Técnicas computacionais para o jogo de Xadrez

Cyntia M. Abreu, Iury N. Lazoski, Paulo E. D. Pinto, Eduardo R. Waghabi
Departamento de Informática e Ciência da Computação
Universidade Estadual do Rio de Janeiro
cynmo2004@yahoo.com.br, lazoski@gmail.com, pauloedp@ime.uerj.br, ewaghabi@timbrasil.com.br

Abstract

O desenvolvimento de jogos de xadrez por computador ('motores' de xadrez) sempre exerceu grande atração pública e acadêmica, principalmente pela intrigante questão relacionada a máquinas 'pensantes'.

O objetivo inicial de construir programas que pudessem vencer humanos nesse jogo já foi alcançado, mas o interesse científico no tema continua pois a área constitui-se um laboratório de pesquisa de algoritmos e soluções, que são utilizadas em muitos outros campos.

O presente trabalho descreve várias técnicas computacionais aplicáveis à geração de jogos de xadrez.

Ilustra também muitas dessas técnicas através do ICE, um 'motor' desenvolvido com objetivos didáticos, mas que atingiu um nível de jogo mais que 'razoável'.

1. Introdução

Este trabalho apresenta um resumo das técnicas discutidas em [1], para implementação de jogos de xadrez por computadores. Muitas dessas técnicas foram efetivamente utilizadas no Projeto ICE, que construiu um motor para o jogo de xadrez, disponível na web no endereço http://www.nxn.kit.net/ICE.

Por sua enorme combinação de possibilidades, o xadrez é o jogo mais utilizado mundialmente para a pesquisa de algoritmos de busca e heurísticas de jogos, que não surpreendentemente têm grande aplicação em diversos outros projetos computacionais, comerciais ou acadêmicos.

O desenvolvimento de programas autômatos (ou motores, como serão adiante chamados) de xadrez, é tema de grande interesse mundial. As fabricantes de softwares ChessBase (alemã) e Convekta (russa) vendem centenas de milhares de dólares anuais em software especializado, a maioria tendo motores como motivo principal ou incidental (caso dos banco de dados de partidas) . No meio acadêmico, as universidades de Alberta, em Edmonton, Canadá, e Vrije, em Amsterdã, Holanda, entre outras, produziram várias pesquisas sobre o assunto, tema de teses de mestrado a pós doutorado.

No meio acadêmico nacional, no entanto, há pouco suporte à pesquisa deste assunto, o que se por um lado estimulou a escolha deste tema, por outro, trouxe sérias dificuldades quanto às fontes bibliográficas, tendo se baseado, primordialmente, em textos e documentos extraídos da Internet.

A história da mecanização do jogo de xadrez é fascinante e contém muitos episódios notáveis. O primeiro deles foi a criação do famoso turco de Maelzel, que atraiu multidões no século XIX e era, na verdade, uma farsa: consistia de engenhoca mecânica com uma pessoa escondida no seu interior para o comando das peças. Desde então, até o surgimento do Deep Blue, o progresso foi grande. Com a vitória deste último sobre o grande campeão Garry Kasparov, em 1997, chegou-se à supremacia das máquinas sobre os humanos neste jogo. E isso parece ser um fato permanente. Detalhes sobre toda essa história podem ser encontrados em [1, 4, 23, 16].

Este trabalho está organizado da seguinte maneira: a Seção 2 descreve as técnicas e considerações básicas para a construção de um motor de xadrez; a terceira Seção discute técnicas consideradas avançadas, necessárias para a criação de um bom motor. A Seção 4 comenta os mecanismos utilizados no Projeto ICE e a Conclusão apresenta algumas perspectivas desta área.

2. Construindo um motor de Xadrez

Para construir um motor de xadrez é necessário escolher um modelo de representação, ter um bom esquema de geração de lances, saber avaliar posições, buscar com efetividade dentre as milhões de combinações possíveis em um jogo, dentre outros requisitos.

2.1. Modelos de representação

Uma das características mais relevantes de um motor, e certamente a primeira decisão a ser tomada na sua construção, é a escolha das estruturas de dados que representarão o tabuleiro e outras entidades afins. Esta decisão afeta a maior parte da codificação, e por si só cria vantagens e barreiras que delineiam o próprio estilo de jogo do programa.

Os principais modelos de representação são: Modelo Matricial, Bitboards e Modelo 0x88, descritos a seguir.

2.1.1 O Modelo Matricial

A representação mais intuitiva de um tabuleiro de xadrez dá-se por uma matriz bidimensional de 8 linhas por 8 colunas, perfazendo 64 'casas', a dimensão exata de um tabuleiro real. Cada 'casa' pode ser, neste caso, uma estrutura que indique a cor da casa, a peça que a ocupa, e quaisquer informações que o programador achar úteis. Algumas das informações armazenadas costumam ser:

- Número de peças brancas que atacam (defendem) a casa.
- Número de peças negras que atacam (defendem) a casa.
- Valor posicional da casa (absoluto ou dinâmico, sendo calculado de acordo com cada posição, neste caso).
- Indicação se a casa possui um caminho diagonal livre até um dos Reis (esta informação permite saber com facilidade quando um lance realizado é um xeque).
- Indicação se a casa possui um caminho vertical ou horizontal até um dos Reis.

Embora intuitivo, o esquema com uma matriz bidimensional traz muitos problemas pelo manuseio de dois índices para a determinação de uma casa, e só é utilizado por programas amadores com finalidade didática. Normalmente, a matriz bidimensional é substituída por um vetor com 64 posições, e o índice de cada casa passa a ser único, obtido através da fórmula simples $linha * 8 + coluna$. Como esta é a forma como geralmente as linguagens lidam com matrizes bidimensionais, pode-se esperar um pequeno ganho de performance apenas em razão da mudança. Mais performance é ganha quando se considera a facilidade obtida na codificação da geração de lances.

2.1.2 O Modelo 0x88

Um modelo conhecido por Modelo 0x88 tornou-se popular por utilizar aritmética binária para supostamente aumentar a performance do motor matricial e melhorar a clareza e elegância do código. Este segundo efeito pode parecer redundante, mas é muito importante para facilitar a codificação dos trechos de conhecimento específico do código.

Neste caso, quer-se contornar problemas encontrados com a representação em forma de vetor quando da verificação dos limites do tabuleiro. Suponha que deseja-se varrer horizontalmente um tabuleiro representado em forma de vetor. Para isto, deve-se somar ou subtrair 1 ao índice do tabuleiro ao realizar a busca. O problema é que nestes casos, ao somar 1 ao índice da última casa de uma linha, obtém-se a primeira casa da linha seguinte, sem indicação de que a linha anterior, na verdade, chegou ao fim.

O modelo 0x88 representa o tabuleiro com um vetor de 128 casas, sendo a fórmula para a indexação das casas alterada para $linha*16+coluna$. Para entender a representação, pode-se pensar em dois tabuleiros, um ao lado do outro, sendo que apenas o da esquerda é o tabuleiro 'real'.

A idéia por trás desta modificação é que as representações binárias dos índices que compõem o tabuleiro da direita possuem o bit 0x08 ligado, o que não acontece no tabuleiro 'real' da esquerda. Assim, quando a linha 'acaba', seja da esquerda para a direita ou vice-versa, o índice alcança o tabuleiro direito, e basta uma pequena verificação para testar o final da linha.

Pode-se combinar os testes horizontais com os verticais, onde sabe-se que o índice está fora da faixa quando o bit 0x8 está ligado, e portanto o único teste necessário para a validação do índice é um 'e' binário com a constante 0x88, daí o nome do modelo.

A utilização do modelo 0x88 é uma forma intermediária entre o modelo matricial e o uso de bitboards, que veremos adiante. Embora o uso de bitboards seja o nosso preferido, hoje existe uma grande discussão na comunidade especializada sobre qual representação dentre bitboards e o modelo 0x88 é a mais rápida. Como os dois tipos de representação são muito diferentes na implementação, resultados práticos e teóricos que confirmem a superioridade de uma sobre a outra serão difíceis de se obter.

2.1.3 Bitboards

No extremo da idéia de utilizar aritmética binária para agilizar as operações estão os bitboards, tabuleiros onde cada casa é representada por um único bit. Sendo um bit estrutura obviamente insuficiente para armazenar todas as informações necessárias por casa, ainda que se adote uma postura espartana, a abordagem requer que o tabuleiro seja então representado por mais de um bitboard.

Aparentemente esta abordagem foi originalmente desenvolvida por Slate e Atkin em meados dos anos 70, porém existem indícios de que outro grupo desenvolveu a mesma idéia simultaneamente.

Na prática, tudo que um bit pode representar com seus dois estados é a presença ou ausência de uma peça na determinada casa. Assim, a representação por bitboards possui

um tabuleiro para cada tipo de peça, branca e negra. Em C, a declaração de TBoard, a estrutura para o tabuleiro principal seria:

```
typedef unsigned char TByte;
typedef unsigned long long TBitboard;
typedef struct {
    TBitboard rei[2];
    TBitboard damas[2];
    TBitboard torres[2];
    TBitboard bispos[2];
    TBitboard cavalos[2];
    TBitboard peoes[2];
    TBitboard pecas[2];
    TBitboard enPassant;
    TByte mskRoque[2];
    TByte vez;
    int numLance;
    TBitboard chaveHash;
} TBoard;
```

O poder dos bitboards começa a ficar aparente quando ele é combinado com as operações bitwise e a pré-computação.

Para obter um bitboard que represente todas as peças brancas, por exemplo, basta utilizar um 'ou' binário com todos os bitboards de peças brancas. Para todas as peças, some bitwise também as peças negras. Para extrair peça a peça de um bitboard, pode-se construir funções que extraiam o bit menos significativo de um inteiro, e combiná-las com 'e's binários para realizar varreduras extremamente rápidas. Muitas outras soluções podem ser criadas para os vários problemas que se apresentem durante a construção do motor: o céu é o limite para programadores hábeis e criativos.

No início da computação, a escassez de espaço de armazenamento tornava como única solução viável, ainda que lenta, calcular absolutamente tudo durante a execução. Hoje a situação inverteu-se e geralmente os programadores dispõem de mais espaço de armazenamento secundário do que podem consumir, tornando-se viável a pré-computação.

Com Bitboards, a pré-computação é feita calculando-se previamente e armazenando tabelas de movimentos para geração de lances, tabelas de ataque para avaliação de posição, tabelas de estruturas comuns de peões para o meio-jogo e final. Mais uma vez, a ferramenta trouxe liberdade para os programadores implementarem, sem perda de velocidade do motor, diversas extensões e verificações que trazem qualidade ao jogo.

As vantagens mencionadas fazem com que esta seja a principal escolha hoje para a representação do tabuleiro.

2.1.4 Outras estruturas

É claro que o conjunto de estruturas necessárias para um motor de xadrez não se limita ao tabuleiro. Outra estrutura muito importante, por exemplo, é a estrutura que guarda um lance. Mais uma vez, as informações presentes dentro desta estrutura variam de implementação a implementação, pois representam as próprias idéias dos programadores.

Como conjunto básico de informação, deve-se incluir um código para identificar a peça jogada e a casa de destino. Embora suficiente, este esquema é certamente simples demais e ignora várias técnicas importantes para que o motor jogue de forma razoável. Para tal, precisamos também de uma indicação da ocorrência de uma captura, e de um valor que qualifique o lance, para fins de ordenação, conforme mostrado adiante.

Opções muito comuns são também a casa de origem e a peça capturada, para que o motor possa 'desfazer' o lance caso necessário. Outras sugestões serão citadas ao longo do texto para satisfazer particularidades das diversas técnicas.

Outras estruturas como a tabela hash e as tabelas auxiliares do livro de aberturas, banco de dados de finais e outras heurísticas serão mencionadas adiante.

2.2. Geração de lances

A geração de lances é algo muito complexo. Como se a dificuldade em codificar todas as regras de movimentações e suas exceções não bastasse, a rotina de geração de lances é chamada cada vez que a busca alcança um novo nó e decide pesquisar mais profundamente, o equivalente a dizer que é executada exaustivamente. Construir uma rotina de geração de lances rápida é, portanto, fundamental para um bom motor.

A rotina deve ser, além de rápida, flexível para receber parâmetros que indiquem formas diferentes de gerar os lances. Isto é devido à necessidade do motor, em situações diferentes de busca, de analisar apenas determinados tipos de lances, como capturas e xeques. Estas razões ficarão mais evidentes quando falarmos sobre extensões de busca, adiante.

Os xeques são um problema incômodo para a geração de lances. Alguns programas não verificam se um lance é xeque, geram todos os lances normalmente e continuam a pesquisar até que o Rei seja capturado. Quando isto acontece, a rotina de geração de lances retorna um valor que identifica o xeque, e o motor trabalha retroativamente determinando que o lance anterior era impossível.

Esta abordagem é muito rápida, mas possui alguns problemas:

- Dificulta identificar se a captura do Rei ocorreu porque o lance anterior removeu uma peça que impedia a captura (peça 'cravada', no jargão enxadrístico), ou se porque houve um xeque ignorado há dois lances atrás. Isto é importante porque no primeiro caso o

lance é impossível e deve ser removido da lista de lances.

- Torna a busca ineficiente, pois a ordenação não levará em conta os lances que defendem o xeque, quando apenas estes deveriam ser pesquisados.
- Potencializa problemas de horizonte, quando a busca terminar em um xeque e o motor não solicitar uma extensão de busca para verificar se há mate ou perda de material após o xeque, uma vez que o mesmo não identificou que estava em xeque.

