MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.034 Artificial Intelligence
Fall, 2011

Recitation 3, Thursday, September 29

Game trees, minimax, & alpha-beta search Prof. Bob Berwick, 32D-728

1. Minimax search in a game tree.

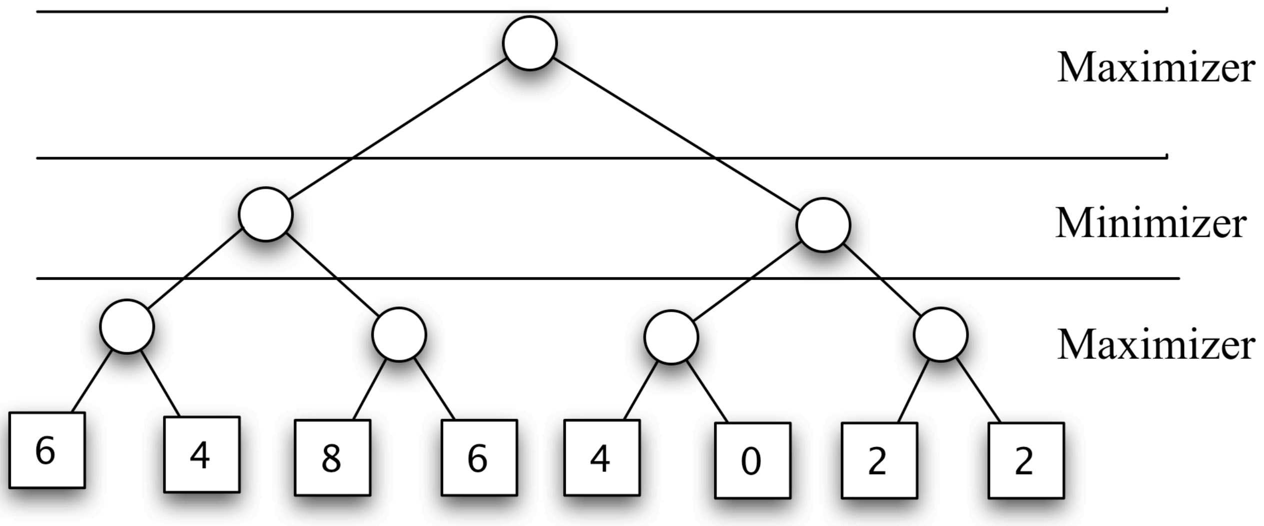
Notation: Let SV denote the 'static value' or 'worth' of a particular game state/board position. `SV(state)` returns the static value of the game state. These are the numbers we give you as the leaf nodes in the game search tree. They are generally in the perspective of the current player. Here is the minimax algorithm:

- If the limit of search has been reached, return the static value of the current position relative to the appropriate player.
- Otherwise, if level is a minimizing level, call Minimax on the children of the current position, and report the MINIMUM of the results.
- Otherwise, if level is a maximizing level, call Minimax on the children of the current position and report the MAXIMUM of the results.

Or in (somewhat wordy) pseudo-code:

```
function max-value(state, depth)
     1. if state is an end-state (end of game) or depth is 0
           return SV(state)
     2. v = -infinity
     3. for s in get-all-next-moves(state)
            v = max(v, min-value(state, depth-1))
     4. return v

function min-value(state, depth)
     1. if state is an end-state (end of game) or depth is 0
           return SV(state)
     2. v = +infinity
     3. for s in get-all-next-moves(state)
           v = min(v, max-value(state, depth-1))
     4. return v
```

Let's try minimax out on the following game tree, and figure out the line of play as well for the top-most player, the **Maximizer:**



What is the value that minimax returns here at the topmost, maximizer node? What is the line of play?

2. Alpha-beta search. Now let's think about this game tree some more. Consider the static value at the left-most **minimizer** node in the tree above. It is 6. Now, how was that computed? What happens if we change the fourth static evaluation from the left, containing a 6, to something lower, like 0? What if we make it something much higher, like 1000? Does the minimax value at the left-most minimizer node change as a result?

This result suggests that we can **avoid** ever computing the static evaluations of some game positions – a good thing, since this is expensive, in general. The upshot what is called **alpha-beta search.**

Alpha-beta search is simply an **efficient** form of minimax that gets the **same** result (evaluation at top and line of play) but with often (far) **fewer** static evaluations.

- Basic idea: track best move's value so far during minimax search, from perspective of both the maximizer and the minimizer
- For the **Maximizer**, that value is **Alpha** (goal: make it LARGE) – this is the **guaranteed best** that Maximizer can get (a **guaranteed lower bound** on the maximum value)
- For **Minimizer,** the value is **Beta** (goal: make it SMALL) – this is the **guaranteed best** that Minimizer can get (i.e., lose, a **guaranteed upper bound** on the minimum value)
- Initially, alpha= –infinity; beta = +infinity; so this is initially a range [alpha, beta] = [–infinity, +infinity]. As we search the tree, this range should narrow such that alpha=beta, and the values **should never cross** (see pruning below)
- Pruning:
  – When the Minimizer is examining its moves, determining **beta,** if any are **less than or equal to alpha** (**worse** for Maximizer), then the Minimizer's parent, the Maximizer, would **never** make that move because the move that yielded alpha is already better – so the Minimizer can abandon this node! (In other words: cut-off if ever **beta < alpha**)
  – When the Maximizer is examining its moves, determining **alpha,** if any are **greater than beta** (worse for Minimizer) then the Maximizer's parent, the Minimizer, would never make that move because the move that yielded beta is already better – so Maximizer can abandon this node! (In other words: cut-off if ever **alpha > beta**)

