Instito de Matemática e Estatística
Universidade de São Paulo

Anatomia de um Motor de Xadrez

Autor:
Hugo Vinicius Mori Dantas Santana

Supervisor:
José Coelho de Pina

01 de Dezembro de 2014

Sumário

Capítulo 1

Introdução

O xadrez é um jogo de tabuleiro de natureza recreativa e competitiva para dois jogadores. Este jogo é um dos mais populares do mundo e é praticado em diversos lugares como escolas, clubes, torneios ou pela internet [39].

Desde a invenção do xadrez, os enxadristas tentam buscar a melhor estratégia e tática para vencer uma partida. Muitos pesquisadores desenvolveram teorias ou máquinas para este jogo durante muitos anos [28].

Nas décadas de 1940 e 1950, houve um súbito avanço na pesquisa sobre xadrez. Isto foi possível graças à invenção do computador. Um grande número de enxadristas, percebendo o poder do computador, iniciaram o seu trabalho relacionado ao **motor de xadrez** (*chess engine*) [28].

O primeiro torneio de xadrez foi realizado em Londres, 1851. Em especial, o primeiro torneio de motores de xadrez ocorreu na cidade de Nova York, em 1970. Desde então, estas competições são realizadas regularmente.

O objetivo deste trabalho é apresentar as teorias relacionadas ao xadrez por computador e mostrar como poderiam ser implementadas estas estratégias num computador na forma de motor de xadrez e, finalmente, mostrar as relações deste assunto com as disciplinas oferecidas no curso de bacharelado em ciência da computação.

Este texto está organizado da seguinte forma. Um breve histórico do jogo de xadrez é apresentado no capítulo 2. Já, no capítulo 3, veremos uma descrição dos componentes mais comuns de motores de um jogo de xadrez. No capítulo 4, mostraremos como é feito a comunicação entre motores de xadrez e no capítulo 5, aprofundaremos no assunto sobre representação do tabuleiro de xadrez no computador. A busca de um melhor movimento será apresentado no capítulo 6 e no capítulo 7, os métodos de avaliar uma configuração será mostrado. No capítulo 8, mostraremos uma análise de uma implementação de um motor de xadrez. As relações existentes das disciplinas de bacharelado em ciência da computação e o trabalho são listadas no capítulo 9 e, finalmente, no capítulo 10 fazemos alguns comentários finais.

Capítulo 2

História

Entre as primeiras tentativas de criar um jogador artificial de xadrez, o mais famoso foi a máquina mecânica construída pelo Baron von Kempelen em 1769. Mas, este e alguns outros desta época eram trotes e tinham um jogador humano dentro [38].

Por volta de 1890, um engenheiro espanhol, Torres y Quevedo, planejou uma verdadeira máquina que era capaz de jogar rei e torre contra rei (KR *vs* K), ou seja, um jogo composto por três peças somente no tabuleiro.

Apesar do sucesso da máquina de Torres, avanços significativos não vieram até a década de 1940. Durante esta década houve uma explosão de atividades dos engenheiros e matemáticos que, notando o poder de processamento de computadores, começaram a expressar suas ideias sobre xadrez por computador. Entre eles, Tihamer Nemes e Konrad Zuse tentaram criar máquinas mas não tiveram sucesso. Outros como Alan Turing, publicou o primeiro programa, desenvolvido no papel, que era capaz de jogar xadrez. Hoje em dia, os artigos de Adriaan de Groot (1946) e Claude Shannon (1950) são considerados referências fundamentais [28].

O primeiro programa de xadrez que venceu uma pessoa foi o *Mac Hack Six*, em 1967. No ano 1970, foi disputado o *North American Computer Chess Championship*, o primeiro torneio norte-americano de programas de xadrez e, em 1974, aconteceu o primeiro *World Computer Chess Championship*, um campeonato mundial de programas de xadrez.

Nos anos seguintes, surgiram vários programas com diferentes estratégias e algoritmos e várias competições internacionais de programas de xadrez foram disputadas e diferentes programas tornaram-se campeões mundiais. Mas, talvez o mais famoso e que marcou o seu nome na história foi o *Deep Blue*. Em maio de 1997, este programa venceu o Garry Kasparov, campeão mundial de xadrez da época [27].

Depois desta partida, alguns programas venciam e outros perdiam os jogos contra profissionais de xadrez. Até que, depois de duas vitórias convincentes dos programas em 2005 e 2006, muitos passaram a considerar que os programas de xadrez são melhores que os jogadores humanos. Então, o foco dos desenvolvimentos mudou para os programas que disputam jogos de xadrez contra outros programas [27].

Capítulo 3

Anatomia

Um **motor de xadrez** (*chess engine*) é um programa de computador capaz de decidir um movimento em uma partida de xadrez. Tal programa recebe uma **configuração** de um tabuleiro, isto é, o conjunto de casas contendo a informação de qual peça está ocupando cada casa, analisa esta informação considerando somente os movimentos válidos e retorna um movimento que é o melhor possível de acordo com algum critério [3, 30].

Um exemplo de uma iteração de um motor será mostrado por um pseudocódigo. O código abaixo mostra a função contendo o laço principal de um motor.

```
FUNCTION main()
    VAR running = true;
    VAR command;
    WHILE running == true DO
        command = readLine(); // Lê uma linha da entrada.
        IF isValid(command) DO // Verifica o comando.
            execute(command); // Executa o comando lido.
        ELSE DO
            printError(); // Imprime mensagem de erro.
```

A seguir, temos uma função que decide o que fazer dependendo do comando recebido.

```
// Recebe uma variável "command" que o motor deve executar.
FUNCTION execute(command)
    // Dependendo do comando recebido, executa tarefas diferentes.
    IF command == "uci" DO
        // Diz para o motor que o protocolo UCI deve ser usado.
        // ...

    ELSE IF command == "isready" DO
        // Comando para sincronização.
        // ...

    ELSE IF command == "ucinewgame" DO
        // Inicia um novo jogo de xadrez usando o protocolo UCI.
        // ...
```

```
15
16      ELSE IF command == "quit" DO
17          // Termina o jogo atual.
18          // ...
19
20      // ...
21
22      ELSE
23          RETURN -1; // Erro.
```

Um programa de um motor possui quatro componentes principais [1]:

Comunicação : troca de mensagens entre os programas.

Tabuleiro : representação de uma configuração de uma partida.

Busca : algoritmo para analisar os movimentos.

Avaliação : função que calcula a pontuação de uma configuração.

Opcionalmente, pode-se ter dois bancos de dados externos para auxiliar o motor. Um para o *opening book* [4], que possui movimentos considerados bons para o início do jogo e outro para o *endgame tablebases* [25], que contém pontuação (calculado pela função de avaliação) de configurações com poucas peças no tabuleiro.

Capítulo 4

Comunicação

A maioria dos motores não possui uma interface gráfica própria; somente recebe comandos pela entrada padrão e devolve o próximo movimento para a saída padrão. Para interagir com outro programa, o motor envia e recebe comandos especificados por um **protocolo de comunicação**. A utilização de um protocolo possibilita que um motor selecione o melhor movimento numa partida contra outros motores sem a necessidade de criar diferentes comandos para outras implementações de motores. Um protocolo também ajuda os desenvolvedores de um motor, pois o programador não precisa se preocupar em criar a interface gráfica, por exemplo, imagens de tabuleiros e peças, concentrando-se apenas no funcionamento do jogo [3].

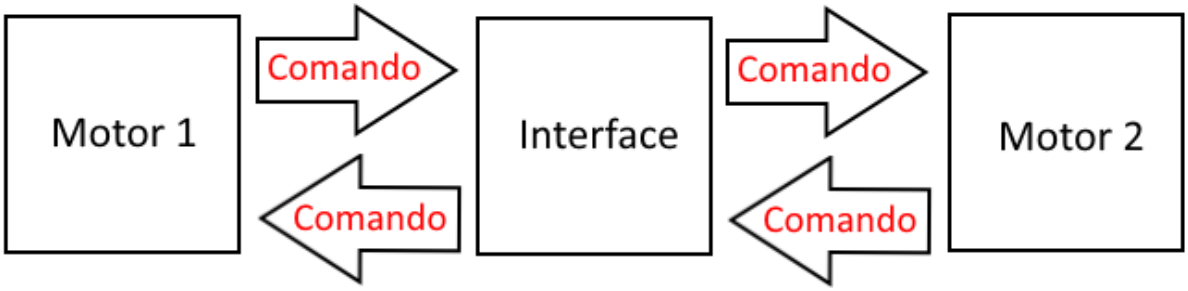

Figura 4.1: Diagrama que ilustra a comunicação entre os motores.

Entre alguns dos protocolos mais utilizados estão o *Chess Engine Communication Protocol* (*CECP*) e o *Universal Chess Interface* (*UCI*).

O *CECP* conhecido também como *XBoard Communication Protocol* foi criado para possibilitar a comunicação entre a interface gráfica *XBoard* e o motor *GNU Chess* que somente aceitava entrada e saída por linha de comando. O *XBoard*, usando este protocolo, envia comandos para o motor, recebe o resultado e mostra o movimento no seu tabuleiro gráfico.

O exemplo abaixo mostra a troca de comandos usando este protocolo.

```
1 xboard
2 new
3 e2e4
4 move c7c5
5 b1c3
6 move b8c6
```

O comando da linha 1 inicia o modo *xboard*, que aceita comandos do protocolo *CECP*. A linha 2 inicia uma nova partida. Os comandos das linhas 3 e 5 são do jogador branco, neste caso o programa *XBoard* e das linhas 4 e 6 são do jogador preto, o motor.

O *UCI* é um protocolo para comunicação criado em Novembro de 2000. Este protocolo atribui algumas tarefas, que tradicionalmente eram de responsabilidade dos motores, para a interface. Uma das tarefas é o uso de *opening book*. No início do jogo, enquanto há movimentos registrados neste banco de dados, a interface pode escolher o próximo movimento e durante esta fase, um motor pode examinar a árvore de jogo para movimentos futuros.

A seguir, há um exemplo deste protocolo.

```
1  uci
2  id name Pulse 1.5-java
3  id author Phokham Nonava
4  uciok
5  position startpos moves e2e4
6  go movetime 10
7  info depth 1 seldepth 3 score cp -17 pv a7a6 nps 0 time 1 nodes 4
8  info depth 1 seldepth 3 score cp -16 pv a7a5 nps 0 time 3 nodes 5
9  info depth 1 seldepth 3 score cp -15 pv b7b6 nps 0 time 3 nodes 6
10 info depth 1 seldepth 3 score cp -2 pv d7d6 nps 0 time 4 nodes 11
11 info depth 1 seldepth 3 score cp -1 pv d7d5 nps 0 time 4 nodes 14
12 info depth 2 seldepth 6 score cp -15 pv d7d5 b1c3 d5e4 c3e4 nps 0 time 8 nodes 82
13 info depth 2 seldepth 6 currmove a7a6 currmovenumber 5 nps 0 time 10 nodes 101
14 bestmove d7d5 ponder b1c3
15 position startpos moves e2e4 d7d5 b1c3
16 go movetime 10
17 info depth 1 seldepth 4 score cp -15 pv d5e4 c3e4 nps 0 time 1 nodes 5
18 info depth 1 seldepth 5 score cp -13 pv e7e6 nps 0 time 2 nodes 23
19 info depth 1 seldepth 5 score cp -1 pv g8f6 nps 0 time 4 nodes 49
20 info depth 2 seldepth 8 score cp -4 pv g8f6 d2d3 nps 0 time 12 nodes 161
21 info depth 2 seldepth 8 currmove e7e6 currmovenumber 2 nps 0 time 20 nodes 162
22 bestmove g8f6 ponder d2d3
```

Na linha 1, há um comando que diz para o motor usar o protocolo *UCI*. O motor que recebeu este comando deve devolver algumas informações, como está nas linhas 2 a 4. Agora, para executar um movimento, enviamos um comando `position`, linhas 5 ou 15, e, depois, pedimos para o motor adversário calcular o seu próximo movimento durante 10 milissegundos, linhas 6 ou 16. Um motor que recebe o comando `go` imprime as informações sobre as buscas executadas e, no final, devolve o melhor movimento encontrado, como nas linhas 7 a 14 ou 17 a 22.

