Epaminondas

Relatório Final

Universidade do Porto

Faculdade de Engenharia

FEUP

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo 13:
Hugo Ari Rodrigues Drumond - 201102900
João Alexandre Gonçalinho Loureiro - 200806067

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

10 de Novembro de 2013

Resumo

O jogo de tabuleiro que iremos desenvolver chama-se Epaminondas. As peças presentes no tabuleiro são todas iguais e movem-se como o Rei no xadrez, exceto quando se agrupam numa linha, coluna ou diagonal. Atualmente, o tabuleiro é 14x12. Um jogador ganha quando, na sua vez de jogar, tiver mais peças que o adversário na linha mais afastada de si. O jogo é iniciado através do predicado start. e as jogadas vão sendo lidas da seguinte forma, XS.YS.XD.YD. Existem inúmeros predicados no nosso programa mais o mais importante é o play que funciona com um ciclo infinito saindo só quando estivermos numa condição de fim de jogo, tal é verificado no continueGame(N). No inicio de play temos umas regras básicas que nos avalião XS,YS,XD,YD, logo de seguida temos o atualizarEstado cujo objetivo é verificar se as regras são válidas através de canimovethere, depois apaga-se a Phalanx se existir um inimigo na linha de deslocamento, move-se a nossa phalanx para o novo sítio e finalmente mudamos a cor do jogador que deve jogar. Caso as regras falhem é pedido que o jogador jogue outra vez, como canimovethere falha, atualizarEtado falha, que faz com que play falhe. O prolog irá então procurar por uma regra com a mesma cabeça, essa cabeça irá fazer com que seja repetido todo o processo de leitura, etc. Trata-se do 1º. trabalho de grupo, da disciplina de Programação em Lógica, realizado pelo Grupo 13.

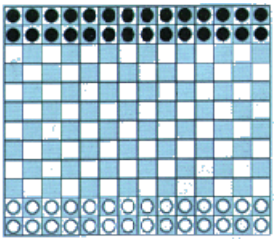
1 Introdução

O fim deste trabalho é criar um jogo de tabuleiro chamado Epaminondas com base na matéria exposta nas aulas teóricas e téorico-práticas. Este trabalho é interessante porque obriga-nos a pensar de um ponto de vista puramente lógico, isto é, cria-se uma base de dados lógica através de cláusulas, sem controlo de fluxo (explícito), e depois procuramos por possíveis soluções. Este paradigma de programação chama-se programação declarativa. Nos pontos 2, 3 e 4.1 é apresentado o esqueleto do nosso programa e nas restantes vamos um pouco mais além do óbvio.

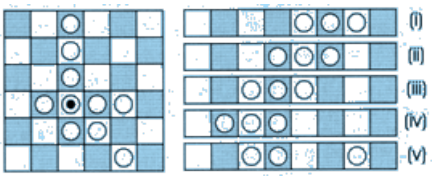
2 O Jogo Epaminondas

Este jogo, inicialmente chamado *Crossings* e jogado num tabuleiro 8x8, foi descrito pela primeira vez no Livro *A Gamut of Games* em 1969. Depois da publicação deste livro, Bob Abbott, reconfigurou o tabuleiro de 8x8 para 14x12, de modo a tornar o jogo nas diagonais mais importante, e renomeou o nome para Epaminondas como homenagem ao general TheBan que inventou a phalanx, uma formação de guerra usada em 371 a.c para derrotar o exército espartano.[1, 2]

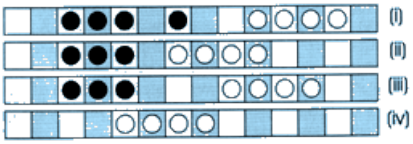
Neste jogo, são posicionadas 28 peças iguais em cada lado maior do tabuleiro, com cores diferentes. São as brancas a iniciar o jogo, e depois joga-se alternadamente.[2]



Os movimentos de cada peça são iguais ao do Rei do xadrez, pode andar uma casa em qualquer direção. Podem ser criados inúmeros grupos de peças, phalanxes, estas só se podem deslocar um número igual ou menor ao número de peças do grupo, na direcção da linha formada pelo grupo. Uma peça pode fazer parte de várias phalanxes. Uma phalanx pode ser dividida, mas só pode andar um número de casas igual ou menor ao numero de elementos da phalanx que se irá mover.[2]



Não é permitido passar um jogada, em cada jogada é obrigatório mover uma phalanx ou uma peça. Todas as peças têm de estar contidas no tabuleiro. Não podem haver peças sobrepostas ou na mesma posição, exceto quando a cabeça de uma phalanx captura outra peça ou phalanx.



