v0 v0 v0

atualizar atualizar atualizar

O Gambito do Explanador: Construindo uma Engine de Xadrez Interpretável por Humanos

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Felipe Zan Coelho
Orientador: Prof. Lucas Nascimento Ferreira

Projeto Orientado à Computação 2

29/06/2025

Sumário

1 Introdução

O xadrez representa uma das mais interessantes intersecções entre o intelecto humano e a computação algorítmica. Desde as suas origens no século VI na Índia, evoluindo através do Chaturanga até sua forma moderna europeia do século XV, o jogo transcendeu fronteiras culturais para se estabelecer como um paradigma de pensamento lógico e planejamento sequencial (preciso citar). A natureza determinística do xadrez, combinada com sua complexidade combinatória, o posicionou naturalmente como um terreno fértil para o desenvolvimento e teste de algoritmos de inteligência artificial.

1.1 Contexto Histórico das Engines de Xadrez

A jornada computacional do xadrez iniciou-se talvez em meados do século XX, quando pioneiros da computação, Alan Turing e David Champernowne conceberam o Turochamp em 1948, um dos primeiros algoritmos teóricos para o jogo (preciso citar). Embora limitado pela tecnologia da época, este trabalho inicial estabeleceu fundamentos conceituais que perduraram por décadas de desenvolvimento subsequente.

O progresso das engines de xadrez espelha a própria evolução da computação moderna. Na década de 1970, programas como Chess 4.0 começaram a demonstrar competência suficiente para desafiar jogadores humanos de nível intermediário (preciso citar). O Cray Blitz, em 1981, tornou-se o primeiro programa a alcançar o título de Mestre, com rating FIDE de 2258, marcando um ponto de inflexão na percepção pública sobre o potencial da inteligência artificial no xadrez (preciso citar).

O momento mais emblemático desta evolução ocorreu em 1997, quando o IBM computador Deep Blue derrotou Garry Kasparov, então campeão mundial, em um match histórico que capturou a atenção global (preciso citar). Este evento demonstrou a superioridade computacional bruta, e também sinalizou o início de uma era onde as engines de xadrez se tornariam ferramentas centrais para análise, treinamento e compreensão do jogo.

Atualmente, engines como Stockfish e Leela Chess Zero operam em patamares de força vastamente superiores a qualquer jogador humano, com ratings estimados superiores a 3600 Elo (preciso citar). Estas engines representam arquiteturas distintas: Stockfish exemplifica a abordagem tradicional baseada em busca alfa-beta otimizada e avaliação heurística refinada, enquanto Leela Chess Zero incorpora redes neurais profundas inspiradas no paradigma popularizado pelo AlphaZero (preciso citar).

1.2 O Problema da Explicabilidade em Sistemas de Xadrez

A ascensão das engines super-humanas trouxe consigo uma questão ainda pouco tratada. Enquanto estas ferramentas produzem análises de melhor movimento de qualidade incomparável, a maneira como são projetadas frequentemente resulta em recomendações opacas, desprovidas de contexto interpretativo acessível aos praticantes humanos. Este fenômeno manifesta-se particularmente na emergência do que jogadores denominam informalmente como "jogadas de computador", que são movimentos aparentemente contraintuitivos que, embora objetivamente superiores, carecem de lógica imediatamente perceptível (preciso citar).

A opacidade das engines modernas contrasta com a necessidade humana de compreensão causal. Jogadores de todos os níveis beneficiariam-se de explicações estruturadas para internalizar conceitos estratégicos e aprimorar seu entendimento posicional. As interfaces tradicionais, mediadas primariamente pelo protocolo UCI (Universal Chess Interface), apresentam apenas variantes principais e avaliações numéricas, deixando aos usuários a tarefa de decodificar as intenções subjacentes (preciso citar).

Tal lacuna torna-se particularmente pronunciada no contexto educacional. Instrutores e estudantes frequentemente se encontram diante de recomendações computacionais brilhantes, porém inexplicáveis, criando uma dependência problemática onde a autoridade algorítmica substitui, ao invés de complementar, o desenvolvimento do julgamento enxadrístico independente.

1.3 Objetivos do Projeto e Contribuições

Este trabalho propõe uma abordagem inicial para o desenvolvimento de uma engine de xadrez que mantém capacidade algorítmica enquanto proporciona explicabilidade interpretativa robusta. O objetivo principal consiste na criação de um sistema que identifica movimentos utilizando busca profunda e articula as razões estratégicas e táticas subjacentes de forma acessível a praticantes humanos.

As contribuições deste projeto manifestam-se em quatro direções. Primeiramente, desenvolvemos uma arquitetura de explicabilidade que coleta métricas interpretativas durante o processo de busca, incluindo medidas de concretude tática, análise de risco posicional e estabilidade avaliativa, sem comprometer consideravelmente a performance algorítmica.

Em segundo lugar, implementamos umsistema de templates contextuais baseado em conhecimento especializado estruturado, capaz de gerar explicações textuais apropriadas ao contexto posicional através de 83 templates categorizados por temas estratégicos. A terceira contribuição reside na definição de métricas quantificáveis como concretude (densidade de elementos táticos na variante principal), risco (variância das avaliações ao longo da busca) e profundidade de estabilidade (convergência avaliativa), fornecendo vocabulário algorítmico para aspectos tradicionalmente qualitativos.

1.4 Organização do Documento

Este relatório estrutura-se em dez seções que abordam aspectos particulares do desenvolvimento e validação da engine referendada. A Seção 2 estabelece os fundamentos teóricos dessas ferramentas. As Seções 3 a 7 detalham os componentes arquiteturais principais: geração de movimentos, sistema de busca, avaliação heurística, explicabilidade e comunicação externa.

Já a seção 8 apresenta a metodologia e resultados dos testes realizados, incluindo análises de performance computacional e estimativas de força relativa. A Seção 9 discute os resultados obtidos, identificando limitações e oportunidades de melhorias futuras. Por fim, a seção 10 sintetiza as contribuições e delineia direções para trabalhos futuros.

2 Fundamentação Teórica

Esta seção estabelece os alicerces conceituais necessários, abordando os princípios matemáticos de teoria dos jogos e especificidades técnicas encontradas durante o desenvolvimento de algoritmos para Xadrez.

2.1 Teoria dos Jogos Aplicada ao Xadrez

O xadrez enquadra-se na categoria que a teoria dos jogos denomina como *jogos de soma zero com informação perfeita*. Esta classificação comunica as propriedades matemáticas que permeiam o jogo e modelam o desenvolvimento algorítmico que foi realizado.

Podemos modelar o xadrez como uma tupla $G = (S, S_0, P, A, T, U)$, onde:

- S representa o conjunto de todos os estados possíveis do jogo (configurações válidas do tabuleiro)
- $S_0 \in S$ denota o estado inicial padronizado
- $P = \{\text{Brancas}, \text{Pretas}\}$ define os dois jogadores
- $A(s)$ especifica o conjunto de ações (movimentos) válidas no estado s
- $T(s, a)$ descreve a função de transição que mapeia estado-ação para novo estado
- $U : S_{\text{terminal}} \times P \to \mathbb{R}$ define a função de utilidade nos estados terminais

A propriedade de soma zero manifesta-se na relação $U(s, \text{Brancas}) = -U(s, \text{Pretas})$ para qualquer estado terminal s. Esta característica implica que todo ganho de um jogador corresponde exatamente à perda equivalente do oponente, eliminando possibilidades de cooperação ou benefício mútuo (preciso citar).

A informação perfeita garante que ambos os jogadores possuem conhecimento completo do estado atual a cada momento, sem elementos ocultos ou probabilísticos após a determinação das cores iniciais. Esta propriedade, combinada com o determinismo absoluto das regras, estabelece que o xadrez possui um valor teórico definido – isto é, com jogo perfeito de ambos os lados, o resultado é predeterminado, embora permaneça computacionalmente inacessível devido à complexidade combinatória (preciso citar).

2.2 Algoritmos de Busca Adversarial

O paradigma central para decisão em jogos adversariais baseia-se no algoritmo Minimax, que explora recursivamente a árvore de possibilidades futuras assumindo jogo ótimo de ambos os participantes.

2.2.1 Algoritmo Minimax

O Minimax opera através da definição recursiva:

$$\text{Minimax}(s,p) = \begin{cases} U(s,p) & \text{se } s \in S_{\text{terminal}} \\ \max_{a \in A(s)} \text{Minimax}(T(s,a),p) & \text{se jogador}(s) = p \\ \min_{a \in A(s)} \text{Minimax}(T(s,a),p) & \text{se jogador}(s) \neq p \end{cases}$$

onde p representa o jogador para o qual calculamos o valor. O algoritmo propaga valores desde os nós terminais até a raiz, com cada jogador otimizando sua utilidade esperada assumindo que o oponente fará o mesmo (preciso citar).

2.2.2 Otimização Alpha-Beta

A complexidade exponencial do Minimax puro ($O(b^d)$ onde b é o fator de ramificação e d a profundidade) torna-se rapidamente intratável. A poda alpha-beta proporciona uma otimização elegante que mantém garantias de optimalidade enquanto reduz dramaticamente o espaço de busca explorado.

O algoritmo mantém dois parâmetros dinâmicos:

- α: melhor valor garantido para o jogador maximizador
- β: melhor valor garantido para o jogador minimizador

A poda ocorre quando $\alpha \geq \beta$, indicando que o ramo atual não pode influenciar a decisão final. No melhor caso, com ordenação ótima de movimentos, a complexidade melhora para $O(b^{d/2})$, efetivamente dobrando a profundidade alcançável no mesmo tempo computacional (preciso citar).

2.3 Complexidade Computacional em Engines de Xadrez

A complexidade do xadrez é conhecimento antigo. Claude Shannon estimou o número total de jogos possíveis em aproximadamente 10^{120} (preciso citar), enquanto estimativas mais recentes por John Tromp sugerem $(4.822 \pm 0.028) \times 10^{44}$ posições legais distintas (preciso citar).

Esta explosão combinatória impõe limitações práticas essenciais que moldam a arquitetura de engines modernas. O horizonte de busca representa a primeira restrição: engines práticas exploram tipicamente 15-25 meias-jogadas, constituindo uma fração pequena do espaço de possibilidades total. Consequentemente, torna-se imperativa a implementação de funções de avaliação sofisticadas que estimem valores posicionais sem exploração exaustiva, substituindo cálculo bruto excessivo. Finalmente, a seletividade de busca emerge como necessidade técnica, manifestando-se através de técnicas como busca quiescente e extensões seletivas que direcionam recursos computacionais limitados para variantes críticas com maior potencial de influenciar a decisão final.

