

Recherche arborescente

Bruno Bouzy

16 novembre 2005

Introduction

Ce document présente la partie « minimax » ou « recherche arborescente » du cours de programmation des jeux de réflexion. Il reprend largement le tutorial de Bruce Moreland trouvé en 2002 sur le Web. « minimax » est à prendre au sens large : minimax au sens strict, alfabéta, alfabéta et ses principales « améliorations » : les tables de transposition, iterative deepening, MTD(f), etc. [Campbell & Marsland 1983] et [Knuth & Moore 1975] sont deux bonnes références sur le sujet alfabéta, mais elles datent un peu pour les dernières améliorations. [Junghanns 1998] est une autre référence générale. Le plan provisoire de ce document est le suivant :

Min-Max

- Alpha-Beta
- Tables de transposition (TT)
- Iterative Deepening (ID)
- MTD(f)
- Principal Variation Search (PVS)
- Null-Move
- Quiescence Search

Minimax Search

In chess, both players know where the pieces are, they alternate moves, and they are free to make any legal move. The object of the game is to checkmate the other player, to avoid being checkmated, or to achieve a draw if that's the best thing given the circumstances. A chess program selects moves via use of a *search function*. A search function is a function that is passed information about the game, and tries to find the best move for side that the program is playing. An obvious sort of search function to use is a *tree-searching function*.

The position that is on the board now is the root position or root node. Positions that can be reached in one move from the root position are reached by branches from the root position. These positions are called successor positions or successor nodes. Each of these successor positions has a series of branches emanating from it, each of which represents a legal move from that position.

The chess tree is very bushy (usually about 35 branches from each position), and very deep. It would be possible to represent every conceivable chess game in a giant tree, then perform a tree-search that finds the game that would be produced if both sides were to play the best possible moves. An obvious algorithm to use to do this is called *min-max search*. It is possible to use min-max search to solve (completely understand) simple games like tic-tac-toe. The

game-tree in tic-tac-toe is not very bushy and it's not very deep, so the entire tree can be traversed, the game can be completely understood, and a move can be made in any position that is guaranteed to be the best move. It is possible to do this with chess in a mathematical sense, but it is not possible to do on any computer that is likely to exist now or even in the far future.

Even so, it is still possible to use min-max search as the basis of a program that plays chess. Rather than min-maxing the entire chess tree, it is very possible to min-max the first few moves. Since the positions at the leaves of the tree aren't necessarily searched all the way out to mate or forced draw, a heuristic function, traditionally called `Evaluate()`, is used to assign values to these positions. These values are guesses, although chess programmers hope that they are educated guesses.

How min-max works

Min-max is a *recursively defined* function. A call to min-max calls min-max once for each successor position, and returns the maximum value returned from any of the sub-calls, from the point of view of the side to move. Here is the algorithm:

```
int MinMax(int depth) {
    int best = -INFINITY;
    if (depth <= 0) return Evaluate();
    GenerateLegalMoves();
    while (MovesLeft()) {
        MakeNextMove();
        val = -MinMax(depth - 1);
        UnmakeMove();
        if (val > best) best = val;
    }
    return best;
}
```

The code is called something like this:

```
val = MinMax(5);
```

This will return the value of the position, given 5-plies of look-ahead. `Evaluate()` returns scores from the point of view of the side to move, so if white is to move and is up a knight, the score will be something like +3 pawns, but if black is to move, the score turned from `Evaluate()` will be -3 pawns.

The function negates its return value in order to reflect the change and perspective that results when the side to move is changed. The function bottoms out by decrementing a depth value. When the value reaches zero, it isn't possible to do any more plies of look-ahead, so the function returns the static evaluation. The function works very intuitively. Essentially, it says return the value of the best move. The best move is defined as the move that leads to the highest score. Each move is evaluated by calling min-max recursively, where once again the function returns the value of the best move, from the opposite perspective.

Alfa-Beta Search

The problem with min-max

Alpha-beta is very similar to min-max, and in fact, there is really only one extra statement. Min-max works by examining the entire game tree, and picking the best path that can be forced. This is very easy to conceptualize, but it is not efficient. Each time an extra ply of depth is searched, the size of the tree searched grows exponentially. Chess tends to have about 35 legal moves in any given position. So if min-max is used to search to one ply of depth, about 35 positions are examined. If the function is used to search two plies, about 35^2 positions are searched. That's about a thousand. This doesn't seem that bad so far, but the number gets enormous very quickly. A six-ply search is almost two billion positions, for example, and a ten-ply search is over two quadrillion. It is important to search as deeply as possible, if the goal is to create a strong chess player by examining the first few plies of the game tree and applying a heuristic evaluation at the leaf nodes. Min-max doesn't allow for a very deep search, because the effective branching factor is extremely high.

