Relatório - Xadrez em Racket

Ricardo Henrique Brunetto[1]

[1]Departamento de Informática – Universidade Estadual de Maringá (UEM)
Maringá – PR – Brasil

ra94182@uem.br

1. Introdução

O presente documento é um relatório do desenvolvimento de um jogo de Xadrez na linguagem funcional Racket para a disciplina 6902 - Paradigma de Programação Lógica e Funcional, ministrada pelo professor Dr. Wagner Igarashi para a turma de 2015 de Bacharelado em Ciência da Computação pela Universidade Estadual de Maringá.

Neste relatório serão abordadas seções conforme socilitado na especificação do trabalho, que pode ser encontrada em anexo a este documento.

2. O Jogo

O jogo foi desenvolvido em Racket com uso do editor de textos Atom e da IDE DrRacket. Fazer-se-á uma breve explanação a respeito das regras do jogo e de seu funcionamento. Em seguida, serão apresentados os aspectos avaliativos, conforme solicitado.

Em suma, o jogo segue as regras de um Xadrez convencional, sem, no entanto, movimentos especiais. Isso significa que o jogador não será capaz de aplicar técnicas como *The Rock*. Não há, também, avaliação quanto ao xeque (ou xeque mate).

Quanto à movimentação das peças, seguem-se as regras-padrão do jogo.

2.1. Variáveis do Jogo

O jogo foi construído utilizando a biblioteca `universe`, o que requireu que fosse desenvolvida uma estrutura de `world` [Lang]. Internamente, a cada interação do usuário com o jogo, é criada uma nova estrutura de `world`.

Dessa forma, desenvolveu-se a esturuta *jogada*. Essa estrutura tem por objetivo simular um `world`, ou seja, encapsular todas as variáveis que se alteram entre as jogadas.

Resumo. `(struct jogada (tab jogador king ptsB ptsP))`

Dentre as variáveis, encontram-se:

- `tab` que se refere a um Tabuleiro (definido posteriormente) $\{mutable - array\}$;
- `jogador` que se refere ao Jogador que está atualmente jogando (definido posteriormente) $\{jogador\}$;
- `king` que se refere à variável que atesta que ambos reis estão vivos (0) ou algum está morto (1) $\{number\}$;
- `ptsB` que se refere aos pontos do jogador das peças brancas $\{number\}$;
- `ptsP` que se refere aos pontos do jogador das peças pretas $\{number\}$.

As principais estruturas do jogo são `peca`, `pos` e `tabuleiro`.

- `peca` é uma **peça do tabuleiro**, devendo possuir:
  `id`: identificação única inteira id $\{number\}$;
  `tipo`: informa a classe da peça, podendo variar entre **peao, torre, cavalo, bispo, rainha, rei** $\{string\}$.
  `cor`: informa a cor da peça (pertinência ao jogador que a controla) $\{color\}$.
  `imagem`: informa o bitmap da peça (interface gráfica) $\{bitmap\}$.
- `pos` é uma **posição do tabuleiro**, devendo possuir:
  `x` e `y`: coordenadas inteiras, onde $0 \leq x, y \leq 7$ $\{number\}$;
  `destinavel`: um booleando para informar se a posição é um destino válido de outra $\{boolean\}$;
  `peca`: que representa a peça que ocupa a posição (pode ser `empty`) $\{peca\}$.
- `tabuleiro` representa o próprio **tabuleiro**, sendo um $array$ 8×8 de `pos`, cada qual com sua `peca`. $\{mutable - array\}$;

```
(struct peca (id tipo cor imagem) #:transparent #:mutable)
(struct pos (x y destinavel peca) #:transparent #:mutable)
(define tabuleiro (mutable-array #[#[A8 B8 C8 D8 E8 F8 G8 H8]
                                   #[A7 B7 C7 D7 E7 F7 G7 H7]
                                   #[A6 B6 C6 D6 E6 F6 G6 H6]
                                   #[A5 B5 C5 D5 E5 F5 G5 H5]
                                   #[A4 B4 C4 D4 E4 F4 G4 H4]
                                   #[A3 B3 C3 D3 E3 F3 G3 H3]
                                   #[A2 B2 C2 D2 E2 F2 G2 H2]
                                   #[A1 B1 C1 D1 E1 F1 G1 H1]]))
```

onde cada Kij, com $K \in \{A, \ldots, H\}$ $i, j \in \{1, \ldots, 8\}$ é uma instância definida de `pos`.