Para contornar estes problemas, pode-se verificar a segurança do Rei em todas as gerações de lances. As formas mais utilizadas para tal são:

- Manter a informação de quais casas estão atacadas por quais peças, atualizada a todo instante. Conhecida como 'tabela de ataques', esta informação pode não só ser utilizada para a verificação de xeques, mas também para outras decisões do motor, como extensões de busca, ordenação de lances e avaliação posicional.
- Criar uma rotina para verificar se uma determinada casa está atacada por determinado bando. Esta rotina pode ser então aplicada à casa do Rei para verificar se a posição final é xeque.

A primeira solução é a mais adotada, porque a tabela se mostra muito útil em outras situações e, portanto, compensa o trabalho extra de mantê-la atualizada.

Os movimentos de roque e captura 'en passant' são exceções às regras de movimentação e devem possuir sinalizadores especiais para coordenar suas execuções. Para o roque, normalmente mantém-se na estrutura do tabuleiro indicadores da possibilidade dos mesmos. Caso uma torre se mexa, o roque para o lado correspondente fica impossibilitado para o resto da partida, e basta desmarcar o indicador do tabuleiro para que o mesmo não seja mais gerado pela rotina. Esta deve ainda verificar se as casas por onde o Rei passará para efetuar o roque não estão atacadas (e o Rei não está em xeque), e esta é mais uma situação onde a tabela de ataques é útil.

Para as capturas 'en passant', normalmente se mantém no tabuleiro informações sobre se uma captura deste tipo é possível em determinada coluna ou casa. Quando um bando avança um peão na casa inicial duas casas à frente (lance que caracteriza a possibilidade da captura 'en passant'), a coluna em questão é habilitada para a captura 'en passant', e o lance será relacionado entre os possíveis pela rotina de geração de lances, se for o caso. Como a regra diz que as capturas 'en passant' só são permitidas se realizadas no lance imediatamente seguinte, ao início de cada jogada de um bando suas colunas de sinalização 'en passant' são reinicializadas.

2.2.1 Geração de lances no Modelo Matricial

Em um modelo matricial como o descrito na Seção anterior, os lances devem ser gerados somando-se e subtraindo-se valores específicos ao índice, de acordo com o tipo da peça que ocupa cada casa. Para os cavalos, peões e Rei um conjunto fixo de valores deve ser somado ao índice que representa a casa para obterem-se os destinos possíveis. Os destinos devem então ser testados para verificar se estão dentro do tabuleiro, se estão ocupados por uma peça do mesmo bando (lance impossível), ou se estão ocupados por uma peça oponente, o que significa uma captura.

As chamadas 'peças deslizantes' (Dama, Bispos e Torres) são sempre as mais difíceis de implementar. Neste caso, é necessário realizar vários loops, um para cada direção possível, e incrementar um índice casa a casa, verificando se o movimento da peça foi interrompido por outra peça ou o fim do tabuleiro. Normalmente as rotinas de geração são separadas por peças e parametrizadas, de forma que os lances da dama podem ser gerados a partir das rotinas do bispo e da torre, combinadas.

Mesmo quando se utiliza o modelo por bitboards, uma implementação semelhante à matricial é utilizada para pré-computar os vetores que serão utilizados na geração de lances, embora neste caso não exista teste para verificar se as casas estão ocupadas. Vejamos o código para o cavalo:

```
int raios[8] = -8, -7, 1, 9, 8, 7, -1, -9;
int lancesCavalo[8] = -17, -15, -10, -6, 6, 10, 15, 17;
(...)
for (raio = 0; raio < 8; raio+=1) {
    /* lances de cavalo */
    casaDestino = casaOrigem + lancesCavalo[raio];
    /* valida limites do tabuleiro */
    if (!(casaDestino < 0 or casaDestino > 63 or
    abs((casaDestino&7)-(casaOrigem& 7)) > 2))
        mskBitboardLancesCavalo[casaOrigem] =
            mskBitboardUnitario[casaDestino];
}
```

2.2.2 Geração de lances com Bitboards

Bitboards proporcionam uma geração de lances rápida, embora um tanto complexa e de difícil manutenção. A premissa original é a utilização de pré-computação tanto quanto possível, valendo-se da vantagem do baixo consumo de espaço da representação.

Como já foi explicado anteriormente, a implementação por bitboards utiliza um bitboard para cada tipo de peça. Assim, quando da geração dos lances, o motor verifica cada bitboard do bando a jogar, procurando por bits ligados que significam a presença da peça na casa.

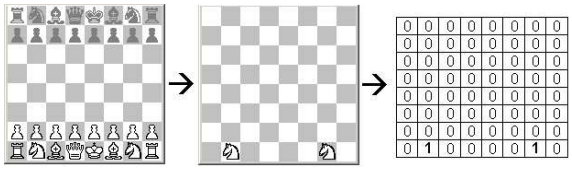

Figura 1. Representação dos cavalos brancos em sua posição original

Para percorrer um bitboard à procura de peças, utiliza-se para cada casa do tabuleiro um outro bitboard com apenas um único bit ligado, representando a casa, que possa ser utilizado através de um e-bitwise para determinar se naquela casa há uma peça. Os bitboards para teste podem ser calculados no momento da geração, utilizando um bitboard original com apenas o primeiro bit ligado, e progressivamente incrementando um fator de shift que posicione o bit ligado na casa desejada. Mais prático e rápido, porém, é valer-se da pré-computação e possuir um vetor de tais bitboards já calculado, onde o índice representa a casa. A Figura 2 ilustra bitboards para os cavalos brancos.

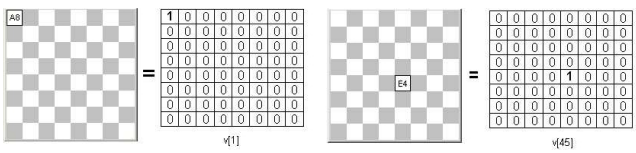

Figura 2. Posições 1 e 45 de um vetor com bitboards pré-computados com apenas uma casa ligada

A geração de lances é praticamente imediata para peões, cavalos e rei: como estas peças têm claramente definidas a partir de suas casas de origem quais são as casas de destino podemos pré-computar vetores de bitboards que, indexados pela casa de origem, tenham bits ligados nas casas de destino da peça em questão.

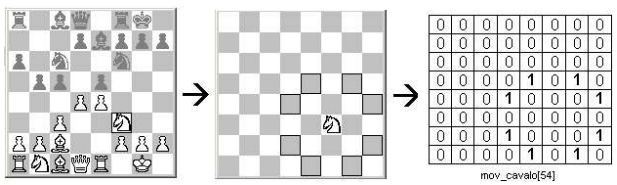

Figura 3. Bitboard com os destinos possíveis de um cavalo em f3(54)

Caso exista uma peça na casa n, basta utiliza o vetor correspondente da peça, indexado por n, obtendo um bitboard com bits ligados representando os movimentos possíveis. Entretanto, isto não é o suficiente. Para garantir que o movimento seja legal, é necessário verificar se a casa de destino está ocupada por uma peça do mesmo bando. Felizmente, isto pode ser feito de forma simples utilizando-se o bitboard que contém todas as peças de bando, e uma simples operação bitwise.

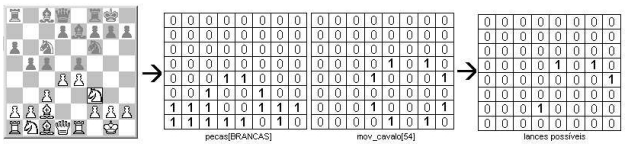

Figura 4. Bitboard com a verificação de lances impossíveis.

Da mesma forma pode-se identificar as capturas (que normalmente possuem tratamento diferenciado na geração de lances) utilizando-se o bitboard que contém todas as peças do bando adversário.

Um desafio bem maior encontra-se quando vamos gerar os lances das peças deslizantes (bispos, torres e damas), pois pela regra do xadrez, as casas destino em uma determinada direção dependem das peças, próprias e do adversário, encontradas no caminho. Embora vários motores implementem soluções originais para bitboards, a solução mais aceita é através dos chamados 'bitboards rotacionados' (no original, rotated bitboards), introduzidos no programa Crafty, por seu autor Robert Hyatt [10], que hoje constituem o principal padrão para a geração de lances com bitboards.

Nesta técnica, os vetores de bitboards individualizados para cada tipo de peça (como descrito acima) contendo os destinos possíveis, são abandonados por uma solução que agrupa todas as peças deslizantes.

A idéia por trás desta abordagem é que a geração de lances para peças deslizantes é particularmente fácil para uma linha (horizontal), porque então as casas envolvidas estão representadas por bits consecutivos no bitboard.

Assim, pode-se obter uma linha de bits do bitboard que representa todas as peças no tabuleiro para obter um índice de 1 byte (8 bits) que represente o 'estado' da linha, ou seja, quais casas estão ocupadas na linha. Então, fazendo uso mais uma vez da pré-computação, consulta-se um array bidimensional que fornece os lances possíveis através de dois índices: estado de ocupação da linha (1 byte = 256 possibilidades) e a posição que a peça a mover ocupa no tabuleiro (64 possibilidades). O array em questão ocuparia o equivalente a 256 x 64 bitboards, ou aproximadamente 64k de memória. A Figura 5 ilustra a idéia.

Com o bitboard resultante de movimentos possíveis, mais

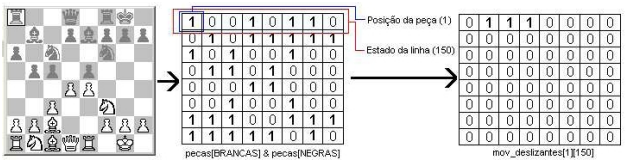

Figura 5. Bitboard indicando os movimentos possíveis.

uma vez verifica-se as capturas de peças do bando adversário, e elimina-se as capturas de peças do próprio bando, por serem ilegais.

Aparentemente a solução é incompleta, pois as peças deslizantes não se movem apenas pelas linhas, mas igualmente por colunas (Torres e Damas) e diagonais (Bispos e Damas). A idéia, que dá o nome à solução, é manter versões de todos os bitboards de peças girados em vários ângulos, de forma que o gerador de lances possa trabalhar com movimentos em diagonal e em colunas como se fossem linhas.

Este é o ponto chave da solução: girar os bitboards para que as colunas e diagonais possam ser interpretadas como linhas. Veja como fica a interpretação destes bitboards, e repare como a leitura das linhas é, na verdade, uma série de casas que forma uma coluna ou diagonal no tabuleiro original:

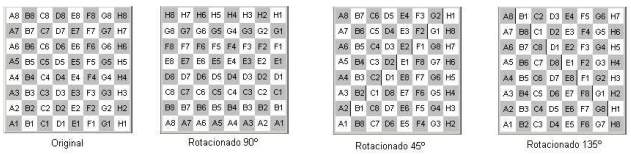

Figura 6. Bitboard rotacionados.

A leitura dos tabuleiros rotacionados em 45º e 135º não é fácil. Como existem mais diagonais (15) que linhas (8) em um tabuleiro de xadrez, as diagonais devem ser arrumadas para caberem em um só bitboard. Ao trabalhar com tais bitboards, precisamos utilizar máscaras de bits que eliminem casas da mesma linha que não estejam na mesma diagonal. Isto pode ser obtido com dois tipos de vetores auxiliares: um associando a cada casa um número de diagonal, e outro com as máscaras de bits para cada diagonal. Com uma numeração inteligente das diagonais nos primeiros vetores podemos utilizar um mesmo vetor de máscaras de bits para as duas rotações diagonais.

Para que tal esquema funcione, como já foi mencionado, é preciso manter versões rotacionadas em 45º, 90º e 135º dos bitboards de todas as peças após cada lance.

Felizmente, a manutenção de tais estruturas não é tão cara ou complexa assim, mais uma vez valendo-se de pré-computação.

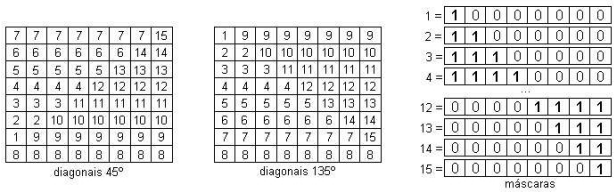

Figura 7. Os dois tipos de vetores auxiliares.

O vetor descrito na Figura 2 pode ser replicado e alterado para refletir as configurações das matrizes dos bitboards rotacionados. Assim, através de poucas operações bit-a-bit pode-se reproduzir um bitboard na forma original para sua versão rotacionada.

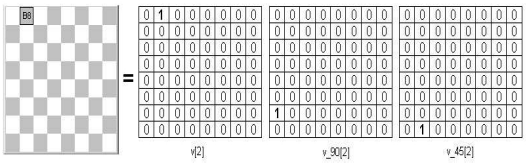

Figura 8. Exemplo de rotação de bitboards.

Por fim, é importante frisar que nem todos os autores estão completamente convencidos de que a utilização de bitboards com processadores de 32 bits é melhor que as representações matriciais (especificamente o modelo 0x88) a ponto de justificar a complexidade do código e o conseqüente custo na escrita e manutenção do mesmo. Entretanto, o surgimento de processadores de 64 bits pode pender definitivamente a balança para o lado dos bitboards.