Capítulo 5

Representação do tabuleiro

Um motor precisa representar internamente a configuração de um tabuleiro para ser usado na busca e avaliação de um movimento.

Figura 5.1: Um exemplo de uma configuração de um tabuleiro. Fonte: [2]

A **representação** ou estrutura de dados, além de conter as casas de todas as peças, deve possuir informações sobre de quem (preto ou branco) é a vez, direito de roque, a casa com possibilidade de fazer captura *en passant* e o número de movimentos relacionados a regra de 50 movimentos [10].

Um roque é um movimento especial que envolve duas peças no mesmo lance e pode ser feita somente uma vez [33]. Uma captura *en passant* é um movimento especial de captura do peão. Se houver algum peão adversário nas casas adjacentes durante o movimento de duas casas iniciais, este peão do oponente pode ser capturado [24]. A regra de 50 movimentos é uma regra tal que um jogador pode pedir um empate se

nenhuma peça foi capturada e se nenhum peão foi movimentado nos últimos 50 movimentos consecutivos [32].

Além disso, uma das funções da representação de um tabuleiro é permitir ao motor encontrar um movimento o mais eficientemente possível [18]. Para isso, um **gerador de movimento** produz uma lista de movimentos válidos. Devemos ter um gerador completamente correto para garantir o funcionamento de um jogo de xadrez.

Dentre as principais formas de representações, estão:

- Centrada nas peças;
- Centrada nas casas; e
- Híbrida.

Representação centrada nas peças

Uma **representação centrada nas peças** mantém, para cada tipo de peça, uma lista contendo as casas do tabuleiro ocupadas por tal tipo de peça.

Para a configuração apresentada na figura 5.1, temos que uma possível lista para os peões brancos seria o seguinte.

```
1 {a4, b2, c4, d2, e3, f4, g3}
```

Nesta representação, em vez de manter o tabuleiro inteiro na memória, usa-se listas para representar cada peça de xadrez: peões brancos, bispos brancos, cavalos brancos, rainha branco, rei branco, torres brancos e, analogamente, peças pretas. Cada lista é associado a uma peça com tipo e cor e contém a informação da casa que está [20].

Por exemplo, a lista de peões brancos acima pode ser implementada da seguinte maneira, usando a indexação da figura 5.2.

```
1 int whitePawns = {24, 9, 26, 11, 20, 29, 22};
```

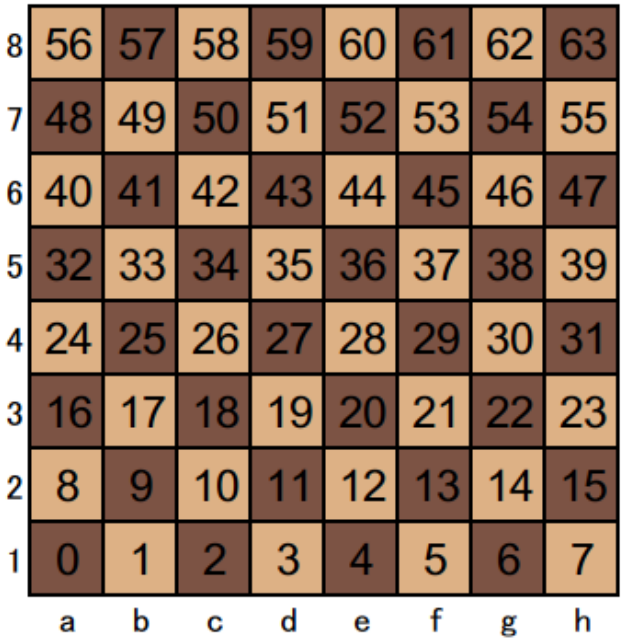

Figura 5.2: Uma possível indexação de tabuleiro.

Uma lista de peças ainda é usada em muitos motores atuais, como, por exemplo, *Pulse*, acompanhada de alguma outra estrutura para aumentar a velocidade de acesso a informações das peças. Isto é, em vez de percorrer o tabuleiro inteiro, que tem 64 casas, para encontrar alguma peça do tabuleiro, uma lista de peças possibilita o acesso direto à peça desejada.

Um tipo de representação muito usado é o *bitboard*. Para cada tipo de peça, usamos uma variável como *bit vector* de 64 bit. Esta variável está relacionada às 64 casas do tabuleiro de xadrez. Como temos 6 tipos de peças para cada lado (preto e branco), precisamos de 12 variáveis para representar todos os tipos de peças. Cada bit da variável representa se uma casa do tabuleiro está ocupada ou não. Para obter informações do tabuleiro, basta fazermos algumas operações binárias [9].

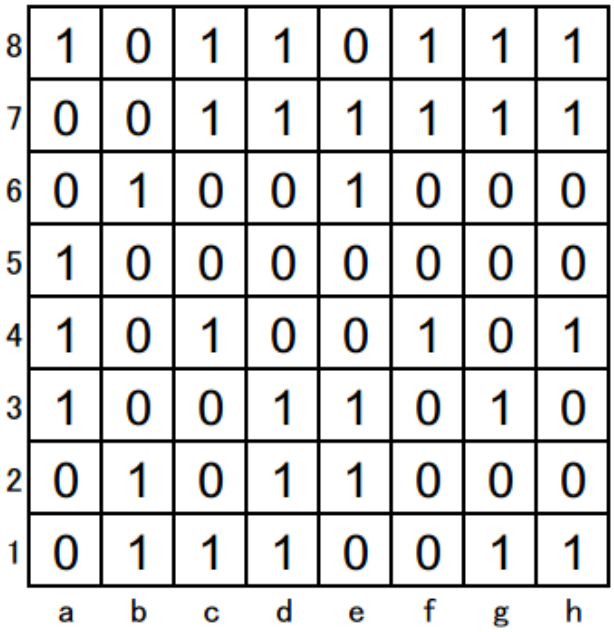

Figura 5.3: Exemplo da configuração da figura 5.1 representado por um *bitboard*.

Esta representação é usada no motor *Pulse* e é declarada da seguinte maneira.

Código 5.1: Bitboard.java:13

```
1 final class Bitboard {
2
3   // Uma variável long de 64 bit.
4   long squares = 0;
5
6   // ...
7 }
```

O *bitboard* da figura 5.3 pode ser representado pelo número binário abaixo, considerando a sequência de bits como de esquerda para direita e de baixo para cima.

```
1 1110011010110001001101010100101100000000100100000111111110110111
```

A configuração da figura 5.1 é representado usando o número abaixo.

```
1 squares = 0x73589AA580483FB7;
```

Este valor foi obtido transformando o número binário anterior para o número hexadecimal.

A vantagem de usar o *bitboard* é que, como as configurações das peças são representadas por variáveis de 64 bit, o uso da memória é pequena. Logo, as variáveis podem ser armazenadas em registradores de 64 bit e as operações binárias podem ser feitas em um ciclo do *CPU*.

Além de usar *bitboard* para representar as configurações das peças, podemos usar esta estrutura para outros propósitos como verificação de quais são as casas que estão sendo atacadas por uma peça. Por exemplo, criamos um vetor de tamanho 64 para cada tipo de peça. Cada elemento deste vetor contém uma variável de 64 bit. Cada bit desta variável vale um se a casa equivalente está sendo atacada e, caso contrário, zero.

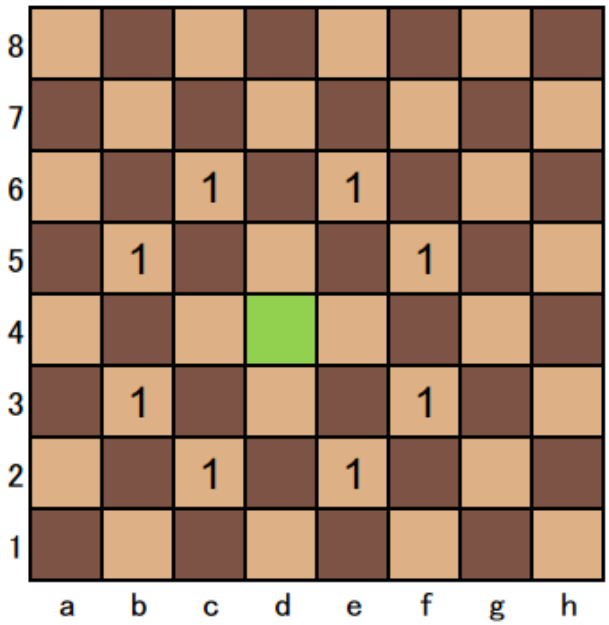

Figura 5.4: Representação gráfica do vetor `knightAttacks[27]`.

Por exemplo, na figura 5.4, a casa em destaque contém um cavalo. Neste caso usamos um vetor `knightAttacks` de tamanho 64 para representar quais casas o cavalo ataca. Cada elemento `knightAttacks[i]` deste vetor armazena quais casas o cavalo que está na casa `i` ataca. Para obter que casa o cavalo em `d4` ataca, acessamos o elemento `knightAttacks[27]`. O índice é equivalente ao da figura 5.2.

Representação centrada nas casas

Uma **representação centrada nas casas** guarda a informação, em um vetor de dimensão 1 ou 2, sobre se uma casa está ocupada e, se estiver, que peça está nesta casa [6].

O mais simples deste tipo de representação é o tabuleiro 8×8. Esta estrutura pode ser uma matriz de dimensão 8×8 que é indexado pela linha e coluna do tabuleiro ou vetor de tamanho 64 que é indexado pelo número 0 a 63. Cada elemento da matriz ou vetor possui a informação da peça que está na casa representada.

O exemplo a seguir mostra como seria a implementação da configuração da figura 5.1 usando 8×8.

