de aplicação em instâncias do problema com diferentes complexidades e analisar os resultados obtidos. Devem ser utilizadas formas convenientes para apresentação dos resultados (tabelas e/ou gráficos).

1. Conclusões e Trabalho Futuro

Que conclusões retira deste projeto? O que mostram os resultados obtidos? Quais as vantagens e limitações da solução proposta? Como poderia melhorar o trabalho desenvolvido?

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).