U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Programação em Lógica

Relatório: *"Mad Bishops"*

Turma 4

Beatriz Mendes: *up201604253*@fe.up.pt

João Malheiro: *up201605926*@fe.up.pt

Índice

Introdução

Este trabalho teve como objetivo fundamental o desenvolvimento de um jogo de tabuleiro em *Prolog* que respeitasse um aglomerado de regras, as quais ditam as jogadas possíveis juntamente com as condições de terminação do jogo.

O tema escolhido foi o jogo “Mad Bishops”.

Neste relatório será abordado a história e regras do jogo, assim como uma descrição da solução implementada incluindo extratos código e diagramas que consideramos relevantes na melhor representação da mesma.

Mad Bishops

"*Mad Bishops*" foi inventado por Mark Steere em Março de 2010. O seu objetivo é capturar todas as peças do adversário.

O jogo "*Mad Bishops*" é jogado por dois jogadores num tabuleiro de 10x10 casas. Inicialmente, cada jogador possui 25 bispos vermelhos ou azuis. Ambos os jogadores jogam em turnos alternados, movendo os bispos da sua cor: um movimento por turno, começando pelo jogador que possui as peças de cor vermelha. Os jogadores são obrigados a jogar nos seus turnos, não podendo passar a vez.

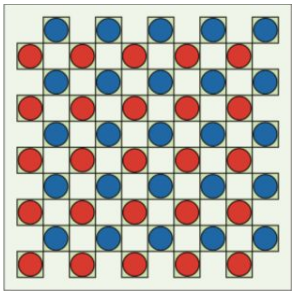

Figura 1: tabuleiro inicial

Bispos

Os bispos movem-se similarmente ao movimento dos bispos em Xadrez: um bispo pode ser movido nas suas diagonais, tanto para uma casa adjacente ou uma casa separada por uma ou mais quadrículas contíguas vazias.

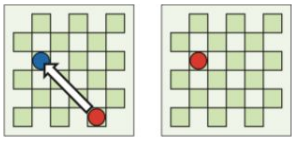

Figura 2: capturar adversário

Capturar Peças

Um bispo captura uma peça do adversário substituindo-a e colocando a sua.

Se um bispo se encontra numa posição em que pode **capturar** uma peça do adversário, é **obrigado** a fazê-lo.

Por exemplo, na figura 3, o jogador ao qual pertence a peça vermelha apenas lhe é permitido fazer o jogada mais à direita, de forma a capturar a peça do adversário.

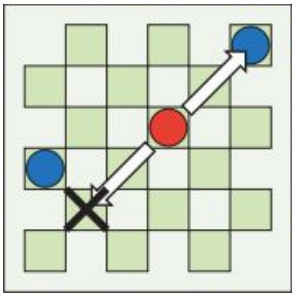

Figura 3: movimento obrigatório

No entanto, caso o jogador tenha mais peças que possa mover sem capturar, é permitido que o faça.
No exemplo da figura 4, o jogador tem a opção se fazer ambos os movimentos.

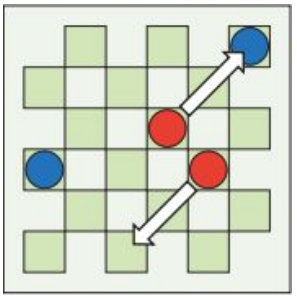

Figura 4: movimentos possíveis

Vencedor

Vence o jogo, o último jogador com peças suas no tabuleiro.

Lógica do jogo

1. Representação do tabuleiro

Estado Inicial

```
initialBoard([
        [0, 2, 0, 2, 0, 2, 0, 2, 0, 2],
        [3, 0, 3, 0, 3, 0, 3, 0, 3, 0],
        [0 ,2, 0, 2, 0, 2, 0, 2, 0, 2],
        [3, 0, 3, 0, 3, 0, 3, 0, 3, 0],
        [0, 2, 0, 2, 0, 2, 0, 2, 0, 2],
        [3, 0, 3, 0, 3, 0, 3, 0, 3, 0],
        [0, 2, 0, 2, 0, 2, 0, 2, 0, 2],
        [3, 0, 3, 0, 3, 0, 3, 0, 3, 0],
        [0, 2, 0, 2, 0, 2, 0, 2, 0, 2],
        [3, 0, 3, 0, 3, 0, 3, 0, 3, 0]
        ]).
```

