Alpha beta pruning

and its place in computer game play

Kieran Sockalingam
University of Oxford

October 2015

Introduction

Alpha beta pruning is an improvement over naïve minimax, a recursive algorithm for determining the best choice for the next move in a game; usually a two-player zero sum game such as chess. I shall provide examples of my implementation and performance analysis of alpha beta pruning using a chess engine I wrote this summer which implemented minimax.

The goal of a chess engine is to deterministically select the next move to play when given the state of the board at the start of its turn. This engine achieves this goal by enumerating each possible move it can take, and evaluating in some way which is the best choice.

I shall outline the way in which the state of the game is represented in memory, the procedure used to select the best move to make, and the advantage of alpha beta pruning over minimax. I shall also discuss some other techniques which are used in conjunction with alpha beta pruning to make the engine as performant as possible.

State Representation

We need some way of representing the state of the game in memory. The representation should easily allow enumeration of the possible moves the engine can take. In chess, the state of the game consists of a board of 64 squares, and each square is in one of 13 states: empty, or filled with one of the 12 types of pieces. Each square thus fits into a byte with no issues, so the simplest way of representing the board is with a byte array. This takes 64 bytes of memory, which is conveniently exactly one cache line on an Intel Core i7 processor.

```
char data[8][8];
```

This representation is very convenient because it makes operations such as adding, removing or moving pieces conceptually simple, and it's easy to write functions to check if there exists a clear line of movement in a given direction for a given piece. For example, this code checks if there exists a clear horizontal line of movement between the given coordinates.

```
bool Board::existsClearHorizontalLine(int row, int columnS, int columnE)
{
        if (columnS == columnE) return isSpaceEmpty(row, columnS);
        int step = ((columnE - columnS) > 0) - ((columnE - columnS) < 0);
        int cp = columnS + step;
        while (cp != columnE) {
                if (!isSpaceEmpty(row, cp)) return false;
                cp += step;
        }
        return true;
}
```

However, it's not very efficient because checking if a piece can move to a square takes several memory accesses (although this isn't too bad as they won't be cache-misses) and involves traversing around the array checking several bytes, as well as using branches a lot. The following describes a much more efficient representation, which takes advantage of the coincidence that a chessboard has 64 squares and that the program shall be run on a 64 bit processor.

A chessboard can be represented by a collection of BITBOARDS. A bitboard is a single 64 bit word, where each bit represents one square on the board and can be used to store the locations of a single type of piece (Adel'son-Vel'skii et al, 1970). For example, a collection of seven bitboards, one each for pawns, castles, knights, bishops, queens, kings and white pieces represents a chessboard. Logical bitwise operations can then be used to get the location of pieces: e.g. black knights is `knights & ~white`. The locations that can be attacked by a given side can then be found by bitwise operations and dictionary lookups indexed by a bitboard. This technique is called MAGIC BITBOARDS (Kannan, 2007) and works by masking the bits which could affect a piece's movement, and then multiplying them by a MAGIC CONSTANT which gives a small value which can be used to index a table. The magic constants are calculated such that collisions are not a problem. This is orders of magnitude faster than the array representation, because the processor can perform bitwise operations on the entire board in one instruction, and each bitboard fits into a single physical register. I shall be implementing bitboards with the magic bitboard technique in the near future.

Speed is very important because the program will need to check if a given piece can move to a given location billions and billions of times before it makes a move.

The Evaluation Function

The EVALUATION FUNCTION is a heuristic for determining how advantageous a particular game state is for the engine. Chess is a zero sum game, which means that one player's advantage is the other's disadvantage, so the evaluation function can simply be maximised for the engine (the MAX-PLAYER) and minimised for the opponent (the MIN-PLAYER). There are many factors one can take account of when writing an evaluation function, such as the material value of the pieces remaining, the strategic value of their positions on the board, the number of available moves from this position (MOBILITY), whether friendly pieces are defended and whether the opponent's pieces are threatened.

