ALPHA BETA SEARCH

Advanced Algorithms Course

Complexity of Chess

- In 1950, Claude Shannon made an analysis of chess where he concluded that:
  - In average there are 30 legal moves available for each player anytime in the game.
  - Thus a move for White and then one for Black gives about 10^3 possibilities.
  - A typical game lasts about 40 moves.
  - There will be 10^{120} variations to be calculated from the initial position.
  - A machine operating at the rate of one variation per micro-second would require over 10^{90} years to calculate the first move!

Complexity of Chess

- Let's suppose we have the most powerful computer in the world available to play chess
  - Sunway TaihuLight (http://www.nsccwx.cn/wxcyw/)
  - Capable of 96 PetaFLOPS (96 x 10^{15} FLOPS)
  - Suppose one variation is calculated for each FLOating Point Operation per Second:

$$\frac{10^{120}}{96*10^{15}} = 1{,}04 * 10^{103} \text{ seconds}$$

  - We need 3,3*10^{93} centuries to decide the first move!

Alpha-Beta Pruning

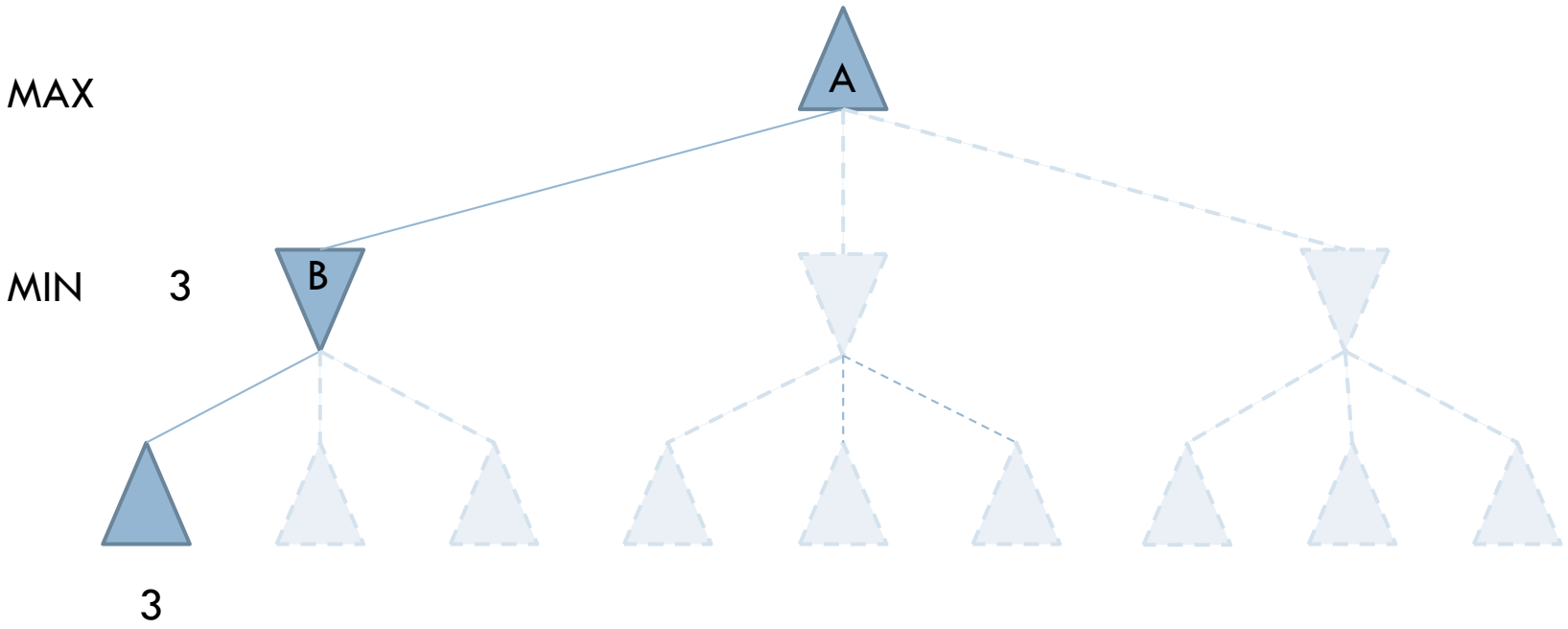
- We can reduce tree exploration by avoiding subtrees we know are useless to explore.
- Alpha–beta pruning gets its name from the two parameters that describe bounds on the backed-up values that appear anywhere along the path:

  α = Minimal value that MAX is guaranteed to achieve.

  β = Maximal value that MIN can hope to achieve.

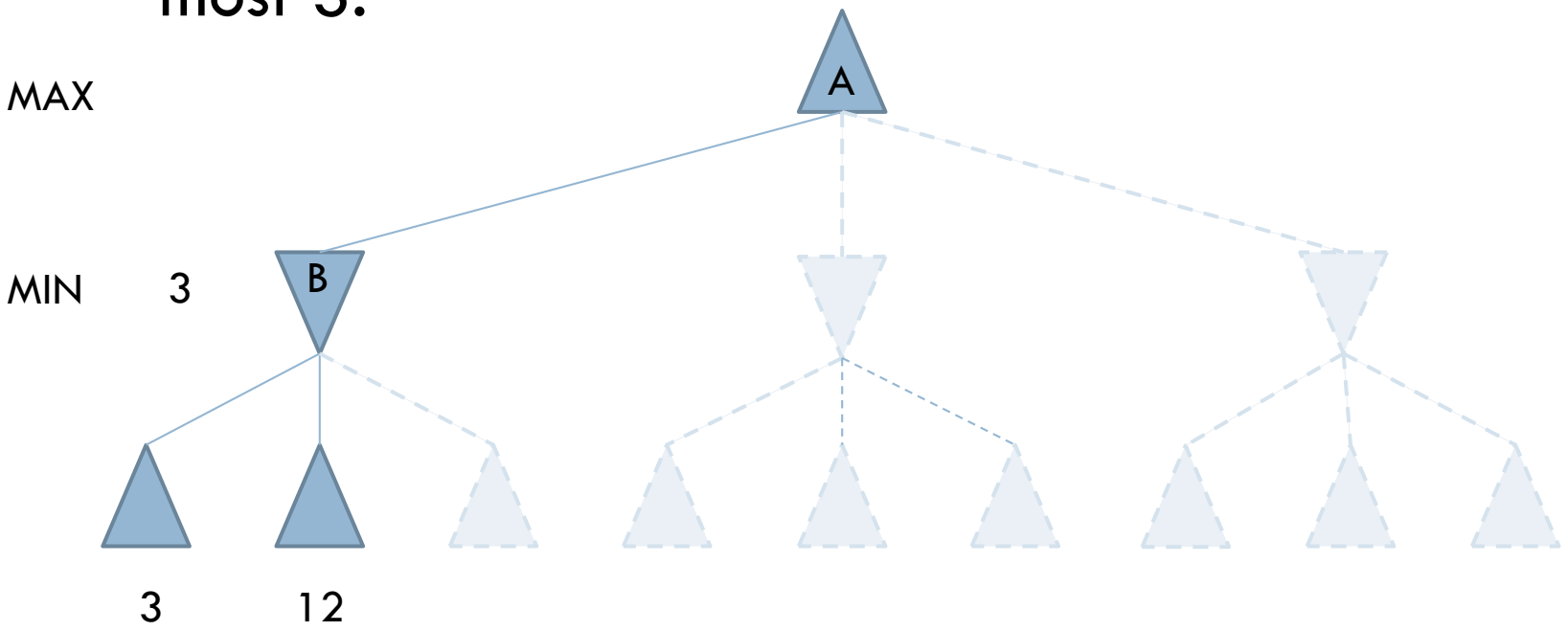
Alpha-Beta Pruning

- The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3.