Primeiro, atribuímos um valor para cada tipo de peça e depois criamos uma matriz 8×8 para armazenar as informações das peças.

```
// Atribuímos um valor para cada tipo de peça.
static final int WHITE_PAWN = 0;
static final int WHITE_KNIGHT = 1;
static final int WHITE_BISHOP = 2;
static final int WHITE_ROOK = 3;
static final int WHITE_QUEEN = 4;
static final int WHITE_KING = 5;
static final int BLACK_PAWN = 6;
static final int BLACK_KNIGHT = 7;
static final int BLACK_BISHOP = 8;
static final int BLACK_ROOK = 9;
static final int BLACK_QUEEN = 10;
static final int BLACK_KING = 11;

static final int NOPIECE = 12;

// Declaramos o tabuleiro.
int board[8][8];
```

Agora, atribuímos os valores para cada linha do tabuleiro.

```
// Os valores da matriz da linha 1 até linha 8.
board[0] = {NOPIECE, WHITE_KNIGHT, WHITE_BISHOP, WHITE_KING, NOPIECE, NOPIECE, WHITE_KNIGHT,
            WHITE_ROOK};
board[1] = {NOPIECE, WHITE_PAWN, NOPIECE, WHITE_PAWN, WHITE_QUEEN, NOPIECE, NOPIECE,
            NOPIECE};
board[2] = {WHITE_ROOK, NOPIECE, NOPIECE, WHITE_BISHOP, WHITE_PAWN, NOPIECE, WHITE_PAWN,
            NOPIECE};
board[3] = {WHITE_PAWN, NOPIECE, WHITE_PAWN, NOPIECE, NOPIECE, WHITE_PAWN, NOPIECE,
            BLACK_KNIGHT};
board[4] = {BLACK_PAWN, NOPIECE, NOPIECE, NOPIECE, NOPIECE, NOPIECE, NOPIECE, NOPIECE};
board[5] = {NOPIECE, BLACK_PAWN, NOPIECE, NOPIECE, BLACK_PAWN, NOPIECE, NOPIECE, NOPIECE};
board[6] = {NOPIECE, NOPIECE, BLACK_PAWN, BLACK_PAWN, BLACK_QUEEN, BLACK_PAWN, BLACK_PAWN,
            BLACK_PAWN};
board[7] = {BLACK_ROOK, NOPIECE, BLACK_BISHOP, BLACK_KING, NOPIECE, BLACK_BISHOP,
            BLACK_KNIGHT, BLACK_ROOK};
```

A representação usando somente o tabuleiro 8×8 apresenta problemas na eficiência para verificar se o movimento é válido, pois deve-se desconsiderar movimentos de peças para fora do tabuleiro.

Uma outra possível representação da configuração de um tabuleiro de xadrez é por meio de uma matriz com uma **moldura**. A representação é feita através de um tabuleiro 12×12. Esta representação adiciona duas linhas e duas colunas nas bordas do tabuleiro 8×8. Esta borda, que representa as casas fora do tabuleiro, é necessária para verificar se o movimento é válido. Duas colunas ou linhas, em vez de uma, são necessárias para garantir a verificação do movimento do cavalo. Na fase de geração do movimento, se o destino de uma peça for uma casa na borda, o movimento é desconsiderado.

-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	56	57	58	59	60	61	62	63	-1	-1
-1	-1	48	49	50	51	52	53	54	55	-1	-1
-1	-1	40	41	42	43	44	45	46	47	-1	-1
-1	-1	32	33	34	35	36	37	38	39	-1	-1
-1	-1	24	25	26	27	28	29	30	31	-1	-1
-1	-1	16	17	18	19	20	21	22	23	-1	-1
-1	-1	8	9	10	11	12	13	14	15	-1	-1
-1	-1	0	1	2	3	4	5	6	7	-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

Figura 5.5: Exemplo da indexação do tabuleiro 12×12. As casas com valor -1 não são válidas.

A configuração da figura 5.1 usando esta representação é apresentada a seguir.

```
// Os valores da matriz da linha 1 até linha 12.
board[0] = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1};
board[1] = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1};
board[2] = {-1, -1, NOPIECE, WHITE_KNIGHT, WHITE_BISHOP, WHITE_KING, NOPIECE, NOPIECE,
            WHITE_KNIGHT, WHITE_ROOK, -1, -1};
board[3] = {-1, -1, NOPIECE, WHITE_PAWN, NOPIECE, WHITE_PAWN, WHITE_QUEEN, NOPIECE, NOPIECE,
            NOPIECE, -1, -1};
board[4] = {-1, -1, WHITE_ROOK, NOPIECE, NOPIECE, WHITE_BISHOP, WHITE_PAWN, NOPIECE,
            WHITE_PAWN, NOPIECE, -1, -1};
board[5] = {-1, -1, WHITE_PAWN, NOPIECE, WHITE_PAWN, NOPIECE, NOPIECE, WHITE_PAWN, NOPIECE,
            BLACK_KNIGHT, -1, -1};
board[6] = {-1, -1, BLACK_PAWN, NOPIECE, NOPIECE, NOPIECE, NOPIECE, NOPIECE, NOPIECE,
            NOPIECE, -1, -1};
board[7] = {-1, -1, NOPIECE, BLACK_PAWN, NOPIECE, NOPIECE, BLACK_PAWN, NOPIECE, NOPIECE,
            NOPIECE, -1, -1};
board[8] = {-1, -1, NOPIECE, NOPIECE, BLACK_PAWN, BLACK_PAWN, BLACK_QUEEN, BLACK_PAWN,
            BLACK_PAWN, BLACK_PAWN, -1, -1};
board[9] = {-1, -1, BLACK_ROOK, NOPIECE, BLACK_BISHOP, BLACK_KING, NOPIECE, BLACK_BISHOP,
            BLACK_KNIGHT, BLACK_ROOK, -1, -1};
board[10] = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1};
board[11] = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1};
```

Existe, ainda, uma representação mais eficiente para a geração do movimento, o **tabuleiro 0x88**. 0x88 é o número hexadecimal `88` (`136` em decimal e `10001000` em binário) [5].

Nesta representação, consideramos um tabuleiro de tamanho 128 e utilizamos 8 bit para representar uma casa. Cada casa é indexada por dois números hexadecimais. O primeiro algarismo (4 bits mais significativo)

representa a linha da casa e o segundo algarismo (4 bits menos significativo) representa a coluna.

Por exemplo, a casa terceira linha (2) e oitava coluna (7) tem índice 27 em hexadecimal e a casa terceira linha (2) e nona coluna (8), que é inválida, tem índice 28. Assim, temos 128 índices possíveis, do número 00 até 7F, onde somente 64 são válidos.

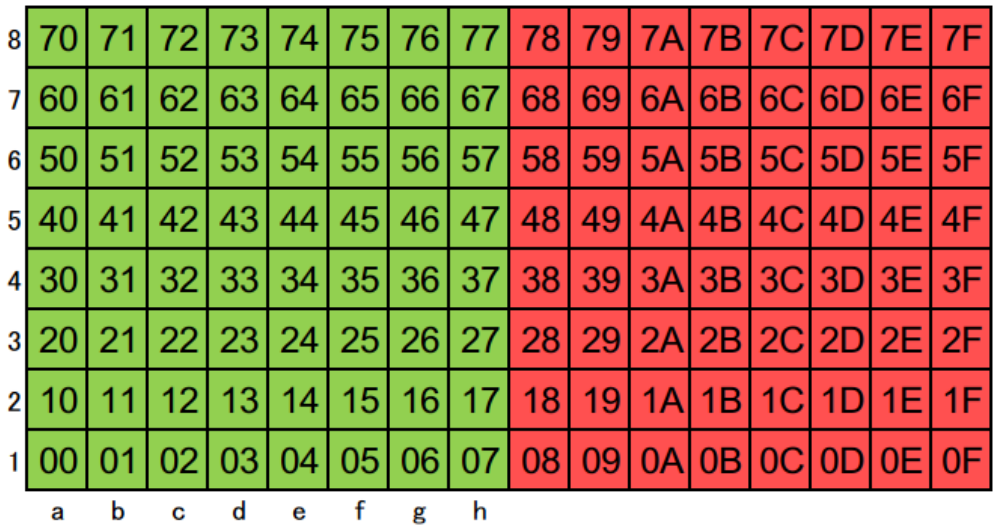

Figura 5.6: Exemplo da indexação do tabuleiro 0x88. A parte verde é válida e a vermelha, não.

Esta representação é chamada de 0x88 pois, para verificar se uma casa é válida, basta fazer um cálculo simples usando este número 88. Isto é, ao fazermos a conjunção lógica do índice da casa com o valor 10001000, se a casa é válida, obtemos o valor zero, caso contrário, o resultado é diferente de zero [36, 37].

Por exemplo, para verificar se a casa terceira linha e oitava coluna, ou seja, 27 em hexadecimal ou 00100111 em binário é válida, calculamos o seguinte.

```
1 00100111 & 10001000
```

O resultado deste cálculo é 00000000 e, portanto, esta casa está dentro do tabuleiro. Agora, para verificar se a casa terceira linha e nona coluna, ou seja, 28 ou 00101000, fazemos o cálculo abaixo.

```
1 00101000 & 10001000
```

O resultado do cálculo acima é 00001000 e, portanto, esta casa está fora do tabuleiro.

Representação híbrida

Devido a diferentes algoritmos presentes no motor, por eficiência, é comum usar mais de uma representação do tabuleiro. Por exemplo, a representação por *bitboard* muitas vezes mantém um tabuleiro 8×8 para

determinar a peça que está numa casa específica enquanto que representações centradas nas casas mantém uma lista de peças para evitar varrer o tabuleiro inteiro para verificar se um movimento é válido [10].

Capítulo 6

Busca

Para selecionar o próximo movimento, o motor deve examinar os movimentos possíveis para a configuração dada e dentre estes escolher um melhor movimento.

Uma **busca** é feita numa estrutura de árvore para examinar os movimentos. O nó desta árvore representa as configurações e as arestas, os movimentos. Então, o objetivo da busca é encontrar um caminho da raiz, que é a configuração atual do tabuleiro, até a folha com a melhor pontuação, considerando que os dois jogadores sempre escolhem o melhor movimento.

Para selecionar um movimento dentre as possíveis o motor utiliza uma função de avaliação. Uma função de avaliação recebe uma configuração e retorna um valor inteiro. Quanto maior esta pontuação melhor é a configuração para as brancas e quanto menor melhor é a configuração para as pretas.

O pseudocódigo abaixo mostra um exemplo de uma função de busca.

```
1  // Recebe a configuração atual: position.
2  // Recebe a profundidade a ser examinada: depth.
3  // Devolve a pontuação da configuração após executar o movimento escolhido.
4  FUNCTION search (position, depth)
5      VAR i;
6      VAR bestValue; // Melhor pontuação.
7      VAR value; // Pontuação atual.
8      VAR moves[MAXWIDTH]; // Movimentos possíveis para a configuração atual.
9
10     IF depth == 0 THEN
11         RETURN evaluate(position); // Avalia a configuração.
12
13     bestValue = -infinite;
14     FOR i = 0 TO sizeof(moves) DO // Para cada movimento possível.
15         makeMove(moves[i]); // Executa o movimento.
16         value = -search(moves[i], depth-1); // Busca recursivamente.
17         IF value > bestValue THEN
18             bestValue = value; // Armazena a melhor pontuação atual.
19             saveMove(moves[i]); // Armazena o melhor movimento atual.
20         undoMove(moves[i]);
21     RETURN bestValue; // Devolve a pontuação do melhor movimento.
```

Existem, pelo menos, duas maneiras mais utilizadas para percorrermos a árvore do jogo a fim de encontrar um melhor movimento, as buscas minimax e alfa-beta. Em particular, o motor *Pulse*, que é o principal programa analisado neste trabalho, utiliza o algoritmo alfa-beta para as buscas.

Busca minimax

Para uma árvore de jogo de duas pessoas, o melhor caminho ou sequências de movimentos pode ser encontrado usando a busca em profundidade minimax [34]. A **minimax** é um método para minimizar a perda máxima possível [29, 16]. O funcionamento do algoritmo é muito simples:

- O jogador branco escolhe o movimento, ou seja, a aresta, que tem como destino o nó com a pontuação mais alta.
- O jogador preto escolhe o movimento que tem como destino o nó com a pontuação mais baixa.