O fator de ramificação médio no xadrez aproxima-se de 35 movimentos por posição, embora varie consideravelmente conforme a fase do jogo e características posicionais particulares (preciso citar).

2.4 Protocolo UCI (Universal Chess Interface)

O UCI emergiu como padrão de facto para comunicação entre engines e interfaces gráficas, substituindo gradualmente protocolos anteriores como o WinBoard/CECP. Desenvolvido por

Rudolf Huber e Stefan Meyer-Kahlen em 2000, o UCI estabelece um conjunto de comandos textuais que permitem controle remoto das engines (preciso citar).

O protocolo define seis comandos essenciais que governam a interação completa entre interface e engine. O comando `uci` estabelece a sessão através de inicialização e identificação da engine, seguido por `isready` que verifica prontidão para análise. A configuração posicional realiza-se via `position`, que aceita tanto notação FEN quanto sequências de movimentos, enquanto `go` inicia análises com parâmetros específicos de tempo, profundidade ou número de nós. As operações completam-se através de `stop` para interrupção controlada da análise e `quit` para encerramento gracioso da engine.

A arquitetura sem estado do UCI facilita análises independentes de posições arbitrárias, embora limite as possibilidades de comunicação contextual complexa, restrição essa relevante para sistemas explicáveis que necessitam transmitir informações além das variantes e avaliações básicas.

2.5 Métricas de Avaliação de Engines

2.5.1 Sistema de Rating Elo

O sistema Elo, originalmente desenvolvido para xadrez humano por Arpad Elo, proporciona um método estatistico de robustez satisfatória para quantificar força relativa entre entidades competidoras (preciso citar).

A probabilidade esperada de vitória do jogador A contra B é dada por:

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$

onde R_A e R_B representam os ratings respectivos. Após cada partida, os ratings atualizam-se segundo:

$$R'_A = R_A + K(S_A - E_A)$$

onde S_A é o resultado real (1 para vitória, 0.5 para empate, 0 para derrota) e K é o fator de desenvolvimento.

Para engines, organizações como CCRL (Computer Chess Rating Lists) mantêm rankings baseados em milhares de partidas automatizadas, fornecendo estimativas precisas de força relativa (preciso citar).

2.5.2 Métricas de Acurácia

Além da força absoluta, a acurácia de uma engine pode ser medida através da concordância com engines de referência mais fortes. Define-se acurácia como:

$$\text{Acurácia} = \frac{1}{|P|} \sum_{p \in P} \mathbf{1}[a_{engine}(p) = a_{referência}(p)]$$

onde P representa um conjunto de posições teste e $\mathbf{1}[\cdot]$ é a função indicadora.

Métricas complementares incluem perda média em centipawns, que quantifica o custo avaliativo dos desvios da linha principal recomendada pela engine de referência.

3 Geração de Movimentos

A geração de movimentos é o alicerce arquitetural de motores de xadrez, devendo ser correta e rápida. Este módulo deve satisfazer as regras do jogo e apresentar boa performance, uma vez que é invocado milhares de vezes durante uma única análise. Nossa implementação utiliza geração de pseudo-movimentos com validação de legalidade posterior.

3.1 Visão Geral da Implementação

O sistema de geração de movimentos estrutura-se em algumas camadas de abstração, cada uma responsável por aspectos particulares da lógica geral. A abordagem utilizada consiste na separação entre *pseudo-legalidade* e *legalidade plena*. Movimentos pseudo-legais respeitam as regras básicas de movimentação das peças mas podem, por exemplo, deixar o próprio rei em xeque. A validação de legalidade subsequente elimina tais movimentos através de análise de ameaças, garantindo conformidade com as regras oficiais.

Esta separação proporciona vantagens computacionais já que algoritmos de busca frequentemente podem operar com pseudo-movimentos em contextos específicos, enquanto a validação completa reserva-se para situações onde a legalidade absoluta é mandatória.

3.2 Representação do Tabuleiro

Nossa implementação utiliza uma representação matricial 8×8 com elementos do tipo `Piece`, onde cada célula armazena informações sobre a peça presente (se houver) e sua cor. Esta abordagem, denominada *mailbox*, proporciona acesso direto por coordenadas com complexidade $O(1)$ para consultas individuais.

Listing 1: Estrutura básica do tabuleiro

```
class GameState {
private:
    Piece board[8][8];
    Player currentPlayer;
    bool canCastleKingside[2];
    bool canCastleQueenside[2];
    Position enPassantTarget;
    int halfmoveClock;
    int fullmoveNumber;

public:
    Piece getPiece(int row, int col) const;
    void setPiece(int row, int col, const Piece& piece);
    bool isSquareEmpty(int row, int col) const;
    // ... demais funcionalidades
};
```

A representação inclui metadados essenciais como direitos de roque, possibilidade de en passant, e contadores de meias-jogadas. Estes elementos integram-se naturalmente na estrutura FEN (Forsyth-Edwards Notation), facilitando interoperabilidade com ferramentas externas (preciso citar).

3.3 Geração de Pseudo-Movimentos Legais

O algoritmo central de geração percorre todas as peças do jogador ativo, invocando geradores para cada tipo da peça.

Listing 2: Estrutura do gerador principal

```
std::vector<Move> MoveGenerator::generatePseudoLegalMoves(
    const GameState& gameState) {

    std::vector<Move> moves;
    Player currentPlayer = gameState.getCurrentPlayer();

    for (int row = 0; row < 8; row++) {
        for (int col = 0; col < 8; col++) {
            Piece piece = gameState.getPiece(row, col);
            if (piece.getColor() == currentPlayer) {
                generatePieceMovesFromPosition(
                    gameState, Position(row, col), moves);
            }
        }
    }

    return moves;
}
```

3.4 Implementação para Peças Não-Deslizantes

Peças não-deslizantes (rei, cavalo, peão) possuem padrões de movimento discretos e bem definidos. A implementação utiliza vetores de deslocamento pré-computados, iterando sobre as possibilidades válidas para cada tipo.

3.4.1 Movimentação do Cavalo

O cavalo exemplifica a abordagem para peças não-deslizantes, utilizando oito vetores de deslocamento fixos:

$$\begin{aligned}&\text{GerarMovimentosCavalo}(\textit{posição}, \textit{estado}) = \\ &\qquad \text{Para cada } \delta \in \{(\pm 2, \pm 1), (\pm 1, \pm 2)\} : \\ &\qquad\quad \text{Se PosicaoValida}(\textit{posição} + \delta) \text{ e CasaAcessivel}(\textit{posição} + \delta) : \\ &\qquad\qquad \text{AdicionarMovimento}(\textit{posição} \rightarrow \textit{posição} + \delta)\end{aligned} \tag{1}$$

3.4.2 Movimentação do Peão

Os peões apresentam complexidade adicional devido à assimetria direcional, capturas diagonais, movimento duplo inicial e promoção. A implementação trata cada caso especificamente:

$$
\begin{aligned}
&\text{GerarMovimentosPeao}(posição, estado) = \\
&\qquad \text{direção} \leftarrow (\text{Brancas: -1, Pretas: } +1) \\
&\qquad \text{Se CasaVazia}(posição + direção): \\
&\qquad\quad \text{AdicionarMovimento}(posição \rightarrow posição + direção) \\
&\qquad\quad \text{Se PosicaoInicial}(posição) \text{ e CasaVazia}(posição + 2 \times direção): \\
&\qquad\qquad \text{AdicionarMovimento}(posição \rightarrow posição + 2 \times direção) \\
&\qquad \text{Para diagonais } \in \{(-1,-1),(-1,+1)\} \times direção: \\
&\qquad\quad \text{Se ExistePecaAdversaria}(posição + diagonal): \\
&\qquad\qquad \text{AdicionarCaptura}(posição \rightarrow posição + diagonal)
\end{aligned}
\tag{2}
$$

3.5 Implementação para Peças Deslizantes

Peças deslizantes (torre, bispo, dama) requerem algoritmos de trajetória que exploram direções específicas até encontrar obstáculos ou bordas do tabuleiro. A implementação emprega vetores direcionais e iteração controlada.

$$
\begin{aligned}
&\text{GerarMovimentosDeslizantes}(posição, direções) = \\
&\qquad \text{Para cada } \vec{d} \in direções: \\
&\qquad\quad posição_{atual} \leftarrow posição + \vec{d} \\
&\qquad\quad \text{Enquanto PosicaoValida}(posição_{atual}): \\
&\qquad\qquad \text{Se CasaVazia}(posição_{atual}) : \text{AdicionarMovimento}(posição \\
&\qquad\qquad \text{Senão Se PecaAdversaria}(posição_{atual}) : \text{AdicionarCaptura} \\
&\qquad\qquad \text{Senão: Parar} \\
&\qquad\qquad posição_{atual} \leftarrow posição_{atual} + \vec{d}
\end{aligned}
\tag{3}
$$

3.6 Movimentos Especiais

3.6.1 Roque

O roque representa o movimento especial mais complexo, exigindo verificação simultânea de várias condições: rei e torre não terem se movido, casas intermediárias livres, rei não estar em xeque e não passar por casa atacada.

Listing 3: Validação de roque

```
bool MoveGenerator::canCastle(const GameState& gameState, bool kingside)
    {
    Player player = gameState.getCurrentPlayer();

    // verificar direitos de roque
    if (kingside && !gameState.canCastleKingside(player)) return false;
    if (!kingside && !gameState.canCastleQueenside(player)) return false
        ;

    // verificar se rei est  em xeque
    if (isInCheck(gameState, player)) return false;

    // verificar casas intermedi rias livres e n o atacadas
    int row = (player == Player::WHITE) ? 7 : 0;
    int kingCol = 4;
    int rookCol = kingside ? 7 : 0;
    int direction = kingside ? 1 : -1;

    for (int col = kingCol + direction;
         col != rookCol;
         col += direction) {

        if (!gameState.isSquareEmpty(row, col)) return false;
        if (isSquareAttacked(gameState, Position(row, col),
                             getOpponent(player))) return false;
    }

    return true;
}
```

3.6.2 En Passant

A captura en passant requer detecção do movimento duplo do peão adversário na jogada anterior, implementada através do rastreamento da variável `enPassantTarget` no estado do jogo.

