Game-Playing & Adversarial Search

MiniMax, Search Cut-off, Heuristic Evaluation

This lecture topic:
Game-Playing & Adversarial Search
(MiniMax, Search Cut-off, Heuristic Evaluation)
Read Chapter 5.1-5.2, 5.4.1-2, (5.4.3-4, 5.8)

Next lecture topic:
Game-Playing & Adversarial Search
(Alpha-Beta Pruning, [Stochastic Games])
Read Chapter 5.3, (5.5)

(Please read lecture topic material before
and after each lecture on that topic)

You Will Be Expected to Know

- **Basic definitions (section 5.1)**
- **Minimax optimal game search (5.2)**
- **Evaluation functions (5.4.1)**
- **Cutting off search (5.4.2)**
- **Optional: Sections 5.4.3-4; 5.8**

Types of Games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	**battleship Kriegspiel**	bridge, poker, scrabble nuclear war

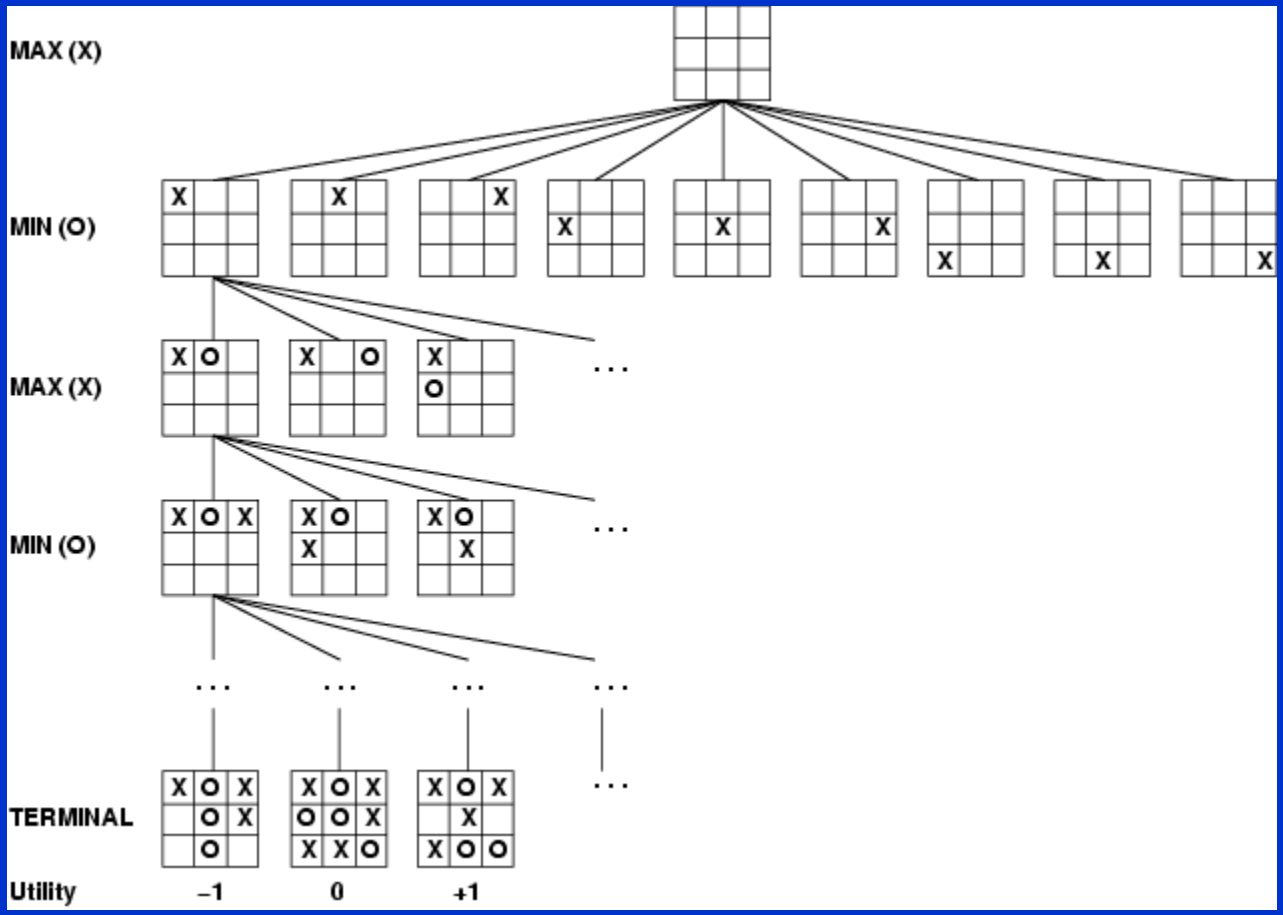
Not Considered:

Physical games like tennis, croquet, ice hockey, etc.
(but see “robot soccer” http://www.robocup.org/)

Typical assumptions

- **Two agents whose actions alternate**
- **Utility values for each agent are the opposite of the other**
  - This creates the adversarial situation
- **Fully observable environments**
- **In game theory terms:**
  - "Deterministic, turn-taking, zero-sum, perfect information"
- **Generalizes: stochastic, multiple players, non zero-sum, etc.**
- **Compare to, e.g., "Prisoner's Dilemma" (p. 666-668, R&N)**
  - "NON-turn-taking, NON-zero-sum, IMperfect information"

Game tree (2-player, deterministic, turns)



How do we search this tree to find the optimal move?

Search versus Games

- **Search – no adversary**
  - Solution is (heuristic) method for finding goal
  - Heuristics and CSP techniques can find *optimal* solution
  - Evaluation function: estimate of cost from start to goal through given node
  - Examples: path planning, scheduling activities

- **Games – adversary**
  - Solution is strategy
    - strategy specifies move for every possible opponent reply.
  - Time limits force an *approximate* solution
  - Evaluation function: evaluate “goodness” of game position
  - Examples: chess, checkers, Othello, backgammon

Games as Search

- **Two players: MAX and MIN**

- **MAX moves first and they take turns until the game is over**
  - Winner gets reward, loser gets penalty.
  - "Zero sum" means the sum of the reward and the penalty is a constant.

