Computer Chess Programming Theory

Contents

Negamax Search

This is the fundamental structure around which the rest of chess tree searching algorithms are based. To put it simply, Negamax tree searching implements the idea that "The worse your opponent's best reply is, the better your move."

Implementing this is surprisingly easy. It uses the fact that chess is a symmetric game, and that therefore the analysis function must give symmetric scoring. That is to say that at any point, the score for white is exactly minus the score for black, or equivalently the sum of the two scores always equals zero.

This is quite straightforward to understand. If white is winning by one pawn, then clearly black is losing by the same amount. This principal can be extended to positional advantages, i.e. if white has doubled rooks on one file, then white has a bonus score, whereas black's position is weaker by the same amount because of this.

Negamax simply implements the following rough search steps;

```
Loop through all moves

  Play move

  move_score = - Opponent's_best_move_score

  if ( move_score > best_move_score ) then ( best_move = move )

  Undo Move

End of Loop
```

As you can see, this is a recursive algorithm. To calculate the score for the first player's move, you must calculate the best of his opponent's moves. For each of his opponent's moves, you must calculate the best score for the replies to those moves, and so on. Clearly we need to define some sort of cutoff depth, or else this will continue for ever. To do this, we implement the following slightly-refined version of the above code;

```
(Set depth initially to required value)

SEARCHING_FUNCTION {

  Decrease depth by 1

  Loop through all moves

    Play move

    if ( depth = 0 ) move_score = static_position_score
    else move_score = - Opponent's_best_move_score

    if ( move_score > best_move_score ) then ( best_move = move )

    Undo Move
```

```
  End of Loop

  Return best_move_score

} END
```

Here, the static position score routine simply returns a score for the current position based on various considerations. We can imagine a simple one which simply adds up the points values for all the current side's pieces, and subtracts the points values for the opponent's pieces. Clearly this will be very fast, but will play extremely bad chess because two positions can be very much better or worse for one player, but with the same material scores. As a simple example, imagine the endgame position where white has a king on h1 and a pawn on h2, black has a king on a3 and a pawn on a2. Clearly this is won for black, regardless of who is to play next. However, they both have exactly the same material.

Analysis Function

Modern chess engines use a complicated set of analysis functions depending on what material is on the board. I have several separate endgame analysis functions for certain types of endgame such as kings and pawns only. The main analysis routine, however, implements the following simple ideas;

Firstly, loop through the board, and make up an array of attack and defence values for each square. This is a list of how many times each square is attacked by an opponent piece or defended by a friendly one. Also keep a list of how many pawns are on each file for future reference.

Next, loop through the board again, but this time do a much deeper analysis. Award points for how much empty squares are attacked and defended. Analyse each piece on the board separately by considering factors which are important for that particular type of piece. For example, reward knights for occupying squares near the centre of the board, and reward kings for staying out of trouble in the corners. Also add on a material score for each piece you own on the scale of a pawn=100 points. In ColChess I used variable piece values depending on the stage of the game, i.e. bishops become more valuable in open endgames.

Finally, add on a load of other important general factors, for example reward pawn chains, bishop pairs, connected rooks and control of the board. Penalise doubled or tripled pawns and also blocked or 'hung' pieces (that is, pieces which are attacked and not defended). I do checkmate testing elsewhere, so that is not included in the static analysis function.

Specialised endgame functions have different weightings, for example passed pawns become much more dangerous when there are no opposing pieces around to stop them from promoting.

Iterative Deepening

Often in games the computer player will be forced to search for a certain time, rather than to a fixed depth. Iterative deepening allows it to do this, but has many other important bonuses connected with the transposition table. The theory is simply this;

Start at a shallow depth, say 2 ply. That means that you search each of your moves, and for each of them you find your opponent's best reply and immediately return that static score. Once you have searched the tree to depth 2 ply, then increase the depth to 3 and search again. Continue doing this until you've searched to the minimum required depth and either (a) you've run out of time, or (b) you have also reached the maximum allowed depth.

The first obvious advantage of this method is that you will always have some result to show from your search, and you know that it won't be a total mistake, at least to a few ply. Imagine a program which guesses the search depth to work to based on the time allocation and the board complexity, and then searches the first thirty of 31 available moves at a particular game stage. Then the program runs out of time and is forced to return the best move found so far. These 30 moves might all be complete blunders, immediately losing a piece or worse. However, the 31st move might be the only one to save his pieces, and win the game. This naive program never even considered it!

Using iterative deepening, you know that you have a good foundation to build on. A 2 ply search takes no time at all, usually a tiny fraction of a second on fast computers. Therefore you know that when you search to depth 3 ply, you already have a good idea of which move might end up being the best. Furthermore, you can use all of the information derived in previous shallower searches to speed up vastly the subsequent deeper searches. Counter-intuitively, searching all the shallower depths first normally *reduces* the time for the overall search compared to just starting at a deep level initially.

