Relatório Final do 1º Trabalho Prático de Programação em Lógica

Ricardo Jorge de Sousa Teixeira
ei08040@fe.up.pt

João Nuno Santos de Gusmão Guedes
ei07043@fe.up.pt

Universidade do Porto
FEUP Faculdade de Engenharia

Resumo

O trabalho consiste no desenvolvimento do jogo de tabuleiro Absorção recorrendo ao Prolog como linguagem de implementação. O projecto tem ainda um módulo de visualização gráfica 3D a ser implementado em linguagem C++. A aplicação final deverá permitir três modos de jogo distintos: Humano/Humano, Humano/Computador e Computador/Computador. Serão incluídos diversos níveis de jogo para o computador.

Conteúdo

1 Introdução

Este trabalho tem como objectivo adquirir e consolidar conhecimentos na linguagem Prolog, que apresenta um paradigma diferente daquele a que estamos acostumados.
Começaremos por implementar a funcionalidade de jogo Humano/Humano recorrendo a regras de lógica e posteriormente recorreremos a algoritmos específicos de inteligência artificial e teoria dos jogos para implementar o modo Humano/Computador e Computador/Computador.
Escolhemos este jogo porque, apesar de se basear num jogo de tabuleiro de xadrez convencional, nos atraíram algumas particularidades do jogo, tais como poderem-se empilhar e capturar peças do adversário.
Pareceu-nos também que seria um jogo relativamente simples de implementar/aprender mas que, no entanto, oferecia uma estratégia potencialmente complexa:

- cada um dos jogadores começa com 18 peças de duas faces, em pilhas de 6 em que cada uma das faces representa o jogador;
- cada jogada subdivide-se em 3 passos, nos quais o objectivo fundamental é progredir no tabuleiro e capturar as peças do adversário;
- o jogo termina quando um dos jogadores capturar as peças do adversário até que lhe sobrem apenas 3 peças, ou então o impeça de empilhar/separar peças;

2 Descrição do Problema

2.1 Peças

36 Peças com dois lados diferentes, cada uma com o valor de 1 ponto.

2.2 Tabuleiro

Tabuleiro com 8x8 casas. Pode ser usado um tabuleiro de xadrez mas não é necessário que o tabuleiro tenha casas com cores alternadas.

2.3 Preparação

Os jogadores colocam-se em dois lados opostos do tabuleiro. Cada jogador escolhe uma cor e começa por colocar 6 pilhas de 3 peças com a sua cor virada para cima na linha mais proxima de si. Estas peças estão sob o seu controlo.

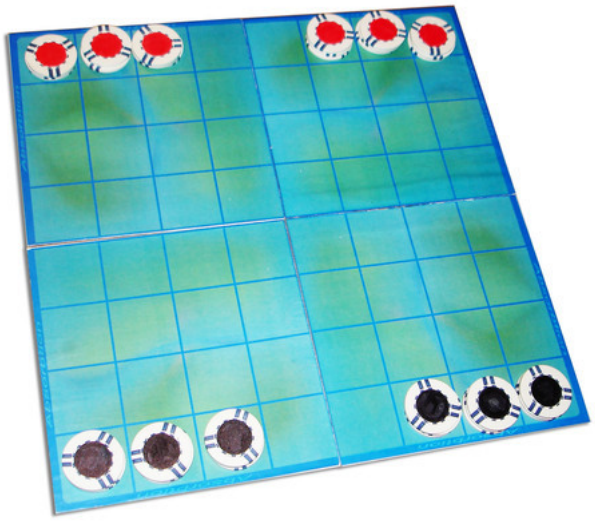


Figura 1: Disposição inicial das peças.

2.4 Movimento

As pilhas podem se mover em todas as direcções - ortogonalmente (cima, baixo, esquerda, direita) ou diagonalmente.

- uma pilha Pequena (1 peça) pode mover-se uma ou duas casas em linha recta;
- uma pilha Média (2 peças) pode mover-se uma casa;
- uma pilha Grande (3 peças) não se pode mover;

Nenhuma peça se pode mover através de outra peça ou pilha.