2.3. Busca

Observe dois enxadristas disputando uma partida: apesar das jogadas iniciais acontecerem com alguma fluência, logo um dos lados vê-se em uma situação mais complexa, onde pensar exaustivamente parece ser a única solução para apresentar um bom lance ao adversário. Desta forma, uma partida pode estender-se por horas, ou até dias, assumindo que os mesmos têm tempo ilimitado para refletir em cima do tabuleiro.

O que um humano faz ao examinar uma posição é bem parecido com uma rotina de busca. Ele tenta prever jogadas do antagonista em resposta às suas, até onde sua memória suporta, e assim escolhe o que lhe parece ser o melhor lance. A técnica utilizada pelo computador não é muito diferente como vamos observar em detalhes a seguir.

2.3.1 Árvore de Jogo

Um programa de xadrez utiliza uma rotina de busca para tentar encontrar o melhor lance para o lado que está jogando. Para alcançar este objetivo a rotina percorre a árvore

de busca avaliando os lances gerados. Esta árvore de jogo é a representação de todas as jogadas legais geradas a partir de uma posição. O nó inicial é utilizado para representar a posição atual e as possíveis próximas jogadas encontram-se dispostas nos ramos subseqüentes da árvore.

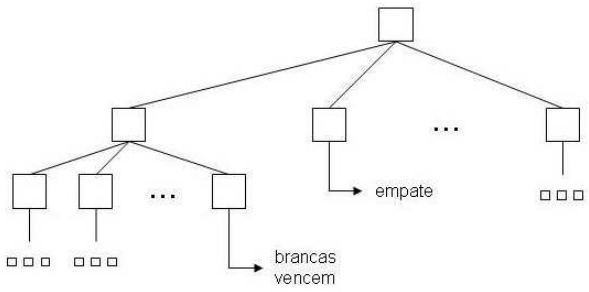

Figura 9. Árvore de jogo para um jogo de xadrez.

Cada ramo partindo do nó inicial, leva a uma nova posição do tabuleiro (ver Figura 10 e a transição entre os nós da árvore representam, desta forma, lances possíveis para cada posição analisada (ver Figura 11). Nas folhas da árvore encontramos as posições finais, que significam uma situação de empate ou vitória de um dos lados.

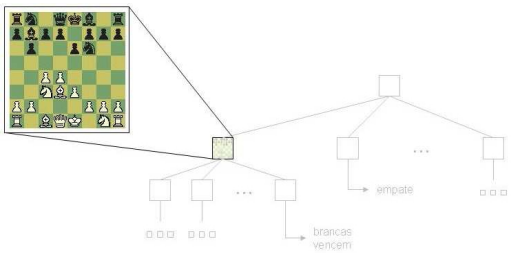

Figura 10. Cada nó representa uma posição do tabuleiro.

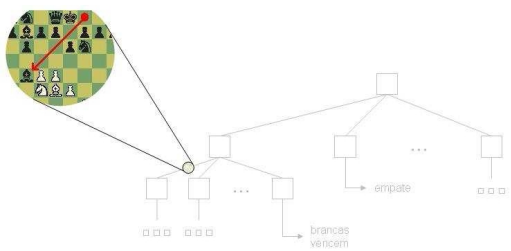

Figura 11. A ligação entre os nós representa um lance possível.

2.3.2 MiniMax

A árvore de busca por si só apenas guarda as posições e jogadas legais. É necessário uma rotina que realize a busca dentro da árvore. O MiniMax realiza a busca supondo um desenvolvimento de jogo onde ambos os jogadores executam sempre os melhores lances. Este algoritmo é comumente utilizado para jogos simples como o 'jogo da velha' (ver Figura 12).

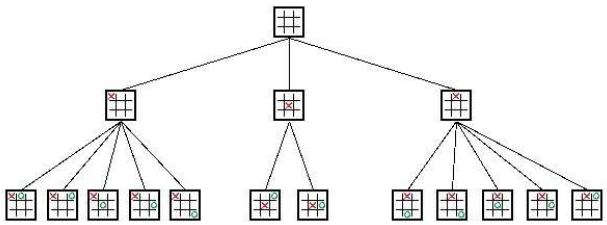

Figura 12. Árvore de jogo dos três primeiros níveis do jogo da velha (as posições redundantes foram suprimidas).

Nesta figura, em cada nível apenas um jogador está ativo e eles vão alternando turnos a medida que descemos na árvore. Na raiz da árvore, procuramos o melhor lance para o jogador A, no próximo nível, o melhor para o jogador B, e assim por diante. Significa afirmar que nesta busca alternamos a maximização e minimização do resultado retornado. Quando o jogador A está no seu turno, um lance que o faz ganhar retorna 1 e um que o faz perder, -1. Então o jogador A vai tentar maximizar o valor e o jogador B, minimizá-lo [3]. Se não podemos descer até a folha da árvore (onde teríamos uma posição de vitória, derrota ou empate) a função de avaliação utilizará uma heurística para determinar um valor para a posição (ver detalhes adiante).

No caso do jogo da velha, a simplicidade do jogo acarreta uma árvore de busca pequena e rapidamente pode-se chegar às folhas da árvore e avaliar a posição final como vitória, empate ou derrota. No xadrez, tal possibilidade só é viável nos estágios finais do jogo. Até lá, é preciso estabelecer uma profundidade na qual as 'folhas' serão examinadas por uma rotina de avaliação que tentará estabelecer um valor numérico que represente qual lado possui vantagem.

Uma vez que o MiniMax necessita conhecer todos os possíveis lances para realizar sua análise, ele não é recomendado em jogos como o poker, onde a mão do adversário não é conhecida. O mais usual é sua utilização em jogos com perfeita informação (como o xadrez), onde os lances de ambos os bandos são facilmente identificados.

O código para chamada da função descrita a seguir é:

```
valor = MinMax(5);
```

Na variável 'valor' será retornado o valor da posição pesquisado até a profundidade de nível 5 da árvore gerada.

```
int MinMax(int profundidade)
{   if (BandoAJogar() == B) // Branco 'maximiza'.
        return Max(profundidade);
    else // Preto é o bando 'minimizador'.
        return Min(profundidade);
}
int Max(int profundidade)
{   int melhor = -INFINITO;
    if (profundidade <= 0) return Avaliacao();
    GeraLancesLegais();
    while (LancesPendentes()) {
        ExecutaProximoLance();
        valor = Min(profundidade - 1);
        DesfazLance();
        if (valor > melhor) melhor = valor;
    }
    return melhor;
}
int Min(int profundidade)
{   int melhor = INFINITO; // !!!
    if (profundidade <= 0) return Avaliacao();
    GeraLancesLegais();
    while (LancesPendentes()) {
        ExecutaProximoLance();
        valor = Max(profundidade - 1);
        DesfazLance();
        if (valor < melhor) melhor = valor; // !!!
    }
    return melhor;
}
```

O número de posições que deve ser procurado por este algoritmo é L^P onde L é a largura da árvore (média do número de lances possíveis em cada posição) e P é a profundidade da árvore. Podemos observar que o algoritmo pesquisa todos os nós da árvore até uma determinada profundidade. Analisado do lado matemático, ele é completamente capaz de resolver o problema, porém nenhuma máquina nos dias atuais ou em um futuro próximo o utiliza para um jogo de xadrez, pois para conseguir executá-lo o programa fica limitado a um número ínfimo de níveis na árvore de busca. Por este motivo, esta função de busca somente serve de base para a criação de novos algoritmos mais eficientes.

2.3.3 Negamax

O algoritmo do MiniMax é fácil de entender mas o código é bastante redundante. O algoritmo que apresentamos agora trata-se de uma melhoria do MiniMax obtendo o mesmo resultado. Para eliminar a distinção entre as posições do programa e do oponente, o valor de uma posição é sempre avaliado do ponto de vista do jogador ativo, isto é, negando o valor que é avaliado pelo oponente. O pseudo-código a seguir ilustra o que foi descrito:

```
int NegaMax(int profundidade)
{   int melhor = -INFINITO;
    if (profundidade <= 0) return Avaliacao();
    GeraLancesLegais();
    while (LancesPendentes()) {
        ExecutaProximoLance();
        valor = -NegaMax(profundidade - 1); // !!!
        DesfazLance();
        if (valor > melhor) melhor = valor;
    }
    return melhor;
}
```

2.3.4 Poda AlfaBeta

A poda AlfaBeta foi o primeiro grande refinamento para reduzir o número de posições a serem pesquisadas, permitindo assim uma busca mais profunda no mesmo intervalo de tempo. A idéia é que em grandes partes da árvore, não interessa saber o exato valor de uma posição, apenas se ela é melhor ou pior do que a que já temos pesquisada até então.

Podemos visualizar melhor o que o AlfaBeta faz no excelente exemplo de Moreland [15]: Imagine que João e Pedro fizeram uma aposta, na qual Pedro saiu vitorioso. Então, João dispôs algumas mochilas à sua frente. Ele agora tem que dar algo a Pedro, mas as regras para determinar o que ele deve lhe dar são bastante obtusas. Cada mochila contém alguns itens. Pedro vai obter um destes itens. Ele pode escolher de qual mochila o item deve ser retirado, mas João escolherá o que retirar da mochila. Obviamente, o que acontecerá é que João dará a Pedro o pior item que estiver dentro da mochila escolhida; logo, o objetivo de Pedro é escolher a mochila que tiver o melhor pior item.

É facil enxergar como o MiniMax resolve este problema: um dos lados está maximizando as mochilas e o outro minimizando os itens. Mas como o AlfaBeta pode ajudar a diminuir o número de itens a serem examinados? Continuemos com o exemplo:

Caso a primeira mochila contenha um sanduíche e as chaves de um carro novo, fica claro que se esta mochila for a escolhida, será obtido o sanduíche, que é o item menos valioso. Então a próxima mochila é examinada e se for encontrado outro sanduíche na mochila, ou algo considerado pior que o sanduíche (um sanduíche estragado, por exemplo), esta mochila pode ser descartada sem olhar quaisquer outros itens. A razão para tanto é que se esta mochila for escolhida, o melhor a ser obtido não é melhor que o sanduíche, portanto ela pode ser certamente rejeitada.

Encontrar um sanduíche estragado na segunda mochila é como exceder beta. Não há razão para continuar olhando

outros itens, já que se sabe que não é possível obter nada melhor do que o melhor item já encontrado (o sanduíche).

Entretanto, é fundamental que este sanduíche estragado seja encontrado o quanto antes, para evitar que os outros itens sejam analisados desnecessariamente. Encontrar o melhor item da mochila por último torna o AlfaBeta nada melhor que o MiniMax. Por isso, o ganho com o AlfaBeta está intrinsecamente ligado à ordenação dos lances (ver adiante). O código para implementar a busca AlfaBeta precisa de dois novos parâmetros que representam os valores dos limites, um representando um limite inferior que um nó maximizante poderá receber em última instância (alfa), e o outro representando um limite superior que um nó minimizante poderá ter (beta). A seguir, um trecho simplificado do código implementado no ICE.

```
int AlphaBeta(int profundidade, int alpha, int beta) {
    // verifica final de busca
    if (profundidade == 0) { return Avaliacao();}
    // gera lances
    GeraListaLances();
    // obtem lances um a um, já ordenados
    while (ObtemMelhorLance() != -1) {
        Make(); // faz o lance no tabuleiro principal
        valor = -AlphaBeta(profundidade - 1, -beta,
            -alpha);
        UnMake(); // desfaz o lance
        // verifica poda
        if (valor > alpha) {
            alpha = valor;
            if (valor >= beta) { // beta foi excedido
            return beta;
        }
    }
    return alpha;
}
```

O código para chamada desta função é:

```
valor = AlphaBeta(5, -INFINITO, +INFINITO);
```

Sob circunstâncias ótimas a busca AlfaBeta ainda precisa pesquisar aproximadamente $L^{P/2}$ posições (onde L é a largura e P é a profundidade da árvore) [16]. Muito menos que o MiniMax, mas ainda exponencial. Ele permite alcançar aproximadamente o dobro de profundidade no mesmo intervalo de tempo. Mais posições precisarão ser pesquisadas se a ordenação de lances não for perfeita.

2.3.5 Busca Quiescente

No AlfaBeta apresentado o algoritmo realiza a pesquisa na árvore até uma profundidade previamente determinada e retorna o melhor lance de acordo com uma rotina de avaliação. O problema com este resultado é que um lance considerado bom numa profundidade pode ser avaliado como ruim se analisado algumas jogadas a frente. Por exemplo, um lance em que uma rainha toma um cavalo pode ser considerado bom, porém se o cavalo estava protegido e em seguida outra peça toma a rainha, com certeza esta posição não pode ser avaliada como boa. Este fato é comumente chamado de 'problema de horizonte', quando o motor toma decisões erradas por não conseguir enxergar além de seu 'horizonte' [14].

Para contornar esta situação foi criada a rotina de avaliação de busca quiescente, que desce na árvore examinando os próximos lances para avaliar apenas possíveis capturas e posições de xeque, desta forma analisando a estabilidade da posição.

A seguir, um trecho simplificado do código implementado no ICE.

```
int Quiescence(int alpha, int beta, int ply) {
    /* realiza a avaliação da posição */
    valor = Eval(&tabPrincipal);
    if (valor >= beta) return beta;
    if (valor > alpha) alpha = valor;
    /* gera lances para Quiescence */
    if (GeraListaLances(ply, MSKQUIESCENCE, 0, 0))
        return 0;
    while (ObtemMelhorLance() != -1) {
        /* obtem lances um a um */
        Make(); /* faz o lance no tabuleiro principal */
        valor = -Quiescence(-beta, -alpha, ply+1);
        UnMake();/* desfaz o lance */
        if (valor >= beta) return beta;
        if (valor > alpha) alpha = valor;
    }
    return alpha;
}
```

Antes de prosseguir com a busca, o algoritmo verifica alfa e beta e retorna o valor imediatamente em caso de uma poda. A função que gera lista de lances para esta etapa não gera todos os lances possíveis, mas apenas aqueles que oferecem perigo à 'quietude' da posição.