This can be seen easily with the following example. Imagine you are planning to search a position to depth 10, say, and you start off iterative deepening at 2 ply, then 3 ply etc... until you reach 6 ply where you spot that all but one of the possible moves loses to checkmate! Clearly there is no point fully searching this one remaining move to 10 ply as you have no choice but to play it! You simply quit the search there and return the only safe move. Of course this move may also lose to checkmate in more than 6 ply, but that's no worse than any of the other move options and it makes it less likely that your opponent has spotted the win.

Alpha-Beta Pruning

Alpha-beta pruning is a technique for enormously reducing the size of your game tree. Currently using the negamax algorithm we are searching every reply to every move in the tree. In the average chess position there are about 30 legal moves, and let's say for sake of argument that our program analyses 50,000 moves per second. Now, let's see how deep we can search.

Depth (ply)	Number of Positions	Time to Search
2	900	0.018 s
3	27,000	0.54 s
4	810,000	16.2 s
5	24,300,000	8 minutes
6	729,000,000	4 hours
7	21,870,000,000	5 days

You can see that the search tree quickly gets extremely large, and that's assuming a rather fast analysis function. Searching to 6 or 7 ply in the middlegame is vital for a good chess program playing blitz. The total time limit might be 5 minutes, perhaps giving 5-10 seconds per move. Using our current method we can't even search to ply 4. What we need to do is reduce the tree size enormously. This is where alpha-beta pruning comes in.

Alpha-beta pruning is just a complicated-sounding name for "Don't check moves which cannot possibly have an effect on the outcome of your search." The theory is essentially simple, but takes a bit of thought before you get used to it. Imagine the following scenario;

Your chess program is busy searching all its moves at the top level. So far it has searched the first six moves, and the best score has been 15 points. It starts searching the seventh move and considers its opponent's replies. Remember that your score is minus your opponent's best score. The program steps through all of its opponent's replies to this one move, getting the following scores;

```
-20,-22,-15,-16,-11,-18,-20,-30,-70,-75
```

Now the best amongst these is -11, which gives your program a score for its move of **-(-11)** which is 11. This is worse than the previous best move at this level so this particular move is ignored. But we have wasted a lot of time here searching the 6th to 10th of the opponent's replies. As soon as one of the opponent's replies scored -11, *we knew that this move of ours could not possibly beat the best score so far* of 15. Whatever the rest of the replies scored, the best reply was guaranteed to score *at least* -11, meaning that our move would score *at most* 11 and therefore that we would disregard it. So what we should have done here was to quit the search as soon as the score of -11 was discovered, and stop wasting our time.

This is the principle of alpha-beta pruning. Its implication to negamax search goes something like this;

```
initially alpha = -INFINITY, beta=INFINITY

search(position,side,depth,alpha,beta) {

  best_score = -INFINITY

  for each move {

    do_move(position, move)

      if ( depth is 0 )   move_score = static_score(position, side)
      else   move_score = - search(position, opponent side, depth-1, -beta, -
alpha)

    undo_move(position,move)

    if ( move_score > best_score )   best_score = move_score
    if ( best_score > alpha )   alpha = best_score
    if ( alpha >= beta )   return alpha
  }

  return best_score

}
```

Apologies for moving into pseudo-c but this algorithm requires that the correct variables are passed to the next level and it would be impossible to show it in any other way. I hope that all the symbols are obvious.

Whenever we are searching we check to see if the move we have is greater than alpha. If it is then we replace alpha by the new score. This way alpha keeps track of the best score so far.

If a move scores less than alpha then we're not interested in it because it's not good enough to worsen the score of the move to which it is a reply. If the score is between alpha and beta then it *will* reduce the score of the opponent's previous move, but not enough to make the same previous move bad. If a reply move scores equal to or greater than beta then this move is so good that the opponent's last move becomes bad, and the opponent would have never played it and let this situation arise, so we need search no more. We return this value of the best score immediately. This is called a fail-high cutoff.

Alpha-beta search effectively reduces the branching factor at each node from 30-40 to about 7 or 8 provided that you have a reasonably good initial move ordering routine. It is clearly advantageous to have the best moves near the top of the list so that all the other moves are much more likely to cause cutoffs early. Even basic maths can tell you that such a reduction in the branching factor will almost double the depth to which you can search in a fixed time. Alpha-beta search is used by all good chess programs. It doesn't change the outcome on a fixed depth search one bit as all the branches it prunes out are irrelevant and wouldn't have altered the move score. Moreover, with twice the search depth you can get vastly better moves and play much stronger chess.

Principal Variation Search

This is a further refinement on negamax search with alpha-beta cutoffs. The idea behind it is rather simple, but it is a little tricky to implement because of a few annoying subtleties. Basically, one expects to have a good move ordering function which takes the list of all legal moves at each position, and naively puts the moves which are likely to prove to be the best near the top. One usually considers captures first, followed by pawn pushes and checks, moves towards the centre, and then the rest. This tends to put the best move at or near the top of the list in the majority of cases.