Portanto, dada uma configuração, se temos uma pontuação maior que zero, o jogador branco está melhor no jogo, se temos uma pontuação menor que zero, o preto está bem e se a pontuação é igual a zero, está empatado.

O exemplo a seguir mostra como este algoritmo funciona. Suponha que temos tempo suficiente para buscar dois movimentos somente, ou seja, podemos expandir dois níveis da árvore.

Figura 6.1: Raiz da árvore.

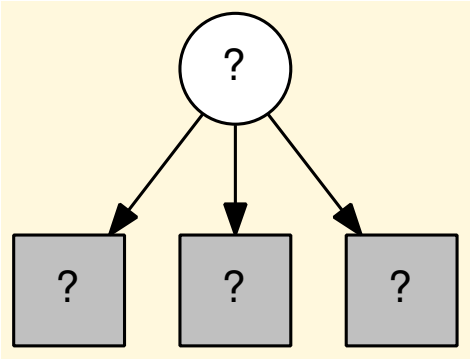

Figura 6.2: Expansão de um movimento.

Na figura 6.1 temos a raiz da árvore, isto é, a configuração atual do tabuleiro. Não sabemos ainda a pontuação deste nó. Na figura 6.2 aparecem três nós que é o resultado de três movimentos distintos executados. A pontuação continua desconhecida.

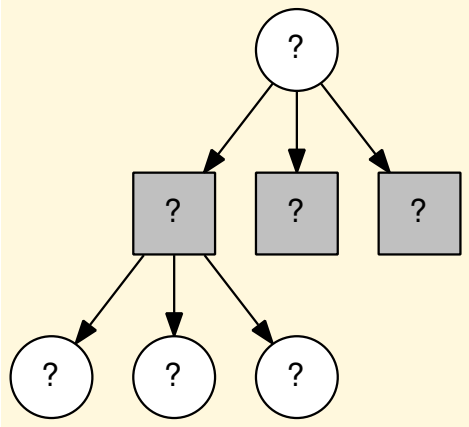

Figura 6.3: Expansão de dois movimentos.

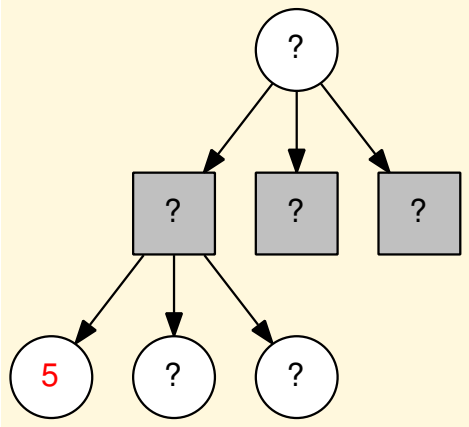

Figura 6.4: Cálculo da pontuação do nó.

Na figura 6.3 expandimos os primeiros nós terminais. Agora iniciamos o cálculo das pontuações até a raiz. Na figura 6.4 calculamos a pontuação do nó mais a esquerda e obtemos o valor `5`.

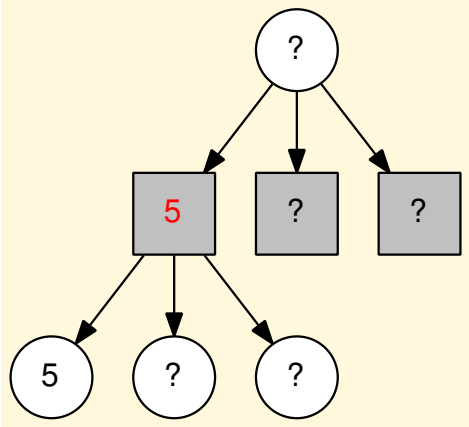

Figura 6.5: Atualiza a pontuação de um nó.

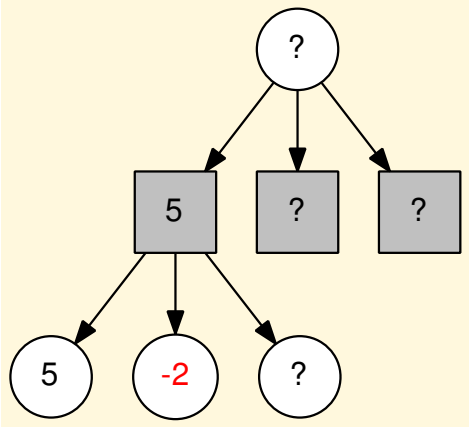

Figura 6.6: Cálculo da pontuação do próximo nó.

Na figura 6.5 atualizamos o valor do nó pai como sendo o melhor valor atual.

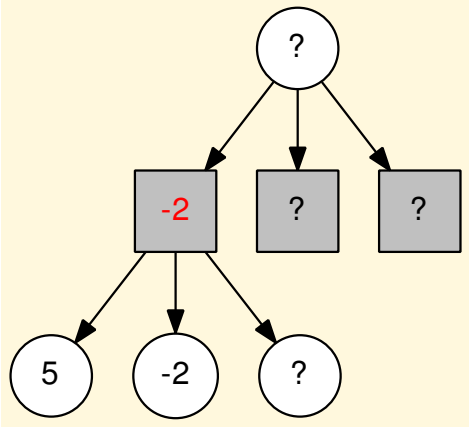

Figura 6.7: Escolhe a melhor pontuação e atualiza.

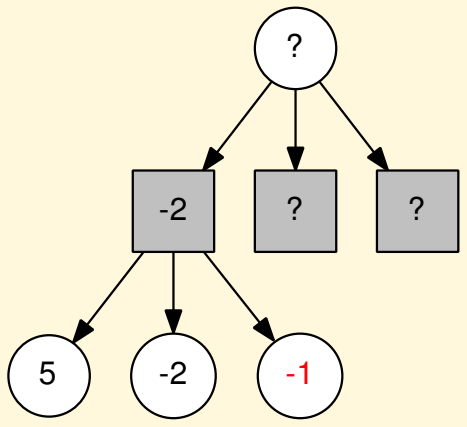

Figura 6.8: Cálculo da pontuação do próximo nó.

Na figura 6.7 comparamos o valor `5` e `-2` e escolhemos `-2` como melhor valor, pois o jogador preto deve minimizar a pontuação.

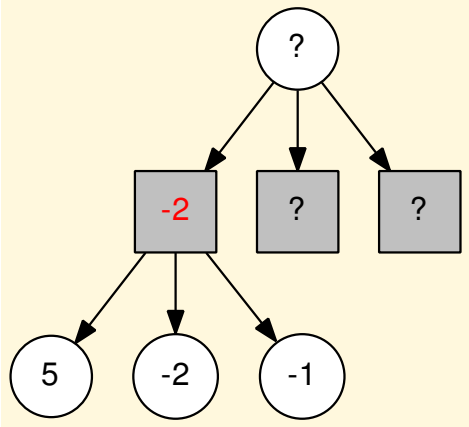

Figura 6.9: Não atualiza a pontuação do nó.

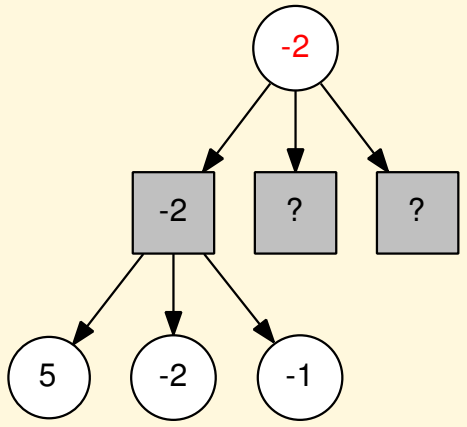

Figura 6.10: Atualização da pontuação da raiz.

Na figura 6.9 comparamos o valor `-2` e `-1` e a atualização do valor não ocorre pois `-2 < -1`. Na figura 6.10 atualizamos a pontuação do nó raiz, o que significa que a configuração atual tem pontuação `-2`. Isto significa que o branco está melhor que o preto.

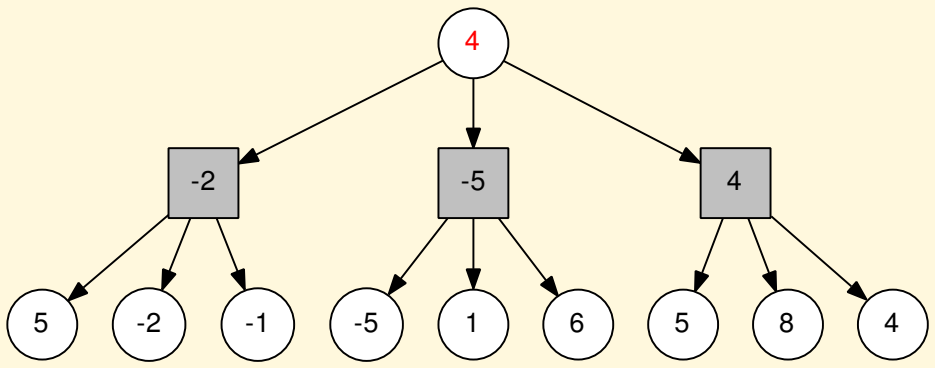

Figura 6.11: Resultado final.

Finalmente, expandindo todos os nós, calculando as pontuações e escolhendo valores segundo os critérios mostrados acima obtemos a árvore final da figura 6.11. A pontuação da raiz vale `4`, pois o branco escolhe o movimento que leva a uma configuração com maior pontuação e `max{-2, -5, 4} = 4`. Portanto, o melhor movimento é a aresta que está conectada ao nó com valor `4` e podemos supor que o jogador preto (adversário) deve escolher a aresta que liga o nó circular com valor `4` como próximo movimento.

Infelizmente, a busca minimax para o jogo de xadrez é muito custoso, pois precisamos visitar todos nós. Para uma árvore com exatamente W arestas para cada nó, existem W^H nós terminais, onde H é a altura. O jogo de xadrez produz uma árvore com valor de W aproximadamente a 35 [28]. Devido ao tamanho da árvore, não é possível fazer a busca de todos os movimentos, logo, é definido um limite, chamado de horizonte.

O algoritmo de busca que veremos a seguir, comprovadamente, gasta uma quantidade de espaço não superior ao do presente algoritmo. Isto permite que o algoritmo de busca examine nós mais profundos da árvore do jogo e assim escolher um melhor movimento.

Algoritmo alfa-beta

A definição do limite ainda não é suficiente pois a busca exaustiva dos nós é impossível. Mas, felizmente, podemos reduzir esta busca, pois alguns nós podem ser desconsiderados.

O **algoritmo alfa-beta** é um aprimoramento significativo da busca minimax [7]. Este algoritmo mantém dois valores, **alfa** e **beta**. Estes valores representam o mínimo valor para o jogador que tenta maximizar a pontuação e o máximo valor que o jogador que tenta minimizar deve respeitar, respectivamente. Estes valores podem ser usados para provar que certos movimentos não influenciam nos resultados das buscas e portanto podem ser desconsiderados (podados) [28].

O exemplo a seguir mostra como funciona este algoritmo.