Para capturar é necessário atacar com um grupo com mais elementos que o adversário na linha de ataque. A peça/phalanx capturada é removida do tabuleiro para sempre. Um jogador ganha quando, na sua vez de jogar, tiver mais peças que o adversário na linha mais afastada de si.[2]

3 Arquitetura do Sistema

O jogo é iniciado com um predicado chamado start: mostra o tabuleiro inicial; lê as coordenadas através de READ\1 e chama o predicado recursivo play(E,XS,YS,XD,YD). Este predicado faz o seguinte: na parte inicial testa os valores XS,YS,XD,YD de modo a ver se são válidos; caso sejam chama atualizarEstado(E,XS,YS,XD,YD,N); caso este objetivo seja completado, é mostrado o novo tabuleiro; logo depois é visto se estamos em situação de fim de jogo, se estivermos falha(tem cut atrás); são lidos os novos valores de XS,YS,XD,YD e é chamado play com os novos valores.

A função de atualizarEstado(E,XS,YS,XD,YD,N) é "corrigir"os valores de XS,YS,XD,YD visto que são mostrados de forma diferente; verificar se a jogada é válida através de canimovethere(E,XS1,YS1,XD1,YD1); se sim, mudarestado(E,XS1,YS1,XD1,YD1,N) e depois mudar a vez.

Na verificação das regras canimovethere(E,XS,YS,XD,YD): chama regras(E,XS,YS,XD,YD) e caso esta falhe é chamado badMoveFriend(E) cujo objetivo é pedir novamente que se jogue. A cláusula regras(E,XS,YS,XD,YD) contém 3 regras que verificam: se é a minha vez; se existe sobreposicao e se se pode mover uma phalanx(uma peça é um phalanx com um só elemento) ou se se pode capturar uma peça.

Foram construídas várias regras úteis: getSizeTilColour('tipodelinha',); headPhalanx('tipodelinha',); checkColour(,); changechar(,)r;, findEnemyHead('tipodelinha',); movePhalanxUP(,); movePhalanxLEFT(,).Cuja a finalidade é óbvia.

4 Lógica do Jogo

4.1 Representação do Estado do Jogo

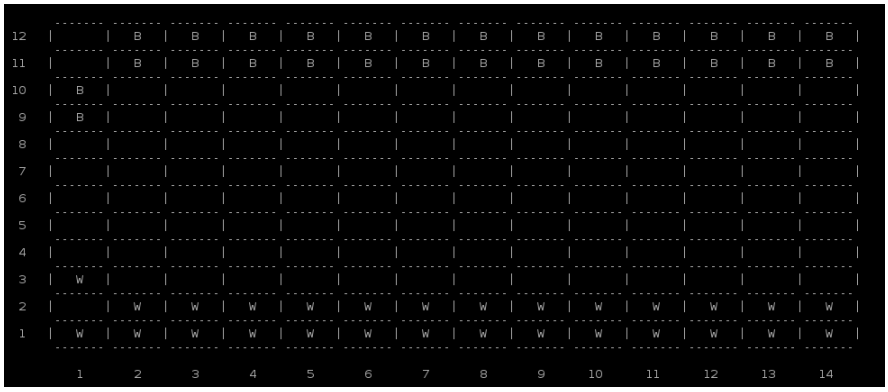
Iremos criar um estado inicial, uma lista de listas da forma:
[[2,2,2,2,2,2,2,2,2,2,2,2,2,2],[2,2,2,2,2,2,2,2,2,2,2,2,2,2],[0,0,0,0,0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,0,0,0,0,0],[1,1,1,1,1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,1,1,1,1,1,1]]