So assuming that your preliminary move ordering is good, it is unnecessary to search the rest of the moves quite so thoroughly as the first. You don't need to know what they score exactly, you just need to make sure that they aren't better than the best move so far. To this end, we just search the first move properly, and then for each subsequent move we do a search with a narrow window of 1 point using the following code;

```
move_score = - search(position, opponent side, depth-1, -alpha-1, -alpha)
```

instead of ...

```
move_score = - search(position, opponent side, depth-1, -beta, -alpha)
```

If we were correct in our assumption, and the best move really was at the front of the list, then this search will return an approximate value much faster than doing a full search. However, if we were wrong then it will exit, hopefully quickly, with a fail high beta cutoff, indicating that this move will actually improve alpha, and that it therefore needs to be searched properly. (Remember that 'improving alpha' means that this move is going to beat the best score so far)

The expectation is that the gain in time caused by carrying out so many searches with narrow search bounds will vastly offset the penalty of occasionally having to search again after a cutoff. As with all alpha-beta methods, the better the preliminary move ordering, the more efficiently the pruning works, and the faster your program runs.

Aspiration Windows

The speed with which a search returns is clearly going to depend on the width of the initial alpha-beta window given to it. The narrower the search, the more nodes will fail with either alpha- or beta-cutoffs at some point, thus pruning the tree drastically. This is the point of PV search. However, we can use this to our advantage right from the root node.

Thanks to the process of iterative deepening, we are always lucky enough to have a guess as to what the next search will score. After we have searched the root position to depth 2 ply, we have a best

move and a score, and we can therefore make a preliminary guess that a search to 3 ply will return the same best move and a similar score. Now of course, sometimes this is not true, and in practice the score will be slightly different in almost every occasion. However, we could concede that, on average, the score from a search to depth 'n+1' won't be different from the search to depth 'n' by more than a certain amount, which we label the aspiration window. Most programs use a value around one third of a pawn score.

Now, given this little bit of guesswork, we can now help our engine to return from the search more quickly simply by calling the root node search with *reduced* alpha-beta bounds. Instead of using +/- infinity, which is a totally safe choice, we could use a window around the previous score returned, such as alpha = old_score - 30, beta = old_score + 30. That way, most of the time we would still return the correct score, and we expect it to lie within these bounds, but thanks to the reduced window size, we expect to return this value much more quickly.

Of course, there is a slight problem with this method, namely when the search value lies outside the given bounds. In this case we must re-search the position using the returned score as one bound, and either +infinity or -infinity as the other, depending on whether we failed high or low. We need to finetune the value of the search window size so that this happens rarely, but we must be careful not to make it so large that the benefit is no longer tangible.

Sadly, due to a feature known as 'search instability', many algorithms that we might use in our engine will cause the utterly counter-intuitive problem that the engine fails high (returns a score >= beta) and then fails low (returns a score <= alpha) on the re-search. This is a very annoying problem, and fortunately occurs only rarely. In these cases you must just resort to a full width search, with alpha and beta set to +/- infinity. The expectation is that the hash tables will be rather full, so hopefully this won't be *too* serious.

Transposition Table

The transposition table is a method of storing work which we have already done so that we don't have to do it again. If we search to depth 6, finding all but three of the moves we could play lead to loss by checkmate, then we should store this so that when we come to search to depth 7, we needn't bother searching these positions at all.

The transposition table in chess engines work by creating a 'hash-value' for the current board position based on a random table of large integers. The expectation is that each board position has a separate hash-code, or identification number if you like. In practice it is possible, though extremely unlikely, for two boards to have the same code so I generate two hash keys in different ways so that the probability of this becomes miniscule.

Before a search is done, the engine checks in the hashtable to see if the current board position has been searched before. If it has been searched before to at least the depth to which it is currently

being searched then there's no need to search it again. If it hasn't been searched sufficiently before, then a search proceeds as normal.

As soon as a search is completed then the hash table is updated with the new results, storing the board hash key (or code) and the score which the search returned. I also store the depth to which this position was searched, and the 'type' of the score. Lots of the time the position will be scored exactly, but due to alpha-beta pruning the score returned is often only a lower or an upper bound. Whereas these values aren't so useful as having an exact score, it is still a good thing to store them because if, for example, the upper bound on the score is worse than alpha then you needn't bother searching this position any deeper. Even though you don't know what it would score exactly, you know that it wouldn't be good enough to make a difference.

Killer Moves

Move ordering is all-important in chess programming. The nearer your best move is to the top of the move list, the quicker all the other moves will fail-high and the quicker the whole process will be.

Killer move heuristic is a good way of ensuring the moves which repeatedly perform well at a certain ply are searched before the rest. Often you will find that your opponent has one good move that you must stop him or her from playing. This move will become a killer move because most of the moves you can play lead to your opponent playing this one good move, and winning a piece or more.

It is therefore a good idea to keep track of the moves which repeatedly cause beta cutoffs at each level, and place them nearer the top of the move list, depending on how often they are played. If these killer moves are searched first then the pruning will be much better as the computer won't have to waste time searching lots of other (bad) moves first.