The Search Tree

When the engine is selecting which move it should play, it needs to consider all the possible future eventualities that may occur, and pick the move that leads to the best set of future eventualities. The future eventualities can be organised into a tree, called the SEARCH TREE. The root of the tree is the current game state. The next layer contains all possible game states after the engine's move, and the next layer contains all possible states after the opponent's move, and so on. Given unlimited processing time and memory, the engine could generate the entire tree. (The tree will not be infinitely big because there is a maximum of fifty turns in a game). Then it can select its move such that every possible branch can lead to a successful checkmate, and the engine is guaranteed to win if it is possible to win.

However, the number of possible chess games is far too big for this strategy to ever be viable. It was estimated to be around `10^120` by Claude Shannon. (Shannon, 1950).

Instead, we can use the evaluation function as an estimate for how a game is going, and only search the tree to a desired depth of future moves. The faster the engine can search, the deeper it can search, resulting in better quality, more informed moves. The depth of the tree is measured in PLIES. One ply is one move by one player, so a turn of one move each is two plies. DEEP BLUE, the first chess computer engine to beat a reigning world champion (Deep Blue vs Garry Kasparov, 1997), searched ahead to at least 12 plies, but sometimes up to 20 depending on the situation.

After generating the tree to the desired depth, and running the evaluation function on each leaf to come up with a score for it, a naïve approach would be to select the next move that leads to the highest average score for the leaves to which it leads. However, this is suboptimal, because some branches are far more likely to be played than others, because the opponent is also trying to win, not just playing randomly. Furthermore, the engine will have control over half of the choices. This is solved by minimax, by predicting that the opponent will try to minimise the evaluation function.

Minimax

Minimax is an algorithm for assigning scores to all the nodes which are not leaves. It begins with the assumption that the engine (the max-player) will always move to maximise the score given by the evaluation function, and the opponent (the min-player) will always move to minimise it. For each node, we can recursively define its score to be the maximum of its children's scores if it is the max-player's turn, or minimum otherwise. The leaves are scored by the evaluation function.

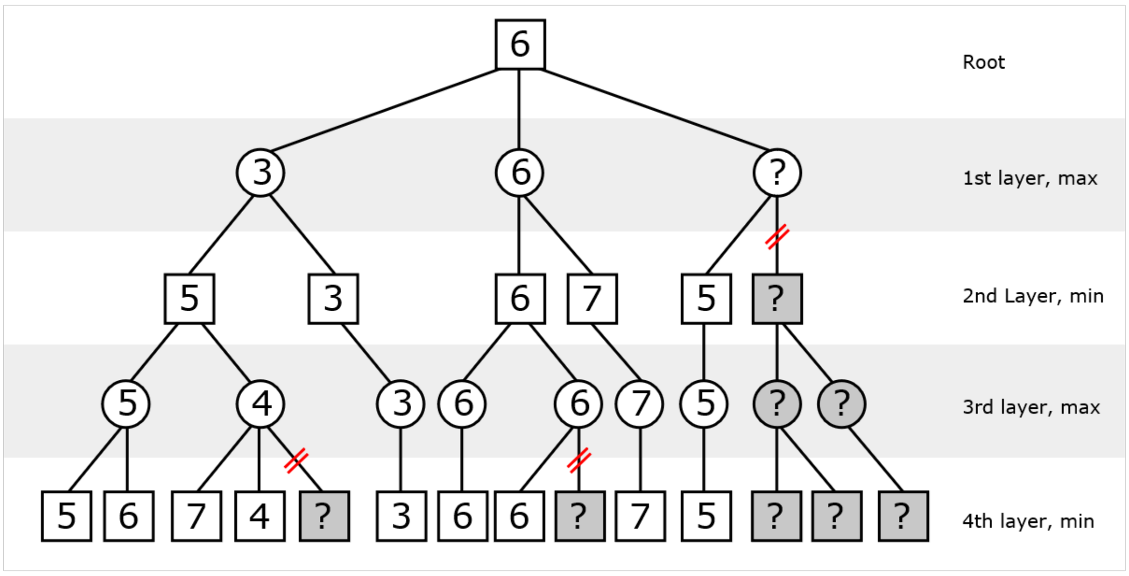
The engine then makes its move by selecting the child with the highest score.

```
/* Process a child using minimax
i       : the index of the child to search
depth   : the depth remaining to search
ourTurn : true if it is the engine's turn, false otherwise

minimax is a double which contains the score of the current node, acting as a "best
found" while the search is in progress.

minimaxBypass is a boolean which is true initially, merely ensuring that the first
child to be processed is selected as the current "best found", because the minimax
score is not initialised.

minimaxLine is a pointer to a Forecast object, which is used to build the
Principle Variation, which is the path through the tree containing the node at each
layer which was either the maximum or the minimum accordingly, and is used for
debugging and insight purposes.

For each child, this code calls generateForecast, which builds the next layer of
the tree and starts the minimax process on it. Thus, this function is recursive,
but via generateForecast. This separation of code is merely for clarity. After
building the tree, we check if the child is the new best found, and update
accordingly.
*/

void Forecast::ThreadProcessChild(int i, int depth, bool ourTurn)
{
        children[i]->generateForecast(depth - 1, !ourTurn, false, table, alpha, beta);
        if (minimaxBypass) {
                minimax = children[i]->minimax;
                minimaxLine = children[i];
                minimaxBypass = false;
        }
        if (!ourTurn) {
                if (children[i]->minimax < minimax) {
                        minimax = children[i]->minimax;
                        minimaxLine = children[i];
                }
        }
        else {
                if (children[i]->minimax > minimax) {
                        minimax = children[i]->minimax;
                        minimaxLine = children[i];
                }
        }
        return;
}
```