Remember this: **Maximizer calculates alpha; Minimizer calculates beta; <u>cut-off</u> if alpha ever crosses beta or vice-versa.**

Pseudo-code (here SV = 'static value' of a node)

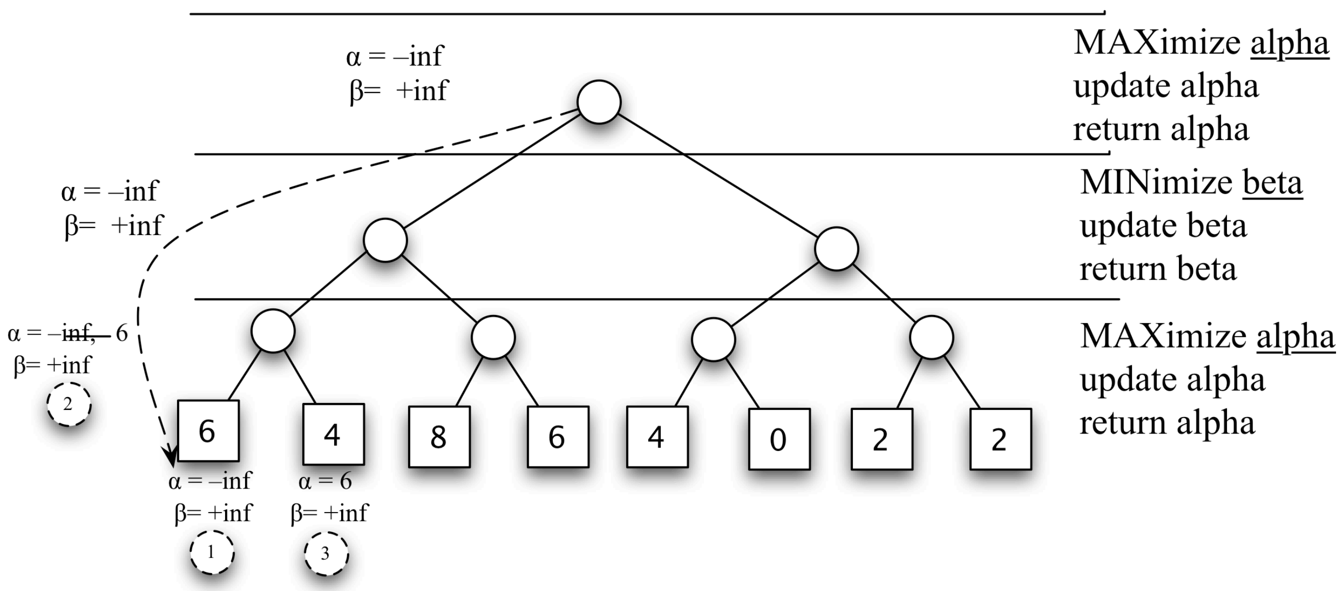
```
function max-value(state, depth, alpha, beta):
        1. if state is an end-state (end of game) or depth is 0 then
               return SV(state)
        2. v = -infinity                    # initial ALPHA value for node
        3. for s in get-all-next-moves(state)
               v=max(v, min-value(state, depth-1, alpha, beta)
               alpha = max(alpha, v)
               if alpha ≥ beta then         # alpha-beta cutoff!
                  return alpha              # exit from loop immediately     4.
return v

function min-value(state, depth, alpha, beta):
        1. if state is an end-state (end of game) or depth is 0 then
               return SV(state)
        2. v = +infinity                    # initial BETA value for node
        3. for s in get-all-next-moves(state)
               v=min(v, max-value(state, depth-1, alpha, beta)
               beta = min(beta, v)
               if alpha ≥ beta then         # alpha-beta cutoff!
                  return beta               # exit from loop immediately
        4. return v
```

Initially we set Alpha and Beta to their worst-case values: Alpha= –infinity (or some very large negative value), and Beta=+infinity.