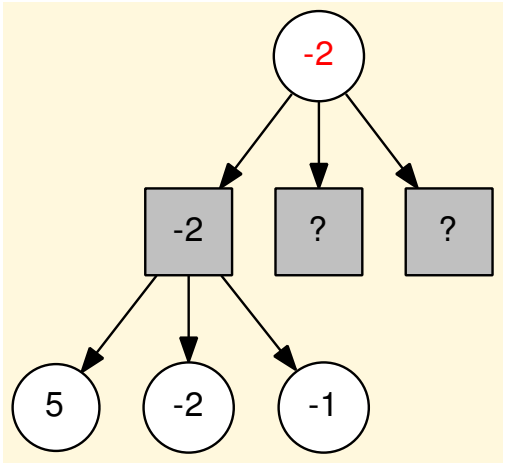

Figura 6.12: Estado inicial do exemplo do algoritmo alfa-beta.

Inicialmente, considere a árvore da figura 6.12. Neste momento, temos a garantia de que a pontuação dos movimentos escolhidos a seguir sempre será maior ou igual a -2. Logo, definimos $\alpha = -2$.

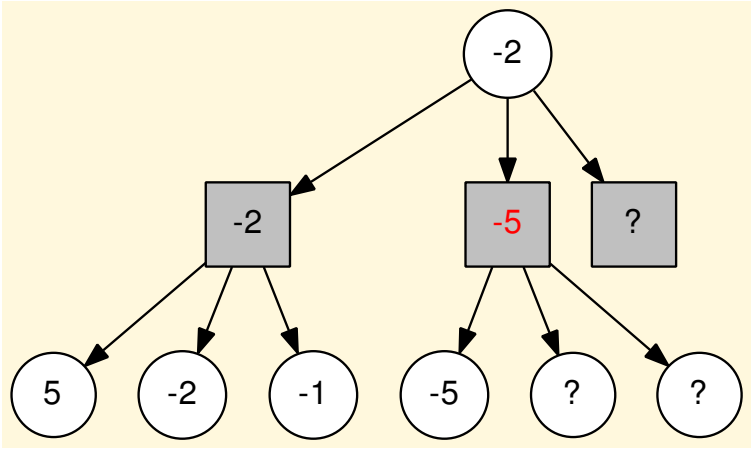

Figura 6.13: Próximo estado do exemplo do algoritmo alfa-beta.

Neste momento (figura 6.13), o jogador preto encontra um nó com valor -5.

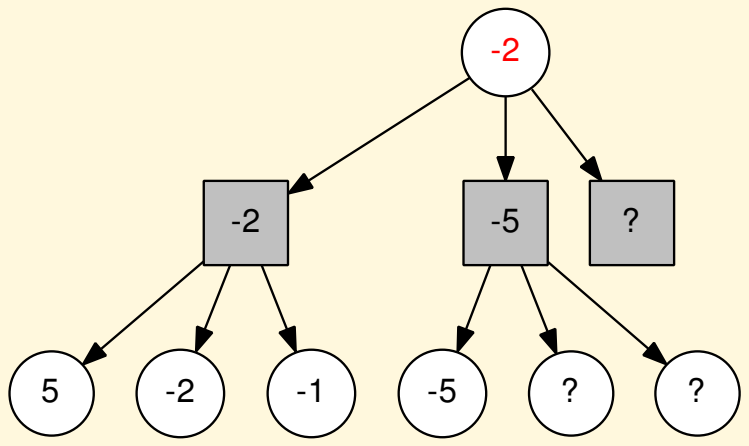

Figura 6.14: Desconsiderando uma subárvore.

Como esta pontuação é menor que $\alpha = -2$ e este valor não é favorável ao jogador branco, desconsideramos esta subárvore, voltamos um movimento (6.14) e seguimos a busca em outra parte da árvore.

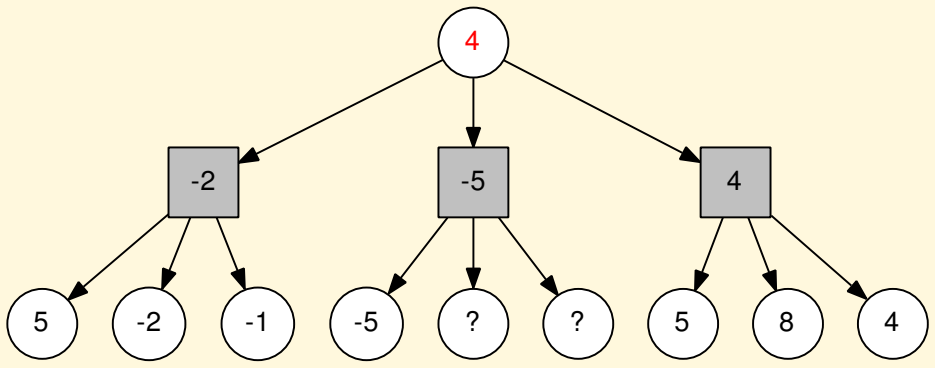

Figura 6.15: Estado final da busca.

No final, obtemos o mesmo valor encontrado pela busca minimax sem ter calculado valores de dois nós. Analogamente, podemos definir o valor beta para desconsiderar ou podar algumas subárvores.

A melhoria do algoritmo alfa-beta pode ser considerável. Se uma árvore de busca minimax tem x nós, uma árvore de alfa-beta pode ter aproximadamente $\sqrt{x}$ nós [7]. Mas, a quantidade de nós que pode ser desconsiderado depende de como é ordenado a árvore de busca [34]. Se sempre escolhêssemos para analisar o melhor movimento primeiro, poderíamos eliminar a maioria dos nós. Mas, não conhecemos qual movimento é melhor sempre. Analogamente, se escolhêssemos o pior movimento, nenhum nó seria eliminado. Por este motivo, a ordenação dos nós é importante para uma busca [34].

A seguir apresentamos como é implementado este algoritmo de busca no motor *Pulse*.

A busca está implementada na classe `Search`. Este motor utiliza o algoritmo alfa-beta e busca em profundidade iterativa para analisar os movimentos até uma certa profundidade. Quando a busca encontra a profundidade desejada, é executada uma pequena busca chamada de repouso e finalmente devolve o próximo

movimento.

O algoritmo alfa-beta está implementado no método `search` que é chamado pelo método `searchRoot`.

Código 6.1: Search.java:423

```
final class Search implements Runnable {
    // ...
    // Método que executa a busca a partir de uma determinada raiz.
    private void searchRoot(int depth, int alpha, int beta) {
      // ...
    }

    // O método searchRoot chama search recursivamente.
    private int search(int depth, int alpha, int beta, int ply) {
      // ...
    }

    // ...
}
```

Abaixo apresentamos o método `search` que executa uma iteração de busca.

Código 6.2: Search.java:447

```
final class Search implements Runnable {
    // ...
    private int search(int depth, int alpha, int beta, int ply) {
        // ...
        // Inicialização.
        int bestValue = -Value.INFINITE;
        int searchedMoves = 0;
        boolean isCheck = position.isCheck();

        // Gera os movimentos possíveis para a configuração "position"
        // e faz a busca até a profundidade "depth".
        MoveList<MoveEntry> moves = moveGenerators[ply].getMoves(position, depth, isCheck);
        for (int i = 0; i < moves.size; ++i) {
          int move = moves.entries[i].move;
          int value = bestValue;

          position.makeMove(move);
          if (!position.isCheck(Color.opposite(position.activeColor))) {
            ++searchedMoves;
            // Faz a busca recursivamente.
            value = -search(depth - 1, -beta, -alpha, ply + 1);
          }
          position.undoMove(move);

          // Se receber um comando para parar, devolve o melhor valor encontrado.
          if (abort) {
            return bestValue;
          }

          // Poda.
          if (value > bestValue) {
            bestValue = value;
```

```
33
34            // Se encontrar um valor melhor, atualiza.
35            if (value > alpha) {
36              alpha = value;
37              savePV(move, pv[ply + 1], pv[ply]);
38
39              // Verifica se o valor é
40              // maior que beta.
41              if (value >= beta) {
42                // Se for maior exclui este.
43                break;
44              }
45            }
46          }
47        }
48    }
49    // ...
50 }
```

Para melhorar a eficiência da busca, existem alguns aprimoramentos. A seguir apresentamos algumas destas técnicas.

Tabela de transposição

Os motores de xadrez, durante as buscas, se deparam com uma mesma configuração do tabuleiro várias vezes, ao percorrer a árvore do jogo de maneiras diferentes, através de sequências de movimentos diferentes. Este efeito é chamado de **transposição** [23].

Uma **tabela de transposição** é um banco de dados que contém as informações das buscas anteriores. Estas informações podem ser: que configuração foi analisado, qual foi a profundidade do nó e o que foi feito, isto é, qual foi o movimento escolhido. Este movimento pode não ser a melhor para ser executado no momento, mas pode ser usado na ordenação dos movimentos.

Sempre que a busca encontra uma configuração repetida, ela consulta a tabela e decide o que fazer a seguir.

Busca em profundidade iterativa

A **busca em profundidade iterativa** é uma estratégia de busca que facilita o gerenciamento de tempo e ordenação dos movimentos.

A busca é feita da seguinte maneira. Inicialmente, é feito a busca em profundidade somente considerando os nós de altura um. A seguir incrementa esta altura e executa a próxima iteração de busca. Este processo é repetido até que o tempo determinado para a busca esgote. Se o tempo esgotar antes de completar a análise de todos os nós da altura atual, podemos voltar para a altura da iteração anterior e escolher como o próximo movimento o melhor desta iteração e, na próxima busca, continuamos de onde paramos [14, 28].

No *Pulse*, a busca em profundidade iterativa é executada no método `run`, que é o primeiro método chamado pela componente busca.

Código 6.3: Search.java:291

```
1  final class Search implements Runnable {
2    // ...
3
4   public void run() {
5    // ...
6    // Inicia a busca em profundidade interativa
7      for (int depth = initialDepth; depth <= searchDepth; ++depth) {
8       // Profundidade atual.
9       currentDepth = depth;
10      currentMaxDepth = 0;
11      protocol.sendStatus(false, currentDepth, currentMaxDepth, totalNodes, currentMove,
            currentMoveNumber);
12
13      searchRoot(currentDepth, -Value.INFINITE, Value.INFINITE);
14
15      // Faz a ordenação das arestas para que, na próxima iteração, começamos
16      // a análise pelo melhor movimento.
17      rootMoves.sort();
18
19      // Verifica se deve parar.
20      checkStopConditions();
21
22      // Se deve parar, para.
23      if (abort) {
24        break;
25      }
26     } // Próxima iteração.
27
28  // ...
29 }
```

Janela de aspiração

Uma **janela de aspiração** é uma técnica que reduz o espaço de busca do algoritmo alfa-beta. Antes de cada iteração da busca, definimos os valores `alfa` e `beta` como: pontuação da iteração anterior menos valor do peão e pontuação da iteração anterior mais valor do peão, respectivamente. Assim, como este novo intervalo `[alfa, beta]` é um subconjunto do intervalo original alfa-beta, pois o valor do peão é sempre menor ou igual que a iteração anterior, a análise necessária durante a busca diminui [28, 8].

Busca de repouso

Muitos motores, ao final da busca, executam uma análise adicional chamada **busca de repouso**. O propósito desta busca é analisar somente as configurações relativamente em repouso. Estes são as configurações em que podemos ter avaliação precisa sem a necessidade de buscas a mais, ou seja, configurações em que não

temos movimentos como cheque, promoções ou captura. Parar a busca na profundidade desejada e avaliar o nó diretamente pode ser perigoso.