Alpha beta pruning

Alpha beta pruning is an improvement to minimax that does not affect its result, but reduces the amount of computation required to compute said result. The algorithm 'prunes' the tree, stopping the search for branches where further computation could not possibly affect the result.

Consider the following situation.



On the first layer, the engine is looking for the child with the maximum score. Having already scored the first two nodes from the left at 3 and 6 respectively, we start searching the last branch. On the second layer, we find a 5. This is a minimising layer, so the least child is chosen. Thus, the missing node on the first layer cannot be greater than 5. This means that the 6 node is chosen, regardless of further computation on the last branch. Thus we have saved time by not computing scores for all the nodes marked with a question mark. Alpha beta pruning works by breaking out of the search loop in these situations. It maintains two values, alpha and beta, which bound the search, and can get tighter after processing each child.

```
/* Process children using minimax with alpha beta pruning
depth   : the depth remaining to search
ourTurn : true if it is the engine's turn, false otherwise
alpha   : the alpha value
beta    : the beta value

minimax is a double which contains the score of the current node, acting as a "best
found" while the search is in progress.

minimaxBypass is a boolean which is true initially, merely ensuring that the first
child to be processed is selected as the current "best found", because the minimax
score is not initialised.

minimaxLine is a pointer to a Forecast object, which is used to build the
Principle Variation, which is the path through the tree containing the node at each
layer which was either the maximum or the minimum accordingly, and is used for
debugging and insight purposes.

Alpha and beta are values which represent the bounds on the search; that is, alpha
is the maximum value that the maximising player is guaranteed to get, and beta is
the minimum value that the minimising player is guaranteed to get.

For each child, this code calls generateForecast, which builds the next layer of
the tree and starts the alpha beta process on it. Thus, this function is recursive,
but via generateForecast. This separation of code is merely for clarity. After
building the tree, we check if the child is the new best found, and update
```

```cpp
accordingly. We also update the bounds. If the bounds do not allow for further
improvement, we stop searching the remaining children.
*/
void Forecast::AlphaBetaProcessChildren(int depth, bool ourTurn, double alpha, double beta)
{
	if (ourTurn) {
		for (int i = 0; i < children.size(); i++) {
			children[i]->generateForecast(depth - 1, !ourTurn, false, alpha, beta);
			if (minimaxBypass || children[i]->minimax > minimax) {
				minimaxBypass = false;
				minimax = children[i]->minimax;
				minimaxLine = children[i];
			}
			alpha = std::max(alpha, minimax);
			if (beta <= alpha) {
				break;
			}
		}
	}
	else {
		for (int i = 0; i < children.size(); i++) {
			children[i]->generateForecast(depth - 1, !ourTurn, false, alpha, beta);
			if (minimaxBypass || children[i]->minimax < minimax) {
				minimaxBypass = false;
				minimax = children[i]->minimax;
				minimaxLine = children[i];
			}
			beta = std::min(beta, minimax);
			if (beta <= alpha) {
				break;
			}
		}
	}
}
```

