**1. What is Alpha-Beta Pruning?**

Before diving into the specific terms, it's helpful to understand **alpha-beta pruning** itself.

* **Alpha-Beta Pruning** is a technique used in search algorithms (like those in chess engines) to eliminate branches of the search tree that don't need to be explored because they won't affect the final decision. This makes the search faster and more efficient.

**2. Key Concepts: Fail-Hard Beta Cutoff, Fails High, and Fails Low**

Now, let's explain the three terms you mentioned:

**a. Fail-Hard Beta Cutoff**

**Simple Explanation:** Imagine you're evaluating possible moves in chess. If you find a move that's so good that your opponent can't counter it effectively, you can stop considering other moves because this move is already good enough.

**Detailed Breakdown:**

* **Beta (β):** This represents the minimum score that the maximizing player (e.g., White) is assured of. Think of it as a threshold.
* **Fail-Hard Beta Cutoff:** When you find a move that **reaches or exceeds this beta threshold**, you can immediately stop looking at other moves in that branch. There's no need to explore further because the opponent won't allow this branch to be chosen—they'll choose a different path instead.

**Analogy:** Imagine you're shopping for a new phone. You've set a minimum satisfaction level (beta). If you find a phone that meets or exceeds this level, you decide it's good enough and stop looking for other options.

**b. Node (Move) Fails High**

**Simple Explanation:** This happens when a move is so strong that it's better than what you were initially aiming for. Because it's so good, you don't need to check other moves in that sequence.

**Detailed Breakdown:**

* When evaluating a move, if its score is **greater than or equal to beta**, it **fails high**. This means the move is so effective that the opponent will avoid this path, making further exploration unnecessary.

**Analogy:** Continuing the phone shopping example, if you find a phone that not only meets but exceeds your satisfaction level, you recognize it's an excellent choice and stop considering other phones.

**c. Node (Move) Fails Low**

**Simple Explanation:** This occurs when a move isn't good enough compared to what you're already confident you can achieve. Since it's not promising, you don't need to consider it further.

**Detailed Breakdown:**

* When evaluating a move, if its score is **less than or equal to alpha**, it **fails low**. This means the move doesn't improve your position enough, so there's no benefit in exploring it further.

**Analogy:** If you find a phone that doesn't meet your minimum satisfaction level, you decide it's not worth considering, so you move on to other options.

**3. Putting It All Together**

Imagine you're evaluating a series of chess moves to decide the best one:

1. **Start with Alpha (α) and Beta (β):**
   * **Alpha (α):** The best score the maximizing player is assured of so far.
   * **Beta (β):** The minimum score the opponent is assured of.
2. **Evaluate Each Move:**
   * **If a move's score ≥ β:**
     + **Fail-Hard Beta Cutoff:** Stop evaluating further moves in this branch because this move is good enough, and the opponent will choose differently.
     + **Fails High:** This move is so strong that exploring more moves here is unnecessary.
   * **If a move's score ≤ α:**
     + **Fails Low:** This move isn't promising enough, so skip it and consider other moves.
   * **Otherwise:**
     + Update **Alpha (α)** with the new higher score and continue evaluating.

**4. Visual Example**

Let's visualize this with a simple tree:

Current Position

/ \

Move1 Move2

/ \ / \

Move1A Move1B Move2A Move2B

* **Evaluating Move1:**
  + **Move1A:** Score = 50
    - Update **Alpha (α)** to 50.
  + **Move1B:** Score = 70
    - **Fails High:** 70 ≥ β (let's say β = 60)
    - **Cutoff:** Stop evaluating Move1B's siblings. No need to consider other moves under Move1.
* **Evaluating Move2:**
  + **Move2A:** Score = 40
    - **Fails Low:** 40 ≤ α (50)
    - **Skip:** Don't consider Move2A's siblings.
  + **Move2B:** Not evaluated due to fail low.

**Result:** Only Move1 is considered, and Move2 is skipped because its best move fails low.

**5. Why Does This Matter?**

By using **fail-hard beta cutoffs** and recognizing when moves **fail high** or **fail low**, the chess engine:

* **Saves Time:** It doesn't waste resources evaluating moves that won't influence the final decision.
* **Enhances Efficiency:** It focuses on the most promising moves, making the search process faster and more effective.

