Xadrez

Universidade Estadual de Maringá - Ciência da Computação
(Departamento de Informática)

Professor: Igarashi

Alunos

Marcos Henrique Souza Bosco - RA117873

Maria Fernanda Almeida Oliveira - RA118597

Maringá

2023

Introdução

O presente relatório visa explorar e discutir a aplicação de algoritmos de busca competitiva no contexto do jogo de xadrez. O xadrez, um dos jogos de tabuleiro mais icônicos e desafiadores, tem sido objeto de estudo para desenvolver estratégias inteligentes e algoritmos de tomada de decisão eficazes. A busca competitiva, representada principalmente pelos algoritmos Minimax e Alpha-Beta Pruning, desempenha um papel fundamental na capacidade das máquinas de tomar decisões inteligentes em um ambiente de jogo complexo e estratégico, como o xadrez.

Os algoritmos de busca competitiva visam maximizar a eficiência da busca em árvores de jogadas possíveis, permitindo que uma IA (Inteligência Artificial) tome decisões informadas em um ambiente competitivo. O algoritmo Minimax forma a base desses algoritmos, considerando todas as possíveis jogadas até um certo nível de profundidade. No entanto, a complexidade exponencial das árvores de busca torna essa abordagem impraticável sem otimizações. É aqui que o Alpha-Beta entra em cena, reduzindo drasticamente o número de estados a serem explorados, mantendo a mesma garantia de encontrar a melhor jogada.

A aplicação desses algoritmos no xadrez envolve uma avaliação cuidadosa das posições do tabuleiro, que é realizada através de funções de avaliação. Essas funções atribuem pontuações a diferentes posições no tabuleiro com base nas peças, suas posições e características estratégicas, como mobilidade e segurança do rei. Além disso, considerações especiais são aplicadas para situações como xeque-mate e empate, aprimorando ainda mais a precisão das decisões da IA.

Neste relatório, examinaremos a implementação prática dos algoritmos Minimax e Alpha-Beta aplicados ao jogo de xadrez. Abordaremos também a elaboração de funções de avaliação que levam em consideração diversos fatores estratégicos do jogo. Além disso, discutiremos a integração de engines de xadrez, como o Stockfish, para validar nossas implementações e aprimorar ainda mais a qualidade das decisões da IA.

Através dessa exploração, pretendemos ilustrar a importância desses algoritmos de busca competitiva no desenvolvimento de sistemas de IA capazes de competir em níveis avançados de xadrez.

Tecnologias utilizadas

No decorrer do projeto, exploramos o potencial da biblioteca Chess.js como base para a implementação da lógica do jogo de xadrez. Através dessa biblioteca, fomos

capazes de criar e manipular tabuleiros de xadrez, gerar movimentos legais, avaliar posições e muito mais. Ela foi uma facilitadora da representação e interação com o tabuleiro.

Uma outra faceta importante da nossa implementação envolveu a criação de um servidor que permitisse renderizar a lógica do Stockfish, a IA de xadrez desenvolvida pelo Google. Para essa tarefa, empregamos a biblioteca *stockfish.js,* que nos possibilita carregar a engine do Stockfish e utilizar suas capacidades de cálculo de movimentos e tomada de decisões.

Além disso, optamos por construir a interface do projeto utilizando a biblioteca React. A escolha pelo React se deu pois através dela, conseguimos criar uma interface amigável para os jogadores interagirem com o jogo. A renderização do tabuleiro, em particular, se beneficiou da capacidade do React em atualizar apenas as partes necessárias da interface, melhorando o desempenho geral da aplicação.

Ao longo do desenvolvimento, a integração dessas bibliotecas - Chess.js para a lógica do jogo, stockfish.js para aprimorar a tomada de decisões e React para a interface - permitiu que nosso projeto se tornasse uma aplicação completa e funcional.

Etapas de implementação

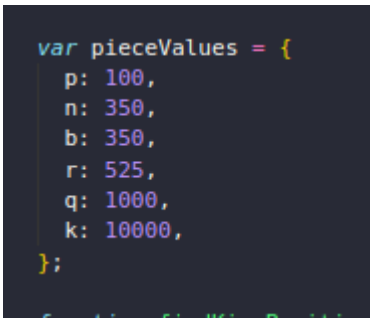
O código foi organizado em funções que avaliam a posição do tabuleiro, calculam os melhores movimentos possíveis e interagem com um motor de xadrez, como o Stockfish, para tomar decisões informadas. Abaixo serão descritas as funções principais para o desenvolvimento do sistema:

Função **`evaluatePlays`**

```
function findKingPosition(board, color) {
  for (var i = 0; i < 8; i++) {
    for (var j = 0; j < 8; j++) {
      const piece = board[i][j];
      if (piece && piece.type === 'k' && piece.color === color) {
        return [i, j];
      }
    }
  }
  return null;
}

export const evaluatePlays = (game) => {
  var board = game.board();
  var kingPosition = findKingPosition(board, game.turn());

  var value = 0;
  for (var i = 0; i < 8; i++) {
    for (var j = 0; j < 8; j++) {
      const piece = board[i][j];
      if (piece) {
        value += pieceValues[piece.type] * (piece.color === 'b' ? -1 : 1);

        if (piece.type !== 'k' && kingPosition) {
          const distanceToKing = Math.max(
            Math.abs(i - kingPosition[0]),
            Math.abs(j - kingPosition[1])
          );

          if (distanceToKing <= 1) {
            value += 10;
          }
        }
      }
    }
  }

  return value;
};
```

Essa função foi implementada para avaliar a posição do tabuleiro. Como o objetivo é maximizar a avaliação do jogador atual e minimizar a avaliação do oponente, multiplicamos o valor da peça por um número negativo caso este seja o oponente (‘b’). Também levamos em consideração a proximidade da peça em relação ao rei, pois caso esteja próximo, a peça deve protegê-lo. Desta forma, adicionamos uma pontuação a avaliação dessa peça.

Além disso, adicionamos uma certa pontuação a cada peça de acordo com a estratégia do xadrez. Essa pontuação é feita na variável ***pieceValues:***



```
var pieceValues = {
  p: 100,
  n: 350,
  b: 350,
  r: 525,
  q: 1000,
  k: 10000,
};
```

Onde:

- P = peão
- N = cavalo
- B = bispo
- R = Torre
- Q = rainha
- K = rei

Função **minimax**

```
export const minimax = (depth, alpha, beta, maximizingPlayer, game) => {
  if (
    depth === 0 ||
    game.isDraw() ||
    game.isCheckmate() ||
    game.isStalemate() ||
    game.isThreefoldRepetition()
  ) {
    return evaluatePlays(game);
  }

  const possibleMoves = game.moves();
  const randomPossibleMoves = randomMoves(possibleMoves);

  let bestValue = maximizingPlayer ? -Infinity : Infinity;

  for (const possibleMove of randomPossibleMoves) {
    game.move(possibleMove); //simula jogada
    const value = minimax(depth - 1, alpha, beta, !maximizingPlayer, game); //
    game.undo(); //desfaz jogada

    if (maximizingPlayer) {
      bestValue = Math.max(bestValue, value);
      alpha = Math.max(alpha, value);
    } else {
      bestValue = Math.min(bestValue, value);
      beta = Math.min(beta, value);
    }

    if (beta <= alpha) {
      break;
    }
  }

  return bestValue;
};
```

1. Começamos verificando se atingimos um limite de profundidade (número de jogadas olhadas para frente) ou se o jogo terminou de alguma forma (empate, xeque-mate, empate por repetição). Se isso acontecer, retornaremos uma avaliação da posição atual.
2. Obtemos a lista de movimentos que podem ser feitos no tabuleiro.
3. Embaralhamos aleatoriamente a ordem dos movimentos. Isso é feito para introduzir alguma aleatoriedade na escolha, para que o algoritmo não siga sempre o mesmo caminho.
4. Inicializamos uma variável chamada `bestValue` com um valor muito grande (*Infinity*) ou muito pequeno (-Infinity), dependendo se estamos maximizando ou minimizando o valor.
5. Percorremos cada movimento da lista de movimentos:
   a. Fazemos o movimento no tabuleiro.
   b. Chamamos recursivamente o próprio algoritmo (`minimax`) com um nível de

profundidade reduzido. Isso simula a próxima jogada, alternando entre os jogadores.
c. Desfazemos o movimento para considerar outras opções.
d. Atualizamos *bestValue* com o valor encontrado (maior valor se estivermos maximizando, menor se estivermos minimizando).
e. Também aplicamos a técnica de poda Alpha-Beta: se encontrarmos uma jogada que não é vantajosa para nós, paramos de olhar para essa linha de jogadas, pois sabemos que não a escolhemos.

6. Retornamos o valor de *bestValue* que representa o melhor valor de avaliação encontrado ao longo das simulações.

Desta forma, a função acima implementa o algoritmo Minimax com poda Alpha-Beta, considerando a recursão para explorar possíveis sequências de movimentos e avaliando a posição do tabuleiro a cada nível de profundidade.