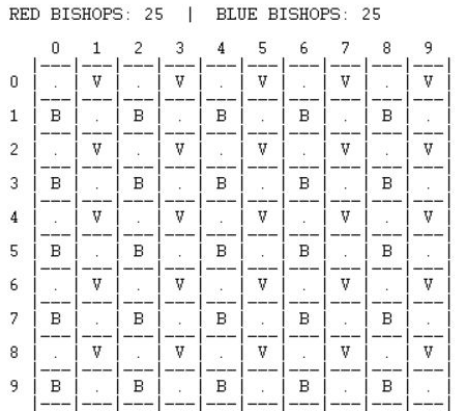

Figura 5: tabuleiro em estado inicial

Estado intermédio

```
mediumBoard([
        [0, 1, 0, 2, 0, 1, 0, 2, 0, 3],
        [3, 0, 1, 0, 2, 0, 1, 0, 1, 0],
        [0, 2, 0, 3, 0, 1, 0, 3, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 2, 0],
        [0, 2, 0, 2, 0, 3, 0, 1, 0, 3],
        [2, 0, 1, 0, 1, 0, 1, 0, 1, 0],
        [0, 3, 0, 2, 0, 3, 0, 3, 0, 1],
        [1, 0, 1, 0, 3, 0, 1, 0, 2, 0],
        [0, 1, 0, 3, 0, 2, 0, 3, 0, 3],
        [2, 0, 1, 0, 1, 0, 3, 0, 1, 0]
        ]).
```

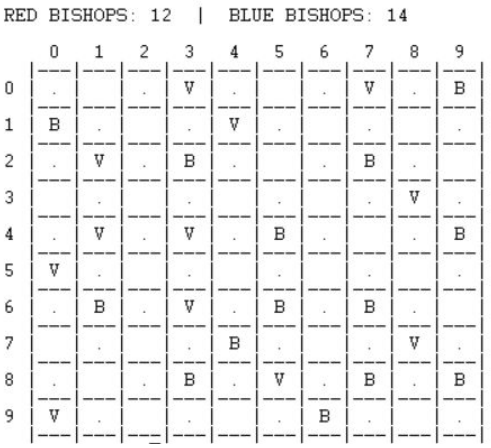

Figura 6: tabuleiro em estado intermédio

Estado final

```
finalBoard([
        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 3, 0, 1, 0, 1, 0],
        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
        [0, 1, 0, 3 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
        [0, 1, 0, 1, 0, 1, 0, 3, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
        ]).
```

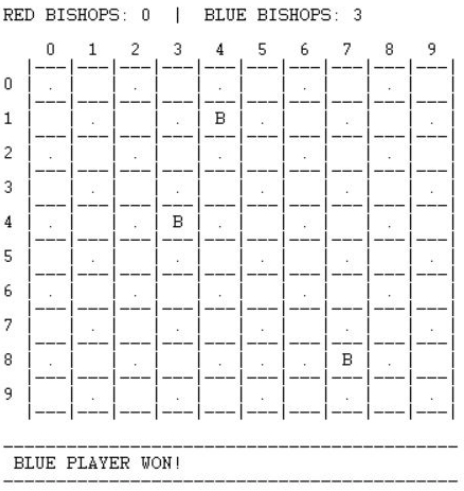

Figura 7: tabuleiro em estado final

2. Representação do jogo

O predicado principal na representação do jogo é *displayGame*, implementado com recursividade. Este é chamado através de *start.* Este predicado recebe o estado do jogo, o qual é constituído por uma lista de listas que contém, respetivamente, o **tabuleiro**, o **jogador** atual (*"RED"* ou *"BLUE"*) e as peças restantes.

Como ilustrado na *figura 8,* os componentes anteriormente referidos são impressos na consola.

