Game

MinMax

Alpha-Beta

Motivation

- Why study games?
  - Games are fun!
  - Historical role in AI
  - Studying games teaches us how to deal with other agents trying to foil our plans
  - Huge state spaces – Games are hard!
  - Nice, clean environment with clear criteria for success

The Simple Case

- Chess, checkers, Tic-Tac-Toe, Othello, go …
- Two players alternate moves
- **Zero-sum**: one player’s loss is another’s gain
- **Perfect Information**: each player knows the entire game state
- **Deterministic**: no element of chance
- Clear set of legal moves
- Well-defined outcomes (e.g. win, lose, draw)

More complicated games

- Most card games (e.g. Hearts, Bridge, “Belot” …etc.) and Scrabble
  - non-deterministic
  - lacking in perfect information.
- Cooperative games

Game setup

- **Two players**: A and B
- **A** moves first and they take turns until the game is over. Winner gets award, loser gets penalty.
- **Games as search:**
  - **Initial state**: e.g. board configuration of chess
  - **Successor function**: list of (move,state) pairs specifying legal moves.
  - **Goal test**: Is the game finished?
  - **Utility function**: Gives numerical value of terminal states. E.g. win (+1), lose (-1) and draw (0) in tic-tac-toe

- **A** uses **search tree** to determine next move.

How to Play a Game by Searching

- **General Scheme**
  - Consider all legal moves, each of which will lead to some new state of the environment (‘board position’)
  - **Evaluate** each possible resulting board position
  - Pick the move which leads to the best board position.
  - Wait for your opponent’s move, then **repeat**.

- **Key problems**
  - Representing the ‘board’
  - Representing legal next boards
  - Evaluating positions
  - Looking ahead

Game Trees

- Represent the problem space for a game by a tree
  - **Nodes** represent ‘board positions’ (state)
  - **edges** represent legal moves.
- **Root node** is the position in which a decision must be made.
- **Evaluation function *f*** assigns real-number scores to `board positions.’
- **Terminal nodes (leaf)** represent ways the game could end, labeled with the desirability of that ending (e.g. win/lose/draw or a numerical score)

MAX & MIN Nodes

- When I move, I attempt to **MAXimize** my performance.
- When my opponent moves, he attempts to **MINimize** my performance.

TO REPRESENT THIS:

- If we move first, label the root MAX; if our opponent does, label it MIN.
- Alternate labels for each successive tree level.
  - if the root (level 0) is our turn (MAX), all even levels will represent turns for us (MAX), and all odd ones turns for our opponent (MIN).

Evaluation functions

- Evaluations how good a ‘board position’ is
  - Based on static features of that board alone
- Zero-sum assumption lets us use one function to describe goodness for both players.
  - f(n)>0 if we are winning in position n
  - f(n)=0 if position n is tied
  - f(n)<0 if our opponent is winning in position n
- Build using expert knowledge (Heuristic),
  - Tic-tac-toe: f(n)=(# of 3 lengths possible for me)
    - (# possible for you)

Chess Evaluation Functions

- **Alan Turing's**
  f(n)=(sum of your piece values)- (sum of opponent's piece values)

- More complex: weighted sum of positional features:

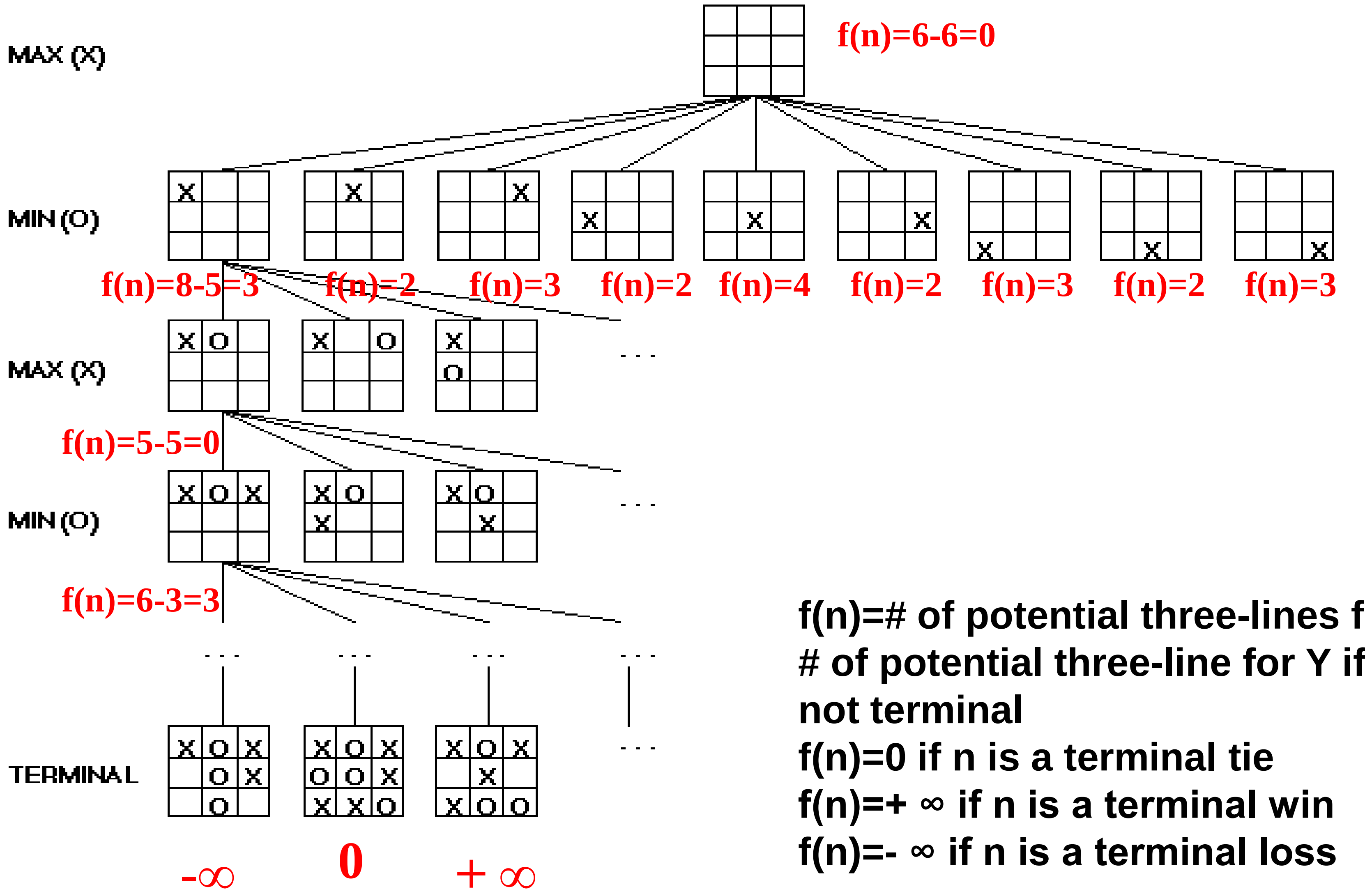
$$\sum w_i \, feature_i(n)$$

- Deep Blue has > 8000 features
  (IBM Computer vs. Gary Kasparov)