Através da regra de interface play iremos chamar as size e atualizarestado que testa as regras do jogo e atualiza o estado, caso tal respeite as regras do jogo. A regra atualizarestado chama pelas cabeças canimovethere, mudarestado e turnchange. Se a regra canimovethere não corresponder isso implica o falhanço do "if"em prolog e portanto o estado não é alterado nem a vez, sendo pedida outra vez um movimento. Se o objetivo canimovethere for bem sucedido, então podemos proceder à alteração do estado através da regra mudarestado. Restando mudar a vez do jogador, através de por exemplo de turnchange(C) (através de retract e asserta). Sendo este ciclo repetido novamente,até chegarmos a uma situação de fim de jogo.

```
109 start :-
110   estadoInicial(E),
111   show(E),
112   read(XS), read(YS), read(XD), read(YD), %le desta forma XS.YS.XD.YD.
113   play(E, XS, YS, XD, YD).
114 start :- !, 1=1.
115
116 play( E, XS, YS, XD, YD) :-
117   xDIFy(XS,YS,XD,YD),
118   size(XS,YS),
119   size(XD,YD),
120   atualizarEstado(E,XS,YS,XD,YD,N),
121   show(N),
122   !,continueGame(N),
123   read(XS1), read(YS1), read(XD1), read(YD1), % le desta forma XS.YS.XD.YD.
124   play( N, XS1, YS1, XD1, YD1).
125 %If sizes are not allowed retry.
126 play(E,_,_,_,_):-badMoveFriend(E).
127
128 xDIFy(XS,YS,XD,YD) :-
129   XS =\= XD; YS =\= YD.
130
131 atualizarEstado( E, XS, YS, XD, YD, N) :-
132   XS1 is XS - 1,
133   YS1 is 12 - YS, % Porque na verdade o que aparece no tabuleiro tem o índice trocado.
134   XD1 is XD - 1,
135   YD1 is 12 - YD1,
136   canimovethere(E,XS1,YS1,XD1,YD1,PhalanxSizeAdversary,XN,YN),
137   deletePhalanx(E,XS1,YS1,XD1,YD1,PhalanxSizeAdversary,XN,YN,E1),
138   changeState(E1,XS1,YS1,XD1,YD1,N),
139   turnchange(C).
140 atualizarEstado(E,_,_,_,_,_):-badMoveFriend(E).
```

O estado vai sendo atualizado consoante as jogadas de cada jogador, o 0 significa que não há nenhuma peça nessa posição. O 1 representa uma peça branca e o 2 uma preta. Quando se imprimi o 1 passa a W e o 2 a B.

Um estado com posições intermédias,



[[0,2,2,2,2,2,2,2,2,2,2,2,2,2],[0,2,2,2,2,2,2,2,2,2,2,2,2,2],[2,0,0,0,0,0,0,0,0,0,0,0,0,0],
[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],[0,0,0,0,0,0,0,0,0,0,0,0,0,0],
[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],
[1,0,0,0,0,0,0,0,0,0,0,0,0,0],[0,1,1,1,1,1,1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1,1,1,1,1,1,1]]

Um estado com posições finais, qualquer configuração em que existam mais peças do adversário na minha linha mais recuada, na vez dele jogar. E vice-versa.

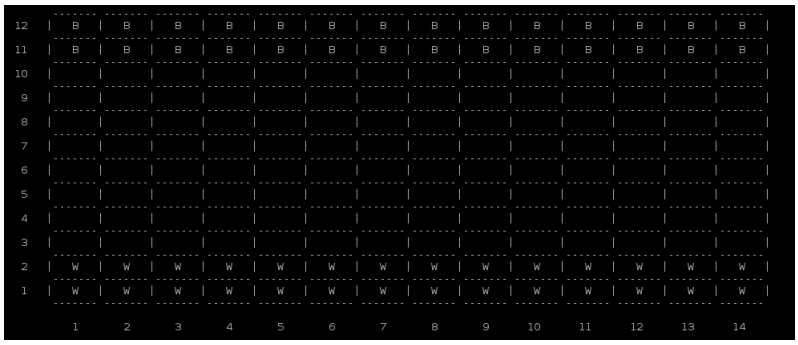
4.2 Visualização do Tabuleiro

De modo a imprimir o estado atual do tabuleiro, tem-se de construir um predicado recursivo que escreva o conteúdo de cada linha na consola e depois dê um break line. Tal pode ser feito através das seguintes cláusulas:

```
sepforprint([],_,_,_).
sepforprint( [H|T], _totalVerticalLines, _totalHorizontalLines, _currentHorizontal):-
  write('    '),
  numbers(vert,_currentHorizontal),
  _currentHorizontal1 is _currentHorizontal - 1,
  printline(H),
  write('   |'),
  nl,
  write('     '),
  write('   '),
  write('  '),
  lineseparator(_totalVerticalLines),
  nl,
  sepforprint(T,_totalVerticalLines,_totalHorizontalLines,_currentHorizontal1).
```

```
printline([]).
printline( [H|T] ):-
  write('   |   '),
  printchar(H),
  printline(T).
```

Resultando:



4.3 Validação das Jogadas

A regra principal do nosso programa é play(E,XS,YS,XD,YD), no ínicio desta são verificados casos básicos de regras: xDIFy(XS,YS,XD,YD) verifica se as coordenadas são exatamente iguais se forem falha; size(XS,YS) vê se XS e YS estão dentro do tabuleiro e o mesmo para XD e YD. No fim de play é testado se estamos em condição final através do continueGame(N). O resto das regras são verificadas em canimovethere(E,XS,YS,XD,YD), cuja cabeça está em atualizarEstado(E,XS,YS,XD,YD,N) que por sua vez está em play(E,XS,YS,XD,YD).