2.3.6 Janela de Aspiração

A janela de aspiração é uma pequena melhora na busca AlfaBeta. Normalmente, a primeira chamada ao algoritmo seria:

```
valor = AlphaBeta(profundidade,
                  -INFINITO, +INFINITO);
```

A janela de aspiração muda isto para:

```
valor = AlphaBeta(profundidade,
                  base-janela, base+janela);
```

onde 'base' é uma estimativa para o resultado esperado e 'janela' é uma medida para desvios que esperamos para esse resultado.

Este método pesquisará menos posições porque impõe limites à busca AlfaBeta já na raiz da árvore. O efeito colateral obtido é que o resultado da busca pode cair fora da janela de aspiração, caso no qual uma nova pesquisa deve ser realizada [15]. De qualquer forma, uma boa escolha para a variável janela em geral apresenta um ganho razoável.

2.3.7 Aprofundamento Iterativo

Também chamado de aprofundamento progressivo, este método significa chamar uma rotina com profundidade fixa várias vezes (no caso do motor de xadrez, esta rotina é o AlfaBeta), aumentando gradativamente a profundidade a cada nova chamada, até que o tempo se esgote ou a profundidade máxima seja alcançada [15]. Este método tem várias vantagens:

- A primeira e mais óbvia é não precisar escolher uma profundidade antes de realizar a busca; o resultado da última busca está sempre disponível caso esta não possa ser completada em tempo hábil. Isto é particularmente importante em jogos com tempo limitado, onde o motor pode ficar sem lances para jogar se for muito ambicioso ao escolher a profundidade da busca.
- Várias avaliações de posição e melhores movimentos já serão armazenados em uma tabela de transposição (ver adiante), o que significa que as buscas mais profundas terão uma ordenação de lances muito melhor do que uma busca que inicia imediatamente no nível mais profundo.
- Utilizar os valores retornados em cada busca para ajustar os valores das janelas de aspiração da próxima busca, se esta técnica for utilizada.

O código abaixo extraído do ICE exemplifica parte do que foi descrito:

```
int Busca(int tempoBusca) {
    int profundidade, avaliacao;
    TPv pv;
    InicializaTempoBusca(tempoBusca); // inicializa tempo
    // aprofundamento iterativo
    for (profundidade = 2; profundidade < MAXDEPTH;
        profundidade++) {
        flagTimeOut = 0;
        // realiza a busca com janela inicial maior possível;
        // a variante principal é re-enviada à busca para
        // auxiliar a ordenação de lances
        avaliacao = AlphaBeta(profundidade, -INFINITO,
            +INFINITO, &pv);
        // verifica se tempo se esgotou
        if (flagTimeOut) {
            // obtem lance da PV para retorno
            CopiaLance(&pv.lances[0]); break;
        }
    }
}
```

Alguns motores usam ainda a técnica de variante principal. Outros preferem um refinamento mais aprimorado deste mesmo algoritmo que é conhecido como Negascout.

2.4. Rotina de avaliação da posição

Quando a busca atinge a profundidade máxima pré-determinada, o motor deve avaliar a posição final para atribuir a ela um valor que determinará, através do algoritmo escolhido, se o primeiro lance será ou não o escolhido. Este procedimento é padrão em todos os jogos que utilizam rotinas de busca derivadas do mini-max, mas é especialmente complicado, e importante, no caso do xadrez.

Na implementação de motores para jogos simples como o 'jogo-da-velha', a quantidade de possibilidades é pequena o suficiente para que o motor não precise se preocupar em avaliar posições indefinidas. Basta determinar se a posição está ganha, perdida, ou empatada, e caso nenhuma das três possibilidades se aplique, solicitar a continuidade da busca. As únicas posições que possuem um critério objetivo de avaliação são as posições onde um dos lados encontra-se em xeque-mate, ou a posição está empatada, seja por afogamento (um dos jogadores não possui movimentos possíveis), ausência de material para mate, ou repetição da mesma posição pela terceira vez. O conjunto dessas alternativas é muito pequeno comparado às demais possibilidades de posições, para as quais a determinação de um valor para a posição é mais complexa.

É na rotina de avaliação onde é programada a maior parte das heurísticas relativas a cada jogo. Originalmente, todo o conhecimento específico ficava nesta, mas posteriormente foram adicionadas heurísticas à geração e ordenação dos lances para melhorar o jogo dos motores.

Os critérios utilizados dentro de tais rotinas variam muito de implementação a implementação, dependendo do gosto, qualidade (tanto como programador ou enxadrista) do autor, e até mesmo das características das estruturas utilizadas, e pode-se dizer que são os principais determinantes da 'personalidade' do motor. Por ser tarefa impossível discorrer sobre todas as nuances implementadas em tais rotinas, vamos detalhar apenas os aspectos mais importantes e comuns a todos os motores.

2.4.1 Material

O único critério realmente objetivo é a contagem de material, isto é, a soma dos valores atribuídos às peças que estão em jogo. Na realidade, vários autores confirmam que a implementação de um motor com apenas a verificação do

material na rotina de avaliação é suficiente para produzir um adversário formidável.

É também um dos mais simples critérios para se implementar, bastando somar o valor das peças de cada bando, e subtrair um total do outro para determinar que lado possui mais material. Para facilitar o algoritmo de busca, normalmente utiliza-se (total do material do motor − total do material do oponente), onde um placar positivo significa que o motor possui mais material. Entretanto, muitos motores multiplicam o resultado por −1, em certas circunstâncias, para refletir a convenção atual, com o valor sempre do ponto de vista das Brancas.

Classicamente a contagem de material é realizada utilizando-se o peão como unidade, e normalmente os valores são os seguintes: peões, 1 ; cavalos e bispos, 3; torres, 5; damas, 9; atribuindo-se um valor muito grande ao rei para que sua captura, embora ilegal, gere um desequilíbrio tão grande na posição que possa ser facilmente notada pela busca.

Estes valores fazem parte do dogma enxadrístico repetido (às vezes com pequenas alterações, como conferir 2,5 aos cavalos, ou 10 à dama) à exaustão pelos livros especializados. No entanto pequenas alterações nestes valores podem produzir resultados interessantes, principalmente para as tentativas recentes, desde que os motores ficaram melhores do que a maioria absoluta dos jogadores, de abaixar a qualidade do jogo de forma a tornar a experiência um pouco mais agradável.

O critério material determina o valor básico da posição, e todos os demais critérios modificam este valor através de pequenos bônus. Quanto aos bônus, a única recomendação é que sejam medidos na mesma unidade escolhida para o material (normalmente centésimos ou milésimos de peão). Os valores dos bônus são escolha pessoal do programador, e provavelmente serão ajustados ao longo de toda a vida do motor.

2.4.2 Segurança do Rei

O critério subjetivo mais importante é, pela própria definição do jogo, a segurança do Rei. O motor precisa avaliar e conferir um valor tanto à segurança de seu Rei quanto à de seu oponente. Alguns dos mais comuns critérios utilizados para tal são:

- Fornecer bônus para o roque, embora este bônus possa ser reduzido por diversos motivos tais como: rocar para o lado onde o adversário possui colunas abertas, rocar sem a proteção de peões, rocar quando as peças grandes do adversário já foram trocadas ou capturadas.
- Calcular uma tabela de ataques e defesas nas casas vizinhas ao Rei, especificamente nos peões que protegem o Rei no roque, e nas casas imediatamente à frente. Isto é importante para que o motor confira bônus tanto em trazer peças para a defesa quando seu Rei está atacado, quanto atacar corretamente o Rei adversário. Alguns motores possuem uma análise mais sofisticada que avalia os tipos de peças que estão envolvidas no ataque e defesa, e bonifica de acordo com a eficiência do conjunto.
- Bonificar linhas abertas de ataque (colunas ou diagonais) até o Rei adversário ou casas vizinhas. Isto é diferente da análise anterior porque avalia, na realidade, as possibilidades de colocação de peças que ataquem as casas vizinhas ao Rei.

O valor dos bônus precisa levar em consideração o lado que está jogando, o lado do Rei que está sendo avaliado, e a característica do critério (positivo ou negativo).

2.4.3 Estrutura de Peões

Uma grande parte da técnica do xadrez é dedicada aos peões, não só porque a estrutura que formam delineia o campo de batalha, mas também pela regra que permite que um peão se transforme em uma peça à escolha do jogador (normalmente uma Dama) quando alcança a oitava casa do tabuleiro, procedimento conhecido como promoção.

A natureza dos peões é proteger uns aos outros, pois devido ao seu baixo custo é má prática utilizar peças para protegê-los. Sendo assim, a teoria diz que um jogador deve manter o menor número possível de 'ilhas' de peões, ou seja, grupos isolados de peões. Pelo mesmo raciocínio, o peão dito 'isolado' (que não possui peões vizinhos para protegê-lo) é uma fragilidade na posição, salvo raríssimas exceções. Ambos os casos (muitas ilhas de peões e peões isolados) devem ser devidamente penalizados pela rotina de avaliação.

Peões passados são aqueles que não podem ser atacados por mais nenhum peão adversário. Tais peões são particularmente valiosos porque são candidatos potenciais à promoção citada anteriormente. Por isto, atribuir um bônus apropriado aos peões passados faz com que o motor evite que o adversário obtenha os mesmos, assim como incentiva seqüências de lances que deixem o motor com um destes peões.

Todos os critérios acima têm implementação simples sob qualquer tipo de representação. Porém o uso específico de bitboards facilita a utilização de um tipo de verificação especialmente útil: o reconhecimento de padrões de peões. Nesta, se pré-calcula padrões conhecidos de peões, junto com avaliações também pré-calculadas. Assim, pode-se implementar grande quantidade de conhecimento específico com baixo ou nenhum esforço de processamento.

2.4.4 Critérios Posicionais

Embora os critérios mais importantes sejam a segurança do Rei, e a estrutura de peões, a teoria enxadrística é pródiga em temas que podem ser explorados pelo motor, bastando para isto o conhecimento do programador (ou algum assessor específico para este objetivo) e a sensibilidade de evitar tornar a rotina de avaliação tão lenta a ponto da perda de velocidade do motor não ser compensada pelo ganho de conhecimento do mesmo.

Os critérios restantes, reunidos aqui sob o título de posicionais, devem-se, assim como todos os demais itens da rotina de avaliação, exclusivamente ao gosto do programador. Entretanto, a maioria dos motores avalia um conjunto comum de critérios, seguindo uma lista dos mais utilizados:

- Par de bispos: bonificação positiva para o lado que mantém seu par de bispos. Alguns motores vão além e verificam também as exceções encontradas na prática do xadrez onde cavalos podem ser mais valiosos que bispos.
- Centralização de peças e peões: A teoria diz que o centro é o local mais importante do tabuleiro, por que nele as peças ampliam sua ação e podem mais facilmente deslocar-se até os locais de maior interesse. Sendo assim, pontos são atribuídos para as peças que ocupam ou dominam casas no centro.
- Mobilidade de peças: em idéia muito próxima à anterior, dá-se pontos para peças que atacam muitas casas. Isto muitas vezes traduz-se em bonificar peças centralizadas, mas há exceções importantes que esta abordagem trata.
- Colunas abertas e semi-abertas: bonifica-se a ocupação de tais colunas por peças ditas 'pesadas' (torres e dama).
- Diagonais abertas: idem, mas com diagonais ocupadas por bispos.
- Casas frágeis: casas que não podem mais ser mais atacadas por nenhum peão, devido ao avanço dos mesmos, são passíveis de ocupação por uma peça que não poderá ser facilmente incomodada (normalmente um cavalo). A ocorrência de tais casas deve gerar variações correspondentes no valor da posição de forma a fazer com que o motor evite, sempre que possível, a criação das mesmas, e que explore adequadamente tais casas no campo adversário.
- Peças en prise: peças que não estão defendidas devem sempre ser evitadas pelo motor, que da mesma forma avalia positivamente para si a ocorrência de tais peças no bando adversário.

2.5. Ordenação de lances

Xadrez é um jogo complexo e mesmo a melhor das rotinas de avaliação não consegue ponderar todos os fatores melhor que um jogador amador forte. Para construir um motor que vença os mais fortes jogadores profissionais, a chave é pesquisar profundamente a árvore de lances, de forma que o motor consiga 'enxergar' tão mais à frente que o oponente que isto compense a má avaliação da posição.

Isto não seria possível se, em ordem de avaliar uma posição, o motor precisasse olhar todas as respostas possíveis a todos os lances possíveis, e assim por diante. Por isto, como vimos em busca, o algoritmo MiniMax tem importância apenas histórica, pois sua utilização é nula mesmo entre os mais amadores programas de xadrez. Sua melhoria mais significativa, o AlfaBeta, é largamente utilizada, pois reduz grandemente a árvore mínima de pesquisa.

Infelizmente para os motores de xadrez, o AlfaBeta, em seu pior caso, degenera para o MiniMax, e para evitar que isto aconteça deve-se objetivar sempre pesquisar o melhor lance primeiro. Isto gera um irônico paradoxo: se soubéssemos o melhor lance a pesquisar, não precisaríamos realizar a pesquisa, bastaria jogar o lance.

