Projeto de Xadrez: motor baseado em bitboards

Paulo Henrique Silva Dias[1] and Henrique Campanha Garcia[1]

[1]Instituto de Ciência e Tecnologia – Universidade Federal de São Paulo (UNIFESP), São José dos Campos – SP – Brasil

Resumo

Este documento descreve o desenvolvimento de um motor de xadrez implementado como trabalho final da disciplina de Realidade Virtual e Aumentada. Apresentamos a arquitetura interna baseada em *bitboards*, os algoritmos de geração de movimentos, a estratégia de busca (com otimizações clássicas) e uma interface mínima para interação e validação. Resultados de testes de correção (*perft*) e medidas de desempenho foram coletados; direções de trabalho futuro também são discutidas.

1 Introdução

Este relatório descreve um motor de xadrez desenvolvido como trabalho final da disciplina Realidade Virtual e Aumentada. O objetivo foi conceber e implementar um motor funcional capaz de: (1) gerar movimentos legais completos; (2) buscar jogadas com algoritmos clássicos aprimorados; (3) oferecer uma avaliação material e posicional configurável; e (4) prover utilitários para validação e uma interface mínima de interação com o usuário.

A identidade visual e a estrutura do documento partem do template fornecido e foram adaptadas para descrever de forma concisa o desenvolvimento do motor.

2 Objetivos

Os objetivos principais do projeto foram:

- Implementar geração correta de movimentos, incluindo roque, en passant e promoções.
- Construir um mecanismo de busca (minimax com poda alpha-beta) com otimizações práticas.
- Definir uma função de avaliação combinando peso material e heurísticas posicionais.
- Validar a geração de movimentos com uma rotina *perft* e fornecer utilitários (FEN/PGN).
- Disponibilizar uma interface de uso mínimo (linha de comando) e suporte inicial ao protocolo UCI (Universal Chess Interface).

3 Metodologia

3.1 Representação e arquitetura

Escolhemos uma representação baseada em *bitboards* (64 bits) para obter operações bitwise eficientes. A representação interna inclui:

- Bitboards separados por tipo de peça e por cor (ex.: peões brancos, cavalos negros, etc.), tipicamente utilizando `System.UInt64` nas implementações C#.
- Máscara de ocupação global e máscaras auxiliares para acelerar testes de ataque e movimento.
- Estado extra para direitos de roque, possibilidade de en passant e contador de meio-movimento para regras de empate.

Essa escolha favorece operações por palavra (64 bits) e permite gerar movimentos e calcular ataques com alta performance.

3.2 Geração de movimentos

A geração de movimentos foi organizada por tipo de peça:

- Cavalos e reis: tabelas de deslocamento pré-computadas.
- Peões: regras específicas para avanço, captura, en passant e promoções.
- Torres, bispos e dama: ataques do tipo *sliding* implementados com técnicas de lookup (com possibilidade futura de substituir por *magic bitboards* completos).

Validações adicionais garantem que são retornados apenas movimentos legais (ou seja, que não deixam o rei em xeque).

A função de avaliação combina componentes materiais (pesos por peça) e termos posicionais (tabelas peça-casa, peões passados, estrutura de peões e segurança do rei). Todos os parâmetros são ajustáveis por arquivo de configuração.

3.3 Interface e utilitários

Foram desenvolvidos utilitários mínimos para uso e validação:

- Conversores FEN/PGN para leitura e exportação de posições e partidas.
- Ferramenta *perft* para validação da geração de movimentos.
- Interface em linha de comando com entrada por notação algébrica e opção de salvar partidas em PGN.
- Esqueleto de integração com UCI para compatibilizar com clientes externos (implementação parcial — comandos básicos como `uci`, `isready`, `position` e `go` foram considerados).

4 Implementação

4.1 Linguagem e dependências

O motor foi implementado em C# (Unity), priorizando integração com o ciclo de desenvolvimento do Unity e facilidade de depuração via Editor. O projeto foi desenvolvido visando compatibilidade com Unity 2020+ e utiliza as APIs padrão do Unity (`UnityEngine`) para execução no Player e utilitários do Editor. Dependências externas (quando aplicáveis) estão documentadas em `Packages/manifest.json` e limitam-se a bibliotecas leves utilizadas em protótipos ou ferramentas auxiliares.

4.2 Organização do código

Principais módulos e arquivos:

- `Board.cs` – representação do tabuleiro, estruturas de dados (bitboards/arrays) e leitura/escrita em formatos padrão (por ex. FEN).
- `MoveGenerator.cs` – geração de movimentos legais, filtros de validade e ordenamento inicial.
- `Search.cs` – controle da busca, *iterative deepening*, alpha-beta com *move ordering* e uso de tabela de transposição.
- `Evaluation.cs` – função de avaliação (material e termos posicionais), modular para fácil ajuste de parâmetros.
- `PerftTool.cs` – ferramenta de validação executável no Editor para contagem de nós (*perft*) e verificação de corretude.
- `UciAdapter.cs` (opcional) – adaptador para expor a engine por meio do protocolo UCI, implementado como componente separado quando necessário.
- `Editor/` – scripts utilitários e janelas customizadas do Unity Editor (ex.: executores de testes, visualizadores de bitboard).

A estrutura segue a convenção de Assembly Definitions (quando aplicável) para acelerar o build e permitir testes isolados.

4.3 Testes de correção

Foram implementadas ferramentas de validação no Editor (PerftTool) e testes automatizados usando o Unity Test Runner quando apropriado. Executaram-se testes *perft* em posições de referência e compararam-se os resultados para profundidades 1–5. Divergências identificadas (comportamentos incorretos relacionados a roque e en passant) foram corrigidas e revalidadas. Os logs de execução e os testes unitários correspondentes estão organizados em `Tests/` para facilitar reexecução e auditoria.

5 Desafios no projeto

Durante o desenvolvimento enfrentamos diversos desafios práticos e de engenharia. Destacam-se:

5.1 Uso da ferramenta Unity

Embora o motor seja essencialmente uma biblioteca de lógica de jogo, integrá-lo ao ciclo do Unity trouxe desafios:

- Separação clara entre lógica pura (motor de xadrez) e código dependente do Unity (`MonoBehaviour`, APIs de renderização/Editor), para permitir execução segura em threads de busca e testes fora do Player.
- Gerenciamento do ciclo de vida de objetos e serialização no Editor, evitando dependências que mascarassem bugs lógicos quando executado fora do ambiente do Unity.
- Ferramentas de depuração no Editor exigiram investimentos extras (visualizadores de bitboard, janelas de execução de perft) para inspecionar o estado interno sem afetar a performance.

5.2 Mensagens de xeque-mate (detecção e exibição)

A correta detecção de fim de jogo e a comunicação ao usuário também trouxeram dificuldades:

- Garantir que a condição de xeque-mate/fim de jogo seja detectada de forma consistente tanto no gerador de movimentos quanto no loop de jogo (UCI/CLI/Unity).
- Exibir mensagens claras e localizadas para o usuário (por exemplo, distinção entre empate por insuficiência de material, afogamento, repetição, e xeque-mate).
- Integrar essas mensagens com a UI do Unity de modo que callbacks e threads de busca não acionem APIs do Unity fora do thread principal.

5.3 Desenvolvimento da IA

O desenvolvimento da camada de IA levantou desafios clássicos de motores de xadrez:

- Balancear precisão da avaliação e custo computacional: tabelas posicionais, avaliação de estruturas de peões e heurísticas adicionais aumentam a qualidade mas também o custo de avaliação por nó.
- Ordenação de lances e heurísticas (killer moves, history, quiescence) demandaram afinação para obter cortes eficientes e boa performance em profundidade.
- Manter determinismo e reproduzibilidade nos testes (logs, hashes de posição) para facilitar depuração de falhas de busca.

6 Experimentos com a IA adversária

6.1 Descrição do módulo `Assets/Scripts/ChessAI.cs`

O arquivo `Assets/Scripts/ChessAI.cs` implementa a camada de decisão do motor. Ele integra a representação do tabuleiro, o gerador de movimentos e o módulo de busca para selecionar o lance a ser jogado. O componente pode operar como `MonoBehaviour` (expondo

propriedades no Inspector) ou como uma classe de backend invocada pelo controlador do jogo.

Responsabilidades principais:

- Manter o estado da instância da IA (configurações, limites de tempo/profundidade, tabela de transposição).
- Expor uma API para solicitar o melhor lance (por exemplo, `GetBestMove` ou `RequestMove`).
- Orquestrar a geração de movimentos, poda alpha-beta, busca iterativa (*iterative deepening*) e avaliação de posições.
- Registrar métricas de execução (nós visitados, nós por segundo, tempo por iteração) e fornecer hooks para depuração (*perft*, logs).
- Integrar-se de forma segura com o loop do Unity; caso a busca rode em background, não acessar APIs do Unity fora da thread principal.