Pawn	1.0
Knight	3.0
Bishop	3.25
Rook	5.0
Queen	9.0

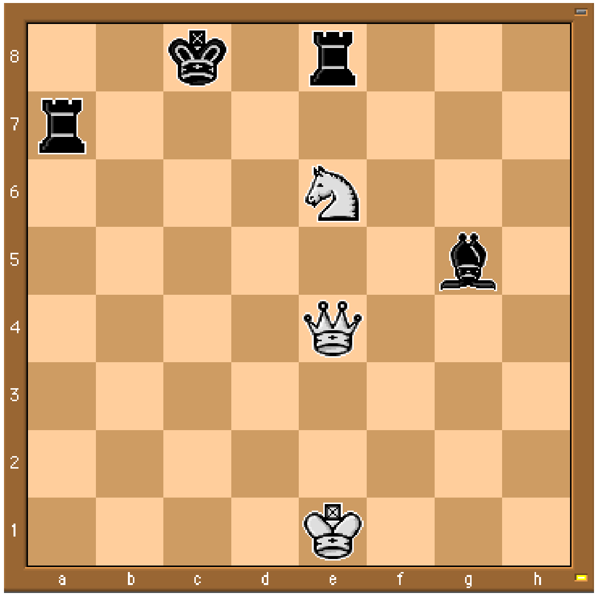
Pieces values for a simple Turing-style evaluation function

A Partial Game Tree for Tic-Tac-Toe

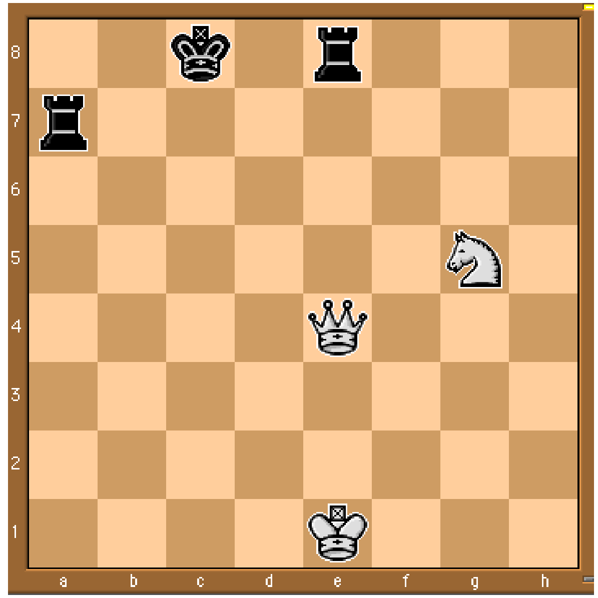


f(n)=# of potential three-lines for X – # of potential three-line for Y if n is not terminal
f(n)=0 if n is a terminal tie
f(n)=+ ∞ if n is a terminal win
f(n)=- ∞ if n is a terminal loss

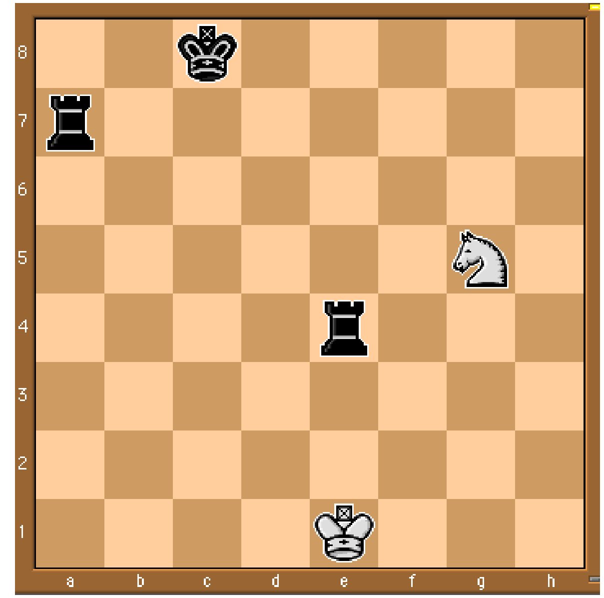
Some Chess Positions and their Evaluations



White to move
f(n)=(9+3)-(5+5+3.25)
=-1.25



… Nxg5??
f(n)=(9+3)-(5+5)
=2



Uh-oh: Rxg4+
f(n)=(3)-(5+5)
=-7
And black may force checkmate

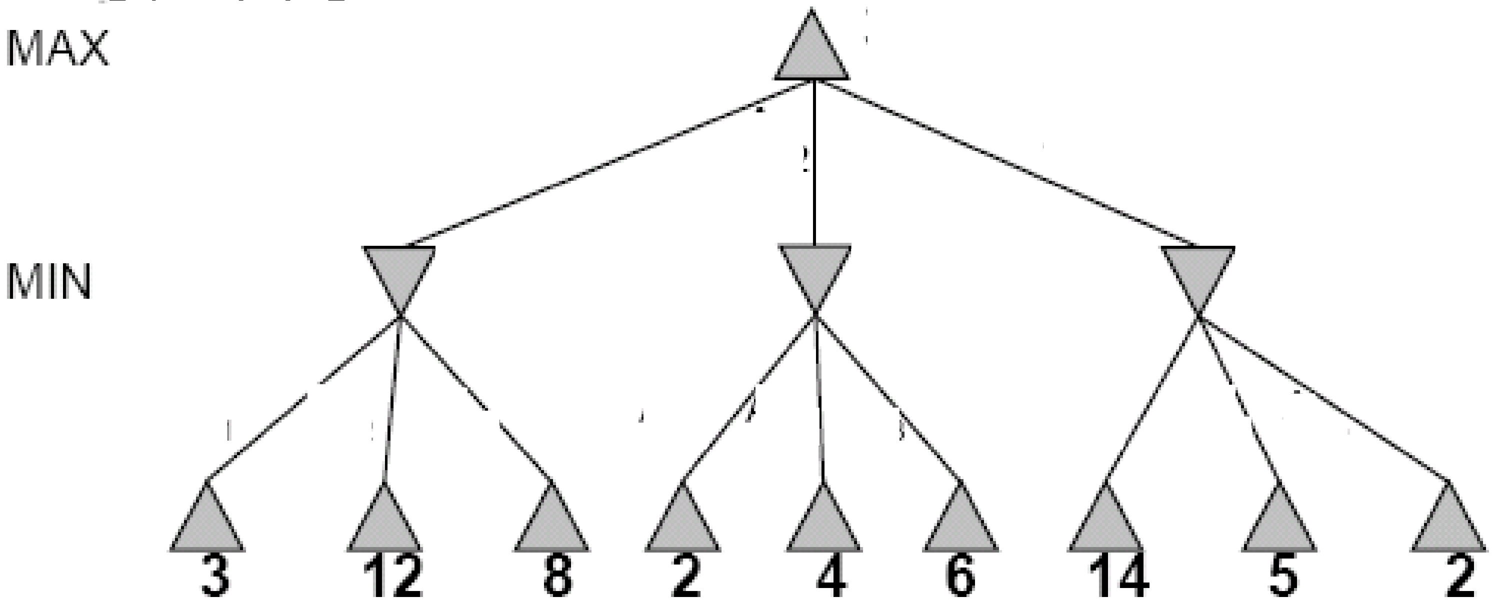
So, considering our opponent's possible responses would be wise.

MinMax Algorithm

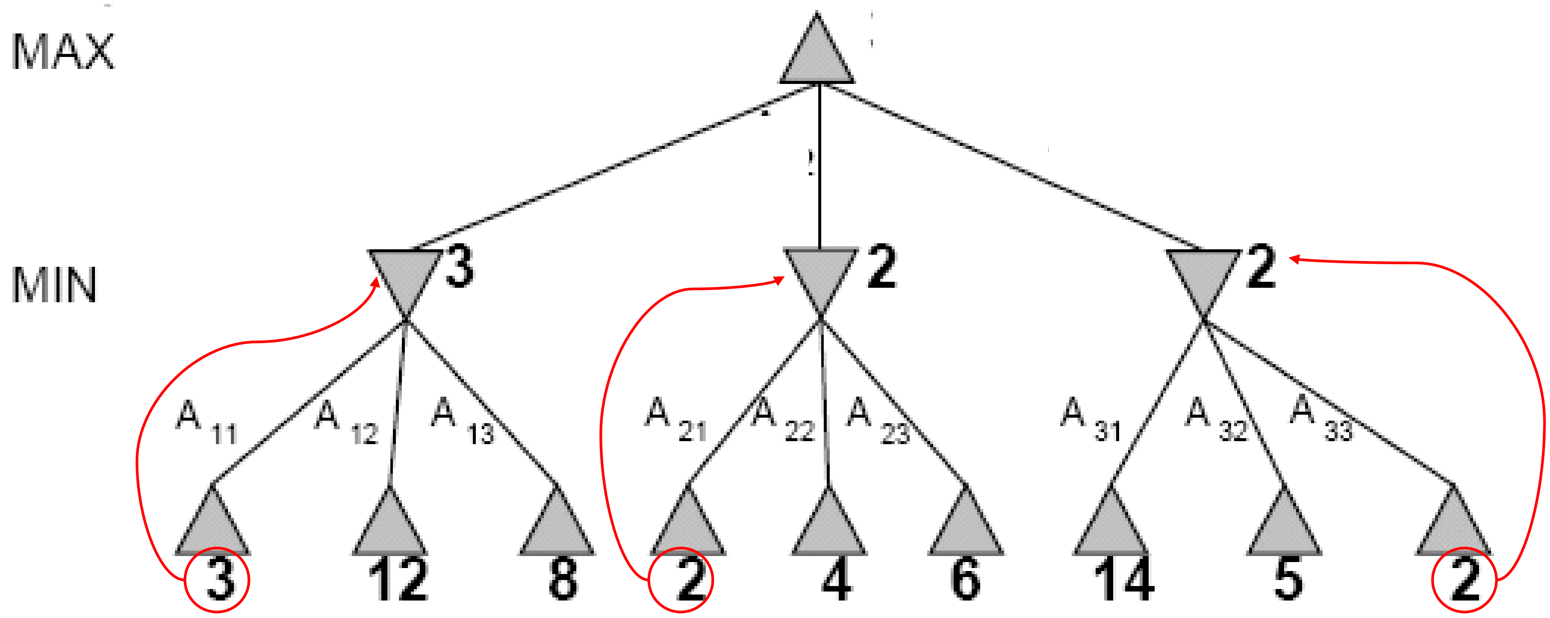
- Two-player games with perfect information, the **minmax** can determine the best move for a player by enumerating (evaluating) the entire game tree.
- Player 1 is called Max
  - Maximizes result
- Player 2 is called Min
  - Minimizes opponent’s result

MinMax Example

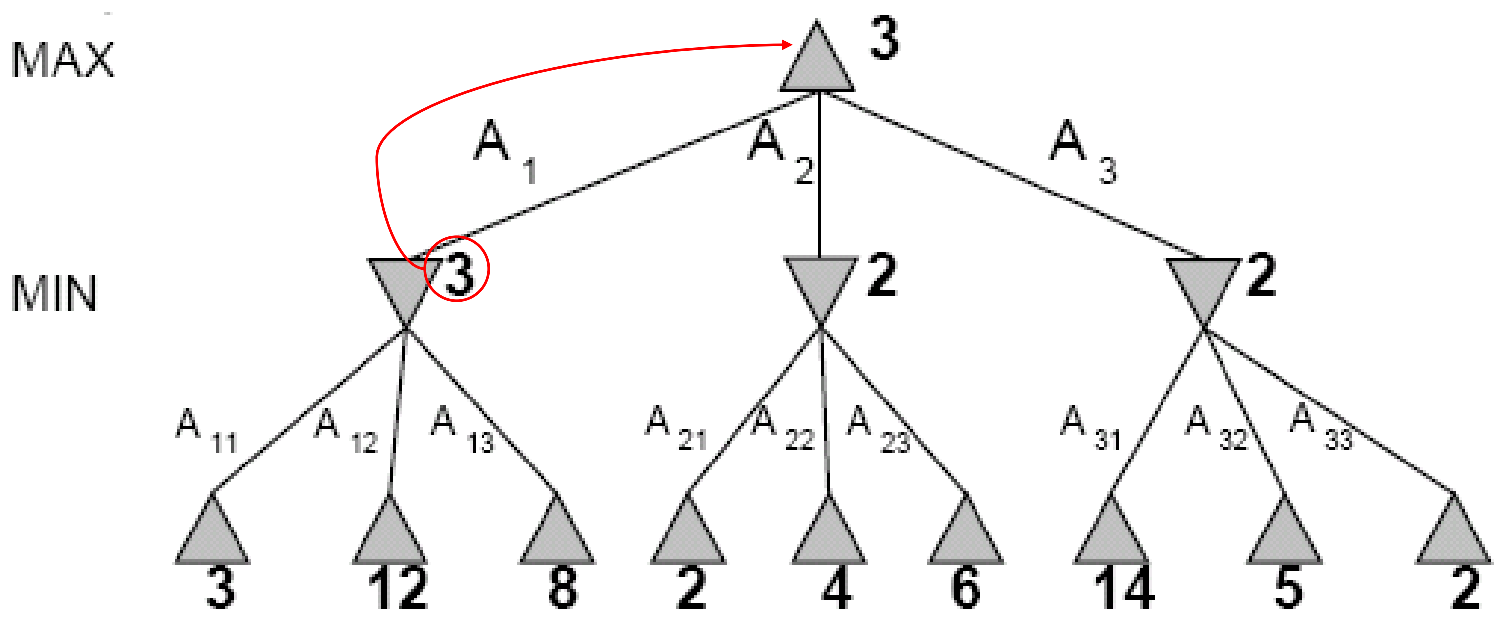
- Perfect play for deterministic games
-
- Idea: choose move to position with highest minimax value
  = best achievable payoff against best play
-
- E.g., 2-ply game:



Two-Ply Game Tree



Two-Ply Game Tree



MinMax steps

```
Int MinMax (state s, int depth, int type)
{
   if( terminate(s)) return Eval(s);
   if( type == max )
   {
        for ( child =1, child <= NmbSuccessor(s); child++)
        {
                value = MinMax(Successor(s, child), depth+1, Min)
                if( value > BestScore) BestScore = Value;
        }
   }
   if( type == min )
   {
        for ( child =1, child <= NmbSuccessor(s); child++)
        {
                value = MinMax(Successor(s, child), depth+1, Max)
                if( value < BestScore) BestScore = Value;
        }
   }
   return BestScore;
}
```

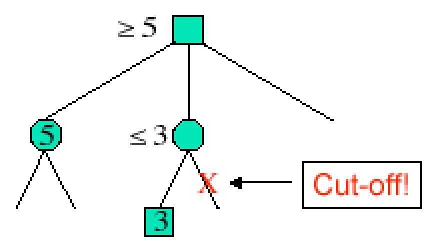
MinMax Analysis

- Time Complexity: $O(b^{d)}$ ☹
- Space Complexity: $O(b*d)$ ☺
- Optimality: Yes ☺

Problem: Game → *Resources Limited!*

- Time to make an action is limited

Some nodes in the search can be *proven* to be irrelevent to the outcome of the search



- Can we do better ? Yes !
- How ? Cutting useless branches !

How do we deal with resource limits?

- **Evaluation function**: return an estimate of the expected utility of the game from a given position, i.e.:
  - **Generate Search Tree up to certain level (i.e.: 4)**

- **Alpha-beta pruning**:
  - return appropriate minimax decision without exploring entire tree

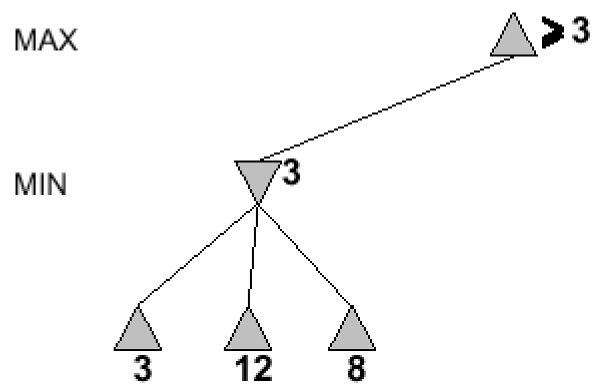
Alpha-Beta Algorithm

- It is based on process of eliminating a branch of the search tree “pruning” the search tree.

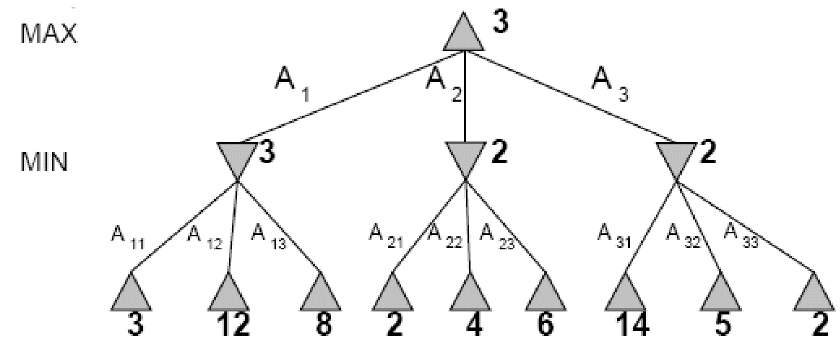
- It is applied as standard minmax tree:

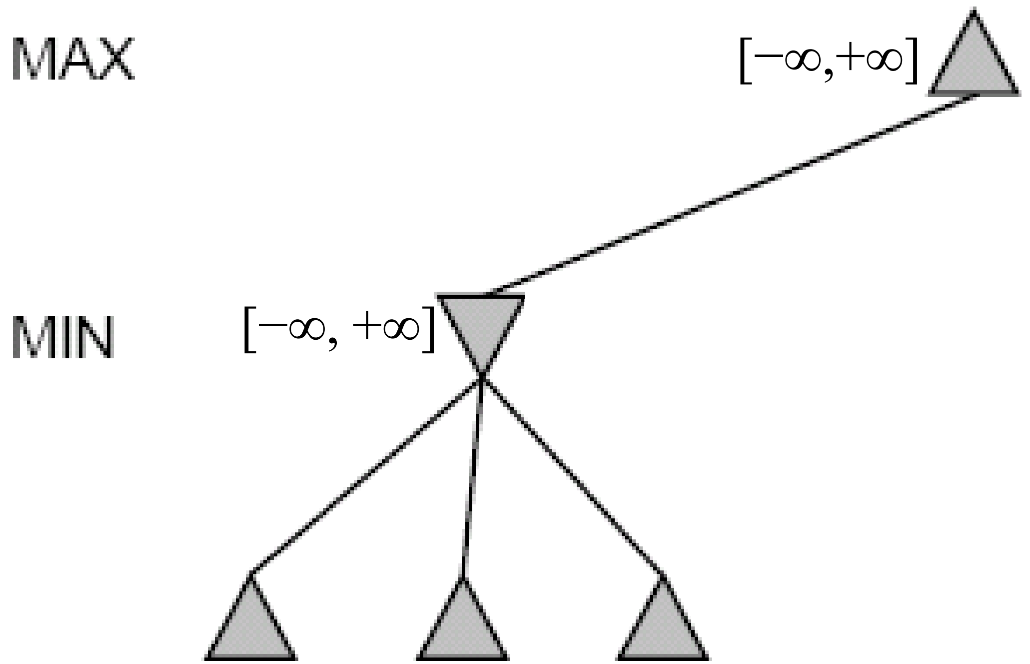
  – it returns the same move as minimax
  – prunes away branches that are not necessary to the final decision.