In ColChess I keep track of the top two killer moves, and the scores associated with them. Whenever a better move comes along then I displace the lower killer move, and place the better move in either first or second place depending on how much it scores. In Beowulf I abandoned the score idea as I thought that it didn't really help. I think I was right, but lots of these tweaks are very dependent on your particular program.

History Heuristic

This history heuristic is similar to the killer heuristic, but independent of the current depth. I keep a track of all the best moves which have been played, and then I keep a table size 64*64 indexed by 'from' square and 'to' square. All these values start off at 0 and are increased by one for each time the corresponding move is the best. Therefore the good moves score much more highly than the bad

moves, and I can add the value from this table onto the move's preliminary score before move ordering, creating a much more accurate ordering sequence.

I have also recently implemented a version of this which stores a different history table for each ply, but I haven't really noticed any great improvements from using this algorithm so it is only optional.

Quiescence Search

The problem with abruptly stopping a search at a fixed depth is something called the 'horizon effect'. It might be that you have just captured an opponent's pawn at depth 0, then you return that score being justifiably proud. However, if you had searched another ply deeper you would have seen that the opponent could recapture your queen!

To get round this problem, ColChess, and Beowulf (like all good chess programs) implement a method called 'quiescence searching'. As the name suggests, this involves searching past the terminal search nodes (depth of 0) and testing all the non-quiescent or 'violent' moves until the situation becomes calm. This enables programs to detect long capture sequences and calculate whether or not they are worth initiating.

There are several problems with this method which need to be overcome. Firstly, it could indeed cause an explosion in the size of the game tree if used unwisely. Beowulf uses a rather simple implementation of this algorithm. However, ColChess has three levels of quiescence searching, all with different limits to overcome this problem;

1. Full width quiescence search
2. Reduced width quiescence search
3. Capture search

Full width search is not much different to the original search at depth>0, generating all the possible available moves and testing to see which one is the best. Clearly this is slow so it is used only in exceptional circumstances. By default this is not performed at all, but with the use of quiescence width extensions it occasionally comes into play. Generally it is only used in extremely dangerous situations where the side to move is in check and may well lose out because of this.

Reduced width search is carried out at only a shallow quiescence depth and the exact length of this depends on how dangerous the position is. Generally it is only 0 or 1 ply, but occasionally 2 or 3 ply beneath the terminal leaf nodes (depth = 0). In this search regime I consider only a subset of all the available moves, namely those moves which have potential to drastically change the current static score. Those are captures, checks and pawn advances. If ColChess comes to do a reduced-width quiescence search at a particular node and it finds that it is in check then it does a full-width search instead.

A **capture search** is performed after the maximum specified reduced-width search. Just as the name suggests, this involves testing only the capture moves at every node. Because this is much faster, and very important to do, I let this search continue until each branch is quiescent and there are no more captures possible. That is to say this is an infinite depth search, though of course there are only a finite number of pieces to be captured!

Of course there is another very important problem with quiescence searching. It is often more advisable for a player *not* to initiate a capture line because that player will lose out in the long run by doing so. In this case, at every node I give the computer the option of 'not moving' and just accepting the static move evaluation at that point. If this turns out to be better than any of the capturing move options then it will return that value. Often the best move is a quiescent one and forcing the computer to make a violent move might severely worsen its position.

The only exception to this is of course when I do a full-width quiescence search. Because this involves considering all of the possible moves, I do not allow the computer the possibility of accepting the current static score. This means that I can catch slightly deeper checkmates if they are forced because the quiescence search considers extremely narrow, but dangerous lines.

Quiescence Width Extensions

I have already mentioned this method in a small amount of detail, but it is worth clarifying. This is simply a way of improving the accuracy of the quiescence search when you think that it is necessary. Clearly there is little point in performing a full width search in all conditions, but if one of the kings is under heavy attack, it might well be worth it.

I keep track of a variable at each ply which tallies up the total number of extensions so far. This is set to zero initially. Any form of check at any point in the move analysis sets it to one. If at any point one of the sides has only one legal move in any position then the extension value is increased (to a maximum of 2 ply) and similarly if a side is in check and avoids it by moving its king. I always test positions that are in the principal variation with a depth of 3 ply.

This algorithm, amongst others, helped ColChess to make two large jumps in test score results between versions 5.4, 5.5 and 5.6.

Bitboards

Most top chess programs these days use bitboards. Bitboards are a wonderful method of speeding up various operations on chessboards by representing the board by a set of 64 bit numbers. For those of you who understand binary notation and bitwise operations then skip the next few paragraphs. Beowulf has an advanced rotated bitboard move generator. More about that later.

Binary notation is a simple way of representing numbers using only the digits 0 and 1. In conventional decimal notation, you write numbers like 45,386 where each place represents a certain number of 10's or 100's or units etc... Binary representation instead uses a 'base' of 2 instead of 10. That means that instead of representing the number of 1's, 10's, 100's etc... we keep count of the number of 1's, 2's, 4's, 8's and so on, doubling each time. A little bit of thought shows that it is possible to represent each number uniquely using this method, and that every whole number can be represented this way.