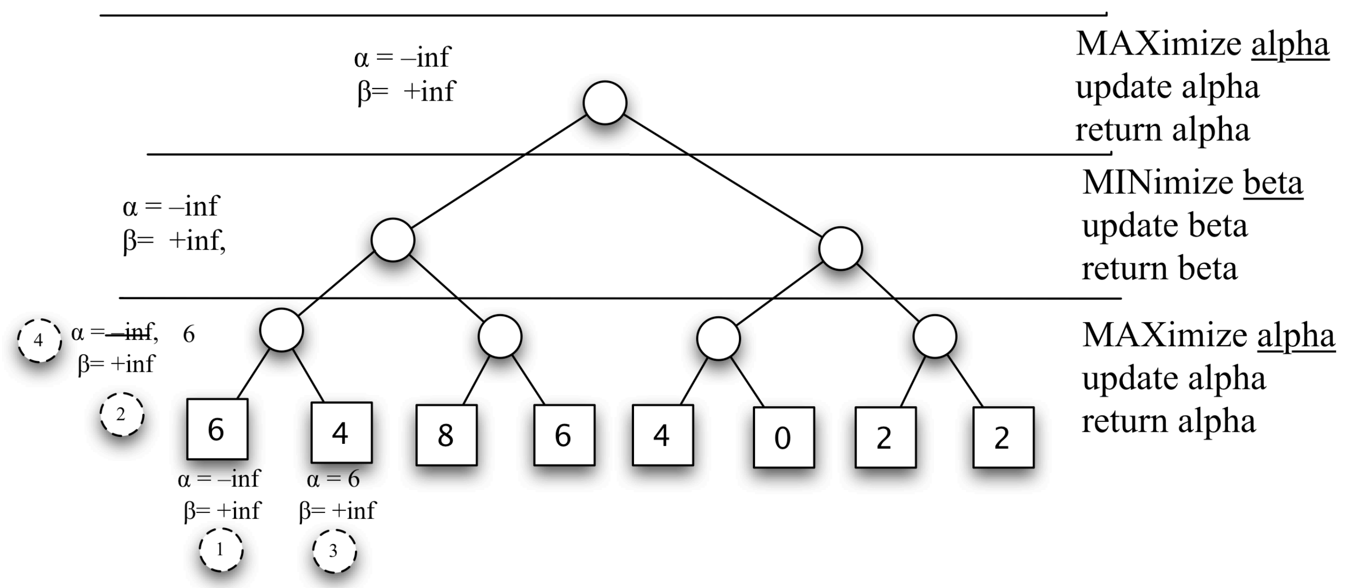
Note 1. In the code the **Minimizer** loop returns the value for **Beta** (remember, it's setting the upper bound, the **worst case** for the minimizer), while the **Maximizer** loop returns the value for **Alpha** (it is setting the lower bound, the **worst case** for the maximizer).
Note 2. The loops over daughters of a node are cut-off if ever alpha exceeds beta or vice-versa. **Repeat the mantra after me: ALPHA CAN NEVER BE GREATER THAN BETA (OR VICE-VERSA) IN ALPHA-BETA SEARCH.**

Note 3. The Alpha (conversely Beta) values are related as we move from level to level in the game tree: the Alpha (resp. Beta) values are passed *up* to be used at the next level in the minimax tree to become the refined Beta (resp. Alpha) bounds at the next level. What is the reasoning here? For example, assuming initial Alpha, Beta values of, say, –/+infinity, then if the Maximizer can set Alpha to be equal to, e.g., 100, we know that from the Minimizer's point of view one level above, the Minimizer **can do no worse than this**, so Beta at this level may now be decreased from +infinity to, tentatively, 100 (thought their loss might wind up being even less). In other words, if I (the maximizer) am guaranteed to *win* $100, then my opponent (the minimizer) cannot *lose* more than this amount so far (they might wind up losing less, as other parts of the tree are explored). One person's ceiling is another person's floor (as the song goes).

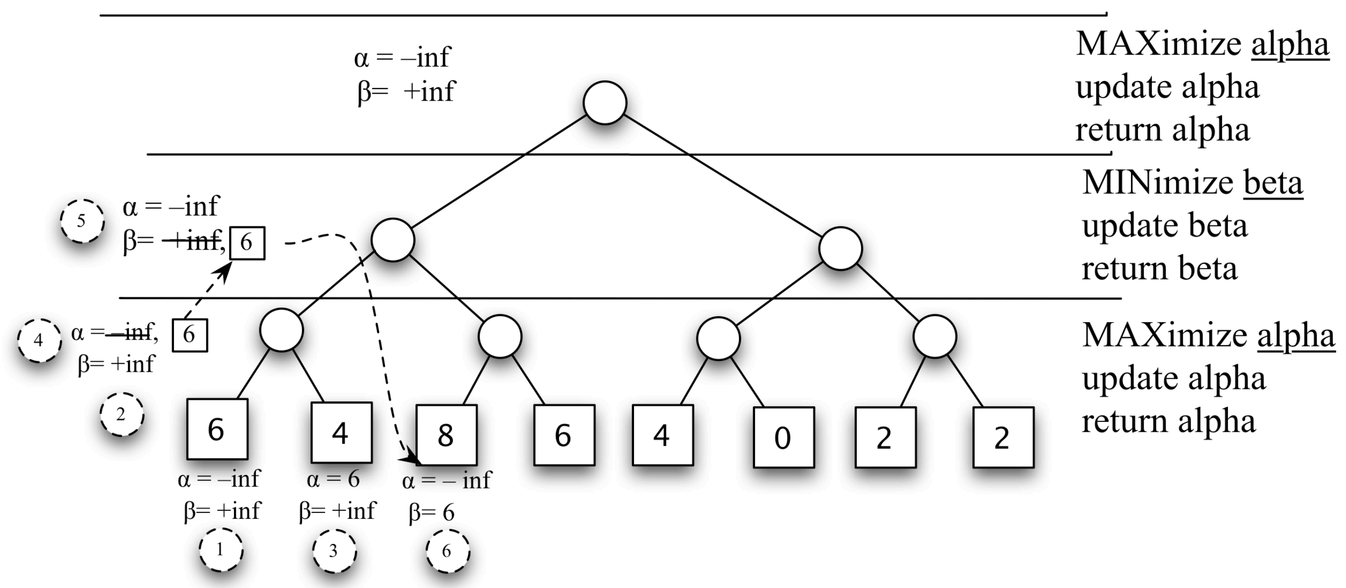
So now let's see how all this works and now alpha-beta search can save us effort with static evaluations in some cases. Let's go through the *same* minimax tree as before (so we should get the same answer!), but using alpha-beta pruning. Let's pay careful attention to how the values of alpha and beta are narrowed down, as well as how they are passed both up the tree (as return values) and down the tree. The first snapshot shows just the initialization of alpha, beta and diving down to the first static evaluation node on the left, at the bottom. Step numbers are shown inside dotted circles. Since this is a maximization node, we (tentatively) set Alpha to be 6 (it might get larger, since we are maximizing):