3.7 Validação de Legalidade

A conversão de pseudo-movimentos para movimentos legais elimina aqueles que deixariam o próprio rei em xeque. A implementação simula cada movimento e verifica a segurança do rei resultante:

Listing 4: Validação de legalidade

```
std::vector<Move> MoveGenerator::generateLegalMoves(
    const GameState& gameState) {

    std::vector<Move> pseudoLegalMoves =
```

```
        generatePseudoLegalMoves(gameState);
    std::vector<Move> legalMoves;

    for (const Move& move : pseudoLegalMoves) {
        GameState tempState = gameState;
        tempState.makeMove(move);

        if (!isInCheck(tempState, gameState.getCurrentPlayer())) {
            legalMoves.push_back(move);
        }
    }

    return legalMoves;
}
```

3.8 Otimizações de Performance

Foi preciso implementar várias otimizações para atingir resultado satisfatório. Atualmente movimentos são gerados apenas quando necessários, evitando computação desnecessária em contextos de poda alpha-beta. Além disso arrays de movimentos são reutilizados entre chamadas para minimizar alocações dinâmicas. Condições de validade são verificadas uma única vez por direção de movimento e em posições com muitas peças, o cálculo de casas atacadas é memorizado temporariamente.

3.9 Testes de Correção e Velocidade

A validação do sistema de geração de movimentos baseia-se nas seguintes duas metodologias: testes de correção funcional e benchmarks de performance.

Os testes funcionais utilizam posições conhecidas com contagens de movimentos documentadas (perft tests) para verificar se a implementação produz os números corretos para diferentes profundidades de análise (preciso citar).

Os benchmarks de velocidade medem movimentos gerados por segundo em posições representativas, permitindo identificar gargalos computacionais e validar as tentativas de otimizações implementadas. Nossa implementação atual alcança cerca 2.5 milhões de movimentos pseudo-legais por segundo em um i5-12500H com 2.50 GHz.

4 Sistema de Busca

O módulo de busca constitui o núcleo intelectual da engine, responsável por explorar metodicamente as consequências futuras de movimentos candidatos e identificar as continuações mais promissoras. Nossa implementação integra técnicas tradicionais de busca adversarial com um sistema original de coleta de métricas explicáveis, permitindo encontrar bons movimentos e também quantificar aspectos interpretativos relevantes para compreensão humana.

4.1 Arquitetura Geral do Módulo de Busca

A arquitetura de busca separa as responsabilidades algorítmicas: exploração da árvore de jogadas, avaliação posicional, coleta de estatísticas e gestão de recursos computacionais. Esta separação facilita manutenção, teste e extensão das funcionalidades.

O sistema opera através de chamadas iterativas de aprofundamento, onde cada nível de profundidade adicional refina os resultados anteriores. Esta abordagem, conhecida como *iterative deepening*, proporciona certas vantagens: fornece resultados utilizáveis a qualquer momento (comportamento anytime), melhora a ordenação de movimentos através de informações dos níveis anteriores, e permite controle preciso de recursos computacionais (preciso citar).

Listing 5: Estrutura principal do sistema de busca

```
class SearchEngine {
private:
    std::unique_ptr<Evaluator> evaluator;
    SearchMetrics currentMetrics;
    std::vector<Move> principalVariation;

public:
    SearchResult search(GameState& position, int maxDepth,
                        double timeLimit);
    double alphaBeta(GameState& position, int depth,
                     double alpha, double beta, bool maximizing,
                     SearchMetrics& metrics);

private:
    void collectExplainabilityMetrics(const GameState& position,
                                      const std::vector<Move>& pv,
                                      SearchMetrics& metrics);
};
```

4.2 Implementação do Algoritmo Alpha-Beta

Nossa implementação do algoritmo alpha-beta segue a formulação clássica com algumas mudanças particulares para coleta de métricas explicáveis. O algoritmo mantém os invariantes essenciais da poda enquanto simultaneamente captura informações sobre variância avaliativa, densidade tática e profundidade de convergência.

Listing 6: Implementação do Alpha-Beta com métricas

```
double SearchEngine::alphaBeta(GameState& position, int depth,
                               double alpha, double beta,
                               bool maximizing, SearchMetrics& metrics) {

    // incrementar contador de n s visitados
    metrics.nodesSearched++;

    // condi   o de parada: profundidade zero ou posi   o terminal
```

```
    if (depth == 0 || position.isGameOver()) {
        double eval = evaluator->evaluate(position);

        // coletar m tricas de vari ncia avaliativa
        if (metrics.evaluations.size() < MAX_EVAL_HISTORY) {
            metrics.evaluations.push_back(eval);
        }

        return eval;
    }

    std::vector<Move> moves = moveGenerator.generateLegalMoves(position)
        ;

    // ordenar movimentos para melhor performance da poda
    orderMoves(moves, position, depth);

    double bestValue = maximizing ? -INFINITY : INFINITY;
    Move bestMove;

    for (const Move& move : moves) {
        // fazer movimento
        position.makeMove(move);

        // busca recursiva
        double value = alphaBeta(position, depth - 1,
                                 alpha, beta, !maximizing, metrics);

        // desfazer movimento
        position.undoMove();

        // atualizar melhor valor e movimento
        if (maximizing) {
            if (value > bestValue) {
                bestValue = value;
                bestMove = move;

                // atualizar variante principal se estamos na raiz
                if (depth == initialDepth) {
                    updatePrincipalVariation(move, position);
                }
            }
            alpha = std::max(alpha, bestValue);
        } else {
            if (value < bestValue) {
                bestValue = value;
                bestMove = move;
            }
```

```
            beta = std::min(beta, bestValue);
        }

        // poda alpha-beta
        if (beta <= alpha) {
            metrics.betaCutoffs++;
            break;
        }
    }

    return bestValue;
}
```

4.3 Busca Quiescente

A busca quiescente estende a análise além da profundidade nominal em posições taticamente instáveis, explorando capturas e cheques até alcançar uma posição "quieta". Esta técnica mitiga o problema do horizonte, onde a engine poderia avaliar erroneamente posições no meio de sequências táticas complexas (preciso citar).

A implementação da busca quiescente segue o paradigma recursivo padrão, iniciando com avaliação estática da posição atual como baseline. Se esta avaliação já supera o limiar beta, estabelece-se corte imediato. Caso contrário, o algoritmo explora exclusivamente movimentos táticos (capturas, cheques, promoções) ordenados pela heurística MVV-LVA (Most Valuable Victim - Least Valuable Attacker), propagando recursivamente as avaliações até convergir em posições tranquilas onde não existem ameaças imediatas pendentes.

4.4 Aprofundamento Iterativo

O aprofundamento iterativo realiza sucessivas buscas com profundidade crescente, refinando nível a nível a análise e melhorando a ordenação de movimentos. Este processo termina quando o tempo disponível se esgota ou a profundidade máxima é atingida.

Listing 7: Controle do aprofundamento iterativo

```
SearchResult SearchEngine::iterativeDeepening(GameState& position,
                                               int maxDepth,
                                               double timeLimit) {

    SearchResult result;
    auto startTime = std::chrono::high_resolution_clock::now();

    for (int depth = 1; depth <= maxDepth; depth++) {
        SearchMetrics depthMetrics;

        // verificar limite de tempo
        auto currentTime = std::chrono::high_resolution_clock::now();
        auto elapsed = std::chrono::duration<double>(
```

```
            currentTime - startTime).count();

        if (elapsed >= timeLimit * 0.8) {
            break; // reservar tempo para finaliza  o
        }

        // busca na profundidade atual
        double score = alphaBeta(position, depth,
                                 -INFINITY, INFINITY,
                                 position.getCurrentPlayer() == Player::
                                    WHITE,
                                 depthMetrics);

        // atualizar resultado
        result.bestScore = score;
        result.principalVariation = currentPV;
        result.depth = depth;
        result.metrics = depthMetrics;

        // coletar m tricas de explicabilidade
        collectExplainabilityMetrics(position, currentPV, depthMetrics);
    }

    return result;
}
```

4.5 Ordenação de Movimentos

A eficácia da poda alpha-beta depende criticamente da qualidade da ordenação de movimentos. Nossa implementação utiliza algumas heurísticas tradicionais para tentar maximizar a probabilidade de explorar os melhores movimentos primeiro:

- **Hash move**: Movimento da variante principal em iterações anteriores
- **MVV-LVA**: Capturas ordenadas por valor da vítima e valor do atacante
- **Killer moves**: Movimentos que causaram cortes beta em nós irmãos
- **History heuristic**: Movimentos historicamente bem-sucedidos
- **Movimentos de xeque**: Priorizados pela pressão que exercem

Listing 8: Sistema de ordenação de movimentos

```
void SearchEngine::orderMoves(std::vector<Move>& moves,
                              const GameState& position,
                              int depth) {

    std::sort(moves.begin(), moves.end(),
```

```
            [&](const Move& a, const Move& b) {

        int scoreA = getMoveOrderingScore(a, position, depth);
        int scoreB = getMoveOrderingScore(b, position, depth);

        return scoreA > scoreB;
    });
}

int SearchEngine::getMoveOrderingScore(const Move& move,
                                       const GameState& position,
                                       int depth) {

    int score = 0;

    // hash move recebe prioridade m xima
    if (move == hashMove[depth]) {
        score += 1000000;
    }

    // capturas usando MVV-LVA
    if (move.isCapture()) {
        int victimValue = getPieceValue(position.getPiece(move.to));
        int attackerValue = getPieceValue(position.getPiece(move.from));
        score += victimValue * 100 - attackerValue;
    }

    // killer moves
    if (isKillerMove(move, depth)) {
        score += 50000;
    }

    // history heuristic
    score += historyTable[move.from.index()][move.to.index()];

    // movimentos de xeque
    if (givesCheck(move, position)) {
        score += 10000;
    }

    return score;
}
```

4.6 Coleta de Métricas Explicáveis

O aspecto mais distintivo de nossa implementação reside na coleta metódica de métricas que quantificam aspectos interpretativos da análise. Estas métricas foram especialmente desenvol-

vidas para capturar elementos que humanos consideram relevantes na avaliação de movimentos e posições.