Para mostrar o benefício desta busca, considere o seguinte exemplo. O último movimento que a busca encontrou foi capturar um peão com a rainha. Se a busca parar neste momento e avaliar esta configuração, devemos ter uma pontuação boa, pois capturamos um peão. Mas, se existisse um próximo movimento do adversário que captura a rainha, o ganho total seria negativo. Para evitar que ocorra situações como a do exemplo, usamos a busca de repouso [28, 22].

No *Pulse*, se a busca chegou na profundidade desejada, inicia-se a busca de repouso.

Código 6.4: Search.java:425

```
final class Search implements Runnable {
  // ...
  private int search(int depth, int alpha, int beta, int ply) {
    // Se encontramos a profundidade destino,
    // depth <= 0, iniciamos a busca de repouso.
    if (depth <= 0) {
      return quiescent(0, alpha, beta, ply);
    }

    // ...
}
```

Capítulo 7

Avaliação

Uma **função de avaliação** é usada para heuristicamente determinar o **valor** ou a **pontuação** de uma configuração, isto é, quanto mais positivo melhor para brancas e quanto mais negativo melhor para pretas. Esta função considera alguns critérios para calcular este valor [13].

No *Pulse*, a função de avaliação está definida na classe `Evaluation`. Para calcular a pontuação de uma configuração, o motor executa o método `evaluate(position)`, considerando os critérios material e mobilidade.

Código 7.1: Evaluation.java:9

```
1 final class Evaluation {
2     //...
3     int evaluate(Position position) {
4         // ...
5     }
6     // ...
7 }
```

Nas seções seguintes, apresentamos alguns destes critérios.

Material

O critério **material** é a diferença da soma dos valores das peças de cada lado (branco e preto). Este critério é o que tem peso maior na pontuação da configuração. O valor de cada peça foi determinado pela experiência humana e aprendizagem e os mais aceitos valem 1, 3, 3, 5 e 9 para peão, cavalo, bispo, torre e rainha, respectivamente [35].

Em xadrez, possuir certos conjuntos de peças pode ser considerado bom ou ruim. A função de avaliação deve incrementar ou decrementar a pontuação da configuração ao encontrar tais conjuntos [15].

Estes conjuntos podem ser:

- Incrementar a pontuação se possuir um par de bispos;
- Incrementar a pontuação da torre se existir poucos peões no tabuleiro;
- Incrementar a pontuação do cavalo quando existir muitos peões no tabuleiro;
- Decrementar a pontuação se possuir um par de torres;
- Decrementar a pontuação se possuir um par de cavalos;
- Decrementar a pontuação dos peões em frente das torres e incrementar os peões centrais; e
- Decrementar a pontuação se não possuir peões.

No *Pulse*, é definido o peso do critério material como abaixo.

Código 7.2: Evaluation.java:13

```
//...
final class Evaluation {
    // ...

    // Peso do material.
    static int materialWeight = 100;

    // ...
}
```

As peças possuem os valores a seguir.

Código 7.3: PieceType.java:27

```
final class PieceType {
    // ...
    static final int PAWN_VALUE = 100;
    static final int KNIGHT_VALUE = 325;
    static final int BISHOP_VALUE = 325;
    static final int ROOK_VALUE = 500;
    static final int QUEEN_VALUE = 975;
    static final int KING_VALUE = 20000;
    // ...
}
```

Abaixo, apresentamos o método que calcula o valor do material. Se um jogador possuir um par de bispos, a pontuação é melhor.

Código 7.4: Evaluation.java:48

```
final class Evaluation {
    // ...
    // Calcula e devolve o valor do material da configuração "position".
    private int evaluateMaterial(int color, Position position) {
        //...

        // Calcula o valor do material considerando
        // os pesos de cada tipo de peça.
        int material = position.material[color];
```

```
10
11          // Adiciona um bônus se possuir um par de bispos.
12          if (position.pieces[color][PieceType.BISHOP].size() >= 2) {
13            material += 50;
14          }
15
16          return material;
17      }
18      // ...
19  }
```

Mobilidade

O critério **mobilidade** considera, para uma dada configuração, os movimentos válidos possíveis que um jogador pode escolher. Em xadrez, quanto mais movimentos forem possíveis, uma configuração é considerada forte [17].

O cálculo de mobilidade, muitas vezes, não é uma simples soma dos movimentos possíveis. Geralmente, é calculado peça por peça e pode ter valores diferentes para cada tipo de peça. Por exemplo, bispos e cavalos são melhores que torres no início do jogo, movimentos para frente são melhores que para trás ou movimentos verticais são melhores que horizontais.

No *Pulse*, o peso do critério mobilidade é definido como mostramos a seguir.

Código 7.5: Evaluation.java:14

```
1  final class Evaluation {
2      // ...
3
4      // Peso da mobilidade.
5      static int mobilityWeight = 80;
6
7      // ...
8  }
```

O cálculo segue abaixo. Para cada tipo de peça é calculada a pontuação separadamente, pois cada peça tem movimentos diferentes.

Código 7.6: Evaluation.java:48

```
1  final class Evaluation {
2    // ...
3    // Calcula o valor da mobilidade.
4    private int evaluateMobility(int color, Position position) {
5        assert Color.isValid(color);
6        assert position != null;
7
8        // Para cada tipo de peça, calcula diferentes valores.
9        int knightMobility = 0;
10       for (long squares = position.pieces[color][PieceType.KNIGHT].squares; squares != 0;
             squares = Bitboard.remainder(squares)) {
11         int square = Bitboard.next(squares);
```

```
12          knightMobility += evaluateMobility(color, position, square, Square.knightDirections);
13        }
14
15        int bishopMobility = 0;
16        for (long squares = position.pieces[color][PieceType.BISHOP].squares; squares != 0;
              squares = Bitboard.remainder(squares)) {
17          int square = Bitboard.next(squares);
18          bishopMobility += evaluateMobility(color, position, square, Square.bishopDirections);
19        }
20
21        int rookMobility = 0;
22        for (long squares = position.pieces[color][PieceType.ROOK].squares; squares != 0;
              squares = Bitboard.remainder(squares)) {
23          int square = Bitboard.next(squares);
24          rookMobility += evaluateMobility(color, position, square, Square.rookDirections);
25        }
26
27        int queenMobility = 0;
28        for (long squares = position.pieces[color][PieceType.QUEEN].squares; squares != 0;
              squares = Bitboard.remainder(squares)) {
29          int square = Bitboard.next(squares);
30          queenMobility += evaluateMobility(color, position, square, Square.queenDirections);
31        }
32
33        // Devolve a soma ponderada dos valores acima.
34        return knightMobility * 4
35            + bishopMobility * 5
36            + rookMobility * 2
37            + queenMobility;
38    }
39
40  // ...
41  }
```

Tabela de peça-casa

O critério **tabela de peça-casa** utiliza uma tabela para cada peça de cada cor e, para cada casa do tabuleiro, define um valor. A pontuação deste critério é a soma destes valores [21].

Apresentamos um exemplo de tabela para os peões a seguir.

8	0	0	0	0	0	0	0	0
7	50	50	50	50	50	50	50	50
6	10	10	20	30	30	20	10	10
5	5	5	10	25	25	10	5	5
4	0	0	0	20	20	0	0	0
3	5	-5	-10	0	0	-10	-5	5
2	5	10	10	-20	-20	10	10	5
1	0	0	0	0	0	0	0	0
	a	b	c	d	e	f	g	h

Figura 7.1: Um exemplo de uma tabela para peões.

Outro exemplo, da tabela para os cavalos, é mostrado abaixo.

8	-50	-40	-30	-30	-30	-30	-40	-50
7	-40	-20	0	0	0	0	-20	-40
6	-30	0	10	15	15	10	0	-30
5	-30	5	15	20	20	15	5	-30
4	-30	0	15	20	20	15	0	-30
3	-30	5	10	15	15	10	5	-30
2	-40	-20	0	5	5	0	-20	-40
1	-50	-40	-30	-30	-30	-30	-40	-50
	a	b	c	d	e	f	g	h

Figura 7.2: Um exemplo de uma tabela para cavalos.

Estrutura de peões

O critério **estrutura de peões** considera todas as casas dos peões no tabuleiro, excluindo outras peças. Existem muitas ideias relacionadas a essa estrutura como, por exemplo, peão solitário, duplo ou ilhas, entre outros. Na avaliação, quando uma configuração possui algumas das estruturas consideradas boas, a pontuação é incrementada [19].

Na figura 7.3, apresentamos um exemplo de ilhas de peões.

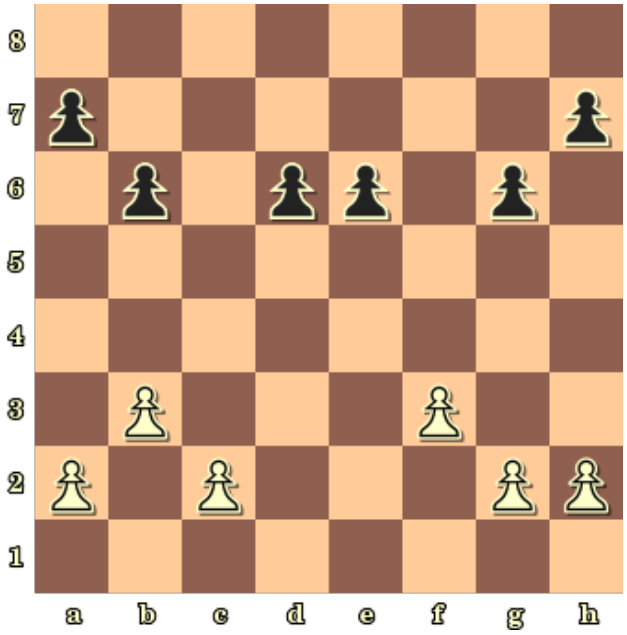

Figura 7.3: Um exemplo de uma estrutura de ilhas de peões.

Neste caso, as brancas possuem duas ilhas e as pretas, três. Na estrutura de ilhas de peões, o lado que tem menos ilhas é considerado melhor.

Controle central

O critério **controle central** considera o controle ou a ocupação do centro do tabuleiro. Existem duas regiões consideradas como centro: o **centro** (*center*) e o **centro estendido** (*extended center*) [11]. Destacamos estas casas na figura 7.4. A figura do lado esquerdo mostra o centro e a do lado direito, o centro estendido.

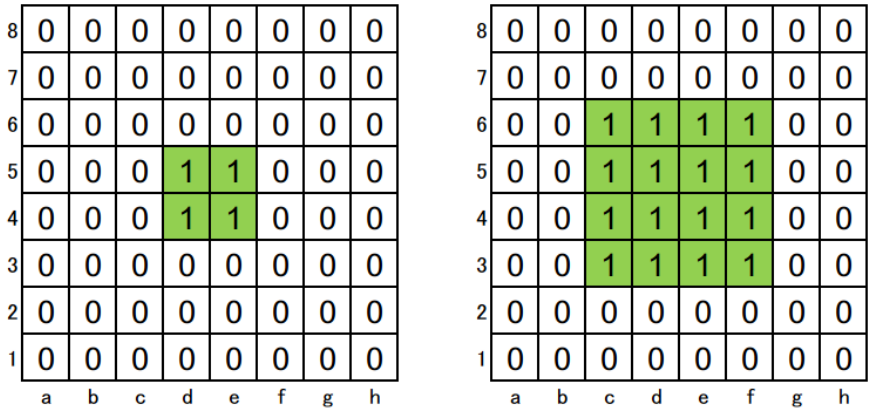

Figura 7.4: Um exemplo de centros.