Steps 3-4: Carrying on, as shown just below, we evaluate the right-most sister of the bottom node, with a static value of 4. Since at this level we are *maximizing* (setting Alpha), this is smaller than the alpha value we have already, of 6. So we now can set Alpha=6 for this entire node, shown as Step #4.

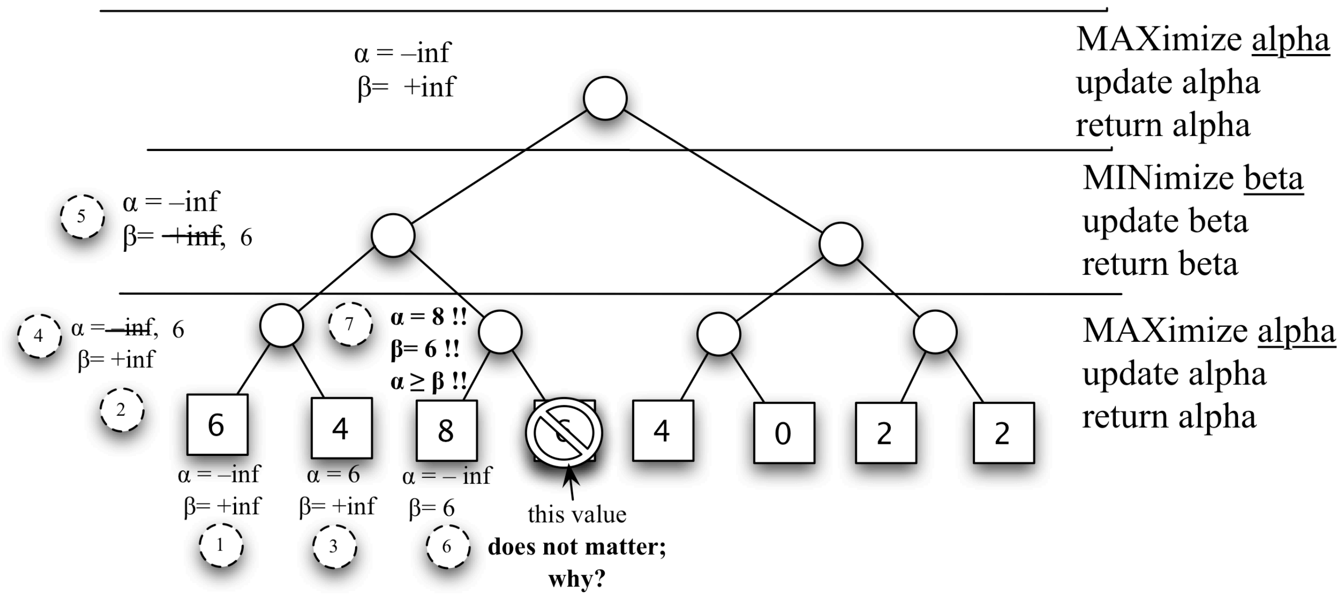


Steps 5-6: Now we are done with this MAXimizing subnode, so we can *return* from the MAXimizing level at this point, with Alpha=6, to the MINimizing level right above it. Since Alpha=6, recall this means that one level up, at the MINimizing level, we can (tentatively) set Beta to 6, and this is shown as Step #5 below, along with arrow indicating how the Alpha value has been transferred to its Beta value counterpart. (Note that the Alpha value at this level is still –Infinity, because this value can only be updated by running minimax on the left *above* this one). Finally, we can pass these values of α, β of (–inf, 6) to the down to the leaf nodes at the Maximizer level again, shown as Step #6:

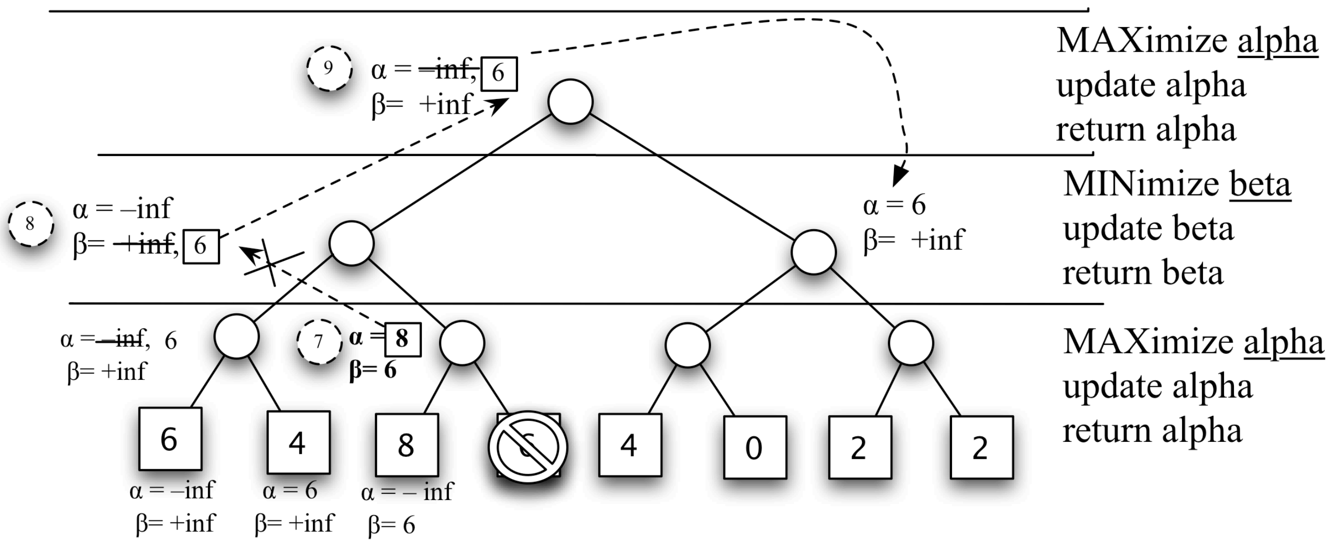


Step 7, alpha-beta cut-off: OK, since we're at a MAXimizing level, the static evaluation value of 8 at the left-most node tells us that we can set α tentatively to 8. If we do that, we arrive at Step #7 as shown in the figure immediately below. **Note now that we have set α =8 and β = 6.**

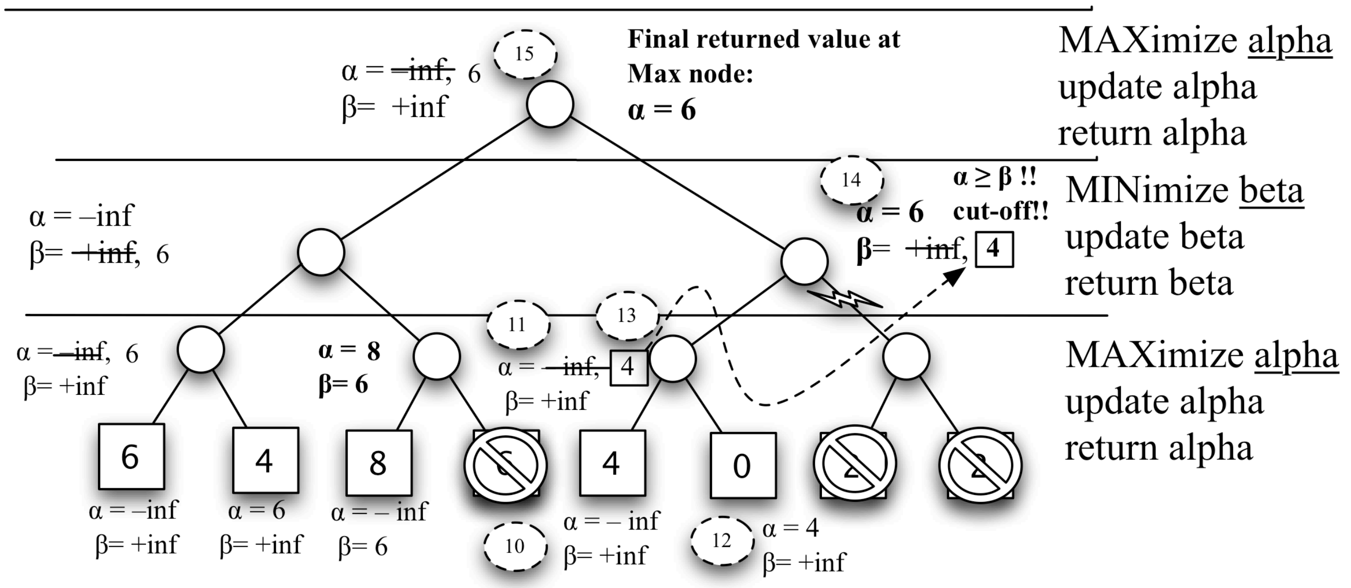
DANGER DANGER WILL ROBINSON!! α is NOW GREATER than β !! Therefore, we can EXIT from this MAXimizer call immediately **without ever having to evaluate the node on the right with its static evaluation of 6.** (We have drawn a big "NO" symbol through this node. This static evaluation, and that of any other static evaluation to the right underneath this maximizing node CANNOT NOT MATTER to the final result returned here. Can you remember WHY? If you ever forget, just trying making the crossed out static value either very small or very large, and you'll see it makes no difference to the α value.)



Steps 8-9: Note that the value returned from the MAXimizer search then is α = 8. Now we're back at the MINimizer level, and have the job of setting β, so α = 8 at the MAXimizer level implies β = 8 for the MINimizer, using the same reasoning as before. Taking the minimum of the two β values, 6 and 8, we can now fix β=6, at Step #8. Since β=6 at this MINimizer level, this implies that the most that can be **lost** from this node is 6; therefore, the minimum that can be **gained** at the next level up, the higher MAXimizer level, is at least 6, that is, (tentatively) α = 6, marked as Step #9, with β still at +infinity. We then pass the α = 6, β= +inf values down one level.