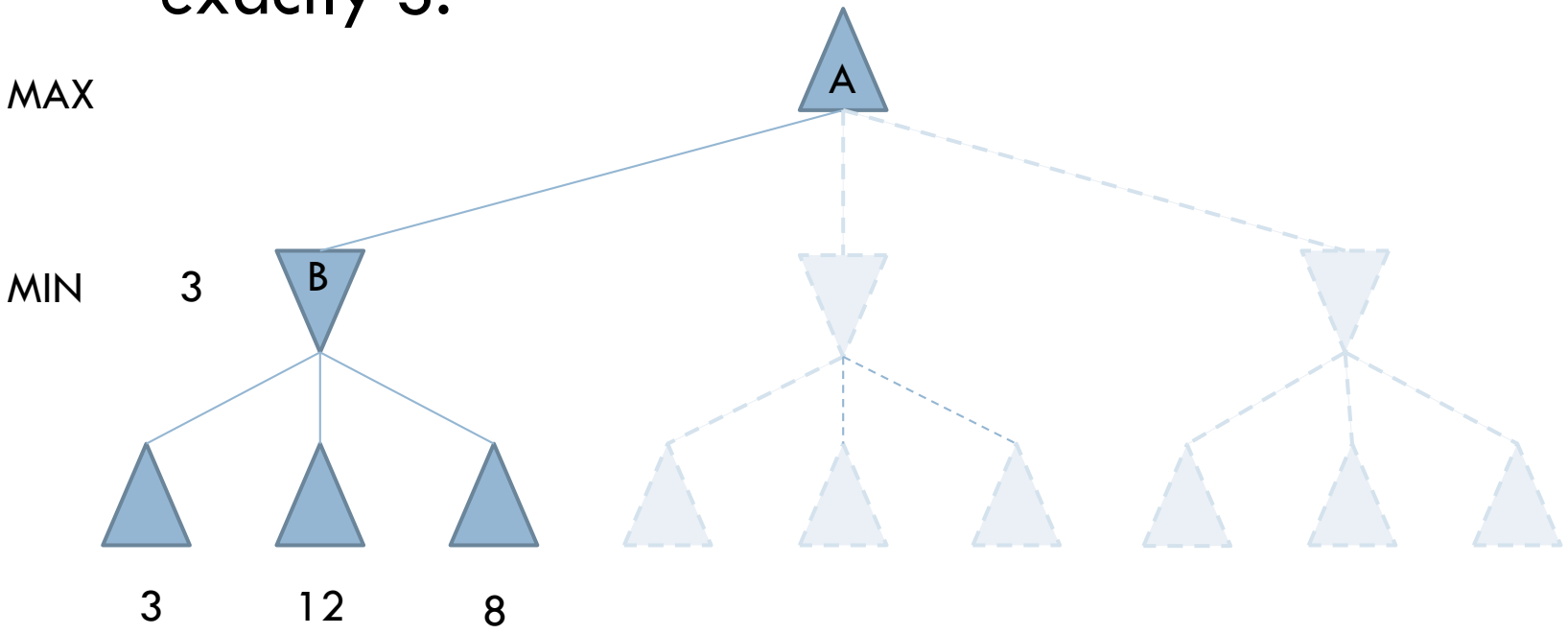
Alpha-Beta Pruning

- The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3.



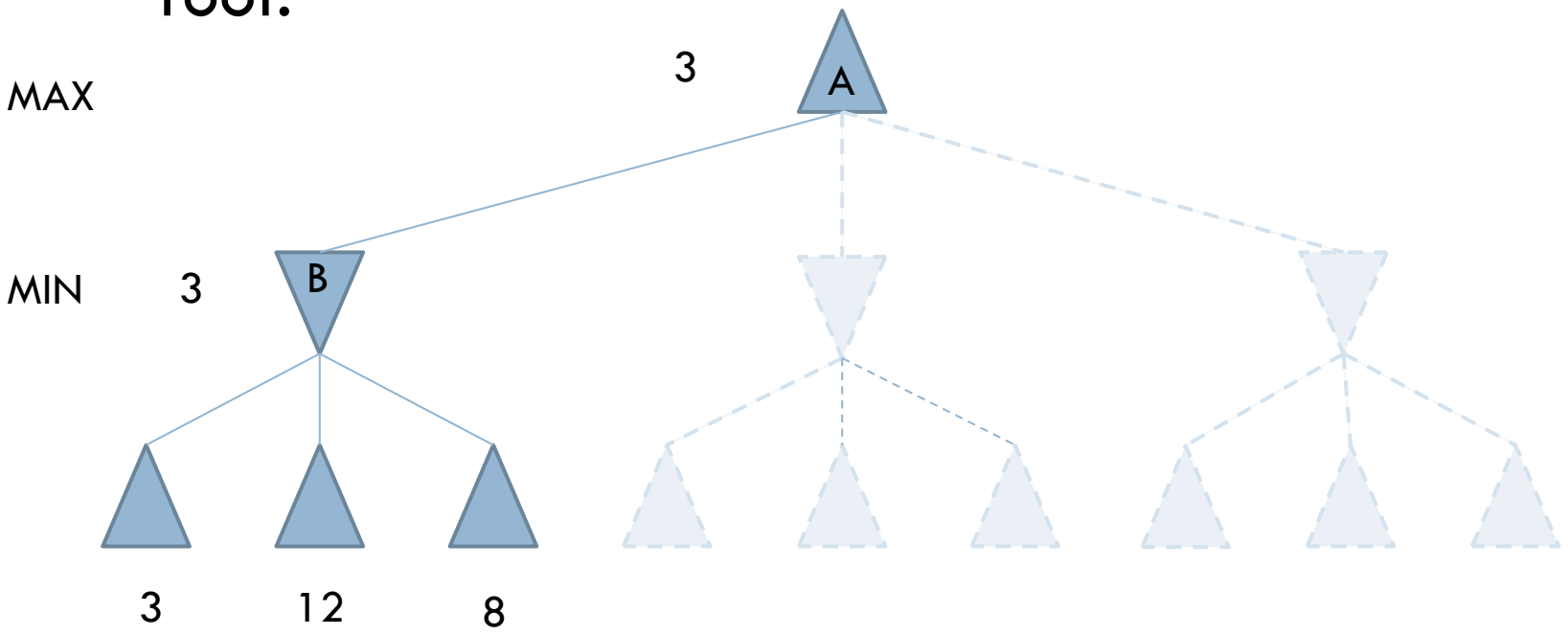
Alpha-Beta Pruning

- The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3.



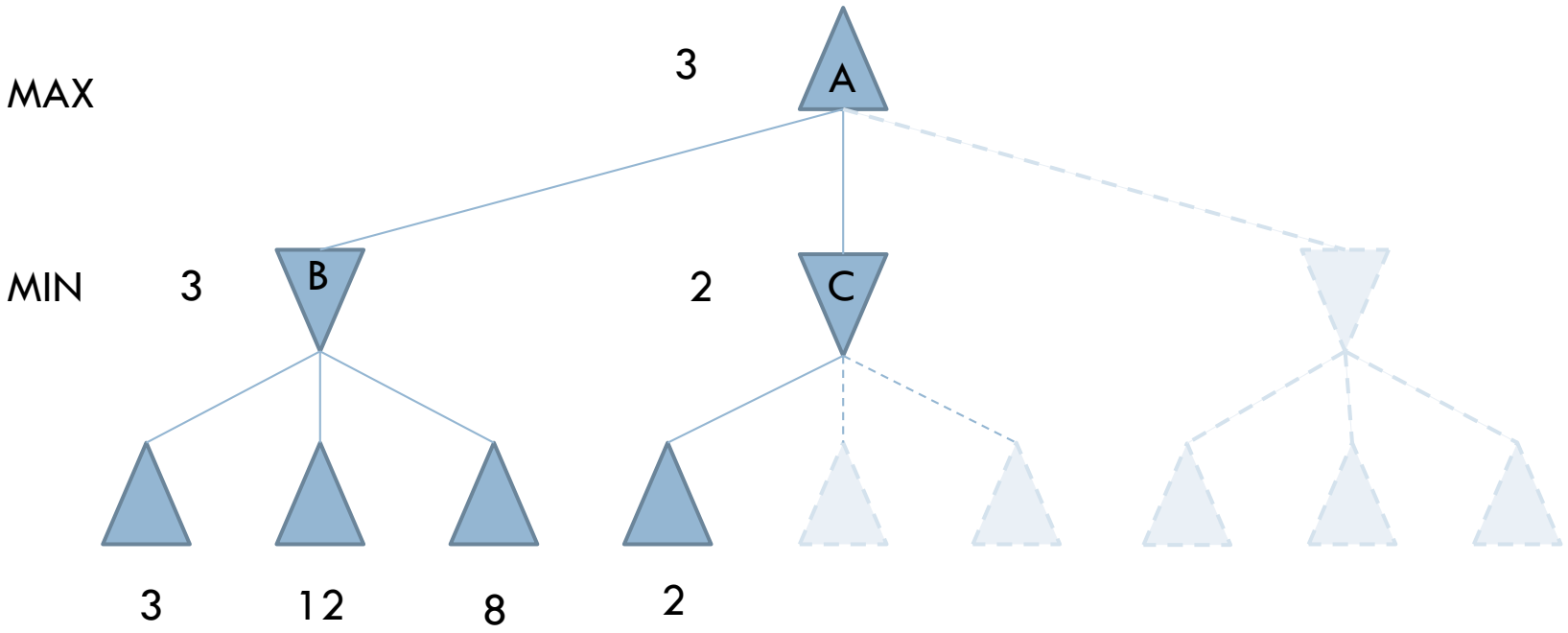
Alpha-Beta Pruning

- Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root.