2.5 Jogadas

Joga-se alternadamente. Cada vez que um jogador joga é chamado um turno. No seu turno o jogador tem as seguintes opções (por ordem):

1. separar ou Juntar uma vez (obrigatório);
2. mover uma pilha não envolvida na Separação ou na Junção (opcional);
3. absorver (se possível);

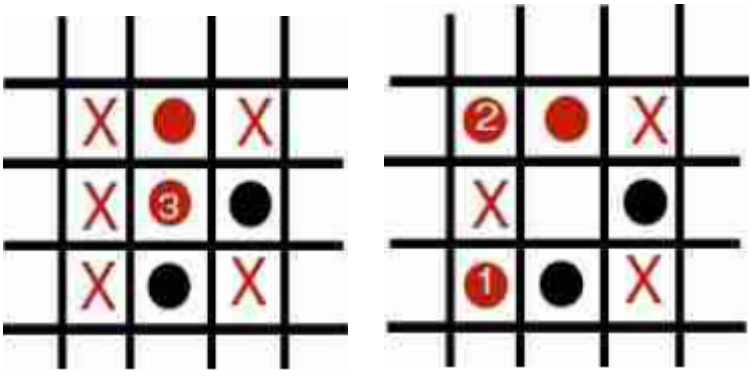
O jogador deve Separar ou Juntar uma vez por turno. Se o jogador não conseguir Separar ou Juntar no seu turno, perde o jogo.

2.5.1 Separar

Se uma pilha é separada, o jogador remove a pilha e coloca as suas peças em pilhas pequenas em posições adjacentes (ortogonais ou diagonais) à posição original.

- as pilhas Grandes(3) podem-se separar em duas pilhas Médias e uma Pequena(2,1) ou em três pilhas Pequenas(1,1,1);
- as pilhas Médias(2) podem-se separar em duas pilhas Pequenas(1,1);
- as pilhas Pequenas(1) não se podem separar;

Se alguma das pilhas resultantes for Pequena, pode-se mover uma ou duas casas imediatamente mas não pode voltar ao espaço ocupado pela pilha original. Se não existirem, para todas as peças resultantes, casas livres suficientes adjacentes à pilha separada, essa pilha não pode ser separada.



(a) Possibilidades (b) Separação

Figura 2: Processo de separação.

2.5.2 Juntar

Para juntar peças o jogador deve seleccionar pilhas completas adjacentes e combiná-las numa só pilha (máx. 3 peças). A nova pilha é colocada no tabuleiro numa das anteriores localizações das peças Juntas. Nenhuma pilha pode ter mais do que 3 peças de altura.

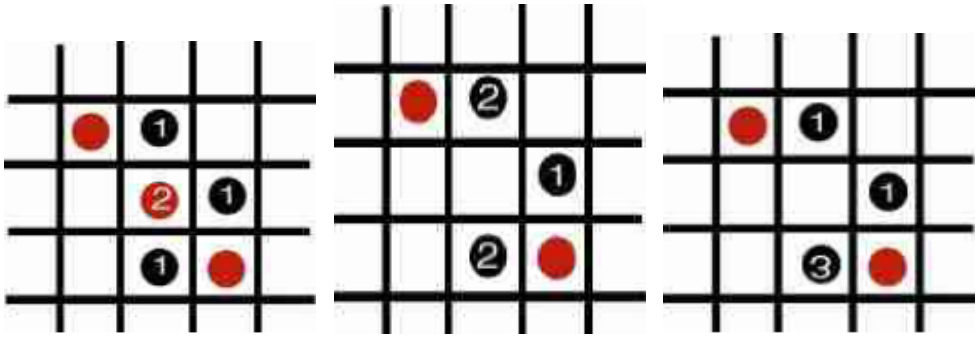
2.5.3 Mover

No seu turno um jogador pode mover uma pilha não envolvida numa Separação ou Junção.

2.5.4 Absorção

A Absorção ocorre no final de cada turno. Uma pilha (ou pilhas) é obrigatoriamente absorvida quando está rodeada ORTOGONALMENTE por oponentes maiores que o seu valor.

A pilha absorvida é removida do tabuleiro e as suas peças são viradas ao contrário (passam a ser do jogador que as absorveu). Estas peças são então adicionadas às pilhas captoras.
Exemplo:
Uma pilha Média(2) é rodeada por três pilhas Pequenas (valor total 3) logo a pilha Média é removida e virada ao contrário (Fig. 3a).
O jogador que Absorveu tem 2 peças para usar. Esse jogador pode adicionar uma peça a duas das pilhas Pequenas (1+1 e 1+1) resultando em duas pilhas Médias (2+2) (Fig. 3b), ou então pode adicionar ambas as peças a uma das pilhas Pequenas (1+1+1) resultando numa pilha Grande (Fig. 3c).



(a) Pilha rodeada (b) Uma hipótese (c) Outra hipótese

Figura 3: Absorção.