For example, the number 25 can be converted into binary by taking the largest power of 2 which is smaller than or equal to it, keeping a record that this number is included, and then continuing with the remainder after the largest power of two has been subtracted. With the number 25, the largest power of two not greater than it is 16 (2*2*2*2). We keep track of this, and then look at the remainder, 25-16 = 9. The largest power of two not larger than 9 is 8 (2*2*2). We count this too. However, we are now left with only 9-8=1. This is smaller than the next two smallest powers of 2 (4 and 2), so we place two zeros there. All we are left with is 1, leaving us with the final answer 25d = 11001b, using the subscripts 'd' and 'b' for decimal and binary respectively. Each of these '0's and '1's is called a 'bit'. Hence a 5 bit number can be anything from 00000b to 11111b, or 0 to 31 in decimal.

Bitwise operators act on numbers in binary notation. The simplest one is NOT, which does exactly what it says. For each '1', it replaces it with a '0', and vice versa. Hence, NOT(1b) = 0b, NOT(10b) = 01b. We can also define operators that act on two numbers and produce a third, rather like plus and minus. One of these is AND, where 'x AND y' returns a number which has a '1' in every place where there is a '1' in both x and y, and '0's elsewhere. Hence, 11001b AND 10010b = 10000b. 'OR' is similar, where it simply places a '0' if there is a '0' at that location in both input numbers, or a '1' otherwise. Hence, 11001b OR 11010b = 11011b.

Bitboard notation uses these above ideas in a rather clever way. Supposing, for example, we generate a 64 bit number which represents the pawn structure on the chess board. We do this by starting at a8, and moving across then down in the sense a8,b8,c8,...,h8,a7,b7,c7,...,g1,h1. For each square, we put a '1' if there is a pawn, or a '0' otherwise. Simple. Then we have just one number on which we can perform all sorts of useful operations.

An example of the kind of operation we might want to perform is this;

Imagine we have a bitboard of all white pieces, one of all black pieces, and one of all white pawns. How do we generate all possible white pawn moves? In conventional notation, we cycle through the board, pick out each pawn, and then evaluate the moves directly. In bitboard notation we can do much better. Firstly we take the pawn bitboard and shift it right 8 bits. Right-shift does just what it says, moving the bits 8 places to the right, and filling the 8 most significant bits on the left with zeros. Effectively, this is the same as a divide by 2^8, or 256. What this gives us is a list of all the places on the board that are exactly one square in front of a white pawn.

Next we need to make sure that we don't push a pawn into another piece. To do this we simply take

an AND of this shifted board with another board consisting of all the empty squares on the board. Either this is stored separately, or it is simply equal to ~(WHITEPIECES | BLACKPIECES), where the tilde sign represents bitwise NOT, and the | sign represents bitwise OR. We can simply add in all initial 2 square pushes by taking all pawns on Rank 2 (a2-h2), and then shifting them right 16 (two full ranks), and then checking that both the target square and the intermediate 'step-over' square are empty. The command (WHITEPAWNS & (255<<48)) gives us all of white's unmoved pawns. '<<' means 'shift left', 255 is a full row of 8 1's, and thus 255 << 48 gives us a row of 1's between a2 and h2. We can now get the legal moves by taking the AND of these pawns with ~(ALLPIECES << 8) and also ~(ALLPIECES << 16). The final target squares are the result of this final board >> 16.

Hopefully this approximately makes sense. If not then write out the boards as they would appear in this bit notation. For example, we can easily write out the opening position in bitwise format like so;

```
ALLPIECES;
11111111
11111111
00000000
00000000
00000000
00000000
11111111
11111111

WHITEPAWNS;
00000000
00000000
00000000
00000000
00000000
00000000
11111111
00000000
```

etc....

Here I have formatted the 64 bit number in to 8 rows of 8 bits each, like a chessboard.

Moves for **knights and kings** are also easy to calculate. Simply generate an array of bitboards which, for each square in the board, give the available knight and king moves. For example, KnightMoves[c2] might look like this;

```
00000000
00000000
00000000
00000000
01010000
10001000
00000000
10001000
```

It is then a simple matter of taking each knight and king, and reading off the value for KnightMoves or KingMoves at that point, and then removing those which involve an illegal capture of a friendly piece. You know how to do this by now - it's just (for our example above) KnightMoves[c2] & ~WhitePieces, assuming that this knight on c2 is a white knight. Note that so far we've not considered whether or not the moves are actually legal or whether they leave us in check. This is

time consuming, so we do a 'lazy evaluation' on this. That means we work it out only when we need to.

Sliding Pieces are handled rather differently. They use a technique known as rotated bitboards. Here, by using a lot of precalculation and a bit of mind-bending transformations, we can actually generate sliding moves rather quickly. We do this by calculating occupancy numbers for ranks, files and diagonals. For the rank, this is simple, and it means the following;

Take the rank, and then consider all the pieces (of any colour) on that rank. Now, ignoring the rest of the board, and just considering these 8 squares, convert this into a decimal number. In bitboard notation, what we've effectively done is;

```
occupancy = (AllPieces >> (Rank*8)) & 255;
```

where here 'Rank' is the rank in bitboard notation. We started at a8 = square 0 and worked across first, so the rank is 0 for the top row (a8-h8) and 1 for a7-h7 etc.. up to 7 for a1-h1. This is simply equal to the value of the square number divided by 8 and rounded down.