4.6.1 Concretude Tática

A concretude mede a densidade de elementos táticos tangíveis (capturas, cheques, ameaças diretas) na variante principal. Esta métrica responde à questão: "quão tática é esta continuação?"

$$\text{Concretude} = \frac{\text{Número de elementos táticos na PV}}{\text{Profundidade da PV}} \times 100\%$$

4.6.2 Análise de Risco

O risco quantifica a variabilidade das avaliações ao longo da árvore de busca, indicando quão sensível é a posição a pequenas mudanças. Posições de alto risco caracterizam-se por grande variância avaliativa.

$$\text{Risco} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(e_i - \bar{e})^2}$$

onde e_i são as avaliações coletadas e $\bar{e}$ é a média.

4.6.3 Profundidade de Estabilidade

Esta métrica identifica o ponto na árvore de busca onde as avaliações convergem, indicando quando a análise atinge conclusões estáveis.

Listing 9: Coleta de métricas de explicabilidade

```
void SearchEngine::collectExplainabilityMetrics(
    const GameState& position,
    const std::vector<Move>& pv,
    SearchMetrics& metrics) {

    // calcular concretude t tica
    int tacticalElements = 0;
    GameState tempState = position;

    for (const Move& move : pv) {
        if (move.isCapture() || move.isCheck(tempState) ||
            move.isPromotion()) {
            tacticalElements++;
        }
        tempState.makeMove(move);
    }

    metrics.concreteness = pv.empty() ? 0.0 :
        (double)tacticalElements / pv.size() * 100.0;

```

```
    // calcular risco (vari ncia das avalia  es)
    if (metrics.evaluations.size() > 1) {
        double mean = std::accumulate(metrics.evaluations.begin(),
                                      metrics.evaluations.end(), 0.0) /
                      metrics.evaluations.size();

        double variance = 0.0;
        for (double eval : metrics.evaluations) {
            variance += (eval - mean) * (eval - mean);
        }
        variance /= (metrics.evaluations.size() - 1);

        metrics.risk = sqrt(variance);
    }

    // determinar profundidade de estabilidade
    metrics.stabilityDepth = findStabilityDepth(metrics.evaluations);

    // classificar estilo baseado nas m tricas
    classifyPlayingStyle(metrics);
}
```

4.7 Análise de Performance e Profundidade Alcançada

Nossa implementação atual atinge desempenho regular, explorando aproximadamente 9.000 nós por segundo em hardware de referência. A profundidade típica alcançada em análises de 5 segundos varia entre 5 e 7 meias-jogadas, dependendo da complexidade posicional e eficácia da poda.

Os testes de performance mostram que a coleta de métricas explicáveis impõe overhead computacional baixo (¡ 5%), validando a viabilidade da abordagem integrada. Este resultado deriva da otimização das rotinas de coleta e reutilização de cálculos já realizados pelo algoritmo de busca principal.

5 Sistema de Avaliação

A função de avaliação constitui o elemento que confere inteligência posicional à engine, traduzindo configurações complexas do tabuleiro em valores numéricos que orientam o processo de busca. Nossa implementação baseia-se em uma formulação heurística que incorpora aspectos materiais e posicionais, seguindo especificações para balancear os componentes.

5.1 Visão Geral da Função Heurística

A função de avaliação implementada segue uma arquitetura composicional que combina nove componentes distintos, cada um capturando aspectos particulares da força posicional. Esta abordagem modular permite ajustes independentes de cada elemento e facilita a compreensão dos fatores que contribuem para a avaliação final.

Podemos definir a formulação matemática desta como:

$$\begin{aligned} V_{posição} = \Delta_{seguranca\_rei} + \Delta_{mobilidade} + V_{material\_posicional} \\ + V_{defesas\_recebidas} + V_{defesas\_exercidas} \\ + V_{ameacas\_exercidas} - V_{ameacas\_recebidas} \end{aligned} \tag{4}$$

onde Δ representa diferenças entre os jogadores (próprio menos oponente) e V denota valores absolutos dos componentes respectivos.

Esta formulação equilibra considerações defensivas e ofensivas, reconhecendo que a força posicional emerge da interação complexa entre segurança, mobilidade, coordenação de peças e pressão exercida sobre o adversário (preciso citar).

5.2 Componentes de Avaliação Material

O componente material tradicional é expandido através da incorporação de valores posicionais que reconhecem que peças identicas têm eficácia variável dependendo de sua localização no tabuleiro. Esta abordagem utiliza tabelas posicionais (piece-square tables) derivadas da experiência enxadrística acumulada.

Listing 10: Avaliação material posicional

```
double Evaluator::calculatePiecePositionalValue(const Piece& piece) {
    double baseValue = getPieceBaseValue(piece.type);
    double positionalBonus = 0.0;

    int row = piece.position.row;
    int col = piece.position.col;

    // ajustar coordenadas para perspectiva das brancas
    if (piece.color == Player::BLACK) {
        row = 7 - row;
    }

    switch (piece.type) {
        case PieceType::PAWN:
            positionalBonus = pawnTable[row][col];
            break;
        case PieceType::KNIGHT:
            positionalBonus = knightTable[row][col];
            break;
        case PieceType::BISHOP:
            positionalBonus = bishopTable[row][col];
            break;
        case PieceType::ROOK:
            positionalBonus = rookTable[row][col];
            break;
        case PieceType::QUEEN:
```

```
            positionalBonus = queenTable[row][col];
            break;
        case PieceType::KING:
            positionalBonus = evaluateKingPosition(piece);
            break;
    }

    return baseValue + positionalBonus;
}
```

As tabelas posicionais codificam princípios estratégicos tradicionais do xadrez (preciso citar aqui). Peões centrais avançados recebem bonificações, cavalos preferem posições centralizadas, bispos valorizam diagonais longas, e torres beneficiam-se de colunas abertas e sétima fileira (preciso citar).

5.3 Avaliação de Mobilidade das Peças

A mobilidade quantifica a flexibilidade tática de cada jogador, medindo o número de movimentos legais disponíveis para suas peças. Esta métrica correlaciona-se fortemente com a força posicional, pois posições superiores tipicamente proporcionam mais opções estratégicas.

Listing 11: Cálculo de mobilidade

```
double Evaluator::evaluateMobility(const Player& player,
                                   const GameState& gameState) {

    double mobilityScore = 0.0;
    MoveGenerator moveGen;

    for (const auto& piece : player.remaining_pieces) {
        std::vector<Move> moves = moveGen.generatePieceMovesFromPosition
            (
            gameState, piece.position);

        // peso baseado no tipo da pe a
        double mobilityWeight = getMobilityWeight(piece.type);
        mobilityScore += moves.size() * mobilityWeight;

        // bonifica  o para pe as atacando centro
        for (const Move& move : moves) {
            if (isCenterSquare(move.to)) {
                mobilityScore += CENTER_CONTROL_BONUS;
            }
        }
    }

    return mobilityScore;
}
```

A implementação pondera diferentemente a mobilidade de cada tipo de peça, reconhecendo que a mobilidade da dama tem impacto superior à de um peão. Adicionalmente, movimentos que atacam casas centrais recebem bonificações, refletindo a importância estratégica do controle central.

5.4 Segurança do Rei

A avaliação de segurança do rei constitui um dos aspectos mais críticos da função heurística, pois posições com reis expostos costumeiramente levam a ataques devastadores. Nossa implementação considera fatores variados que contribuem para a segurança regia.

Listing 12: Avaliação de segurança do rei

```
double Evaluator::evaluateKingSafety(const Player& player,
                                     const Player& opponent) {

    const Piece* king = player.findKing();
    if (!king) return -KING_SAFETY_PENALTY;

    double safetyScore = 0.0;
    Position kingPos = king->position;

    // estrutura de pe es protetores
    safetyScore += evaluatePawnShield(kingPos, player);

    // proximidade de pe as inimigas
    for (const auto& enemyPiece : opponent.remaining_pieces) {
        double distance = calculateDistance(kingPos, enemyPiece.position
            );
        double threatValue = getPieceBaseValue(enemyPiece.type) / (
            distance + 1);
        safetyScore -= threatValue * PROXIMITY_THREAT_WEIGHT;
    }

    // casas atacadas ao redor do rei
    int attackedSquares = 0;
    for (int dr = -1; dr <= 1; dr++) {
        for (int dc = -1; dc <= 1; dc++) {
            Position adjacent(kingPos.row + dr, kingPos.col + dc);
            if (isValidPosition(adjacent) &&
                isSquareAttacked(adjacent, opponent)) {
                attackedSquares++;
            }
        }
    }

    safetyScore -= attackedSquares * ATTACKED_SQUARE_PENALTY;

    // roques realizados
```

```
    if (player.hasCastled()) {
        safetyScore += CASTLING_BONUS;
    }

    return safetyScore;
}
```

5.5 Estrutura de Peões

A estrutura de peões forma o esqueleto estratégico da posição, influenciando tanto a segurança quanto as possibilidades táticas futuras. Nossa avaliação considera peões passados, isolados, dobrados e cadeias de peões.

Listing 13: Avaliação da estrutura de peões

```
double Evaluator::evaluatePawnStructure(const Player& player,
                                        const Player& opponent) {

    double pawnScore = 0.0;
    std::vector<const Piece*> pawns = player.getPawns();

    for (const Piece* pawn : pawns) {
        // pe es passados
        if (isPassedPawn(pawn, opponent)) {
            int rank = (pawn->color == Player::WHITE) ?
                       pawn->position.row : 7 - pawn->position.row;
            pawnScore += PASSED_PAWN_BASE * (7 - rank);
        }

        // pe es isolados
        if (isIsolatedPawn(pawn, player)) {
            pawnScore -= ISOLATED_PAWN_PENALTY;
        }

        // pe es dobrados
        if (isDoubledPawn(pawn, player)) {
            pawnScore -= DOUBLED_PAWN_PENALTY;
        }

        // apoio de outros pe es
        if (isPawnSupported(pawn, player)) {
            pawnScore += PAWN_SUPPORT_BONUS;
        }
    }

    return pawnScore;
}
```

5.6 Fatores Táticos e Posicionais

Além dos componentes estruturais, a função incorpora elementos táticos que capturam ameaças imediatas e oportunidades de curto prazo. Este subsistema analisa padrões de ataque e defesa entre as peças.