α-β Example

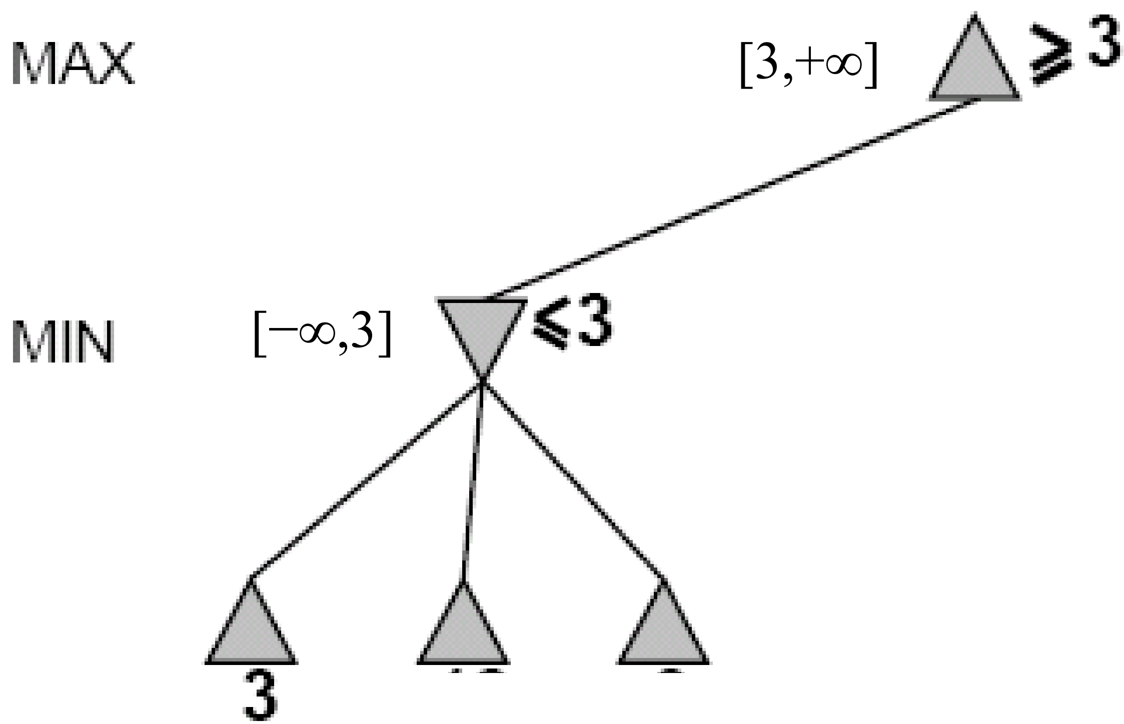


Alpha-Beta

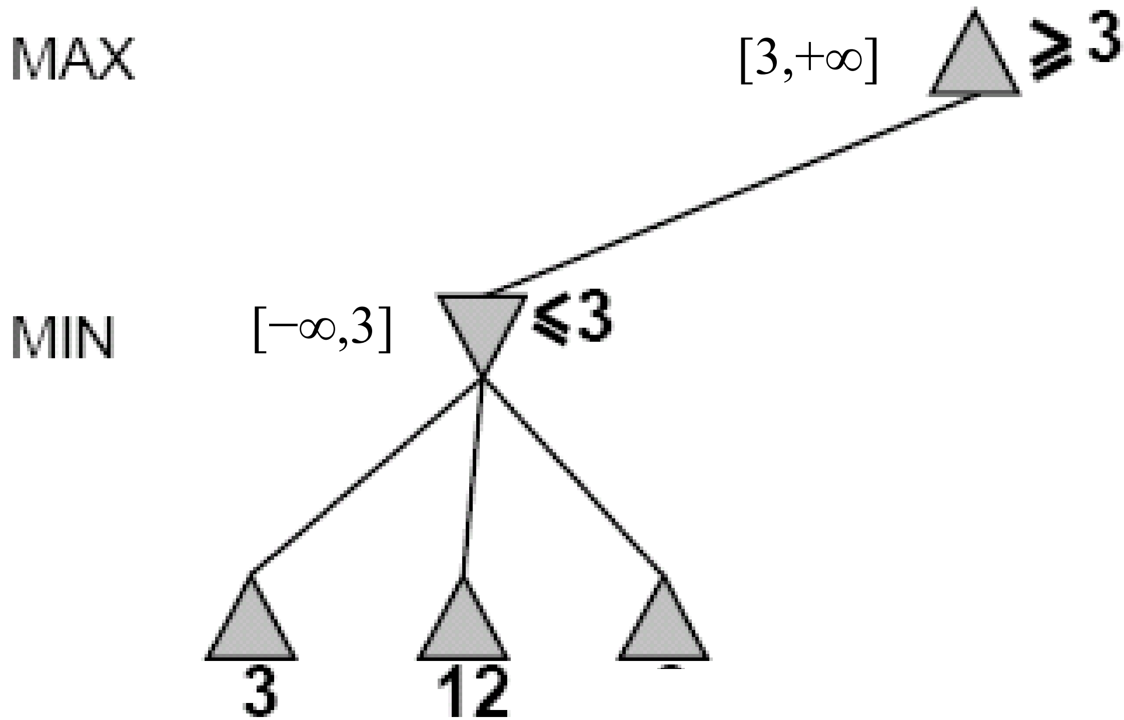

MAX
MIN
3
A 1
A 2
A 3
3
2
2
A 11
A 12
A 13
A 21
A 22
A 23
A 31
A 32
A 33
3
12
8
2
4
6
14
5
2


MAX
MIN
[−∞,+∞]
[−∞, +∞]