O tabuleiro, em si, é impresso através de recursividade. Em *displayGame* é chamado o predicado *printTable*. Este imprime cada linha, recursivamente, chamando *printList* com a "head" do tabuleiro e a si próprio com a "tail", que por sua vez, imprime cada "casa" através do predicado *printCell.*

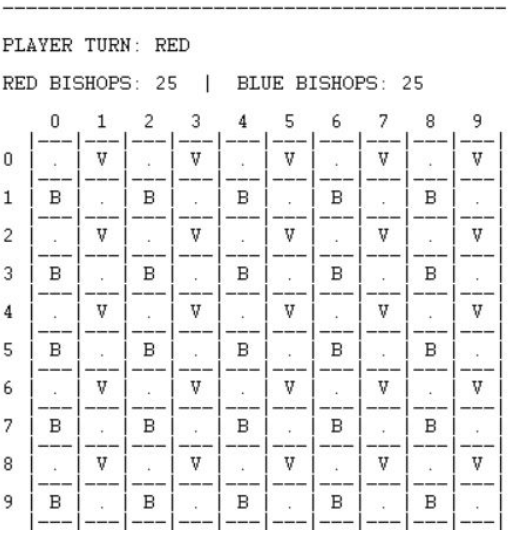

Figura 8: resultado de displayGame

Legenda:
[0] espaço ocupado [1] espaço livre [2] bispo vermelho [3] bispo azul
[B] bispo azul [V] bispo vermelho

3. Lista de jogadas válidas

- **valid_moves(+Board, +Player, -ListOfMoves)**

O predicado utilizado para obter todas as possíveis jogadas é o **valid_moves**. Este predicado implementa um *findall* que procura todas os pontos, no formato *[FromX,FromY,ToX,ToY]*, que representam jogadas válidas, isto é, que verificam todas as condições de uma **valid_play**.

```
valid_moves(B, Player, ListMoves):-
    findall([FromX,FromY,ToX,ToY],valid_play(B,Player,point(FromX,FromY),point(ToX,ToY)),ListMoves).
```

- **valid_play(+B, +Player, +PFrom, +PTo)**

Este predicado reúne as condições que definem uma jogada válida.

Inicialmente, é verificado se o *input* introduzido é válido e se a peça que o utilizador pretende mover é sua.

```
valid_play(B, Player, PFrom, PTo):-
    between_board(PFrom),
    between_board(PTo),
    check_player_piece(B, Player,PFrom),
    valid_kill(B, Player,PFrom,PTo);
    valid_engage(B, Player, PFrom,PTo).
```

Como referido anteriormente, uma jogada pode ser ***kill*** ou ***engage***. O predicado verifica se é uma jogada válida do tipo *kill* e, caso não seja, verifica se é uma jogada válida do tipo *engage*. Esta verificação é feita através dos seguintes predicados:

- **valid_kill(+B, +Player, +PFrom, +PTo)**

  Este predicado testa se uma jogada é uma *kill*, começando por ver se o ponto de destino tem uma peça do jogador adversário (**check_destiny_target**). De seguida, verifica se os pontos estão posicionados diagonalmente através do cálculo dos declives (**is_Diagonal**) e acaba por testar se a diagonal existente está vazia com auxílio do predicado **empty_spaces**.

```
valid_kill(B, Player, PFrom, PTo):-
    check_destiny_target(B,Player, PTo),
    is_diagonal(PFrom,PTo),
    empty_spaces(B,PFrom,PTo).
```

- **valid_engage(+B, +Player, +PFrom, +PTo)**

Este é utilizado para testar se uma jogada constitui um *engage*. Existem algumas regras comuns ao *valid_kill*, nomeadamente *is_diagonal* e *empty_spaces***.** No entanto, existem também algumas diferentes como **check_destiny_empty** que testa se o ponto de destino da jogada está desocupado e **valid_kill** que vai verificar se existe alguma *kill* possível a partir do ponto de destino original *PTo* para qualquer outro ponto.

Para além destas regras, é também chamado o predicado *valid_kill* de modo a que se verifique se essa peça a jogar não pode efectuar uma jogada kill.

```
valid_engage(B, Player, PFrom, PTo):-
    check_player_piece(B,Player,PFrom),
    findall([PFrom], valid_kill(B,Player,PFrom, point(_X,_Y)), ListOfKills),
    length(ListOfKills, 0),
    check_destiny_empty(B,PTo),
    is_diagonal(PFrom,PTo),
    empty_spaces(B,PFrom,PTo),
    valid_kill(B,Player,PTo,point(_X,_Y)).
```

4. Execução de jogadas

- **update(state(board(+B, +PiecesP1, +PiecesP2),+Player), state(board(-NewB, -NewPiecesP1, -NewPiecesP2), -NewPlayer), +TypePlayer, +Level)**

Predicado chamado no ciclo de jogo e que vai ser responsável por alterar o estado de jogo.