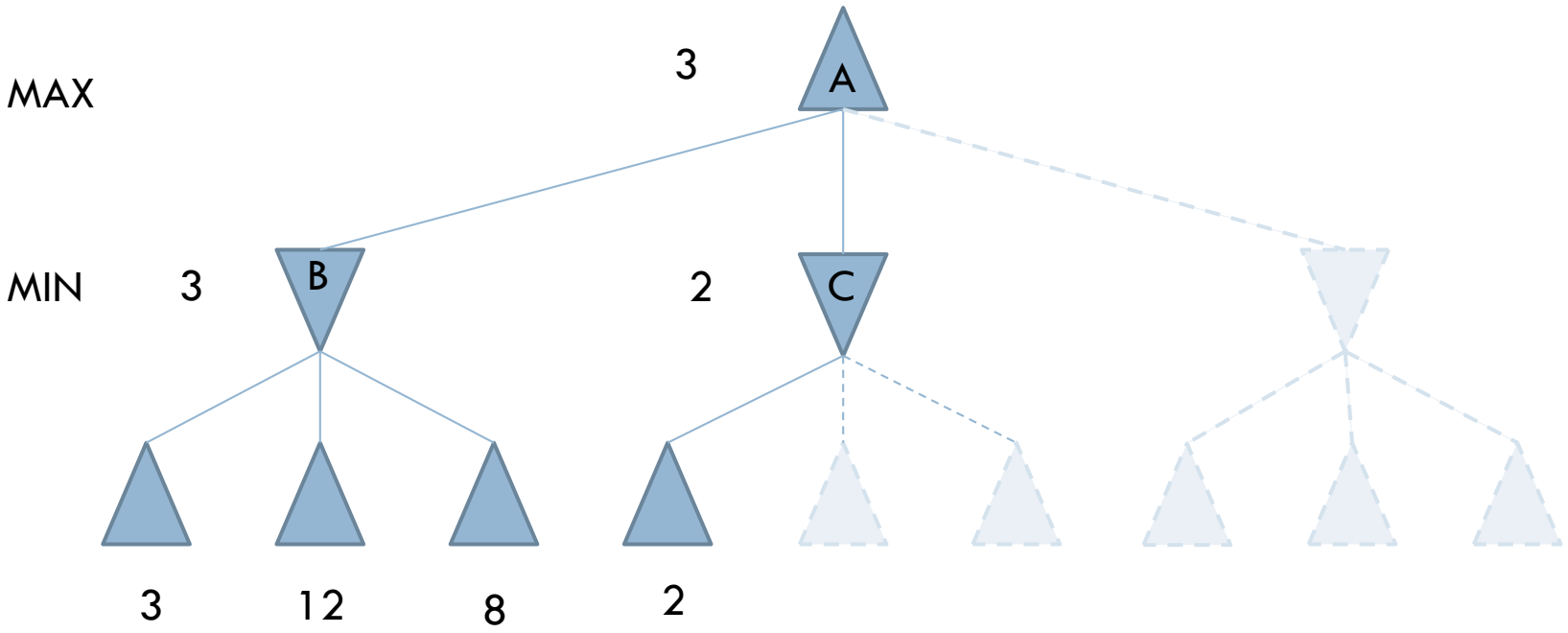
Alpha-Beta Pruning

- The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2.



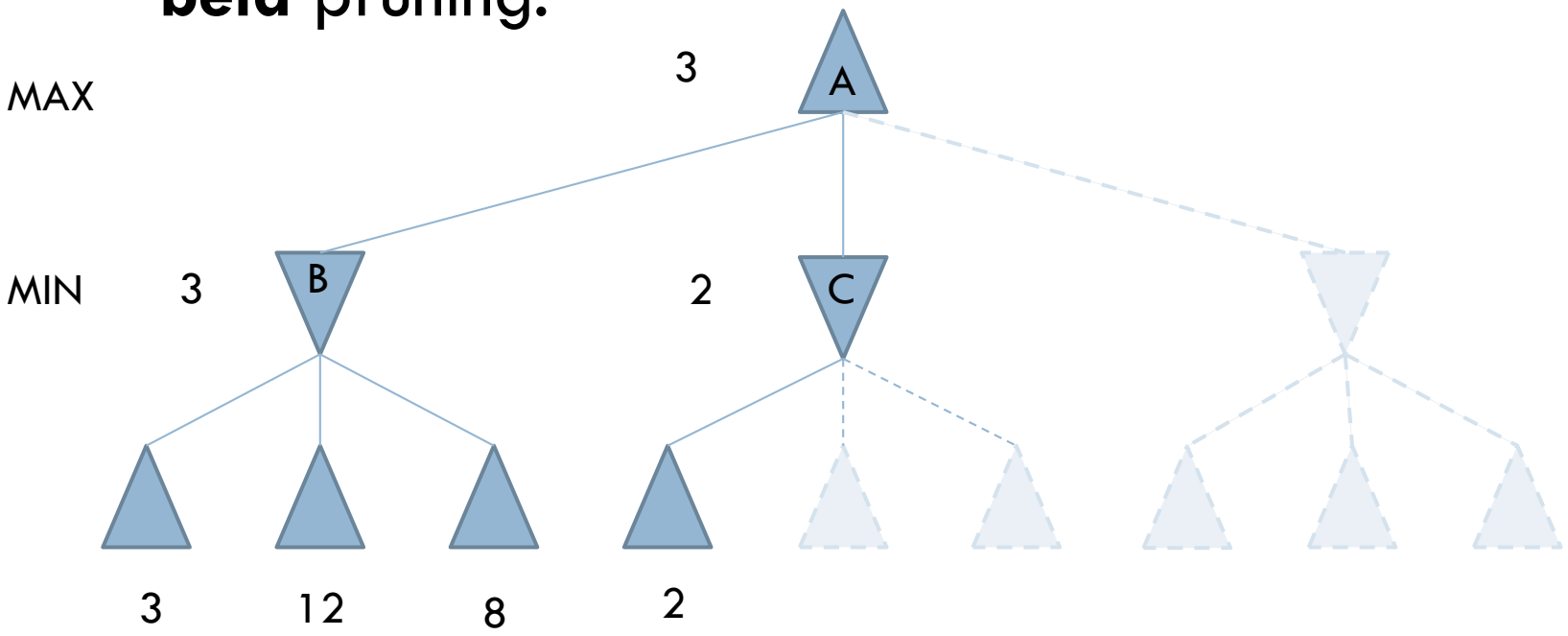
Alpha-Beta Pruning

- But we know that B is worth 3, so MAX would never choose C.