Steps 10-15: And now we can quickly roar through the rest of the tree, noting a final alpha-beta cutoff….note that when the dust settles, we get the *same* returned value 6 for the game tree (and the same line of play) as we did before with pure minimax, just as claimed – the line of play is always to the *left* down from the root node:



How effective is alpha-beta pruning? In this case, in all, we **saved 3 static evaluations** via **two cut-offs**, out of 8 possible static evaluations, about ½ of them. This turns out to be the theoretical best savings. It depends on the order in which children are visited. If children of a node are visited in the worst possible order, it may be that no pruning occurs. For max nodes, we want to visit the best child first so that time is not wasted in the rest of the children exploring worse scenarios. For min nodes, we want to visit the worst child first (from our perspective, not the opponent's.) When the optimal child is selected at every opportunity, alpha-beta pruning causes all the rest of the children to be pruned away at every other level of the tree; only that one child is explored. This means that on average the tree can searched twice as deeply as before—a huge increase in searching performance.
Note that the best case holds for the game tree above: at each of the last MAXimizer nodes, the leaves below are ordered from maximum to minimum values, i.e., (6, 4) for the children of the first node; (8, 6) for the second; (4, 0) for the next, and so on. More generally, we assume that the *leftmost* alternative is always the best move, for *all* levels. Suppose the tree is *d* levels deep (excluding the static evaluation layer), and has branching factor *b*. Then one can show that in general the number of static evaluations *s* needed is:

$$s = 2b^{d/2} - 1 \text{ for } d \text{ even}$$
$$s = b^{(d+1)/2} + b^{(d-1)/2} - 1 \text{ for } d \text{ odd}$$

In our case, d=2 and b=2, so the formula yields $2 \times 2^1 - 1 = 4 - 1 = 3$ as confirmed. Note that the value in both cases is reduced by the *square root* of the number of nodes in a full static evaluation, b^d.

Note that we can't in general ensure that the static evaluations in a game tree are 'optimistically' ordered in this way, because that would involve evaluating the nodes, exactly what we want to avoid. For example, suppose the order of the leaves of the tree above were the reverse: (2, 2); (0, 4); (6, 8); (4, 6). You should see that alpha-beta search on this reversed order gives zero cut-offs, because at the bottom, maximizer level the leaves are in *ascending* order under each maximizer node, so we have explore each of them. This is the worst case when maximizing. (The converse is true when minimizing.)

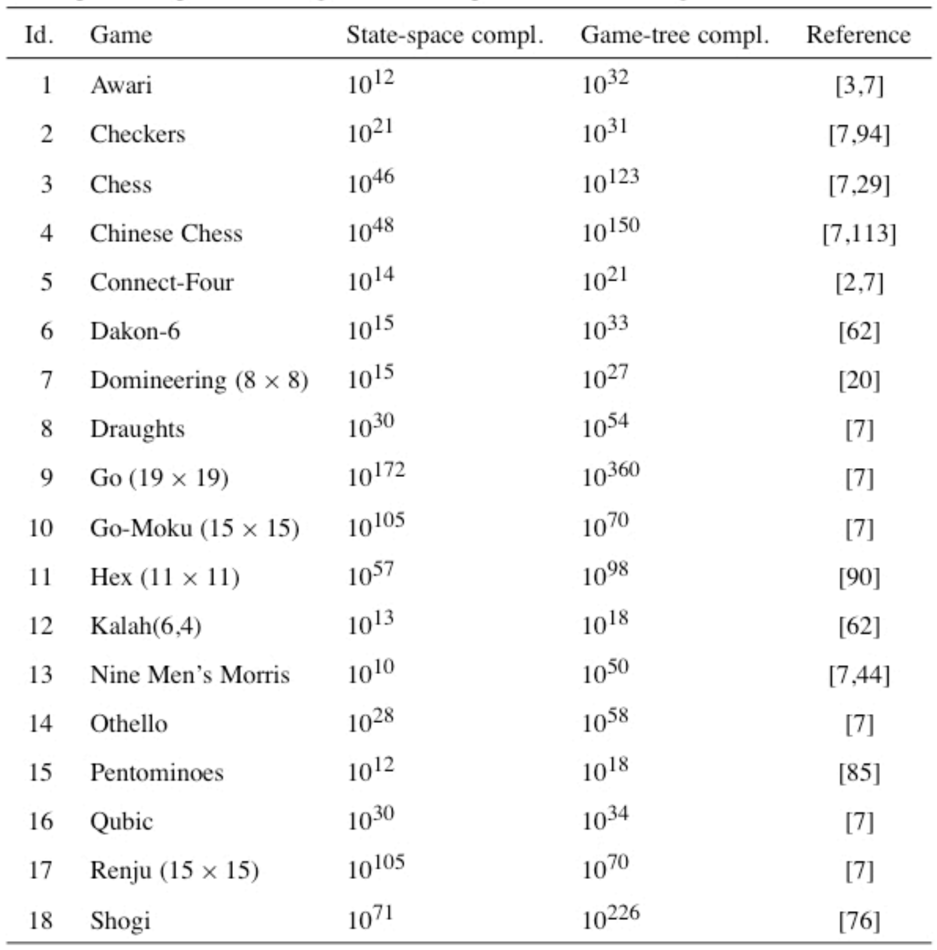
There are several ways to obtain the information about how to order nodes. First, one can develop heuristic methods that seem to do a good job of approximating such an ordering, sometimes thinking about the particular game itself. Second, what is sometimes done is to use a 'history heuristic' to remember what previous searches of the game tree, from previous moves (in the current game or in the past) have led to strong alpha-beta pruning, and/or strong evaluations at the 'root' node, and then 'hash' code these so that if they arise again, they can be efficiently re-used.