5.6.1 Sistema de Ameaças

O componente de ameaças quantifica a pressão exercida sobre peças adversárias, considerando tanto o valor das peças atacadas quanto a probabilidade de materialização da captura.

Listing 14: Avaliação de ameaças

```
double Evaluator::valueOfPiecesThreatened(const Piece& attacker,
                                          const Player& opponent,
                                          bool outgoing,
                                          int excludeId) {

    double threatValue = 0.0;
    std::vector<Position> attackedSquares =
        getAttackedSquares(attacker);

    for (const Position& square : attackedSquares) {
        const Piece* target = opponent.getPieceAt(square);

        if (target && target->id != excludeId) {
            double pieceValue = getPieceBaseValue(target->type);

            // reduzir valor se pe a est  defendida
            if (isSquareDefended(square, opponent)) {
                pieceValue *= DEFENDED_PIECE_REDUCTION;
            }

            // bonificar amea as a pe as mais valiosas
            if (target->type == PieceType::QUEEN) {
                pieceValue *= QUEEN_THREAT_MULTIPLIER;
            }

            threatValue += pieceValue;
        }
    }

    return threatValue;
}
```

5.6.2 Sistema de Defesas

Paralelamente, o sistema avalia a coordenação defensiva, quantificando quão bem as peças se protegem mutuamente e controlam casas estratégicas.

5.7 Balanceamento dos Componentes

O balanceamento dos diferentes componentes requer calibração para evitar que aspectos específicos dominem excessivamente a avaliação. Nossa implementação utiliza os seguintes pesos (empiricamente ajustados através de testes contra posições conhecidas).

Componente	Peso Relativo	Justificativa
Segurança do Rei	100%	Fundamental para sobrevivência
Material + Posição	100%	Base da avaliação tradicional
Mobilidade	15%	Flexibilidade tática
Ameaças	25%	Pressão imediata
Defesas	20%	Coordenação de peças

Tabela 1: Pesos relativos dos componentes avaliativos

5.8 Validação da Função de Avaliação

A validação da função de avaliação baseia-se em testes sistemáticos com posições de referência conhecidas. Utilizamos suítes de testes posicionais onde a superioridade de um lado é estabelecida por análise de engine mais forte, verificando se nossa função produz avaliações consistentes.

Adicionalmente, comparações com engines estabelecidas em posições táticas e estratégicas validam a qualidade discriminativa da função. Os resultados demonstram correlação positiva significativa com avaliações de referência, confirmando a adequação da abordagem implementada.

6 Sistema de Explicabilidade

O sistema de explicabilidade, e sua relação para com o sistema de busca, constitui a contribuição mais interessante da engine desenvolvida, transformando análises algorítmicas opacas em narrativas interpretáveis que facilitam a compreensão humana das decisões tomadas. Esta arquitetura integra coleta de métricas estatísticas durante a busca com um sistema de geração de explicações contextuais

6.1 Arquitetura do Sistema de Explicação

O sistema de explicabilidade opera através de uma arquitetura que separa a coleta de dados interpretativos da geração das explicações finais. Esta separação permite flexibilidade na personalização das explicações para diferentes audiências e contextos de uso.

Listing 15: Arquitetura do sistema explicativo

```
class ExplanationEngine {
private:
    std::vector<ExplanationTemplate> templates;
    std::map<ExplanationCategory, std::vector<ExplanationTemplate*>>
        categoryMap;

```

```
public:
    std::vector<std::string> explainMove(const Move& move,
                                         const GameState& before,
                                         const GameState& after,
                                         const SearchMetrics& metrics);

    std::string explainSearchMetrics(const SearchMetrics& metrics);

    std::string recommendPlayingStyle(const SearchMetrics& metrics);

private:
    void analyzeMoveCharacteristics(ExplanationContext& context);
    std::vector<ExplanationTemplate*> selectRelevantTemplates(
        const ExplanationContext& context);
    bool evaluateConditions(const ExplanationTemplate& template,
                            const ExplanationContext& context);
    std::string fillTemplate(const std::string& template_text,
                             const ExplanationContext& context);
};
```

A arquitetura permite extensibilidade através da adição de novos templates sem modificação do código principal, e suporta personalização dinâmica baseada nas características específicas de cada posição analisada.

6.2 Métricas de Explicabilidade Implementadas

O sistema coleta três métricas principais que quantificam aspectos interpretativos que julgamos como importantes para o jogo humano: concretude, risco e estabilidade. Estas métricas foram especificamente desenhadas para capturar elementos que jogadores humanos consideram intuitivamente relevantes.

6.2.1 Concretude Tática

A métrica de concretude quantifica a densidade de elementos táticos tangíveis na variante principal, fornecendo uma medida objetiva de quão "tática"é uma sequência de movimentos.

$$C = \frac{\sum_{i=1}^{n} w_i \cdot I_i}{n} \times 100\%$$

onde I_i é um indicador binário de elemento tático no movimento i, w_i são pesos específicos para diferentes tipos de elementos táticos, e n é o comprimento da variante principal.

6.2.2 Análise de Risco

O risco captura a volatilidade avaliativa ao longo da árvore de busca, indicando quão sensível é a posição a variações nas continuações escolhidas.

$$R = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(e_i - \bar{e})^2} \times \frac{100}{\sigma_{ref}}$$

onde e_i são as avaliações coletadas em nós folha, $\bar{e}$ é a média, e σ_{ref} é um desvio padrão de referência para normalização.

Esta métrica distingue entre posições "quietas"(baixa variância) e posições "explosivas"(alta variância), informação crucial para compreender a natureza da posição analisada.

6.2.3 Profundidade de Estabilidade

A estabilidade identifica o ponto na busca onde as avaliações convergem, indicando quando a análise atinge conclusões robustas.

Listing 16: Cálculo da profundidade de estabilidade

```
int ExplanationEngine::calculateStabilityDepth(
    const std::vector<double>& evaluations) {

    if (evaluations.size() < 3) return -1;

    const double STABILITY_THRESHOLD = 0.2; // 20 centipawns
    int stability_depth = -1;

    for (size_t i = 2; i < evaluations.size(); i++) {
        bool is_stable = true;
        double baseline = evaluations[i];

        // verificar estabilidade nos pr ximos movimentos
        for (size_t j = i + 1; j < std::min(i + 3, evaluations.size());
           j++) {
            if (std::abs(evaluations[j] - baseline) >
               STABILITY_THRESHOLD) {
                is_stable = false;
                break;
            }
        }

        if (is_stable) {
            stability_depth = i;
            break;
        }
    }

    return stability_depth;
}
```

6.3 Sistema de Detecção de Temas Táticos e Posicionais

Dentro do sistema de explicabilidade, a contribuição mais significativa reside na identificação e implementação de 39 algoritmos especializados para detecção automática de temas táticos e posicionais. Este subsistema opera como uma camada interpretativa adicional que analisa cada movimento candidato em busca de padrões estratégicos reconhecíveis.

Os algoritmos de detecção dividem-se em duas categorias principais: temas positivos (1-29), que identificam características benéficas ou desejáveis nos movimentos analisados, e temas negativos (30-39), que detectam aspectos problemáticos ou contraproducentes. Esta classificação binária permite ao sistema oferecer tanto recomendações quanto advertências.

6.3.1 Arquitetura dos Algoritmos de Detecção

Cada algoritmo de detecção opera através da comparação sistemática entre o estado do tabuleiro antes e depois do movimento candidato. Esta abordagem diferencial permite identificar mudanças específicas que caracterizam temas estratégicos particulares, desde melhorias simples na mobilidade das peças até padrões complexos como a criação de pins ou ameaças táticas múltiplas.

A implementação utiliza métodos auxiliares especializados que calculam propriedades específicas das posições: mobilidade de peças individuais e coletiva, casas atacadas e defendidas, distâncias entre peças, identificação de casas fracas e análise de estruturas de peões. Estes componentes fundamentais são reutilizados através dos diferentes algoritmos, garantindo consistência metodológica e eficiência computacional.

6.3.2 Temas Positivos: Detecção de Características Benéficas

Os vinte e nove temas positivos abrangem aspectos cruciais do jogo posicional e tático. A ativação de torres exemplifica a categoria de melhoria de peças: o algoritmo compara a mobilidade da torre antes e depois do movimento, detectando aumentos significativos (superiores a duas casas) que indicam aproveitamento mais efetivo da peça. Similar princípio aplica-se à detecção de aumento geral de mobilidade, onde o sistema soma os movimentos disponíveis para todas as peças do jogador e identifica melhorias na flexibilidade tática global.

O controle de território constitui outro eixo fundamental. Os algoritmos detectam quando cavalos passam a controlar múltiplas casas centrais estratégicas, quando torres ocupam colunas abertas ou se posicionam em fileiras avançadas (sétima para brancas, segunda para pretas), e quando peões estabelecem controle diagonal sobre casas centrais.

A categoria de temas táticos inclui detecção de forks (ataques duplos), onde o algoritmo conta peças adversárias simultaneamente atacadas pela peça movida, e identificação de pins, através da análise de alinhamentos entre peças atacantes e alvos valiosos como rei ou dama. A implementação considera tanto pins absolutas (que impedem movimento por expor o rei) quanto relativas (que desencorajam movimento por expor peça valiosa).

Elementos de desenvolvimento e coordenação completam o espectro positivo. O sistema reconhece desenvolvimento correto na abertura verificando movimento de peças menores (cavalos e bispos) de suas casas iniciais nos primeiros dez lances, detecta roques através de verificação

direta de movimento especial, e identifica criação ou manutenção do par de bispos através de contagem específica desta configuração estrategicamente valiosa.

6.3.3 Temas Negativos: Identificação de Aspectos Problemáticos

Os dez temas negativos concentram-se em detectar violações de princípios estratégicos fundamentais que frequentemente caracterizam movimentos inferiores. O enfraquecimento das casas próximas ao rei exemplifica esta categoria: o algoritmo verifica se movimentos de peões adjacentes ao rei removem proteção de casas importantes, expondo-o a ataques futuros.

Problemas de desenvolvimento constituem foco significativo dos temas negativos. O sistema detecta desenvolvimento prematuro da dama nos primeiros oito lances, movimento repetitivo da mesma peça durante a abertura (violando princípios de desenvolvimento eficiente), e ataques prematuros com desenvolvimento insuficiente, onde peças não desenvolvidas (três ou mais) indicam base inadequada para iniciativas ofensivas.