Este executa **get_move** que, dependendo do tipo de jogador (*TypePlayer*: utilizador - *human* ou inteligência artificial - *bot*), retorna o *move*.

De seguida, a jogada vai ser efectuada com auxílio dos predicado **move**:

- **move(move(point(FromX,FromY),point(ToX,ToY)), board(B,PiecesP1,PiecesP2), board(NewBoard,NewPiecesP1,NewPiecesP2), Player)**

  O predicado **move** utiliza **get_piece** para saber qual é a peça a mover; **replace_in_table** para criar o novo tabuleiro de jogo com a jogada efetuada e, por fim, verifica se o movimento efectuado foi kill com objetivo de decrementar o número de peças *(NewPiecesP1, NewPiecesP2*).

Por fim, o turno é passado para o próximo jogador em **change_player** e este processo é repetido.

```
update(state(board(B, PiecesP1, PiecesP2),Player), state(board(NewB, NewPiecesP1, NewPiecesP2), NewPlayer), TypePlayer, Level):-
    get_move(state(board(B, PiecesP1, PiecesP2),Player),point(FromX,FromY), point(ToX,ToY),TypePlayer, Level),
    move(move(point(FromX,FromY), point(ToX,ToY)), board(B,PiecesP1,PiecesP2), board(NewB,NewPiecesP1,NewPiecesP2), Player),
    change_player(Player,NewPlayer).
```

De seguida, encontra-se um diagrama que representa, de uma forma simples, o ciclo do jogo:

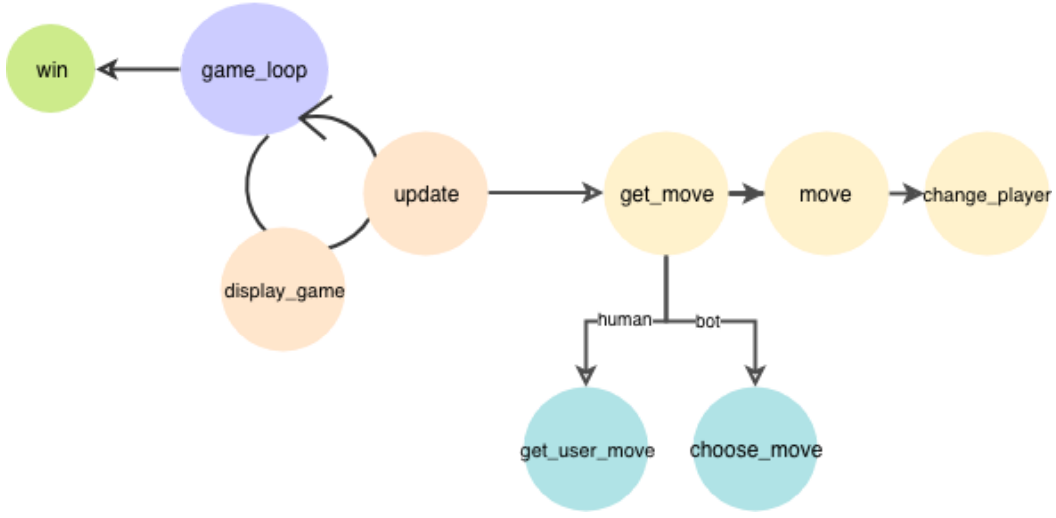

Figura 9: ciclo de jogo

5. Final de jogo

- **game_over(board(+B, +PiecesP1, +PiecesP2), -Winner)**

Este predicado é utilizado para verificar se já se deu o final da partida e, se sim, colocar o vencedor no argumento *Winner.*

Aqui é testado se o valor de peças de algum dos jogadores é igual a 0. Em caso afirmativo, é declarado o vencedor e o jogo termina. Caso contrário, quando o número de peças de ambos os jogadores for maior que zero, o ciclo de jogo continua.

```
game_over(board(_B,PiecesP1,_PiecesP2), Winner):-
    PiecesP1 is 0, Winner is 2.

game_over(board(_B,_PiecesP1,PiecesP2), Winner):-
    PiecesP2 is 0, Winner is 1.

game_over(board(_B,_PiecesP1, _PiecesP2), 0).
```

6. Avaliação do Tabuleiro

- **value(state(board(+B ,+PiecesP1, +PiecesP2), +Player), -Value)**

value é o predicado que mediante um estado de jogo vai atribuir um valor ao tabuleiro.