Now here is where the precalculation comes in handy. We have already generated a large array indexed by square number (0-63) and occupancy number for the rank on which this square resides (0-255). For each entry, we precalculate the bitboard representing the squares to which a horizontally sliding piece can move with the given occupancy. We simply look up this bitboard and we instantly have a bitboard representing all the horizontal targets this sliding piece has. We then filter out illegal captures of friendly pieces with a simple AND, as before.

For vertical (file) moves, we do exactly the same, but with a board which we have stored separately which stores the position of all pieces rotated through 90 degrees. This way we can simply read off the occupancy number of the file under question by reading off the occupancy of a *rank* in this 'rotated bitboard'. Simple, huh?

Now diagonal sliders are a lot more tricky, but effectively use the same principle as above. We store a bitboard 'rotated' through 45 degress either clockwise or anti-clockwise. We treat the two diagonal directions separately (that is diagonals in the direction a1-h8 and those in the direction a8-h1). We also have to store the length of the diagonals under question, and the amount we have to shift the board to get that diagonal at the front. For example, the a1h8 bitboard might look like this;

```
a8,a7,b8,a6,b7,c8,a5,b6,
c7,d8,a4,b5,c6,d7,e8,a3,
b4,c5,d6,e7,f8,a2,b3,c4,
d5,e6,f7,g8,a1,b2,c3,d4,
e5,f6,g7,h8,b1,c2,d3,e4,
f5,g6,h7,c1,d2,e3,f4,g5,
h6,d1,e2,f3,g4,h5,e1,f2,
g3,h4,f1,g2,h3,g1,h2,h1
```

which may not look like much, but if you actually write it out slightly differently, then you should immediately see the point;

```
a8,
a7,b8,
a6,b7,c8,
a5,b6,c7,d8,
a4,b5,c6,d7,e8,
a3,b4,c5,d6,e7,f8,
a2,b3,c4,d5,e6,f7,g8,
a1,b2,c3,d4,e5,f6,g7,h8,
b1,c2,d3,e4,f5,g6,h7,
c1,d2,e3,f4,g5,h6,
d1,e2,f3,g4,h5,
e1,f2,g3,h4,
f1,g2,h3,
g1,h2,
h1
```

Now you can see that all we have is a bitboard with the diagonals arranged sequentially. Now, for example, if we want the occupancy number for the diagonal in the a1h8 sense starting on square b4, we simply do the following;

1. Calculate the necessary shift, in this case 15.
2. Shift the board 15 squares right so that the top row now starts a3,b4,c5,d6,e7,f8
3. Look up the length of this diagonal, in this case 6 squares.
4. Take the bitwise AND with a string of 6 1's, or 63 (=111111b) to get the occupancy.
5. Now lookup the target bitboard using the square b4 and the occupancy obtained.
6. You now have a list of possible target squares.
7. AND your list of target squares with ~WhitePieces to get the final list

Not exactly simple, but hopefully understandable. The a8h1 diagonals are exactly the same, but with a bitboard rotated the other way. Now do you begin to see the power of bitboards? These rotated bitboards can be calculated when they are needed, but this is rather slow. It is far simpler just to update them whenever a move is played, just as you update the standard board.

Once we have a bitboard with all the possible destination squares set to '1', we now have to cycle through them one by one. To do this, we require two routines. Firstly we require a routine which locates the position of the first bit in the bitboard which is set, and secondly we need a simple macro which then sets this bit to zero. This was we just run some code like the following;

```
while (moves) {
  target = GetFirst(moves);
  ...
  Do required stuff with the target square
  ...
  RemoveBit(moves,target);
}
```

Internal Iterative Deepening

The way to speed up a chess program very rarely lies with extremely fast move generation, nor does it lie with rapid check testing. The best way to achieve immediate and noticeable speed increases is to improve the move ordering.

Alpha-beta pruning works most efficiently when the best move is searched first. The better the first move, the quicker the search will terminate with the subsequent moves because it becomes obvious that they are inferior. There is a minimal tree size that can be achieved with just a-b pruning. This was determined by Knuth & Moore (1975). That is equal to;

$$B^{[D/2]} + B^{[D/2]} - 1$$

where B is the branching factor, and D is the depth. This doesn't include the quiescence search, nor does it include other (large) gains that can be obtained by (theoretically unsound) techniques like null move pruning and extended futility pruning.

However, to obtain this optimal tree size it is necessary to search the best move first in each case. Clearly the closer your move ordering gets to optimal the smaller your tree size becomes. Internal iterative deepening is a method which can be used in conjunction with transposition tables, history heuristic and killer moves in order to improve your move ordering significantly. This method was used in Beowulf to great effect.