- **Formal definition as a search problem:**
  - **Initial state:** Set-up specified by the rules, e.g., initial board set-up of chess.
  - **Player(s):** Defines which player has the move in a state.
  - **Actions(s):** Returns the set of legal moves in a state.
  - **Result(s,a):** Transition model defines the result of a move.
  - **Terminal-Test(s):** Is the game finished? True if finished, false otherwise.
  - **Utility function(s,p):** Gives numerical value of terminal state s for player p.
    - E.g., win (+1), lose (-1), and draw (0) in tic-tac-toe.
    - E.g., win (+1), lose (0), and draw (1/2) in chess.

- **MAX uses search tree to determine "best" next move.**

An optimal procedure: The Min-Max method

Will find the optimal strategy and best next move for Max:

- **1. Generate the whole game tree, down to the leaves.**
- **2. Apply utility (payoff) function to each leaf.**
- **3. Back-up values from leaves through branch nodes:**
  - a Max node computes the Max of its child values
  - a Min node computes the Min of its child values
- **4. At root: Choose move leading to the child of highest value.**

Two-ply Game Tree

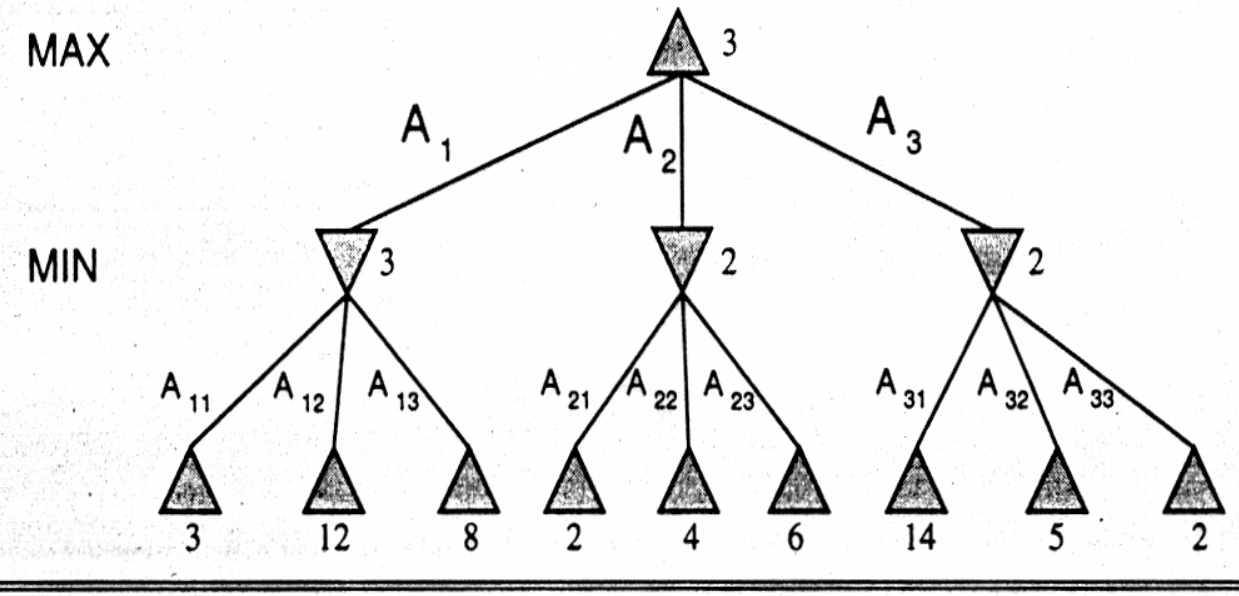
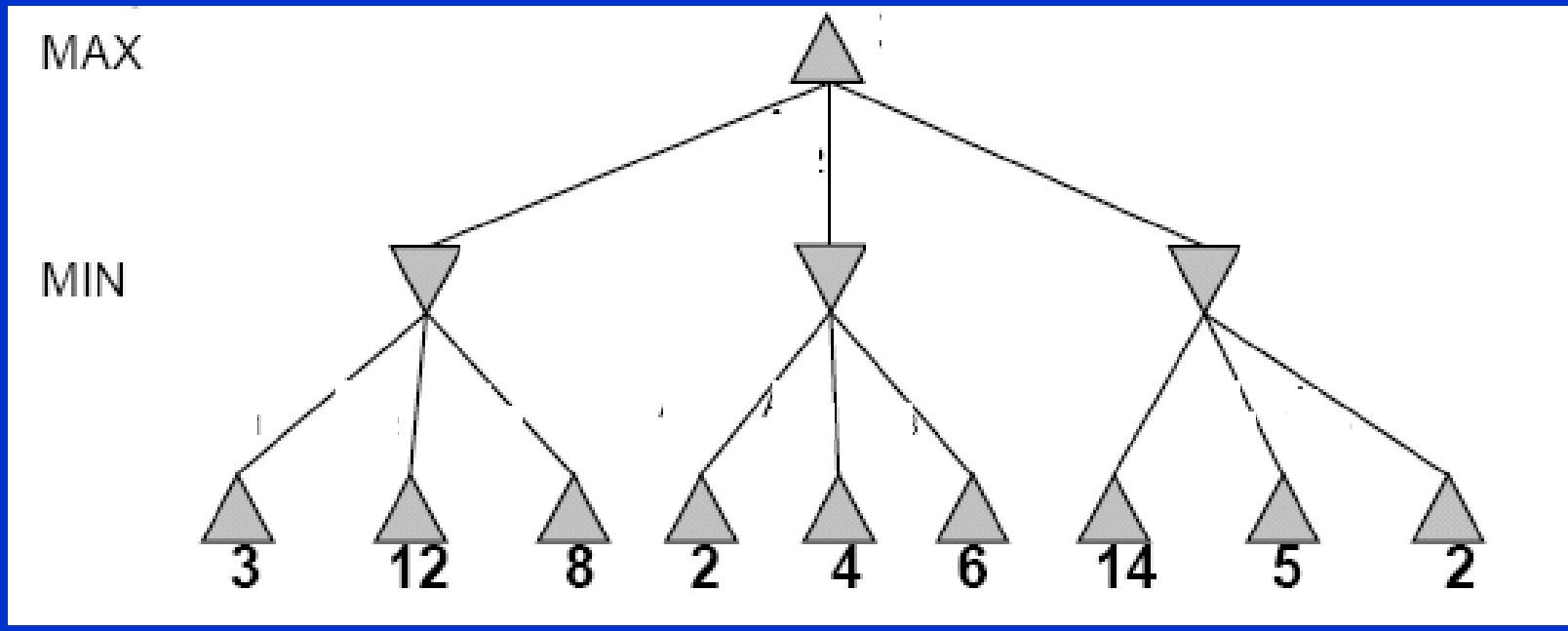
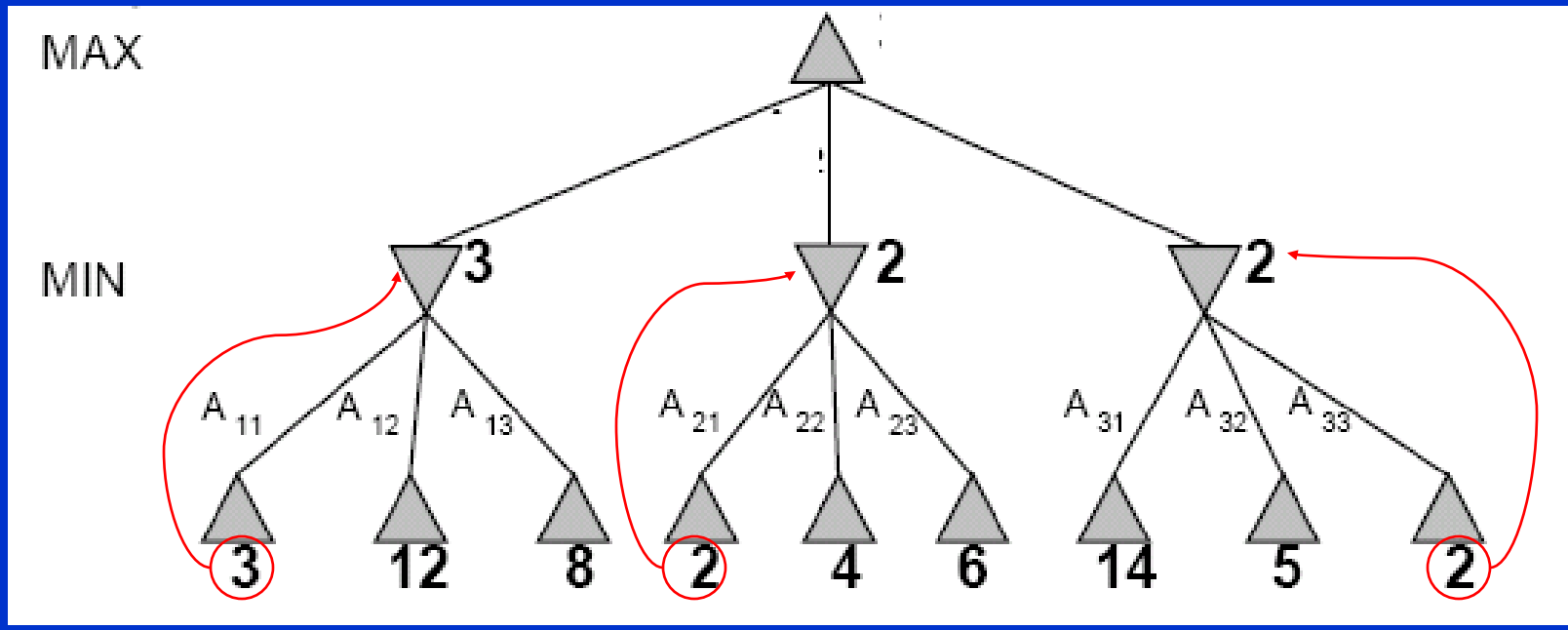