A deterioração da estrutura de peões recebe atenção especial através de algoritmos que identificam criação de peões dobrados (resultantes de capturas), peões isolados (sem apoio lateral de peões adjacentes), e peões atrasados (posicionados atrás de peões vizinhos). Estas configurações geralmente representam fraquezas permanentes que compromitem perspectivas de longo prazo.

6.3.4 Sistema de Confiança e Validação

Cada detecção inclui score de confiança calibrado conforme a precisão algorítmica: detecções absolutas como xeque ou roque recebem confiança máxima (1.0), enquanto análises mais interpretativas como melhoria da pior peça ou flexibilidade posicional operam com confiança moderada. Este sistema permite priorização automática das explicações mais confiáveis.

O sistema registra automaticamente todas as detecções em arquivo de log detalhado, facilitando análise posterior e refinamento dos algoritmos. Esta funcionalidade viabiliza validação sistemática da precisão das detecções através de comparação com análise humana especializada.

6.4 Sistema de Templates de Explicação

O sistema utiliza uma abordagem baseada em templates para gerar explicações contextuais. Os templates são organizados em categorias temáticas e contêm condições de aplicabilidade e texto parametrizável.

6.4.1 Estrutura dos Templates

Listing 17: Estrutura de template explicativo

```
struct ExplanationTemplate {
    ExplanationCategory category;
    std::string template_text;
    std::vector<Condition> conditions;
    int priority;
    double confidence_threshold;

```

```
    // exemplos de condi  es:
    // - is_capture: movimento    captura
    // - threatens_queen: amea a dama advers ria
    // - improves_king_safety: melhora seguran a do rei
    // - centralizes_piece: centraliza pe a
    // - creates_passed_pawn: cria pe o passado
};

enum class ExplanationCategory {
    KING_SAFETY,
    PIECE_ACTIVITY,
    PAWN_STRUCTURE,
    TACTICAL_MOTIFS,
    STRATEGIC_THEMES,
    MATERIAL_CONSIDERATIONS,
    ENDGAME_TECHNIQUE
};
```

6.5 Classificação Automática de Estilo de Jogo

O sistema mantém 83 templates organizados por categoria para conseguir oferecer explicações movimento a movimento consideranto os aspectos posicionais e táticos envolvidos. Além disso analisa as métricas coletadas para classificar automaticamente o estilo da variante encontrada, oferecendo insights sobre a natureza da continuação recomendada.

Listing 18: Classificação de estilo

```
PlayingStyle ExplanationEngine::classifyPlayingStyle(
    const SearchMetrics& metrics) {

    // an lise multidimensional baseada nas m tricas
    if (metrics.concretenessScore > 60 && metrics.riskScore > 70) {
        return PlayingStyle::AGGRESSIVE;
    }

    if (metrics.concretenessScore < 30 && metrics.riskScore < 40) {
        return PlayingStyle::POSITIONAL;
    }

    if (metrics.concretenessScore > 50 && metrics.riskScore < 60) {
        return PlayingStyle::TACTICAL;
    }

    if (metrics.riskScore < 30 && metrics.stabilityDepth <= 3) {
        return PlayingStyle::DEFENSIVE;
    }

    return PlayingStyle::DYNAMIC;
```

```
}

std::string ExplanationEngine::generateStyleRecommendation(
    PlayingStyle style, const SearchMetrics& metrics) {

    switch (style) {
        case PlayingStyle::AGGRESSIVE:
            return "Estilo agressivo detectado. Esta linha busca
               iniciativa "
                   "atrav s de ataques diretos, priorizando press o
                      sobre "
                   "seguran a absoluta.";

        case PlayingStyle::POSITIONAL:
            return "Abordagem posicional identificada. O movimento visa
               "
                   "melhorias estruturais de longo prazo, fortalecendo "
                   "gradualmente a posi  o.";

        case PlayingStyle::TACTICAL:
            return "Sequ ncia t tica detectada. A continua  o
               explora "
                   "combina  es concretas para obter vantagem material
                      "
                   "ou posicional imediata.";

        case PlayingStyle::DEFENSIVE:
            return "Estrat gia defensiva adotada. O movimento consolida
                "
                   "a posi  o , neutralizando amea as e preparando "
                   "contraataque futuro.";

        case PlayingStyle::DYNAMIC:
            return "Jogo din mico caracterizado. A linha equilibra "
                   "elementos t ticos e posicionais, mantendo "
                   "flexibilidade estrat gica.";
    }

    return "";
}
```

6.6 Geração de Explicações Contextuais

O processo de geração de explicações integra informações sobre o movimento específico, mudanças posicionais e métricas de busca para produzir análises coerentes e informativas.

Listing 19: Geração contextual de explicações

```
std::vector<std::string> ExplanationEngine::explainMove(
```

```
    const Move& move, const GameState& before,
    const GameState& after, const SearchMetrics& metrics) {

    std::vector<std::string> explanations;

    // criar contexto de an lise
    ExplanationContext context;
    context.move = move;
    context.position_before = before;
    context.position_after = after;
    context.metrics = metrics;

    // analisar caracter sticas do movimento
    analyzeMoveCharacteristics(context);

    // selecionar templates relevantes
    auto relevant_templates = selectRelevantTemplates(context);

    // limitar n mero de explica  es por categoria
    std::map<ExplanationCategory, int> category_count;

    for (auto* template_ptr : relevant_templates) {
        if (category_count[template_ptr->category] >= 2) continue;

        if (evaluateConditions(*template_ptr, context)) {
            std::string explanation = fillTemplate(
                template_ptr->template_text, context);
            explanations.push_back(explanation);
            category_count[template_ptr->category]++;

            if (explanations.size() >= 5) break;
        }
    }

    return explanations;
}
```

O sistema garante diversidade temática limitando explicações por categoria e priorizando templates com maior relevância contextual. Isso evita redundância enquanto oferece perspectivas complementares sobre o movimento analisado.

6.7 Validação e Refinamento

A qualidade das explicações precisa ser validada através das seguintes metodologias: análise de coerência lógica, verificação de precisão factual e avaliação de utilidade interpretativa. E como a arquitetura desenhada facilita experimentos com diferentes abordagens explicativas, incorporamos mecanismos de feedback na interface padrão para que recebamos comunicação por parte dos usuários acerca dos resultados práticos, o que visa permitir refinamento dos templates

e condições de aplicabilidade.

7 Comunicação da Engine

A integração com o ecossistema de ferramentas de xadrez requer a implementação de protocolos de comunicação padronizados. Nossa engine adota o protocolo UCI como interface primária, enquanto incorpora extensões específicas para transmitir informações explicativas que perpassam as capacidades tradicionais do protocolo. Esta seção detalha a arquitetura de comunicação e as soluções desenvolvidas para conciliar compatibilidade universal com as funcionalidades desenvolvidas.

7.1 Implementação do Protocolo UCI

O Universal Chess Interface representa o padrão de fato para comunicação entre engines e interfaces gráficas, oferecendo um conjunto bem definido de comandos que garantem interoperabilidade ampla. Nossa implementação aderente ao UCI permite integração imediata com ferramentas populares disponíveis.

Listing 20: Estrutura principal da interface UCI

```
class UCIInterface {
private:
    std::unique_ptr<SearchEngine> engine;
    std::unique_ptr<ExplanationEngine> explainer;
    GameState currentPosition;
    bool debug_mode;

public:
    void run();
    void processCommand(const std::string& command);

private:
    void handleUCI();
    void handleIsReady();
    void handlePosition(const std::string& args);
    void handleGo(const std::string& args);
    void handleStop();
    void handleQuit();

    // extens es para explicabilidade
    void sendExplanations(const SearchResult& result);
    void sendMetricsInfo(const SearchMetrics& metrics);
};
```

7.2 Comandos UCI Suportados

Nossa implementação oferece suporte completo aos comandos do protocolo UCI, garantindo compatibilidade com interfaces gráficas existentes.

7.2.1 Inicialização e Identificação

Listing 21: Comandos de inicialização

```
void UCIInterface::handleUCI() {
    std::cout << "id name Chess Engine Explic vel v1.0" << std::endl;
    std::cout << "id author Felipe" << std::endl;

    // op   es configur veis
    std::cout << "option name Hash type spin default 128 min 1 max 4096"
              << std::endl;
    std::cout << "option name Threads type spin default 1 min 1 max 8"
              << std::endl;
    std::cout << "option name ExplanationLevel type combo default Medium
        "
              << "var Basic var Medium var Advanced" << std::endl;

    std::cout << "uciok" << std::endl;
}

void UCIInterface::handleIsReady() {
    // verificar se engine est  pronta para an lise
    if (engine && explainer) {
        std::cout << "readyok" << std::endl;
    }
}
```

7.2.2 Configuração de Posição

O comando `position` permite configurar a posição de análise tanto a partir da posição inicial quanto de notação FEN específica, com suporte para sequências de movimentos subsequentes.

Listing 22: Processamento de posições

```
void UCIInterface::handlePosition(const std::string& args) {
    std::istringstream iss(args);
    std::string token;
    iss >> token;

    if (token == "startpos") {
        currentPosition.resetToStartingPosition();

        // processar movimentos subsequentes se presentes
        if (iss >> token && token == "moves") {
            while (iss >> token) {
                Move move = parseUCIMove(token);
                if (move.isValid()) {
                    currentPosition.makeMove(move);
                } else {
```

```
                    std::cerr << "Movimento inv lido: " << token << std
                        ::endl;
                    return;
                }
            }
        }
    } else if (token == "fen") {
        // reconstruir string FEN
        std::string fenString;
        for (int i = 0; i < 6 && (iss >> token); i++) {
            if (i > 0) fenString += " ";
            fenString += token;
        }

        if (!currentPosition.setFromFEN(fenString)) {
            std::cerr << "FEN inv lida: " << fenString << std::endl;
            return;
        }

        // processar movimentos se presentes
        if (iss >> token && token == "moves") {
            while (iss >> token) {
                Move move = parseUCIMove(token);
                if (move.isValid()) {
                    currentPosition.makeMove(move);
                }
            }
        }
    }
}
```

7.2.3 Controle de Análise

O comando go inicia a análise com parâmetros específicos de tempo, profundidade ou número de nós. Nossa implementação suporta os principais modos de controle temporal.