Para controle dos jogadores, desenvolveu-se uma estrutura `jogador` que armazena uma *string* $nome$ e uma $color$ cor. Por definição, apenas dois jogadores podem participar de uma partida.

```
(struct jogador (nome cor) #:mutable)
```

Além disso, são definidas as seguintes variáveis, que auxiliam na construção de uma `jogada` ($World$):

```
(define ranking empty)
(define jogadorIA1 empty) ;Define quem é o jogador IA 1
(define jogadorIA2 empty) ;Define quem é o jogador IA 2
(define jogadorHumano1 empty) ;Define quem é o jogador Humano 1
(define jogadorHumano2 empty) ;Define quem é o jogador Humano 2
(define nomeJogador1 empty) ;Define o nome do jogador humano 1
(define nomeJogador2 empty) ;Define o nome do jogador humano 2
;Variável para controlar os cliques (selecionar origem = 0 / selecionar destino = 1)
(define select 0)
(define movimentos 0) ;Variável para contar os movimentos
(define jogador-atual empty);Define quem é o jogador atual
;Lista de possibilidades de locomoção temporárias
(define possibilidades-temporarias empty)
;Posição de origem (de onde um jogador deseja fazer o movimento)
(define posicao-origem empty)
;Variável que atesta que ambos reis estão vivos (0) ou algum está morto (1)
(define king-is-dead 0)
(define pts-branco 0) ;Placar do jogador branco
(define pts-preto 0) ;Placar do jogador preto
```

A movimentação das peças é realizada através de **expressões Lambda**. São definidas duas funções principais para a definir as possibilidades de destino das peças: `unitario`, que Aplica uma única vez (unitário) as funções para cada possibilidade, filtrando com uma função de validação; e `get-recursivo-possibilidades`, que aplica recursivamente as funções em Lfuncoes para cada possibilidade.

Em suma, ambas retornam uma lista de posições (`pos`) que determinada peça em uma posição $posX$ pode ter como destino. Para tanto, recebem uma Lista de Funções (`Lfuncoes`) a ser aplicada nas coordenadas de $posX$. A diferença está no fato de que a `get-unitario-possibilidades` aplica uma única vez cada função e as valida com uma função de validação `fval`, enquanto a `get-recursivo-possibilidades` aplica cada função em `Lfuncoes` recursivamente para cada posição gerada.

