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

Ricardo Boia[up201505244] and Tiago Castro[up201606186]

FEUP-PLOG, Turma 3MIEIC3, Grupo Chess Loop_3

Abstract. Este projeto teve como objetivo o desenvolvimento de uma aplicação em *SICStus Prolog* que apresenta as soluções válidas em modo texto de *Chess Loop puzzles*, dadas as dimensões do tabuleiro, número de peças e os seus tipos, através da aplicação de restrições. Na resolução do problema foi utilizada uma lista com os tipos de peça alternados, e restringiu-se o ataque de cada peça apenas à seguinte peça da lista, sendo que a última ataca a primeira. Destaca-se também a geração automática de problemas validos e a respetiva solução executada pelo predicado *random_problem*.

Keywords: *Chess Loop* · Prolog · Restrições.

1 Introduction

Tinhamos como objetivo neste projeto a construção de um programa em Programação em Lógica com Restrições para a resolução de *Chess Loop puzzles*.

Estes *puzzles* consistem em posicionar um número igual de peças de dois tipos (rei, rainha, torre, bispo ou cavalo) num tabuleiro com dimensões variadas de modo a que cada peça ataque apenas uma outra de tipo diferente e nenhuma do mesmo tipo. Para além disso, os ataques devem formar um ciclo alternativo.

Este artigo possui a seguinte estrutura:

- ⋆ **Descrição do Problema** - Descrição detalhada do problema a tratar.
- ⋆ **Abordagem** - Descrição da modelação do problema como um PSR, de acordo com as subsecções seguintes:
  - **Variáveis de Decisão** - Descrição das variáveis de decisão e os seus domínios.
  - **Restrições** - Descrição das restrições rígidas e flexíveis do problema e a sua implementação utilizando o *SICStus Prolog*.
  - **Estratégia de Pesquisa** - Descrição da estratégia de etiquetagem
  - **Geração aleatório de problemas** - Descrição da estratégia utilizada para gerar automaticamente problemas válidos (*labeling*) utilizada.
- ⋆ **Visualização da solução** - Explicação dos predicados que permitem visualizar a solução em modo de texto.

- ⋆ **Resultados** - Exemplos de aplicação em instâncias do problema com diferentes dimensões e análise dos resultados obtidos.
- ⋆ **Conclusões e Trabalho Futuro** - conclusões retiradas com o projeto, discussão dos resultados obtidos, avaliação da solução proposta, e como a melhorar.

2 Descrição do Problema

Este programa foi construído com o intuito de apresentar a solução de problemas *Chess Loop*. O objetivo destes *puzzles* é, dadas as dimensões do tabuleiro, dois tipos de peças de xadrez e o número de peças (número igual de peças de cada tipo), arranjar uma disposição do tabuleiro única (que estará sujeita a simetrias) em que cada peça ataque apenas uma peça de outro tipo e nenhuma do mesmo tipo. Para além disso os ataques devem formar um ciclo entre eles.

As peças podem ser de cinco tipos:

- ⋆ Rei - ataca todas as posições adjacentes, nas oito direções possivéis.
- ⋆ Rainha - ataca em toda a linha em todas as direções. No máximo uma peça por direção (o ataque não passa por cima das peças).
- ⋆ Torre - ataca em toda a linha na horizontal e vertical. No máximo uma peça por direção (o ataque não passa por cima das peças).
- ⋆ Bispo - ataca em toda a linha nas diagonais. No máximo uma peça por direção (o ataque não passa por cima das peças).
- ⋆ Cavalo - ataca em "L". Ataca as oito posições mais próximas que não estejam na mesma linha, coluna ou diagonal.

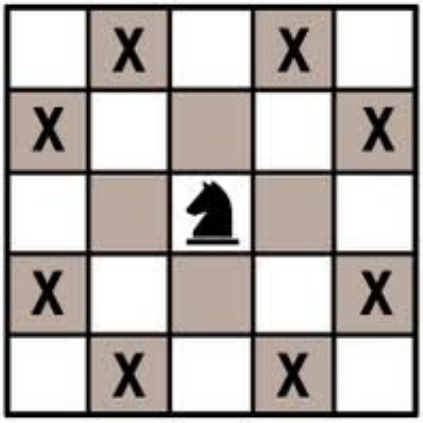

Fig. 1. Movimento do cavalo

3 Abordagem

Para a resolução deste problema foi utilizada uma lista de tamanho igual ao número de peças presentes no tabuleiro. Esta lista foi preenchida com os dois tipos de peça, de maneira alternada ([Tipo1, Tipo2, Tipo1, Tipo2, ...]). Desta forma, para resolver o problema, basta restringir o ataque de cada uma das peças apenas e só à peça seguinte na lista (e no caso da última à primeira da lista). O predicado encarregado de resolver o problema é o **solve(+(Nº peças de cada tipo),+(Nº linhas), +(Nº colunas), +Tipo1, +Tipo2, -Posições**, que cria e prepara as listas e variavéis a ser utilizadas durante a execução do programa.