Os motores não podem, obviamente, saber qual o melhor lance antes de pesquisar, e com o objetivo de reduzir o grau da árvore a pesquisar, a comunidade conta com várias técnicas, hoje já solidamente estabelecidas, para estimar quais os melhores lances em uma posição, de forma que estes sejam pesquisados primeiro.

A árvore de lances do xadrez possui em média grau 35, o que claramente inviabiliza a pesquisa completa mesmo em profundidades baixas como 7 (O que traduz em aproximadamente 66 bilhões de posições, entre folhas e nós, que um bom motor em um hardware rápido levaria quase 10 horas para pesquisar).

As técnicas a serem apresentadas conseguem comprovadamente reduzir o grau da árvore para abaixo de 7, e vários motores têm reportado obter árvores de grau 2 (o que possibilitaria a pesquisa da mesma árvore de profundidade 7 citada anteriormente em meros 0,1 milésimos de segundo).

Imediatamente após gerar os lances, todo bom motor de xadrez deve aplicar algumas das técnicas abaixo para priorizar lances potencialmente bons. Normalmente junto ao lance gerado existe um atributo de valor preenchido pela própria rotina de geração para orientar a ordenação e busca dos mesmos. As técnicas para preenchimento destes valores estão apresentadas na ordem em que são mais comumente aplicadas:

2.5.1 A Variante principal

A variante principal, ou PV (de principal variation), fornece um excelente primeiro lance a pesquisar quando se usa

o Aprofundamento Iterativo (que, na prática, é utilizado em 99% dos motores), pois representa o melhor lance encontrado na pesquisa da profundidade anterior. É, portanto, o mais forte candidato da profundidade seguinte.

A técnica é simples e recompensadora: a geração de lances deve atribuir um valor tão alto a este lance que a ordenação obrigatoriamente indique-o como primeiro a pesquisar.

2.5.2 Heurística 'Killer Move'

Também amplamente utilizada, esta heurística parte do princípio de que se um lance gerou uma poda em outro ponto da árvore, na mesma profundidade, provavelmente é um bom lance e deve ser pesquisado primeiro em outros nós.

Na prática, guarda-se um número arbitrário de lances por profundidade em uma tabela (comumente 2), e toda vez que há uma poda, caso o lance já esteja em uma das posições, incrementa-se um contador para que se saiba quão freqüentemente este lance produziu podas. Quando o lance não está entre as posições, o motor o inclui em uma posição vazia, ou substitui o lance menos utilizado pelo novo.

Na geração de lances, o motor então verifica se o lance gerado é um 'killer move' para a profundidade que está sendo pesquisada, e neste caso ordena o lance de forma que o mesmo seja priorizado na busca.

A ordem entre os 'killer moves' pode ser obtida através do contador de uso, que deve indicar a força relativa do lance.

2.5.3 Promoções

Promoções de peões alteram drasticamente o material no tabuleiro, pois a peça de menor valor é substituída por outra de muito maior valor, quase sempre uma Dama. Há, pois, um ganho líquido potencial de 8 'peões', tornando o lance um candidato muito forte a melhor lance.

2.5.4 Capturas

Capturas são, juntamente com promoções, o único meio de alterar o equilíbrio material, mas ocorrem com muito mais freqüência. Em qualquer posição, entretanto, há diversas capturas legais para serem feitas, mas a maior parte delas absolutamente inútil, que não seriam sequer consideradas por um humano, tais como uma Dama capturando um peão protegido por outro peão.

Embora as capturas devam ser priorizadas na busca, a identificação de quais capturas são boas é a chave para uma boa ordenação, e dois esquemas são os correntemente utilizados para tal:

2.5.4.1 'MVV/LVA'

O nome complicado desta técnica vem dos acrônimos em inglês para Vítima Mais Valiosa / Atacante Menos Valioso ('most valuable victim' / 'less valuable attacker'), e significa exatamente isto: dividir o valor da vítima pelo valor do atacante.

Desta forma, espera-se priorizar os lances onde uma peça pouco valiosa captura uma peça muito valiosa, por exemplo quando um peão captura uma dama. Da mesma forma, a captura inversa terá baixa prioridade.

A principal vantagem desta técnica é o custo: praticamente nada. Como ambas as peças envolvidas na captura são conhecidas pela rotina de geração de lances, basta dividir o valor das duas para obter um índice bem razoável da qualidade da captura. Entretanto, muitas más capturas ainda serão pesquisadas inutilmente através desta técnica. Enquanto uma captura por uma torre de um cavalo defendido possuirá valor 5/3, uma captura de um peão não defendido por uma dama possuirá valor 1/9, ainda que o melhor lance muito provavelmente seja a captura do peão.

Pensando neste problema alguns desenvolvedores começaram a utilizar uma idéia denominada 'SEE (Static Exchange Evaluation)'.

2.5.4.2 'Static Exchange Evaluation (SEE)'

O conceito da idéia 'SEE' é quantificar a qualidade das capturas. Para tal, não há uma estratégia específica, ou mesmo um consenso geral no meio sobre como implementar a idéia. Em geral, se quer uma solução que não seja tão lenta a ponto de inviabilizar seu uso.

As soluções geralmente envolvem tabelas de ataque complexas, que guardam tanto o alvo do ataque quanto a origem, de forma a poder identificar o valor da peça atacante. Isto é particularmente útil em casos onde há várias capturas na mesma casa, e alguns motores ordenam as peças que atacam/defendem a casa por valor, de forma a capturar sempre com a peça de menor valor disponível, e quantificar ao final se a captura é boa.

Note que tais cálculos 'estáticos' (daí o nome) podem ser completamente refutados por um lance intermediário, como um xeque, ou outra ameaça. Assim, a função que realiza a avaliação 'SEE' deve ser tão leve quanto possível, e deixar parte do trabalho para própria busca que, dadas velocidade e profundidade suficientes, naturalmente achará os erros nas capturas.

2.5.5 Heurística de Histórico de Lances

Tão utilizada quanto a 'killer move', esta heurística também guarda lances que geraram podas, mas de forma menos clara: tais lances incrementam uma tabela referenciada pela casa de origem e casa de destino da peça. Comumente o incremento é feito da forma 2^P (P é a profundidade), isto é, o bit correspondente à profundidade é ligado, de forma que lances que geraram podas em profundidades maiores tenham maior prioridade.

Note que a tabela histórica deve ser inicializada a cada início de pesquisa, o que garante que a notação casa origem/casa destino não seja ambígua, pois se referencia à mesma posição. Da mesma forma que na heurística anterior, a geração de lances associa lances encontrados na tabela histórica a valores altos, que façam os mesmos serem pesquisados antes.

2.5.6 Tabelas De/Para

Para avaliar todos os demais lances que não se encaixam nas técnicas anteriores, normalmente se atribui um valor dependendo da peça e da casa de destino e origem. Para isto, desenvolvem-se tabelas que, seguindo a teoria do xadrez, quantificam quais são as melhores casas para determinadas peças.

É claro que, sendo estáticas, tais tabelas seguem apenas o bom senso de centralizar peças para obter a maior mobilidade e influência possível, mas falham em detectar exceções inerentes a cada posição. A idéia é que, se tais exceções existem, sejam rapidamente identificadas nas primeiras buscas e passem a fazer parte de algumas das heurísticas acima.

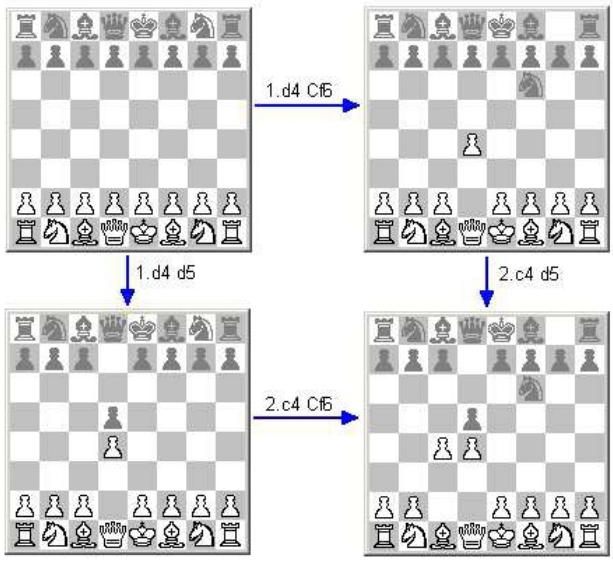

Figura 13. Duas alternativas para uma mesma posição.

3. APERFEIÇOANDO UM MOTOR DE XADREZ

As técnicas descritas na Seção 2 são fundamentais em qualquer implementação de um motor de xadrez. Elas não são, porém, suficientes para codificar um motor que seja sequer capaz de vencer um forte jogador amador, quanto mais um Grande-Mestre profissional.

Como conseguem, então, os motores comerciais (e mesmo alguns amadores), vencer a maioria dos jogos contra a elite do xadrez mundial?

Esta Seção trata das principais técnicas avançadas conhecidas hoje que devem ser aplicadas em um motor para se obter bons resultados contra forte oposição, humana ou não. Porém o arsenal de ferramentas dos motores cresce a cada dia, graças a uma extensa e qualificada comunidade de programadores experimentando diariamente novas idéias.

3.1. Tabelas de transposição

É consenso geral entre os programadores de motores que não existe um bom motor de xadrez que não implemente uma tabela de transposição.

O xadrez é por natureza um jogo de transposição, isto é, duas ou mais seqüências diferentes de lances freqüentemente produzem a mesma posição no tabuleiro (ver Figura 13). Jogadores humanos entendem isto instintivamente, e naturalmente economizam esforço de cálculo em posições que já avaliaram.

Mas esta compreensão não é tão natural nos motores. Se não houver uma ferramenta capaz de identificar que uma posição já foi pesquisada, muito tempo será gasto em trabalho já realizado [16].

Esta é a função primeira das tabelas de transposição: armazenar as posições pesquisadas durante uma busca, com seu valor correspondente, em uma tabela, e a cada posição verificar se a mesma já foi pesquisada antes. Neste caso, o motor recupera o valor diretamente da tabela, sem a necessidade de continuar a busca daquele ponto, sendo evidente o ganho de processamento ao se utilizar esta técnica [22].

A implementação mais comum de tabelas de transposição utiliza a técnica de hashing. Sendo o número de posições possíveis em uma partida de xadrez, na prática, infinito, o programador deve lidar com a certeza de colisões. A prática comum dos motores é manter apenas uma entrada por chave, e em caso de colisões a entrada é substituída.

Da mesma forma é importante uma geração eficiente da chave hash com uma função que garanta a distribuição ótima das posições pela tabela, e que também evite chaves próximas para posições semelhantes no tabuleiro. Esta providência é necessária para garantir que as colisões, quando ocorram, produzam o menor impacto possível, já que corresponderão a lances pesquisados em diferentes momentos do jogo.

Cada motor armazena informações diferentes em suas tabelas de transposição, mas os itens mais comuns são:

- Chave hash: a chave hash da posição atual do tabu-

leiro, utilizada para verificar se a entrada na tabela corresponde à posição procurada (identificando colisões).

- Valor: corresponde ao melhor valor da função de avaliação obtido para a posição em busca realizada anteriormente.
- Profundidade da avaliação: caso se queira saber a avaliação de uma posição à uma profundidade maior que a disponível na tabela de transposição, a entrada é ignorada e a busca prossegue normalmente. Ao final da busca, o valor obtido será atualizado na entrada da posição.
- Melhor lance: indica qual foi o primeiro lance da Variante Principal retornada pela busca ao final. Esta informação é utilizada pela rotina de ordenação de lances com grande ganho de performance no caso da utilização de algoritmos derivados do AlfaBeta.
- Tipo de retorno: utilizado em algoritmos AlfaBeta com janelas, indica se o valor da tabela foi obtido através de uma poda de beta, alfa, ou se o valor é preciso. Neste caso, apenas os valores precisos são utilizados dentro da busca, mas os demais podem ser utilizados para nortear as janelas, e ainda a ordenação de lances.

A tabela de transposição de um motor geralmente tem seu tamanho configurável por um parâmetro, de forma a poder ser aumentado em competições onde o hardware disponível é superior ao utilizado no desenvolvimento. Como o acesso acontece por indexação de memória, não há perda de performance na busca em tabelas grandes, enquanto existe ganho no número de posições guardadas, e consequentemente na força do motor.

3.2. Extensões e reduções de busca

Se os motores levam vantagem em relação aos jogadores humanos sobre a precisão e profundidade do cálculo, falham ao selecionar os lances. Mesmo com excelente ordenação, motores passam grande parte de sua busca olhando lances que os mais fracos jogadores humanos dispensariam instantaneamente.

Foi pensando nisto que vários programadores conceberam a idéia de extensões e reduções de busca. Estender a busca significa aumentar a profundidade que se quer pesquisar quando determinadas condições ocorrem, indicando que a posição é interessante e merece mais atenção. Da mesma forma, o objetivo das reduções é cancelar a pesquisa de uma variante o mais rápido possível, quando se percebe que a posição sendo pesquisada não merece atenção, ou há claros indícios de que um dos lados não deveria deixar tal posição ocorrer no tabuleiro. A Figura 14 ilustra o processo.