Esse modelo é, na verdade, uma abstração do comportamento do cálculo das possibilidades para as peças. Uma peça como um peão, por exmeplo, faz uso de um cálculo unitário (verifica os arredores, apenas) enquanto a torre, por outro lado, faz uso de um cálculo recursivo (verifica os arredores e os arredores dos arredores, sucessivamente, até encontrar uma posição considerada inválida). Os critérios para determinar qual a próxima posição da torre difere dos critério de determinação da próxima posição do bispo, por exemplo. Nesse ínterim entra a lista de expressões **lambda**. Dessa forma, tem-se:

```
; Lista de funções para as possibilidades de locomoção do Cavalo
(define Lf-Cavalo (list (list (lambda(x)(sub1 x))  (lambda(y)(+ y 2)))
                        (list (lambda(x)(add1 x))  (lambda(y)(+ y 2)))
                        (list (lambda(x)(+ x 2))   (lambda(y)(add1 y)))
                        (list (lambda(x)(- x 2))   (lambda(y)(add1 y)))
                        (list (lambda(x)(- x 2))   (lambda(y)(sub1 y)))
                        (list (lambda(x)(+ x 2))   (lambda(y)(sub1 y)))
                        (list (lambda(x)(add1 x))  (lambda(y)(- y 2)))
                        (list (lambda(x)(sub1 x))  (lambda(y)(- y 2)))))
; Lista de funções para as possibilidades de locomoção do Bispo
(define Lf-Bispo  (list (list (lambda(x)(sub1 x))  (lambda(y)(add1 y)))
                        (list (lambda(x)(sub1 x))  (lambda(y)(sub1 y)))
                        (list (lambda(x)(add1 x))  (lambda(y)(add1 y)))
                        (list (lambda(x)(add1 x))  (lambda(y)(sub1 y)))))
; Lista de funções para as possibilidades de locomoção da Torre
(define Lf-Torre  (list (list (lambda(x)(sub1 x))  (lambda(y)y))
                        (list (lambda(x)(add1 x))  (lambda(y)y))
                        (list (lambda(x)x)         (lambda(y)(add1 y)))
                        (list (lambda(x)x)         (lambda(y)(sub1 y)))))
; Lista de funções para as possibilidades de locomoção do Peao Preto
(define Lf-Peao-P (list (list (lambda(x)(add1 x))  (lambda(y)(add1 y)))
                        (list (lambda(x)(add1 x))  (lambda(y)y))
                        (list (lambda(x)(add1 x))  (lambda(y)(sub1 y)))))
; Lista de funções para as possibilidades de locomoção do Peao Branco
(define Lf-Peao-B (list (list (lambda(x)(sub1 x))  (lambda(y)(add1 y)))
                        (list (lambda(x)(sub1 x))  (lambda(y)y))
                        (list (lambda(x)(sub1 x))  (lambda(y)(sub1 y)))))
; Lista de funções para as possibilidades de locomoção do Rei
(define Lf-Rei    (list (list (lambda(x)(sub1 x))  (lambda(y)y))
                        (list (lambda(x)(sub1 x))  (lambda(y)(add1 y)))
                        (list (lambda(x)(sub1 x))  (lambda(y)(sub1 y)))
                        (list (lambda(x)x)         (lambda(y)(add1 y)))
```

```
(list (lambda(x)x)          (lambda(y)(sub1 y)))
(list (lambda(x)(add1 x))   (lambda(y)(sub1 y)))
(list (lambda(x)(add1 x))   (lambda(y)y))
(list (lambda(x)(add1 x))   (lambda(y)(add1 y)))))
```

2.2. Jogadores e Modos de Jogo

Foi desenvolvido o jogo de Xadrez com três modos de jogo:

- Humano vs. Humano
- Humano vs. Computador
- Computador vs. Computador

O usuário seleciona o modo de jogo através de uma interface gráfica:

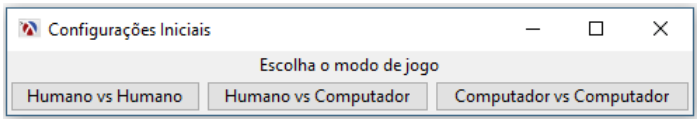

Figure 1. Escolha do modo de jogo

Para cada jogador humano, há uma janela de seleção:

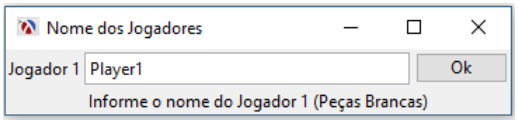

Figure 2. Obtenção do nome do jogador humano 1

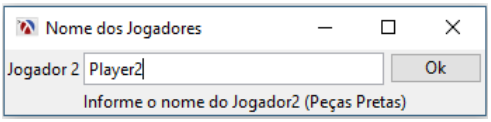

Figure 3. Obtenção do nome do jogador humano 2

Ao contrário do restante do jogo, estas janelas são construídas com a biblioteca `racket/gui`.