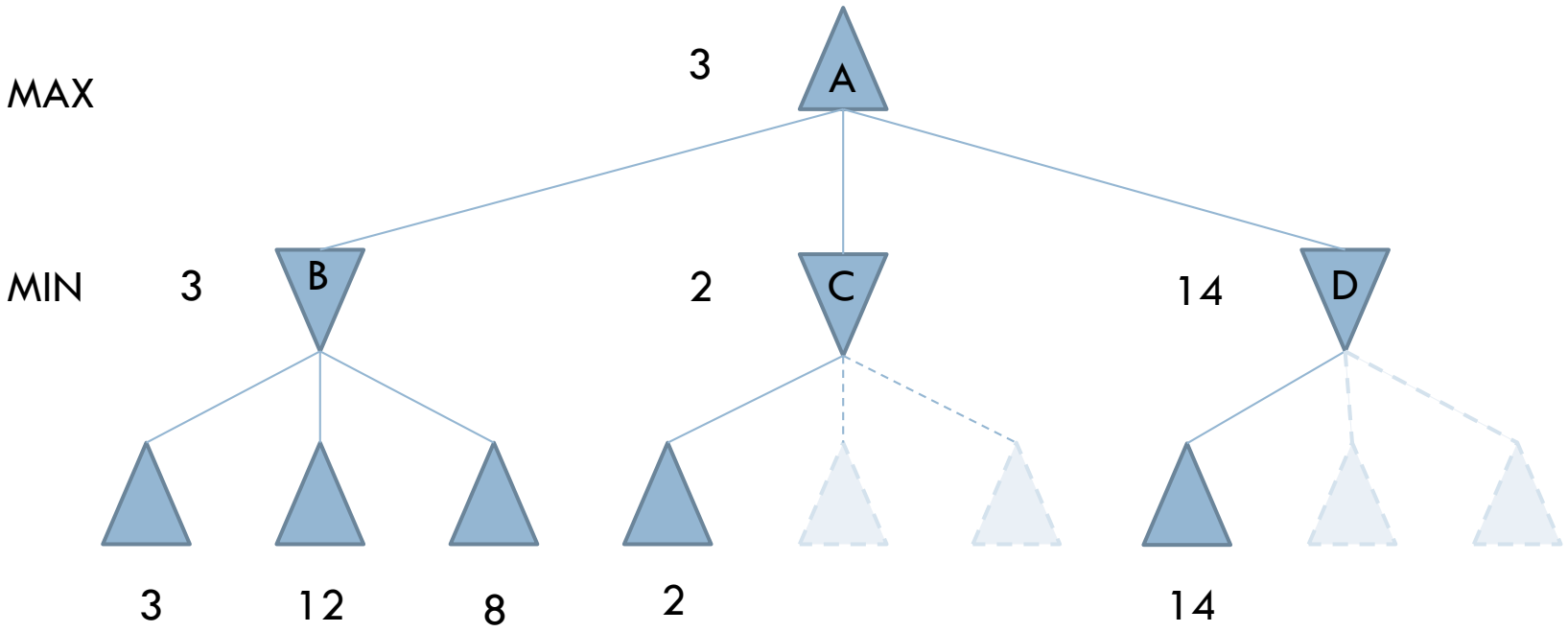
Alpha-Beta Pruning

- Therefore, there is no point in looking at the other successor states of C. This is an example of **alpha–beta** pruning.



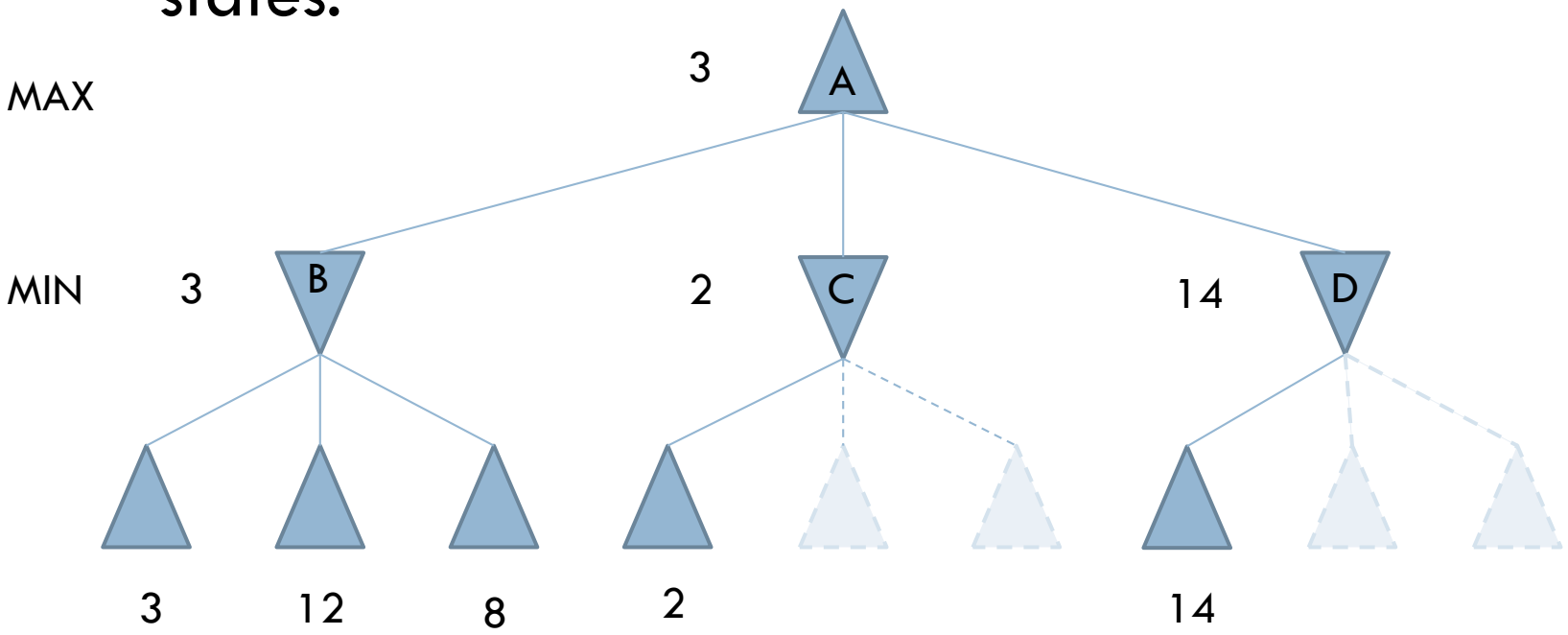
Alpha-Beta Pruning

- The first leaf below D has the value 14,so D is worth *at most* 14.



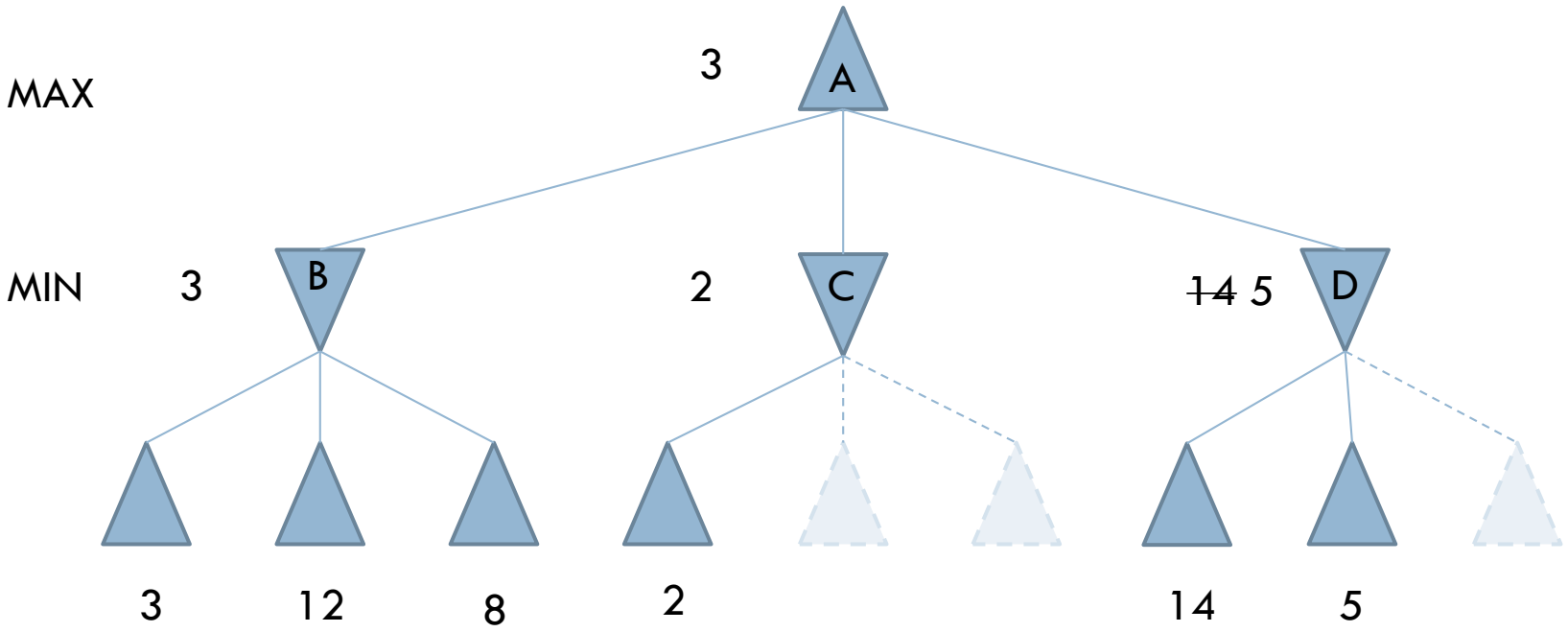
Alpha-Beta Pruning

- This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states.



