1 Implementation

The search process is encapsulated in the `AlphaBetaEngine` class and is primarily driven by two functions: *search*() and *f*4().

1.1 Root Search: *search*() @alphabeta.cpp:L318

The *search*() function serves as the entry point for the agent's decision-making process. Its primary responsibilities are:

- **Opening Strategy** @L324 & L332: To adopt an aggressive opening, the agent prioritizes flipping the center piece (D2) if the board is entirely hidden. If playing as the second player (with one piece revealed), it employs a heuristic to flip a piece farthest from the opponent's revealed piece to minimize immediate capture risks.
- **Unrevealed Piece Tracking** @L63: Maintains a strict state of `unrevealed_count`. This list is updated incrementally after every ply. Explicit tracking ensures that captured hidden pieces do not skew the probability calculations for Star1 pruning.
- **Iterative Deepening** @L373: The search performs iterative deepening starting from depth 1, allowing the agent to return the best-found move immediately if the time limit is exceeded.
- **Store Choice** @L390: The agent stores the hash value of the current board and the chosen move to avoid cyclic moves when in a dominating position.

1.2 Negascout: *f*4() @alphabeta.cpp:L154

- **Null window search** @L193-226: Instead of using $n = max(\alpha, m) + 1$ as discussed in class, I use decimals to reward positions closer to targets. Consequently, the window is adjusted to $n = max(\alpha, m) + 0.001$. Note that the null window is **not** used for chance nodes, where the upper bound is set to the original β.
- **Star1** @L88: Unlike Star0, which uses infinite bounds, Star1 requires specific maximum and minimum scores. I restricted the score range by investigating the distribution of the original sparse scores using $RET\_LOG()$ (@L15). Since my agent usually searches 2 to 3 layers deep (up to 15 layers in end games, verified via $log\_position()$), I set V_MIN and V_MAX to ± 320 (derived from $\pm(AB\_WIN\_SCORE + depth)$).

  To ensure calculation correctness, I use backpropagation: temporarily decrementing the count of the piece in `unrevealed_count` before searching deeper with *f*4(), and adding the count back upon return.
- **Transposition Table (TT)** @transposition_table.cpp: Before expanding a node, the Zobrist hash is used to probe the TT. If a valid entry is found, the stored score is returned, and both the upper and lower bounds are adapted. Although any hit could theoretically aid move ordering, I found that strictly using entries that are both a hit AND not shallower than the current depth (@L8) yields better results.

- **Zobrist Hash** @zobrist.cpp: I use `pcg64` to generate hash values. When a position is given, the hash is calculated via *compute_zobrist_hash*() (@L13). It is updated incrementally via *update_zobrist_hash*() (@L32) by XORing only the side to move, move source, destination, and piece type, avoiding the need to iterate through all 32 cells.

- **Leaf Node Evaluation** @L79: If a terminal node is reached, the function returns a win/loss score plus the depth to reward faster wins. Otherwise, it evaluates the position via *pos_score*() (@L404).

- **Time Control**: I allocate 5000ms for a given position. If a search is interrupted by a timeout, it returns 0 (@L155, 196, 205). However, if a partial solution was found, it is still returned (@L391). I previously attempted to allocate time dynamically based on estimated remaining plies (@L307), but this degraded performance, so it was discarded.

1.3 Enhancements

1.3.1 Successful Enhancements

- **History Heuristic** @L213, 233, 256: As noted in Slide 8 (p.46), "Transposition tables plus history heuristic provide the best combination." Since TT is required for this assignment, I incorporated the history heuristic to assist in move ordering. The improvement is demonstrated in the experiment results. The weight is calculated as 2^{depth}, and the entries are aged through right shifts to prevent overflow.

- **Move Ordering** @L264: Moves are sorted to maximize pruning efficiency. The priority order is:
  1. Moves from Transposition Table
  2. Captures (Using predefined `yummy_table` @alphabeta.h:L33)
  3. Flips (To avoid draws)
  4. History Heuristic Moves

- **Dynamic Material Values**: The evaluation function for non-terminal nodes is a hybrid of a pre-calculated material lookup table (generated via `gen_val.cpp` and loaded at @L52) and dynamic positional scoring.
  - **Pre-calculated Material Table**: A 2916×2916 matrix stores the theoretical win/loss value for every possible material combination. The scores are assigned based on five cases:
    1. Identical materials: 0
    2. All pieces eliminated: $+ELIMINATION\_SCORE$ if we win, otherwise $-ELIMINATION\_SCORE$.
    3. One side owns pieces that can capture all opposing pieces (@L72): Separated into 5 tiers, where tier i implies owning i pieces capable of clearing the board.

(Note: If only 1 opponent piece remains, we assign Tier 2 instead of 1, as we can likely trap it at the corner).

4. Fallback: If none of the above apply, calculate based on static material values.

– **Positional Heuristics** @L425: Bonus points are awarded for pieces closer to target.

1.3.2 Discarded Enhancements

- **Flipping Budget and Cooldown** @L188: I implemented this based on Slide 12 (p.62). Although the agent outperformed the baseline (see experiments), it became prone to draws due to reluctance to flip, often wandering aimlessly or losing if the flipped piece could not capture an opponent.

- **Evaluation Function from Paper** @piece_score.cpp: I attempted to adapt the concept from Chance Node Searching and Related Problems in Computer Chinese Dark Chess. However, the reliance on floating-point ratios (player score vs. total score) made it difficult to control, so it was ultimately discarded.

2 Experiment Results

The experimental baseline is a Star 0 agent. The testing follows an incremental progression: for any given row, Agent A incorporates all optimizations listed in the rows **above** it. Agent B introduces the specific optimization listed in the current row. Thus, each row measures the marginal gain of that specific feature.

Optimization	Games	A Wins	B Wins	Draws	Win Rate (A vs B)
TT	90	20	28	42	22.2%-31.1%
O(1) hash	60	6	14	40	10.0%-23.0%
modify alpha beta when fetching tt entry	90	17	19	54	18.9%-21.1%
fix unrevealed list (ver1)	90	11	19	60	12.2%-21.1%
History heuristic	120	20	27	73	16.6%-22.5%
Negascout	120	33	43	44	27.5%-35.8%
Star 1+fix unrevealed list (ver2)	120	21	28	71	17.5%-23.3%
Flip budget+cooldown	120	8	19	93	6.7%-15.8%

Implementation Challenges & Adjustments:

- **Modify Alpha-Beta during TT Probe:** Initially, I hypothesized that narrowing the α-β window based on TT bounds might reduce the frequency of exact hash hits, potentially weakening the search. However, after consulting with Prof. Hsu, I confirmed this concern was unfounded. In the final implementation, α and β are passed by reference during the TT probe and are immediately updated if a valid entry allows for a narrower window.

- **Fix Unrevealed List (ver1):** The initial assignment framework did not disclose position updates when it was the opponent's turn, which prevented accurate tracking of flip results. My initial workaround—deriving the unrevealed count by subtracting visible pieces from the total—was flawed because it treated captured hidden pieces as merely "unrevealed," leading to overly optimistic flip probabilities.

  My first attempt at a fix (**ver1**) used the formula:

  $$count = \min(stored\_count, init\_total - visible\_on\_board)$$

  However, this logic failed to account for "hidden deaths." Consider this scenario:

  1. We have 2 unrevealed red soldiers ($stored\_count = 2$) and other 3 are revealed ($visible\_on\_board = 3$).
  2. One red soldier is captured ($stored\_count = 2$, $visible\_on\_board = 2$).
  3. One of the unrevealed soldier is flipped and becomes visible ($visible\_on\_board = 3$).

  In this case, the actual unrevealed count should be 1. However, the formula calculates $\min(2, 5 - 3) = 2$, incorrectly implying 2 soldiers are still hidden. This bug caused a severe regression when Star1 was integrated (resulting in 4 wins and 38 losses over 60 games), necessitating the strict incremental tracking used in **ver2**.

- **Negascout**: Initially performed poorly when distance was used as a penalty; performance improved after switching to rewarding closer distances.

A more detailed explanation of other optimizations can be found by clicking the optimization name in the leftmost column.