AlgoChess

Team Members:
Mohamed Hussein (Github: MHussein311)
Zayne Bournand   (Github: zaybournand)
Dajana Seitllari   (Github: seitllarid)

Github: https://github.com/zaybournand/AlgoChess.git
Demo Video: https://youtu.be/VMxQ0dJ5iD0

Problem:

We are solving the problem of how to select the best move in a chess game using search algorithms. We're comparing a fixed-depth tree search (Minimax) with a more adaptive method (Selective Deepening) that looks deeper when something important might happen, like a capture or checkmate. We are also comparing a naive minimax algorithm with a minimax algorithm that utilizes alpha-beta pruning.

Motivation:

Fixed-depth search will stop searching at a fixed length, and therefore may stop right before important moves like captures or checkmate, leading to bad decisions. We want to see if going deeper in critical positions makes move selection more accurate without slowing things down too much.

Features Implemented:

- Fixed-depth Minimax search for chess move selection.
- Selective Deepening Minimax search (adaptive depth based on unstable positions like captures/checks).
- Alpha-Beta Pruning to significantly optimize both fixed-depth and selective searches.
- Naive Minimax search (without pruning) for performance benchmarking.
- Simulation framework to compare move quality and runtime across many board states.
- Stockfish engine integration for judging move quality.
- Analysis of algorithm performance in general and specifically in unstable scenarios.
- Visualization of results using bar charts to show comparisons.
- Parallelization of simulations using multiprocessing.

Description of Data:

- A publicly available dataset of 100,000+ real chess board states in FEN (Forsyth-Edwards Natations) format, from Kaggle's "Chess Evaluations" dataset found here: https://www.kaggle.com/datasets/ronakbadhe/chess-evaluations/data?select=chessData.csv and each data point includes a FEN string and the current player to move.

Tools/Languages/APIs/Libraries Used:

- **Language**: Python 3
- **Libraries**: time, csv, os, cProfile, pstats, multiprocessing, matplotlib, numpy, python-chess.

Algorithms Implemented:

- **Fixed-depth Minimax:** Explores the game tree to a set depth.
- **Selective Deepening Minimax:** Extends the search depth in unstable positions (like checks or captures).
- **Naive Minimax:** A fixed-depth search without any pruning for performance benchmarking.
- **Alpha-Beta Pruning:** An optimization integrated into both Fixed-depth and Selective Deepening Minimax to dramatically reduce the search space.
- **Board Representation & Move Generation:** Custom implementation of simplified chess rules including pawn, knight, bishop, rook, queen, and king movement, legality checks, and check detection.

Additional Data Structures/Algorithms Used:

- **2D List:** Used for board state representation.
- **Python Dictionaries & List:** Used for storing piece values, moves, and simulation results.
- **Recursion:** Used for Minimax search implementations.
- **Parallel Processing:** Implemented using Python's multiprocessing.Pool to run simulations concurrently.
- **FEN String:** A standard data representation used for input and internal processing.
- **Bar Charts:** A simple algorithm/data structure used for visualizing comparative data.

Distribution of Responsibilities and Roles:

Mohamed Hussein: Focused on optimizing the search algorithms. This included implementing Alpha-Beta Pruning and refining the `main` simulation loop. Also created the graphical analysis to visualize the project's data and helped write the final document.

Zayne Bournand: Developed the project foundation with the initial search algorithms and the essential board logic, and built the `main` simulation framework and the evaluation function. Also wrote the `README.md` and helped with the document.

Dajana Seitllari: Contributed to the project's presentation and clarity. She created the video, added clarifying comments to the code, and helped write the final document.

Analysis:

1. Public Dataset Adoption: Instead of generating random board states, we switched to using a large public dataset of 100,000+ real chess positions in FEN format from Kaggle. This provided more realistic and diverse test scenarios, directly addressing a recommendation in our proposal and improving the validity of our comparison.
2. Expanded Piece Logic: While initially focusing on core piece types, we expanded our `board.py` to include simplified movement logic for Knights, Bishops, and Rooks. This enriched the complexity of the simulated game environment, allowing for more comprehensive testing of our search algorithms in varied tactical situations (like more diverse capture and check scenarios).
3. Refined Instability Detection: We defined the `is_unstable` function to identify critical positions based on immediate checks, captures by the current player, and captures threatened by the opponent. This ensures our Selective Deepening algorithm focuses its deeper search efforts where tactical outcomes are most impactful.
4. Multiprocessing Integration: Recognizing the significant computational cost of minimax search, we integrated Python's `multiprocessing` module. This allowed us to parallelize the simulation of individual board evaluations across multiple CPU cores, drastically reducing the overall runtime (about ~10-14x speedup observed). This made large-scale simulations more feasible and directly leveraged the power of modern multi-core processors.
5. Enhanced Performance Profiling: We implemented `cProfile` and `pstats` to gain precise insights into our code's performance. This profiling was crucial for identifying computational bottlenecks (like in move generation and check detection) and validating the effectiveness of our optimization strategies like multiprocessing.