Chess has seen the development of many, many other heuristic methods. Some of these are: (1) transposition tables, or a hash memory of common, previously encountered board positions with associated scores, regardless of how we got there; (2) refutation moves, that is, alpha-beta search until we get at least a (sufficiently good) alpha value in response to the opponent's play, but not necessarily the best alpha-beta score; and (3) principal-variation moves (used with progressive deepening, see just below).

Note that alpha-beta is basically depth-first search. So, apart from alpha-beta, a second method that is commonly used in game tree searching is called *progressive deepening*, which is a way of embedding this depth-first search into a breadth-first framework. (Otherwise, we might run out of time diving far down into the tree and not have come up with a good sequence of moves to any level!) The way to do this is to use the notion of *iterative* deepening: first set *depth* = 1 and solve this tree by alpha-beta. This gives a *principal variation* set of moves, down to this fixed depth. Then we increment the *depth* by 1, repeating the alpha-beta search *starting from the line of play already established,* and so on, until one runs out of time, making sure that the move we first search on the next iteration at *depth*+1 is the best move we got from the search at the previous, shallower *depth*. Much more could be said about all of this – see, e.g., Adriaan de Groot,1965**,** *Thought and Choice in Chess.* Mouton & Co Publishers, The Hague, The Netherlands. Second edition 1978**.** ISBN 90-279-7914-6, or, more recently, Chess Metaphors: Artificial Intelligence and the Human Mind by Diego Rasskin-Gutman, MIT Press.

Some 'state space' sizes for game trees. Note the huge difference between checkers, chess, and Go. Reference: H. Jaap van den Herik, Jos W.H.M. Uiterwijk, Jack van Rijswijck, Games solved: Now and in the future, *Artificial Intelligence* 134 (2002) 277–311

Table 6
State-space complexities and game-tree complexities of various games



Id.	Game	State-space compl.	Game-tree compl.	Reference
1	Awari	10^{12}	10^{32}	[3,7]
2	Checkers	10^{21}	10^{31}	[7,94]
3	Chess	10^{46}	10^{123}	[7,29]
4	Chinese Chess	10^{48}	10^{150}	[7,113]
5	Connect-Four	10^{14}	10^{21}	[2,7]
6	Dakon-6	10^{15}	10^{33}	[62]
7	Domineering (8×8)	10^{15}	10^{27}	[20]
8	Draughts	10^{30}	10^{54}	[7]
9	Go (19×19)	10^{172}	10^{360}	[7]
10	Go-Moku (15×15)	10^{105}	10^{70}	[7]
11	Hex (11×11)	10^{57}	10^{98}	[90]
12	Kalah(6,4)	10^{13}	10^{18}	[62]
13	Nine Men's Morris	10^{10}	10^{50}	[7,44]
14	Othello	10^{28}	10^{58}	[7]
15	Pentominoes	10^{12}	10^{18}	[85]
16	Qubic	10^{30}	10^{34}	[7]
17	Renju (15×15)	10^{105}	10^{70}	[7]
18	Shogi	10^{71}	10^{226}	[76]

```
ply
 40 +                                                 * Deep Blue
    .
    .
    .
    |
    |
 14 +
    |
    |
 12 +
    |
    |
 10 +                                           * Deep Thought
    |                                      * Hitech
    |
  8 +                                  * Belle
    |                               * Belle          * Gary Kasparov
    |
  6 +                       * Belle                * Anatoly Karpov
    |               * Belle                          * Bobby Fischer
    |
  4 +
    |
    |
  2 +
    |
    |
  0 +____+____+____+____+____+____+____+____+____+____+____+____+____+
    0    1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.6  2.8  3.0  3.2  3.4

         U.S. Chess Federation Rating x 10^3
```

