FEUP-PLOG, Turma 3MIEIC01,
Grupo Hamle_2

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

Resolução de Problema de Decisão/Otimização usando Programação em Lógica com Restrições

– Hamle –

December 13, 2015

Springer

Resumo/Abstract

Tendo como material de estudo a programação em lógica com restrições, é-nos proposta a resolução de um problema de decisão relativo a um jogo de tabuleiro. Para o efeito, o nosso grupo selecionou o jogo Hamle, como objeto de trabalho.
Teremos como objetivo a determinação de soluções para o puzzle adotado, bem como a geração aleatória de tabuleiros de maiores dimensões que permitam a análise do tempo de execução e numero de calculos para casos diferentes.
Procuramos, com isto, avaliar o desempenho da Programação Lógica com Restrições aplicada à determinação de soluções para Puzzles 2D, para diferentes modo de *labeling*.

Faculdade de Engenharia da Universidade do Porto, *Daniel Reis*
Dezembro de 2015 *Guilherme Pinto*

Contents

1

Introdução

Neste relatório estudaremos a geração de soluções com base na programação em lógica com restrições, demonstrando os resultados obtidos e comparando os tempos de execução para tabuleiros de diferentes dimensões.
Relativamente ao problema abordado, o Hamle consiste num jogo de tabuleiro quadrangular (6x6), com dez peças pretas posicionadas em locais específicos. Por sua vez, cada uma destas peças apresentará um número correspondente ao deslocamento que deverá efetuar em qualquer umas das quatro direções possíveis. Após o deslocamento, nenhuma peça preta poderá ser adjacente a outra, assim como todas as células vazias deverão estar interligadas, ou seja, nao poderã existir dois ou mais grupos distintos de células vazias.
Quanto à estruturação do relatório, orientar-nos-emos por uma análise detalhada às restrições impostas, descrevendo de seguida o código implementado, estudando os resultados apresentados em terminal e discutindo os valores e soluções obtidas.

2

Descrição do Problema

O problema em análise deve ser abordado segundo as restrições descritas nas regras do jogo:

- Cada peça preta deve ser deslocada, em qualquer uma das quatro direções possíveis, o número de células correspondente ao valor que lhe está atribuído.
- No final do movimento das peças pretas, nenhuma delas deverá ser adjacente a qualquer outra peça.
- Além da restrição anterior, na conclusão dos deslocamentos das peças, deve-se verificar que todas as células vazias deverão estar interligadas entre si, ou seja, selecionando qualquer posição livre do tabuleiro, deve ser possível encontrar um caminho até qualquer outra célula desocupada, sem saltar por cima de qualquer peça preta.

O alcance de uma solução correta é apenas possível recorrendo a implementação das restrições apresentadas. Dado que qualquer uma delas é dependente dos resultado das restantes, é importante recorrer a programação lógica com restrções de modo a determinar uma resolução rápida e eficaz.

3

Abordagem

Neste capítulo será realizada uma minuciosa análise ao jogo Hamle e ao método de resolução implementado.
Começaremos por descrever as variáveis de decisão e seus respetivos domínios, seguidas de uma avaliação das restrições a serem trabalhadas. Será ainda indicada a forma de avaliação da solução obtida bem como a estratégia de pesquisa aplicada ao *labeling*, no cálculo das possíveis soluções.

3.1 Variáveis de Decisão

Como variavel de decisão, é gerada, na função *solution*, uma lista denominada *Result* a qual terá a dimensão de um tabuleiro, simulando a junção de uma linha do tabuleiro ao final da anterior, isto é, tendo como exemplo um tabuleiro NxN, temos que $length(Result, Length)$, onde $Length = N * N$.
O domínio da variável *Result* será compreendido entre 0 e $N-1$, sendo que 0 representará as células vazias e os algarismos compreendidos entre 1 e $N-1$ corresponderão à deslocação respetiva a cada peça preta presente no tabuleiro original. Sendo assim, a cada algarismo V presente num dado índice I da lista *Result*, então, necessáriamente, existirá na lista original (representativa do estado inicial do tabuleiro), o mesmo valor N no índice $I + N * V$ (deslocamento para baixo), no índice $I - N * V$ (deslocamento para cima), no índice $I + V$ (deslocamento para a direita) ou no índice $I - V$ (deslocamento para a esquerda). Não pode deixar de ser referido que no deslocamento horizontal, o índice de destino deverá corresponder à mesma linha do índice de origem, pelo que $floor(Idestino/N) = floor(Iorigem/N)$.

3.2 Restrições

No presente subcapítulo, descreveremos a implementação das diversas restrições que constituem o problema da determinação de soluções para o jogo Hamle. Para qualquer umas das restrições foi criada uma função auxiliar para o respetivo cálculo ou verificação.

3.2.1 Peças não-adjacentes

Esta é a restrição mais simples das que consistem o jogo, consistindo numa restrição rígida.

```
54 %**********************************%
55 %*****BLACKS ADJACENCY CONSTRAIN******%
56 %**********************************%
57 blacksNotAdjacent(ResultBoard):-
58     transpose(ResultBoard, TResultBoard),
59     checkAdjacentCells(TResultBoard),
60     checkAdjacentCells(ResultBoard).
61
62 checkAdjacentCells([]).
63 checkAdjacentCells([Head | Tail]):-
64     checkAdjacentCellsLine(Head),
65     checkAdjacentCells(Tail).
66
67
68 checkAdjacentCellsLine([_ | []]).
69 checkAdjacentCellsLine([First, Second | Tail]):-
70     First #>0 #=> Second #= 0,
71     checkAdjacentCellsLine([Second | Tail]).
72 checkAdjacentCellsLine([_ | Tail]):-
73     checkAdjacentCellsLine(Tail).
74
```

Tendo como argumento *ResultBoard* (uma lista de listas respetiva à divisão da lista de variáveis *Result*, sendo que cada sublista representa uma linha do tabuleiro), temos como objetivo garantir que nenhuma posição com valor superior a 0 tenha na sua proximidade outro valor igualmente superior a 0. Desse modo, ao percorrer uma linha do tabuleiro (sublista de *ResultBoard*), impomos que, dada uma posição, referente a uma peça preta, deverá ser seguida de um espaço em branco (valor 0). Uma vez que esta condição se verifica tanto na horizontal como na vertical, gera-se uma matriz transposta a *ResultBoard* que posteriormente será verificada com as restrição em análise.

3.2.2 Movimento de peças pretas

Relativamente ao movimento das peças pretas, podemos dividir a restrição em duas fases: a verificação das possibilidades de deslocação de cada peça e a respetiva validação e colocação em tabuleiro. Salienta-se ainda o facto de ser a única restrição flexível do problema.

```
80  createTuples(0,[]).
81
82  createTuples(NumberBlacks,[L|ListTuples]):-
83      NumberBlacks> 0, N1 #= NumberBlacks - 1, length(L,1),createTuples(N1,ListTuples).
84
85  checkIfRight(Index-Value,N,[Dest]):-
86      OriginalFloor is floor((Index-1) / N),
87      Dest #= (Index + Value),
88      DestFloor is floor((Dest-1) / N),
89      DestFloor =:= OriginalFloor.
90  checkIfRight(_,_,[]).
91
92  checkIfLeft(Index-Value,N,[Dest]):-
93      OriginalFloor is floor((Index-1) / N),
94      Dest #= (Index - Value),
95      DestFloor is floor((Dest-1)/N),
96      DestFloor =:= OriginalFloor.
97  checkIfLeft(_,_,[]).
98
99  checkIfUp(Index-Value,N,[Dest]):-
100     Dest is (Index - Value*N),
101     Dest >= 1.
102 checkIfUp(_,_,[]).
103
104 checkIfDown(Index-Value,N,[Dest]):-
105     Dest is (Index + Value*N),
106     Dest =< (N*N).
107 checkIfDown(_,_,[]).
```

Nestas funções são recriadas as possiveis posições que cada peça poderá ocupar, dependendo da célula onde se encontra e da direção na qual se poderá deslocar. Estas posições a serem determinadas são calculadas com base no índice de cada célula, pelo que se recorre à função $floor/1$ para considerar que, em caso de deslocamento horizontal, o indice de origem corresponde à mesma linha de tabuleiro que o índice de destino.

```
109 tablingTuples([],[],_).
110
111 tablingTuples([Tuple|ListTuples],[Piece|ListBlacks],N):-
112     checkIfRight(Piece,N,Dest1),
113     checkIfLeft(Piece,N,Dest2),
114     checkIfUp(Piece,N,Dest3),
115     checkIfDown(Piece,N,Dest4),
116     table([[Tuple]],[Dest1,Dest2,Dest3,Dest4]),
117     tablingTuples(ListTuples,ListBlacks,N).
118
119 putPiecesInBoard([],[],_).
120 putPiecesInBoard([Tuple|ListTuples],[_-Value | ListBlacks],Result):-
121     element(Tuple,Result,Value),
122     putPiecesInBoard(ListTuples, ListBlacks ,Result).
123
124 restrictMovements(Board,NumberWhites,N,Result):-
125     NumberBlacks #= (N*N)-NumberWhites,
126     length(ListTuples,NumberBlacks),
127     findall(Index-Value,(nth1(Index,Board,Value),Value>0),ListBlacks),
128     tablingTuples(ListTuples,ListBlacks,N),
129     all_different(ListTuples),
130     putPiecesInBoard(ListTuples,ListBlacks,Result).
131
```

Após a determinação dos possíveis destinos para cada peça preta, procede-se à colocação de cada uma no seu novo local recorrendo ao *table*, que nos permite criar um subdomínio relativo a cada objeto, com as diferentes posições para onde se poderá mover. Neste momento, é também restringida a possibilidade de quaisquer peças ocuparem a mesma célula.

3.2.3 Interconectividade das células vazias

Embora o conceito de interconectividade não seja complicado de assimilar, não fomos capazes de implementar corretamente a restrição pretendida neste ponto. Na imagem seguinte apresentamos o algoritmo que nos permite detetar o número de células vazias às quais podemos aceder, partindo de uma posição livre.

```
137 whiteInterconnection([RHead | RTail], NumberWhites, N):-
138     generateBoard(N, N, RegisterBoard),
139     findFirstWhite(1, RHead, Col),
140     floodWhites(1, Col, [RHead | RTail], RegisterBoard, _, NumberWhites).
141
142 findFirstWhite(Col, [Head | _], Col):-
143     Head #= 0.
144 findFirstWhite(Col, [_ | RTail], FinalCol):-
145     NextCol is Col + 1,
146     findFirstWhite(NextCol, RTail, FinalCol).
147
148 floodWhites(Row, Col, ResultBoard, RegisterBoard, FinalRegisterBoard, ConnectedWhites):-
149     Value #= 0,
150     getPosition(Row, Col, Value, ResultBoard),
151     getPosition(Row, Col, 0, RegisterBoard),
152     setPosition(Row, Col, 1, RegisterBoard, R1),
153     NextRow is Row + 1,
154     PrevRow is Row - 1,
155     NextCol is Col + 1,
156     PrevCol is Col - 1,
157     floodWhites(NextRow, Col, ResultBoard, R1, R2, W1),
158     floodWhites(PrevRow, Col, ResultBoard, R2, R3, W2),
159     floodWhites(Row, NextCol, ResultBoard, R3, R4, W3),
160     floodWhites(Row, PrevCol, ResultBoard, R4, FinalRegisterBoard, W4),
161     ConnectedWhites #= 1 + W1 + W2 + W3 + W4.
162 floodWhites(_,_,_,R,R,0).
```

O objetivo do nosso grupo seria, após detetar a primeira célula vazia da primeira linha de tabuleiro, expandir a procura através das células desocupadas na vizinhança dessa primeira posição, recorrendo ao auxílio de um tabuleiro de registo, que nos identificaria as células anteriormente visitadas durante o processo.
Tendo testado o algoritmo, podemos assegurar que se encontra funcional, embora não seja viável ao pretendido neste projeto.

3.3 Função de Avaliação

Após a execução das restrições e alcance de uma solução que as satisfaça, será possível visualizar um novo tabuleiro com a respetiva movimentação de cada peça preta. Tal poderá ser observado uma vez que cada peça conservará o seu valor de deslocamento, assim como no tabuleiro original.

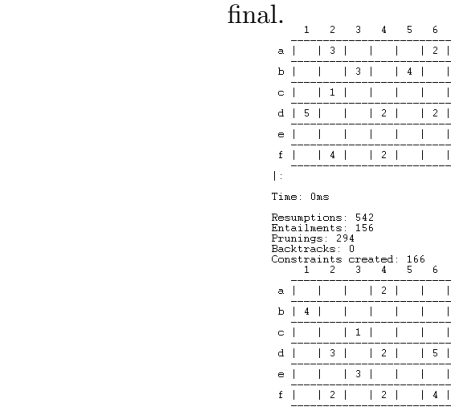
3.4 Estratégia de Pesquisa

São utilizadas, como estratégias de etiquetagem, os modos $default$ e ffc da função *labeling*, ou seja, recorre-se às opções $leftmost$, *step*, *up* e *all*. Estes modos poderão ser escolhidos pelo utilizador, no menu da aplicação pressionando '1' para o modo $default$ ou '2' para o modo ffc.

4

Visualização da Solução

Na imagem que se segue, apresentamos a visualização do problema, representando o primeiro tabuleiro o problema inicial, e o segundo tabuleiro o resultado final.



Como funções de auxílio para impressão da informação implementamos as funções *displayBoard*, *displayRows*, *displayRow*, *displayColIndexes* e *displayRowIndexes* as quais descrevemos a respetiva funcionalidade de seguida:

- *displayBoard* - Responsável pela impressão dos índices de cada coluna, da fronteira superior e da totalidade do tabuleiro.
- *displayRows* - Responsável pela impressão de todas a linhas do tabuleiro, assim como os respetivos indices e fronteiras inferiores de cada linha.
- *displayRow* - Responsável pela impressão da informação de cada linha do tabuleiro.
- *displayRowIndex* - Responsável pelo calculo e impressão do índice correto para cada linha do tabuleiro.
- *displayColIndexes* - Responsável pelo cálculo e impressão dos índices de todas as colunas do tabuleiro.

```
displayBoard([]).
displayBoard([Head | Tail]):-
    length(Head, Size),
    write('   '), displayColIndexes(Size, 0), nl,
    write('   '), displayBorder(Size),
    displayRows([Head | Tail], 97).

displayBorder(0):- write('-'), nl.
displayBorder(Number):-
    write('----'),
    NextNumber is Number - 1,
    displayBorder(NextNumber).

displayRows([], _).
displayRows([Head | Tail], RowNumber):-
    displayRowIndex(RowNumber),
    length(Head, Size),
    displayRow(Head),
    write('   '), displayBorder(Size),
    NewRowNumber is RowNumber + 1,
    displayRows(Tail, NewRowNumber).

displayRow([]):-
    write('|'), nl.
displayRow([0 | Tail]):-
    write('| '), write(' '), write(' '), displayRow(Tail).
displayRow([Head | Tail]):-
    write('| '), write(Head), write(' '), displayRow(Tail).

displayColIndexes(Size, Size).
displayColIndexes(Size, Index):-
    NewIndex is Index + 1,
    NewIndex < 10,
    write('  '), write(NewIndex),  write(' '),
    displayColIndexes(Size, NewIndex).
displayColIndexes(Size, Index):-
    NewIndex is Index + 1,
    write('  '), write(NewIndex),
    displayColIndexes(Size, NewIndex).

displayRowIndex(CharCode):- format(' ~c ', [CharCode]).
```

5

Resultados

Foram incluídas as funções fornecidas pelos docentes da disciplina de Programação em Lógica para cálculo e determinação do tempo de execução do programa e de quantidade de $resumptions$, $entailments$, $Prunings$ e $Backtracks$.
Tendo em conta os resultados obtidos, o tempo de execução do não variou de forma a que pudessem ser tiradas conclusões, quer no modo $default$, quer no modo ffc, apresentando sempre uma resultado de 0ms.
Contudo, salienta-se que a quantidade de *Resumptions* varia de forma ligeiramente exponencial, servindo como exemplo para todos os restantes parâmetros:

6

Conclusões e Trabalho Futuro

Como conclusão do trabalho realizado, mostramo-nos descontentes pela incapacidade de recriar a restrição de interconectividade das células brancas. Devido a esta falha na geração de resultados, não nos é possível apresentar uma solução fiel ao jogo original.

Relativamente aos resultados obtidos, com os computadores utilizados, não nos foi possível tirar conclusões relativas aos tempos de execução da aplicação. Contudo, podemos concluir que a quantidade de *resumptions*, *prunings* e *entailments* acompanha de igual forma (exponencialmente) com o aumento da àrea do tabuleiro.

Mais uma vez, lamentamos a falha na implementação da última restrição apresentada, bem como o facto da geração aleatória de novos tabuleiros bloquear, por vezes, a aplicação, por motivos que nos foram impossíveis de detetar. Tendo a consciencia da penalização que advém, esforçar-nos-emos por emendar estas fragilidades no projeto de forma a assegurar o nosso conhecimento na matéra de Programação Lógica com Restrições.

References

1. Slides de apoio à unidade curricular de Programação em Lógica da Faculdade de Engenharia da Universidade do Porto