3.1 Variáveis de Decisão

Neste projeto, as variáveis de decisão utilizadas são **Rows**, **Cols** e **Pos** (posições), que irá depender dos dois anteriores. O domínio de *Rows* será entre 1 e o número de linhas do tabuleiro, o de *Cols* será entre 1 e o número de colunas do tabuleiros e o de Pos será entre 1 e a multiplicação entre o número de linhas e colunas. O valor da posição irá incrementar ao longo do tabuleiro começando da esquerda para a direira, e de cima para baixo, como demonstrado no seguinte exemplo:

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

3.2 Restrições

Existem dois grandes grupos de restrições neste trabalho que são aplicadas a pares de peças: restrições para que uma ataque a outra (**eat**), e restrições para que uma não ataque as outras não envolvidas na jogada (**dont_eat**). Para cada um dos grupos existem cinco predicados, um para cada um dos tipos de peça.

Rei

A peça atacada tem que estar numa das 8 posições adjacentes ao rei. Todas as outras são restringidas para fora dessas 8 posições.

```
eat(1, R1, R2, C1, C2, _, _):-
  (R2 #= R1+1 #/\ (C2 #= C1 #\/ (C2 #= C1+1 #\/ C2 #= C1-1))) #\/
  (R2 #= R1 #/\ (C2 #= C1+1 #\/ C2 #= C1-1)) #\/
  (R2 #= R1-1 #/\ (C2 #= C1 #\/ (C2 #= C1+1 #\/ C2 #= C1-1))).

dont_eat(1, R1, R2, C1, C2, _, _, _, _):-
    R2 #> R1+1 #\/ R2 #< R1-1 #\/ C2 #> C1+1 #\/ C2 #< C1-1.
```

Rainha

A peça atacada está posicionada numa das diagonais, na mesma coluna ou na mesma linha que a rainha. Todas as outras são restringidas fora das diagonais, coluna e linha atacadas, exceto na direção que a peça atacada se encontra, visto que as posições após a peça encontram-se fora de perigo.

As restrições aplicadas são uma junção das restrições aplicados nos predicados da torre e do bispo.

Torre

A peça atacada está posicionada na mesma coluna ou na mesma linha que a torre. Todas as outras são restringidas fora da linha e da coluna exceto no sentido em que a peça atacada se encontra, sendo que toda as posições nesse sentido que seguem a peça posicionada mantêm-se livres.

```
eat(3, R1, R2, C1, C2, _, Max):-
    domain([X],1,Max),
    ((R2 #= R1 #/\ C2 #= C1+X) #\/
    (R2 #= R1 #/\ C2 #= C1-X) #\/
    (R2 #= R1+X #/\ C2 #= C1) #\/
    (R2 #= R1-X #/\ C2 #= C1)).
```

Max é a maior dimensão do board X é uma variavel comprimida entre 1 Max. X é usado para posicionar a peça nas linhas/colunas já que a maior distância entre a peça atacante a peça atacada vai ser igual à maior dimensão.

```
dont_eat(3, R1, R2, C1, C2, _, Max, R, C):-
    (R2 #= R1 #/\ C2 #= C1) #<=> 0,
    restrict_N(R1, R2, C1, C2, 1, Max, R, C),
    restrict_E(R1, R2, C1, C2, 1, Max, R, C),
    restrict_S(R1, R2, C1, C2, 1, Max, R, C),
    restrict_W(R1, R2, C1, C2, 1, Max, R, C).
```

Bispo

A peça atacada está posicionada numa das diagonais do bispo. Todas as outras são restringidas fora das diagonais exceto no sentido em que a peça atacada se encontra, sendo que toda as posições nesse sentido que seguem a peça posicionada mantêm-se livres.

```
eat(4, R1, R2, C1, C2, Min, _):-
    domain([X],1,Min),
    ((R2 #= R1+X #/\ C2 #= C1+X) #\/
    (R2 #= R1+X #/\ C2 #= C1-X) #\/
    (R2 #= R1-X #/\ C2 #= C1+X) #\/
    (R2 #= R1-X #/\ C2 #= C1-X)).
```