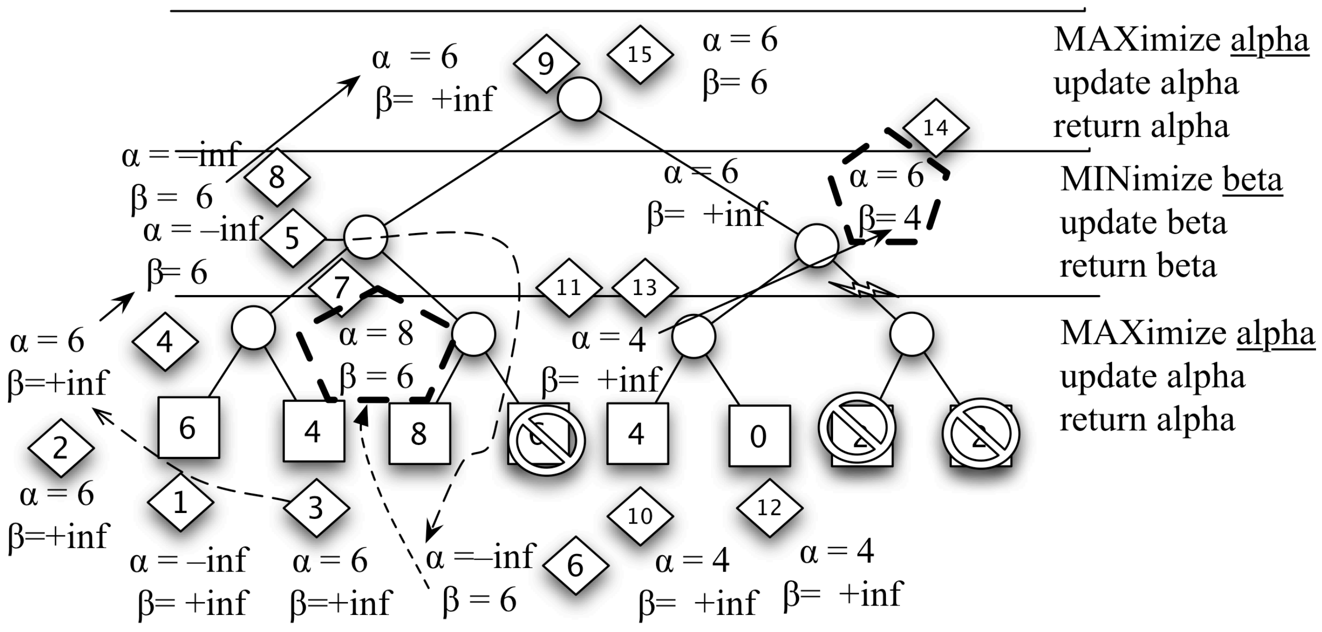
Calvin and HOBBES
by WATERSON
HA HA! YOUR CLEVER LITTLE STRATEGY HAS TRAGICALLY BACKFIRED! LOOK AT THIS JUMP! WATCH AND WEEP, FUZZ FACE!
THAT'S YOUR MOVE, RIGHT? I GET TO GO NOW, RIGHT? IT'S TOO LATE FOR YOU TO CHANGE YOUR MIND, RIGHT?
NOT SO FAST... MY HAND'S STILL ON IT.
JUMP, JUMP, JUMP! I WIN!
YOU WIN?!? AAUGGHH! YOU WON LAST TIME! I HATE IT WHEN YOU WIN!
AARRGGH! MFF! GNNK! I HATE THIS GAME! I HATE THE WHOLE WORLD!!
AGHHH! WHAT A STUPID GAME!
YOU MUST HAVE CHEATED! YOU MUST HAVE USED SOME SNEAKY, UNDERHANDED MIND-MELD TO MAKE ME LOSE! I HATE YOU!
I DIDN'T WANT TO PLAY THIS IDIOTIC GAME IN THE FIRST PLACE!
I KNEW YOU'D CHEAT! I KNEW YOU'D WIN!
OH! OOH! AARG!
© 1987 Universal Press Syndicate
AAAAAAAAAAAAA
HACK PANT PANT
WATERSON
LOOK, IT'S JUST A GAME.
I KNOW. YOU SHOULD SEE ME WHEN I LOSE IN REAL LIFE!

And finally, a quote to read and think about:

"With the supremacy of the chess machines now apparent and the contest of "Man vs. Machine" a thing of the past, perhaps it is time to return to the goals that made computer chess so attractive to many of the finest minds of the twentieth century. Playing better chess was a problem they wanted to solve, yes, and it has been solved. But there were other goals as well: to develop a program that played chess by thinking like a human, perhaps even by learning the game as a human does. Surely this would be a far more fruitful avenue of investigation than creating, as we are doing, ever-faster algorithms to run on ever-faster hardware.

"This is our last chess metaphor, then—a metaphor for how we have discarded innovation and creativity in exchange for a steady supply of marketable products. The dreams of creating an artificial intelligence that would engage in an ancient game symbolic of human thought have been abandoned. Instead, every year we have new chess programs, and new versions of old ones, that are all based on the same basic programming concepts for picking a move by searching through millions of possibilities that were developed in the 1960s and 1970s.

Like so much else in our technology-rich and innovation-poor modern world, chess computing has fallen prey to incrementalism and the demands of the market. Brute-force programs play the best chess, so why bother with anything else? Why waste time and money experimenting with new and innovative ideas when we already know what works? Such thinking should horrify anyone worthy of the name of scientist, but it seems, tragically, to be the norm. Our best minds have gone into financial engineering instead of real engineering, with catastrophic results for both sectors." – Gary Kasparov, 2010 *New York Review of Books*.


MAXimize alpha
update alpha
return alpha
MINimize beta
update beta
return beta
MAXimize alpha
update alpha
return alpha
α = 6
β= +inf
α = 6
β= 6
α = –inf
β = 6
α = –inf
β= 6
α = 6
β= +inf
α = 6
β= 4
α = 6
β=+inf
α = 8
β = 6
α = 4
β= +inf
α = 6
β=+inf
α = –inf
β= +inf
α = 6
β=+inf
α = –inf
β = 6
α = 4
β= +inf
α = 4
β= +inf
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
6
4
8
4
0