The bag example

Fortunately, there is a way to reduce the branching factor. Furthermore, it's *perfectly safe*, and in fact there is absolutely no downside, it's a pure win. It relies on the idea that if you already have a choice you know is not bad, when you are looking at other choices, and you find one that you know is no better, you don't have to go to the trouble of figuring out exactly how bad it is.

I'll try an example. Let's say that your worst enemy has a bunch of bags sitting in front of him. The bags are there because he lost a bet, and now he has to give you something. Each bag contains a few things. You are going to get one of the things. You get to pick which bag the thing will come out of, but he gets to pick what you get out of that bag. You are going to look through one bag at a time, and you are going to look through the items in each bag one item at a time. Obviously, what is going to happen is that your enemy is going to give you the worst thing in the bag you choose, so your goal is to pick the bag that has the nicest worst thing. It's easy to see how you'd apply min-max to this problem. You are the maximizing player, and you are going to take the best bag. Your enemy is the minimizing player. He's going to try to make sure that the best bag is as bad as possible, by giving you the worst thing out of it. All you have to do to use min-max is take the bag that has the best worst thing in it, if that makes any sense. The problem with min-max is that you have to look at everything in every bag, which takes a lot of time.

But how can we do this more efficiently than min-max? Let's start with the first bag, and look at everything, and score the bag. Let's say that this bag contained a peanut butter sandwich and the keys to a new car. You figure that the sandwich is worth less, so if you take this bag you will get a sandwich. The fact that there are also car keys in the bag doesn't matter, as long as we assume that your opponent can also evaluate items correctly, which we will. Now you start in on the next bag. The process you use is a little different than min-max this time. You look at items one at a time, and compare them against the best thing you know you can get (the sandwich). As long as items are better than the sandwich, you handle this as you do in min-max -- you just keep track of the worst one -- since maybe the worst one is better than the sandwich,

which would mean that you'll take this bag over the one with the sandwich. But if you find another sandwich in this bag, or something you think is worse than the sandwich, you discard this bag without looking at any more items. The reason is that you know that if you take this bag, the absolute best you can do is no better than the sandwich (and might be worse). Let's say that the first item in the bag is a twenty-dollar bill, which is better than the sandwich. If everything else in the bag is no worse than this, that's the item that your opponent will be forced to give you if you take this bag, and this becomes your bag of choice. The next thing in the bag is a six-pack of pop. You think this is better than the sandwich, but it's worse than the twenty, so this bag is still your bag of choice. The next thing is a rotten fish. This is worse than the sandwich. You say no thanks, hand the bag back, and forget about this bag. It doesn't matter what else is in the bag. There could be the keys to another car, which doesn't matter, since you are going to get the fish. There could be something much worse than the fish (I leave this to your imagination). This doesn't matter either, since the fish is already bad enough, and you know you can do better by taking the bag with the sandwich.

The algorithm

Alpha-beta works just like this. The idea is that two scores are passed around in the search. The first one is *alpha*, which is *the best score that can be forced by some means*. Anything worth less than this is of no use, because there is a strategy that is known to result in a score of alpha. Anything less than or equal to alpha is no improvement.

The second score is *beta*. Beta is the *worst-case scenario for the opponent*. It's the worst thing that the opponent has to endure, because it's known that there is a way for the opponent to force a situation no worse than beta, from the opponent's point of view. If the search finds something that returns a score of beta or better, it's too good, so the side to move is not going to get a chance to use this strategy. When searching moves, each move searched returns a score that has some relation to alpha and beta, and the relation is very important and might mean that the search can stop and return a value.

If a move results in a score that was *less than or equal to alpha*, it was just a bad move and it can be forgotten about, since, as I stated a few paragraphs ago, there is known to be a strategy that gets the moving side a position valued at alpha.

If a move results in a score that is *greater than or equal to beta*, this whole node is trash, since the opponent is not going to let the side to move achieve this position, because there is some choice the opponent can make that will avoid it. So if we find something with a score of beta or better, it has been proven that this whole node is not going to happen, so the rest of the legal moves do not have to be searched.

If a move results in a score that is greater than alpha, but less than beta, this is the move that the side to move is going to plan to play, unless something changes later on. So alpha is increased to reflect this new value. It's possible, and in fact quite common, that none of the legal moves result in a score that exceeds alpha, in which case this position was junk from our point of view, and we will avoid it by making a different choice somewhere above here in the game tree.

Finding the rotten fish in the second bag was like exceeding beta. If there had been no fish in the bag, determining that the six-pack of pop bag was better than the sandwich bag would have been like exceeding alpha. Here is the algorithm, with changes from min-max highlighted:

```
AlphaBeta(int depth, int alpha, int beta)
{
    if (depth == 0) return Evaluate();
    GenerateLegalMoves();
    while (MovesLeft()) {
        MakeNextMove();
        val = - AlphaBeta(depth - 1, -beta, -alpha);
        UnmakeMove();
        if (val >= beta) return beta;
        if (val > alpha) alpha = val;
    }
    return alpha;
}
```

If the highlighted characters are removed, what is left is a min-max function. As can be seen, there aren't many changes. The function is passed the depth it should search, and -INFINITY as alpha, and +INFINITY as beta:

```
val = AlphaBeta(5, -INFINITY, INFINITY);
```

This does a five-ply search. The trick that avoids writing a Min function in addition to a Max function is also used in the case of alpha-beta function. The only added complication is that alpha and beta are also flipped around. When the function recurses, alpha and beta negate and switch places. This causes a situation that is more complicated than the bag example, but just as provably better than min-max. What ends up happening is that in many places in the tree, beta is easy to exceed, and so a tremendous amount of work is avoided.