Min é a menor dimensão do board sendo que X é uma variavel comprimida entre 1 e Min. X é usado para posicionar a peça nas diagonais visto que a maior diagonal vai ter um numero de celulas igual à menor dimensão.

```
dont_eat(4, R1, R2, C1, C2, Min, _, R, C):-
    (R2 #= R1 #/\ C2 #= C1) #<=> 0,
    restrict_NE(R1, R2, C1, C2, 1, Min, R, C),
    restrict_SE(R1, R2, C1, C2, 1, Min, R, C),
    restrict_SW(R1, R2, C1, C2, 1, Min, R, C),
    restrict_NW(R1, R2, C1, C2, 1, Min, R, C).
```

Cavalo

A peça atacada está posicionada numa das 8 posições para onde o cavalo se pode mover. Todas as outras peças são restringidas fora dessas 8 posições.

```
eat(5, R1, R2, C1, C2, _, _):-
    (R2 #= R1+2 #/\ (C2 #= C1+1 #\/ C2 #= C1-1)) #\/
    (R2 #= R1-2 #/\ (C2 #= C1+1 #\/ C2 #= C1-1)) #\/
    (C2 #= C1+2 #/\ (R2 #= R1+1 #\/ R2 #= R1-1)) #\/
    (C2 #= C1-2 #/\ (R2 #= R1+1 #\/ R2 #= R1-1)).

dont_eat(5, R1, R2, C1, C2, _, _, _, _):-
    (R2 #= R1 #/\ C2 #= C1) #<=> 0,
    ((R2 #= R1+2 #/\ (C2 #= C1+1 #\/ C2 #= C1-1)) #\/
    (R2 #= R1-2 #/\ (C2 #= C1+1 #\/ C2 #= C1-1)) #\/
    (C2 #= C1+2 #/\ (R2 #= R1+1 #\/ R2 #= R1-1)) #\/
    (C2 #= C1-2 #/\ (R2 #= R1+1 #\/ R2 #= R1-1))) #<=> 0.
```

restrict_X

Os predicados de restrição dos varios sentidos percorrem o sentido correspondente desde a peça que ataca e proibem certas posiçoes das peças. Segue o exemplo do restrict_N

```
restrict_N(_, _, _, _, M, M, _, _).
restrict_N(R1, R2, C1, C2, N, M, R, C):-
    N < M,
    ((#\(C #= C1 #/\ R #< R1 #/\ R1-N #< R)) #=> ((R2 #= R1-N #/\ C2 #= C1) #<=> 0)),
    Next is N + 1,
    restrict_N(R1, R2, C1, C2, Next, M, R, C).

%R1 C1- posição da peça atacante
%R2 C2- posição da celula avaliada
%R C- posição da peça atacada
```

A restrição (R2 #= R1-N #/\C2 #= C1) #<=> 0 vai ser aplicada se a posição a ser avaliada NÃO se encontra depois da peça atacada. Primeiro é visto se as peças se encontram na mesma coluna, de após isso, se a peça atacada se encontra numa linha menor que a que ataca e após isso se a celula a ser avaliada está depois da peça atacada.

3.3 Estratégia de Pesquisa

Após serem testadas as oitenta combinações possíveis de variavéis de seleção e ordenação, e medidos os tempos de execução para diferentes tipos de problemas, chegámos à conclusão que a melhor combinação de opções era **ff** (first_fail), **bisect** e **up**. A escolha do **bisect** foi bastante fácil, visto que esta foi, por larga margem, a opção de seleção com os melhores tempos. Já em relação à opção de ordenação, decidimos escolher o **ff**, já que, apesar de a diferença ser pequena, esta foi opção com o menor tempo de execução, tanto utilizando **up**, como **down**. Finalmente, o **up** foi escolhido por ser a opção que melhor "combina" com o **ff**, visto que ambos iniciam a pesquisa pelo menor domínio. Todas as medições de tempo encontram-se em anexo, na secção 7.1.

3.4 Geração automática de problemas

Foi também implementado o predicado **random_problem(-(Nº linhas), -(Nº colunas), -Tipo1, -Tipo2, -(Nº peças de cada tipo))** que gera aleatoriamente um problema válido e resolve-o, apresentando a solução ao utilizador. Isto foi conseguido ao gerar aleatoriamente um valor para cada uma das variavéis passados como argumento e chamado o predicado **solve** para resolver o problema. Caso o problema não seja válido, irão ser gerados novos valores para as variavéis até que o problema o seja, e será mostrado ao utilizado a solução assim como os valores dados a cada uma das variavéis.

4 Visualização da solução

Para a visualização da solução do problema é utilizado o predicado **display_solution**, que irá receber como argumentos as dimensões do tabuleiro, e as listas do tipo, linha e coluna de cada peça.

