Introduction to Artificial Intelligence

LECTURE 4:

Game Principles

Overview

- The minimax (or min-max) algorithm
- Resource limitations
- Alpha-beta pruning

What kind of games?

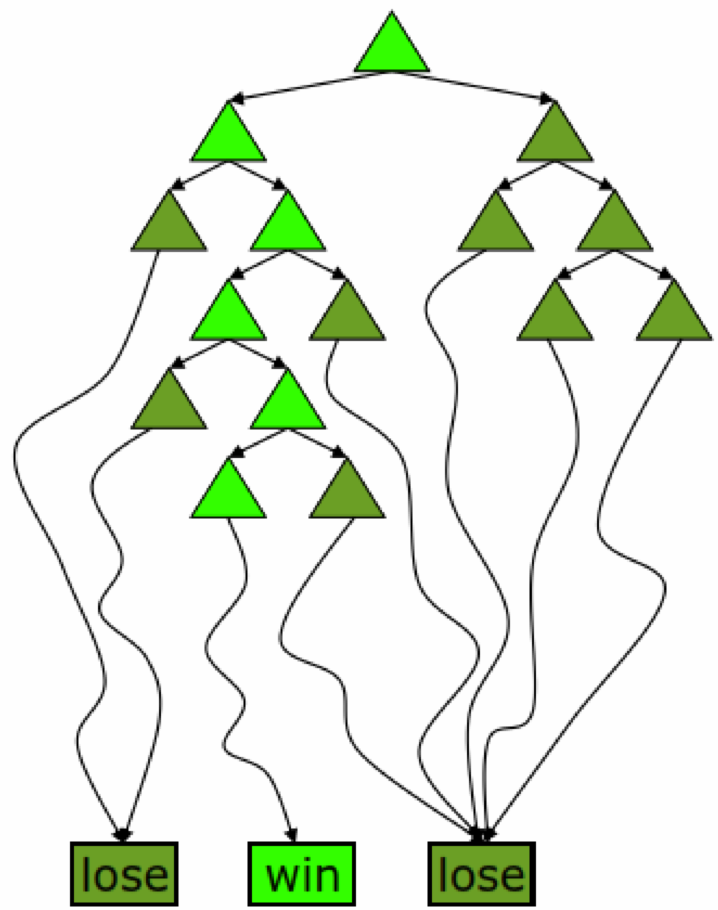
- **Abstraction**: To describe a game we must capture every relevant aspect of the game. Such as:
  - Chess
  - Tic-tac-toe
  - …
- **Accessible environments:** Such games are characterized by perfect information

- **Search:** game-playing then consists of a search through possible game positions

- **Unpredictable opponent:** introduces **uncertainty** thus solution is a strategy specifying a move for every possible opponent reply

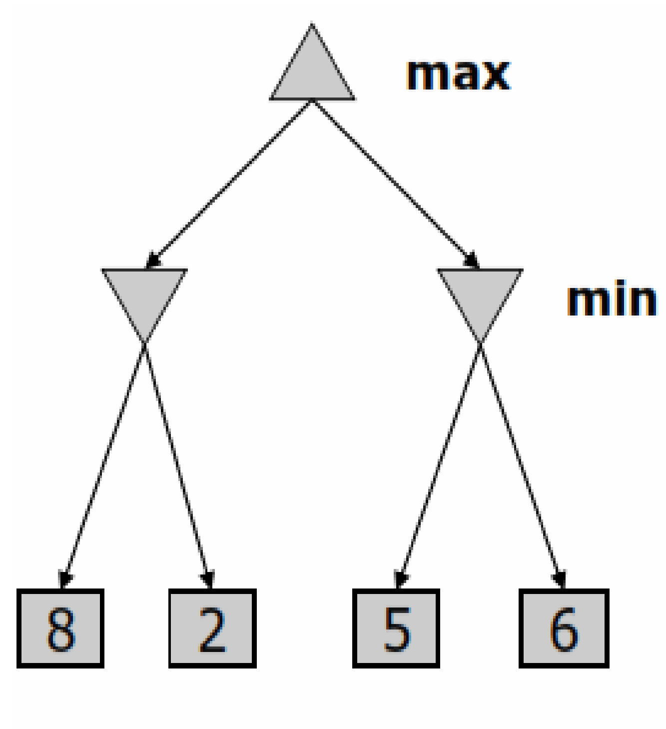
Deterministic Single-Player

- Deterministic, single player, perfect information:
  - Know the rules
  - Know what actions do
  - Know when you win
  - E.g. Rubik's cube
- **… it's just search!**
- Slight reinterpretation:
  - Each node stores a value: the best outcome it can reach
  - This is the maximal outcome of its children (the max value)
  - Note that we don't have path sums as before (utilities at end)
- After search, can pick move that leads to best node