Figure 5.2 A two-ply game tree as generated by the minimax algorithm. The △ nodes are moves by MAX and the ▽ nodes are moves by MIN. The terminal nodes show the utility value for MAX computed by the utility function (i.e., by the rules of the game), whereas the utilities of the other nodes are computed by the minimax algorithm from the utilities of their successors. MAX's best move is A_1, and MIN's best reply is A_{11}.

Two-Ply Game Tree

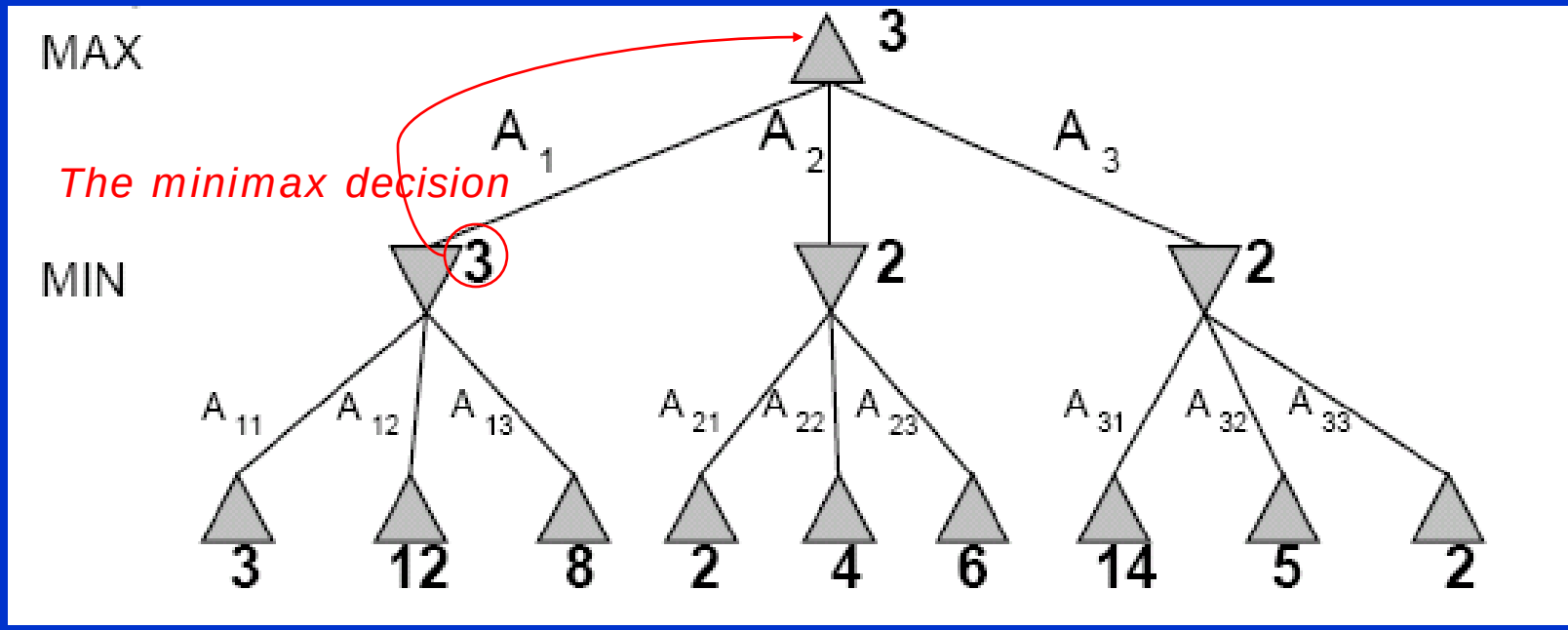


Two-Ply Game Tree



Two-Ply Game Tree

Minimax maximizes the utility of the worst-case outcome for Max



Pseudocode for Minimax Algorithm

function MINIMAX-DECISION(*state*) **returns** *an action*
  inputs: *state*, current state in game
  return arg $\max_{a \in \text{ACTIONS}(state)}$ MIN-VALUE(Result(*state*,*a*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for *a* in ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v,MIN-VALUE(Result(*state*,*a*)))
  return *v*

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for *a* in ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v,MAX-VALUE(Result(*state*,*a*)))
  return *v*

Properties of minimax

- **Complete?**
  - Yes (if tree is finite).
- **Optimal?**
  - Yes (against an optimal opponent).
  - Can it be beaten by an opponent playing sub-optimally?
    - No. (Why not?)
- **Time complexity?**
  - $O(b^m)$
- **Space complexity?**
  - O(bm) (depth-first search, generate all actions at once)
  - O(m) (backtracking search, generate actions one at a time)

Game Tree Size

- **Tic-Tac-Toe**
  - b ≈ 5 legal actions per state on average, total of 9 plies in game.
    - "ply" = one action by one player, "move" = two plies.
  - 5^9 = 1,953,125
  - 9! = 362,880 (Computer goes first)
  - 8! = 40,320 (Computer goes second)
  - → **exact solution <u>quite reasonable</u>**

- **Chess**
  - b ≈ 35 (approximate average branching factor)
  - d ≈ 100 (depth of game tree for "typical" game)
  - $b^d \approx 35^{100} \approx 10^{154}$ nodes!!
  - → **exact solution <u>completely infeasible</u>**

- **<u>It is usually impossible to develop the whole search tree.</u>**

Cutting off search

MinimaxCutoff is identical to MinimaxValue except

1. Terminal? is replaced by Cutoff?
2. Utility is replaced by Eval

Does it work in practice?

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

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

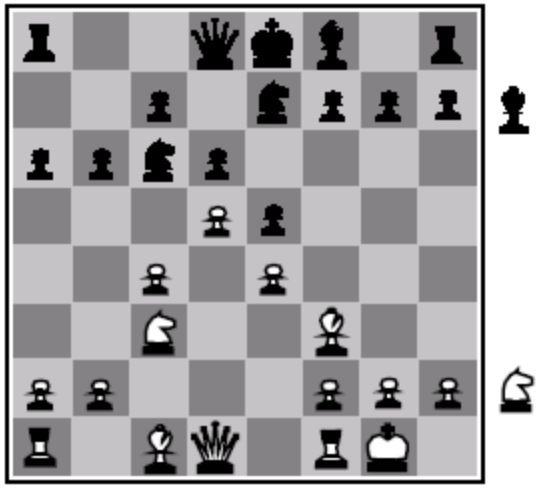
(Static) Heuristic Evaluation Functions

- **An Evaluation Function:**
  - Estimates how good the current board configuration is for a player.
  - Typically, evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent's score from the player's.
  - Often called "static" because it is called on a static board position.
  - Othello: Number of white pieces - Number of black pieces
  - Chess: Value of all white pieces - Value of all black pieces

- **Typical values :**
  - -infinity (loss) to +infinity (win) or [-1, +1] or [0, +1].