Uma pilha que não pode ser completamente Absorvida pelas pilhas captoras não pode ser Absorvida.
Exemplo:
Uma pilha Grande(3) rodeada por duas pilhas Médias (2 e 2) não poderia ser absorvida visto que não se pode empilhar todas as 3 peças Absorvidas nas pilhas captoras.

Qualquer pilha pode estar envolvida em mais do que uma Absorção.
Nenhuma pilha pode ter mais do que 3 peças de altura.

2.6 Vitória

O jogador que conseguir fazer com que o seu oponente tenha 3 ou menos pontos em seu controlo ganha. Se um jogador não conseguir Separar ou Juntar é obrigado a desistir e dar a vitória ao seu opositor.

3 Representação do Estado do Jogo

O jogo desenrola-se num tabuleiro 8x8, implementado em Prolog por uma lista de listas.

Cada item do tabuleiro é guardado no formato <Jogador>-<Número de Peças>. Por exemplo:

- 0-0 representa uma casa vazia.
- 1-2 representa uma pilha Média do 1º jogador.

Inicialmente o tabuleiro irá conter as 18 peças de ambos os jogadores, colocadas nas posições respectivas:

Código fonte 1 Representação do estado inicial.

```
tabuleiro_inicial(
[
[1-3,1-3,1-3,0-0,0-0,1-3,1-3,1-3],
[0-0,0-0,0-0,0-0,0-0,0-0,0-0,0-0],
[0-0,0-0,0-0,0-0,0-0,0-0,0-0,0-0],
[0-0,0-0,0-0,0-0,0-0,0-0,0-0,0-0],
[0-0,0-0,0-0,0-0,0-0,0-0,0-0,0-0],
[0-0,0-0,0-0,0-0,0-0,0-0,0-0,0-0],
[0-0,0-0,0-0,0-0,0-0,0-0,0-0,0-0],
[2-3,2-3,2-3,0-0,0-0,2-3,2-3,2-3]
]).
```

Um exemplo de um estado de jogo intermédio:

Código fonte 2 Representação de estado intermédio.

```
tabuleiro_exemplo(
[
[1-3,1-3,0-0,0-0,0-0,1-1,1-3,1-3],
[0-0,0-0,0-0,0-0,0-0,1-2,0-0,0-0],
[0-0,0-0,1-1,1-1,0-0,0-0,0-0,0-0],
[0-0,0-0,0-0,0-0,1-1,0-0,0-0,0-0],
[0-0,0-0,0-0,0-0,0-0,2-1,2-2,0-0],
[0-0,0-0,0-0,0-0,0-0,0-0,0-0,0-0],
[0-0,0-0,0-0,0-0,0-0,2-2,0-0,0-0],
[2-3,2-3,2-3,0-0,0-0,2-1,0-0,2-3]
]).
```

Um exemplo de um estado de jogo final em que a vitória foi do Jogador 2:

Código fonte 3 Representação de final de jogo.

```
tabuleiro_final(
[
[0-0,0-0,0-0,1-1,0-0,0-0,0-0,0-0],
[0-0,0-0,0-0,0-0,0-0,0-0,0-0,0-0],
[2-2,0-0,2-1,0-0,2-2,0-0,1-2,0-0],
[0-0,2-1,0-0,0-0,0-0,2-1,0-0,0-0],
[2-1,0-0,2-3,0-0,2-3,0-0,0-0,0-0],
[0-0,0-0,0-0,0-0,2-1,0-0,0-0,0-0],
[0-0,0-0,0-0,0-0,0-0,0-0,0-0,0-0],
[2-3,2-3,2-3,0-0,0-0,2-3,2-3,2-3]
]).
```

4 Representação de um Movimento

Para representar movimentos tivemos antes de mais de escolher um sistema para identificar cada uma das casas do tabuleiro. Optámos por um sistema de duas coordenadas numéricas no formato *Linha-Coluna* em que Linha e Coluna são dois algarismos de 1 a 8 que identificam respectivamente o número da linha e coluna onde a casa se encontra. Quanto ao movimento em si, e tal como foi descrito anteriormente, cada turno é composto pelas seguintes opções:

- separar(OrigemLinha-OrigemColuna,[Destinos],TabEntrada,TabSaida,Jogador)
  Divide uma pilha, com uma pilha de origem e pilha(s) de destino. Exemplo:

  ```
  separar(1-3,[1-4,1-2,1-2],TabuleiroEntrada,TabuleiroSaida,Jogador)
  ```

  Pilha de 3 peças na posição (1,3) é dividida em peças para as posições (1,4), (1,2) e (1,2). Quando se repetem peças é porque se empilham na mesma posição.