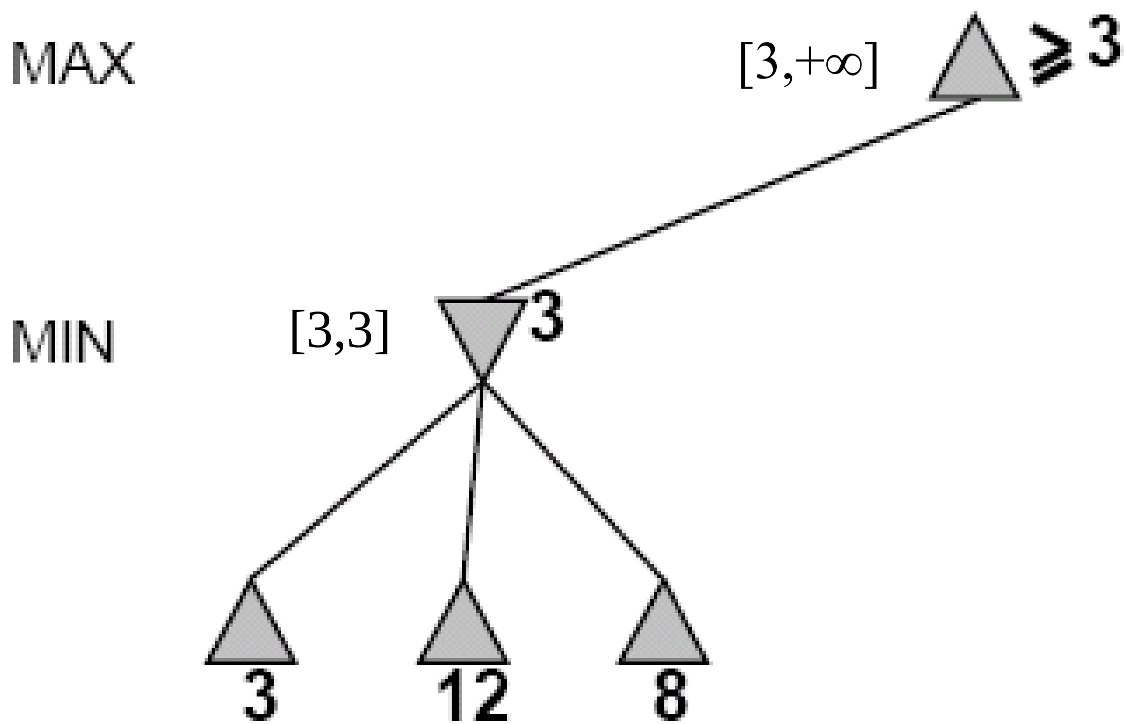
Alpha-Beta Example (continued)



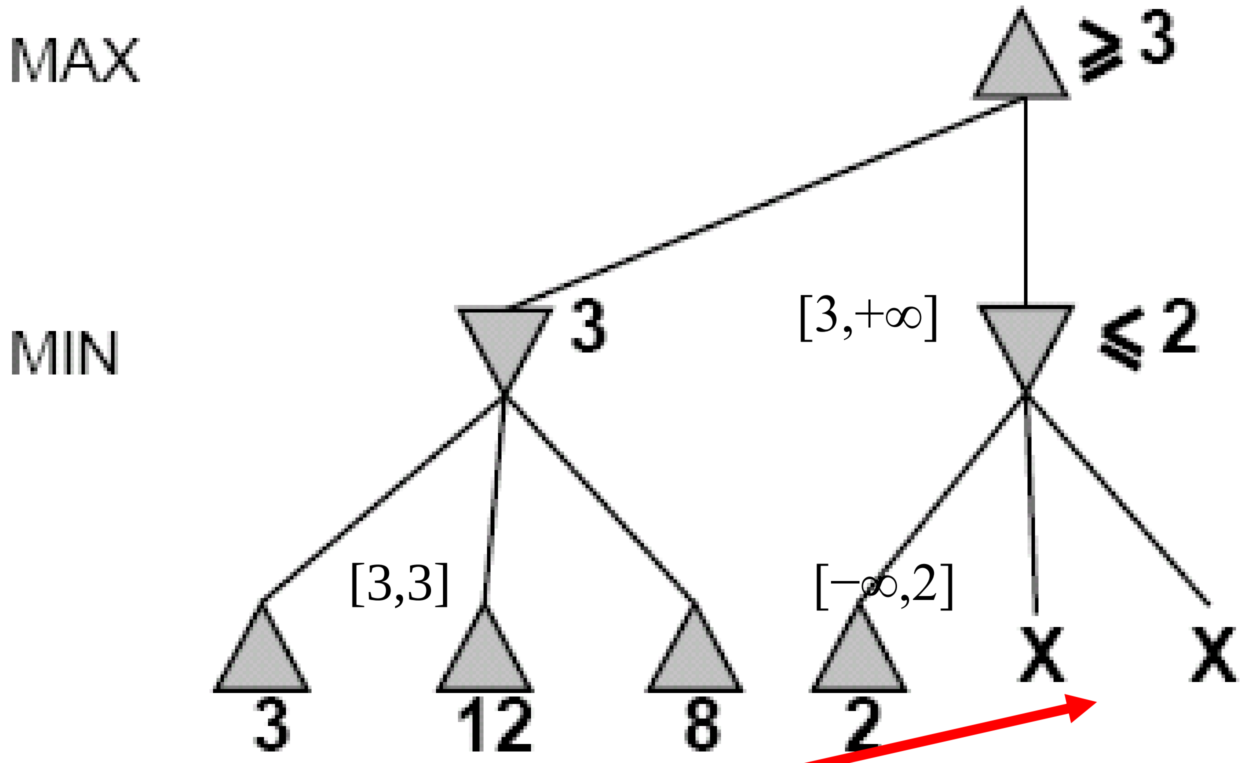
Alpha-Beta Example (continued)



Alpha-Beta Example (continued)

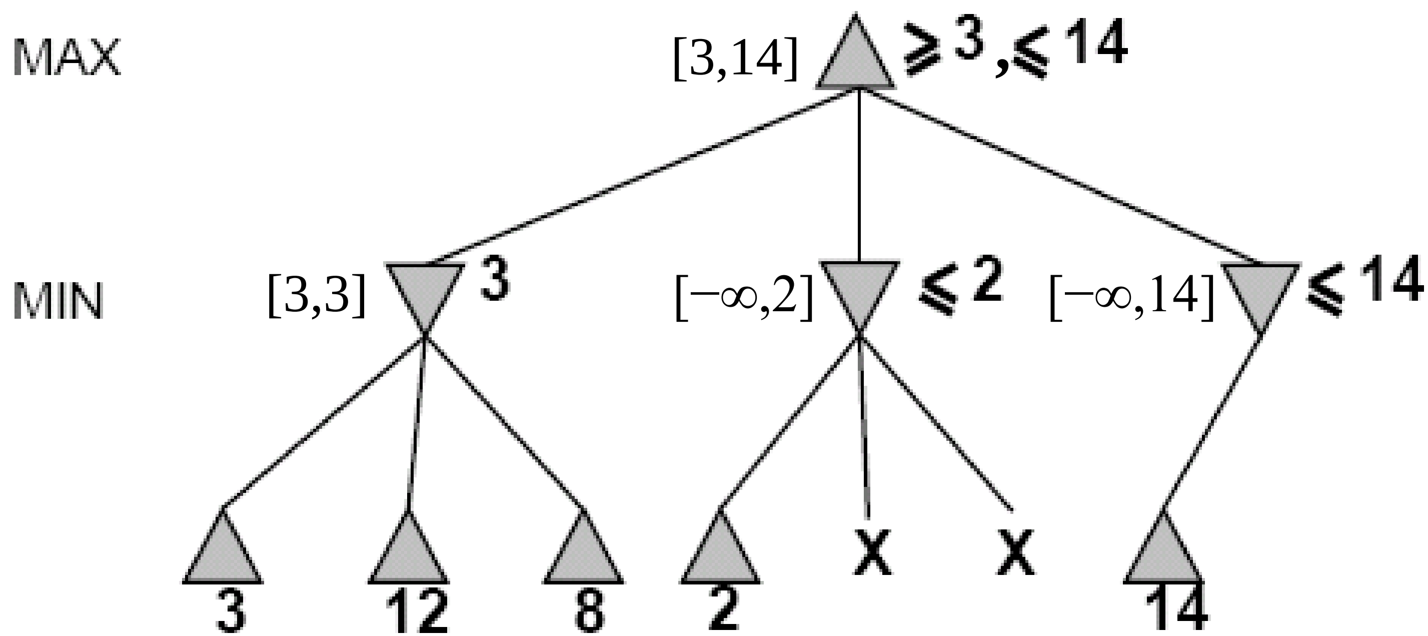


Alpha-Beta Example (continued)