6. Introduction of Naive Minimax Benchmark and Alpha-Beta Pruning: We made a significant change by implementing a "Naive Minimax" (without Alpha-Beta Pruning) as a performance benchmark. We then optimized both our Fixed-Depth and Selective Deepening algorithms by integrating Alpha-Beta Pruning. This created a two-phased comparison that demonstrates the massive performance gains of Alpha-Beta, which is the most fundamental optimization in game tree search.

Time Complexity:

The core Minimax search algorithms (both Fixed-Depth and Selective Deepening) exhibit a worst-case time complexity of O(b^d), where b is the average branching factor (number of legal moves) and d is the search depth. This exponential growth accounts for the significant computational time.

The introduction of Alpha-Beta Pruning dramatically improves the practical performance of our search algorithms by pruning sections of the game tree that are guaranteed not to be selected. In the best-case scenario (with perfect move ordering), this can reduce the time complexity from O(b^d) to roughly O(b^(d/2)).

Within the game logic, the most expensive operations, dominating the cost per node in the search tree, are:

- `generate_legal_moves`: O(M·N·K), where M is the number of pseudo-legal moves, N is board size, and K relates to piece ranges. This is due to iteratively checking king safety for each pseudo-legal move, which involves deep checks and board copying.
- `is_king_in_check` and `generate_pseudo_legal_moves`: Both are approximately O(N·K), as they iterate over the board and piece movement patterns.

While all these operations contribute to the exponential nature, the multiprocessing implementation effectively reduced the time for the total simulation from O(L·Cavg) to approximately O((L/P)·Cavg), where L is the number of loaded boards, Cavg is the average cost per board evaluation, and P is the number of CPU cores.

Reflection:

Experience:

The project was a valuable learning experience. While we could have started earlier to give ourselves more time to debug and address runtime issues, we were able to collaborate effectively. Each member contributed to the project, and together we explored and analyzed an interesting topic. With better time management, the process could have been smoother, but overall was a good experience.

Challenges:

Our primary challenges revolved around performance optimization due to the exponential nature of search algorithms. Initially, even simple Minimax was very slow in Python. This led to the need for profiling to pinpoint bottlenecks (primarily in move generation and king safety checks) and subsequently implementing multiprocessing and Alpha Beta Pruning to achieve acceptable simulation runtimes. Additionally, managing the complexity of chess rules for pieces like Rooks, Bishops, and Knights, while simplifying others, required careful balancing to maintain the project's scope and correctness.

Changes in project and/or workflow:

If we were to start this project again, beginning earlier would definitely be a priority, especially considering the unexpected issues we encountered along the way. Better distribution of roles could also have improved our efficiency. Additionally, setting clear deadlines for each part of the project would help keep everyone accountable and ensure steady progress. These adjustments would make our workflow smoother and reduce last-minute pressure.

Learned:

Mohamed Hussein – This project taught me crucial lessons in memory safety, as my initial attempts to parallelize calls to the external Stockfish engine led to segmentation faults from resource exhaustion. I solved this by implementing a memory-safe, two-phase design that first runs our custom engines in parallel and then uses a single, persistent Stockfish instance sequentially to judge the results, which prevents crashes and resource contention.

Zayne Bournand – I gained a deep understanding of search algorithms, specifically Minimax and Selective Deepening. A major learning experience was confronting and mitigating the performance bottleneck of exponential complexity, which I successfully addressed by implementing multiprocessing to significantly reduce execution time.

Dajana Seitllari – This project helped me understand how different search algorithms work, specifically chess algorithms. How the algorithms make the decisions and then comparing them to an engine (Stockfish). The difference between speed and quality are both affected by the strategy taken by search depth. Doing the video breakdown helped me understand the complete full flow of the project and how important each part is into its contribution.