Listing 23: Controle de busca

```
void UCIInterface::handleGo(const std::string& args) {
    std::istringstream iss(args);
    std::string token;

    int depth = -1;
    int time_ms = -1;
    int nodes = -1;
    bool infinite = false;

    while (iss >> token) {
        if (token == "depth") {
```

```
            iss >> depth;
        } else if (token == "movetime") {
            iss >> time_ms;
        } else if (token == "nodes") {
            iss >> nodes;
        } else if (token == "infinite") {
            infinite = true;
        }
        // adicionar suporte para wtime, btime, winc, binc se
           necess rio
    }

    // determinar par metros de busca
    SearchParameters params;
    if (depth > 0) {
        params.maxDepth = depth;
    } else {
        params.maxDepth = 6; // profundidade padr o
    }

    if (time_ms > 0) {
        params.timeLimit = time_ms / 1000.0;
    } else {
        params.timeLimit = 5.0; // 5 segundos padr o
    }

    // executar busca
    SearchResult result = engine->search(currentPosition, params);

    // enviar resultado
    std::cout << "bestmove " << result.bestMove.toUCIString() << std::
       endl;

    // enviar informa  es explicativas se dispon veis
    sendExplanations(result);
}
```

7.3 Integração com Interfaces Gráficas

Como dito, a compatibilidade UCI permite integração rápida com interfaces gráficas populares. Durante o desenvolvimento, testamos nossa engine com algumas GUIs para verificar conformidade e estabilidade.

7.3.1 Testes de Compatibilidade

- **Arena Chess GUI**: Integração com suporte a torneios automáticos
- **Cute Chess**: Testes de força

- **PyChess**: Verificação de parsing

7.4 Gerenciamento de Tempo

O gerenciamento eficaz de tempo constitui aspecto crítico para uso prático da engine. Nossa implementação monitora o tempo de análise e interrompe a busca apropriadamente para respeitar limites impostos.

Listing 24: Controle temporal

```
class TimeManager {
private:
    std::chrono::high_resolution_clock::time_point startTime;
    double allocatedTime;
    bool shouldStop;

public:
    void startClock(double timeLimit) {
        startTime = std::chrono::high_resolution_clock::now();
        allocatedTime = timeLimit;
        shouldStop = false;
    }

    bool timeExpired() const {
        auto now = std::chrono::high_resolution_clock::now();
        auto elapsed = std::chrono::duration<double>(now - startTime).
           count();
        return elapsed >= allocatedTime * 0.95; // margem de seguran a
    }

    void requestStop() {
        shouldStop = true;
    }

    bool stopRequested() const {
        return shouldStop || timeExpired();
    }
};
```

7.5 Extensões para Informações Explicativas

Embora o protocolo UCI padrão não preveja transmissão de explicações estruturadas, utilizamos o comando `info` para comunicar informações adicionais de forma compatível com interfaces existentes.

Listing 25: Transmissão de explicações via UCI

```
void UCIInterface::sendExplanations(const SearchResult& result) {
    // gerar explica  es do movimento
    std::vector<std::string> explanations = explainer->explainMove(
```

```
        result.bestMove, currentPosition,
        getPositionAfterMove(result.bestMove), result.metrics);

    // enviar como coment rios UCI
    for (size_t i = 0; i < explanations.size(); i++) {
        std::cout << "info string explanation" << (i+1) << " "
                  << explanations[i] << std::endl;
    }

    // enviar m tricas de explicabilidade
    std::cout << "info string concreteness "
              << std::fixed << std::setprecision(1)
              << result.metrics.concretenessScore << std::endl;

    std::cout << "info string risk "
              << std::fixed << std::setprecision(1)
              << result.metrics.riskScore << std::endl;

    std::cout << "info string style "
              << getStyleString(result.metrics.playingStyle) << std::
                 endl;
}

void UCIInterface::sendMetricsInfo(const SearchMetrics& metrics) {
    std::cout << "info nodes " << metrics.nodesSearched << std::endl;
    std::cout << "info nps " << calculateNPS(metrics) << std::endl;
    std::cout << "info time " << metrics.timeElapsed << std::endl;
    std::cout << "info depth " << metrics.depth << std::endl;

    // informa  es espec ficas de explicabilidade
    if (debug_mode) {
        std::cout << "info string beta_cutoffs "
                  << metrics.betaCutoffs << std::endl;
        std::cout << "info string quiescence_nodes "
                  << metrics.quiescenceNodes << std::endl;
    }
}
```

7.6 Robustez e Tratamento de Erros

A implementação incorpora tratamento robusto de erros para garantir estabilidade operacional em diferentes contextos de uso. Comandos malformados são detectados e reportados apropriadamente sem comprometer a funcionalidade da engine.

Listing 26: Tratamento de erros

```
void UCIInterface::processCommand(const std::string& command) {
    try {
        std::istringstream iss(command);
```

```
        std::string cmd;
        iss >> cmd;

        if (cmd == "uci") {
            handleUCI();
        } else if (cmd == "isready") {
            handleIsReady();
        } else if (cmd == "position") {
            std::string args;
            std::getline(iss, args);
            handlePosition(args);
        } else if (cmd == "go") {
            std::string args;
            std::getline(iss, args);
            handleGo(args);
        } else if (cmd == "stop") {
            handleStop();
        } else if (cmd == "quit") {
            handleQuit();
        } else if (!cmd.empty()) {
            std::cerr << "Comando n o reconhecido: " << cmd << std::
                endl;
        }
    } catch (const std::exception& e) {
        std::cerr << "Erro processando comando '" << command
                  << "': " << e.what() << std::endl;
    }
}
```

7.7 Performance da Comunicação

A eficiência da interface de comunicação é crucial para responsividade em análises interativas. Nossa implementação utiliza buffering apropriado e minimiza latência de resposta, garantindo experiência fluida para o usuário.

Testes de latência demonstram tempos de resposta consistentemente abaixo de 1ms para comandos de configuração, enquanto comandos de análise iniciam processamento imediatamente após recepção. A arquitetura threaded permite interrupção responsiva de análises em progresso.

8 Avaliação e Testes

A validação é, certamente, elemento necessário para estabelecer a confiabilidade e utilidade prática da engine desenvolvida. Nossa metodologia de avaliação incluiu correção funcional, performance computacional, força de jogo e qualidade das explicações geradas.

8.1 Metodologia de Avaliação

A estratégia de testes foi estruturada em três categorias principais:

1. **Testes de Correção**: Verificação da implementação correta das regras de xadrez
2. **Testes de Performance**: Medição de velocidade e eficiência algorítmica
3. **Testes de Força**: Estimativa de rating ELO através de comparações com engines estabelecidas

8.2 Testes Funcionais da Engine

Os testes funcionais servem para verificar a correção da implementação através de cenários controlados que exercitam todas as funcionalidades principais.

8.2.1 Testes Básicos

Listing 27: Estrutura dos testes funcionais

```
class EngineTests {
public:
    bool testInitialization();
    bool testPieceMovement();
    bool testMoveGeneration();
    bool testEvaluation();
    bool testSearch();
    bool testExplanation();

    void runAllTests() {
        std::vector<std::pair<std::string, std::function<bool()>>> tests
            = {
            {"Inicializa  o", [this]() { return testInitialization();
                }},
            {"Movimento de pe as", [this]() { return testPieceMovement
                (); }},
            {"Gera  o de movimentos", [this]() { return
                testMoveGeneration(); }},
            {"Avalia  o heur stica", [this]() { return testEvaluation
                (); }},
            {"Busca alpha-beta", [this]() { return testSearch(); }},
            {"Sistema de explica  o", [this]() { return
                testExplanation(); }}
        };

        for (auto& [name, test] : tests) {
            std::cout << "Teste: " << name << " - ";
            std::cout << (test() ? "PASSOU" : "FALHOU") << std::endl;
        }
    }
};
```

8.3 Análise de Performance Computacional

A análise de performance quantifica a eficiência algorítmica através de benchmarks específicos que medem velocidade de busca, utilização de memória e escalabilidade com a profundidade.

8.3.1 Métricas de Velocidade

Listing 28: Benchmark de performance

```cpp
struct PerformanceMetrics {
    uint64_t nodes_per_second;
    double memory_usage_mb;
    int max_depth_achieved;
    double avg_branching_factor;
    double time_per_move;
};

PerformanceMetrics benchmarkEngine(int time_limit_seconds) {
    PerformanceMetrics metrics;
    GameState position;
    position.resetToStartingPosition();

    auto start = std::chrono::high_resolution_clock::now();

    SearchResult result = engine.search(position, time_limit_seconds);

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration<double>(end - start).count();

    metrics.nodes_per_second = result.metrics.nodesSearched / duration;
    metrics.max_depth_achieved = result.depth;
    metrics.time_per_move = duration;
    metrics.memory_usage_mb = measureMemoryUsage();

    return metrics;
}
```

8.4 Estimativa de Força ELO

A estimativa de rating ELO baseia-se em comparações sistemáticas com o Stockfish, utilizando análise de concordância de movimentos em posições diversificadas.

8.4.1 Metodologia de Comparação

Listing 29: Script de avaliação ELO (fragmento)

```python
def evaluate_engine_strength(our_engine, stockfish_path, test_positions):
    """
```

```
    Compara nossa engine com Stockfish em posi  es de teste
    """
    agreements = 0
    total_positions = 0

    for fen in test_positions:
        # Analisar com nossa engine
        our_move = analyze_position(our_engine, fen, depth=5)

        # Analisar com Stockfish
        sf_move = analyze_stockfish(stockfish_path, fen, depth=5)

        if our_move == sf_move:
            agreements += 1

        total_positions += 1

    agreement_rate = agreements / total_positions
    estimated_elo = estimate_elo_from_agreement(agreement_rate)

    return {
        'agreement_rate': agreement_rate,
        'estimated_elo': estimated_elo,
        'total_positions': total_positions
    }
```

8.4.2 Resultados da Avaliação de Força

A análise comparativa com o Stockfish em 10 conjuntos de 10 posições produziu os seguintes resultados:

Métrica	**Valor**
Taxa de concordância com Stockfish	60%
Posições analisadas	100
ELO estimado	2120
Classificação	Engine intermediária
Força relativa	Jogador expert humano

Tabela 2: Estimativa de força da engine

Esta classificação posiciona nossa engine em nível competitivo para uso educacional e análise casual, embora ainda distante de engines comerciais de elite.