Na avaliação, o controle central é considerado junto a mobilidade e tabela de peça-casa. As peças ocupando as casas centrais do tabuleiro tem peso maior no cálculo da pontuação [12].

Capítulo 8

Pulse

Neste capítulo apresentamos uma análise de uma implementação de um motor de xadrez.

O programa analisado foi o *Pulse* [31] que é escrito na linguagem Java ou C++. Para este trabalho, a versão escrita em Java foi escolhida.

O *Pulse* foi escolhido pois é um motor de código aberto e foi escrito para o propósito didático, e, portanto, o programa é bem estruturado e fácil de entender.

Este motor tem como características:

- Compatibilidade ao protocolo *UCI*;
- Representação de tabuleiro 0x88;
- Uso de *bitboard* como lista de peças;
- Algoritmo de busca alfa-beta;
- Busca em profundidade iterativa;
- Busca de repouso; e
- Função de avaliação que considera material e mobilidade.

Nos códigos apresentados a seguir foram adicionados alguns comentários para facilitar o entendimento.

Comunicação

Existem bibliotecas já prontas que leem os comandos da entrada padrão e armazenam as informações numa estrutura que é mais fácil para os motores utilizarem. Logo, não precisamos implementar a parte que faz a leitura da entrada padrão, o que deve ser feito é a manipulação da estrutura de dados que a biblioteca cria.

Para a linguagem Java, existe uma implementação chamada `JCPI` [26] e o motor *Pulse* utiliza esta biblioteca para receber e enviar comandos *UCI*.

Código 8.1: Pulse.java:9

```java
// Utiliza a biblioteca jcpi.
import com.fluxchess.jcpi.AbstractEngine;
import com.fluxchess.jcpi.commands.*;
import com.fluxchess.jcpi.models.GenericBoard;
import com.fluxchess.jcpi.models.GenericColor;
import com.fluxchess.jcpi.models.GenericMove;
import com.fluxchess.jcpi.protocols.IProtocolHandler;

//...

final class Pulse extends AbstractEngine implements Protocol {
  // ...

  // Manipula vários comandos com auxílio de estruturas da classe JCPI.
  public void receive(EngineInitializeRequestCommand command) {
    //...
  }

  // Define algumas opções.
  public void receive(EngineSetOptionCommand command) {
    //...
  }

  // Comando para debug.
  public void receive(EngineDebugCommand command) {
    //...
  }

  // Pergunta se o motor está pronto para começar os cálculos.
  public void receive(EngineReadyRequestCommand command) {
    //...
  }

  // Descarta todos os resultados e reinicia.
  public void receive(EngineNewGameCommand command) {
    //...
  }

  // Inicia a análise da configuração.
  public void receive(EngineAnalyzeCommand command) {
    //...
  }

  // Inicia o cálculo ou busca de um movimento.
  public void receive(EngineStartCalculatingCommand command) {
    //...
  }

  // Interrompe o cálculo atual.
  public void receive(EngineStopCalculatingCommand command) {
    //...
  }
```

```
53
54    // Envia o melhor movimento encontrado.
55    public void sendBestMove(int bestMove, int ponderMove) {
56      //...
57    }
58
59    // Envia o estado da busca.
60    public void sendStatus(boolean force, int currentDepth, int currentMaxDepth, long
          totalNodes,
61    int currentMove, int currentMoveNumber) {
62      //...
63    }
64
65    // Envia um movimento.
66    public void sendMove(RootEntry entry, int currentDepth, int currentMaxDepth, long
          totalNodes) {
67      //...
68    }
69
70    //...
71  }
```

A manipulação dos comandos recebidos pela entrada padrão é feita principalmente pela classe `Pulse`. Esta classe trata as informações recebidas com auxílio da classe `JCPI` que é uma implementação do protocolo de comunicação *UCI* na linguagem Java. Temos vários métodos `receive` sobrecarregados que recebe comandos diferentes e executa tarefas adequadas.

Representação do tabuleiro

A principal representação de tabuleiro utilizada neste motor é o tabuleiro 0x88.

A implementação está na classe `Square`.

Código 8.2: Square.java:13

```
1  final class Square {
2     // ...
3
4     // Cada variável representa uma casa do tabuleiro 0x88.
5     static final int a1 = 0; static final int a2 = 16;
6     static final int b1 = 1; static final int b2 = 17;
7     static final int c1 = 2; static final int c2 = 18;
8     static final int d1 = 3; static final int d2 = 19;
9     static final int e1 = 4; static final int e2 = 20;
10    static final int f1 = 5; static final int f2 = 21;
11    static final int g1 = 6; static final int g2 = 22;
12    static final int h1 = 7; static final int h2 = 23;
13
14    static final int a3 = 32; static final int a4 = 48;
15    static final int b3 = 33; static final int b4 = 49;
16    static final int c3 = 34; static final int c4 = 50;
17    static final int d3 = 35; static final int d4 = 51;
18    static final int e3 = 36; static final int e4 = 52;
```

```
19      static final int f3 = 37; static final int f4 = 53;
20      static final int g3 = 38; static final int g4 = 54;
21      static final int h3 = 39; static final int h4 = 55;
22
23      static final int a5 = 64; static final int a6 = 80;
24      static final int b5 = 65; static final int b6 = 81;
25      static final int c5 = 66; static final int c6 = 82;
26      static final int d5 = 67; static final int d6 = 83;
27      static final int e5 = 68; static final int e6 = 84;
28      static final int f5 = 69; static final int f6 = 85;
29      static final int g5 = 70; static final int g6 = 86;
30      static final int h5 = 71; static final int h6 = 87;
31
32      static final int a7 = 96; static final int a8 = 112;
33      static final int b7 = 97; static final int b8 = 113;
34      static final int c7 = 98; static final int c8 = 114;
35      static final int d7 = 99; static final int d8 = 115;
36      static final int e7 = 100; static final int e8 = 116;
37      static final int f7 = 101; static final int f8 = 117;
38      static final int g7 = 102; static final int g8 = 118;
39      static final int h7 = 103; static final int h8 = 119;
40
41      // ...
42  }
```

Definimos a indexação do tabuleiro 0x88 da mesma maneira da figura 5.6, exceto que neste caso usamos números decimais. Cada variável inteiro indica qual é a casa.

Código 8.3: Square.java:106

```
1   final class Square {
2       // ...
3
4       // Verifica se a casa "square" está dentro do tabuleiro.
5       static boolean isValid(int square) {
6           return (square & 0x88) == 0;
7       }
8
9       //...
10  }
```

Este método é o que verifica se uma casa é válida ou não. (Exemplo mostrado na seção 5 do capítulo 5).

Código 8.4: Square.java:120

```
1   final class Square {
2     // ...
3
4     // Recupera a coluna da casa "square".
5     static int getFile(int square) {
6       assert isValid(square);
7
8       int file = square & 0xF;
9       assert File.isValid(file);
10
11      return file;
```

```
  }

  // Recupera a linha da casa "square".
  static int getRank(int square) {
    assert isValid(square);

    int rank = square >>> 4;
    assert Rank.isValid(rank);

    return rank;
  }

  // ...
}
```

Aqui temos dois métodos que devolvem, dado uma casa, a linha ou a coluna. Como mostrado na seção 5 do capítulo 5, cada casa do tabuleiro é representada pelo número `0LLL0CCC`, onde `LLL` é a linha e `CCC` é a coluna, logo, aplicando as operações binárias acima, podemos recuperar que linha ou coluna uma casa representa.

Para auxiliar o tabuleiro 0x88, temos também o uso de *bitboard*. Esta representação está implementada principalmente no arquivo `Bitboard.java`. Nela estão a definição da variável que representa o *bitboard* e alguns métodos para manipular esta variável.

Código 8.5: Bitboard.java:13

```
final class Bitboard {

  // Uma variável long de 64 bit.
  long squares = 0;

  // ...
}
```

O *bitboard* é representado pela variável `squares` de tipo `long` que tem tamanho 64 bit.

Código 8.6: Bitboard.java:27

```
final class Bitboard {
    //...

    // Transforma uma casa de um bitboard
    // para uma casa de um tabuleiro 0x88.
    private static int toX88Square(int square) {
        assert square >= 0 && square < Long.SIZE;
        return ((square & ~7) << 1) | (square & 7);
    }

    // Transforma uma casa de um tabuleiro 0x88
    // para uma casa de um bitboard.
    private static int toBitSquare(int square) {
        assert Square.isValid(square);
        return ((square & ~7) >>> 1) | (square & 7);
    }

    //...
}
```

Temos um método que, fazendo algumas operações binárias, transforma uma casa de um tabuleiro 0x88 para *bitboard* e outro método que transforma uma casa de um *bitboard* para 0x88.

Código 8.7: Bitboard.java:43

```
final class Bitboard {
    // ...

    // Adiciona uma casa.
    void add(int square) {
      assert Square.isValid(square);
      assert (squares & (1L << toBitSquare(square))) == 0;

      squares |= 1L << toBitSquare(square);
    }

    // Remove uma casa.
    void remove(int square) {
      assert Square.isValid(square);
      assert (squares & (1L << toBitSquare(square))) != 0;

      squares &= ~(1L << toBitSquare(square));
    }

    // ...
}
```

Temos um método que adiciona e outro que remove uma peça do tabuleiro, utilizando operações binárias.

Busca

Os principais métodos da componente de busca foram mostrados no capítulo 6, portanto, aqui, vamos apresentar alguns trechos de códigos que auxiliam as buscas.

Código 8.8: Search.java:23

```
final class Search implements Runnable {

    // Declaração de um thread.
    private final Thread thread = new Thread(this);

    // Declarações dos semáforos.
    private final Semaphore wakeupSignal = new Semaphore(0);
    private final Semaphore runSignal = new Semaphore(0);
    private final Semaphore stopSignal = new Semaphore(0);

    // ...
}
```

Acima, temos declarações de um thread e semáforos. No *Pulse*, além do thread principal, há um thread específico para a tarefa de busca. Os semáforos são usados para a sincronização.

Código 8.9: Search.java:172

```
final class Search implements Runnable {
    //...
    Search(Protocol protocol) {
        //...
        // Executa o thread para a busca.
        thread.setDaemon(true);
        thread.start();
    }
    //...
}
```

O thread de busca é executado no construtor da classe `Search`.

Código 8.10: Search.java:207

```
final class Search implements Runnable {
    //...
    // Inicia o thread.
    synchronized void start() {
        if (!running) {
            try {
                wakeupSignal.release();
                // Envia um sinal para iniciar a busca.
                runSignal.acquire();
            } catch (InterruptedException e) {
                //...
            }
        }
    }

    // Para o thread.
    synchronized void stop() {
        if (running) {
            abort = true;

            try {
                // Envia um sinal para parar.
                stopSignal.acquire();
            } catch (InterruptedException e) {
            //...
            }
        }
    }
    //...
}
```

Os semáforos são usados, por exemplo, nos métodos `start` e `stop`.

A seguir, destacamos alguns trechos de códigos relacionados à geração de movimentos, que é um componente fundamental para a busca.