Primeiro, irá ser criado um tabuleiro vazio (lista de listas) com as dimensões indicadas. Depois o tabuleiro irá ser preenchido com as peças de acordo com as informações presentes na listas passados como argumento. Finalmente irá ser invocado o predicado **display_board** que irá desenhar o tabuleiro, com recurso a *unicode*, de modo a utilizar o simbolo corresponde para cada tipo de peça e tornar a visualização da solução mais fácil para o utilizador.

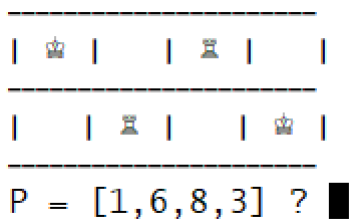

Fig. 2. Exemplo da solução em mode texto

5 Resultados

De modo a analisar a eficiência do código implementado, foram realizados testes variando as dimensões do tabuleiro, o número de peças e o tipo de peças. Os resultados observados foram os seguintes:

- **variação das dimensões do tabuleiro** - analisando a tabela que se encontra na secção 7.2, podemos observar que o tempo de execução aumenta com o aumento das dimensões do tabuleiro, tal como seria de esperar, visto que o número de linhas e colunas a analisar também aumenta.
- **variação do número de peças de cada tipo** - analisando a tabela que se encontra na secção 7.3, podemos observar que o tempo de execução aumenta com o número de peças presentes no tabuleiro, tal como seria de esperar, visto que o número de vez que o predicado **dont_eat** é chamado aumenta significativamente.
- **variação do tipo de peças** - analisando a tabela que se encontra na secção 7.4, podemos observar que o tempo de execução da rainha e do bispo é consideravelmente maior que o do rei e da torre. Podemos atribuir isso ao facto da aplicação de restrições às diagonais ser um processo mais demorado que os restantes.

6 Conclusões e Trabalho Futuro

Finalizado o projeto, podemos concluir que as restrições de Prolog são bastante úteis na resolução de certos problemas, como os de decisão ou otimização.

Ao longo do projeto tivemos vários dificuldades, derivados não só da falta de prática com este tipo de programação, mas também devido à complexidade do problema selecionado, tendo sido bastante difícil encontrar a melhor abordagem para o trabalho.

Foram testados todos os problemas existentes na página referida no enunciado, sendo que todos eles foram resolvidos com sucesso. Todavia, uma limitição da solução apresentada é a complexidade temporal do programa, visto que o facto de termos de restringir o ataque a apenas uma peça obriga o programa a percorrer a lista de peças várias vezes, sendo este o principal aspeto a melhorar no trabalho.

References

1. Chess Loop Puzzles, https://www2.stetson.edu/ efriedma/puzzle/chessloop/. Last accessed 23 Dec 2018

7 Anexos

7.1 Medição dos tempos variando as opções de *labeling*

Up

	first_fail	anti_first_fail	leftmost	max	max_regret	min	most_constrained	occurrence
Bisect	0.0500	0.0640	0.0578	34.7535	0.0601	2.1792	0.0590	0.0552
Enum	0.1142	0.1118	0.1049	0.1081	0.1200	0.1053	10.1096	0.1082
Middle	0.0802	0.0972	0.0783	0.0813	0.0907	1.0810	0.0864	0.0932
Median	0.0784	0.0855	0.0828	0.0891	0.0855	1.0869	0.1082	0.0877
Step	0.0816	0.0830	0.0832	0.0815	0.0842	0.9238	0.0845	0.0822

Down

	first_fail	anti_first_fail	leftmost	max	max_regret	min	most_constrained	occurrence
Bisect	0.0542	0.0589	0.0718	7.5694	0.0601	27.2474	0.0597	0.0623
Enum	0.1239	0.1352	0.1206	0.1263	0.1142	0.1149	0.1342	0.1266
Middle	0.0908	0.1332	0.0905	1.0716	0.0972	0.0949	0.0908	0.0903
Median	0.0906	0.0809	0.0968	1.1043	0.0769	0.0860	0.0787	0.0958
Step	0.0894	0.0987	0.0861	1.0578	0.0864	0.0753	0.1033	0.0976

7.2 Medição dos tempos variando apenas as dimensões do taabuleiro

2x4	10x20	50x100	200x400	500x1000
0.0004	0.0021	0.0122	0.0749	0.3760

7.3 Medição dos tempos variando apenas o número de peças de cada tipo (reis e torres)

2	4	6	8	10
0.0021	0.0639	0.0809	0.3062	0.6425

7.4 Medição dos tempos variando apenas um tipo de peça (Tipo1: Cavalo)

Rei	Rainha	Torre	Bispo
0.1179	0.9513	0.1746	0.8369