A possible weakness

The algorithm is *heavily dependent upon the order in which moves are searched*. If you consistently search the worst move first, a beta cutoff is never achieved, and so the algorithm works just like min-max, meaning very inefficiently. The algorithm ends up looking at the entire game tree, just like min-max. If the program always manages to pick the best move to search first, the math is such that the effective branching factor is equal to approximately the square root of the expected branching factor. This is the best possible case for the alpha-beta algorithm. Since chess has a branching factor of 35, this means that the alpha-beta algorithm performed efficiently upon a chess tree will produce a branching factor of approximately six. This is a massive improvement, and it allows you to search twice as deeply in the same number of nodes. This is why move ordering is extremely important when alpha-beta search is used.

Table de transposition

Aux Echecs, les deux séquences « 1. e4 d6 2. d4 » et « 1. d4 d6 2. e4 » conduisent à la même position. Donc au cours d'une recherche arborescente, il est possible de rencontrer un nœud déjà exploré. L'idée pour de ne pas explorer deux ou plusieurs fois le même nœud est de stocker le résultat d'une exploration dans une table de « transposition ». Au début de l'exploration d'un nœud, on interroge la table pour savoir si un résultat existe déjà. Si c'est le cas, on utilise le résultat tel quel. A la fin d'une exploration effective on écrit le résultat dans la table. Cela donne le code suivant :

```
int AlphaBeta(int depth, int alpha, int beta) {
    int hashf = hashfALPHA;
    if ((val = ProbeHash(depth, alpha, beta)) != valUNKNOWN)
        return val;
    if (depth == 0) {
        val = Evaluate();
        RecordHash(depth, val, hashfEXACT);
        return val;
    }
    GenerateLegalMoves();
    while (MovesLeft()) {
        MakeNextMove();
        val = -AlphaBeta(depth - 1, -beta, -alpha);
        UnmakeMove();
        if (val >= beta) {
            RecordHash(depth, beta, hashfBETA);
            return beta;
        }
        if (val > alpha) {
            hashf = hashfEXACT;
            alpha = val;
        }
    }
    RecordHash(depth, alpha, hashf);
    return alpha;
}
```

Bien sûr, il faut avoir défini la structure de données d'un élément de la table :

```
#define hashfEXACT 0
#define hashfALPHA 1
#define hashfBETA 2
typedef struct tagHASHE {
    U64 key;
    int depth;
    int flags;
    int value;
    MOVE best;
} HASHE;
```

et les deux fonctions d'accès à la table:

```
int ProbeHash(int depth, int alpha, int beta) {
    HASHE * phashe = & hash_table[ZobristKey() % TableSize()];
    if (phashe->key == ZobristKey()) {
        if (phashe->depth >= depth) {
            if (phashe->flags == hashfEXACT)
                return phashe->val;
            if ((phashe->flags == hashfALPHA) && (phashe->val <= alpha))
                return alpha;
            if ((phashe->flags == hashfBETA) && (phashe->val >= beta))
                return beta;
        }
        RememberBestMove();
    }
    return valUNKNOWN;
}

void RecordHash(int depth, int val, int hashf) {
    HASHE * phashe = & hash_table[ZobristKey() % TableSize()];
    phashe->key = ZobristKey();
    phashe->val = val;
    phashe->hashf = hashf;
    phashe->depth = depth;
    phashe->best = BestMove();
}
```

Expliquons les différents champs de la structure. Le premier `key` est la clé ou nombre de Zobrist [Zobrist 1990]. Il sert premièrement trouver rapidement dans la table l'élément correspondant à la position courante et deuxièmement à effectuer une vérification.

On peut considérer que `key` est codé en deux parties. Les N premiers bits correspondent à l'index dans la table ou « hash code ». Le hashcode est obtenu avec `ZobristKey() % TableSize()`. Deux positions du jeu peuvent avoir le même hashcode. En revanche, il est très peu probable que deux positions aient la même clé ou nombre de Zobrist, ce qui s'appelle une collision. En effet, les nombres de Zobrist sont construits avec des nombres aléatoires [Zobrist 1990] sur $N+M$ bits avec M assez grand pour rendre très peu probable une collision. Quand on trouve un enregistrement existant dans la table, on commence donc par vérifier que la clé de l'enregistrement est bien égale au nombre de Zobrist de la position courante avec le test `de(phashe->key == ZobristKey())`. Si le test est positif, on est quasiment certain que les informations que l'on va lire dans la table correspondent à la position courante. Si le test est négatif, on est dans le cas d'une collision, il ne faut surtout pas utiliser l'information de cet enregistrement qui ne correspond pas à la position courante. La fonction retourne alors `valUNKNOWN` et l'algorithme alfabéta va explorer l'arbre situé sous la position courante.

