

# Xadrersi - TP1

Relatório Intercalar



Mestrado Integrado em Engenharia Informática e  
Computação

Programação em Lógica

**Grupo Xadresi:3**

João Sá - up201506252

Mário Esteves - up201607940

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

22 de Dezembro de 2017

## Resumo

Este relatório tem como objetivo descrever o primeiro trabalho proposto da U.C. Programação em Lógica. Este foi desenvolvido usando a linguagem PROLOG, sendo que tinha como fim criar um programa que permitisse jogar Xadrsersi.

Foram cumpridos os objetivos deste trabalho, com a exceção da criação das rotinas de inteligência artificial com uma estratégia de jogo que não fosse aleatória, e o número de erros causados ao jogar humano contra computador ainda são relativamente elevados. Vamos na mesma abordar os predicados implementados para esse fim.

Com essa exceção todos os requisitos principais na implementação do jogo foram alcançados, permitindo a operação normal do jogo.

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>4</b>
<b>2</b>	<b>O Jogo Xadrersi</b>	<b>4</b>
<b>3</b>	<b>Lógica do Jogo</b>	<b>6</b>
3.1	Representação do Estado do Jogo . . . . .	6
3.2	Visualização do Tabuleiro . . . . .	8
3.3	Lista de Jogadas Válidas . . . . .	11
3.4	Execução de Jogadas . . . . .	13
3.5	Avaliação do Tabuleiro . . . . .	14
3.6	Final do Jogo . . . . .	15
3.7	Jogada do Computador . . . . .	15
<b>4</b>	<b>Interface com o Utilizador</b>	<b>17</b>
<b>5</b>	<b>Conclusões</b>	<b>18</b>

## 1 Introdução

Descrever os objetivos e motivação do trabalho. Descrever num parágrafo breve a estrutura do relatório.

## 2 O Jogo Xadrersi

Criado por Andy Lewicki consiste numa combinação do Xadrez com o Reversi, joga-se num tabuleiro tradicional de Xadrez (8x8) inicialmente vazio, os jogos têm sempre exactamente 8 jogadas.

Cada jogador começa com 8 peças - 1 Rei, 1 Rainha, 2 Torres, 2 Bispos e 2 Cavalos.

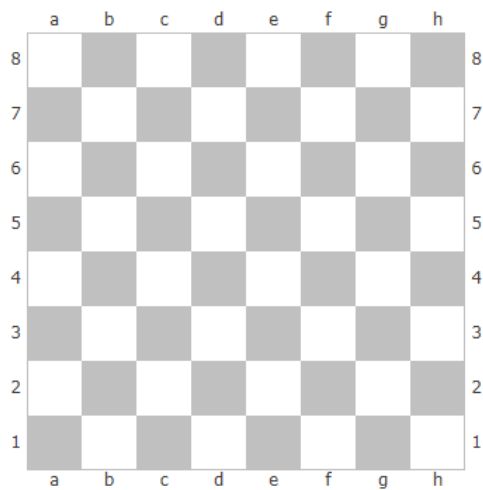


Figura 1: Estado inicial do jogo.

### Regras:

- As brancas começam o jogo e os jogadores vão alternadamente colocando uma peça em qualquer casa vazia do tabuleiro.
- As brancas começam o jogo, por colocar o Rei em qualquer casa do tabuleiro, e as pretas respondem colocando qualquer peça excepto o Rei.

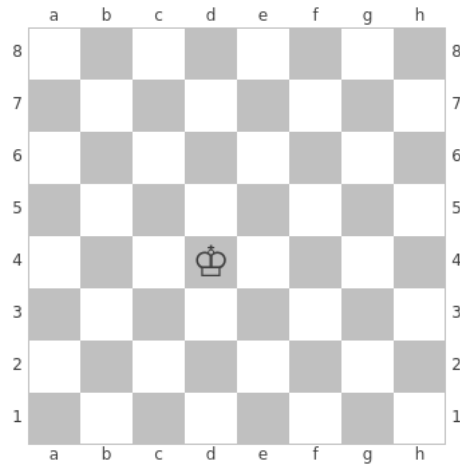


Figura 2: Rei branco posicionado no 1º turno.