O critério utilizado pelo nosso jogo tem em conta a diferença do número de peças do jogador com as do adversário depois de uma *valid_play (***value_kills***)*, dando assim mais valor às jogadas kill que deem um melhor balanço nesse algoritmo. Para além disso, o nosso critério contabiliza o total de jogadas kill do jogador oposto no seguimento de cada uma das jogadas válidas estudadas e valoriza quanto menos opções o adversário tiver ao seu dispor (**value_killable**).

```
value(state(board(B,PiecesP1,PiecesP2),Player),Value):-
    value_kills(PiecesP1, PiecesP2, Player, Value_kills),
    value_killable(B, Player, Value_killable),
    Value is Value_kills - Value_killable.
```

- **value_kills(+PiecesP1, +PiecesP2, +Player, -Value)**

  Calcula a diferença entre o número de peças de um jogador e do outro e instância Value com esse valor após aplicar um fator que visa manter o impacto deste valor ao longo de todo o jogo.

- **value_killable(+B, +Player, -Value)**

  A partir de um ***findall***, este predicado calcula a quantidade de jogadas *kill* que o adversário vai ter e atribui esse valor a Value.

Após estas regras, o *value* do predicado pode ser sumarizado na seguinte fórmula:

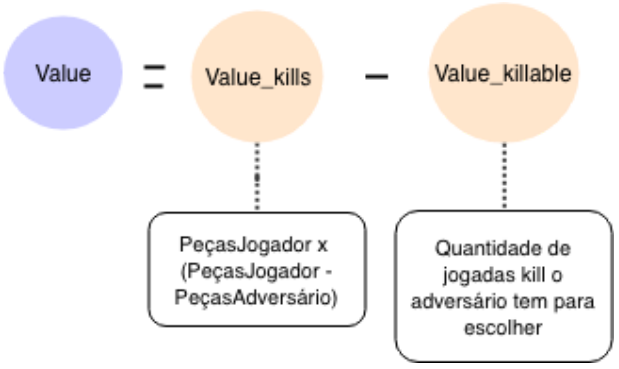

Figura 10: fórmula de cálculo do valor de um tabuleiro

7. Jogada do Computador

- **choose_move(+Board, +Level, -Move)**

O predicado **choose_move** 'redireciona' para os predicados **ai_easy** ou **ai_medium** de acordo com o nível do jogo (*Level*).

Ambos os predicados '*ai*' representam jogadas do computador, diferindo da especificidade de regras que seguem para escolher uma jogada.

```
choose_move(state(board(B, _PiecesP1,_PiecesP2),Player), Level, move(point(FromX,FromY),point(ToX,ToY))):-
    Level is 1, ai_easy(B, Player,move(point(FromX,FromY),point(ToX,ToY))).

choose_move(state(board(B, PiecesP1, PiecesP2),Player), Level, move(point(FromX,FromY),point(ToX,ToY))):-
    Level is 2, ai_medium(state(board(B,PiecesP1,PiecesP2),Player),move(FromX,FromY,ToX,ToY)).
```

- **ai_easy(B, Player, move(point(FromX,FromY),point(ToX,ToY)))**

  Neste nível, as jogadas escolhidas pelo computador são apenas escolhas aleatórias de entre as jogadas possíveis (*valid_ moves*).

```
ai_easy(B, Player, move(point(FromX,FromY),point(ToX,ToY))):-
    valid_moves(B, Player, ListMoves),
    length(ListMoves, _ListSize),
    generate_random_num(0,_ListSize,RandomNum),
    nth0(RandomNum, ListMoves, Movement),
    list_to_move(Movement,move(point(FromX,FromY),point(ToX,ToY))).
```

- **ai_medium(state(board(B,PiecesP1,PiecesP2),Player),Move)**

  No nível médio, é utilizado um **findall** que simula jogadas e atribui um valor ao tabuleiro (*value*), como explicado anteriormente na secção anterior.