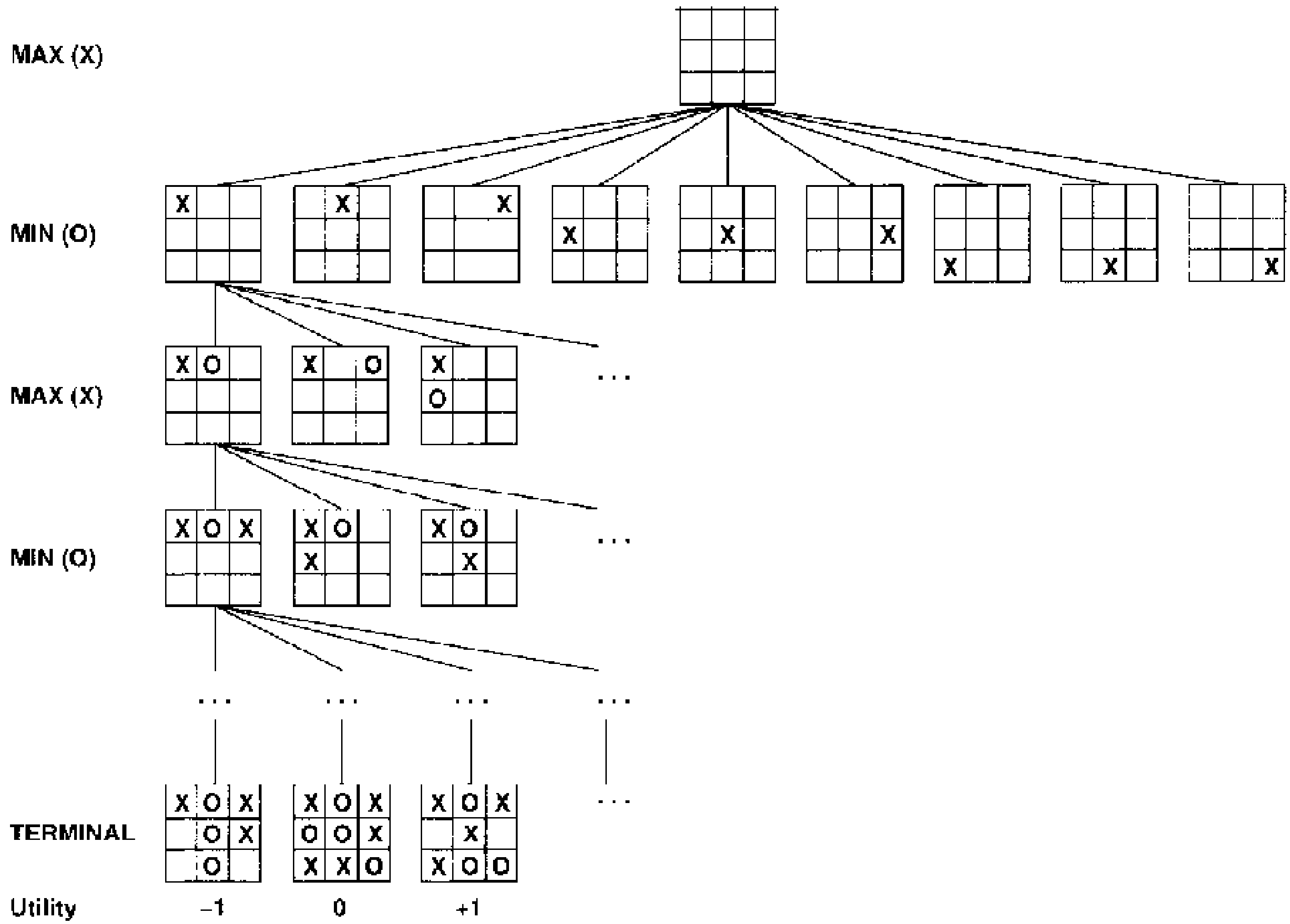


Deterministic Two-Player

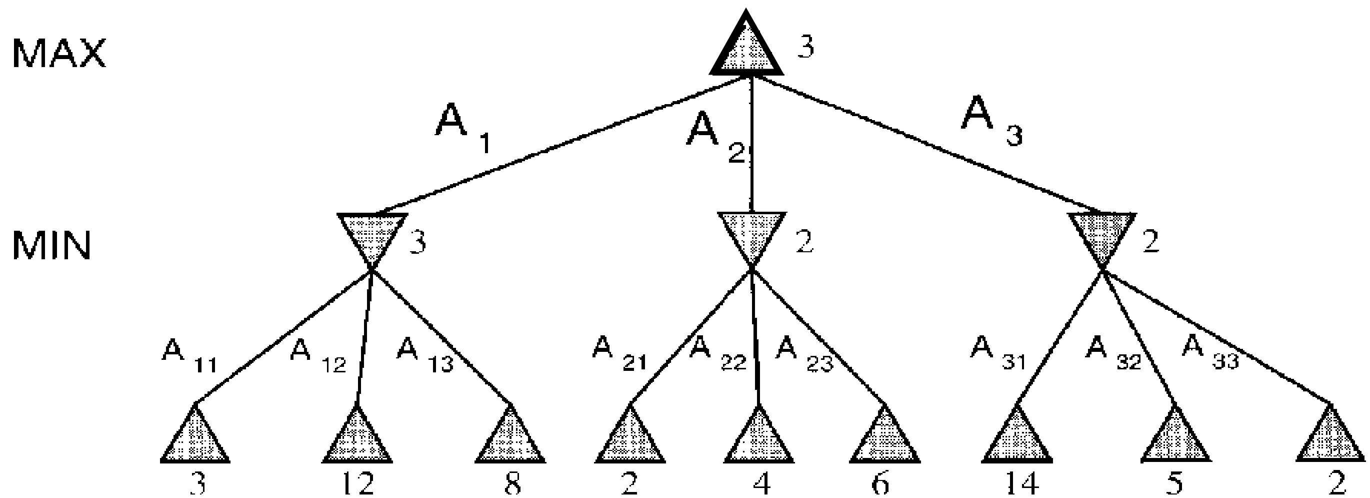
- E.g. tic-tac-toe, chess, checkers
  - Initial state: board position and turn
  - Operators: definition of legal moves
  - Terminal state: conditions for when game is over
  - Utility function: a <u>numeric</u> value that describes the outcome of the game. E.g., -1, 0, 1 for loss, draw, win.
- Zero-sum games
  - One player maximizes result
  - The other minimizes result
- Minimax search
  - A state-space search tree
  - Players alternate
  - Each layer, or ply, consists of a round of moves
  - Choose move to position with highest minimax value = best achievable utility against best play