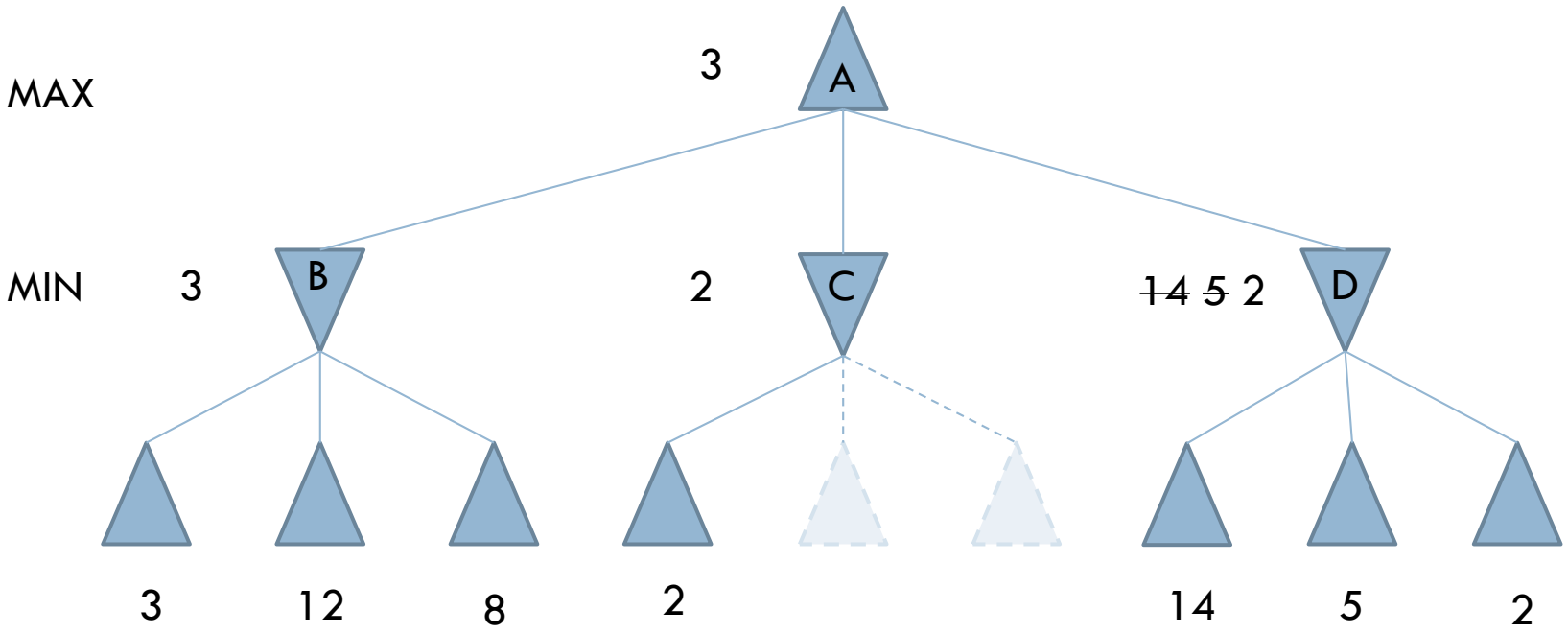
Alpha-Beta Pruning

- The second successor of D is worth 5, so again we need to keep exploring.



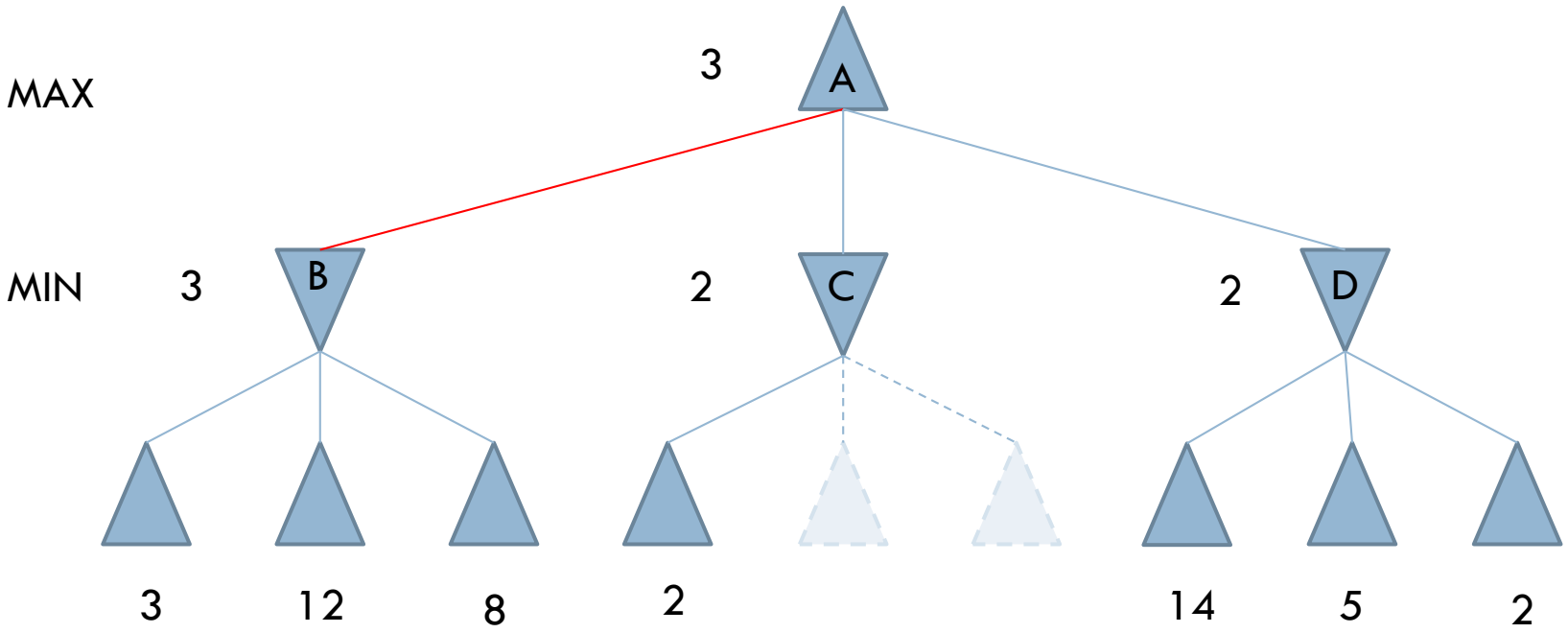
Alpha-Beta Pruning

- The third successor is worth 2, so now D is worth exactly 2.



Alpha-Beta Pruning

- MAX's decision at the root is to move to B, giving a value of 3.



Alpha-Beta Pruning

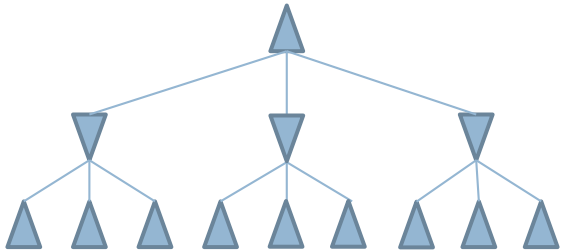
- Another way to look at this example:

MINIMAX(root) = max(min(3, 12, 8), min(2, x, y), min(14, 5, 2))
= max(3, min(2, x, y), 2)
= max(3, z, 2) where $z = \min(2, x, y) \leq 2$
= 3.

- In other words, the value of the root and hence the minimax decision are *independent* of the values of the pruned leaves x and y.

Alpha-Beta Pruning Implementation

- Implementation



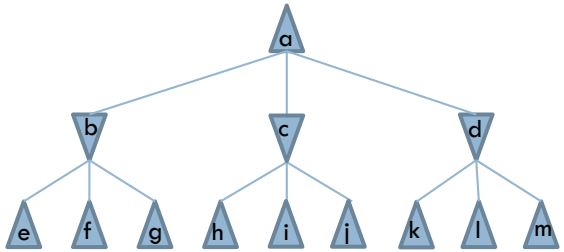
```
tree(a,[b,c,d]).
tree(b,[e,f,g]).
tree(c,[h,i,j]).
tree(d,[k,l,m]).

staticval(e,3).
staticval(f,12).
staticval(g,8).
staticval(h,2).
staticval(i,3).
staticval(j,20).
staticval(k,14).
staticval(l,5).
staticval(m,2).

min_to_move(X):-
             member(X,[b,c,d]).
max_to_move(X):-
             member(X,[a,e,f,g,h,i,j,k,l,m]).

moves(Pos, Moves) :-
  tree(Pos, Moves).
```

Alpha-Beta Pruning Implementation

- Implementation

```
tree(a,[b,c,d]).
tree(b,[e,f,g]).
tree(c,[h,i,j]).
tree(d,[k,l,m]).
```