8.5 Comparação com Stockfish

A comparação com o Stockfish indicou que a função de avaliação implementada captura adequadamente elementos posicionais, mas beneficiar-se-ia de refinamento tático e expansão do conhecimento de aberturas.

8.5.1 Critérios de Qualidade

1. **Precisão factual**: As explicações refletem corretamente as características da posição?
2. **Clareza linguística**: O texto é compreensível?
3. **Relevância contextual**: As explicações abordam os aspectos mais importantes da posição?
4. **Diversidade temática**: Diferentes linhas estratégicas são cobertos?
5. **Consistência**: Explanações similares são geradas para posições similares?

8.5.2 Limitações da Avaliação Explicativa

A validação do sistema explicativo claramente requer metodologia qualitativa para avaliar clareza, precisão e utilidade das explicações geradas.

No entanto devemos destacar que, devido às limitações de tempo no desenvolvimento do projeto, não foi possível implementar a avaliação qualitativa por meio de entrevistas e formulários com usuários humanos conforme inicialmente planejado. Esta lacuna representa uma oportunidade importante para trabalhos futuros, tendo em vista que a validação com usuários reais é, em última análise, essencial para verificar a eficácia pedagógica do sistema explicativo desenvolvido.

A ausência desta validação não compromete os aspectos técnicos do sistema, mas limita nossa compreensão sobre a utilidade das explicações geradas para diferentes perfis de usuários. Recomendamos que estudos futuros incorporem metodologias de avaliação com participantes humanos de diversos níveis de conhecimento enxadrístico.

8.6 Limitações Identificadas

A avaliação sistemática revela algumas limitações que representam oportunidades para desenvolvimento futuro:

- **Profundidade de busca limitada**: Restrições computacionais limitam análise a 5-7 meias-jogadas
- **Conhecimento tático**: Performance inferior em posições com combinações complexas
- **Biblioteca de aberturas**: Ausência de conhecimento específico de aberturas
- **Finais avançados**: Limitações em finais técnicos específicos
- **Calibração explicativa**: Explicações ocasionalmente enfatizam aspectos menos relevantes

Estas limitações são consistentes com o escopo educacional do projeto e não comprometem a utilidade prática do sistema para seu público-alvo.

9 Resultados e Discussão

A análise integrada dos resultados obtidos revela um sistema funcional que atinge seus objetivos primários de combinar competência algorítmica com explicabilidade interpretativa. Esta seção sintetiza os achados principais, analisa suas implicações e contextualiza as contribuições dentro do panorama mais amplo de engines de xadrez e sistemas explicáveis.

9.1 Performance da Engine Implementada

Nossa implementação demonstra performance técnica satisfatória considerando seus objetivos educacionais. A velocidade de busca de aproximadamente 9.000 nós por segundo, que, embora modesta comparada a engines comerciais otimizadas, oferece responsividade adequada para análise interativa e uso pedagógico.

A profundidade de busca alcançada (5-7 meias-jogadas) situa-se dentro de parâmetros esperados para hardware convencional, permitindo análises suficientemente profundas para capturar a maioria dos elementos táticos de curto prazo. O overhead imposto pelo sistema de explicabilidade permanece baixo, validando a viabilidade da integração entre performance e interpretabilidade.

A estimativa de força ELO realizada posiciona a engine em nível intermediário, competitiva contra jogadores expert e, por vezes, grande-mestres humanos. Esta classificação é apropriada para uma implementação inicial.

9.2 Acerca do Sistema de Explicabilidade

O sistema de explicabilidade representa a contribuição mais significativa deste trabalho, preenchendo lacuna identificada entre competência algorítmica e compreensão humana pois as métricas de explicabilidade desenvolvidas oferecem quantificação objetiva de aspectos intuitivamente relevantes para jogadores humanos. Dentro disso, poder automaticamente classificar o estilo de jogo recomendado.

O sistema de templates categorizado oferece adaptação contextual enquanto a diversidade temática observada durante a avaliação posicional (segurança do rei, atividade das peças, estrutura de peões, etc.) garante cobertura de elementos estratégicos tradicionais do xadrez.

9.3 Análise Comparativa com Outras Engines

A comparação com o Stockfish revelou padrões interessantes que iluminam as características de nossa abordagem. A concordância superior em posições posicionais versus táticas sugere que nossa função de avaliação captura adequadamente nuances estratégicas de longo prazo, embora beneficiaria-se de refinamento em cálculo tático preciso.

A performance em finais básicos indica boa convergência, porém as limitações em aberturas específicas refletem ausência de bibliotecas especializadas, já que resolvemos manter foco na implementação de princípios gerais.

9.4 Limitações Identificadas

A profundidade de busca mais lenta limita um pouco a capacidade de resolver posições táticas complexas que requerem cálculo extremamente extenso.

A ausência de certas otimizações mais avançadas (tabelas de transposição extensas, paralelização, algoritmos de redução seletiva) dá-se em função dos prazos reduzidos para um projeto desse tamanho. Porém atualizaremos o código da engine para implementar tais melhorias.

O sistema explicativo, embora funcional, ocasionalmente enfatiza aspectos secundários em posições complexas. Esta limitação reflete a dificuldade inerente de automatizar julgamentos sobre relevância contextual, tarefa que requer compreensão semântica sofisticada do domínio enxadrístico.

9.5 Casos de Uso e Aplicações Práticas

Revisão de partidas e aplicações educacionais constituem o caso de uso primário. Observando tal fato, e para demonstrar a funcionalidade completa do sistema, desenvolvemos uma interface web interativa, que permite aos usuários inserir arquivos PGN e então experimentar as capacidades explicativas da engine.

9.6 Contribuições

Este trabalho contribui para a comunidade enxadrística em algumas ângulos. Primeiramente, demonstra viabilidade técnica de integrar coleta de métricas explicáveis em algoritmos de busca adversarial sem compromisso significativo de performance.

Secundariamente, propõe métricas quantificáveis (concretude, risco, estabilidade) que capturam aspectos interpretativos relevantes, oferecendo vocabulário formal para discussão de explicabilidade em domínios de jogos estratégicos.

Por fim a arquitetura de templates categorizados representa humilde contribuição metodológica para geração automática de explicações contextuais.

10 Conclusão

Com o presente trabalho demonstramos a viabilidade inicial de se desenvolver engines de xadrez que combinem competência algorítmica com explicabilidade interpretativa, ajudando a avançar contra a lacuna observada entre a capacidade computacional das engines modernas e a necessidade humana de compreensão causal.

10.1 Síntese dos Resultados Obtidos

A engine desenvolvida atinge todos os objetivos estabelecidos inicialmente. Do ponto de vista técnico, implementamos sistema funcional completo que inclui geração correta de movimentos, função de avaliação heurística sofisticada, algoritmo de busca alpha-beta otimizado e interface UCI compatível com ferramentas padrão da área.

O sistema de explicabilidade, constituindo a contribuição mais distintiva do trabalho, demonstra capacidade de reduzir a opacidade das análises algorítmicas atuais, aproximando-as de

narrativas interpretáveis. As métricas desenvolvidas oferecem quantificação objetiva de aspectos qualitativos tradicionalmente dependentes de intuição humana.

A validação empírica confirma performance inicial adequada para aplicações de revisão de partidas e em contextos educacionais. O overhead computacional imposto pelo sistema explicativo permaneceu com baixo impacto, validando a integração dos referidos módulos.

10.2 Trabalhos Futuros

As limitações identificadas e o potencial demonstrado pelo sistema apontam para direções importantes de desenvolvimento futuro como a implementação de tabelas de transposição avançadas, paralelização e algoritmos de redução seletiva para aumentar profundidade de busca. A incorporação de bibliotecas de aberturas e finais para melhorar força absoluta em fases específicas do jogo e o desenvolvimento de mais componentes para detecção e avaliação de motivos táticos complexos

Referências

[1] Marc Barthelemy. Fragility of chess positions: Measure, universality, and tipping points. *arXiv preprint arXiv:2410.02333*, 2024.

[2] CCRL. Computer chess rating lists (about). CCRL Website, 2023.

[3] Chessify. Behind the numbers: Understanding chess engine evaluations. *Chessify Blog*, 2023.

[4] Steven J. Edwards. Forsyth–edwards notation (fen) specification. PGN Specification Annex, 1994.

[5] Valery Filippov. The best chess engines in 2024: Stockfish vs. leela chess zero. *Grandmaster Chess Coach Blog*, 2024.

[6] Frederic Friedel. Arpad emre elo – 100th anniversary. *ChessBase News*.

[7] Investopedia. Zero-sum game definition (chess example). Investopedia, updated June 29, 2025.

[8] Stefan Meyer-Kahlen and Rudolf Huber. Universal chess interface (uci) specification. ShredderChess Documentation, 2000.

[9] Martin Müller. Alpha–beta search and enhancements (tutorial slides). In *Proc. AAAI-14 Workshop on Computer Games*, 2014.

[10] Toolify AI News. Decoding human vs ai chess moves: Blurring boundaries in the game.

[11] S. J. Shapiro and T. P. Hales. Ai engine ratings: Humans vs. machines. *Journal of Computer Play*, 15(1):10–12, 2025.

[12] Alan M. Turing and David G. Champernowne. Turochamp (chess algorithm). *Unpublished algorithm* (conceived 1948), referenced in Copeland's *The Essential Turing* (Oxford Univ. Press, 2004).

[13] ChessProgramming Wiki. Piece-square tables (example bonuses). *Chessprogramming.org*, 2019.

[14] ChessProgramming Wiki. Quiescence search. *Chessprogramming.org*, 2021.

[15] ChessProgramming Wiki. Perft (performance test). *Chessprogramming.org*, 2022.

[16] Wikipedia. Human–computer chess matches – chess 4.5 (1976–1978). Last accessed June 30, 2025.

[17] Wikipedia. Human–computer chess matches – cray blitz (1981). Last accessed June 30, 2025.

[18] Wikipedia. Perfect information (game theory). Last accessed June 30, 2025.

[19] Wikipedia. Deep blue vs. garry kasparov (1997) – impact. *Wikipedia*, 2025.