Código 8.11: MoveGenerator.java:13

```
final class MoveGenerator {

  // Cria uma lista que armazena os movimentos.
  private final MoveList<MoveEntry> moves = new MoveList<>(MoveEntry.class);

  // Devolve uma lista contendo somente movimentos válidos.
  MoveList<MoveEntry> getLegalMoves(Position position, int depth, boolean isCheck) {
    assert position != null;

    // Uma lista que recebe todos os movimentos possíveis.
    MoveList<MoveEntry> legalMoves = getMoves(position, depth, isCheck);

    int size = legalMoves.size;
    legalMoves.size = 0;

    // Para cada movimento em "legalMoves", executa uma verificação.
    for (int i = 0; i < size; ++i) {
      int move = legalMoves.entries[i].move;

      position.makeMove(move);
      if (!position.isCheck(Color.opposite(position.activeColor))) {
        legalMoves.entries[legalMoves.size++].move = move;
      }
      position.undoMove(move);
    }

    return legalMoves;
  }

  //...
}
```

O método acima recebe uma configuração, uma profundidade e uma indicação de xeque e devolve uma lista de movimentos válidos.

Código 8.12: MoveGenerator.java:35

```
final class MoveGenerator {
  //...

  // Devolve uma lista contendo todos os movimentos possíveis.
  MoveList<MoveEntry> getMoves(Position position, int depth, boolean isCheck) {
    assert position != null;

    moves.size = 0;

    if (depth > 0) { // Gera os movimentos.
      // Armazena na lista "moves" os movimentos possíveis de cada peça
      // na situação da configuração "position".
      addMoves(moves, position);

      // Considera também o movimento de encastelamento.
      if (!isCheck) {
        int square = Bitboard.next(position.pieces[position.activeColor][PieceType.KING].
            squares);
```

```
18          addCastlingMoves(moves, square, position);
19        }
20      } else { // Gera os movimentos relativamente em repouso ou quiescent.
21
22        addMoves(moves, position);
23
24        if (!isCheck) {
25          int size = moves.size;
26          moves.size = 0;
27          for (int i = 0; i < size; ++i) {
28            if (Move.getTargetPiece(moves.entries[i].move) != Piece.NOPIECE) {
29              moves.entries[moves.size++].move = moves.entries[i].move;
30            }
31          }
32        }
33      }
34
35      moves.rateFromMVVLVA();
36      moves.sort();
37
38      return moves;
39    }
40
41    //...
42  }
```

O `getMoves` é um método que gera os movimentos possíveis, sem considerar se cada movimento é válido.

Avaliação

Os métodos que calculam o material e a mobilidade foram mostrados no capítulo 7.

Código 8.13: Evaluation.java:23

```
1  final class Evaluation {
2      // ...
3      // Calcula a pontuação da posição.
4      int evaluate(Position position) {
5          assert position != null;
6
7          // Inicialização.
8          int myColor = position.activeColor;
9          int oppositeColor = Color.opposite(myColor);
10         int value = 0;
11
12         // Avaliação do material.
13         int materialScore = (evaluateMaterial(myColor, position) - evaluateMaterial(
               oppositeColor, position))
14             * materialWeight / MAX_WEIGHT;
15         value += materialScore;
16
17         // Avaliação da mobilidade.
18         int mobilityScore = (evaluateMobility(myColor, position) - evaluateMobility(
               oppositeColor, position))
```

```
            * mobilityWeight / MAX_WEIGHT;
        value += mobilityScore;

        value += TEMPO;

        assert Math.abs(value) < Value.CHECKMATE_THRESHOLD;
        // Devolve a pontuação final.
        return value;
    }

    // ...
}
```

Este método chama, internamente, os métodos `evaluateMaterial` e `evaluateMobility` e devolve como pontuação final a soma dos valores obtidos.

Capítulo 9

Disciplinas relacionadas

A seguir apresentamos algumas disciplinas oferecidas no curso de bacharelado em ciência da computação que teve relação com este trabalho.

- **MAC0110 - Introdução à Computação**
  MAC0122 - Princípios de Desenvolvimento de Algoritmos
  Aprendemos os conceitos básicos de programação e conhecemos a linguagem Java: análise de códigos do *Pulse* (capítulo 8).
- **MAC0211 - Laboratórios de Programação I**
  Conhecimento sobre interface de linha de comando e latex: ajudou na elaboração deste texto.
- **MAC0323 - Estruturas de Dados**
  Conhecimento sobre estrutura de dados básicos da computação. Esta disciplina foi uma das mais importantes para este trabalho, principalmente para a representação do tabuleiro (capítulo 5).
- **MAC0329 - Álgebra Booleana**
  Conhecimento sobre operadores binários: ajudou a entender as operações feitas nos tabuleiros 0x88 e *bitboard* (capítulo 5).
- **MAC0242 - Laboratório de Programação II**
  Conhecimento mais avançado da linguagem Java: importante para a leitura dos códigos (capítulo 8).
- **MAC0328 - Algoritmos em Grafos**
  Conhecimento de algoritmos de buscas em grafos: algoritmos de busca em profundidade (capítulo 6).
- **MAC0338 - Análise de Algoritmos**
  Conhecimento de como analisar a eficiência de algoritmos: busca minimax e alfa-beta (capítulo 6)
- **MAC0425 - Inteligência Artificial**
  Conhecimento da busca e função de avaliação: busca minimax, algoritmo alfa-beta, funções como material e mobilidade (capítulo 6 e 7). Esta, também, foi uma disciplina importante.
- **MAC0332 - Engenharia de Software**
  Conhecimento de planejamento: ajudou na elaboração deste trabalho.

Capítulo 10

Comentários finais

Apesar de um motor de xadrez ser composto por somente quatro componentes, estas partes, cada um, podem ser complexos. O volume da teoria relacionada a cada parte é muito grande e os conceitos apresentados neste trabalho são somente uma parte dos estudos existentes.

Mesmo assim, implementar um motor simples mas funcionando não é uma tarefa difícil e, portanto, qualquer programador pode pesquisar sobre o assunto e criar o seu próprio motor sem muita dificuldade. O desafio é criar um motor forte, ou seja, que possui uma boa representação de tabuleiro, um algoritmo de busca eficiente e uma função de avaliação que calcula a pontuação com resultado que leva a uma estratégia boa.

Finalmente, o que ficou faltando neste trabalho foi considerar o paralelismo no motor de xadrez. Com o avanço dos processadores, o paralelismo vem sendo muito comum nos programas de computadores e seria natural a aplicação desta tecnologia no motor. Mas, isto ficará como trabalho futuro.

Bibliografia

[1] *Architural Overview*. (visitado em novembro de 2014). 2013. URL: http://www.fam-petzke.de/cp_aod_en.shtml.

[2] *Chess Diagram Setup*. (visitado em novembro de 2014). 2014. URL: http://www.jinchess.com/chessboard/composer/.

[3] *Chess engine - Wikipedia, the free encyclopedia*. (visitado em novembro de 2014). 2014. URL: http://en.wikipedia.org/wiki/Chess_engine.

[4] *Chess opening book - Wikipedia, the free encyclopedia*. (visitado em novembro de 2014). 2014. URL: http://en.wikipedia.org/wiki/Chess_opening_book.

[5] *chessprogramming - 0x88*. (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/0x88.

[6] *chessprogramming - 8x8 Board*. (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/8x8+Board.

[7] *chessprogramming - Alpha-Beta*. (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Alpha-Beta.

[8] *chessprogramming - Aspiration Windows*. (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Aspiration+Windows.

[9] *chessprogramming - Bitboards*. (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Bitboards.

[10] *chessprogramming - Board Representation*. (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Board+Representation.

[11] *chessprogramming - Center*. (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Center.

[12] *chessprogramming - Center Control*. (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Center+Control.

[13] *chessprogramming - Evaluation*. (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Evaluation.

[14] *chessprogramming - Iterative Deepening*. (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Iterative+Deepening.

[15] *chessprogramming - Material*. (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Material.

[16] *chessprogramming - Minimax*. (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Minimax.

[17] *chessprogramming - Mobility*. (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Mobility.

[18] *chessprogramming - Move Generation.* (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Move+Generation.

[19] *chessprogramming - Pawn Structure.* (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Pawn+Structure.

[20] *chessprogramming - Piece-Lists.* (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Piece-Lists.

[21] *chessprogramming - Piece-Square Tables.* (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Piece-Square+Tables.

[22] *chessprogramming - Quiescence Search.* (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Quiescence+Search.

[23] *chessprogramming - Transposition Table.* (visitado em novembro de 2014). 2014. URL: https://chessprogramming.wikispaces.com/Transposition+Table.

[24] *En passant - Wikipédia, a enciclopédia livre.* (visitado em novembro de 2014). 2014. URL: http://pt.wikipedia.org/wiki/En_passant.

[25] *Endgame tablebase - Wikipedia, the free encyclopedia.* (visitado em novembro de 2014). 2014. URL: http://en.wikipedia.org/wiki/Endgame_tablebase.

[26] *fluxroot/jcpi - Github.* (visitado em novembro de 2014). 2014. URL: https://github.com/fluxroot/jcpi.

[27] *Human–computer chess matches - Wikipedia, the free encyclopedia.* (visitado em novembro de 2014). 2014. URL: http://en.wikipedia.org/wiki/Human_computer_chess_matches.

[28] T. A. Marsland. "Computer chess and search". Em: (3 de abr. de 1991).

[29] *Minimax - Wikipédia, a enciclopédia livre.* (visitado em novembro de 2014). 2014. URL: http://pt.wikipedia.org/wiki/Minimax.

[30] *Motor de xadrez - Wikipédia, a enciclopédia livre.* (visitado em novembro de 2014). 2014. URL: http://pt.wikipedia.org/wiki/Motor_de_xadrez.

[31] *Pulse Engine - Flux Chess Project.* (visitado em novembro de 2014), (Pulse Chess is released under the MIT License.) 2014. URL: http://www.fluxchess.com/pulse/.

[32] *Regra dos 50 movimentos - Wikipédia, a enciclopédia livre.* (visitado em novembro de 2014). 2014. URL: http://pt.wikipedia.org/wiki/Regra_dos_50_movimentos.

[33] *Roque (xadrez) - Wikipédia, a enciclopédia livre.* (visitado em novembro de 2014). 2014. URL: http://pt.wikipedia.org/wiki/Roque_(xadrez).

[34] Stuart J. Russell e Peter Norvig. *Artificial Intelligence A Modern Aproach.* third edition. Upper Saddle River, New Jersey 07458: Pearson Education, Inc., 2010.

[35] Claude Shannon. "Programming a Computer for Playing Chess". Em: *Philosophical Magazine* 41.314 (1949).

[36] *The Board Representation - Simple computer programming.* (visitado em novembro de 2014). 2013. URL: http://simplecomputerprogramming.blogspot.com/2013/01/the-board-representation.html.

[37] *Why is a 0x88 chessboard array of size 128 bytes when 0x88 is decimal 136? - Chess Stack Exchange.* (visitado em novembro de 2014). 2013. URL: http://chess.stackexchange.com/questions/4359/why-is-a-0x88-%20chessboard-array-of-size-128-bytes-when-0x88-is-decimal-136.

[38] *Wolfgang von Kempelen - Wikipedia, the free encyclopedia.* (visitado em novembro de 2014). 2014. URL: http://en.wikipedia.org/wiki/Wolfgang_von_Kempelen.

[39] *Xadrez - Wikipédia, a enciclopédia livre.* (visitado em novembro de 2014). 2014. URL: http://pt.wikipedia.org/wiki/Xadrez.

Índice Remissivo