Em canimovethere(E,XS,YS,XD,YD) é chamado as regras(E,XS,YS,XD,YD). Lá encontram-se as regras restantes: é verificado se a cor na posição XS,YS tem o mesmo valor que a cor de quem deve jogar; temos um outra regra que verifica se XS,YS e XD,YD têm a mesma côr, se tiverem falha e de seguida temos a regra mais complicada que verifica se se pode comer ou mover uma phalanx(engloba todos os tipos de linhas).

```
115 play( E, XS, YS, XD, YD) :-
116   xDIFy(XS,YS,XD,YD),
117   size(XS,YS),
118   size(XD,YD),
```

```
149 regras([H|T], XS, YS, XD, YD, N, XN, YN) :-
150   turn(C),
151   checkcolour(E,XS,YS,C),
152   regra('sobreposicao',E,XS,YS,XD,YD),
153   regra('moverPhalanx',E,XS,YS,XD,YD,N,XN,YN).
```

4.4 Lista de Jogadas Válidas

Não tivemos tempo para completar esta funcionalidade, mas estava previsto criar um predicado recursivo que percorre-se todos os XS,YS,XD,YD e caso as regras fossem válidas inseríamos numa lista uma lista com os movimentos válidos, por exemplo, [[1,2,3,4],[1,4,3,7]]. Este ciclo era repetido até extinguir as possibilidades de XS,YS,XD,YD ou até um valor razoável se houver pelo menos um movimento válido.

4.5 Execução de Jogadas

Depois de todas as regras serem validadas, a execução da jogada é feita em duas partes. Na primeira parte é eliminada a phalanx inimiga caso o movimento efetuado pelo jogador seja para comer essa phalanx, isto é feito com recurso à regra chamada deletePhalanx(E,XS,YS,XD,YD,PhalanxSizeAdversary,XN,YN,E1), caso não haja nenhuma phalanx para comer esta regra apenas devolve o mesmo tabuleiro. Na segunda parte é efectuada a movimentação da phalanx do jogador com a regra changeState(E,XS,YS,XD,YD,E1), que move a phalanx o número de casas pretendido.

Tanto na primeira como na segunda regra existem oito casos diferentes, que são as oito movimentações possíveis, e para se saber qual a orientação a usar utilizam-se os valores de X,Y de origem(S) e de destino(Y).

```
612 changeState(E,XS,YS,XD,YD,E1):-
613   YS == YD,
614   XS < XD,
615   turn(C),
616   HorizontalDifference is XD-XS,
617   getSizeTilColour('horizontalLine',HorizontalDifference,_,E,C,XS,YS,N),
618   movePhalanxRIGHT(E,XS,XD,YS,N,E1).
619
620 changeState(E,XS,YS,XD,YD,E1):-
621   YS == YD,
622   XS > XD,
623   turn(C),
624   HorizontalDifference is XD-XS,
625   getSizeTilColour('horizontalLine',HorizontalDifference,_,E,C,XS,YS,N),
626   movePhalanxLEFT(E,XS,XD,YS,N,E1).
627
628 % changeState :- estadoInicial(E), changeState(E,0,0,0,2,E1),show(E1).
629 changeState(E,XS,YS,XD,YD,E1):-
630   XS == XD,
```

```
684 deletePhalanxLEFT(E,X,Y,N,E1):-
685   N>0, X>=0,
686   X1 is X-1,
687   N1 is N-1,
688   deletePhalanxLEFT(E,X1,Y,N1,E2),
689   changechar(0,X,Y,E2,E1).
690 deletePhalanxLEFT(E,_,_,0,E).
691
692 % deletePhalanxRIGHT:- estadoInicial(E), deletePhalanxRIGHT(E,2,0,2,E1), sh
693 deletePhalanxRIGHT(E,X,Y,N,E1):-
694   N>0, X>=0,
695   X1 is X+1,
696   N1 is N-1,
697   deletePhalanxRIGHT(E,X1,Y,N1,E2),
698   changechar(0,X,Y,E2,E1).
699 deletePhalanxRIGHT(E,_,_,0,E).
700
701 % deletePhalanxUP:- estadoInicial(E), deletePhalanxUP(E,2,1,2,E1), show(E1
```