Pour comprendre comment créer un nombre de Zobrist pour une position d'un jeu, se reporter à [Zobrist 1990]. Cela repose sur des nombre aléatoires et l'opérateur XOR. Cette technique garantit une probabilité de collision entre deux positions arbitrairement faible pourvu que l'on ait assez de bits pour coder les nombres de Zobrist.

Le champ suivant est le résultat de l'exploration `val`. Pas de commentaire particulier sauf que c'est la valeur alfabéta recherchée.

Les champs suivants sont les conditions dans lesquelles le résultat a été obtenu : la profondeur `depth` et le type de retour `hashf` : exact ou de type alfa ou de type bêta. En effet, si on a écrit dans la table un resultat obtenu avec un `depth` petit, il ne sera pas valide si on interroge la table avec un `depth` grand. De plus, dans le contexte alfabéta, on a vu que le résultat trouvé, la valeur alfabéta, n'est pas toujours la valeur minimax ; cela dépend si on a une valeur alfa, une valeur bêta ou un résultat exact. Donc on stocke le type du résultat : valeur alfa (`hashfALFA`), valeur bêta (`hashfBETA`) ou valeur exacte (`hashfEXACT`). Le type du résultat est utilisé lors de la lecture de la table pour savoir quelle valeur retourner (cf `ProbeHash` ci-dessus).

Le dernier champ `best` sert en combinaison avec « iterative deepening » présenté dans la prochaine section.

L'amélioration avec table de transposition est très efficace. Pour les jeux de plateaux tels que le Go, ou le Clobber, on peut passer d'un nombre de positions en $O(n!)$ à un nombre en $O(2^n)$. Pour des valeurs de n pas très grandes, le rapport entre les deux est déjà énorme.

A la fin de la recherche arborescente avec table de transposition, la table est remplie de valeurs alfabéta utilisables pour jouer en réflexe.

Avec les ordinateurs d'aujourd'hui on prend pour N une valeur autour de 20 à 25 selon la taille de la mémoire disponible. Cela donne 2^{20} éléments possibles dans la table $\approx 10^6$ éléments. Si comme dans l'exemple ci-dessus, chaque élément fait 6 champs de 4 octets = 24 octets, la taille de la table fait environ 24 mégaoctets. Avec 23 bits, cela ferait environ 200 Mégaoctets soit la taille de la mémoire d'un ordinateur actuel. C'est ce qui peut se faire pour un programme d'Echecs. Pour M , une valeur de 10 environ garantit une probabilité de collision < 0.0001 . Donc dans 32 bits on peut généralement faire tenir $N+M$ bits nécessaires pour le nombre de Zobrist. Si on a besoin de plus, on prend un nombre de Zobrist sur 64 bits.

Ceci dit on peut utiliser des tables de transposition plus petites pour des jeux moins complexes, cela reste utile. Qui plus est, le gain apporté par une grande table de transposition n'est visible que si l'on a le temps de la remplir, c'est-à-dire de faire suffisamment d'exploration préalable

Enfin on peut utiliser les tables de transposition sans alfa-béta amais avec mini-max. En pratique, attention : les erreurs dues à l'utilisation d'une table de transposition sont faciles à faire et quand elle sont faites, elles sont difficiles à trouver.

Iterative deepening

La référence est [Korf 1985]. La question qui se pose à alfabéta est de savoir à quelle profondeur effectuer la recherche. Il est difficile de prévoir à l'avance combien de temps une recherche va prendre.

L'idée est d'explorer à profondeur 1 et si il reste du temps, explorer à profondeur 2, etc jusqu'à une profondeur inconnue mais fixé par le manque de temps.

L'avantage explicite de cette méthode est la bonne utilisation du temps. Après chaque profondeur, un coup est disponible pour être joué. C'est un *algorithme « anytime »*. Cette méthode s'appelle « iterative deepening » (ID) ou approfondissement itératif. Son pseudo-code est le suivant :

```
for (depth = 1;; depth++) {
    val = AlphaBeta(depth, -INFINITY, INFINITY);
    if (TimedOut()) break;
}
```

Un inconvénient apparent de cette méthode est que l'on semble effectuer plusieurs fois la recherche pour tous les noeuds situés à une profondeur $depth-1$ et qu'il semble que cette recherche soit beaucoup plus coûteuse en nombre de nœuds explorés. En fait l'inconvénient n'est que apparent. Si b est le facteur de branchement de l'arbre du jeu et D la profondeur, le nombre total de nœuds explorés par ID à une profondeur D est :

$$N_{ex} = Db + (D-1)b^2 + (D-2)b^3 + \dots + 2.B^{D-1} + B^D$$

En posant $x = 1/b$, on montre facilement que $N_{ex} \leq B^D / (1-x)^2$.