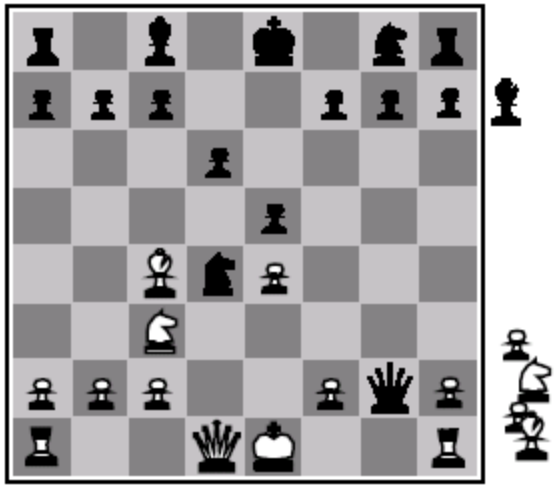
- **Board evaluation X for player is -X for opponent**
  - "Zero-sum game" (i.e., scores sum to a constant)

Evaluation functions



Black to move

White slightly better



White to move

Black winning

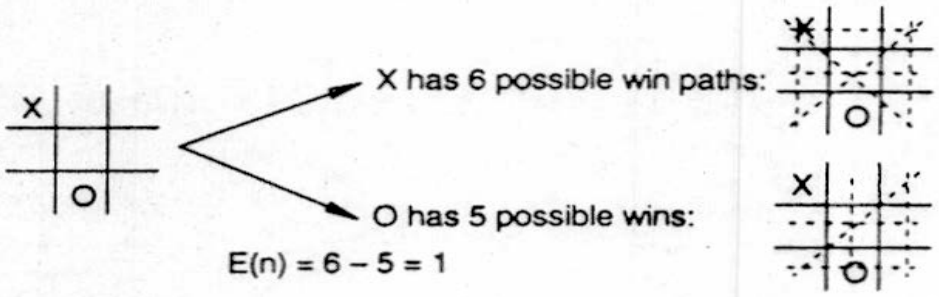
For chess, typically *linear* weighted sum of features

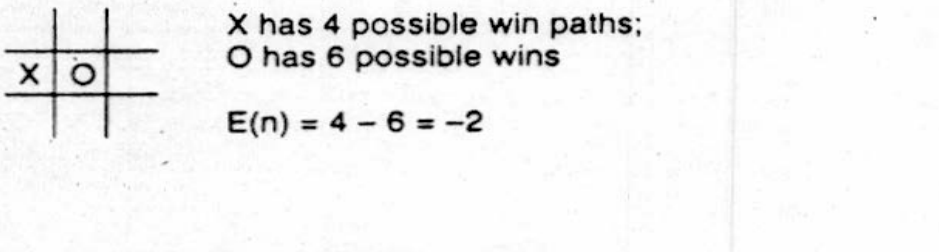
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

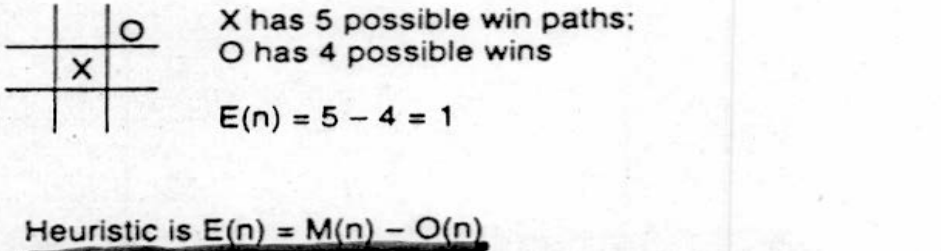
e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (number of black queens), etc.

Applying MiniMax to tic-tac-toe

- **The static heuristic evaluation function:**







Heuristic is E(n) = M(n) – O(n)

where M(n) is the total of My possible winning lines

O(n) is total of Opponent's possible winning lines

E(n) is the total Evaluation for state n

Figure 4.16 Heuristic measuring conflict applied to states of tic-tac-toe.

Minimax Values (two ply)