Example: Tic-Tac-Toe



The Minimax Algorithm



1. Generate the whole game tree, all the way down to the terminal states, i.e. do a complete depth first search.
2. Apply the utility function to each terminal state to get its value.
3. Use the utility of the terminal states to determine the utility of the nodes one level higher up in the search tree.
4. Continue backing up the values from the leaf nodes toward the root one layer at a time.
5. Eventually, the backed-up values reach the top of the tree; at that point, MAX chooses the move that leads to the highest value.
   - **Minimax decision**: maximizes the utility under the assumption that the opponent will play perfectly to minimize it.

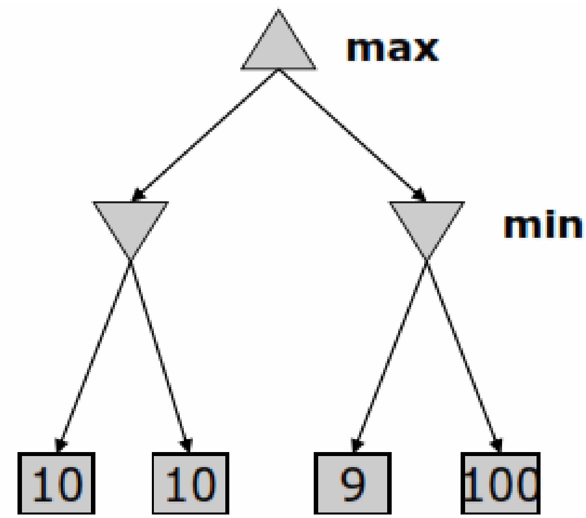
Minimax: Recursive Implementation

```
function MINIMAX-DECISION(game) returns an operator

  for each op in OPERATORS[game] do
      VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
  end
  return the op with the highest VALUE[op]
```

```
function MINIMAX-VALUE(state, game) returns a utility value

  if TERMINAL-TEST[game](state) then
      return UTILITY[game](state)
  else if MAX is to move in state then
      return the highest MINIMAX-VALUE of SUCCESSORS(state)
  else
      return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

The Minimax Algorithm Properties

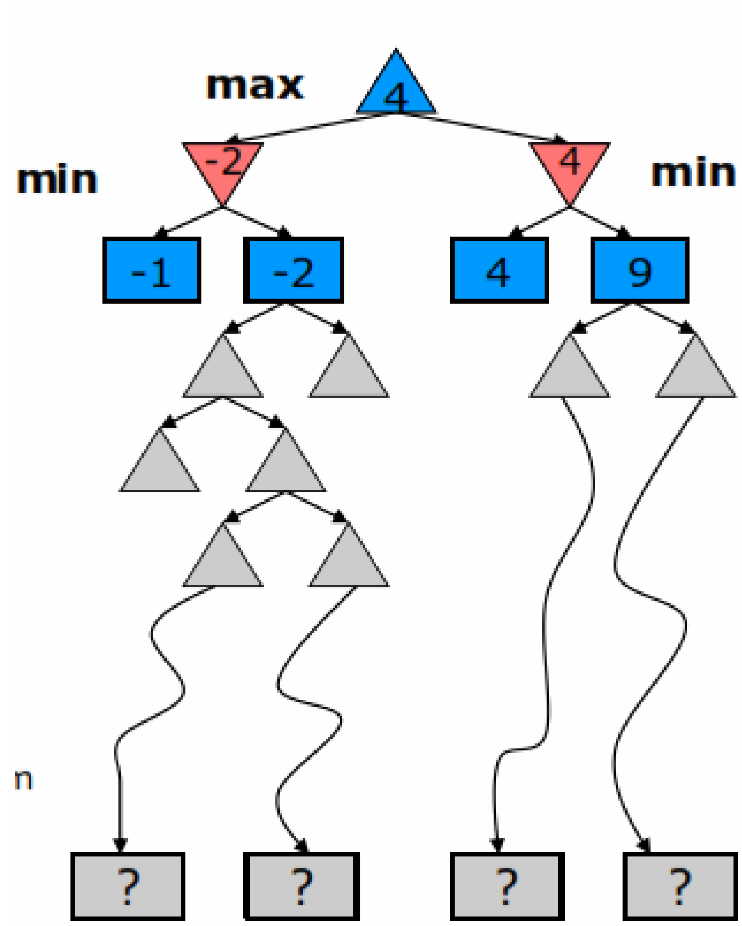
- Performs a complete depth-first exploration of the game tree (for a finite state space)
- Optimal against a perfect player.
- Time complexity?
  - $O(b^m)$
- Space complexity?
  - Depth First Search: does not keep all nodes in memory
  - $O(bm)$



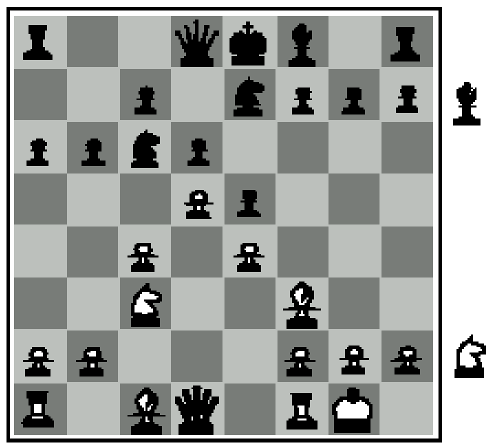
- For chess, b ~ 35, m ~ 100 (50 moves for each player)
  - $\Rightarrow$ *nodes = 100* ^{35}