2.3. Interface Gráfica do Jogo

Após a seleção do modo de jogo, a interface gráfica do jogo é desenhada na tela, conforme ilustrado por 2.3.

Um destaque especial para o momento em que o jogador Humano seleciona uma determinada peça e as possibilidades são desenhadas com pontos verdes centrais em cada posição conforme ilustrado por 5.

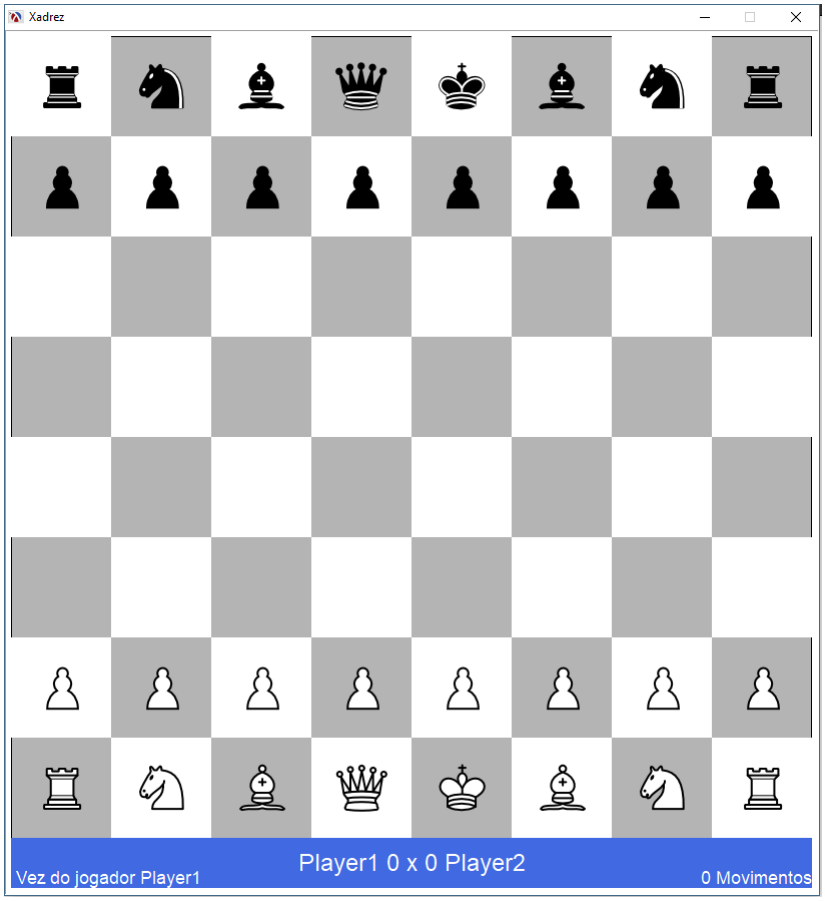

Figure 4. Interface Gráfica do Jogo

2.4. Placar Final

Durante o jogo, há uma barra inferior que mostra o placar atual. O placar é dado conforme os jogadores consomem peças do adversário.

Cada peça tem para si um peso associado, conforme sua importâcia no tabuleiro. A pontuação final é:

- Peao = 1
- Torre = 3
- Cavalo = 5
- Bispo = 7
- Rainha = 10
- Rei = 49 (soma de todas as peças do oponente + 1)

Quando o jogo termina, pode-se jogar novamente (nas mesmas configurações iniciais) através da tecla [ENTER]. A figura 6 mostra a tela final do jogo:

2.5. Inteligência Artificial

A intelência artificial desenvolvida para a resolução do jogo aplica a seguinte heurística:

1. Selecionar aleatóriamente uma peça
2. Buscar as possibilidades de destino (caso vazio, selecionar outra peça)

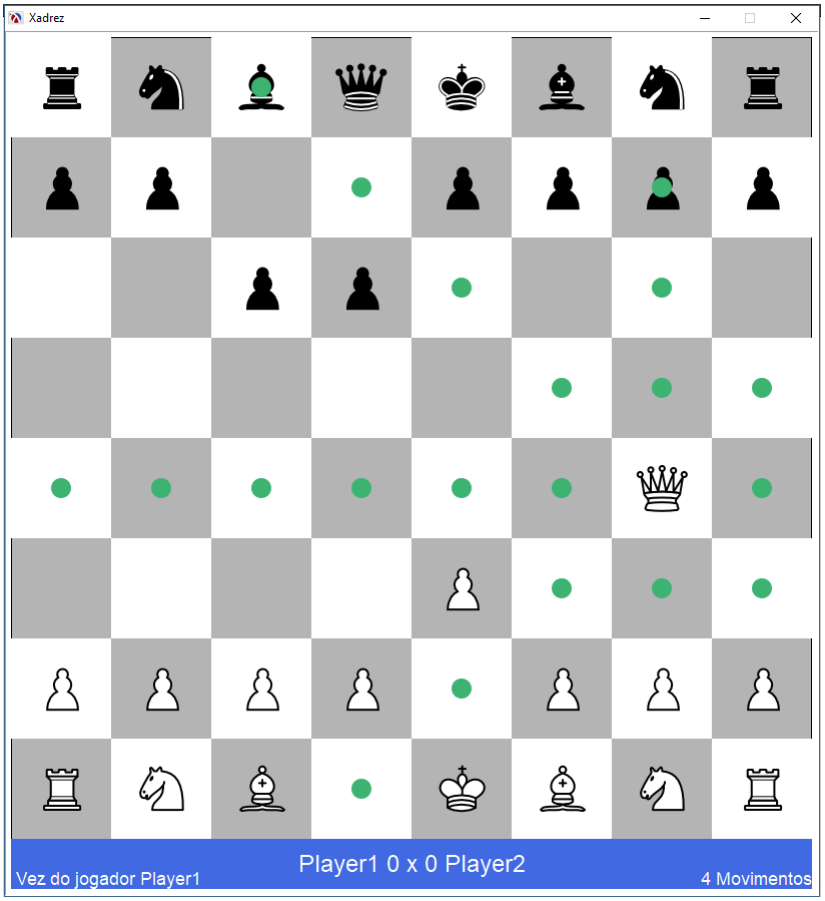

Figure 5. Possibilidades de Locomoção da Rainha Branca

3. Buscar pela posição onde se encontra a peça mais valiosa do jogador (caso não haja, mover-se aleatoriamente).

Assim, o computador tenta, dada uma peça aleatória, encontrar a posição que mais tem importância para o adversário.

Dessa forma, uma partida *Computador vs. Computador* consiste de dois jogadores aplicando tal heurística até que um deles perca o rei.

3. Análise do Algoritmo de IA

Conforme apresentado na seção anterior, fora desenvolvido um algoritmo para jogar sozinho. Para efeitos de teste, foram executadas 40 partidas no modo *Computador vs. Computador* e analisados dados referentes à:

- **Quantidade de Movimentos** totais da partida
- **Pontuação do jogador 1**
- **Pontuação do jogador 2**

De maneira geral, pôde-se construir a tabela 3:

Os resultados altos de **Movimentos** devem-se principalmente à aleatoriedade na selação das peças, o que fez o computador perder oportunidades de finalizar o jogo com maior antecedência, em um número menor de movimentos.

Figure 6. Tela de Final de Jogo

	Média	Mediana	Desvio Padrão
Movimentos	63,25	49	3,256
Pontos - Branco	52,21	37	2,35
Pontos - Preto	39,54	35	5,23

Table 1. Análise dos Dados do algoritmo de IA

References

Lang, R. Racket documentation and reference. `https://docs.racket-lang.org/`. Acessado em: Dezembro de 2017.