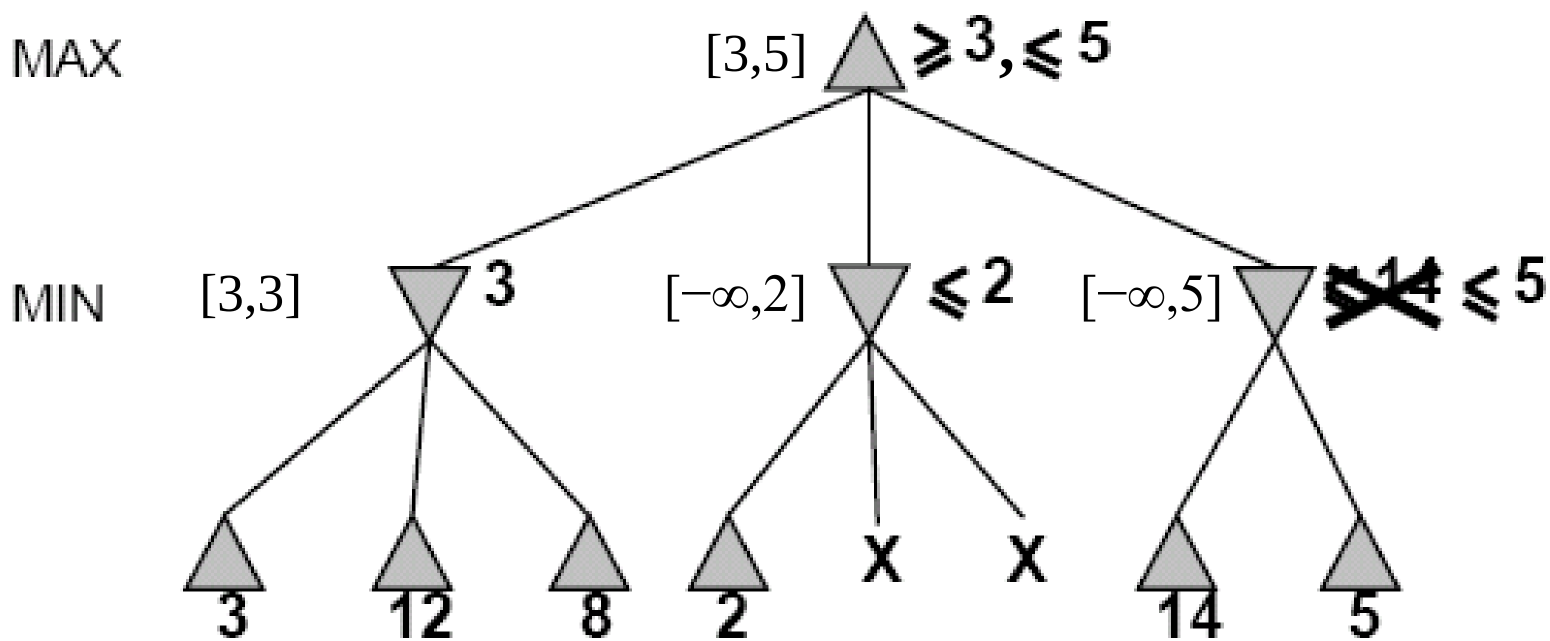


- **We don't need to compute the value at this node.**
- **No matter what it is it can't effect the value of the root node.**

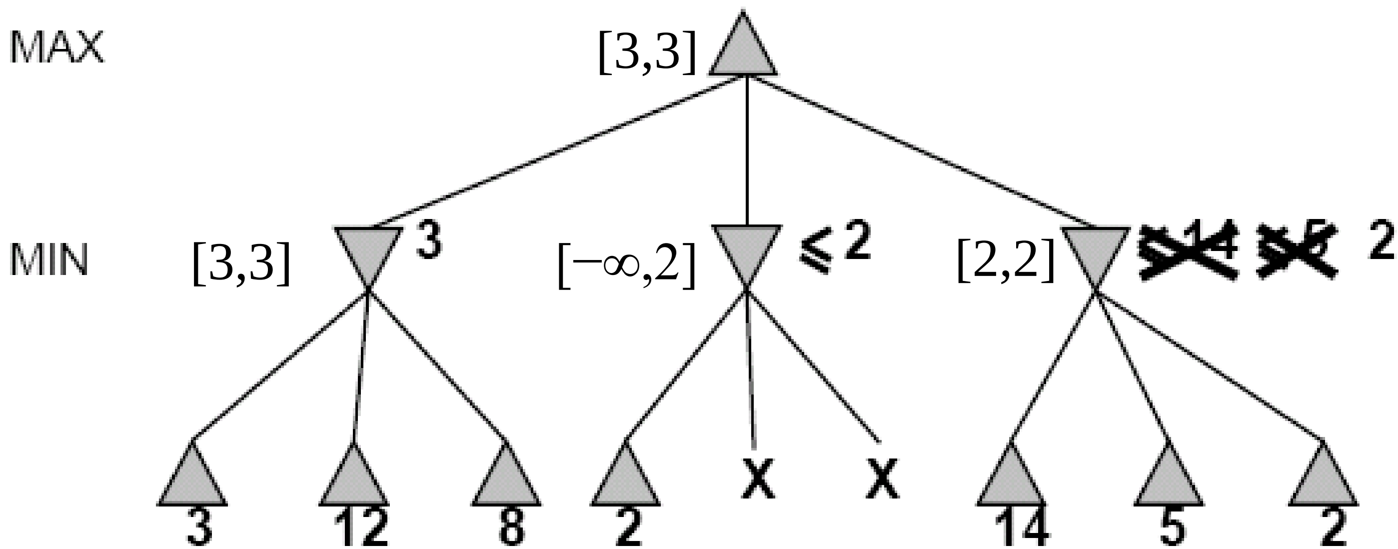
Alpha-Beta Example (continued)