  - if each node takes about 1 ns to explore then each move will take about ***10*** ^{50} **millennia** to calculate.

Resource Limits

- Complete search is too complex and impractical
- Evaluation function: evaluates value of state using heuristics and cuts off search
- New MINIMAX for depth-limited search:
  - CUTOFF-TEST:
    - cutoff test to replace the termination condition (e.g., deadline, depth-limit, etc.)
    - search a limited depth of tree
  - EVAL: evaluation function to replace utility function for non-terminal positions (e.g., number of chess pieces taken)
- Guarantee of optimal play is gone
- But more plies make a BIG difference

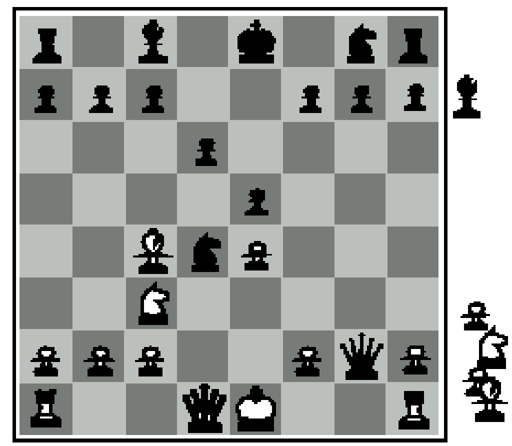


Evaluation Functions



Black to move

White slightly better



White to move

Black winning

- **Weighted linear evaluation function:** to combine *n* heuristics

  $f = w_1 f_1 + w_2 f_2 + \ldots + w_n f_n$