En pratique si $B=10$, le nombre de nœuds explorés par ID est 20% plus grand que celui par un alfa-béta normal. En théorie, *son temps de calcul est en $O(B^D)$ comme alfabéta*.

ID est très efficace car il atténue le fait que alfabéta soit un algorithme de recherche en profondeur d'abord. Le problème de le profondeur d'abord est que l'on peut explorer des nœuds très profondément pour trouver une solution dans la première partie de l'arbre et ne pas voir une solution plus courte dans une deuxième partie de l'arbre. ID garantit que la séquence trouvée est optimale et surtout *la plus courte*. C'est très important, s'il existe une séquence gagnante de deux coups, alfa-béta n'est pas certain de la trouver en un temps petit alors que ID la trouvera.

En plus, de tout cela *ID se combine très bien avec TT*. En effet à chaque nœud exploré, en plus de la valeur du nœud, on peut stocker le meilleur coup du nœud. A la prochaine itération, on peut utiliser lire ce coup proposé par la table et l'explorer en premier. A priori, il était bon à l'itération précédente, donc il a de bonnes chances d'être encore bon. Comme alfabéta est sensible à l'ordre des coups proposés, utilisé le « meilleur coup » stocké dans la table est une très bonne heuristique en pratique.

MTD(f)

La référence à MTD(f) est [Plaat & al 1995].

L'idée de MTD(f) est de faire des recherches arborescentes avec des fenêtres de taille minimale.

Faire les 2 exercices sur alfabéta avec fenêtre de taille minimale, cas « supérieur » et cas « inférieur ».

Le pseudo-code de MTD(f) est le suivant (repris de <http://www.cs.vu.nl/~aske/mtdf.html>):

```
int MTDF(int d, int f) {
    int g = f;
    int upperbound = +INFINITY;
    int lowerbound = -INFINITY;
    for (; lowerbound < upperbound;) {
        beta = (g == lowerbound) ? g + 1 : g;
        g = AlphaBeta(d, beta - 1, beta);
        if (g < beta) upperbound = g;
        else lowerbound = g;
    }
    return g;
}
```

L'idée est très simple. Les deux exercices ont montré que le nombre de coupes est très important quand on effectue une recherche avec une fenêtre de taille minimale (α et β imposés tels que $\alpha = \beta - 1$). On sait [Campbell & Marsland 1983] que :

Si le résultat est β , alors la valeur minimax $\geq \beta$

Si le résultat est α , alors la valeur minimax $\leq \alpha$

Donc en effectuant d'abord une recherche avec la fenêtre f $f+1$ on va savoir si minimax $\leq f$ ou pas. Le point fort de MTD(f) est d'utiliser TT afin de pouvoir ré-utiliser les résultats des itérations précédentes dans l'itération courante (comme dans ID).

La *valeur initiale f est cruciale*. En général, on choisit la valeur obtenue lors du dernier appel à MTD(f).

Aux Echecs, MTD(f) est désormais utilisée. Elle réduit de 10% le temps de calcul alfabéta.

MTD signifie Memory Test Driver. 1° La première idée est d'utiliser la « mémoire » c'est-à-dire les tables de transposition. 2° La deuxième idée est de faire un « test » de savoir si minimax est supérieur ou inférieur à une valeur. Le test coûte largement moins cher que la recherche normale. 3° La troisième idée est celle de « driver » ou classe d'algorithme. MTD(+ ∞) est l'instance de MTD qui commence avec une valeur très grande et qui diminue. MTD(- ∞) idem avec une valeur initiale très petite. L'idée de MT est originalement due à J Pearl, l'auteur de « scout », ou PVS, présenté au chapitre suivant.

Principal Variation Search

An enhancement to alpha-beta

Principal Variation Search (PVS) is a means of getting a small performance improvement out of the alpha-beta algorithm. Any node in an alpha-beta search belongs to one of three types: *Alpha node* : Every move you search will have a value less than or equal to alpha, meaning that none of the moves in here will be any good, probably because the starting position is bad for the side to move. *Beta node* : At least one of the moves will return a score greater than or equal to beta. *Principal variation (PV) node*: One or more of the moves will return a score greater than alpha, but none will return a score greater than or equal to beta.

Sometimes you can figure out what kind of node you are dealing with early on. If the first move you search *fails high* (returns a score greater than or equal to beta), you've clearly got a beta node. If the first move *fails low* (returns a score less than or equal to alpha), assuming that your move ordering is pretty good, you probably have an alpha node. If the first move returns a score between alpha and beta, you probably have a PV node. Of course, you could be wrong in two of the cases. Once you fail high, you return beta, so you can't make a mistake about that, but if the first move fails low or is a PV move, it's still possible that a later move will come back with a higher value.

