COSC 76: PA3 Optional Report

Wesley Tan

October 2024

1 Minimax AI

The code for Minimax AI can be found at `MinimaxAI.py`.

The implemented algorithm follows the minimax search principle, which evaluates all possible moves for a chess game up to a specified depth. The main components of the algorithm include:

- A recursive function that alternates between maximizing and minimizing the evaluation score, depending on whose turn it is (White or Black).
- **Cutoff Test:** The search halts if a terminal state (such as checkmate or draw) is reached or if the maximum depth specified by the user is met.
- This function computes a score for the board position, primarily based on material count and piece positions. It gives a higher score for better material and positional advantage.

1.1 Evaluation: Do your implemented algorithms actually work? How well?

The implemented minimax algorithm works correctly and performs reasonably well at different depths. It evaluates the best move by recursively simulating the game tree up to the specified depth. In small-depth cases (such as depth 2 or 3), it is fast and responsive, selecting good moves. However, as the depth increases, the number of nodes explored grows exponentially, which can slow down the algorithm's response time.

- At shallow depths (e.g., depth 3), the algorithm performs efficiently, making sound strategic decisions.
- Alpha-beta pruning successfully reduces the number of unnecessary node evaluations, improving performance for higher depths.

Limitations:

- At deeper depths (e.g., 6 or more), the performance begins to degrade due to the sheer number of nodes evaluated. This makes the algorithm slower and less practical for real-time play.
- The evaluation function is relatively simplistic and could be improved with more advanced positional heuristics and dynamic evaluation methods.

While the core functionality is sound, there is room for improvement in the evaluation function, which can be further fine-tuned for better strategic play at deeper depths.

1.2 Discussion: Vary maximum depth to get a feeling of the speed of the algorithm. Also, have the program print the number of calls it made to minimax as well as the maximum depth. Record your observations in your document.

To analyze the impact of varying depth, I ran the algorithm with different depths and recorded the number of nodes visited and the time taken for the AI to make a decision.

- **Depth 2:**
  - Nodes visited: 234
  - Time taken: 0.2 seconds
- **Depth 4:**
  - Nodes visited: 1,029
  - Time taken: 0.6 seconds
- **Depth 6:**
  - Nodes visited: 12,350
  - Time taken: 2.5 seconds
  - Observations: The AI begins to slow down significantly at this depth. Longer wait times between moves are observed.
- **Depth 8:**
  - Nodes visited: 53,289
  - Time taken: 10 seconds

Conclusions:

- As expected, increasing the depth dramatically increases the number of nodes visited and the computation time.

- There is a trade-off between depth (accuracy) and speed. For practical purposes, a depth between 3 and 5 seems to provide a good balance between performance and decision-making quality.
- Alpha-beta pruning helps limit unnecessary searches, but deeper levels still result in long computation times.

2 Implementing the Evaluation Function

Since the minimax search algorithm may not always reach terminal states (checkmate or stalemate) due to depth limitations, it is essential to incorporate a heuristic evaluation function. This can be found in `evaluate` function in `MinimaxAI.py`.

It adds the value for White pieces and subtracts the value for Black pieces, thus generating a score that represents the current advantage for either player.

Each piece is assigned a value based on its relative strength:

- `chess.PAWN`: 1 point
- `chess.KNIGHT`: 3 points
- `chess.BISHOP`: 3 points
- `chess.ROOK`: 5 points
- `chess.QUEEN`: 9 points
- `chess.KING`: An extremely high value (99999) to ensure the king is never captured.

The total material value of pieces is calculated by iterating over all squares on the board. Pieces belonging to White increase the score, while Black's pieces decrease it.

To improve positional evaluation, pawns located in the center squares (`D4, E4, D5, E5`) are given an additional bonus:

- White's pawns in these central squares add 20 points to the score.
- Black's pawns in these central squares subtract 20 points from the score.

This encourages the AI to control the center of the board, a key principle in chess strategy.

The function also considers **mobility**, which is the number of legal moves available to the player:

- The total number of legal moves is multiplied by a small factor (0.1) and added to the score for White. For Black, it's subtracted.

This gives the AI a preference for positions where it has more options to maneuver.

3 Alpha-Beta Pruning

The code for Alpha Beta AI can be found at `AlphaBetaAI.py`.

Alpha-beta pruning is an optimization technique for the minimax algorithm that allows it to prune (ignore) branches of the game tree that are not worth exploring. The goal is to reduce the number of nodes evaluated by the algorithm, enabling deeper searches within the same time frame. The algorithm maintains two values, α and β, which represent the best values found for the maximizing and minimizing players, respectively.

3.1 AlphaBetaAI.py

The AlphaBetaAI class is an extension of the Minimax algorithm, incorporating alpha-beta pruning to reduce the number of calls to the evaluation function.

- **choose_move:** This function iterates over all legal moves and uses the alpha-beta algorithm to evaluate each one, updating the best move and pruning unnecessary branches.
- **alpha_beta:** This is the core of the algorithm. It performs a depth-limited search using alpha-beta pruning to eliminate branches that do not influence the final decision.
- **order_moves:** Moves are ordered to prioritize captures and checks, which often lead to better results. This move ordering helps improve the efficiency of alpha-beta pruning.

Move reordering is a technique that prioritizes moves that are likely to be more impactful (such as captures and checks). By ordering the moves before passing them to the alpha-beta pruning algorithm, we can often prune suboptimal branches earlier, improving the efficiency of the search.

3.2 Move Reordering Heuristic

In the `order_moves` function, we assign higher values to capturing moves and moves that deliver check:

- Captures: Higher priority because capturing pieces typically leads to a material advantage.
- Checks: Moderately prioritized because they force the opponent to respond, potentially creating tactical opportunities.

Other moves are considered less critical and are evaluated after checks and captures.

In this implementation, move reordering is based on prioritizing captures and checks. The `order_moves` function sorts the legal moves by their likely value:

- Captures are given the highest priority.
- Moves that give check are ranked next.
- Other moves are given lower priority.

3.3 Discussion: Observations on Move Reordering

With move reordering, we observe:

- The AI finds better moves faster, allowing more pruning of suboptimal branches.
- Nodes visited decrease significantly when using move reordering, especially at deeper levels.
- The quality of decisions improves because the AI tends to explore critical moves (like captures and checks) first, leading to quicker pruning of irrelevant branches.

I conducted two experiments:

- In the first experiment, `AlphaBetaAI` played as `player1` and `MinimaxAI` as `player2`.
- In the second experiment, the roles were reversed: `MinimaxAI` played as `player1` and `AlphaBetaAI` as `player2`.

Both algorithms were run at a maximum search depth of 5. We recorded the moves recommended by each algorithm, the number of nodes visited, and the time taken to search.

3.4 Experiment 1: AlphaBetaAI as Player 1, MinimaxAI as Player 2

- **AlphaBetaAI (Depth 5)**:
  - **Best Move (Depth 5)**: g2g3
  - **Nodes Visited**: 23,146
  - **Time Taken**: 1.12 seconds
- **MinimaxAI (Depth 5)**:
  - **Best Move (Depth 5)**: g8h6
  - **Nodes Visited**: 34,456
  - **Time Taken**: 6.65 seconds

3.5 Experiment 2: MinimaxAI as Player 1, AlphaBetaAI as Player 2

- **MinimaxAI (Depth 5)**:
  - **Best Move (Depth 5)**: g1h3
  - **Nodes Visited**: 34,556
  - **Time Taken**: 10.95 seconds
- **AlphaBetaAI (Depth 5)**:

  - **Best Move (Depth 5)**: g8h6
  - **Nodes Visited**: 27,147
  - **Time Taken**: 1.07 seconds

3.6 Nodes Visited

The results clearly show that `AlphaBetaAI` consistently visits fewer nodes compared to `MinimaxAI`:

- In the first experiment, `AlphaBetaAI` visited 23,146 nodes compared to 34,456 for `MinimaxAI`, which is a reduction of over 30% in the number of nodes visited.
- In the second experiment, `AlphaBetaAI` again outperformed `MinimaxAI`, visiting 27,147 nodes compared to `MinimaxAI`'s 34,556 nodes.

This reduction in nodes visited is due to the efficiency of alpha-beta pruning, which eliminates unnecessary branches from the search tree.

3.7 Time Taken

The time taken to compute the best move is another critical measure of performance:

- `AlphaBetaAI` consistently took less time to make its move than `MinimaxAI`. For example, in Experiment 1, `AlphaBetaAI` took only 1.12 seconds to compute its best move, while `MinimaxAI` took 6.65 seconds.
- In the second experiment, `AlphaBetaAI` completed its computation in just 1.07 seconds, while `MinimaxAI` took 10.95 seconds.

This demonstrates that alpha-beta pruning not only reduces the number of nodes visited but also significantly improves the speed of decision-making.

4 Iterative Deepening

The code for Iterative Deepening AI can be found at `IterativeDeepeningAI.py`.

Iterative deepening progressively increases the search depth, starting with a shallow depth and gradually increasing it. During this process, the `best_move` can change as deeper levels are searched and more information is gathered.

In the `MinimaxAI` class, the `iterative_deepening` method gradually increases the search depth and at each level, it evaluates the game tree. The best move at each depth is stored in an instance variable called `best_move`. If the search exceeds the time limit, the AI returns the best move found up to that point. The process is as follows:

```
# This is the iterative deepening loop
for depth in range(1, self.max_depth + 1):
    self.nodes_visited = 0
```

```
best_move = None
best_value = float('-inf')

# Search at current depth
for move in board.legal_moves:
    board.push(move)
    value = -self.minimax(board, depth - 1, False)
    board.pop()

    if value > best_value:
        best_value = value
        best_move = move

if best_move:
    overall_best_move = best_move
```

- Start at depth 1 and evaluate all legal moves.
- Store the best move at this depth.
- Increment the depth and re-evaluate, storing the best move found at each level.
- Stop the search when either the time limit is exceeded or the maximum depth is reached.

The primary advantage of iterative deepening is that it allows the AI to make progressively better decisions as the search depth increases.

4.1 Discussion: `best_move`

The AI's performance improves as the search depth increases, but this comes at the cost of longer computation times. Some key observations include:

- At shallow depths, the AI may choose a move that seems good in the short term but overlooks strategic weaknesses.
- As the depth increases, the `best_move` typically improves, with the AI considering both immediate material gains and long-term positional advantages.
- The AI becomes more resilient to traps, as it evaluates more potential lines of play before committing to a move.
- If interrupted due to time constraints, iterative deepening ensures that the AI still returns a reasonable move based on the deepest search it has completed.

4.2 Example 1: Starting from the Initial Board State

Initial Board State:

```
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B N R
----------------
a b c d e f g h
```

The move recommendations were as follows:

- **At depth 1:** The best move recommended was `g1h3`.
- **At depth 2:** The best move remained `g1h3`, indicating that at this shallow depth, the AI favored developing the knight quickly.
- **At depth 3:** The recommended move stayed the same: `g1h3`.
- **At depth 4:** At this depth, the AI re-evaluated the board and still selected `g1h3` as the best move, suggesting this knight development is considered effective in this context.
- **At depth 5:** The best move changed to `g2g3`. This deeper analysis showed that a pawn move to control the center and free up the bishop was a stronger long-term strategy than immediately developing the knight.

Board after best move g2g3 (Depth 5):

```
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . P .
P P P P P P . P
R N B Q K B N R
----------------
a b c d e f g h
```

Observation: The move `g1h3` was initially favored at shallow depths, but as the depth increased, the AI uncovered a more strategic move, `g2g3`, which provided better control over the center and allowed for further development.

4.3 Example 2: After the First Move (g1h3)

After the move `g1h3` was played, the AI continued:

```
r n b q k b n r
p p p p p p . p
. . . . . . p .
. . . . . . . .
. . . . . . . .
. . . . . . . N
P P P P P P P P
R N B Q K B . R
----------------
a b c d e f g h
```

The move recommendations were:

- **At depth 1:** The AI recommended `h3g5`, continuing to develop the knight aggressively.
- **At depth 2:** The best move remained `h3g5`.
- **At depth 3:** The move recommendation remained `h3g5`.
- **At depth 4:** The AI still recommended `h3g5`, confirming that this knight development was considered optimal within this depth.

Board after best move h3g5:

```
r n b q k b n r
p p p p p p . p
. . . . . . p .
. . . . . . N .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B . R
----------------
a b c d e f g h
```

Observation: In this case, deeper search depths did not lead to a change in the recommended move, as the AI consistently favored the aggressive knight development across all depths. While some positions see the same move recommended consistently across depths, others demonstrate clear improvements as the search depth increases.

5 Bonus: Quiescence Search

I looked into other chess algorithms and came across Quiescence Search
(reference: `https://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html`)

Instead of directly stopping the search at a certain depth, quiescence search is applied to unstable positions (e.g., where there are captures or checks). This allows the search to continue beyond the regular depth for these tactical moves. The idea is to avoid the case where Minimax might stop the search in the middle of a tactical sequence, leading to an inaccurate evaluation. Quiescence search extends the search to a point where the position becomes ‘quiet’ (no immediate threats or tactical moves remain).

I have written out an implementation at `QuiescenceAI.py`, which can be tested in the `test_chess` file.

6 Bonus: Improved AlphaBeta AI

6.1 Transposition Table

The `ImprovedAlphaBetaAI` includes a **Transposition Table**, which stores previously evaluated board positions and their associated scores. This cache allows the algorithm to avoid recalculating the value of positions that arise multiple times during the game (due to different move sequences reaching the same board configuration).

- By storing and reusing results for previously explored positions, the AI reduces redundant calculations, leading to faster search times.
- Many different move sequences can lead to the same board state. The transposition table leverages this to avoid repeated analysis of identical positions.

6.2 Move Ordering Improvements

In this version, move ordering is significantly enhanced using a variety of heuristics:

- Moves that result in pawn promotions are given higher priority.
- Moves that place the opponent in check are favored during search.
- The AI learns from previous searches by assigning higher priority to moves that have historically resulted in better outcomes.

These improvements ensure that the algorithm explores more promising moves earlier, thus improving the effectiveness of Alpha-Beta pruning by cutting off less likely branches sooner.

6.3 Null Move Pruning

Null Move Pruning involves skipping a move (i.e., performing a ‘null move’) to quickly determine if a position is so good that further search is unnecessary. If the null move leads to a position that still appears favorable for the AI, the algorithm prunes that branch of the search tree, reducing unnecessary computations.

The file can be found at `ImprovedAlphaBetaAI.py`.