E.g, *w's* could be the values of pieces (1 for prawn, 3 for bishop etc.)
f's could be the number of type of pieces on the board

Minimax with Cutoff: Viable Algorithm?

MINIMAXCUTOFF is identical to MINIMAXVALUE except

1. TERMINAL? is replaced by CUTOFF?
2. UTILITY is replaced by EVAL

Does it work in practice?

Assume we have 100 seconds, evaluate 10^4 nodes/s; can evaluate 10^6 nodes/move

$$b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4$$

4-ply lookahead is a hopeless chess player!

4-ply $\approx$ human novice
8-ply $\approx$ typical PC, human master
12-ply $\approx$ Deep Blue, Kasparov

α-β Pruning: Search Cutoff

- Computing the minimax decision without looking at every node
- Idea: prune portions of the search tree that cannot improve the utility value of the MAX or MIN node, by just considering the values of nodes seen so far.

$$
\begin{aligned}
\text{MINIMAX-VALUE}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3,\min(2,x,y),2) \\
&= \max(3,z,2) \qquad \text{where } z <= 2 \\
&= 3
\end{aligned}
$$

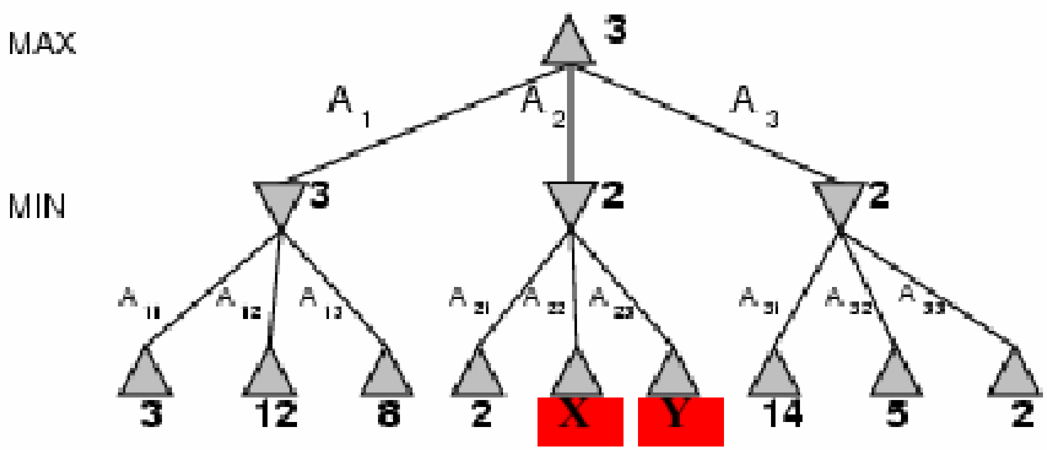
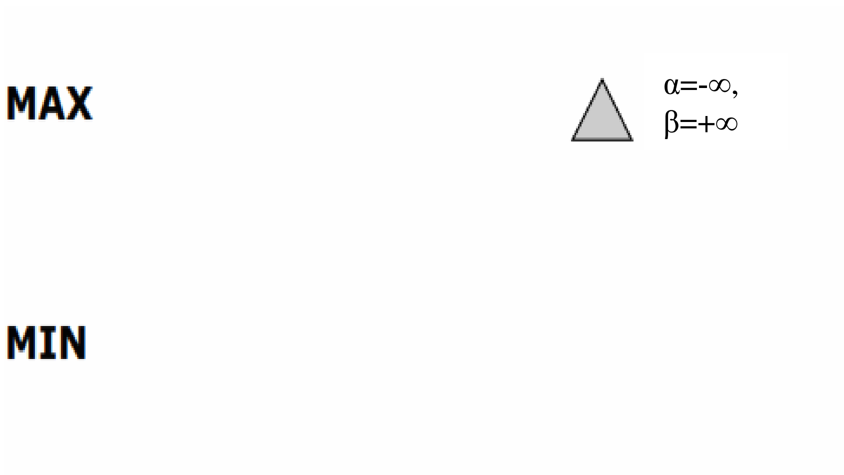


Illustration of α-β Pruning



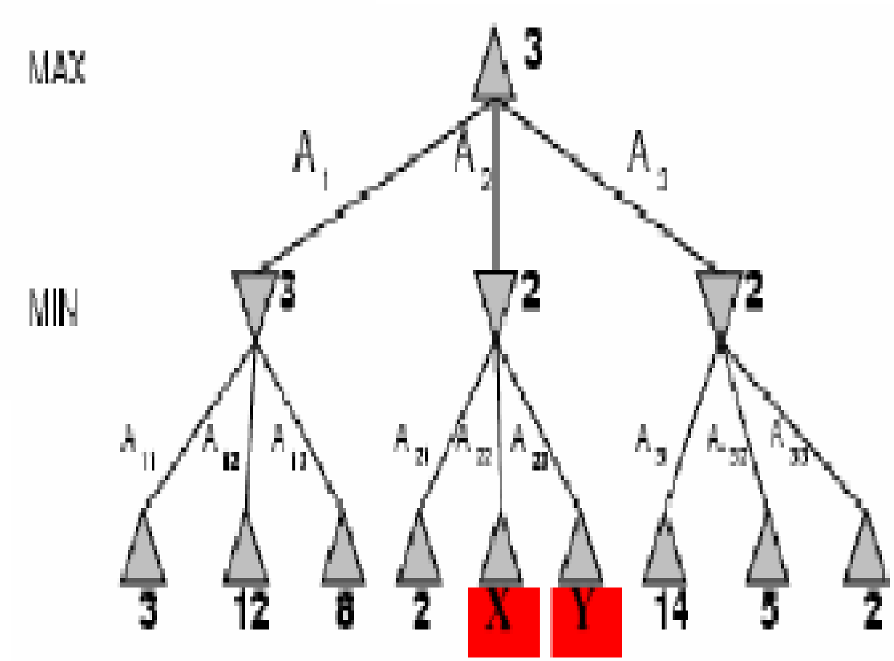
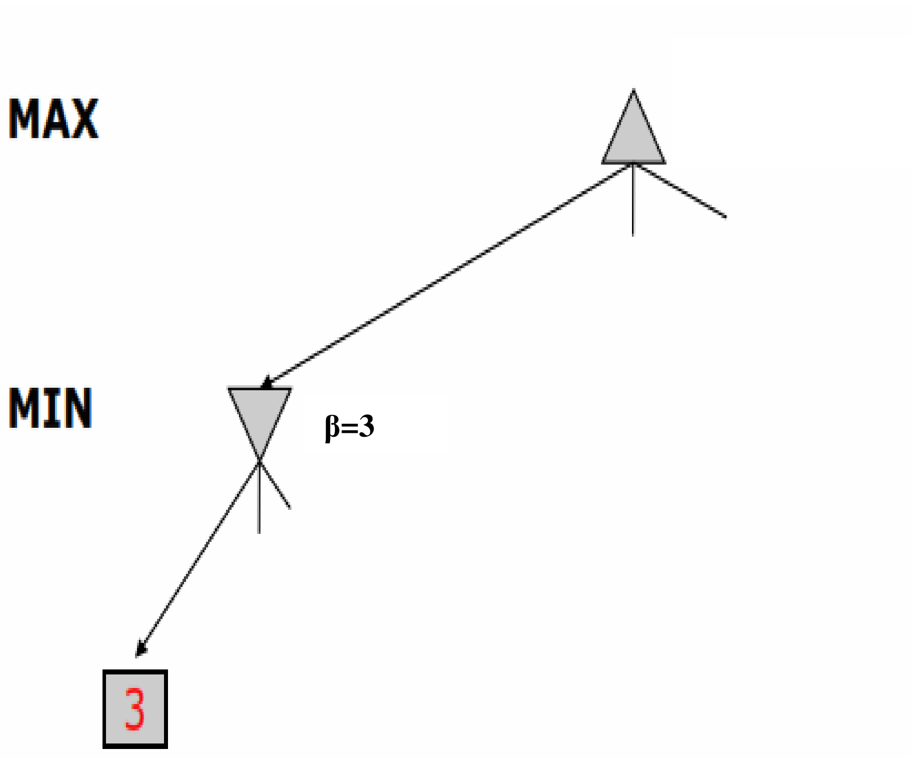


Illustration of α-β Pruning



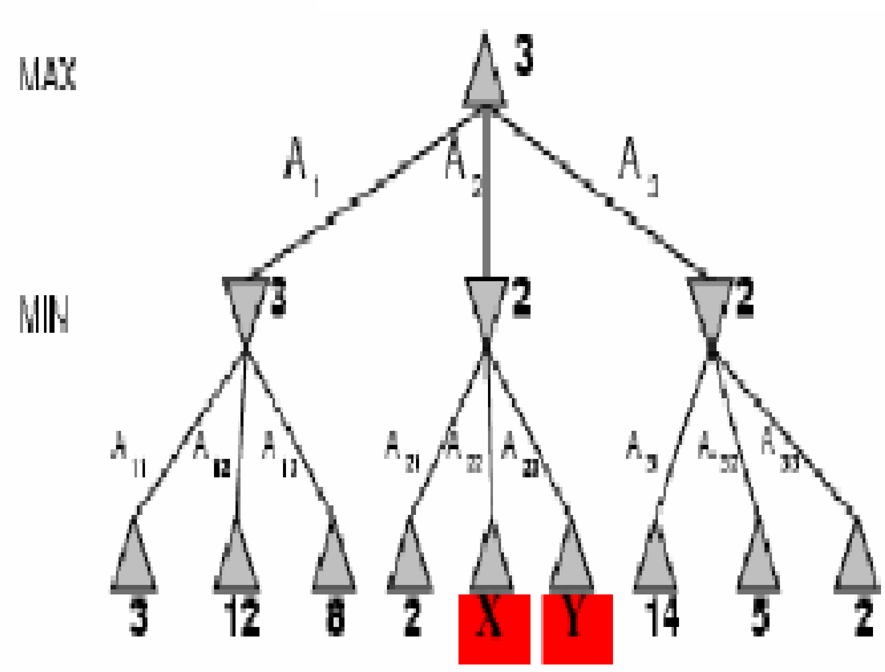
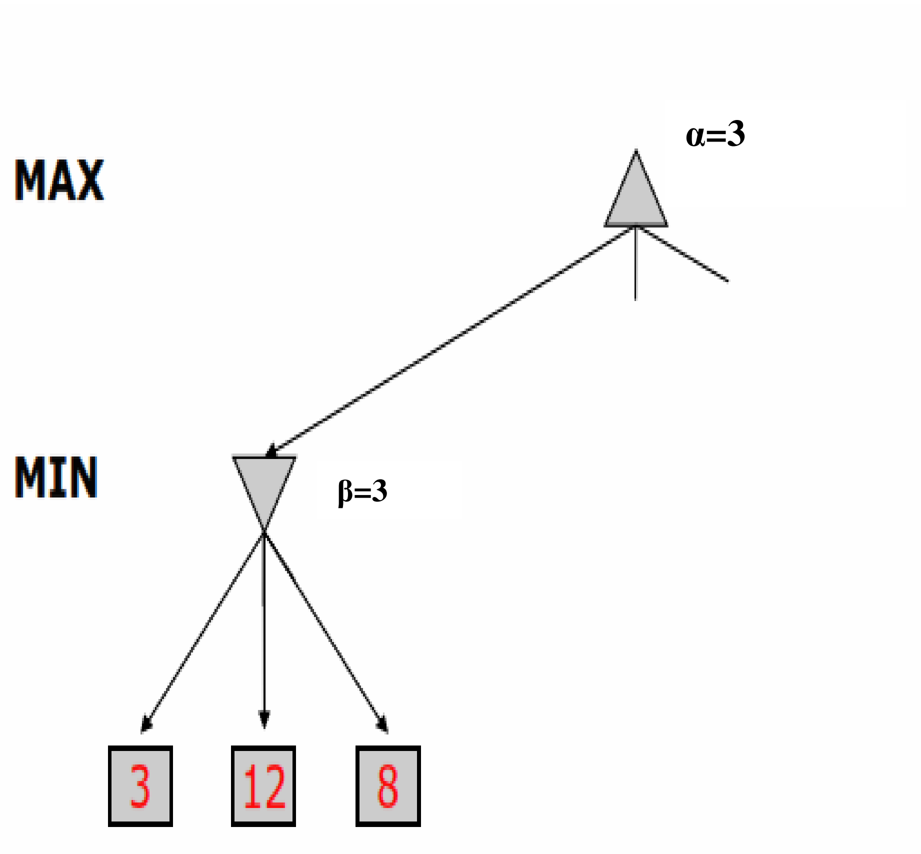


Illustration of α-β Pruning



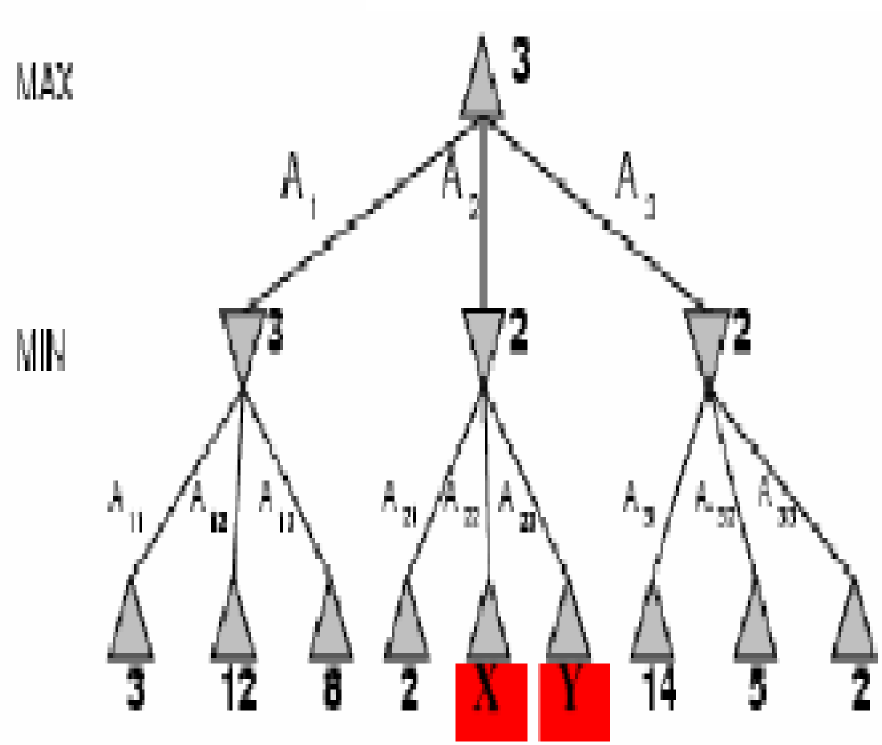
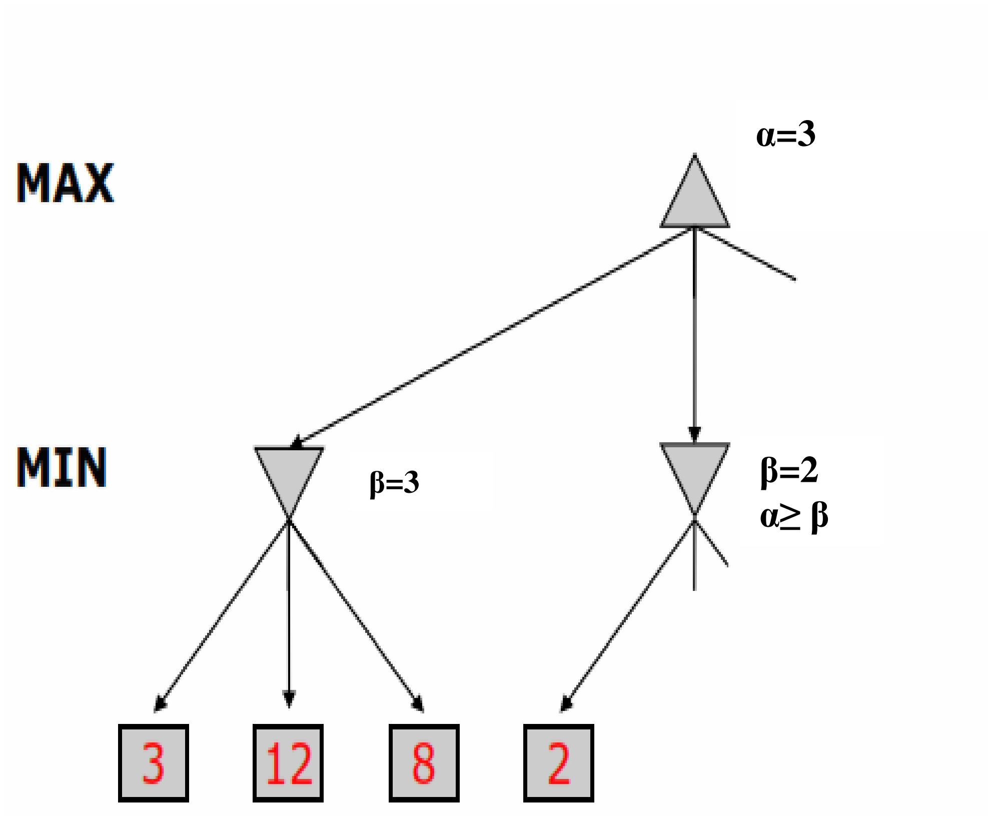


Illustration of α-β Pruning



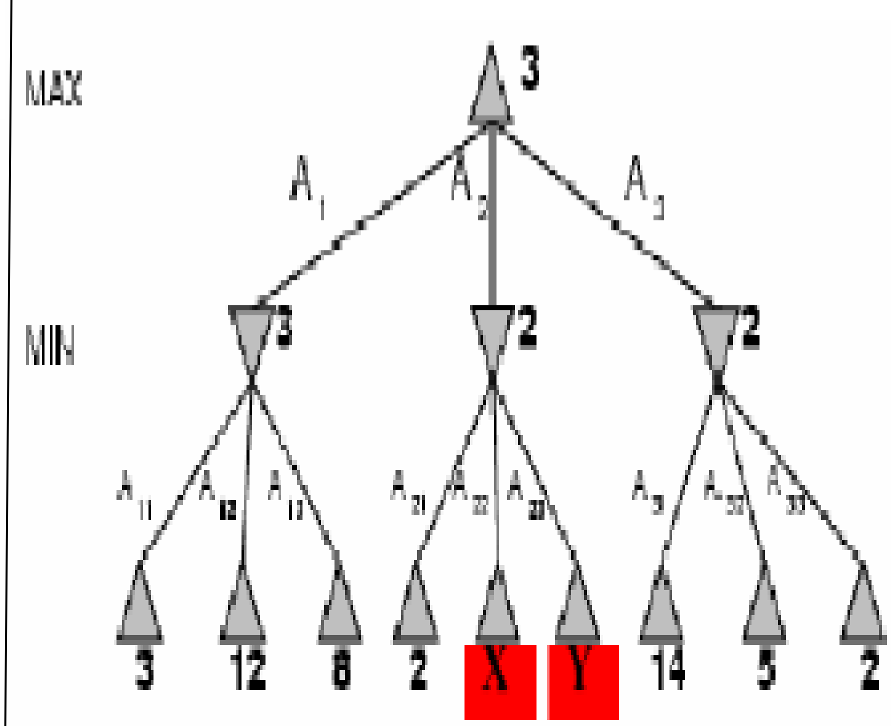
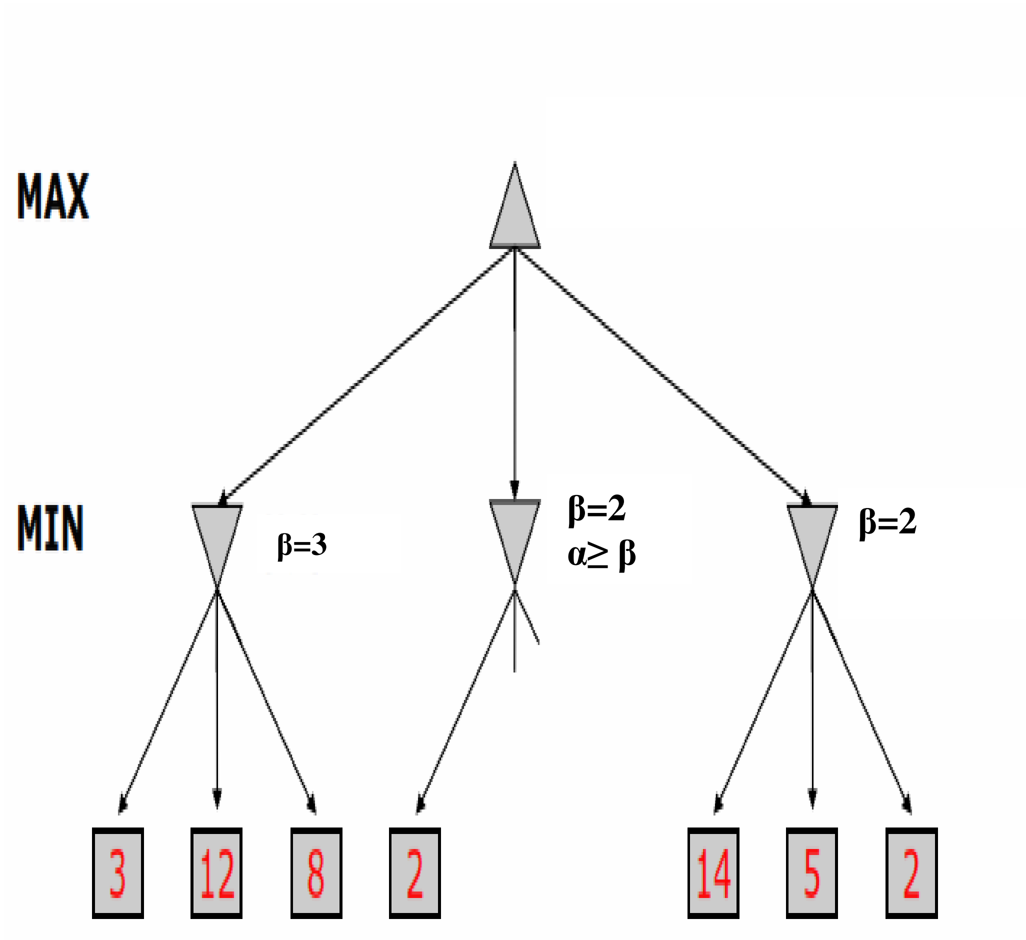
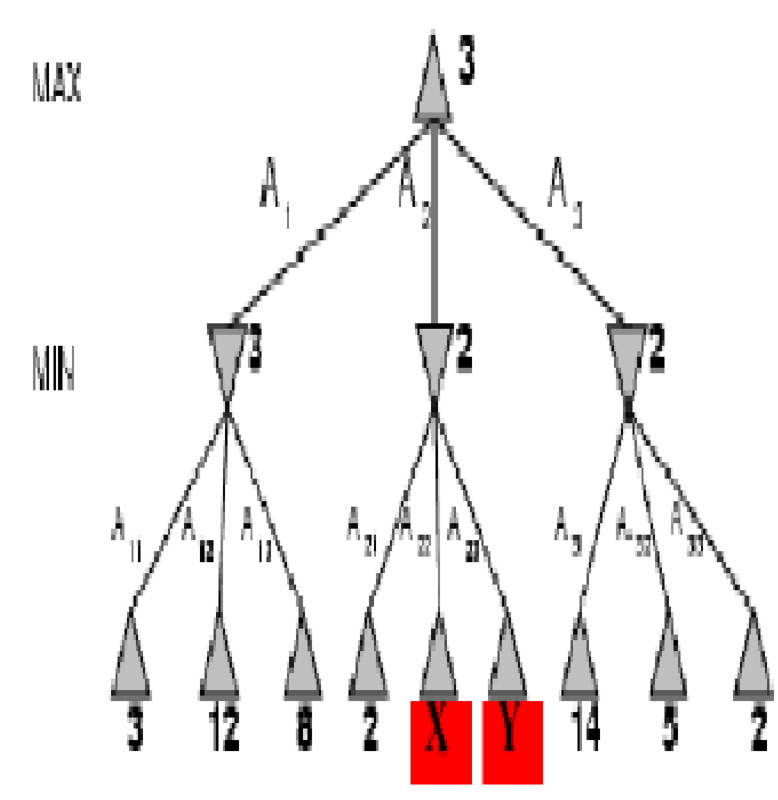


Illustration of α-β Pruning





The α-β Algorithm

Basically MINIMAX + keep track of α, β + prune

```
function MAX-VALUE(state, game, α, β) returns the minimax value of state
   inputs: state, current state in game
           game, game description
           α, the best score for MAX along the path to state
           β, the best score for MIN along the path to state

   if CUTOFF-TEST(state) then return EVAL(state)
   for each s in SUCCESSORS(state) do
       α ← MAX(α, MIN-VALUE(s, game, α, β))
       if α ≥ β then return β
   end
   return α
```

```
function MIN-VALUE(state, game, α, β) returns the minimax value of state

   if CUTOFF-TEST(state) then return EVAL(state)
   for each s in SUCCESSORS(state) do
       β ← MIN(β, MAX-VALUE(s, game, α, β))
       if β ≤ α then return α
   end
   return β
```

Properties of α-β Pruning

- Pruning does not affect final result
- The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined.
  - Good move ordering improves effectiveness of pruning
  - With "perfect ordering," time complexity = $O(b^{m/2})$

    → doubles depth of search