**6. Recap**

* **Fail-Hard Beta Cutoff:** Stops evaluating further moves in a branch when a move meets/exceeds the beta threshold.
* **Fails High:** A move is so good that no need to look further in that branch.
* **Fails Low:** A move isn't good enough compared to the current best, so skip it.

By understanding and implementing these concepts, your chess engine can perform more efficiently, making quicker and smarter decisions during gameplay.

Understanding why your chess engine takes a long time to search to depth 6, despite having a move generator that produces 4.5 million moves per second, involves examining several factors related to how search algorithms operate, especially in the context of chess engines. Let's break down the potential reasons and explore solutions to optimize your engine's performance.

**1. The Nature of Tree Search Algorithms**

**Exponential Growth of Nodes**

* **Branching Factor (b)**: In chess, the average number of legal moves per position (branching factor) is roughly between 30 to 40. Let's use 35 for illustration.
* **Search Depth (d)**: The number of plies (half-moves) the engine searches ahead. In your case, d = 6.
* **Total Nodes (N)**: The total number of positions the engine evaluates can be approximated as:

N=bd=356≈1.8×109 nodesN = b^d = 35^6 \approx 1.8 \times 10^9 \text{ nodes}

This means your engine needs to evaluate approximately 1.8 billion positions to search to depth 6.

**Time Calculation**

* **Move Generation Rate**: 4.5 million moves per second.
* **Total Time (T)**:

T=NMove Generation Rate=1.8×1094.5×106=400 seconds≈6.7 minutesT = \frac{N}{\text{Move Generation Rate}} = \frac{1.8 \times 10^9}{4.5 \times 10^6} = 400 \text{ seconds} \approx 6.7 \text{ minutes}

This theoretical calculation highlights the **inherent challenge**: even with a high move generation rate, the sheer number of nodes makes deeper searches time-consuming.

**2. Implementation Overheads in C#**

**a. Recursion Overhead**

* **Method Calls**: Recursive algorithms like Negamax introduce overhead due to frequent method calls, especially in managed languages like C# where each call has additional stack management.
* **Solution**: Consider implementing **iterative deepening** or using **loops** instead of deep recursion where possible. Additionally, **tail recursion optimization** isn't natively supported in C#, so minimizing recursion depth can help.

**b. State Management**

* **Copying Game State**: Methods like CopyGameState and RestoreGameState involve copying arrays, which are expensive operations in C#.
* **Solution**:
  + **Bitwise Operations**: Leverage bitwise operations to manage board states without copying large data structures.
  + **Undo Moves**: Instead of copying the entire game state, implement an **undo move** mechanism that reverses the changes made by a move. This is significantly faster and reduces memory usage.
  + **Immutable Structures**: Use immutable data structures where feasible to avoid unnecessary copies.

**c. Memory Management**

* **Garbage Collection**: Frequent allocations and deallocations can trigger garbage collection, causing pauses and reducing performance.
* **Solution**:
  + **Object Pooling**: Reuse objects like MoveObjects to minimize allocations.
  + **Structs vs. Classes**: Use **structs** for lightweight objects to allocate them on the stack rather than the heap.

**3. Language Performance Considerations**

**C# vs. C/C++**

* **Performance Gap**: While C# is a high-performance language, it generally can't match the raw speed of optimized C or C++ code, especially for compute-intensive tasks like chess engine searches.
* **Solution**:
  + **Unsafe Code**: Utilize unsafe code blocks in C# to perform low-level memory operations, reducing overhead.
  + **Parallelism**: Implement **parallel search** using multi-threading to take advantage of multi-core processors.
  + **Compiler Optimizations**: Ensure that the compiler optimizations are enabled (e.g., using the Release build configuration with optimizations).

**4. Search Algorithm Optimizations**

**a. Alpha-Beta Pruning Enhancements**

* **Move Ordering**: Prioritize moves that are more likely to cause beta cutoffs (e.g., captures, checks) to maximize pruning efficiency.
* **Implementation**:
  + **Most Valuable Victim - Least Valuable Aggressor (MVV-LVA)**: Order capture moves based on the value of the captured piece versus the capturing piece.
  + **Killer Heuristic**: Remember and prioritize moves that have caused cutoffs in similar positions.