- juntar([Origens],DestinoLinha-DestinoColuna,TabEntrada,TabSaida,Jogador)
  Junta peças de pilhas de origem numa pilha de destino. Neste caso as Origens são Pilhas e não Peças. Exemplo:

  ```
  juntar([1-4,1-2],1-3,TabuleiroEntrada,TabuleiroSaida,Jogador)
  ```

  As peças das Pilhas na posição (1,4) e (1,2) são juntas à pilha na posição (1,3).

- mover(OrigemLinha-OrigemColuna,Destino,TabEntrada,TabSaida,Jogador)
  Move pilha de uma posição de origem para posição de destino. Exemplo:

  `mover(1-3,1-5,TabuleiroEntrada,TabuleiroSaida,Jogador)`

  Move pilha da posição (1,3) para posição (1,5).

- absorcao([Origens],DestinoLinha-DestinoColuna,TabEntrada,TabSaida,Jogador)
  Semelhente a split(), excepto que visto que a pilha de origem será absorvida esta tem que ser obrigatoriamente do adversário. Exemplo:

  `absorcao([1-4,1-2,1-2],1-3,TabuleiroEntrada,TabuleiroSaida,Jogador)`

  Pilha de 3 peças na posição (1,3) é dividida - 1 peça vai para a posição (1,4) e 2 para (1,2).

5 Visualização do Tabuleiro

O tabuleiro tem 8x8 casas iguais e em cada casa podem estar 0,1,2 ou três peças. As casa vazias são representadas por espaços em branco. As peças do jogador 1 são representadas pela letra "X" e as do jogador 2 pela letra "O". Para representar uma pilha usamos conjuntos de letras repetidas, com o seguinte formato (Jogador 1 ou Jogador 2):

pilha Pequena "X" ou "O";

pilha Média "X X" ou "O O";

pilha Grande "XXX" ou "OOO";

A Figura 4 mostra um tabuleiro representando um jogo a decorrer.

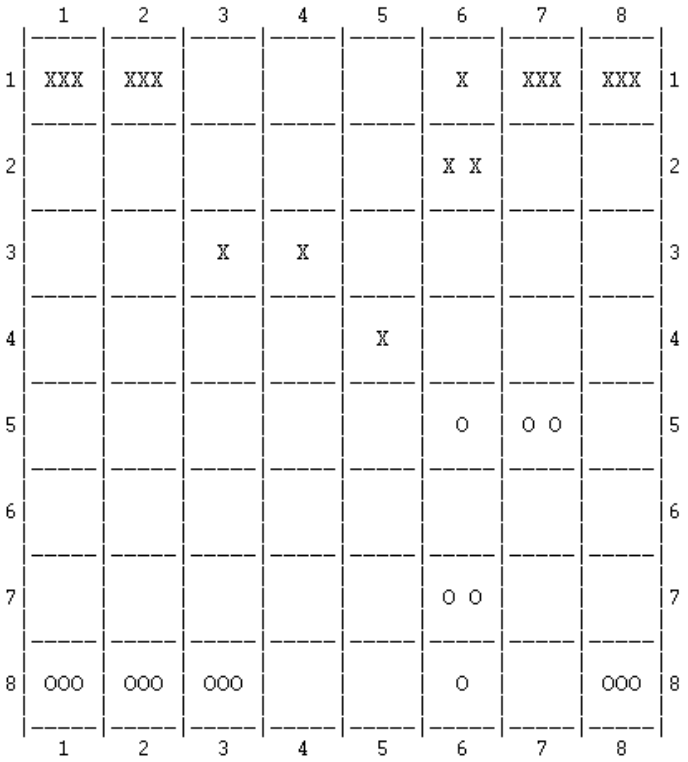


Figura 4: Jogo a decorrer.

Para imprimir o tabuleiro utilizamos o predicado *visualiza_estado(T)* em que T é o tabuleiro que queremos imprimir. Este predicado começa por imprimir o topo do tabuleiro com a numeração de cada coluna e chama o predicado *printll(N,T)*. Os argumentos para o predicado *printll(N,T)* são N - número da linha e T - tabuleiro a imprimir. O predicado *printll* itera então pela lista recursivamente, linha a linha, imprimindo para cada uma o tabuleiro e chamando o predicado *printl(L)* com a linha como argumento. Esse predicado vai então chamar o predicado *escreve(J-P)* para cada célula dessa linha, que vai desenhar a peça ou peças (P) do jogador (J). No final o predicado *visualiza_estado(T)* imprime as linhas finais do tabuleiro. Podemos ver em Código fonte 4 o código com os predicados implementados.