- As peças colocadas, tirando a primeira peça das Brancas (Rei) têm sempre que tocar numa peça já colocada, seja na horizontal, vertical ou diagonal.

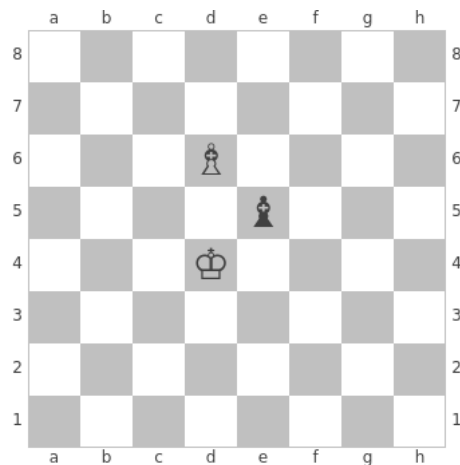


Figura 3: Possível arranjo do tabuleiro após 3 jogadas.

- O Rei preto tem que ser a última peça a ser colocada no tabuleiro e tem de tocar na última peça Branca jogada, ou seja, a última peça das brancas tem que ser colocada de maneira a que o Rei Preto lhe possa tocar na última jogada.
- Os Bispos devem ser colocados em casas de cores diferentes, como no Xadrez.
- Qualquer jogador pode forçar a saída da Rainha adversária, jogando a sua rainha, o adversário é obrigado a jogar a sua própria rainha imediatamente.
- Os jogadores ganham 1 ponto por cada casa vazia atacada por uma ou mais das suas peças, de acordo com as regras clássicas do xadrez. Uma

casa atacada por várias peças é contada uma vez por cada peça atacada em separado.

- Se ambos os jogadores ficarem com o mesmo número de pontos, o jogo termina empatado.

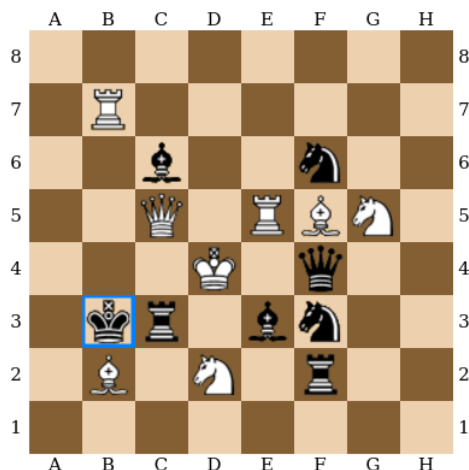


Figura 4: Jogo finalizado onde as brancas ganharam 54-37

### 3 Lógica do Jogo

Nesta secção vamos descrever a lógica do jogo, incluindo estados iniciais e finais do jogo, jogadas permitidas, e a representação interna dos estados do jogo.

#### 3.1 Representação do Estado do Jogo

Na base de conhecimentos do prolog implementamos o seguinte predicado:

---

```
tabuleiro([[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]]).
```

---

Este predicado contém uma lista de listas, na qual cada sublista representa uma linha do tabuleiro, e cada elemento dessa mesma sublista representa uma posição na linha. Todos os predicados construídos para manipulação da lista têm como input as coordenadas X, Y, a começar por 1 (e não por zero, como convencionado para outras linguagens de programação).

A diferença na manipulação da lista, é que os índices do topo são tratados por 1, e não por 8, e aumentam, ao invés do que aparece nas imagens de exemplo do jogo em que a linha do topo é representada pelo índice 8. Para representar

as peças na lista, cada atomo das sublistas pode ter o seguinte valor:

- 0 - Casa vazia;
- 1 - Rei Branco;
- 2 - Rainha Branca;
- 3 - Cavalo Branco;
- 4 - Torre Branca;
- 5 - Bispo Branco;
- 6 - Rei Preto;
- 7 - Rainha Preta;
- 8 - Cavalo Preto;
- 9 - Torre Preta;
- 10 - Bispo Preto.

Para exemplificar a representação do estado do jogo vamos tomar como exemplo a figura 1 e representá-lo no predicado tabuleiro/1:

A representação das peças no ecrã é diferente e vai ser explicada na secção seguinte.