The theory is the following. If you are searching a PV node, that is you are not currently doing a NULL move, and you haven't got a best move to test from the hashtable, then just do a shallow search from the current position at a depth of (usually) 2 ply shallower. The best move returned from this search can then be used as a hashmove and should be searched first. Of course, this makes it much more likely that the best move will be at the front of the move ordering list. The extra time spent in searching for a move at a reduced depth is negligible compared to the gain in speed you can achieve from ordering the best move higher at a much more important ply in the search tree.

In general, IID is avoided if we are NULL moving, and it is also avoided when we are at a pre-pre-frontier node or shallower. That is to say, we are within two ply of the quiescence search.

Null Move Heuristic

As mentioned above in the section on Internal iterative Deepening, there is a minimum search tree that can be obtained simply by using conventional alpha-beta pruning methods. However, this limit is not an absolute limit, and can be avoided if one is willing to make one or two sacrifices in search accuracy. One method used to great effect in all strong modern programs is that of NULL move pruning. ColChess does not use this method, but Beowulf does.

Null move pruning is a clever, and relatively recent method first proposed by Donninger (1993). The algorithm is simple, and in fact simply codes a concept that humans have been using for many years without knowing it. Basically, the theory is the following; If you opt NOT to play a move ("pass"), letting your opponent play two sequential moves, and his score is *still* not any good, then clearly your position is very strong. In programming terminology, if the side to play opts not to move and the score returned by a search at a reduced depth for the opponent is lower than alpha (or the score for the side to play is greater than beta) then simply return this value instead of bothering to search the branch properly. It is clear that this node is so strong that it would never be allowed to occur. Effectively all we have done is just predicted a beta cutoff without actually searching the entire tree.

This method is extremely dangerous however, and if used carelessly it is capable of causing a great number of tactical errors. Most importantly, the problem of *zugzwang* positions. This is a German term meaning that the first person to play in a certain position loses. It reminds me of the old chess excuse, "I had all my pieces exactly where I wanted them, but sadly it was my turn to move!".

NULL move pruning must not be used when there is a chance of zugzwang. There are several ways to avoid this. Firstly, it is never used when the side to play has only a few pieces left. Secondly, it is never used when the side to play is in danger of losing to checkmate in the near future. This can be checked for and stored in the hashtable if necessary. Also, NULL move pruning should never be used when the side to play is in check, as this would obviously lead to an illegal position. Further restrictions are very much up to the individual programmer, and can affect greatly the tactical soundness of the algorithm as well as the degree to which search trees can be reduced in size.

Another parameter which can be altered is the depth to which the search is reduced when the NULL move tree is searched. Generally this depth is between 1 and 3 ply. Many have found that an 'adaptive null move depth reduction' algorithm is preferable (e.g. Heinz et al). This involves reducing the depth by different amounts depending on how deep the search is initially. For example, the depth might only be reduced by one ply near the frontier nodes whereas the depth might be reduced by 3 ply when we are near the root and searching to 10 ply. Again, this is open to a great deal of fine tuning.

Futility Pruning

Futility pruning was pioneered by Heinz et al with their program Dark Thought. More information can be found here. The general concept is quite simple. Is involves frontier nodes, where the search has one ply left before the quiescence search. The idea is this: If the current side to move is not in check, the current move about to be searched is not a capture and not a checking move, and the current positional score plus a certain margin (generally the score of a minor piece) would not improve alpha, then the current node is poor, and the last ply of searching can be aborted. We might as well go straight into the quiescence search, or otherwise just return the current score.

Of course, this is theoretically unsound, and might cause the program to overlook a great number of strong tactical lines. The expectation is that the speed increase gained by using this algorithm will make up for this. After all, if you can search one ply deeper then you effectively avoid all these problems, plus you gain the additional tactical insight that the extra ply gives you.

There is, you will be glad to know, a theoretically sound version of this algorithm, also pioneered by Heinz and collaborators. If we replace the deficit margin above with a different margin representing the maximum possible positional score improvement, then we cannot possibly miss important tactical lines. Effectively, what we are saying there is; "If there is no possible way that we can improve alpha with this move, then don't even bother looking at it." Obviously this prunes far fewer nodes than the above method, but is theoretically sound, and therefore not prone to overlooking well-disguised mistakes.

Extended futility pruning is a very similar technique which is applied at the ply before full futility pruning, that is the "pre-frontier" nodes. The technique is the same, but usually a larger score margin is taken. For example, the value of one rook or more. If the current score plus this *futility margin* is still not greater than alpha then we simply reduce the depth by one ply.

Razoring

This is another technique to use guesswork to prune extra nodes out of the search tree. Very similar in nature to futility pruning, this method involves pruning at pre-pre-frontier nodes. This means that we're two ply away from the quiescence search. The theory is pretty much the same as before.

If the current static positional score, plus this margin is not greater than alpha, then we reduce the depth by one ply and continue the search. The expectation is that this is not an interesting node. In this case, the futility margin must be pretty large to avoid mistakes. Generally the value of one queen is sufficient to avoid errors. However, as you can see from the progression from futility pruning through extended futility pruning to razoring, the greater the depth of the pruning the less theoretically sound these techniques become, and the more prone to mistakes they are.