Principal Variation Search makes the assumption that if you find a PV move when you are searching a node, you have a PV node. It assumes that your move ordering will be good enough that you won't find a better PV move, or a fail-high move (which would cause this to become a beta node), amongst the other moves. Once you've found a move with a score that is between alpha and beta, the rest of the moves are searched with the goal of proving that they are all bad. It's possible to do this a bit faster than a search that worries that one of the remaining moves might be good. If the algorithm finds out that it was wrong, and that one of the subsequent moves was better than the first PV move, it has to search again, in the normal alpha-beta manner. This happens sometimes, and it's a waste of time, but generally not often enough to counteract the savings gained from doing the bad move proof search referred to earlier. Here is the algorithm, overlaid on alpha-beta, with changes highlighted:

```
int AlphaBeta(int depth, int alpha, int beta) {
    BOOL fFoundPv = FALSE;
    if (depth == 0) return Evaluate();
    GenerateLegalMoves();
    while (MovesLeft()) {
        MakeNextMove();
        if (fFoundPv) {
            val = -AlphaBeta(depth - 1, -alpha - 1, -alpha);
            if ((val > alpha) && (val < beta))
                val = -AlphaBeta(depth - 1, -beta, -alpha);
        }
        else val = -AlphaBeta(depth - 1, -beta, -alpha);
        UnmakeMove();
        if (val >= beta) return beta;
        if (val > alpha) {
```

```

        alpha = val;
        fFoundPv = TRUE;
    }
}
return alpha;
}

```

The algorithm is all about the highlighted if-block in the middle of the function. If no PV move has been found, AlphaBeta() is recursed normally. If one has been found, everything changes. Rather than searching with a normal (alpha, beta) window, we search with (alpha, alpha + 1). The premise is that all of the searches are going to come back with a score less than or equal to alpha, and if this is going to happen, the elimination of the top end of the window results in more cutoffs. Of course, if the premise is wrong, the score comes back at alpha + 1 or higher, and the search must be re-done with the wider window. The performance improvement for PVS has been claimed to be 10%.

Search instability issues

If you search with an (alpha, alpha + 1) window, and the score comes back above the window (but doesn't exceed beta), you have to re-search. You'd think the re-search must come back with a score between alpha and beta, but this is not necessarily so. It might not be so because of search instability, which is something that I will discuss at various points. The routine as written above defends against this situation, and handles it properly if it happens. If you are going to write this routine and change things around, be very careful that you don't assume that your search will always be stable. You have to not crash and burn if you get a value back that you supposedly just proved you can't get.

Collecting the principal variation

People often ask how to retrieve the principal variation line after a search has completed. The principal variation is the line (predicted sequence of moves) that the program thinks represents best play for both sides. This is not returned by an un-modified alpha-beta function-- all alpha-beta returns is a score. What follows are modifications to a normal alpha-beta search, which collect the principal variation. Changes are highlighted.

```

typedef struct tagLINE {
    int cmove ;
    MOVE argmove[moveMAX] ;
} LINE;

int AlphaBeta(int depth, int alpha, int beta, LINE * pline)
{
    LINE line;
    if (depth == 0) {

```

```

    pline->cmove = 0;
    return Evaluate();
}
GenerateLegalMoves();
while (MovesLeft()) {
    MakeNextMove();
    val = -AlphaBeta(depth - 1, -beta, -alpha, &line);
    UnmakeMove();
    if (val >= beta) return beta;
    if (val > alpha) {
        alpha = val;
        pline->argmove[0] = ThisMove();
        memcpy(pline->argmove + 1, line. argmove,
            line. cmove * sizeof(MOVE));
        pline->cmove = line. cmove + 1;
    }
}
return alpha;
}

```

This is somewhat inefficient because it uses a lot of stack space, but the code would work and would not be slow. With these changes, if the function returns a value between alpha and beta, it returns not only the value, but the line (series of predicted moves) that resulted in the value. This is true for any node in the tree, including the root, which is why this is worth doing. You may some call to alpha-beta:

```

val = AlphaBeta(depth, -INFINITY, INFINITY, line);

```

What you get is the value of the position, and in the line buffer are the moves that constitute the predicted line. The line data structure is just an array of moves that were made in this line, and an integer that lets you know how many there were in the line. If you call AlphaBeta with depth = 0, the function calls Evaluate() and returns that value. There are no moves searched, so there is no move selected, so along with the score is a line of length zero. If you call the search with some other depth, and you find a move whose score is between alpha and beta, you receive not just the score, but the moves that resulted in the score. In order to create the line for this instance of AlphaBeta, you take the move that was just searched, stick it in the line buffer that was passed in, and append the moves from the local line buffer. You might be asking, Why is there both a line buffer passed in, and a new one allocated on the stack with each recursive call? The reason is that you don't want to wreck a line that has already been created, if you find a partial line somewhere later in the tree, and that line gets refuted. It is not true that if you find a node whose score is between alpha and beta, that that node will work its way all the way to the root of the tree. There are lots of fragmentary PV's that get discarded. For those of you who are aghast because I used memcpy in an inner loop, it costs nothing since it is rarely executed.

Null move pruning

Null-move forward pruning allows a chess program to experience a dramatic reduction in branching factor with some manageable risk of missing something important. It results in a dramatic improvement in search depth. The technique has been described in several publications, but the article that got everyone thinking about null-move is [Donninger 1993].

Imagine a chess position in a tree somewhere. Your program is going to search each of the legal moves in this position to a depth of D . Null-move forward pruning is a step you perform prior to searching any of the moves. You ask the question, If I do nothing here, can the opponent do anything? Note that in the regular case described two paragraphs ago, you are not asking this question. You are asking about the best way to hurt the opponent. Asking if the opponent can hurt you is something very different.

It turns out that there are a lot of cases where the opponent can't hurt you. Before you search any moves, and in fact before you generate any moves, you do a reduced-depth search, with the opponent to move first, and if that search results in a score $\geq \beta$, you simply return β without searching any moves. The idea is that you give the opponent a free shot at you, and if your position is still so good that you exceed β , you assume that you'd also exceed β if you went and searched all of your moves. The reason this saves time is that this initial search is made with *reduced depth*. The depth reduction factor is called R . So rather than searching all of your moves to depth D , you search your opponent's moves with a depth of $D-R$. An excellent value of R is 2. So, if for instance you were going to search all of your moves to depth 6, you end up searching all of your opponent's moves to depth 4. This results in a massive time savings, and a practical speedup of a ply or two. That's a vastly huge amount! Here is the source, added to an alpha-beta search, with changes highlighted:

```
#define R 2

int AlphaBeta(int depth, int alpha, int beta) {
    if (depth == 0) return Evaluate();
    MakeNullMove();
    val = -AlphaBeta(depth - 1 - R, -beta, -beta + 1);
    UnmakeNullMove();
    if (val >= beta) return beta;
    GenerateLegalMoves();
    while (MovesLeft()) {
        MakeNextMove();
        val = -AlphaBeta(depth - 1, -beta, -alpha);
        UnmakeMove();
        if (val >= beta) return beta;
        if (val > alpha) alpha = val;
    }
    return alpha;
}
```

I'm using a trick in this code. I want to see if the score of the null-move search is β or better. If it's not β or better, I don't care how much worse than β it is, so I use a minimal window,

in order to try to get more cutoffs faster. Essentially I'm calling it with (beta - 1, beta), but since you have to reverse alpha and beta and negate them when you recurse, it comes out the way it appears in the source code. It goes without saying that this doesn't work if the side to move is in check (because the opponent immediately takes the king). The degree to which null-move forward pruning will be allowed to happen recursively must also be considered, since if you let several of them follow in a row the search degenerates down to nothing. One obvious attempt is to not allow two null-move searches to occur without an intervening real move. Another idea is to allow exactly two null-move searches before forcing a real move. In practice these seem to work about equally well. Well, almost no down-side.

Unfortunately, null-move forward pruning doesn't work in some cases. A big assumption is made -- the assumption that making a move will result in a higher score than not making a move. Unfortunately, there are a whole class of positions where this is not true, and these are common enough that there is even a name for them. These positions are called zugzwangs. A zugzwang is a position where you are fine if you don't move, but your position collapses if you are forced to move.

Some ideas

It's interesting to fiddle around trying to find a better value for R than two. You can try one, three, four, some larger number, or a number that changes depending upon depth of search, how much material is on the board, etc. I've never been able to do better than straight R=2, but there has been some research that suggests that other values might be good, and there are a few people who are using R=3 in at least part of the tree. It's also tempting to try to figure out a way to use null-move forward pruning in endgames, through some sort of verification search. If your null-move search fails high, you do a normal search to reduced depth, and see if that fails high as well. This seems to me like it; might be very expensive, but it could be worth looking at.

There are other enhancements that are worth trying, but I'm not trying to make a complete list here. You can look in the Donninger article, the article in Computers, Chess, and Cognition, or at any of Ernst Heinz' articles about null-move forward pruning and related topics.

Quiescence Search

Une bonne référence est [Beal 1990]. Chess contains many forcing tactical sequences. If someone takes your bishop with a knight, you'd better take their knight back. Alpha-beta search is not particularly tolerant of this kind of thing. You pass a depth parameter to the function, and when the depth gets to zero, it is done, even if someone's queen is hanging. A method of dealing with this is called quiescent search. When alpha-beta runs out of depth, rather than calling Evaluate(), a quiescent search function is called instead. This function evaluates the position, while being careful to avoid overlooking extremely obvious tactical conditions. Essentially, a quiescent search is an evaluation function that takes into account some dynamic possibilities. The classic quiescent search searches only captures. This causes a problem because captures are not compulsory in chess. If a position is even, but if the only capture available to you is QxP (losing a queen), you aren't compelled to capture the pawn, and the quiescent search shouldn't be forced to capture it either.

```
int Quies(int alpha, int beta) {
    val = Evaluate();
    if (val >= beta) return beta;
    if (val > alpha) alpha = val;
    GenerateGoodCaptures();
    while (CapturesLeft()) {
        MakeNextCapture();
        val = -Quies(-beta, -alpha);
        UnmakeMove();
        if (val >= beta) return beta;
        if (val > alpha) alpha = val;
    }
    return alpha;
}
```

This looks very similar to alpha-beta, but the differences are very important. The function calls the static evaluation, and if the score is good enough that it can force a cutoff without captures being attempted, a cutoff is immediately made (return beta). If the evaluation isn't good enough to cause a cutoff, but it's better than alpha, alpha is updated to reflect the static evaluation. Then the captures are tried, and if any of them causes a cutoff, the search ends. Perhaps none of them is any good, which is also no problem. This function has several possible outcomes. It's possible that the evaluation function will return a high enough score that the function can exit immediately via a beta cutoff. It's also possible that a capture can result in a beta cutoff. It's possible that the static evaluation will be bad, and none of the captures will be any good either. Or it is possible that none of the captures are any good, but the static evaluation can raise alpha a little bit.

Conclusion

Ce document a décrit alfabéta et certaines de ses améliorations.

Il n'a pas décrit [Allester 1988], [Allis & al 1994], [Berliner 1979], [Buro 1995], [Korf & Chickering 1996], [Schaeffer 1989], [Stockman 1979], [Tsuruoka & al 2002].

Il n'a pas décrit non plus beaucoup d'autres travaux sur la recherche arborescente...

Références

[Allester 1988], D. Allester, *Conspiracy numbers for min-max search*, Artificial Intelligence, vol 35, pages 287-310, 1988.

[Allis & al 1994], L. Allis, M. van der Meulen, J. van den Herik, *Proof-number search*, Artificial Intelligence, vol 66, pages 91-124, 1994.

[Beal 1990], D. Beal, *A generalised quiescence search algorithm*, Artificial Intelligence, vol 43, pages 85-98, 1990.

[Berliner 1979], H. Berliner, *the B* Tree Search algorithm: a best-first proof procedure*, Artificial Intelligence, vol 12, pages 23-40, 1979.

[Buro 1995], M. Buro, *ProbCut: An effective selective extension of the $\alpha\beta$ algorithm*, ICGA Journal, vol 18, n°2, pages 71-76, 1995.

[Campbell & Marsland 1983], M. Campbell, T. Marsland, *An analysis of alpha-beta pruning*, Artificial Intelligence, vol 20, pages 347-367, 1983.

[Donninger 1993], Christian Donninger, *Null Move and Deep Search*, ICGA Journal, vol 16, n°3, pages 137-143, 1993.

[Junghanns 1998], A. Junghanns, *Are there practical alternatives to alpha-beta in computer chess?*, ICCA Journal, 21 (1), pages 14-32, 1998.

[Knuth & Moore 1975], D. Knuth, R. Moore, *An analysis of alpha-beta pruning*, Artificial Intelligence, 6 (4), pages 293-326, 1975.

[Korf 1985], R. Korf, *Depth-first iterative deepening: an optimal admissible tree search*, Artificial Intelligence, 27 (1), pages 97-109, 1985.

[Korf & Chickering 1996], R. Korf, D. Chickering, *Best-first minimax search*, Artificial Intelligence, vol 84, pages 299-337, 1996.

Bruce Moreland (<http://www.seanet.com/~brucemo/chess.htm>)

[Pearl 1990], Judéa Pearl, *Heuristique*, CEPADUES, 1990.

Aske Plaat, *A Minimax Algorithm faster than NegaScout*,
(<http://www.cs.vu.nl/~aske/mtdf.html>)

[Plaat & al 1995], Aske Plaat, J Schaeffer, W Pijls, A de Bruin, *Best-First Fixed-Depth Game-Tree Search in Practice*, IJCAI-95, vol 1, pages 273-279, 1995.

[Schaeffer 1989], J. Schaeffer, *The history heuristic and alpha-beta Search, Enhancements in practice*, IEEE Transactions on PAMI, vol 11, n°11, 1989.

[Stockman 1979], G. Stockman, *A minimax algorithm better than alpha-beta ?* Artificial Intelligence, 12 (2), pages 179-196, 1979.

[Tsuruoka & al 2002], Y. Tsuruoka, D. Yokoyama, T Chikayama, *Game-tree search algorithm based on Realization Probability*, ICGA Journal, vol 25, n° 3, pages 145-152, 2002.

[Zobrist 1990], Albert Zobrist, *A new Hashing method with application for game playing*, ICGA Journal, vol 12, n° 2, pages 69-73, 1990.