**b. Transposition Tables**

* **Caching Evaluated Positions**: Store the results of previously evaluated positions to avoid redundant calculations.
* **Implementation**:
  + **Hash Tables**: Use efficient hash tables to store position evaluations.
  + **Zobrist Hashing**: Implement Zobrist hashing to generate unique keys for board positions quickly.

**c. Iterative Deepening**

* **Progressive Deepening**: Search incrementally deeper, using results from shallower searches to inform deeper ones (e.g., better move ordering).
* **Benefits**:
  + **Improved Move Ordering**: Results from shallower searches help prioritize better moves earlier.
  + **Time Management**: Allows for better handling of time constraints by having intermediate results.

**d. Bitboard Optimizations**

* **Bitwise Magic**: Utilize bitwise operations and magic bitboards for efficient move generation and attack calculations.
* **Precomputed Tables**: Use precomputed tables for attacks to reduce computation during the search.

**5. Profiling and Benchmarking**

**Identify Bottlenecks**

* **Profiling Tools**: Use profiling tools like **Visual Studio Profiler**, **JetBrains dotTrace**, or **PerfView** to identify which parts of your code consume the most time.
* **Focus Areas**: Likely candidates include move generation, state copying/restoring, and recursive calls.

**Benchmark Different Components**

* **Isolate and Test**: Benchmark individual components (e.g., move generation, evaluation) to understand their performance characteristics.
* **Optimize Critical Paths**: Focus optimization efforts on the most time-consuming parts identified during profiling.

**6. Practical Steps to Optimize Your Engine**

**a. Implement Move Ordering**

Prioritize moves that are captures or checks to increase the chances of early beta cutoffs.

// Example: Sorting capture moves first based on MVV-LVA

var captureMoves = moveList.moves.Where(move => MoveGenerator.GetMoveCapture(move))

.OrderByDescending(move => GetCaptureScore(move))

.ToList();

var nonCaptureMoves = moveList.moves.Where(move => !MoveGenerator.GetMoveCapture(move))

.ToList();

var orderedMoves = captureMoves.Concat(nonCaptureMoves).ToList();

**b. Use an Undo Move Mechanism**

Instead of copying and restoring the entire game state, implement an undo system.

public static bool MakeMove(int move)

{

// Make the move

// ...

// Push the move onto a stack with the necessary information to undo it

MoveStack.Push(move, capturedPiece, previousCastlePerm, previousEnpassant);

return true;

}

public static void UndoMove()

{

var moveInfo = MoveStack.Pop();

// Restore the game state using moveInfo

// ...

}

**c. Implement Transposition Tables**

Store and retrieve evaluations of previously seen positions.

public static class TranspositionTable

{

private static Dictionary<ulong, int> table = new Dictionary<ulong, int>();

public static bool TryGetValue(ulong key, out int value)

{

return table.TryGetValue(key, out value);

}

public static void Add(ulong key, int value)

{

table[key] = value;

}

}

// Usage in NegaMax

ulong positionKey = GenerateZobristHash();

if (TranspositionTable.TryGetValue(positionKey, out int cachedScore))

{

return cachedScore;

}

// After evaluation

TranspositionTable.Add(positionKey, score);

**d. Optimize Bitboard Operations**

Ensure that bitboard manipulations are as efficient as possible.

// Example: Using bitwise operations for move generation

public static ulong GeneratePawnMoves(int square, Colors side)

{

ulong moves = 0UL;

ulong bit = 1UL << square;

if (side == Colors.white)

{

moves |= (bit << 8) & ~Boards.OccupanciesBitBoards[Colors.both];

// Add more pawn moves (captures, promotions)

}

else

{

moves |= (bit >> 8) & ~Boards.OccupanciesBitBoards[Colors.both];

// Add more pawn moves (captures, promotions)

}

return moves;

}

**e. Parallelize the Search**

Leverage multi-threading to perform parallel searches on different branches.

using System.Threading.Tasks;

private static int NegaMax(int alpha, int beta, int depth)

{

// Base case

if (depth == 0) return Quiescence(alpha, beta);

nodes++;

// Move ordering

MoveObjects moveList = new MoveObjects();

MoveGenerator.GenerateMoves(moveList);

// Parallel processing

int localAlpha = alpha;

object lockObj = new object();

Parallel.For(0, moveList.counter, (i, state) =>

{

int move = moveList.moves[i];