Código fonte 4 Predicados para imprimir um tabuleiro.

```
% Imprimir uma linha
printl([]). % criterio de paragem.
printl([A|R]):-
  write('| '),  escreve(A),  write(' '),  printl(R).
% Imprimir uma lista de linhas
printll(_,[]). %para quando lista vazia
printll(N,[Head|Tail]):-
  % parte de cima da linha
  write(' |     |     |     |     |     |     |     |     |'),nl,
  write(N), %escreve o numero da linha
  printl(Head),  write('|'),  write(N),  nl,
  % parte de baixo da linha
  write(' |     |     |     |     |     |     |     |     |'),nl,
  % separador depois da linha
  write(' |-----|-----|-----|-----|-----|-----|-----|-----|'),nl,
  N2 is N+1,
  printll(N2,Tail).
% Imprimir um estado de jogo
visualiza_estado(T):-
  nl,
  write('    1     2     3     4     5     6     7     8   '),nl,
  %primeiro separador
  write(' |-----|-----|-----|-----|-----|-----|-----|-----|'),nl,
  printll(1,T),
  write('    1     2     3     4     5     6     7     8   '),nl,
  nl.
% Imprimir cada um dos possiveis estados das casas
escreve(1-1):-write(' X ').
escreve(1-2):-write('X X').
escreve(1-3):-write('XXX').
escreve(2-1):-write(' O ').
escreve(2-2):-write('O O').
escreve(2-3):-write('OOO').
escreve(_-0):-write('   '). % Posicao vazia
escreve(___):-write('ERR'). % Erro
```

6 Observações sobre a complexidade do Jogo

Devido a complexidade do jogo e ao facto de ser composto por várias Jogadas por turno, sendo que algumas opcionais, tivemos muitas dificuldades em implementar o mesmo.
Essas dificuldades contribuíram para que não tivemos tempo para completar o trabalho.

7 Código implementado

Como já foi mencionado, o trabalho está incompleto, no entanto conseguimos implementar a modalidade CPUvsCPU, invocável através do predicado:

```
jogoCPUvsCPU.
```

Sendo que cada turno é composto por várias partes, são necessárias várias verificações por turno e cada uma das partes tem a sua verificação.
O que fazemos nessas verificações é uma sequência de refinamentos de listas, em que começamos por todas as casas possíveis e depois vamos, através de regras, filtrando os resultados. Usamos esses predicados de testes para obter as listas de todas as jogadas possíveis de forma a podermos depois usar aleatoriedade aquando da jogada CPUvsCPU. Passo a listar os mais importantes predicados usados nessas verificações:

- separarPossivel(Li-Col,ListaDeDestinos,Tabuleiro,Jogador).
- juntarPossivel(ListaDeOrigens,Destino,Tabuleiro,Jogador).
- moverPossivel(LinO-ColO,Destino,Tabuleiro,Jogador).
- absorcaoPossivel(PecaAbsorvida,ListaDeDestinos,Tabuleiro,Jogador).

foi com estes, e recorrendo extensivamente à instrução *findall* que implementamos toda a capacidade de listagem de jogadas possíveis.

Parte do código de jogada de um utilizador Humano também está implementado, sem no entanto estar essa opção completamente a funcionar.
Gostaríamos de salientar que todo o código que apresentamos funciona, e foi por nós testado em testes individuais, no entanto muito dele não é usado ainda no programa porque se destinava a partes não implementadas.

De notar que para simplificar a codificação das regras, não foi implementada a opção pode-se mover as casas pequenas resultantes imediatamente após a separação. Pelos mesmos motivos também não é verificado se as peças que movemos na fase Mover fizeram parte de uma separação ou junção.

8 Conclusões

Considerando a complexidade do jogo e analisando o trabalho que conseguimos fazer sentimos que deveríamos ter despendido mais tempo neste trabalho. Acreditamos no entanto, visto que conseguimos fazer o modo CPUvsCPU, que poderíamos também ter feito o código de jogo com Humanos. Ainda assim, é nossa opinião que para um jogo tão complexo, conseguir fazer o modo CPUvsCPU é satisfatório.

Referências

[1] Página da Disciplina, "http://paginas.fe.up.pt/~eol/LP/1011/index.htm"

[2] Listas em Prolog, "http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/"

[3] Jogo Absorption, "http://www.boardgamegeek.com/boardgame/63114/absorption/"

[4] Tutorial de Prolog, "http://hilltop.bradley.edu/~chris/prolog.html"