Para fazer a representação do tabuleiro na consola recorre-se ao predicado `writeChessboard/0`. Esse predicado está implementado da seguinte forma:

Este predicado começa por escrever a primeira linha, que marca o topo do tabuleiro. Recorre depois ao predicado `tabuleiro/1` para obter a representação do tabuleiro em matriz, e usa a mesma no predicado `writeChessboardLines/1` para escrever cada linha do tabuleiro:

Este predicado é recursivo, e tem como caso base a lista vazia, caso no qual existe um cut para parar a execução da escrita das linhas do tabuleiro. No caso normal é escrito em primeiro lugar o topo de cada casa da linha que estamos a escrever, seguido da escrita de cada elemento pertencente a cada casa do tabuleiro utilizando o predicado `writeChessElements/1`:



---

```

writeChessElements([]) :- !.
writeChessElements([0|ChessTail]) :- writef(" |"),
    writeChessElements(ChessTail).
writeChessElements([1|ChessTail]) :- writef(" R B |"),
    writeChessElements(ChessTail).
writeChessElements([2|ChessTail]) :- writef(" RaB |"),
    writeChessElements(ChessTail).
writeChessElements([3|ChessTail]) :- writef(" C B |"),
    writeChessElements(ChessTail).
writeChessElements([4|ChessTail]) :- writef(" T B |"),
    writeChessElements(ChessTail).
writeChessElements([5|ChessTail]) :- writef(" B B |"),
    writeChessElements(ChessTail).
writeChessElements([6|ChessTail]) :- writef(" R P |"),
    writeChessElements(ChessTail).
writeChessElements([7|ChessTail]) :- writef(" RaP |"),
    writeChessElements(ChessTail).
writeChessElements([8|ChessTail]) :- writef(" C P |"),
    writeChessElements(ChessTail).
writeChessElements([9|ChessTail]) :- writef(" T P |"),
    writeChessElements(ChessTail).
writeChessElements([10|ChessTail]) :- writef(" B P |"),
    writeChessElements(ChessTail).

```

---

Como estamos a enviar cada linha da matriz para este predicado, simplesmente verificamos qual é o tipo de peça e escrevemos para o ecrã a representação da mesma, de acordo com a seguinte conversão:

- 0 - Casa vazia;
- 1 - R B ;
- 2 - RaB ;
- 3 - C B ;
- 4 - T B ;
- 5 - B B ;
- 6 - R P ;
- 7 - RaP ;
- 8 - C P ;
- 9 - T P ;
- 10 - B P .

Sendo que cada representação é simplesmente as iniciais da peça (exceto a rainha por razões óbvias), seguido da inicial da cor a que pertence.

Uma demonstração do predicado no estado inicial do jogo é mostrada na imagem seguinte:

```
?- writeChessboard.
```


```
true .
```

Figura 5: Output do predicado writeChessboard/0 com o estado inicial do jogo.

E utilizando a figura 4 como referência para um possível estado final de jogo obter-se-ia o seguinte output:

```
?- writeChessboard.
```

	T	B								
		B	P			C	P			
		R	a	B		T	B	B	C	B
				R	B		R	a	P	
	R	P	T	P		B	P	C	P	
	B	B		C	B		T	P		

```
true .
```

Figura 6: Output do predicado writeChessboard/0 com o estado final do jogo presente na Fig. 4.

### 3.3 Lista de Jogadas Válidas

Para a geração da lista de jogadas válidas, temos dois predicados:

- `jogadas_possiveis/1`, que armazena o grupo de peças que podem ser posicionadas no tabuleiro em tal jogada;
- `pos_jogadas_possiveis/1`, que armazena o grupo de posições possíveis para posicionar as peças permitidas.

Para o primeiro predicado temos a seguinte definição inicial:

---

```
jogadas_possiveis(rei).
```

---

Isto porque no turno inicial do jogo a única peça permitida é o rei. Este predicado é modificado com as chamadas ao predicado `posicionar_peca/6`, que de acordo com as regras de jogo modifica isto para:

---

```
jogadas_possiveis(todas).  
    OU  
jogadas_possiveis(rainha).
```

---

Sendo que com `jogadas_possiveis(todas)`, todas as peças exceto o rei são permitidas, pois o rei só pode ser posicionado no primeiro e último turnos, e com `jogadas_possiveis(rainha)`, só a rainha pode ser posicionada, pois existe a regra de que ao ser jogada a rainha, o jogador seguinte só pode jogar a rainha.