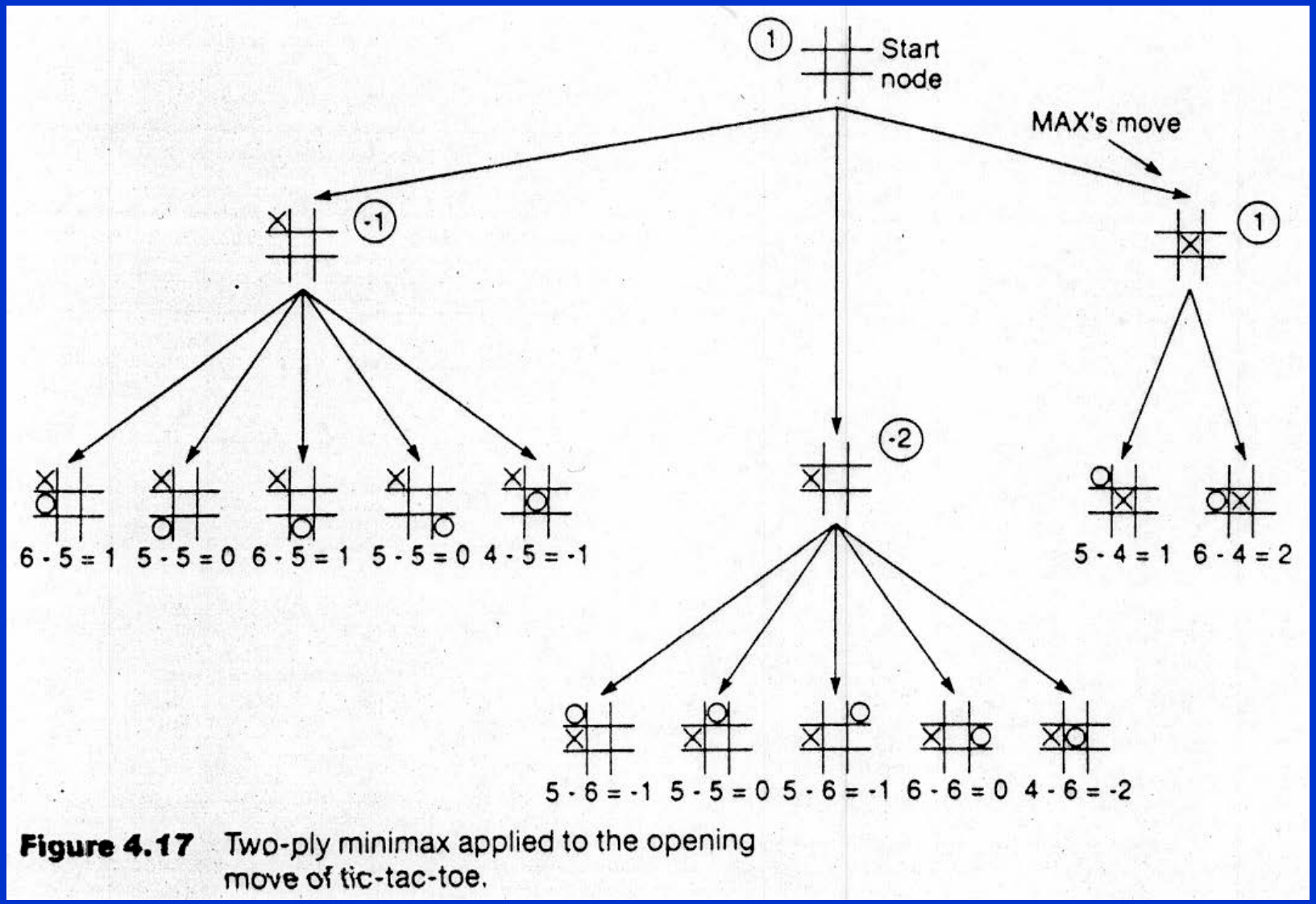


Figure 4.17 Two-ply minimax applied to the opening move of tic-tac-toe.

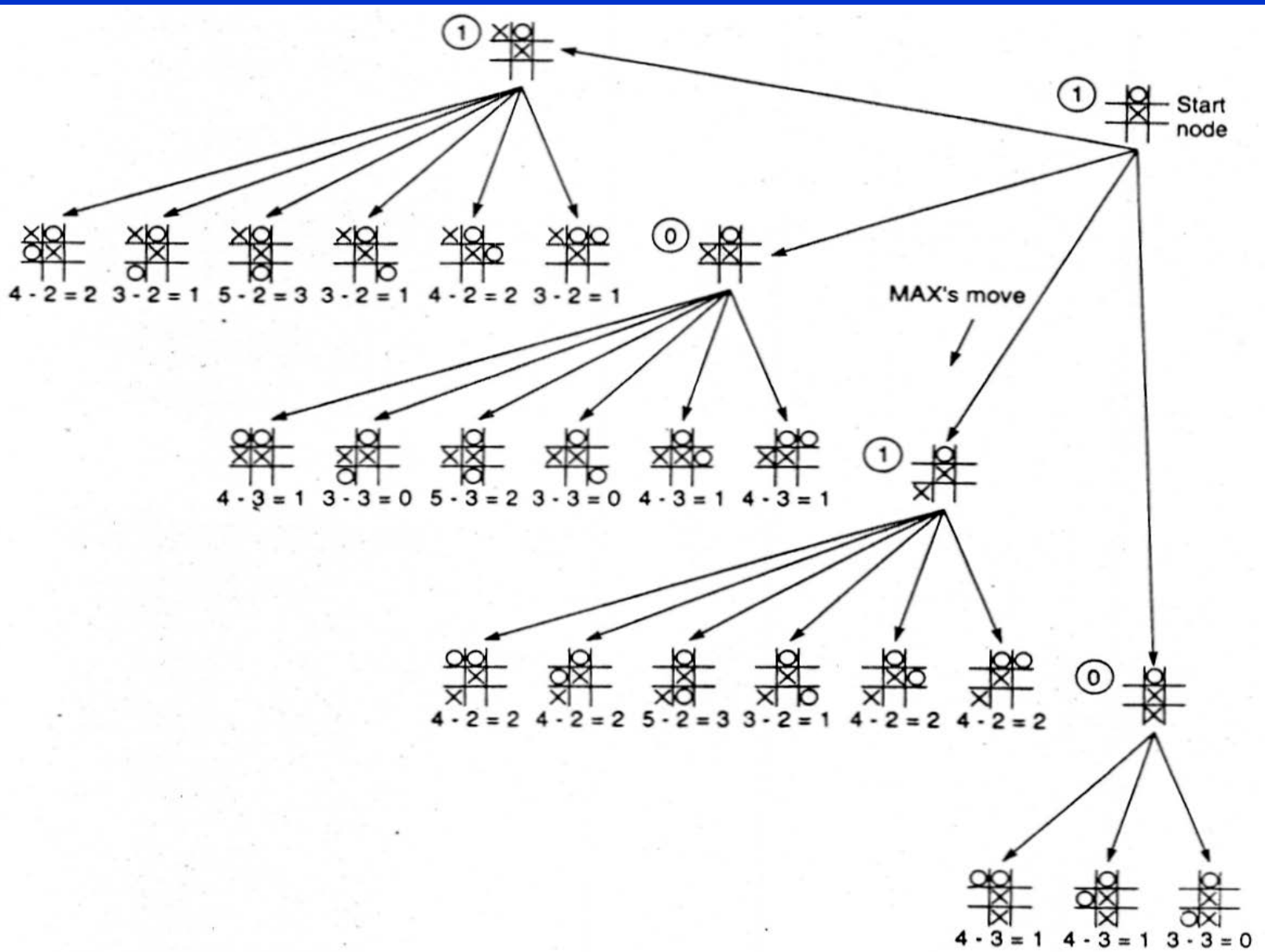


Figure 4.18 Two-ply minimax applied to X's second move of tic-tac-toe.