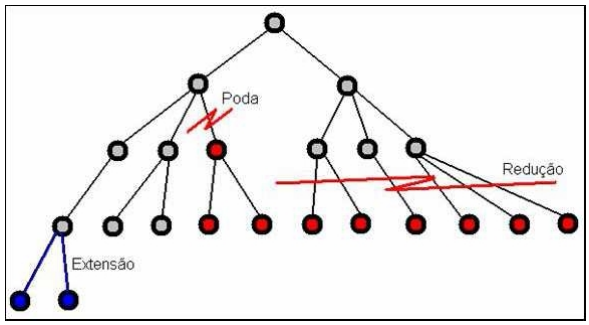

Figura 14. Ilustração de extensões e reduções.

3.2.1 Extensões

Extensões são muito mais comuns que as reduções pela simples razão de que requer menos responsabilidade aumentar a profundidade que reduzir. Uma redução mal-feita pode resultar na poda de uma variante fundamental, e implicar na derrota da partida. Uma extensão mal feita traz o único malefício de aumentar o tempo da busca e possivelmente reduzir a profundidade total. Assim, são necessárias várias extensões erradas para igualar o mal que uma só redução pode fazer.

Outro bom motivo para a preferência dos programadores pelas extensões é a maior facilidade de identificar quando uma posição merece mais atenção. As seguintes extensões são as mais utilizadas:

- Extensão de xeque: terminar uma busca em um xeque é claramente uma péssima idéia, pois a posição pode estar 'quieta' no momento (sem capturas a serem realizadas), mas uma seqüência de xeques levar a um ganho de material ou mate. Esta é a extensão mais importante a ser codificada (e em muitos motores, a única).
- Extensão de recaptura: No xadrez há várias seqüências forçadas que envolvem recapturas. Ao terminar uma busca em uma posição onde uma recaptura é o lance mais óbvio, a busca quiescence certamente levará a mesma em consideração, mas juntamente com outras várias capturas. Recapturar antes de entrar na busca quiescente poupa trabalho à mesma, e tempo em geral.
- Extensão de promoção: Quando um peão está preste a ser promovido, é interessante estender a busca para evitar que a variação brusca no equilíbrio material

decorrente da promoção deixe o motor em uma má posição.

- Extensão de lance único: Quando há apenas um lance possível, pode-se estender a busca em um 'ply' (lance de apenas um dos jogadores) sem perda de performance por motivos óbvios.

- Extensão por ameaça de mate: É fácil ver que não é bom terminar uma busca se existe uma ameaça de mate, pois o motor precisa avaliar se a ameaça de mate pode ser facilmente defendida, se será necessário abrir mão de material, ou se o mate é forçado e toda a linha deveria ser descartada. Felizmente a redução por lance nulo (que será vista adiante) torna muito simples a detecção de ameaça de mate.

Várias outras idéias para extensões surgem freqüentemente e são testadas em diferentes motores amadores e profissionais todos os dias. Estender se a segurança do Rei diminuir rapidamente, ou em casos de sacrifícios para eliminar as defesas, são exemplos de boas idéias em desenvolvimento atualmente.

Muitas extensões funcionam apenas para um determinado motor. O mais famoso exemplo é a chamada 'Extensão Singular', introduzida pela equipe do Deep Blue. Esta conturbada idéia sugere que se estenda a busca caso o motor perceba que a variante sendo pesquisada tem valor muito maior que todas as demais. Na teoria, a idéia é procurar por possíveis armadilhas na posição. Em geral, esta extensão foi abandonada pela comunidade de programadores, e assume-se que gerava ganho de performance apenas no caso do Deep Blue, devido à qualidade e tipo de hardware disponível.

Extensões podem produzir um efeito indesejado se utilizadas sem alguns cuidados. Aumentar a profundidade de busca durante a mesma pode ocasionar em uma 'explosão de sub-árvore', onde o número de extensões aumenta indefinidamente o tamanho da árvore a pesquisar, e a busca pode não terminar dentro do tempo alocado, ou mesmo não terminar nunca. Para evitar isto, muitos autores estão utilizando uma técnica chamada 'Extensões Fracionárias', onde cada condição que dispararia uma extensão e incrementaria em um 'ply' a profundidade a pesquisar, agora incrementa uma fração de 'ply', de forma que apenas quando algumas das extensões ocorrem em conjunto, a busca é efetivamente estendida. Esta é uma abordagem segura e responsável das extensões e vem sendo praticada por cada vez mais autores.

3.2.2 Reduções

Até o início dos anos 90 as reduções eram vistas com muita reserva pela comunidade de programadores de motores, pois todas pareciam demasiadamente 'adivinhações' por parte do motor de quais linhas poderiam ser eliminadas.

Entre as reduções com relativa aceitação está a Redução de Futilidade, introduzida por Heinz [11] no motor Dark Thought (que seria a base do Deep Blue mais tarde), onde o motor avalia se o lance a ser pesquisado, não sendo uma captura, somado a uma chamada margem de futilidade (que, para evitar erros táticos, deve ser igual ou menor ao maior ganho posicional), não melhora alfa, o lance não é bom e não deve ser pesquisado. A técnica é teoricamente errada, mas os autores e praticantes esperam que o ganho de velocidade compense os erros ocasionais.

O primeiro grande sucesso entre as reduções foi publicado no jornal da ICCA (International Computer Chess Association, recentemente renomeada para ICGA, International Computer Games Association) em setembro de 1993, por Chrilly Donniger. A técnica, conhecida por Redução por Lance Nulo (null-move prunning), já era implementada por alguns autores, mas só então sua aplicação foi estudada com o cuidado necessário.

A redução por Lance Nulo implementa uma técnica utilizada desde sempre por jogadores humanos: avaliar uma posição permitindo que um lado faça dois lances seguidos. Esta aparente corrupção das regras do xadrez é excelente forma de identificar ameaças, tanto de um lado como de outro.

No caso da redução por lance nulo, a idéia é: se o lado a jogar, ainda que não jogando e conseqüentemente permitindo o outro a jogar duas vezes seguidas, obtém uma posição muito boa (ou, em termos do algoritmo AlfaBeta, obtém um valor superior a beta), claramente obteria uma posição tão boa ou melhor se fizesse algum lance. Assim, a busca pode retornar o resultado sem precisar pesquisar os lances do bando, pois a variante nunca será permitida pelo oponente [13].

Esta redução nunca é utilizada caso o lado a jogar esteja em xeque, pois isto significaria uma captura imediata do rei. Segue um trecho de código exemplificando a redução:

```
if (flagXeque) {
(...)
} else {
/* NULL-MOVE – redução fixa R=2 */
    if (RealizaNullMove() && profundidade > 2) {
        MakeNullMove(&tabPrincipal);
        valorNullMove = -AlphaBeta(profundidade-3,
            -beta, -beta + 1, &pvTemp);
        UnMakeNullMove(&tabPrincipal);
        if (valorNullMove >= beta) {
            return beta;
        }
    }
```

A redução por lance nulo elimina muitas buscas infrutíferas e permite ao motor pesquisar de um a dois plies mais profundamente, com praticamente nenhum custo. O único

efeito colateral encontrado é a má avaliação de 'zugzwang', termo alemão que dá nome a uma situação específica do xadrez onde qualquer lance de um bando traz fraquezas ou perda de material. Nestas situações, 'passar' a vez é uma vantagem.

A maioria dos autores remedia isto não realizando a redução quando o material fica reduzido, ocasião em que o 'zugzwang' ocorre com mais freqüência. Outros simplesmente ignoram a ocorrência, justificando que o ganho de performance compensa o problema, e que em tais posições o lado em 'zugzwang' normalmente está com o jogo perdido de qualquer maneira [15].

A redução por lance nulo pode também ser utilizada para identificar ameaças e, ironicamente, realizar extensões. Quando o valor obtido após o segundo lance do oponente não só for menor que beta, mas também muito menor que alfa, o oponente está ameaçando uma captura vencedora ou mesmo mate. Neste caso, uma extensão pode ser acionada.

3.3. Livro de aberturas

Geralmente no início de uma partida entre humanos os primeiros lances são efetuados com muito mais rapidez que os demais. Eles fazem assim porque orientam seus lances pela chamada teoria das aberturas, por sua vez baseada em séculos de prática magistral do jogo, que consolidou um grande conjunto de métodos e variantes consideradas corretas.

Todo bom jogador de xadrez possui uma ou mais aberturas prediletas, e ainda uma compreensão genérica dos lances de várias outras, que compõem o que podemos chamar de seu livro de aberturas. Os jogadores profissionais, também por necessidade, chegam a memorizar milhares de variantes. A escolha de variantes muitas vezes se dá por gosto, estilo, humor ou mesmo com a intenção de confundir o adversário.

Bons motores de xadrez possuem, da mesma forma, um livro de aberturas geralmente construído especificamente para as características do motor. Como a teoria de aberturas leva em consideração fundamentos sólidos de planejamento estratégico, que são certamente a pior habilidade dos motores de xadrez, deixar um motor de xadrez jogar sem um livro de aberturas o início de uma partida pode levá-lo a uma derrota rápida.

Livros de Aberturas são amplamente utilizados por motores de xadrez desde o programa MacHACK, em 1967 [23].

Codificar conhecimento suficiente para jogar naturalmente boas aberturas é, em geral, má idéia, porque tal conhecimento traz complexidade ao motor, e conseqüentemente baixa a performance geral. A utilização de livros de aberturas é, portanto, boa prática por utilizar a já frisada pré-computação para obter bons resultados com um mínimo de esforço. As implementações de livros de abertura não seguem qualquer padrão e são as mais variadas possíveis, principalmente porque não são consideradas absolutamente vitais para um bom motor, e também porque a performance não é fator crítico na codificação. Na prática, por pior que seja a programação, o motor não leva mais que frações de segundo para consultar o livro em busca de qual lance jogar, o que sempre é vantajoso em comparação a realizar uma busca.

Para dar variedade ao jogo de seus motores, os autores codificam mais de uma variante para uma mesma posição, e fazem o motor escolher aleatoriamente uma das variantes disponíveis. Alguns utilizam indicadores estatísticos para cada variante, de forma a guiar a probabilidade do motor escolher cada abertura. Note que isto é suficiente para simular diferentes livros de abertura dependendo de com qual cor o motor está jogando. Há ainda implementações com aprendizagem, onde o motor armazena em arquivos os resultados das partidas para mudar o peso da probabilidade de escolha das aberturas (HYATT, 2005). Também em tamanho (número de variantes), os livros variam muito, desde centenas até centenas de milhares de entradas.

A seleção de repertório de aberturas de um motor, nos programas de competição, é trabalho tão meticuloso que geralmente contrata-se um profissional especifico para este fim, normalmente um Grande-Mestre Internacional de xadrez. Esta pessoa tem o trabalho de escolher cuidadosamente cada reposta que o motor deve realizar em várias situações diferentes, e deve conhecer as características de jogo do motor para escolher variantes que possuam maior afinidade.

Um dos principais problemas na construção do repertório é a verificação de que, ao encontrar-se fora de seu livro de aberturas (ou seja, após o primeiro lance do adversário para o qual o motor não encontra resposta em seu livro), o programa 'entenda' a posição que se encontra. Quando a estratégia por traz da abertura é profunda demais para o motor, não raro este se vê em posição muito inferior logo ao sair de seu livro, porque não calcula em profundidade suficiente para entender a posição. Por isto, muitos autores propositadamente programam seus motores para pesquisar por um tempo maior (normalmente o dobro do tempo normal dedicado a cada lance) a primeira busca da partida.

Isto é mais comum nas aberturas chamadas 'gambitos', onde um jogador sacrifica material para obter compensação em desenvolvimento, tempo ou ataque, conceitos vagos que os motores têm dificuldades para mensurar.

3.4. Banco de dados de finais

Bancos de dados de finais (endgame databases) são tabelas comprimidas de posições com poucas peças restantes

no tabuleiro, que respondem objetivamente qual é o melhor lance disponível na posição, e qual é o resultado esperado, seja vitória, empate ou derrota.

Seu uso é possível em jogos estratégicos onde o número de peças reduz-se com o passar do jogo, como é o caso de Xadrez ou Damas, que foi efetivamente resolvido graças aos bancos de dados de finais [12]. É, portanto, de pouca utilidade em jogos como Go, onde o número de peças aumenta com o passar do tempo.

Tais tabelas são calculadas através de análise retroativa a partir das posições com resultado definido. Por exemplo, de uma posição de mate em um final de rei contra rei e torre, calcula-se todas as posições a partir das quais tal mate poderia ser efetuado em um lance, assim em diante. Tais posições são então guardadas nas chamadas 'tabelas de 3 homens' (relativo ao número de peças restantes no tabuleiro) com o lance que leva ao resultado esperado, e a distância até o resultado: neste caso, vitória (mate do lado com a vez) em 1 lance.

As posições que podem ser obtidas a partir de posições já armazenadas na tabela têm então suas distâncias até o resultado corrigidas para refletir o lado a jogar, e então igualmente armazenadas. É necessário verificar se a posição já foi armazenada antes para evitar trabalho desnecessário, ou atribuir equivocadamente uma distância até o resultado. Uma descrição mais detalhada sobre a forma de criação de tais tabelas pode ser obtida em [12].

O número de 'homens' para os quais pode-se construir um banco de dados de finais é obviamente dependente da complexidade do jogo. Para o jogo de Damas já foram calculadas tabelas com 10 peças restantes, enquanto que para o Xadrez é improvável que se alcance, mesmo em um futuro remoto, tal número.