Para o predicado `pos_jogadas_possiveis/1`, a sua modificação ocorre quando é chamado o predicado `xadrersi/3`. Ao chamar-mos o predicado referido se a jogada for válida então é construída uma lista de jogadas possíveis através dos seguintes predicados:

---

```

build_list_possible_plays(T, E) :- retract(pos_jogadas_possiveis(_)),
    assert(pos_jogadas_possiveis([])),
    build_list_possible_plays_line(T, 1, E).

build_list_possible_plays_line([], _, _) :- !.
build_list_possible_plays_line([THead|TTail], Y, E) :-
    build_list_possible_plays_col(THead, 1, Y, E), Y1 is Y + 1,
    build_list_possible_plays_line(TTail, Y1, E).

build_list_possible_plays_col([], _, _, _) :- !.
build_list_possible_plays_col([O|T], X, Y, E) :- X1 is X + 1,
    build_list_possible_plays_col(T, X1, Y, E), !.
build_list_possible_plays_col([H|T], X, Y, brancas) :- H > 5,
    retract(pos_jogadas_possiveis(L)), PrevCol is X - 1, NextCol is X +
    1, PrevLine is Y - 1, NextLine is Y + 1, append(L, [[PrevLine,
    PrevCol], [PrevLine, X], [PrevLine, NextCol], [Y, PrevCol], [Y,
    NextCol], [NextLine, PrevCol], [NextLine, X], [NextLine, NextCol]]),
    L2), assert(pos_jogadas_possiveis(L2)), X1 is X + 1,
    build_list_possible_plays_col(T, X1, Y, brancas), !.
build_list_possible_plays_col([H|T], X, Y, pretas) :- H < 6,
    retract(pos_jogadas_possiveis(L)), PrevCol is X - 1, NextCol is X +
    1, PrevLine is Y - 1, NextLine is Y + 1, append(L, [[PrevLine,
    PrevCol], [PrevLine, X], [PrevLine, NextCol], [Y, PrevCol], [Y,
    NextCol], [NextLine, PrevCol], [NextLine, X], [NextLine, NextCol]]),
    L2), assert(pos_jogadas_possiveis(L2)), X1 is X + 1,
    build_list_possible_plays_col(T, X1, Y, pretas), !.
build_list_possible_plays_col([_|T], X, Y, E) :- X1 is X + 1,
    build_list_possible_plays_col(T, X1, Y, E).

```

---

Resumindo a operação deste predicado: no final de cada ciclo de jogo e admitindo jogada válida, o predicado `build_list_possible_plays/2` é chamado com a equipa seguinte (E), e com o tabuleiro resultante da jogada anterior (T). Após isso todas as linhas da matriz do tabuleiro são corridas recorrendo ao predicado `build_list_possible_plays_line/3`, que corre todas as casas de tal linha do tabuleiro recorrendo ao terceiro predicado `build_list_possible_plays_col/4`.

É neste último predicado que ocorre a verificação da peça de acordo com a equipa, e caso seja uma peça inimiga, todas as casas vizinhas são adicionadas à lista de jogadas possíveis (incluindo jogadas fora do tabuleiro). No predicado que vamos explorar na secção seguinte `posicionar_peca/6`, é que esta lista é então utilizada para verificação da jogada correcta.