MoveGenerator.CopyGameState(out ulong[] bitboardsCopy, out ulong[] occupanciesCopy, out Colors sideCopy, out int castlePermCopy, out int enpassantSquareCopy);

if (!MoveGenerator.MakeMove(move, false))

{

MoveGenerator.RestoreGameState(bitboardsCopy, occupanciesCopy, sideCopy, castlePermCopy, enpassantSquareCopy);

return;

}

int score = -NegaMax(-beta, -localAlpha, depth - 1);

MoveGenerator.RestoreGameState(bitboardsCopy, occupanciesCopy, sideCopy, castlePermCopy, enpassantSquareCopy);

lock (lockObj)

{

if (score > localAlpha)

{

localAlpha = score;

if (depth == originalDepth)

{

bestMove = move;

}

if (score >= beta)

{

state.Stop();

}

}

}

});

return localAlpha;

}

**Note**: Proper synchronization is crucial to avoid race conditions.

**7. Summary and Next Steps**

**Why Depth 6 Takes Long Despite Fast Move Generation**

1. **Exponential Node Growth**: At depth 6, the number of nodes increases exponentially, making the search inherently time-consuming.
2. **Implementation Overheads**:
   * **State Copying**: Frequent copying and restoring of game states can significantly slow down the search.
   * **Recursion and Method Calls**: Deep recursion and numerous method calls introduce additional overheads in C#.
3. **Lack of Advanced Optimizations**: Without techniques like move ordering, transposition tables, and parallelism, the search remains inefficient at higher depths.

**Actionable Steps to Improve Performance**

1. **Implement an Undo Move Mechanism**: Replace state copying with an efficient undo system to reduce overhead.
2. **Optimize Move Ordering**: Prioritize promising moves to enhance alpha-beta pruning effectiveness.
3. **Introduce Transposition Tables**: Cache evaluations of previously seen positions to avoid redundant computations.
4. **Leverage Parallelism**: Utilize multi-threading to explore multiple branches of the search tree simultaneously.
5. **Profile and Benchmark**: Use profiling tools to identify and address specific bottlenecks in your code.
6. **Consider Low-Level Optimizations**: Use unsafe code, structs, and other low-level optimizations in C# to enhance performance.
7. **Iterative Deepening**: Implement iterative deepening to improve move ordering and manage search time effectively.

**Final Thoughts**

Optimizing a chess engine is a complex task that requires careful consideration of algorithmic efficiency and low-level implementation details. While your move generator is impressively fast, the search process's exponential nature and implementation overheads in C# contribute to longer search times at deeper depths. By systematically applying the optimizations outlined above, you can significantly enhance your engine's performance, enabling deeper and faster searches.

Feel free to reach out with specific questions or for further assistance as you implement these optimizations!

Result of basic move scoring and move ordering based on capture score MvL-Lva

8 ♜ . . . ♚ . . ♜

7 ♟ . ♟ ♟ ♛ ♟ ♝ .

6 ♝ ♞ . . ♟ ♞ ♟ .

5 . . . ♙ ♘ . . .

4 . ♟ . . ♙ . . .

3 . . ♘ . . ♕ . ♟

2 ♙ ♙ ♙ ♗ ♗ ♙ ♙ ♙

1 ♖ . . . ♔ . . ♖

a b c d e f g h

Side: white

Enpassant: no

Castling: KQkq

Depth 1: Best Move e2a6 with Score: 5

Depth 2: Best Move e2a6 with Score: 5

Final Best Move: e2a6 with Score: 5

Total Nodes Searched: **93044**

8 ♜ . . . ♚ . . ♜

7 ♟ . ♟ ♟ ♛ ♟ ♝ .

6 ♗ ♞ . . ♟ ♞ ♟ .

5 . . . ♙ ♘ . . .

4 . ♟ . . ♙ . . .