No Xadrez, hoje, trabalha-se com tabelas de 3, 4 e 5 homens, com a tabela de 6 'homens' em processo de cálculo (já disponível para algumas posições), lembrando que o número de 'homens' inclui os dois reis, que são necessários para uma posição válida. O progresso é cada vez mais lento com o aumento de 'homens', assim como cresce a necessidade de espaço para armazenamento.

Embora muitos autores optem por codificar seus próprios bancos de dados, existem alguns bancos de dados de finais independentes disponíveis para utilização com a maioria dos programas, que variam entre si por técnicas de indexação e compressão. Embora as mais famosas sejam as tabelas Nalimov (computadas e programadas por Eugene Nalimov), existem outros bons autores como Ken Thompson e De Koning [20].

O uso dos bancos de dados pelos motores é bem simples e relativamente fácil de ser implementado (dependendo da interface do banco de dados utilizado): ao iniciar a busca de cada nó, o motor conta o número de peças restantes (geralmente tal número é calculado iterativamente e encontra-se já disponível) e, caso possua o banco de dados relacionado, consulta-o, obtendo uma resposta definitiva para a posição.

Bancos de dados de finais possibilitam ao motor de xadrez jogar finais de jogos praticamente sem falhas, com resultados realmente impressionantes [6]. Não é incomum, ao observar uma partida entre dois Grandes-Mestres humanos consultando um motor com acesso a tais tabelas, verificar erros 'grosseiros' de ambas as partes, com a avaliação da posição variando entre vitória para um e outro lado. O fato é que motores com tal informação muitas vezes reconhecem posições aparentemente equilibradas como vitórias incrivelmente distantes (mates em 50 lances ou mais), o que não é plausível para um cérebro humano, mesmo entre os melhores jogadores.

Mesmo com tal capacidade bruta, se tem sugerido que bancos de dados de finais podem não produzir o aumento de força imaginado, podendo até reduzir a força do motor [21]. Tais suposições baseiam-se nos seguintes argumentos:

- O uso de tabelas incompletas pode levar o motor a jogar lances que não compreendem (e não escolheriam caso realizassem uma busca normal), para colocá-lo em uma posição com a qual não saberá lidar.
- Existem falhas nas construções de tabelas de finais como, por exemplo, ignorar roques (pois não se pode saber se os lados ainda possuem tal permissão) e a regra de empates por 50 lances sem captura ou movimento de peões.
- O excesso de informação pode levar um motor a abandonar uma partida que poderia ter ganhado ou empatado, quando jogando contra humanos ou motores que não possuem acesso a bancos de dados de finais.
- O acesso excessivo ao armazenamento secundário do computador (discos rígidos e ópticos) pode reduzir a velocidade do motor e conseqüentemente sua performance.

O assunto ainda está aberto à discussão, e os resultados práticos até o momento têm sido contraditórios. Alguns autores têm realizado testes com o objetivo de mensurar o ganho de força decorrente do uso de tais bancos de dados, com respostas interessantes, como as obtidas por Allgeuer [2], que sugerem não haver benefícios na técnica.

3.5. Interface

O desenvolvimento de uma interface para motores de xadrez é, por si só, complexo e trabalhoso, e normalmente desvia a atenção dos programadores para assuntos diversos dos problemas de algoritmos e inteligência artificial que a

criação do próprio motor enfrenta. Assim, a maioria dos motores amadores utiliza interfaces gráficas prontas, através de protocolos desenvolvidos pelos criadores de tais interfaces, e aperfeiçoados pela comunidade envolvida na programação de motores. A comunicação é normalmente realizada através dos canais padrão de entrada e saída, em formato texto simples.

A mais tradicional interface é, de longe, o Winboard, gratuitamente disponível para Windows e Linux, desenvolvido por Tim MANN (1994) (ver Figura 15). Winboard possui um protocolo simples e direto de comunicação, que durante muito tempo foi a única opção dos programadores amadores, o que certamente auxiliou em sua posição de preferência atual.

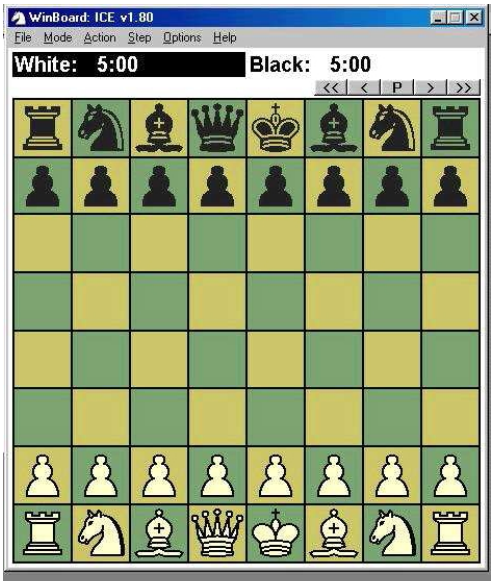

Figura 15. Interface do Winboard.

Posteriormente a software house alemã ChessBase, que já possuía uma bela interface própria para seus programas comerciais (Fritz, Júnior, Shredder), atendeu aos apelos da comunidade e criou um protocolo de comunicação que permitia que programas amadores a utilizassem. O protocolo foi batizado UCI (universal chess interface) e é considerado uma evolução em relação ao protocolo do Winboard. Hoje existem outras interfaces disponíveis, gratuitas e comercializadas, que, no entanto utilizam os dois protocolos mencionados.

4. TÉCNICAS USADAS NO PROJETO ICE

Nesta Seção são apresentadas as principais técnicas utilizadas no motor ICE [1].

4.1. Interface

ICE implementa a versão 2 do protocolo Winboard descrito na seção anterior. Deverá, portanto, funcionar corretamente em conjunto com quaisquer interfaces que implementem este protocolo.

4.2. Plataforma de desenvolvimento

ICE é um motor para o ambiente Win32 (Windows 98, Me, 2000, NT e XP), e foi completamente desenvolvido em linguagem C padrão, utilizando o ambiente Cygwin, (um simulador Linux gratuito para o ambiente Windows), utilizando-se apenas um editor de texto para a codificação.

Em versões recentes, ICE vem sido compilado com o Microsoft Visual C++ 2003 (gratuito, a partir do site da Microsoft), com ganho significativo de performance em relação ao código compilado pelo compilador GNU mais recente.

4.3. Modelo de representação

ICE foi desenvolvido baseando-se em bitboards para a representação interna de suas estruturas. Como já mencionado anteriormente, parece-nos que o ganho gratuito de performance após o estabelecimento (iminente) de processadores de 64 bits é argumento suficientemente forte para a escolha.

Entretanto, não foi implementada a utilização de bitboards rotacionados, o que possivelmente conferirá aumento de performance. Por motivos didáticos, preferiu-se uma técnica mais clara para a geração de lances.

4.4. Busca

Nas primeiras versões ICE implementava uma simples busca alfabeta, com aprofundamento iterativo. A busca foi gradativamente modificada para a inclusão de técnicas (descritas neste documento) como janela de aspiração, redução de lance nulo, e a utilização de tabelas de transposição, que possibilitaram um ganho em performance de aproximadamente 100% a 150%.

ICE utiliza uma tabela de transposição com tamanho configurável por parâmetro, e talvez guarde informação demais por célula, o que diminui a quantidade de posições armazenadas na tabela. No entanto, as informações hoje mantidas são as necessárias para a operação correta da tabela.

A redução por lance nulo utiliza uma profundidade de redução fixa em 2 (o que significa que para cada posição é realizada uma busca com profundidade igual à atual menos 2, 'passando a vez'). Uma alternativa é utilizar reduções variadas dependendo da profundidade na qual o motor está pesquisando no momento. Em experiências com o ICE, a

estratégia de profundidades variadas não funcionou tão bem quanto a fixa em 2.

ICE implementa apenas extensões de xeque, sem limitar o número de extensões. Uma extensão de implementação relativamente barata seria a de ameaça de mate, já que isto pode ser verificado no retorno da redução de lance nulo. Outras extensões dependem de maiores modificações no motor para que certas informações estejam disponíveis facilmente (peças ameaçadas e peças sem defensores, entre outras).

4.5. Ordenação de lances

Após muitos testes, ICE parece funcionar melhor com a seguinte ordem de lances:

- Lance da PV
- Lance da tabela hash
- Promoções para Dama
- Capturas boas (atacante menos valioso que atacado)
- Capturas ruins (atacante mais valioso que atacado)
- Killer moves
- Promoções para peças menores
- Demais lances

Entretanto, o fato de que hoje o maior defeito do ICE é seu alto branching factor (grau médio da arvore de busca) sugere que a ordenação de lances está insuficiente, e que melhorias podem ser feitas neste sentido.

ICE implementa a heurística de histórico (history heuristic), onde os lances que produzem cortes na busca alfabeta têm um contador incrementado. ICE utiliza esta informação para ordenar os lances dentro de uma mesma categoria, isto é, se há duas promoções para Dama disponíveis, ICE pesquisará primeiro o lance que no passado produziu mais cortes, na esperança de que este venha a produzir um corte e, conseqüentemente, melhore a performance da busca.

Dois killer moves são mantidos, mas ICE apenas coloca nestes lugares lances considerados 'quietos', ou seja, lances que não são capturas, promoções ou xeques. A idéia é selecionar, dos lances 'quietos', quais os mais capazes de gerar cortes.

4.6. Rotina de avaliação

A rotina de avaliação da posição do ICE não é exatamente 'minimalista', chegando a possuir alguns refinamentos interessantes, porém possui um número muito inferior de verificações quando comparada com as rotinas dos bons programas amadores.

Primeiramente, ICE realiza uma contagem material em conjunto com as tabelas peça-casa. Estas tabelas, que fornecem valores arbitrários para cada peça em cada casa, são um dos pontos mais 'sensíveis' do motor, e um ajuste fino nelas pode causar grandes alterações da força do jogo do motor, para melhor ou para pior. Por exemplo, os pesos usados para o cavalo estão mostrados na Tabela 1.

5	7	9	11	11	9	7	5
7	10	14	16	16	14	10	7
9	14	19	21	21	19	14	9
11	16	21	24	24	21	16	11
11	16	21	24	24	21	16	11
9	14	19	21	21	19	14	9
7	10	14	16	16	14	10	7
5	7	9	11	11	9	7	5

Tabela 1. Tabela de pesos peça-casa para o cavalo do ICE.

A posição do Rei ainda não é contabilizada porque sua tabela peça-casa varia de acordo com a fase do jogo: na abertura e meio-jogo com peças pesadas (Damas e Torres), o Rei deve ficar longe do centro e protegido. Nos finais, quando a maior parte das peças pesadas foi trocada, o Rei pode (e deve) encaminhar-se para o centro do tabuleiro e participar mais ativamente do jogo.

Em seqüência, o motor avalia se a posição é um final de Reis e peões, informação que posteriormente usa para bonificar decisivamente as tais situações quando o lado a jogar possui peões a mais, posições onde normalmente o lado mais forte ganha facilmente. Esta bonificação adicional (além da contagem material) faz com que o programa favoreça seqüências de lances que levem a tais posições.

ICE verifica também se os bandos realizaram roque para lados opostos, situação onde normalmente o jogo ganha caráter agressivo, com ambos os lados avançando seus peões para fustigar o Rei adversário. ICE usa esta informação para aumentar a bonificação de avanços de peão em direção ao Rei adversário, já que não estará fragilizando o próprio Rei com tais medidas.

Há uma avaliação da diferença de material, que é importante para quando há equilíbrio material com peças diferentes (duas torres contra dama, ou duas peças menores contra torre e peão), pois em tais situações a teoria estabelece que algumas combinações são superiores a outras, ainda que em uma contagem exista equilíbrio. O motor deve conhecer esta informação para evitar realizar trocas desfavoráveis.

Este é outro ponto onde nos parece poder haver grande melhoria, e provavelmente é o melhor local para se verificar

se há material para mate, para evitar simplificações excessivas que deixam o motor sem possibilidade de vitória (ICE ainda não verifica isto). Posteriormente, ICE verifica a estrutura de peões, penalizando peões dobrados, isolados e atrasados em coluna aberta. Também há bonificações para peões passados, que aumentam de acordo com a proximidade da promoção.

A segurança do Rei é avaliada através de uma fórmula complexa, e falha em certos aspectos, que considera o tamanho e forma do escudo de peões à frente do Rei, bem como colunas abertas à frente e ao lado da casa onde está o Rei. Mais uma vez, a segurança só é verificada quando há peças pesadas no tabuleiro.

Na análise de torres, bonifica-se a colocação das mesmas em colunas semi-abertas, aumentando-se a bonificação quando a coluna está completamente aberta. Também são bonificadas baterias (duas torres na mesma coluna) e a colocação da torre na sétima linha quando o Rei adversário está na oitava, ou há peões adversários em suas casas de origem.

ICE realiza uma análise de aberturas com conceitos comuns da teoria de aberturas, para seguir quando seu livro de aberturas chegar ao fim. Aqui, os lances costumeiros de peões centrais são bonificados, assim como realizar o roque cedo. São penalizados lances como mover peças duas vezes, expor Dama cedo demais, bloquear peões centrais com peças e mover o Rei (exceto para o roque).

Os bônus e penalizações são ajustados para que ICE não evite ganhar ou defender material em troca dos conceitos de abertura.

4.7. Livro de aberturas

ICE implementa um livro de aberturas simples, mas com algumas sofisticações, como escolha de linhas através de um percentual.

Comentários são reconhecidos por linhas iniciadas com o caractere #, e os lances utilizam notação de coordenadas.