### 3.4 Execução de Jogadas

Para posicionamento e validação das peças no tabuleiro temos o seguinte grupo de predicados:

---

```
posicionar_pec(rei, X, Y, T, T2, brancas) :- jogadas_possiveis(rei),
    retract(jogadas_possiveis(_)), assert(jogadas_possiveis(todas)),
    retract(quantidade_de_pecas(rei, brancas, 1)),
    assert(quantidade_de_pecas(rei, brancas, 0)), replace(T, X, Y, 1,
    T2), !.
posicionar_pec(rainha, X, Y, T, T2, brancas) :-
    (jogadas_possiveis(rainha), retract(jogadas_possiveis(_)),
    assert(jogadas_possiveis(todas)); jogadas_possiveis(todas),
    retract(jogadas_possiveis(_)), assert(jogadas_possiveis(rainha))),
    retract(quantidade_de_pecas(rainha, brancas, 1)),
    assert(quantidade_de_pecas(rainha, brancas, 0)), replace(T, X, Y,
    2, T2), !.
posicionar_pec(cavalo, X, Y, T, T2, brancas) :-
    jogadas_possiveis(todas), retract(quantidade_de_pecas(cavalo,
    brancas, Q)), Q1 is Q - 1, assert(quantidade_de_pecas(cavalo,
    brancas, Q1)), replace(T, X, Y, 3, T2), !.
posicionar_pec(torre, X, Y, T, T2, brancas) :-
    jogadas_possiveis(todas), retract(quantidade_de_pecas(torre,
    brancas, Q)), Q1 is Q - 1, assert(quantidade_de_pecas(torre,
    brancas, Q1)), replace(T, X, Y, 4, T2), !.
posicionar_pec(bispo, X, Y, T, T2, brancas) :-
    jogadas_possiveis(todas), casa_primeiro_bispo(brancas, nao_existe),
    cor_da_casa(X, Y, Cor), retract(casa_primeiro_bispo(brancas, _)),
    assert(casa_primeiro_bispo(brancas, Cor)),
    retract(quantidade_de_pecas(bispo, brancas, Q)), Q1 is Q - 1,
    assert(quantidade_de_pecas(bispo, brancas, Q1)), replace(T, X, Y,
    5, T2), !.
posicionar_pec(bispo, X, Y, T, T2, brancas) :-
    jogadas_possiveis(todas), casa_primeiro_bispo(brancas, _),
    cor_da_casa(X, Y, _), retract(quantidade_de_pecas(bispo, brancas,
    Q)), Q1 is Q - 1, assert(quantidade_de_pecas(bispo, brancas, Q1)),
    replace(T, X, Y, 5, T2), !.
```

---

Como podemos ver, o predicado posicionar\_pec/6 verifica se as jogadas possíveis em termos de peças se concretizam antes de guardar a quantidade de peças restantes após o posicionamento da peça respetiva.

Estes predicados também existem com a versão para a equipa das peças pretas, no qual o predicado replace/5 tem o seu quarto argumento modificado para ter as versões respetivas das peças da equipa preta.

Finalmente, este predicado também tem as definições de jogadas inválidas antes da definição supra referida:

---

```

posicionar_pecas(rei, X, Y, T, T2, pretas) :- quantidade_de_pecas(rei,
    brancas, 0), quantidade_de_pecas(rainha, brancas, 0),
    quantidade_de_pecas(cavalo, brancas, 0), quantidade_de_pecas(torre,
    brancas, 0), quantidade_de_pecas(bispo, brancas, 0),
    quantidade_de_pecas(rei, pretas, 1), quantidade_de_pecas(rainha,
    pretas, 0), quantidade_de_pecas(cavalo, pretas, 0),
    quantidade_de_pecas(torre, pretas, 0), quantidade_de_pecas(bispo,
    pretas, 0), retract(quantidade_de_pecas(rei, pretas, 1)),
    assert(quantidade_de_pecas(rei, pretas, 0)), replace(T, X, Y, 6,
    T2), !.

posicionar_pecas(P, _, _, T, T, E) :- jogadas_possiveis(P2),
    (quantidade_de_pecas(P, E, X), X = 0, write("Nao existem mais pecas
    deste tipo. Refaca a jogada."), nl; P \= P2, P2 = rei, write("So
    pode posicionar o rei nesta jogada. Refaca a jogada."), nl; P \=
    P2, P2 = rainha, write("So pode posicionar a rainha nesta jogada.
    Refaca a jogada."), nl; P2 = todas, P = rei, write("Os reis so
    podem ser posicionados na primeira e ultima jogadas. Refaca a
    jogada."), nl), !.

posicionar_pecas(_, X, Y, T, T, _) :- ((X < 1; X > 8; Y < 1; Y > 8),
    write("Posicao fora do tabuleiro. Refaca a jogada."), nl; nth1(Y,
    T, Line), nth1(X, Line, P), P \= 0, write("Casa nao esta vazia.
    Refaca a jogada."), nl; \+ pos_jogadas_possiveis(todas),
    pos_jogadas_possiveis(L), \+ member([Y, X], L), write(Essa jogada
    nao e possivel. Refaca a jogada.), nl), !.

posicionar_pecas(bispo, X, Y, T, T, E) :- casa_primeiro_bispo(E, Cor),
    cor_da_casa(X, Y, Cor), write("Nao se pode colocar bispos em casas
    da mesma cor. Refaca a jogada.\n"), !.