4.6 Avaliação do Tabuleiro

4.7 Final do Jogo

Para verificar o fim do jogo, existe uma regra chamada continueGame(E) que recebe o tabuleiro e dependendo de qual jogador está a jogar ela vai ver se esse jogador tem mais peças na linha do tabuleiro mais distante de si do que o adversário tem na linha mais próxima de si, caso isso aconteça esta regra é falsa e o nosso ciclo de jogo que é a regra play. O vencedor é determinado com a regra winner.

```
116 play( E, XS, YS, XD, YD) :-
117   xDIFy(XS,YS,XD,YD),
118   size(XS,YS),
119   size(XD,YD),
120   atualizarEstado(E,XS,YS,XD,YD,N),
121   show(N),
122   !,continueGame(N),
123   read(XS1), read(YS1), read(XD1), read(YD1), % le desta forma XS.YS.XD.YD.
124   play( N, XS1, YS1, XD1, YD1).
125 %If sizes are not allowed retry.
126 play(E,_,_,_,_):-badMoveFriend(E).
127
128 xDIFy(XS,YS,XD,YD) :-
129   XS =\= XD; YS =\= YD.
130
131 atualizarEstado( E, XS, YS, XD, YD, N) :-
132   XS1 is XS - 1,
133   YS1 is 12 - YS, % Porque na verdade o que aparece no tabuleiro tem o índice trocado.
134   XD1 is XD - 1,
135   YD1 is 12 - YD1,
136   canimovethere(E,XS1,YS1,XD1,YD1,PhalanxSizeAdversary,XN,YN),
137   deletePhalanx(E,XS1,YS1,XD1,YD1,PhalanxSizeAdversary,XN,YN,E1),
138   changeState(E1,XS1,YS1,XD1,YD1,N),
139   turnchange(C).
140 atualizarEstado(E,_,_,_,_,_):-badMoveFriend(E).
```

4.8 Jogada do Computador

4.9 Comunicação com Visualizador 3D

5 Interface com o Utilizador

Para inicar o jogo basta escrever start.. Esta regra trata de ir buscar o tabuleiro inicial, mostrá-lo, ler os valores de XS,YS,XD,YD e depois chama a regra principal do program play(E,XS,YS,XD,YD). Esta regra play(E,XS,YS,XD,YD) fica num "while infinito"até continueGame falhar, que indica que estamos em condição final. Como continueGame tem um cut atrás, o prolog vai cortar os "plays"e logo de seguida tenta procurar por outra cabeça start. Essa outra cabeça no nosso caso é sempre verdadeira o que faz com que seja retornado yes no fim do jogo. Caso play falhe irá ser procurado outra regra play, esta nossa outra regra play chama badMoveFriend. O objetivo de badMoveFriend é pedir novamente por uma jogada.

```
109 start :-
110   estadoInicial(E),
111   show(E),
112   read(XS), read(YS), read(XD), read(YD), %le desta forma XS.YS.XD.YD.
113   play(E, XS, YS, XD, YD).
```

```
126 play(E,_,_,_,_):-badMoveFriend(E).
```

```
403 %If a fact/rule fails we want player to have another go.
404 badMoveFriend(E):-
405   nl,
406   write('That move is not allowed.'),
407   nl,
408   show(E),
409   read(XS), read(YS), read(XD), read(YD), % le desta forma XS.YS.XD.YD.
410   play( E, XS, YS, XD, YD).
```

O jogador só tem que fazer start. E ir escrevendo as coordenadas da seguinte forma XS.YS.XD.YD..

6 Conclusões e Perspectivas de Desenvolvimento

O Epaminondas é um jogo cheio de estratégia e regras. Como tal, tivemos de fazer várias regras de remoção, testes, etc dependendo da natureza da linha de movimentação das phalanx e portanto esta parte foi, na nossa opinião, a mais trabalhosa. Infelizmente, só conseguimos acabar a parte jogador vs jogador, isto deve-se ao facto de termos outros projetos em mão. Embora não tenhamos feito a parte relacionada com o computador temos um ideia bastante razóavel de como o faríamos, e ,caso possível, iremos implementá-la num futuro próximo.

Gostaríamos de ter acaba o trabalho em tempo útil mas não nos foi possível. Contudo, foi uma boa experiência criar um jogo pensando em PROLOG.

Bibliografia

[1] Bob Abbott. Epaminondas. http://www.logicmazes.com/games/epam.html, Revised: December 11, 2010. The website of the game creator.

[2] Kerry Handscomb. Abstract games issue 3 autumn 2000. http://www.logicmazes.com/games/epam/index.html, 3 Autumn 2000. Epaminondasm... a game of classical elegance.