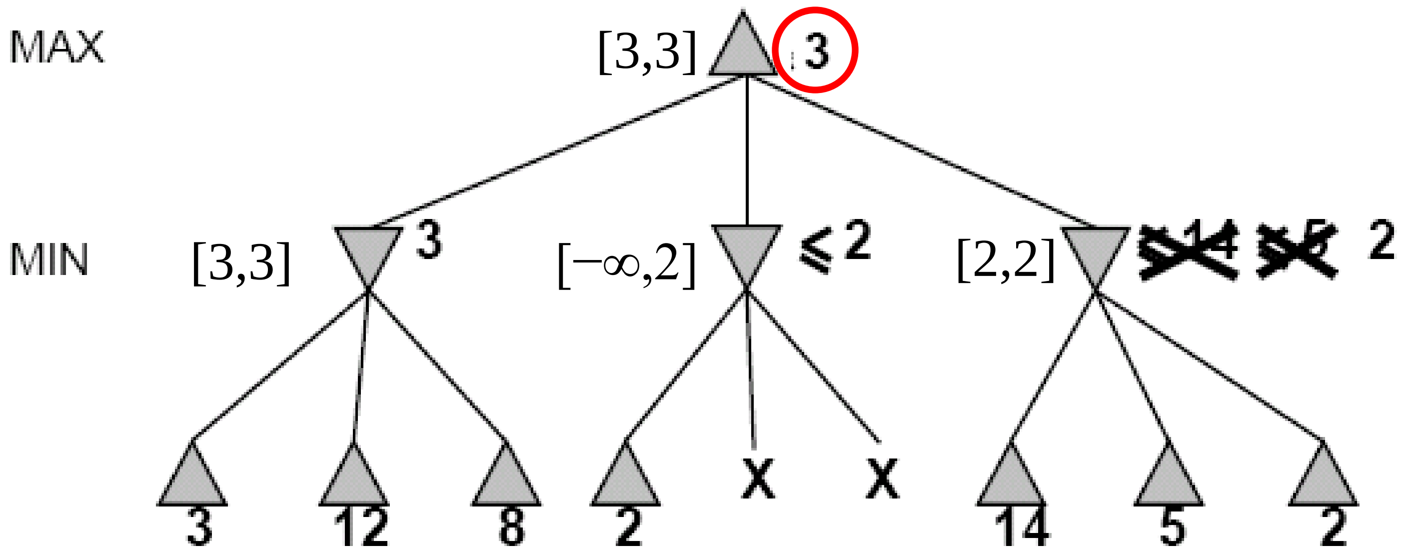
Alpha-Beta Example (continued)



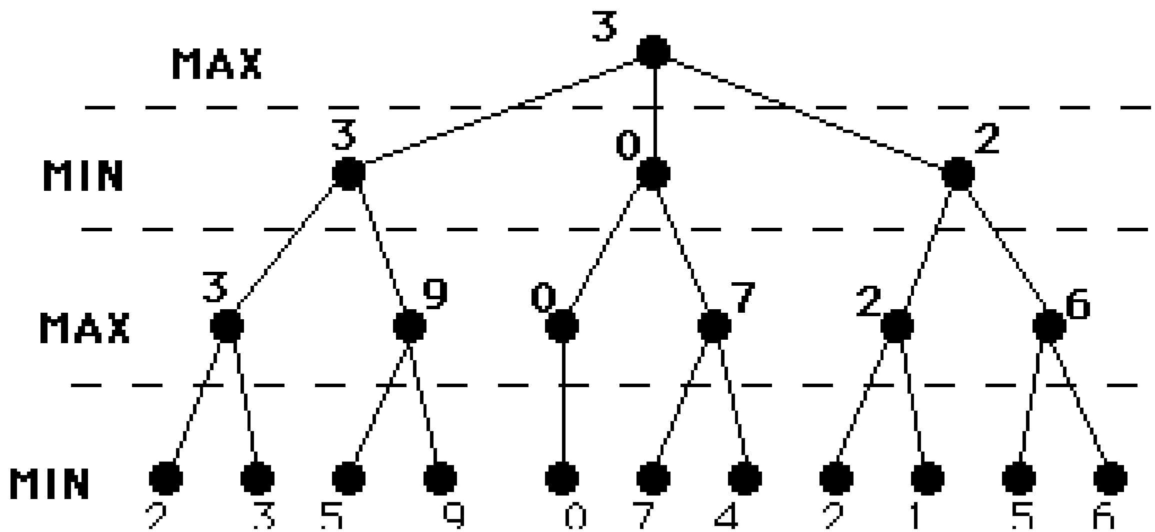
Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



Example Alpha-Beta



Alpha-Beta Analysis

- In perfect case (perfect ordering) the depth is decreased twice in time complexity:

  ➔ O ($b^{d/2}$)

  ➔ which means that the branching factor (b) is decreased to $\sqrt{b}$

Effectiveness of Alpha-Beta Pruning

- Guaranteed to compute same root value as Minimax
- **Worst case**: no pruning, same as Minimax ($O(b^d)$)
- **Best case**: when each player's best move is the first option examined, you examine only $O(b^{d/2})$ nodes, allowing you to search twice as deep!
- For **Deep Blue**, alpha-beta pruning reduced the average branching factor from 35-40 to 6.

Game Trees

- Represent the problem space for a game by a tree
  - **Nodes** represent 'board positions' (state)
  - **edges** represent legal moves.
- **Root node** is the position in which a decision must be made.
- **Evaluation function *f*** assigns real-number scores to `board positions.'
- **Terminal nodes (leaf)** represent ways the game could end, labeled with the desirability of that ending (e.g. win/lose/draw or a numerical score)

MAX & MIN Nodes

- When I move, I attempt to **MAXimize** my performance.
- When my opponent moves, he attempts to **MINimize** my performance.

TO REPRESENT THIS:

- If we move first, label the root MAX; if our opponent does, label it MIN.
- Alternate labels for each successive tree level.
  - if the root (level 0) is our turn (MAX), all even levels will represent turns for us (MAX), and all odd ones turns for our opponent (MIN).

Evaluation functions

- Evaluations how good a ‘board position’ is
  - Based on static features of that board alone
- Zero-sum assumption lets us use one function to describe goodness for both players.
  - f(n)>0 if we are winning in position n
  - f(n)=0 if position n is tied
  - f(n)<0 if our opponent is winning in position n
- Build using expert knowledge (Heuristic),
  - Tic-tac-toe: f(n)=(# of 3 lengths possible for me)
    - (# possible for you)

Chess Evaluation Functions

- **Alan Turing's**
  f(n)=(sum of your piece values)- (sum of opponent's piece values)

- More complex: weighted sum of positional features:

$$\sum w_i \, feature_i(n)$$

- Deep Blue has > 8000 features
  (IBM Computer vs. Gary Kasparov)

Pawn	1.0
Knight	3.0
Bishop	3.25
Rook	5.0
Queen	9.0

Pieces values for a simple Turing-style evaluation function