  Após ser criada a lista com todos os *values* e *moves* possíveis, é chamado o predicado **choose_best_move** que, recursivamente, itera a lista de *moves,* guardando o maior *value* e o seu respectivo *move*. Em caso de empate do v*alue*, é efetuada uma "*coin toss*" que introduz um fator aleatório.

```
ai_medium(state(board(B,PiecesP1,PiecesP2),Player),Move):-
    findall(Value-FromX-FromY-ToX-ToY,(valid_play(B, Player, point(FromX,FromY),point(ToX,ToY)),
    move(move(point(FromX, FromY),point(ToX,ToY)),board(B,PiecesP1,PiecesP2),board(NewBoard,NewPiecesP1,NewPiecesP2),Player),
    change_player(Player,NewPlayer),
    value(state(board(NewBoard,NewPiecesP1,NewPiecesP2),NewPlayer),Value)),ListValues),
    choose_best_move(ListValues, Move,-100,move(0,0,0,0)).
```

De seguida, encontra-se representado um diagrama que demonstra a "bifurcação" do predicado **choose_move**:

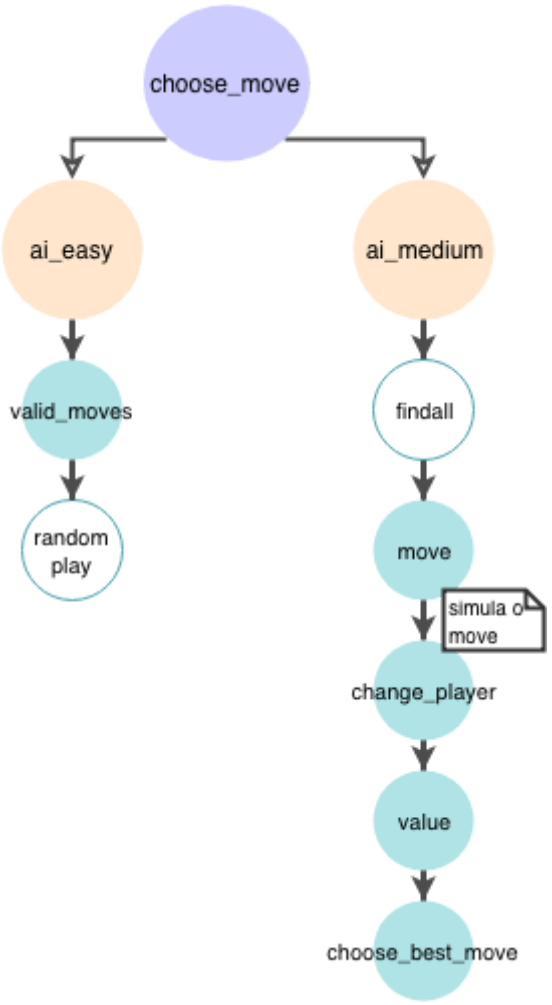

Figura 11: diagrama da jogada do computador

Conclusão

O projeto teve como objetivo fundamental pôr em prática os conhecimentos adquiridos nas aulas da unidade curricular Programação em Lógica.

Durante o desenvolvimento do trabalho foram encontradas algumas dificuldades como reestruturar o nosso raciocínio usual aplicado a outras linguagens de programação. Consideramos que superamos as mesmas e atingimos um melhor conhecimento de como programar em *Prolog*.

Em retrospetiva ao trabalho desenvolvido, vemos a possibilidade de melhoria na parte relativa à inteligência artificial, podendo ter aumentado a profundidade da desta pois permitiria uma *AI* mais robusta e mais eficiente na escolha de jogadas.

Referências

[1] “IG Game Center” http://www.iggamecenter.com/info/en/madbishops.html
[2] “Mark Steere Games” http://www.marksteeregames.com/Mad_Bishops_rules.pdf
[3] “SWI Prolog” http://www.swi-prolog.org/