Estruturas de dados e campos relevantes:

- Referência ao tabuleiro atual (`Board currentBoard`).
- Tabela de transposição (hash table) para cache de posições e cortes.
- Parâmetros de controle: profundidade máxima, limite de tempo, habilitar busca quiescente (*quiescence*).
- Estatísticas de busca: contador de nós, tempo de início/fim, melhores lances por profundidade.

Algoritmos e métodos principais:

- `GetBestMove(Board board, SearchOptions options)` – executa *iterative deepening* até atingir limite de tempo/profundidade e retorna o melhor lance.
- `Search(Position pos, int depth, int alpha, int beta)` – implementação recursiva com poda alpha-beta e ordenação de lances (captures primeiro, *killer moves*, *history heuristic*).
- `QuiescenceSearch(Position pos, int alpha, int beta)` – busca quiescente que avalia capturas e cheques até que a posição esteja quieta.
- `Evaluate(Position pos)` – avaliação estática (material, PST, mobilidade, segurança do rei).
- `MakeMove / UnmakeMove` – aplicação e reversão eficiente de lances (stack de estados).
- Ferramentas auxiliares: `Perft(int depth)`, ordenadores de lances e serialização do estado da IA.

Integração com o Unity:

- Quando é `MonoBehaviour`, expõe propriedades serializáveis no Inspector (profundidade máxima, uso de threads, habilitar logs).

- Se a busca é executada em background, utiliza mecanismos de sincronização (Tasks / threads) e comunica resultados ao thread principal via callbacks seguros.
- Eventos/callbacks notificam a interface quando um lance foi calculado.

6.2 Validação qualitativa

Partidas contra configurações simples e autojogos evidenciaram coerência nas jogadas geradas pela avaliação atual, embora exista espaço para aprimorar conhecimento de abertura e heurísticas posicionais.

7 Melhorias e trabalho futuro

Com base nos desafios identificados e na experiência de implementação, sugerimos as seguintes melhorias priorizadas:

7.1 Deixar a IA mais desafiadora

Possíveis caminhos para aumentar a força do motor:

- Aprimorar a função de avaliação: adicionar termos de segurança do rei mais sofisticados, avaliação de iniciativa, avaliação dinâmica de espaço e redes neurais leves (p.ex. small NN para avaliação posicional).
- Expandir e refinar heurísticas de ordenação de lances (MVV/LVA aprimorado, extensions seletivas, refutações nulas controladas).
- Implementar *iterative deepening* com pesquisa aspiracional, pesquisa distribuída (se aplicável) e uso mais agressivo de tabelas de transposição e *pawn hash*.
- Integrar ou alimentar um pequeno livro de aberturas para evitar erros triviais em fases iniciais.

7.2 Arrumar mensagens de xeque-mate e robustez do fim de jogo

Melhorias práticas para a experiência do usuário e confiabilidade:

- Centralizar a detecção de condições de fim de jogo (cheque-mate, empate por insuficiência de material, repetição, regra dos 50 movimentos) em um único módulo de regras para evitar inconsistências.
- Implementar e testar mensagens localizadas claras na UI do Unity e na interface de linha de comando/UCI, garantindo que a exibição seja acionada apenas pelo thread principal.
- Adicionar testes automatizados que verifiquem cenários de fim de jogo e a correspondência entre estado interno e mensagens exibidas.

7.3 Outras melhorias sugeridas

- Melhor cobertura de testes unitários e de integração (cobertura de perft estendida, divergências regressivas).
- Ferramentas de profiling no Unity para medir custo das avaliações e identificar gargalos.
- Polimento na integração UCI para compatibilidade com clientes de terceiros (tempo, ponderado, ponder, ponderado por movimento).

8 Conclusão

O projeto alcançou seus objetivos centrais: geração correta de movimentos (validada por *perft*), implementação de uma busca com alpha-beta e tabela de transposição, e uma função de avaliação básica e configurável. A adoção de *bitboards* mostrou-se eficaz em termos de performance para um motor didático.

Pontos fortes:

- Arquitetura modular que facilita extensões (melhor avaliação, livro de aberturas, UCI completo).
- Correção validada via *perft*.

Repositório e licença

O código-fonte e materiais relacionados a este projeto encontram-se em:

`https://github.com/Paul-Dias/Projeto-de-Realidade-Virtual-e-Aumentada`

O projeto é disponibilizado sob licença MIT (consulte o arquivo LICENSE no repositório).