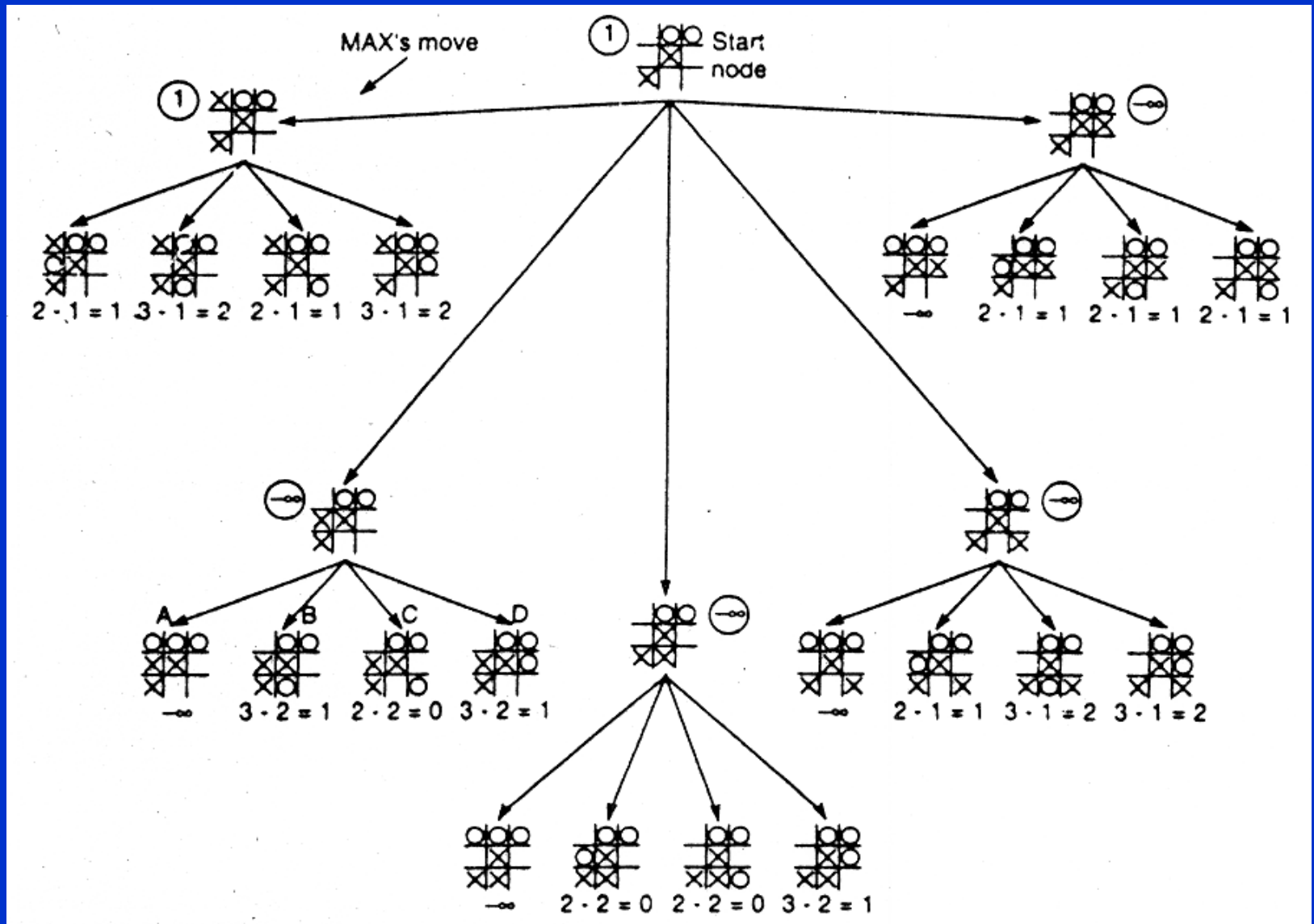
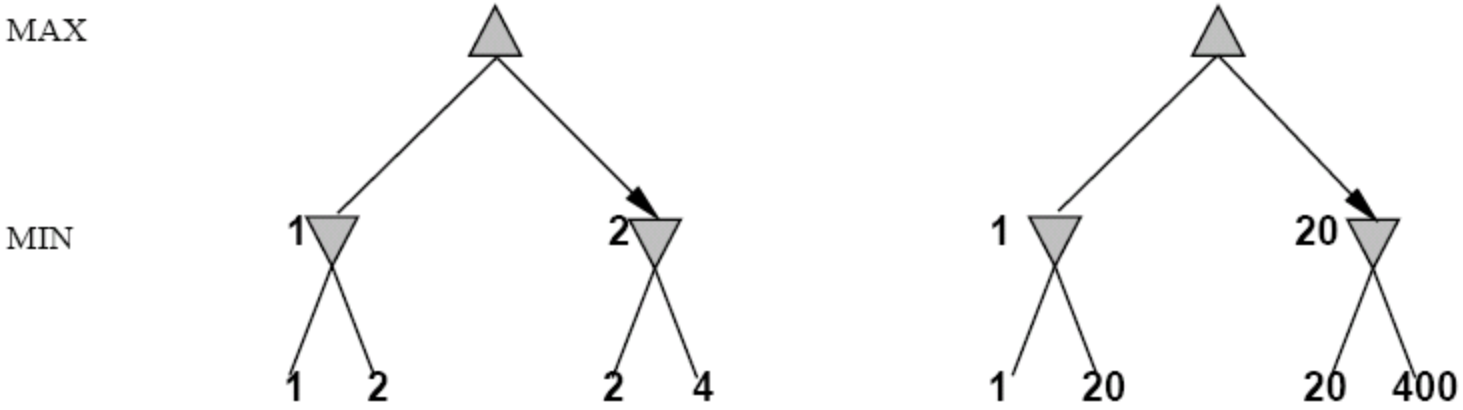


Figure 4.19 Two-ply minimax applied to **X**'s move near end game.