Search Extensions

Search extensions are vital to any good quality chess program. However, when implemented badly they can be detrimental to the overall performance, causing your search tree to increase in size enormously with few advantages. One of the great problems in computer chess is how to search the bad moves to a shallower depth and the good moves more deeply. Of course, it is very difficult to predict the former. No-one knows which moves will turn out to be good before you have played them - you can only guess and hope that you don't overlook something.

However, searching the good moves to a greater depth is altogether more simply. You can't tell if a move is going to be good, but you can certainly tell if it looks as if it might become interesting. This is the whole principle of search extensions.

In a standard tree search, the recursive search procedure is called until the maximum depth is reached then we look at the quiescence search. However, for certain nodes, the quiescence search will not find the tactical gain, and we would be much better off just extending the main search a little more so that we can see if these moves develop into something interesting.

Common extensions are shown below. Remember that there are many more, and the exact way in which different people implement extensions varies greatly. One important addition is the idea of **fractional ply extensions**, which involves storing the depth in multiples of some large number, say 16. In this scheme, a depth increase of 16 is equal to one ply. This way, if a search extension is not quite interesting enough to merit an entire extra ply extension then you could perhaps extend the depth by 8, so that when two of these extensions happen in the same branch we get an extra 'whole' ply.

Check Extensions

- If the side to play is in check then extend the search by one ply.
- This improves tactical strength dramatically.
- Checkmate detection is drastically improved.
- Amy also extends slightly more on double checks and revealed checks

Recapture Extensions

- These occur when one side plays a capture and then the opposition recaptures with an identical value piece.
- This move is pretty much forced, so we search deeper.
- Some programs, such as Amy, extend the depth different amounts depending on the piece captured.

Singular Reply Extensions

- If there is only one legal move in a certain position then extend.
- The extra cost is minimal as the branching factor at this node is 1.
- Can be used in addition to check extensions to find checkmates very quickly.

Threat Extensions

- These are used with NULL move search
- If NULL move search returns CM against, then there is a threat nearby.
- Extend the search so that we can find the best way to avoid it.

Pawn Push Extensions

- If a pawn is pushed to the 7th rank, or promoted, then extend.
- This helps spot winning tactics in the endgame.
- Not so useful if your endgame analysis function is good enough at handling passed pawns.

Other ideas.

- Extend if the king safety drops suddenly.
- Extend if the search score changes drastically without a capture.
- Extend if defences are broken down by a sacrifice.

All of theses ideas are used to great effect in many professional programs. Fractional ply extensions help also. If you have any other ideas then I would be glad to include them here. Feel free to email me at the address below.

Static Evaluation

Of course, no matter how good your search is, you must also have a function which accurately evaluates how good any board position is for each side. There is no point searching to 15 ply depth if you just actively search out positions with terrible pawn structure, or weak king safety.

The static evaluation function is normally called from the quiescence search as soon as a position is *quiescent*, that is to say, quiet. This means that there are no outstanding captures left to resolve, neither king is in check, no pawns look as if they are about to promote, and numerous other terms which vary greatly between programs.

The most basic board analysis is simply to sum the points each side possesses on the board. In general we use the scale where one pawn is equal to 100 points, and then a knight or bishop is worth around 300 points, a rook 500 and a queen 900. These scores are very open to tweaking. Indeed, many programs increase the scores for the pieces slightly. Using this method means that the finest score difference possible is one centipawn or one hundredth of a pawn score.

However, it is obvious that just adding together material scores will not produce a great program. Many more factors need to be considered. Often two players might have identical material, but one player will have an enormous advantage in development or pawn structure, which leads to a simple win. You will certainly need to differentiate between positions which are slightly better for one side and positions which offer a clear advantage for that side. You might want to penalise doubled pawns (and indeed should do so) because they act as a weakness, especially in the endgame where they are difficult to undouble. You might consider that two pawns doubled and blocked on one file might not be worth 200 points, but rather only 150 points. Countless other such considerations should be added, and the weightings tweaked until they produce sensible results.

I have put together a small list of some of the more important terms to be considered in a static evaluation function, though these are little more than the tip of the iceberg. This is an enormous chunk of any chess program and deserves a proportional amount of effort.

Pawn Structure

- Penalise doubled, backward and blocked pawns.
- Encourage pawn advancement where adequately defended.
- Encourage control of the centre of the board.

Piece Placement

- Encourage knights to occupy the centre of the board.
- Encourage bishops to occupy principal diagonals.
- Encourage queens and rooks to defend each other and attack.
- Encourage 7th rank attacks for rooks.

Passed Pawns

- These deserve a special treatment as they are so important.
- Check for safety from opposing king and enemy pieces.
- Test pawn structure for weaknesses, such as hidden passed pawns.
- Add enormous incentives for passed pawns near promotion.

King Safety

- Encourage the king to stay to the corner in the middlegame.
- Try to retain an effective pawn shield.
- Try to stop enemy pieces from getting near to the king.