A implementação de livro de aberturas do ICE não reconhece inversões, isto é, posições iguais alcançadas por ordem diferente de lances. Neste caso, as duas seqüências devem estar representadas no livro, o que introduz possíveis inconsistências.

4.8. Performance e testes

4.8.1 Nós por segundo e Branching Factor

Os dois índices de performance mais populares entre os motores são nós por segundo (NPS) e o branching factor (grau médio da árvore de busca). Estes podem ser calculados a partir da informação padrão que a maioria dos motores fornece durante uma busca: profundidade, valor (no caso do ICE, em centésimos de peão), tempo (no caso do ICE, em centésimos de segundos), número de nós pesquisados, e a melhor linha encontrada.

A performance medida em NPS varia imensamente dependendo da capacidade de processamento da máquina, disponibilidade do processador no momento, e de particularidades do próprio motor (existem motores com alto NPS que são facilmente derrotados por outros com baixo NPS).

Portanto, é uma medida mais utilizada para avaliar melhorias entre versões de um mesmo motor, mantendo-se o hardware tão constante quanto possível.

O branching factor é calculado dividindo-se o número de nós pesquisados em uma profundidade pelo mesmo número da profundidade anterior. Os melhores motores obtém branching factor médio abaixo de 2.

O branching factor é uma medida mais confiável para comparar a qualidade da busca de motores diferentes, independentemente do hardware utilizado. Porém não é o suficiente para concluir se um motor é mais forte que outro, já que não leva em consideração a qualidade da rotina de avaliação dos mesmos.

4.8.2 Suítes de Teste

As versões iniciais do ICE eram testadas apenas comparando-se os valores de NPS e branching factor, bem como resultado de jogos contra versões anteriores e outros motores. A partir da implementação do comando analyze do protocolo Winboard, entretanto, os testes ganharam qualidade, pois foi possível submeter ICE às chamadas suítes de teste.

Suítes de teste são arquivos com centenas de posições (em notação conhecida por Forshyte), seguidas do melhor lance encontrado para a posição.

Existem programas (como o excelente epd2wb, criado pelo autor de motores Bruce MORELAND) que recebem como parâmetros um arquivo contendo uma suíte de testes, e um motor padrão Winboard, e submete o mesmo a cada posição por um número especificado de segundos (normalmente 20), e ao final compara o lance obtido com o melhor lance fornecido pela suíte. Esta é uma excelente forma de verificar a força do motor em diversos tipos de posições. Alguns autores utilizam diferentes suítes de teste dependendo do tipo de alteração que realizaram em seus motores. Sugere-se manter suítes específicas para testar temas estratégicos como jogo de peões, par de bispos, mates forçados, defesas por repetição de lances, etc.

No desenvolvimento do ICE utilizamos duas conhecidas suítes de teste, WAC (300 posições, em média fáceis), e OKelly (176 posições, difíceis). A Tabela 2 registra a evolução do desempenho das últimas versões do ICE.

ICE vem jogando desde outubro de 2005 em um servidor gratuito de xadrez na Internet, FICS (Free Internet Chess

Suíte	Okelly	(176)	WAC	(300)
Versão ICE	Acertos	B-factor	Acertos	B-factor
1.78	83	3.78	217	4.53
1.75	79	3.90	221	4.62
1.70	74	3.94	200	5.43
1.60	63	3.89	156	5.45
1.50	68	4.75	172	5.78

Tabela 2. Evolução do desempenho doi ICE

Server), com o login IceBR, e tem mantido seu elo rating em torno de 2000. Em um match de 5 partidas rápidas (5 minutos para 40 lances) contra um oponente humano de rating FEXERJ (Federação de Xadrez do Estado do Rio de Janeiro) 2000, ICE venceu por 3 a 2.

4.8.3 Melhorias

O branching factor em torno de 4 sugere existir muito trabalho a ser feito na rotina de busca do ICE, principalmente na ordenação de lances. Parte deste trabalho consiste em retirada de possíveis erros de programação, mas principalmente é necessário disponibilizar mais informações da posição à busca para que esta possa ordenar melhor os lances. O cálculo de uma tabela de ataques poderia, por exemplo, facilitar a ordenação de capturas, onde boas capturas seriam priorizadas. O processamento extra para tal seria compensado pela diminuição do branching factor, bem como em rotinas como verificação de xeque e roque, que calcular tais tabelas parcialmente, sem utilizar as informações. A rotina de avaliação seria igualmente beneficiada por tal tabela, já que poderia avaliar outros temas estratégicos como espaço, ataque ao rei, domínio de casas centrais, entre outros.

Para melhorar a busca, ICE pode aumentar seu repertório de extensões e reduções, das quais hoje faz pouco uso. A implementação da redução por lance nulo nas últimas versões já possibilita a implementação de extensão por ameaça de mate a custo baixo, por exemplo. Muitos motores vêm beneficiando-se de técnicas agressivas de redução como a Poda de Futilidade, e alguns testes poderiam ser feitos neste sentido.

Além disto, outras possibilidades de relativo baixo custo seriam a implementação de geração de lances por bitboard rotacionados e avaliação preguiçosa (que na verdade é uma espécie de redução por futilidade). Na rotina de avaliação, fazem falta heurísticas específicas para cada fase do jogo, e ainda o chamado tropismo (avaliação da proximidade de peças ao rei adversário).

Outras possibilidades que requerem maior esforço de programação seriam acesso a banco de dados de finais, geração iterativa de lances e implementação de uma segunda tabela de transposição.

Estimamos que tais melhorias poderiam elevar o nível de jogo do ICE ao de um jogador graduado com o título de MI (Mestre Internacional), aproximadamente 2400 de rating.

5. CONCLUSÃO

Embora o caminho da evolução dos motores de xadrez seja ainda muito promissor, os resultados das últimas disputas entre homens e computadores sugerem que o marco de criar um programa que jogue melhor que qualquer humano foi alcançado. Hoje os motores já possuem conhecimento estratégico suficiente (o que antes era um ponto fraco), enquanto que taticamente superam por larga vantagem os homens.

O futuro competitivo dos programas está em competições entre motores apenas, cada vez mais freqüentes, e sempre acompanhadas com interesse pela mídia. Competições homens versus programas provavelmente serão menos freqüentes na medida em que os homens ofereçam cada vez menos resistência.

Recentemente, testemunhamos a vitória de 5,5 pontos a 0,5 ponto pelo super-computador Hydra contra o Grande Mestre inglês Michael Adams, um jogador entre os 10 melhores do mundo [8]. Nos torneios nacionais argentinos, freqüentemente há a participação de um programa sendo executado em hardware padrão, e que invariavelmente ganha. Os programas de xadrez revolucionaram definitivamente a prática do jogo, auxiliando na preparação e estudo de profissionais e amadores, contradizendo teorias centenárias, e comprovando outras. Novas modalidades foram criadas para acomodar estes novos componentes, tais como o Xadrez Avançado, onde jogadores enfrentam-se sobre o tabuleiro, cada um munido de um motor para auxiliá-lo na escolha de seus lances.

Agora que vencer humanos é um objetivo realizado, o desenvolvimento de motores está diversificando-se em várias direções. No intuito de tornar os motores cada vez mais fortes, esforços têm se concentrado principalmente em processamento paralelo massivo, e desenvolvimento de hardware específico, como chips FPGA do motor Brutus [5].

Outro objetivo que desperta interesse cada vez maior é a curiosa tentativa de fazer com que um programa erre de forma humana. Para a indústria de entretenimento, programas que nunca perdem são interessantes apenas para especialistas e, portanto, de comercialização pouco atrativa. Entretanto, as tentativas de criar motores fracos até o momento resultaram em programas que ora erram de forma infantil, ora jogam muito acima de um jogador forte. Assim, um esforço específico tem sido realizado para identificar quais as características do pensamento humano que levam aos erros desejáveis, e então delinear as possibilidades de codificação de tal comportamento.

Mas a principal vertente ainda é o desenvolvimento de algoritmos e heurísticas para o estado-da-arte em força de jogo. Uma grata surpresa neste campo foi o aparecimento do programa Fruit, de Fabien Letouzey, cuja força equipara-se apenas ao melhor programa comercial disponível. Para surpresa de muitos, Fruit é um programa amador , de código-fonte aberto, escrito em linguagem C pura, e não apresenta nenhum conceito revolucionário. A principal força da criação de Fabien reside em sua lógica impecável e estilo de programação limpo, defensivo, e conseqüentemente livre de bugs. Com Fruit, a comunidade de programadores amadores renovou suas convicções de que vêm caminhando na direção correta, e ainda de que não há grandes segredos sendo mantidos pelas fabricantes dos programas comerciais.

O Brasil possui participação minúscula nas conquistas deste campo, o que é curioso, tendo em vista o destaque que nossos programadores possuem no mercado mundial de desenvolvimento de softwares. O baixo desempenho é causado provavelmente pelo baixo interesse da nossa indústria no desenvolvimento de jogos, ou ainda pela falta de pesquisa pós-graduada no assunto. Vale ressaltar que vários dos autores e pesquisadores neste campo são renomados doutores e pós-doutores de diversas nacionalidades.

Esperamos que este documento, bem como o motor ICE, possam despertar algum interesse nacional neste assunto que vem movimentando tantas grandes mentes ao redor do mundo, colhendo frutos para a pesquisa e mercado.

Referências

[1] Abreu, C.M, Lazoski, Y.N e Waghabi, E.. *Xadrez e Computação*, monografia de Projeto Final, UERJ/RJ, 2006.

[2] Allgeuer, R. *Winboard forum: yace bitbase test*, 2003, disponível online em http://web.archive.org/web/20041013083410/http://f1-1.parsimony.net/forum16635/messages/49266.htm, visitado em 22 de outubro de 2005.

[3] Carvalho, A. *Introdução à inteligência artificial: jogos e busca competitiva*, 2003, disponível online em http://www.icmc.usp.br/ andre/aulas/IA/aula-ia04.pdf, visitado em 22 de outubro de 2005.

[4] Friedel, F. *A short history of computer chess*, Revista CHESSBASE, 2002, disponível online em http://www.chessbase.com/columns/column.asp?-pid=102, visitada em 22 de outubro de 2005.

[5] Revista CHESSBASE, 2002, disponível online em http://www.chessbase.com/newsdetail.asp?newsid=-221, visitada em 22 de outubro de 2005.

[6] Revista CHESSBASE, 2002, disponível online em http://www.chessbase.com/newsdetail.asp?newsid=-239, visitada em 22 de outubro de 2005.

[7] Jimenez, R. *The Rook Endgame Machine of Torres y Quevedo*, Revista CHESSBASE, 2004, disponível online em http://www.chessbase.com/newsdetail.asp?newsid=-1799, visitada em 22 de outubro de 2005.

[8] Revista CHESSBASE, 2005, disponível online em http://www.chessbase.com/newsdetail.asp?newsid=-2476, visitada em 22 de outubro de 2005.

[9] Eppstein, D. *Strategy and board game programming*, 2001, disponível online em http://www.ics.uci.edu/-eppstein/180a/970415.html, visitado em 22 de outubro de 2005.

[10] Hyatt, Robert. *Online technical papers*, 2003, disponível online em http://www.cis.uab.edu/-hyatt/pubs.html, visitado em 22 de outubro de 2005.

[11] Heinz, E. *Darkthought: extended futility prunning*, 1998, disponível online em http://supertech.-lcs.mit.edu/ heinz/dt/node18.html, visitado em 22 de outubro de 2005.

[12] Fierz, M. *Strategy game programming: endgame databases*, 2004, disponível online em http://www.fierz.ch/strategy3.htm, visitado em 22 de outubro de 2005.

[13] Frayn, C. *Computer chess programming theory*, 2005, disponível online em http://www.frayn.-net/beowulf/theory.html, visitado em 22 de outubro de 2005.

[14] Marsland, T. A. *Computer chess and search*, 1991, disponível online em http://www.cs.ualberta.ca/-tony/RecentPapers/report.mac.pdf, visitado em 22 de outubro de 2005.

[15] Moreland, B. *Computer chess: programming topics*, 2001, disponível online em http://www.brucemo.-com/compchess/programming/index.htm, visitado em 22 de outubro de 2005.

[16] Norvig, P. e Russel, S. *Inteligência Artificial.* 2a.ed, São Paulo, Ed. Campus, 2003.

[17] (Philosophical Magazine 41 256,275) em 1951.

[18] Scott, J *Chess GA experiments*, 2002, disponível online em http://satirist.org/learn-game/methods-/ga/chess.html, visitado em 22 de outubro de 2005.

[19] Schroder, E. R. *Programming stuff*, 2003, disponível online em http://members.home.nl/ matador/chess840.htm, visitado em 22 de outubro de 2005.

[20] Tay, A. *A guide to endgames tablebase*, 2001, disponível online em http://www.aarontay.per.sg/winboard/egtb.html, visitado em 22 de outubro de 2005.

[21] Tay, A. *Can use of endgame tablebases weaken play?*, 2003, disponível online em http://www.aarontay.per.sg/winboard/weaktablebase.html, visitado em 22 de outubro de 2005.

[22] Walker, A. N. *G13GAM: game theory*, 1997. Disponível online em http://www.maths.nott.ac.uk/ personal/anw/G13GT1/compch.html, Visitado em 22 de outubro de 2005.

[23] Wall, W. *Computer chess history*, 2004, disponível online em http://www.geocities.com/SiliconValley/-Lab/7378/comphis.htm, visitado em 22 de outubro de 2005.