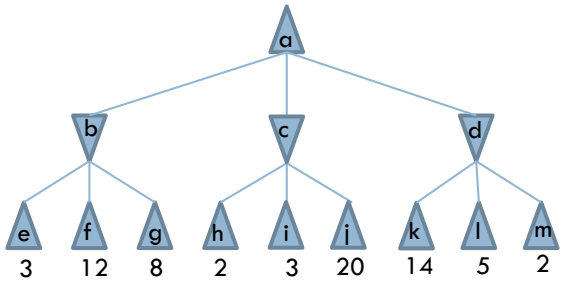
- Label the nodes
- Define successors




```
staticval(e,3).
staticval(f,12).
staticval(g,8).
staticval(h,2).
staticval(i,3).
staticval(j,20).
staticval(k,14).
staticval(l,5).
staticval(m,2).

min_to_move(X):-
            member(X,[b,c,d]).
max_to_move(X):-
            member(X,[a,e,f,g,h,i,j,k,l,m]).

moves(Pos, Moves) :-
  tree(Pos, Moves).
```

Alpha-Beta Pruning Implementation

- Implementation




```
tree(a,[b,c,d]).
tree(b,[e,f,g]).
tree(c,[h,i,j]).
tree(d,[k,l,m]).
```

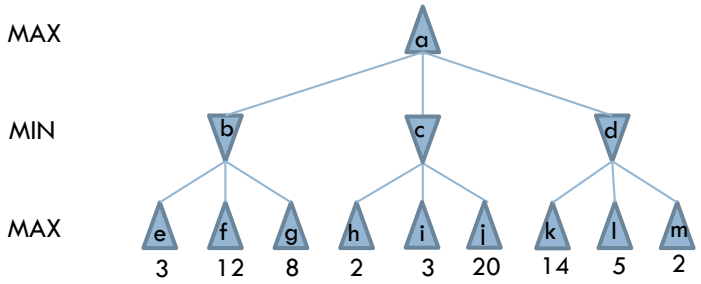
```
staticval(e,3).
staticval(f,12).
staticval(g,8).
staticval(h,2).
staticval(i,3).
staticval(j,20).
staticval(k,14).
staticval(l,5).
staticval(m,2).
```

- Define values for leafs.

```
min_to_move(X):-
            member(X,[b,c,d]).
max_to_move(X):-
            member(X,[a,e,f,g,h,i,j,k,l,m]).

moves(Pos, Moves) :-
  tree(Pos, Moves).
```

Alpha-Beta Pruning Implementation

- Implementation




```
tree(a,[b,c,d]).
tree(b,[e,f,g]).
tree(c,[h,i,j]).
tree(d,[k,l,m]).

staticval(e,3).
staticval(f,12).
staticval(g,8).
staticval(h,2).
staticval(i,3).
staticval(j,20).
staticval(k,14).
staticval(l,5).
staticval(m,2).

min_to_move(X):-
            member(X,[b,c,d]).
max_to_move(X):-
            member(X,[a,e,f,g,h,i,j,k,l,m]).

moves(Pos, Moves) :-
  tree(Pos, Moves).
```

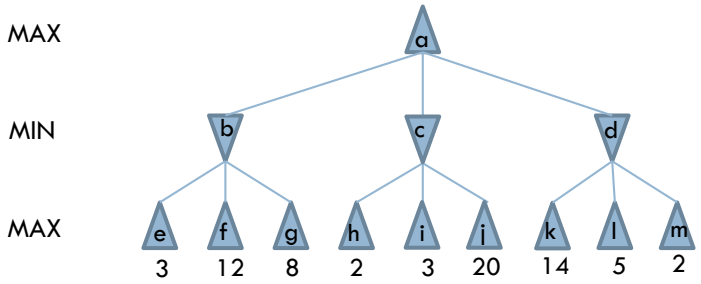
- Define MIN and MAX turns.

Alpha-Beta Pruning Implementation

- Implementation

```
tree(a,[b,c,d]).
tree(b,[e,f,g]).
tree(c,[h,i,j]).
tree(d,[k,l,m]).

staticval(e,3).
staticval(f,12).
staticval(g,8).
staticval(h,2).
staticval(i,3).
staticval(j,20).
staticval(k,14).
staticval(l,5).
staticval(m,2).

min_to_move(X):-
            member(X,[b,c,d]).
max_to_move(X):-
            member(X,[a,e,f,g,h,i,j,k,l,m]).
```

```
moves(Pos, Moves) :-
  tree(Pos, Moves).
```

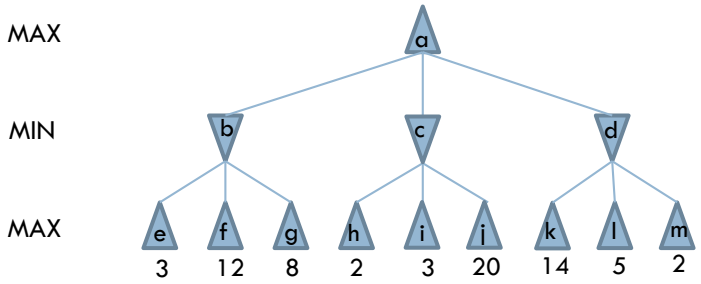


- Define valid moves.

Alpha-Beta Pruning Implementation

- Implementation

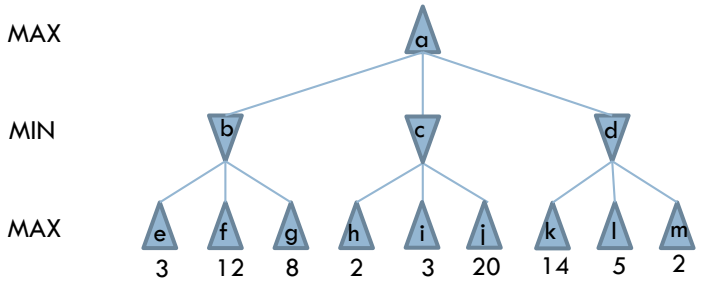
```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```

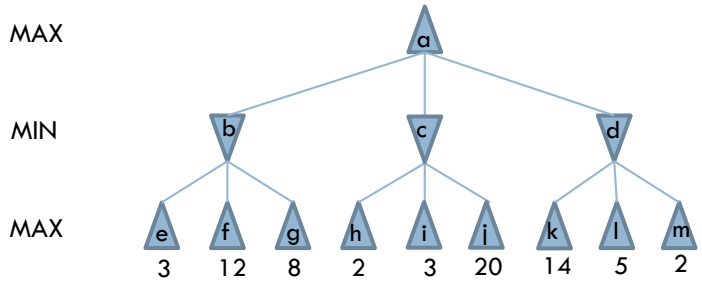



```
?-alphabeta(a, -1000, 1000, GoodPos, Val)
```

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



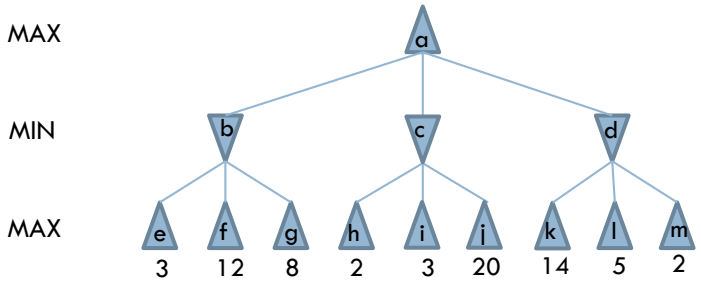

```
?-alphabeta(a, -1000, 1000, GoodPos, Val)
```

Alpha	Beta	GoodPos	Val
-1000	1000		

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



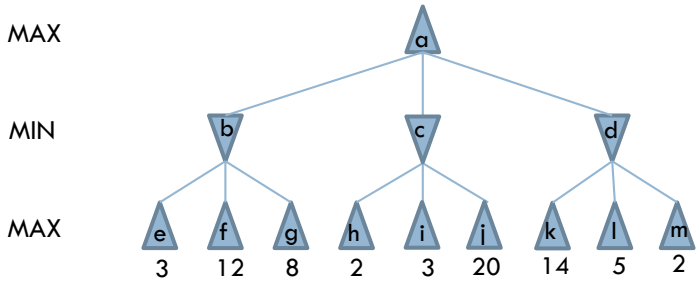
Are there successors or not?

Alpha	Beta	GoodPos	Val
-1000	1000		

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



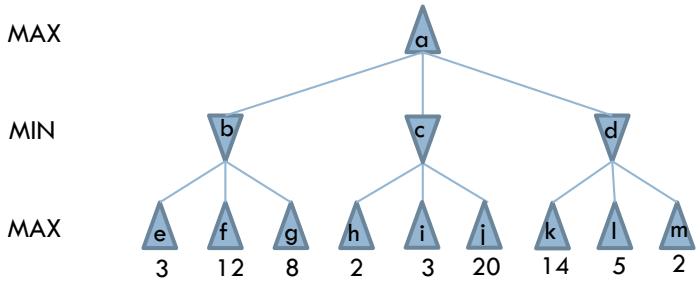
Search successors (PosList)

Alpha	Beta	GoodPos	Val
-1000	1000		

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



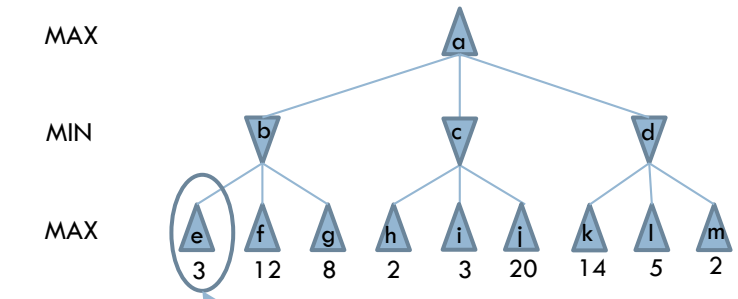
Search recursively until it finds the leaves and their value.

Alpha	Beta	GoodPos	Val
-1000	1000		

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



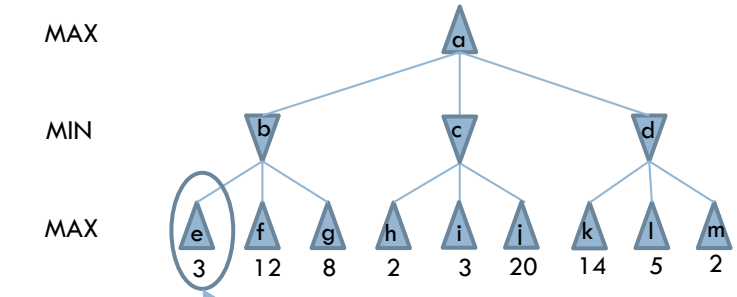
goodenough/7 finally executes when the the value of the first leaf is found.

Alpha	Beta	GoodPos	Val
-1000	1000		

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



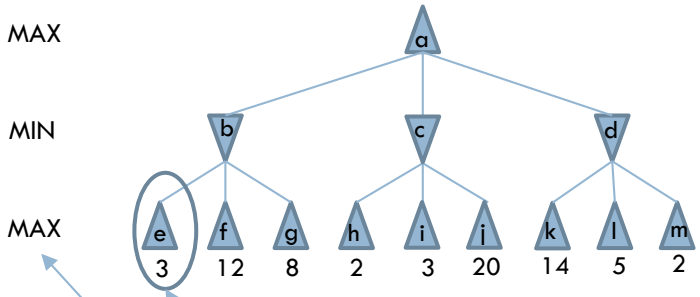
Pos = e, PosList = [f,g], Val = 3.

Alpha	Beta	GoodPos	Val
-1000	1000		3

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```

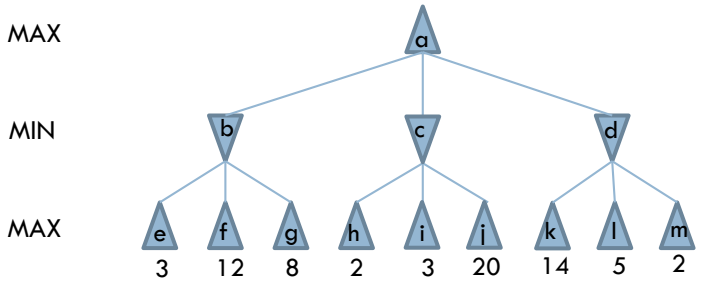


It is MAX turn but 3 < -1000 fails.

Alpha	Beta	GoodPos	Val
-1000	1000		3

Alpha-Beta Pruning Implementation

- Example




```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```

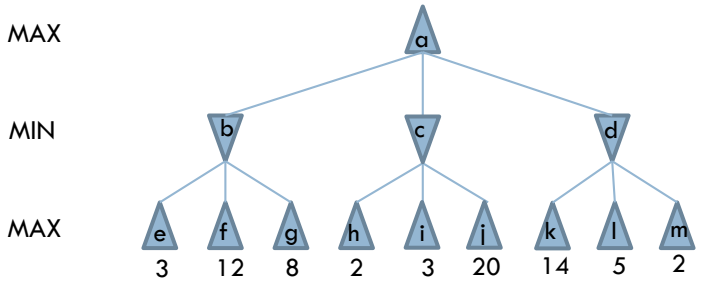
goodenough/7 has a third clause which calls newbounds/6 to compute new alpha and beta values.

Alpha	Beta	GoodPos	Val
-1000	1000		3

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



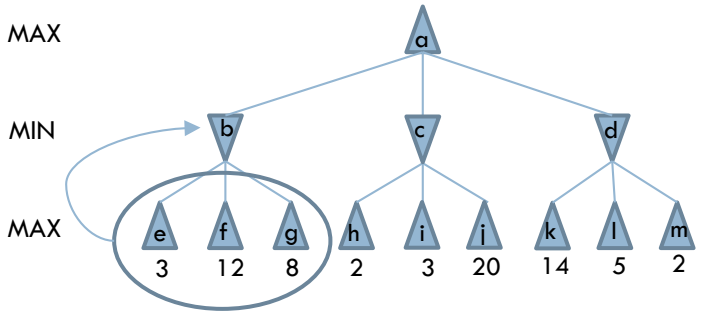
It is MAX turn and since 3 < 1000, the MIN level (b) will prefer this value to the previously instantiated.

Alpha	Beta	GoodPos	Val
-1000	1000		3

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



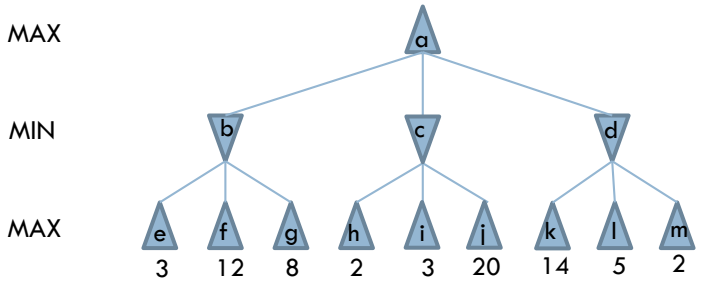
Recall that we are looking at the MAX level and computing what will the previous level (MIN) choose

Alpha	Beta	GoodPos	Val
-1000	3		3

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



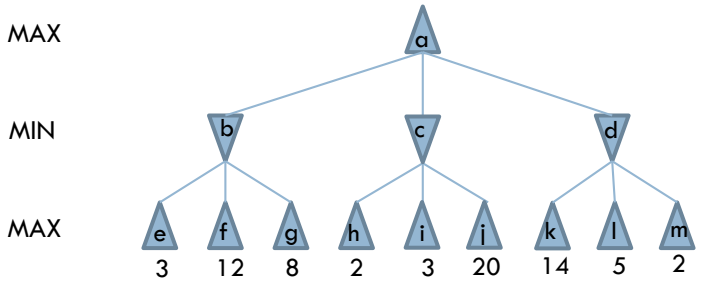
Execution will continue by looking at the values of f and g but both are higher than 3 and thus this value is kept.

Alpha	Beta	GoodPos	Val
-1000	3		3

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



At this point we have examined all the subtree bellow b, we can proceed to the analysis of the next element: c.