Digression: Exact values don't matter



Behaviour is preserved under any *monotonic* transformation of EVAL

Only the order matters:
payoff in deterministic games acts as an *ordinal utility* function

Iterative (Progressive) Deepening

- **In real games, there is usually a time limit T to make a move**

- **How do we take this into account?**
- **Using MiniMax we cannot use "partial" results with any confidence unless the full tree has been searched**
  - So, we could be conservative and set a conservative depth-limit which guarantees that we will find a move in time < T
    - Disadvantage: we may finish early, could do more search

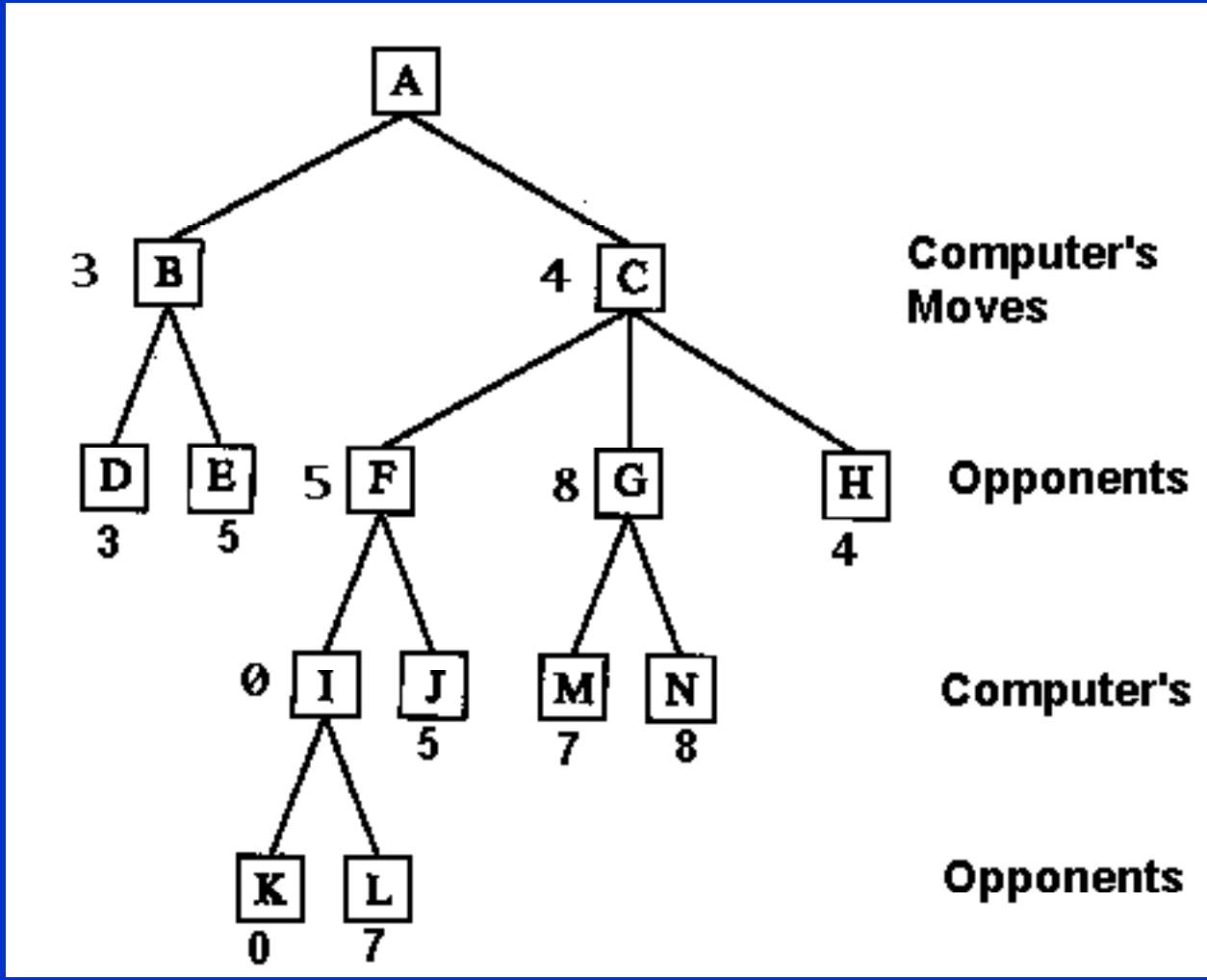
- **In practice, Iterative Deepening Search (IDS) is used**
  - IDS runs depth-first search with an increasing depth-limit
  - When the clock runs out we use the solution found at the previous depth limit
  - With alpha-beta pruning (next lecture), we can sort the nodes based on values found in previous depth limit to maximize pruning during the next depth limit => search deeper

Heuristics and Game Tree Search:
Limited horizon

- **The Horizon Effect**
  - Sometimes there's a major "effect" (e.g., a piece is captured) which is "just below" the depth to which the tree has been expanded
  - The computer cannot see that this major event could happen because it has a "limited horizon" --- i.e., when search stops
  - There are heuristics to try to follow certain branches more deeply to detect such important events, and so avoid stupid play
  - This helps to avoid catastrophic losses due to "short-sightedness"

- **Heuristics for Tree Exploration**
  - Often better to explore some branches more deeply in allotted time
  - Various heuristics exist to identify "promising" branches
  - Stop at "quiescent" positions --- all battles are over, things are quiet
  - Continue when things are in violent flux --- the middle of a battle

Selectively Deeper Game Trees



Eliminate Redundant Nodes

- **On average, each board position appears in the search tree approximately ~10^{150} / ~10^{40} ≈ 10^{100} times.**

  => Vastly redundant search effort.

- **Can't remember all nodes (too many).**

  => Can't eliminate all redundant nodes.

- **However, some short move sequences provably lead to a redundant position.**

  – These can be deleted dynamically with no memory cost

- **Example:**

  1. P-QR4 P-QR4; 2. P-KR4 P-KR4

  leads to the same position as

  1. P-QR4 P-KR4; 2. P-KR4 P-QR4

Summary

- Game playing is best modeled as a search problem
- