Resolução de Problema de Decisão usando Programação em Lógica com Restrições - Chess Num

Mariana Oliveira Ramos[up201806869],
Pedro Varandas da Costa Azevedo da Ponte[up201809694]

Mestrado Integrado em Engenharia Informática e Computação – 3º Ano
Programação em Lógica 2020/2021
Chess Num Grupo ?
Faculdade de Engenharia da Universidade do Porto, R. Dr. Roberto Frias,
4200-464 Porto, Portugal

Resumo. Este artigo tem como objetivo demostrar o processo de desenvolvimento do segundo projeto da Unidade Curricular de Programação em Lógica, assim como os resultados obtidos. O projeto consiste num programa escrito em Prolog, capaz de resolver qualquer instancia do puzzle Chess Num, cuja descrição poderá ser encontrada em https://erich-friedman.github.io/puzzle/chessnum/. Este for modelado como um PSR (Problema de Satisfação de Restrições) e resolvido utilizando a bibliotecao ‚clpfd' do SICStus Prolog, que permite implementação de PLR (Programação em Lógica com Restrições). O programa também possui a capacidade de gerar novos puzzle dinamicamente. Após análise de eficiencia, concluimos que o programa executa de forma quase instantanea para problemas com dimensao (?), subindo sempre de modo exponencial. O gerador de problemas consegue quase sempre concluir, em poucos segundos, a criação de novos puzzles até com diferentes dimensões.

Palavras-chave: puzzle, clp, prolog, plog, feup

1 Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular de Programação em Lógica 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 puzzel 2D Chess Num.

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

2. Descrição do problema
3. Abordagem
    Variaveis de Decisão
    Restrições
    Geração de Puzzels
4. Visualização da solução
5. Experiencias e Resultados
    Análise Dimensional
    Estratégias de Pesquisa
6. Conclusões e Trabalho Futuro

2 Descrição do problema

Chess Num é um puzzle realizado num tabuleiro de xadrez, por default 8x8, que inicialmente é preenchido com números que indicam quantas vezes uma célula é atacada.

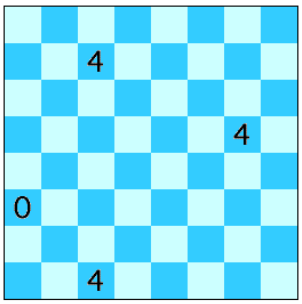


Figura 1: Exemplo do Puzzle Chess Num

O objetivo é preencher 6 casas com 6 peças (um Rei, uma Rainha, uma Torre, um Bispo, um Cavalo, e um Peão) de forma a que não haja duas peças no mesmo quadrado, nenhuma peça esteja num quadrado numerado e os ataques a cada célula correspondam ao numero de ataques indicado no tabuleiro inicial.
Em baixo está representada a solução do tabuleiro anterior.

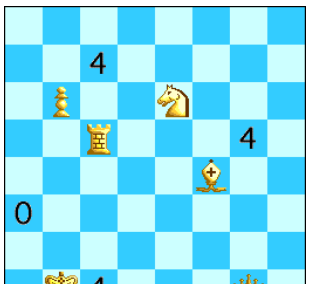


Figura 2: Exemplo do Puzzle Chess Num

Na célula [12] (linha 1 coluna 2) o número de ataques é 4, representando os possíveis ataques do Cavalo, Peão, Torre e Bispo. Na célula [51] o número de ataques é 0. Na célula [36] o número de ataques é 4, realizados pelo Bispo, Rainha, Cavalo e Torre. Por último na célula [72] o número de ataques é 4 representando os ataques possíveis do Rei Rainha, Torre e Bispo.

3 Abordagem

O primeiro passo na abordagem foi tentar perceber como modelar o puzzle como um problema de restrições. Entender as variáveis de decisão a usar no predicado labeling, as restrições necessárias para o problema e restringir essas variáveis.
Foi ainda tida em conta a melhor forma de interagir com os utilizadores, ou seja, a melhor forma do puzzel ser visualizado. Sendo a consola SICStus Prolog muito simples, a representação das peças é feita com as letras K (Rei), Q (Rainha), R (Torre), C (Cavalo), B (Bispo), P (Peão).

3.1 Variáveis de Decisão

A solução pretendida para este puzzel é o mesmo tabuleiro, mas com o preenchimento das peças nas posições (linha e coluna) corretas. Neste sentido, a única variável de decisão que o nosso problema necessita e utilizada no predicado labeling, é uma lista Positions[] que contém a linha e coluna de cada peça que deve ser colocada. A lista tem a seguinte forma:

```
Positions = [KingRow, KingCol, QueenRow, QueenCol, RookRow, Rook
Col, BishopRow, BishopCol, KnightRow, KnightCol, PawnRow, PawnCo
l].
```

Esta variável tem como domínio [0,Tamanho_do_tabuleiro] :

```
domain(Positions, 0, size).
```

3.2 Restrições

Em primeiro lugar, na inicialização da variável de decisão foi imposto que em cada célula o domínio é entre 0 e 7, representando o número da linha/coluna.
De seguida, foi necessário garantir que não existem duas peças na mesma posição. Para isso, foi desenvolvido o predicado differentPossitions(+Possitions) que coloca restrições de forma a garantir que não existem duas peças com a mesma linha e coluna em simultâneo:

```
differentPositions([KingR, KingC, QueenR, QueenC, RookR, RookC,
BishopR, BishopC, KnightR, KnightC, PawnR, PawnC]) :-
    KingPos #= 10 * KingR + KingC,
```

```
    QueenPos #= 10 * QueenR + QueenC,
    RookPos #= 10 * RookR + RookC,
    BishopPos #= 10 * BishopR + BishopC,
    KnightPos #= 10 * KnightR + KnightC,
    PawnPos #= 10 * PawnR + PawnC,
    KingPos #\= QueenPos,
    KingPos #\= RookPos,
    KingPos #\= BishopPos,
    (…)
```

De forma a garantir que o número de ataques por célula é o pretendido utilizamos o predicado sumAttacks. Este predicado garante que uma peça não se encontra na mesma posição que um número. Adicionalmente, define as restrições relativas ao movimento de cada peça nas funções 'validate' e aos ataques na mesma linha/coluna com as funções 'check' para as peças que não podem saltar (Torre, Rainha e Bispo).
A variável 'Attack' é então o número total de ataques na célula [Row,Column], o que corresponde à soma de ataques de todas as peças nessa mesma célula.

```
sumAttacks([KingR, KingC, QueenR, QueenC, RookR, RookC, BishopR,
 BishopC, KnightR, KnightC, PawnR, PawnC], Attack - Row - Column
):-

KingAttack + QueenAttack + RookAttack + BishopAttack + KnightAtt
ack + PawnAttack #= Attack,

(PawnR #\= Row #/\ PawnC #\= Column) #\/ (PawnR #= Row #/\ PawnC
 #\= Column) #\/ (PawnR #\= Row #/\ PawnC #= Column),
validatePawnMove(PawnR, PawnC, Row, Column, PawnAttack),

(RookR #\= Row #/\ RookC #\= Column) #\/ (RookR #= Row #/\ RookC
 #\= Column) #\/ (RookR #\= Row #/\ RookC #= Column),
checkRookPositions([KingR, KingC, QueenR, QueenC, RookR, RookC,
BishopR, BishopC, KnightR, KnightC, PawnR, PawnC], Row, Column,
[K1, Q1, B1, Kn1, P1]),
validateRookMove(RookR, RookC, Row, Column, [K1, Q1, B1, Kn1, P1
], RookAttack),

    ( … efetuamos o mesmo processo para cada peça).
```

Os predicados validateMove têm como objetivo impor as restrições que delimitam os possíveis movimentos de cada peça para [Row,Col], definindo a variável Attack com 0 ou 1.
A baixo apresentamos o exemplo das restrições de movimento impostas à peça Torre.

```
validateRookMove(RookR, RookC, Row, Col, [K1, Q1, B1, Kn1, P1],
Attack) :-

    (((RookR #= Row) #\/ (RookC #= Col)) #/\ K1 #/\ Q1 #/\ B1 #/
\ Kn1 #/\ P1) #<=> Attack.
```

A Torre pode-se mover para todos os lados por isso precisa de cumprir a restrição que dita que a sua linha tem que ser igual à linha final ou a sua coluna igual à coluna final. A variável 'Attack' é 1 se esta restrição se impuser e se nenhum dos valores K1, Q1, B1, Kn1 e P1 tiver o valor de 0.

Previamente, com a função 'check', instanciamos as variáveis `K1, Q1, B1, Kn1, P1.` Estas variáveis podem ter como valores 0 ou 1.
Se o seu valor for 0 então significa que existe uma peça entra a célula a atacar C [Row,Col] e a própria peça [RookR, RookCol] , ou seja a Torre não pode atacar essa célula C. Caso contrário, se for 1, então significa que pode atacar porque não existe nenhuma peça no meio.

```
checkRookPositions([KingR, KingC, QueenR, QueenC, RookR, RookC,
BishopR, BishopC, KnightR, KnightC, PawnR, PawnC], Row, Col, [K1
, Q1, B1, Kn1, P1]) :-
checkPieceRow(Row, Col, KingR, KingC, RookR, RookC, KR),
checkPieceCol(Row, Col, KingR, KingC, RookR, RookC, KC),
K1 #<=> KR #\/ KC,
checkPieceRow(Row, Col, QueenR, QueenC, RookR, RookC, QR),
checkPieceCol(Row, Col, QueenR, QueenC, RookR, RookC, QC),
Q1 #<=> QR #\/ QC,
checkPieceRow(Row, Col, BishopR, BishopC, RookR, RookC, BR),
checkPieceCol(Row, Col, BishopR, BishopC, RookR, RookC, BC),
B1 #<=> BR #\/ BC,
checkPieceRow(Row, Col, KnightR, KnightC, RookR, RookC, KnR),
checkPieceCol(Row, Col, KnightR, KnightC, RookR, RookC, KnC),
Kn1 #<=> KnR #\/ KnC,
checkPieceRow(Row, Col, PawnR, PawnC, RookR, RookC, PR),
checkPieceCol(Row, Col, PawnR, PawnC, RookR, RookC, PC),
P1 #<=> PR #\/ PC.
```

Neste predicado utilizamos os predicados auxiliares 'checkPieceRow' e 'checkPieceCol' para fazer as verificações da Torre com cada peça existente no tabuleiro (Rei, Rainha, Bispo, Cavalo e Peão). Os valores K1, Q1, B1, Kn1, P1 tomam assim os valores 0 ou 1 consoante o resultado da operação lógica "KR v KC". Onde KR indica se a peça (neste caso o rei [KingR,KingC]) se situa no meio da [Row,Col] e [RookR, RookC] na mesma linha, e KC no meio das duas na mesma coluna.

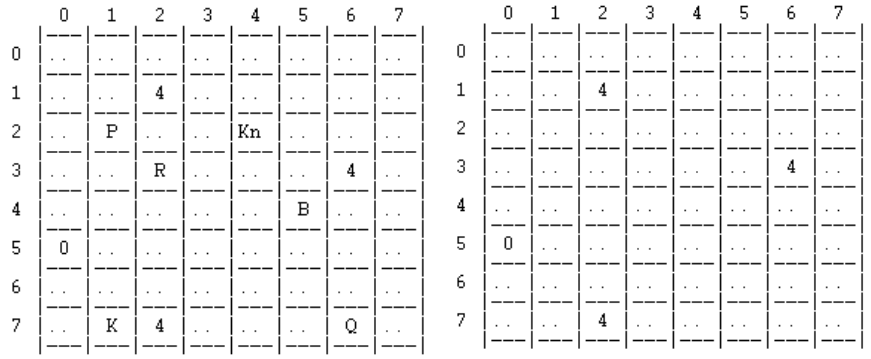
3.3 Gerador Aleatório do Puzzel a Resolver

Primeiramente utilizamos o predicado `generateMatrix` que gera uma matriz vazia com o tamanho pretendido. De seguida, utilizando o predicado random da biblioteca random, geramos números aleatórios entre 0 e 4 e colocamo-los em posições aleatórias com auxilio do predicado `replace_value_in_matrix`. Por último chamamos o predicado que coloca as restrições necessárias para resolver o puzzle.
O programa resolve qualquer puzzle independentemente do tamanho dado.
O puzzle é inicialmente mostrado ao utilizador apenas com os números, pronto a ser resolvido.

4 Visualização da Solução

A solução do puzzle Chess Num consiste num tabuleiro igual ao inicial mas com peças adicionais. Assim, a sua visualização é idêntica à do tabuleiro inicial. Utilizamos então o mesmo predicado `printBoard` que imprime linha a linha (predicado `print_line`) cada célula do tabuleiro (predicado `code`).

Para além do tabuleiro final a nossa solução mostra também o tempo que demorou a resolver o Puzzle. Para isso usamos o predicado `printStatistics`.



Solution Time: 1156ms

5. Experiências e Resultados

5.1 Analise dimensional

Alguns cuidados para a otimização da resolução do puzzle foram tidos em conta, como a escolha da estratégia de pesquisa. No entanto, apesar do programa conseguir gerar e resolver tabuleiros com tamanhos aleatórios de 8x8 a 100x100, a sua análise dimensional torna-se complicada uma vez que o tempo de resolução de um puzzle depende da sua complexidade. Um puzzle 8x8 pode demorar minutos enquanto que um 100x100 ser instantâneo.
Independentemente desse facto decidimos fazer o mesmo puzzle (mesma complexidade) nas dimensões 8x8, 16x16 e 32x32 os resultados temporais foram respetivamente: 609ms, 4469ms e 4406ms.

5.2 Estratégias de Pesquisa

A estratégia de labeling implementada no programa foi a ffc – first fail constraint. Esta estratégia tornou a pesquisa mais eficiente fazendo com que fosse usada a restrição mais rápida: é escolhida a variável de domínio mais pequeno, com menos restrições e mais à esquerda.
A baixo encontra-se uma comparação entre o tempo de pesquisa de 2 tabuleiros, com e sem fcc:

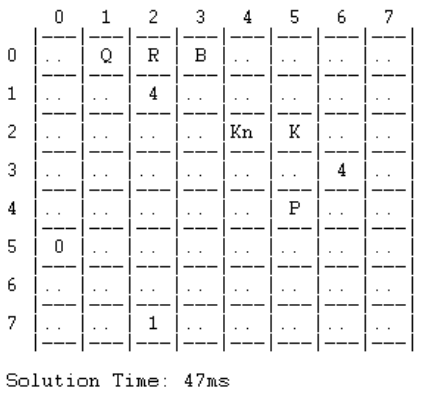


Figure 3 - tempo de solução usando estratégia fcc – 47ms

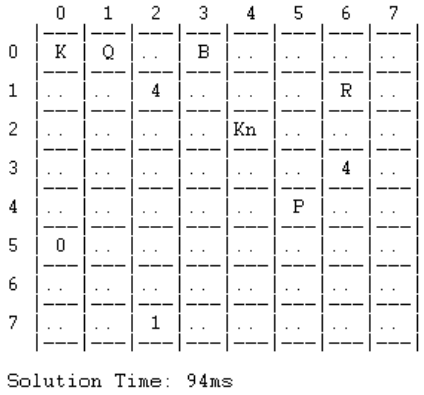


Figure 4 - tempo de solução sem estratégia fcc – 94ms

5 Conclusões e Trabalho Futuro

Após a realização deste projeto em Prolog, conclui-mos que a linguagem Prolog, em particular, o uso de restrições, é muito útil para a resolução de problemas de decisão e otimização.
Os resultados obtidos permitiram-nos perceber, ao longo do desenvolvimento do projeto, as variações de eficiência do mesmo aquando das diferentes restrições impostas e das diferentes estratégias de pesquisa.
Note-se que existem aspetos que podiam ser melhorados, como a escolha de um método mais eficiente e otimizado.
Ao longo do desenvolvimento deste projeto, foram encontradas algumas dificuldades, na consideração que duas peças podem não atacar na mesma casa se uma estiver à frente da outra.
Apesar disso, 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.