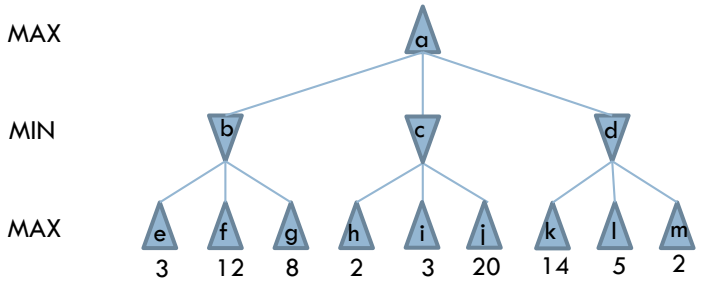
Alpha	Beta	GoodPos	Val
-1000	1000		3

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).
```

```
newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



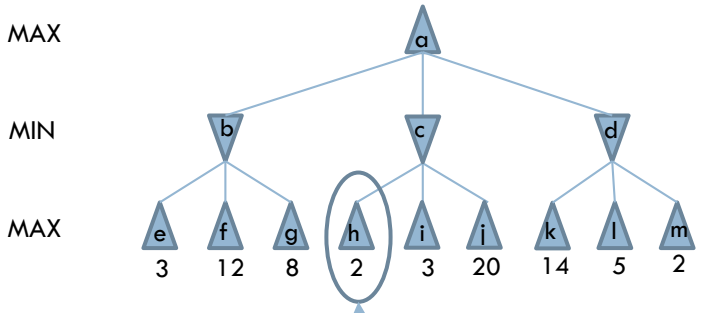
It is now MIN's turn and goodenough/7 will call newbounds/6 which computes Alpha = 3 since, Val (3) > Alpha (-1000)

Alpha	Beta	GoodPos	Val
3	1000		3

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



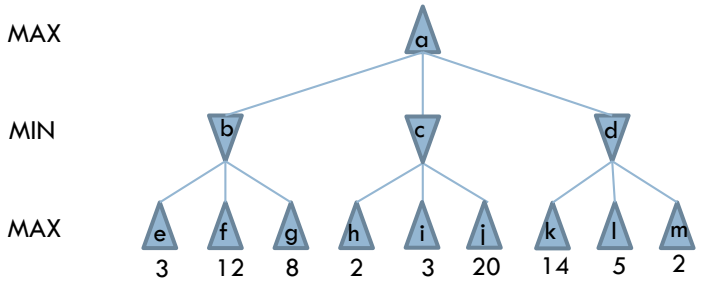
The program continues and is now analyzing the Pos = h with Val = 2 and PosList = [i,j]

Alpha	Beta	GoodPos	Val
3	1000		2

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



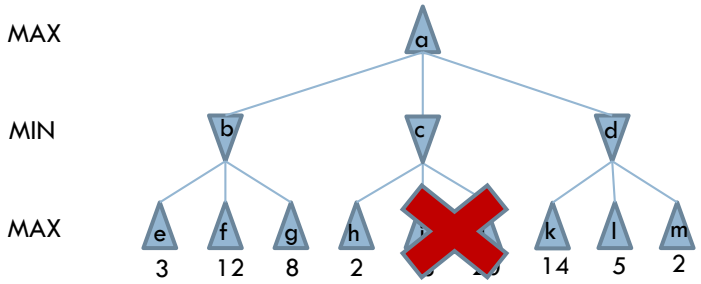
I tis now when pruning occurs! We found a value (2) which is worse than the Alpha which is 3

Alpha	Beta	GoodPos	Val
3	1000		2

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



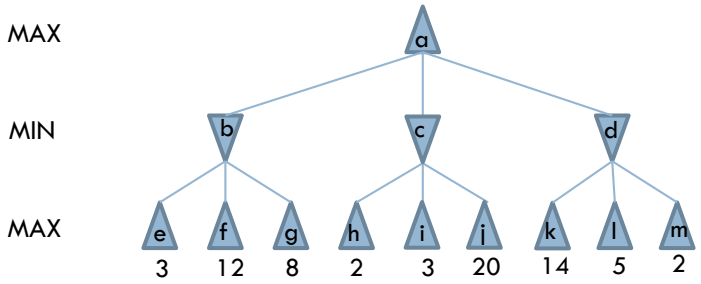
The cut discards the remaining clause of goodenough/7 and the remaining elements are not analyzed.

Alpha	Beta	GoodPos	Val
3	1000		

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```



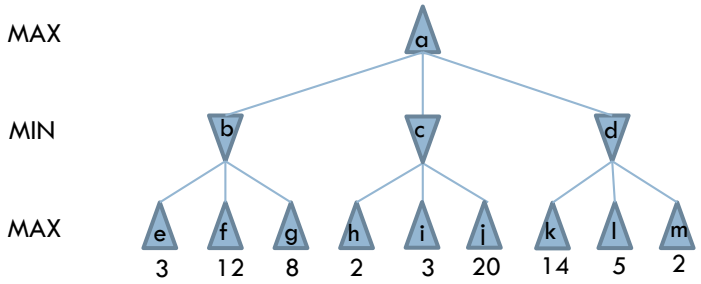
In the remaining of the execution d will be analyzed and Beta will have the values 14, 5 and 2.

Alpha	Beta	GoodPos	Val
3	~~14~~ ~~5~~ 2		

Alpha-Beta Pruning Implementation

- Example

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
  moves(Pos, PosList), !,
  boundedbest(PosList, Alpha, Beta, GoodPos, Val);
  staticval(Pos, Val).

boundedbest([Pos|PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta(Pos, Alpha, Beta, _, Val),
  goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !.
goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move(Pos), Val > Beta, !
  ;
  max_to_move(Pos), Val < Alpha, !.
goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
  newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta),
  boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
  min_to_move(Pos), Val > Alpha,!.
newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
  max_to_move(Pos), Val < Beta, !.
newbounds(Alpha, Beta, _, _, Alpha, Beta).
```

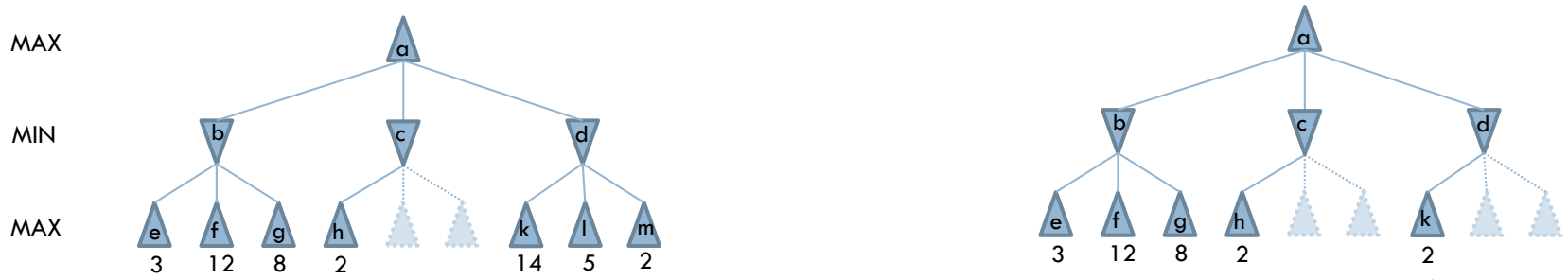


The execution ends with the following values of the main variables:

GoodPos	Val
b	3

Move ordering

- Effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined.



Different order => Different number of nodes that can be pruned.

Real Time Imperfect Decisions

- In the 1950's Shannon realized that it would be impossible to use Minimax to make a computer play chess due to time and space complexity of the game.
- He suggested replacing the utility function by an evaluation function.
- And the terminal test by a cutoff-test.

Evaluation Function

- Utility function can be replaced by an evaluation function:
  - Returns an estimate of the expected utility of the game from a given position.
  - The quality of the evaluation is critical for the performance of the game-playing program.

Evaluation Function

- Most evaluation functions compute separate numerical contributions from each feature of the game and then *combine* them to find the total value.
- Example, in chess one can give a value for each piece: each pawn is worth 1, a knight or bishop is worth 3, a rook 5, the queen 9, etc.
- Other features such as “good pawn structure” and “king safety” can also be used.

Evaluation Function

- These feature values are then simply added up to obtain the evaluation of the position:

$$EVAL(S) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

- Where each w_i is a weight and each f_i is a feature of the position.
- For chess, the f_i could be the numbers of each kind of piece on the board, and the w_i could be the values of the pieces.

Cutoff Test

- Terminal test could be replaced by a cutoff-test.
  - By using a fixed depth limit
  - The depth is chosen so time spent in calculating does not exceeds what the rules of the game allow.

Cutoff Test

- The cut-off can lead to errors due to the approximate nature of the evaluation function:
  - Suppose the program searches to the depth limit, reaching a position where Black is ahead by a knight and two pawns.
  - It would report this as the heuristic value of the state, thereby declaring that the state is a probable win by Black.
  - But White's next move captures Black's queen with no compensation.
  - Hence, the position is really won for White, but this can be seen only by looking ahead.

Search vs. Lookup

- Chess books describing good play in the opening and endgame are common.
- Many game-playing programs use *table lookup* rather than search for the opening and ending of games.
- The best advice of human experts on how to play each opening is copied from books and entered into tables for the computer's use.
- Near the end of the game there are again fewer possible positions, and thus more chance to do lookup.