3 . . ♘ . . ♕ . ♟

2 ♙ ♙ ♙ ♗ . ♙ ♙ ♙

1 ♖ . . . ♔ . . ♖

a b c d e f g h

Side: black

Enpassant: no

Castling: KQkq

Comparying with same result on same position only using Negamax algorithm

Depth 1: Best Move e2a6 with Score: 5

Depth 2: Best Move e2a6 with Score: 5

Final Best Move: e2a6 with Score: 5

Total Nodes Searched: **101391707**

// Visualing not files and ranks

/\*

\* not H file

8 0 1 1 1 1 1 1 1

7 0 1 1 1 1 1 1 1

6 0 1 1 1 1 1 1 1

5 0 1 1 1 1 1 1 1

4 0 1 1 1 1 1 1 1

3 0 1 1 1 1 1 1 1

2 0 1 1 1 1 1 1 1

1 0 1 1 1 1 1 1 1

a b c d e f g h

not A file

8 1 1 1 1 1 1 1 0

7 1 1 1 1 1 1 1 0

6 1 1 1 1 1 1 1 0

5 1 1 1 1 1 1 1 0

4 1 1 1 1 1 1 1 0

3 1 1 1 1 1 1 1 0

2 1 1 1 1 1 1 1 0

1 1 1 1 1 1 1 1 0

a b c d e f g h

not HG file

8 1 1 1 1 1 1 0 0

7 1 1 1 1 1 1 0 0

6 1 1 1 1 1 1 0 0

5 1 1 1 1 1 1 0 0

4 1 1 1 1 1 1 0 0

3 1 1 1 1 1 1 0 0

2 1 1 1 1 1 1 0 0

1 1 1 1 1 1 1 0 0

a b c d e f g h

not AB file

8 0 0 1 1 1 1 1 1

7 0 0 1 1 1 1 1 1

6 0 0 1 1 1 1 1 1

5 0 0 1 1 1 1 1 1

4 0 0 1 1 1 1 1 1

3 0 0 1 1 1 1 1 1

2 0 0 1 1 1 1 1 1

1 0 0 1 1 1 1 1 1

a b c d e f g h

\*/

/\*

WHITE PIECES

Pawns Knights Bishops

8 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0

7 0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0

6 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0

5 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0

4 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0

3 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0

2 1 1 1 1 1 1 1 1 2 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0 1 0 0 1 0 0

a b c d e f g h a b c d e f g h a b c d e f g h

Rooks Queens King

8 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0

7 0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0

6 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0

5 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0

4 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0

3 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0

2 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0

1 1 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0

a b c d e f g h a b c d e f g h a b c d e f g h

BLACK PIECES

Pawns Knights Bishops

8 0 0 0 0 0 0 0 0 8 0 1 0 0 0 0 1 0 8 0 0 1 0 0 1 0 0

7 1 1 1 1 1 1 1 1 7 0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0

6 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0

5 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0

4 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0

3 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0

2 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0

a b c d e f g h a b c d e f g h a b c d e f g h

Rooks Queens King

8 1 0 0 0 0 0 0 1 8 0 0 0 1 0 0 0 0 8 0 0 0 0 1 0 0 0

7 0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0

6 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0

5 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0

4 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0

3 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0

2 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0

a b c d e f g h a b c d e f g h a b c d e f g h

OCCUPANCIES

White occupancy Black occupancy All occupancies

8 0 0 0 0 0 0 0 0 8 1 1 1 1 1 1 1 1 8 1 1 1 1 1 1 1 1

7 0 0 0 0 0 0 0 0 7 1 1 1 1 1 1 1 1 7 1 1 1 1 1 1 1 1

6 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0

5 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0

4 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0

3 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0

2 1 1 1 1 1 1 1 1 2 0 0 0 0 0 0 0 0 2 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1

ALL TOGETHER

8 ♜ ♞ ♝ ♛ ♚ ♝ ♞ ♜

7 ♟︎ ♟︎ ♟︎ ♟︎ ♟︎ ♟︎ ♟︎ ♟︎

6 . . . . . . . .

5 . . . . . . . .

4 . . . . . . . .

3 . . . . . . . .

2 ♙ ♙ ♙ ♙ ♙ ♙ ♙ ♙

1 ♖ ♘ ♗ ♕ ♔ ♗ ♘ ♖

a b c d e f g h

\*/

/\*

\* How to generate Relevant occupancy bit count for sliders

for(int rank = 0; rank < 8; rank++)

{

for(int file = 0; file < 8; file++)

{

int square = rank \* 8 + file; // Get attack mask for given square

Console.Write(Globals.CountBits(Slider piece attack mask(square)) + ", ");

}

Console.WriteLine();

}

It is possible to calculate it on the fly as well, but I think it is better to

have it hard coded, and use it like a look-up table.

\*/