Transposition Tables

In a typical search tree, there are often several nodes which all represent copies of the same game state. These duplicates, referred to as TRANSPOSITIONS, arise because there are several different ways to get to these game states. For example, the following two games both leave the board in the same state:

- `White: e2->e4, Black: e7->e5, White: b1->c3`
- `White: b1->c3, Black: e7->e5, White: e2->e4`

Such nodes are the same, so further computation should not be performed several times on each one, but once and the results cached. This leads to a large speedup.

In order to cache the results, I used a transposition table. It is a large, bucketed hash table, with linked lists at each entry to store nodes in the bucket. When generating a forecast, the engine simply checks if the result already exists in the table, and returns it if so. If not, the result is added to the table when computation finishes. The hash function I used is ZOBRIST HASHING.

This worked well when the engine was using minimax without alpha beta pruning, because the result of a forecast was always the same. However, implementing alpha beta pruning broke the transposition table, because the result of a forecast depends on the bounds alpha and beta.

Alpha beta pruning works much more effectively if the children are considered in an order which is likely to result in the best child being chosen first, because the bounds are reached faster. Thus, the transposition table could be adapted to give a hint as to which child to search first, as opposed to the final result. This should result in performance almost as fast as having the result cached.

Zobrist Hashing

Zobrist Hashing is a hash function designed for computer gameplay of board games, and is very commonly used for indexing transposition tables.

Zobrist hashing works by creating a table of random bitstrings for each possible element in the game; i.e. each piece in each position. In order to make collisions unlikely, I used 64 bit words. There are 13 states for each square and 64 squares, so 13*64 random words are needed. To build the hash for a board, simply start with zero and for each square, XOR the relevant bitstring according to its state. Rather than generate the random bitstrings at runtime, I hardcoded them into the source code, so that hashes remain consistent across multiple runs of the program.

Multithreading

Searching the tree is a highly parallelizable workload, so it makes sense to create worker threads that can run simultaneously to take advantage of a multicore processor. Without alpha beta pruning, this is as simple as spawning a thread and calling the minimax algorithm in it for each child of the root node. Further recursive calls are not put into separate threads because the overhead of creating threads becomes a problem, and there is no advantage because the program will not be run on a processor with more than 20 logical cores.

After implementing alpha beta pruning, this became slightly more complex because the workload is not parallelizable. Each child must be considered in turn, tightening the bounds as we go. I settled on a compromise for this, because both alpha beta pruning and multithreading are huge improvements which must be used. I used a separate call to alpha beta, with fresh bounds, in a separate thread, for each child of the root, and then alpha beta as normal from there. This means there is effectively a separate instance of alpha beta for each root child, so some benefit of alpha beta is lost, but we get to use multithreading.

Performance

In order to benchmark how much of the tree was pruned off using alpha beta, I ran a test forecast from the starting position of a chessboard, at a depth of 5. The following table shows the number of nodes that were processed in the tree.

	Nodes Processed	Percentage of tree pruned
Minimax without transposition table	5 261 737	-
Minimax with transposition table	1 962 529	63%
Alpha beta without transposition table	630 454	88%

Conclusion

Alpha beta pruning was very effective, reducing the workload in my simple test case by as much as 9 times, and it is very easy to implement once minimax is in place. In order to further increase the effectiveness, I will work on the order in which children are selected for processing. At present, they are selected in an arbitrary order, but if they were prioritised in an order that is more likely to yield the best results first, alpha beta would result in more branches being pruned.

Bibliography

Adel'son-Vel'skii, G.M., V.L. Arlazarov, A.R. Bitman, A.A. Zhivotovskii, and A.V. Uskov (1970). Programming a computer to play chess. *Russ. Math. Surv.* 25:221.
Kannan, P. (2007). Magic Move-Bitboard Generation in Computer Chess. *pradu.us*
Shannon, C.E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine* 7:41:314