Função **calculateBestMove**

```
export const calculateBestMove = (maxDepth, game, maximizingPlayer) => {
  var newGameMoves = game.moves();
  const randomnewGameMoves = randomMoves(newGameMoves);

  var bestMove = null;
  var bestMoveValue = -Infinity;

  for (var i = 0; i < randomnewGameMoves.length; i++) {
    var move = randomnewGameMoves[i];
    game.move(move);
    var moveValue = minimax(
      maxDepth - 1,
      -Infinity,
      Infinity,
      !maximizingPlayer,
      game
    );

    if (moveValue > bestMoveValue) {
      bestMoveValue = moveValue;
      bestMove = move;
    }
    game.undo();
  }
  return bestMove;
};
```

Determina o melhor movimento possível para o jogador atual, utilizando o algoritmo Minimax com poda Alpha-Beta. Ela gera todos os movimentos legais, avalia as posições resultantes e escolhe o movimento que maximiza a avaliação.

1. **`newGameMoves`** é uma lista de todos os movimentos possíveis no estado atual do jogo.
2. **`randomnewGameMoves`** é uma lista de movimentos aleatórios selecionados a partir de **`newGameMoves`**. Isso é feito para introduzir uma certa aleatoriedade no processo de escolha de movimentos, o que pode ser útil em alguns cenários.
3. Um loop percorre todos os movimentos aleatórios em **`randomnewGameMoves`**.
4. Para cada movimento, o jogo é atualizado usando **`game.move(move)`** para simular o movimento.
5. O algoritmo minimax é chamado com profundidade reduzida (**`maxDepth - 1`**), limites de alpha e beta (**`-Infinity`** e **`Infinity`**), o jogador alternado (**`!maximizingPlayer`**), e o estado atual do jogo.
6. O valor retornado pelo minimax (**`moveValue)`** é comparado com o valor do melhor movimento encontrado até agora (**`bestMoveValue`**).
7. Se o **`moveValue`** for maior que **`bestMoveValue`**, isso significa que encontramos um movimento melhor. Portanto, **`bestMove`** e **`bestMoveValue`** são atualizados para o movimento atual e seu valor.

Funções **makeMoveWithStockfish** e **makeBestMoveStockfish**

```
const makeMoveWithStockfish = async (fen) => {
  try {
    const response = await axios.post('http://localhost:8080/', {
      fen,
    });
    return response.data;
  } catch (error) {
    console.error('Erro ao fazer o movimento com o Stockfish:', error);
    return null;
  }
};

export const makeBestMoveStockfish = async (game, setGame) => {
  const fen = game.fen();
  const response = await makeMoveWithStockfish(fen);
  if (
    response &&
    typeof response === 'string' &&
    response.includes('bestmove')
  ) {
    const bestMove = response.split(' ')[1];
    const gameTurn = game.turn() === 'w' ? 'BRANCO' : 'PRETO';

    console.log('Melhor movimento STOCKFISH:===>', bestMove, gameTurn);
    game.move(bestMove);

    setGame(new Chess(game.fen()));
  }
};
```

Interagem com a engine de xadrez (Stockfish) através de uma requisição HTTP, enviando a posição atual em formato FEN e recebendo a melhor jogada calculada.

Resultados dos testes

```
export const exaustiveTests = async (game, setGame) => {
  const depths = [1, 2, 3];
  const numGamesPerDepth = 1;
  const depthWins = { BRANCO: 0, PRETO: 0 };

  for (const depth of depths) {
    console.log(`Starting tests for depth ${depth}`);

    const gamesResults = await runDepthTests(
      depth,
      numGamesPerDepth,
      game,
      setGame
    );
    updateWinsCount(depthWins, gamesResults);
  }

  console.log(depthWins);
};

const runDepthTests = async (depth, numGames, game, setGame) => {
  const results = [];

  for (let i = 0; i < numGames; i++) {
    const gameCopy = new Chess(game.fen());
    const gameResults = await runGameTests(gameCopy, depth, setGame);
    results.push(gameResults);
  }

  return results;
};

const runGameTests = async (gameCopy, depth, setGame) => {
  let gameTurn = gameCopy.turn();
  const gameResults = { winner: null };

  while (!gameCopy.isGameOver()) {
    if (gameTurn === 'w') {
      makeBestMove(gameCopy, gameTurn, setGame);
    } else {
      await makeBestMoveStockfish(gameCopy, setGame);
    }

    gameTurn = gameCopy.turn();
  }

  gameResults.winner = gameTurn === 'w' ? 'PRETO' : 'BRANCO';
  return gameResults;
};
```

1. A função **`exaustiveTests`** é a função principal que coordena a realização de testes exaustivos para diferentes profundidades de busca. Ela recebe o estado do jogo atual e a função ***`setGame`*** para atualizar o estado do jogo.
2. A constante `depths` é um array que contém as profundidades de busca que você deseja testar.
3. A constante ***`numGamesPerDepth`*** representa o número de jogos a serem jogados para cada profundidade de busca.