```

---

Como podemos ver, o primeiro predicado definido, define a ultima jogada do jogo, em que so o rei pode ser jogado. Todos os outros são definições de jogadas inválidas em que é retornado o mesmo tabuleiro (T, T nos argumentos), e que ao ser feita uma jogada de certo tipo é apresentada a mensagem de erro respetiva.

### 3.5 Avaliação do Tabuleiro

Como observado na secção de Resumo do relatório, não fazemos uma avaliação profunda do tabuleiro aquando da jogada do computador, sendo essa uma das maiores dificuldades. Portanto toda a avaliação do tabuleiro resume-se à identificação das posições possíveis, bem como das peças a jogar.

Na secção 3.7, iremos explorar mais profundamente as rotinas implementadas para a jogada de computador.

### 3.6 Final do Jogo

Para determinar o final do jogo, faz-se a avaliação das peças restantes no predicado `game_end/1`:

---

```
game_end(Tabuleiro) :- quantidade_de_pecas(rei, brancas, 0),
    quantidade_de_pecas(rainha, brancas, 0),
    quantidade_de_pecas(cavalo, brancas, 0), quantidade_de_pecas(torre,
brancas, 0), quantidade_de_pecas(bispo, brancas, 0),
    quantidade_de_pecas(rei, pretas, 0), quantidade_de_pecas(rainha,
pretas, 0), quantidade_de_pecas(cavalo, pretas, 0),
    quantidade_de_pecas(torre, pretas, 0), quantidade_de_pecas(bispo,
pretas, 0), executar_ataques(Tabuleiro),
    writeChessboard(Tabuleiro), pontos(pretas, PP), pontos(brancas,
PB), write("Pontos pretas: "), write(PP), nl, write("Pontos
brancas: "), write(PB), nl, (PP > PB, write("As pretas venceram o
jogo!"), nl; PB > PP, write("As brancas venceram o jogo!"), nl; PP
= PB, write("O jogo terminou num empate!"), nl).
```

---

Como se pode ver, avalia-se a quantidade de peças, e se todas já tiverem sido usadas, calcula-se o número de pontos para cada equipa, faz-se o display do número de pontos para cada equipa, e imprime-se a mensagem adequada a quem venceu o jogo.

No final do jogo, faz-se a avaliação das peças que atacam outras peças, pelas regras do xadrez, sem ter em conta limites de raio, a não ser que não possa atacar com raio ilimitado (por exemplo, a torre ataca toda a linha onde esta e toda a coluna, mas o rei só ataca as casas vizinhas).

### 3.7 Jogada do Computador

Para as jogadas do computador, foi utilizada uma estratégia aleatoria, e para manter a estruturação do nosso trabalho, fizemos alguns predicados adicionais para o ciclo de jogo:

---

```
xadrersi(c, T, T, E) :- xadrersi(T, E), !.
xadrersi(c, T, T2, E) :- trocar_equipa(E, E2),
    build_list_possible_plays(T2, E2), jogada_computador(T2, T3, E2),
    build_list_possible_plays(T3, E), xadrersi(T3, E).
jogada_computador(T, T2, E) :- jogadas_possiveis(rei),
    pos_jogadas_possiveis(todas), random_between(1, 8, RX),
    random_between(1, 8, RY), jogada_computador(rei, T, T2, E, RY, RX),
    !.
jogada_computador(T, T2, E) :- jogadas_possiveis(rei),
    pos_jogadas_possiveis(JP),
    random_member([JogadaFinalY|JogadaFinalX], JP),
    jogada_computador(rei, T, T2, E, JogadaFinalY, JogadaFinalX), !.
jogada_computador(T, T2, E) :- jogadas_possiveis(rainha),
    pos_jogadas_possiveis(JP),
    random_member([JogadaFinalY|JogadaFinalX], JP),
    jogada_computador(rainha, T, T2, E, JogadaFinalY, JogadaFinalX), !.
jogada_computador(T, T2, E) :- jogadas_possiveis(todas),
    pos_jogadas_possiveis(JP),
    random_member([JogadaFinalY|JogadaFinalX], JP), random_member(P,
[rainha, cavalo, torre, bispo]), quantidade_de_pecas(P, E, Q), Q >
0, jogada_computador(P, T, T2, E, JogadaFinalY, JogadaFinalX), !.
```

---

```
jogada_computador(P, T, T2, E, Y, [X|XT]) :- posicionar_peca(P, X, Y, T,  
T2, E).
```

---

O predicado `xadrersi/4`, em que o primeiro argumento é o átomo `c`, significaria que o computador seria o jogador a fazer a jogada. De seguida temos o predicado `jogada_computador/3`, que aceita o tabuleiro `T`, devolve `T2`, e de acordo com as possibilidades escolhe a peça correspondente às restrições do jogo, sempre gerando aleatoriamente a posição e/ou a peça para a sua próxima jogada. Por fim, o predicado do mesmo nome mas aridade 6, serve para posicionar a peça no tabuleiro.

As maiores dificuldades ainda não resolvidas em relação ao jogo com o computador foram as seguintes:

- Não conseguimos gerar correctamente uma peça aleatória que ainda possa ser utilizada. Isto leva ao problema de que o turno do computador seja passado sem que ele coloque uma peça, e isto leva a que o jogo não termine (a maneira como verificamos a terminação do jogo é através da verificação da quantidade de peças);
- Não conseguimos começar o jogo sem estar com a equipa do humano em brancas. Isto leva a um problema de `stack`, algo que ainda não está claro como é que acontece;
- Ao implementar o modo `computador versus computador`, não encontramos nenhuma rotina que parasse a execução entre turnos, o que nos levou a excluir a implementação desse modo das nossas prioridades.



## 4 Interface com o Utilizador

Para a interface com o utilizador, temos a representação através do predicado mostrado na secção 3.2 acompanhado de mensagens que ilustram o estado atual do jogo. Para o jogador escolher temos 3 mensagens a pedir a peça, e as coordenadas para a colocar, tal como podemos ver a seguir:

So pode posicionar o rei nesta jogada. Refaca a jogada.

Turno: brancas


Peca: | rei.

Coordenadas:

X: | 2.


Y: | 1.

Figura 7: Jogo após uma jogada invalida em que o jogador escolheu o rei.

Ou no caso de mensagem de erro, é apresentada a mensagem de erro acima do tabuleiro:

So pode posicionar o rei nesta jogada. Refaca a jogada.

Turno: brancas



Peca: **K**

Figura 8: Erro no posicionamento de uma peça.

Isto recorre inteiramente ao predicado `writeChessboard` visto na secção 3.2, e também a mais mensagens no predicado de ciclo de jogo:

---

```
xadrersi(Tabuleiro, E) :- modo_de_jogo_atual(humano_vs_humano), !,  
    write("Turno: "), write(E), nl, writeChessboard(Tabuleiro),  
    write("Peca: "), read(P), nl, write("Coordenadas: \nX: "), read(X),  
    nl, write("Y: "), read(Y), posicionar_peca(P, X, Y, Tabuleiro, T2,  
    E), xadrersi(Tabuleiro, T2, E).
```

---

## 5 Conclusões

Caso houvesse mais tempo disponível, a primeira prioridade seria tentar resolver os problemas no que toca à implementação da inteligência artificial, e a segunda prioridade seria implementar uma estratégia mais sólida nas jogadas de computador (como exemplo implementar uma rotina minmax). Por fim, este trabalho ajudou-nos a desenvolver as capacidades de manipulação de listas, e também proporcionou uma oportunidade para aplicar um novo paradigma de programação.