4. A constante ***depthWins*** é um objeto que rastreia o número de vitórias para cada cor (BRANCO ou PRETO) para cada profundidade de busca.
5. O loop `for` em ***exaustiveTests*** itera sobre as profundidades definidas. Para cada profundidade, ele chama a função ***runDepthTests***.
6. A função ***runDepthTests*** realiza os testes para uma determinada profundidade de busca. Ela cria cópias do estado atual do jogo e executa ***numGames*** jogos para essa profundidade. Para cada jogo, chama-se a função ***runGameTests***.
7. A função ***runGameTests*** simula um jogo até o final. Ela alterna entre os turnos das peças BRANCO e PRETO. Para cada turno, chama `makeBestMove` para a cor branca ou ***makeBestMoveStockfish*** para a cor preta, dependendo do turno.
8. A função ***updateWinsCount*** é usada para atualizar o contador de vitórias para cada cor com base nos resultados de cada jogo.
9. Após a execução de todos os testes para cada profundidade, a função ***exaustiveTests*** imprime os resultados de vitórias para cada profundidade e cor.

No geral, esse código realiza uma série de testes exaustivos para diferentes profundidades de busca em um jogo de xadrez, alternando entre a IA que joga como peças brancas e a IA que joga como peças pretas. Ele conta quantos jogos cada lado vence em diferentes profundidades de busca e imprime os resultados. Isso pode ajudá-lo a avaliar o desempenho da sua IA em diferentes níveis de busca.

Como resultados de testes para o IA X IA, tivemos os seguinte:

1 - Percorrendo cada nível de profundidade 1 só vez

2 - Percorrendo cada nível de profundidade 3 vezes

Como resultados de testes para o IA X HUMANO, tivemos os seguinte:

1 - Jogadas com 2 de profundidade => IA ganhou

2 - Jogadas com 3 de profundidade => Humano ganhou 1 vez e IA 3 vezes

Análise dos resultados obtidos

É notável que o jogador Stockfish tenha obtido vitória em todas as competições realizadas. Isso reforça a notória capacidade do Stockfish, que é um dos motores de xadrez mais poderosos e bem otimizados disponíveis. A vitória consistente do Stockfish destaca a extrema eficiência do algoritmo Alpha-Beta em explorar rapidamente as possibilidades de jogadas e selecionar a melhor jogada possível. Essa consistência na vitória é uma indicação da profundidade da busca e da qualidade das avaliações realizadas pelo Stockfish.

Ao analisar a função ***evaluatePlays***, observou-se que atualmente considera apenas os valores atribuídos às peças no tabuleiro e segurança do rei. Embora esse seja um aspecto importante, a estratégia no xadrez vai além dos valores estáticos das peças. A avaliação deve considerar fatores como controle de casas importantes, coordenação entre peças, mobilidade e estrutura de peões. Ignorar esses fatores pode levar a decisões subótimas, onde a IA pode não perceber posições táticas ou estratégicas cruciais.

Por outro lado, a IA desenvolvida demonstrou pontos fortes na avaliação da segurança do rei. A avaliação da segurança do rei é de extrema importância no xadrez, já que um rei vulnerável pode levar a situações de xeque-mate rapidamente. A IA que considera a segurança do rei como parte de sua função de avaliação é capaz de tomar decisões mais cautelosas e proteger sua posição.

Outro ponto foi que ela apresentou uma abordagem robusta de busca competitiva, utilizando o algoritmo Minimax e a poda Alpha-Beta. Essa estratégia permitiu que a IA explorasse diferentes linhas de jogo, analisando cenários possíveis e antecipando as jogadas do adversário.

Para as jogadas de IA X HUMANO, os resultados sugerem que a IA é capaz de analisar mais profundamente as possíveis sequências de jogadas, o que lhe confere uma vantagem em termos de tomada de decisões baseadas em informações futuras. A vitória do jogador humano em uma partida de profundidade 3 pode se dar pelos motivos de avaliação citados acima.

Códigos base

Código base server do stockfish => https://github.com/hyugit/stockfish-server

Código base do xadrez => https://codepen.io/deinman/pen/OpdVaL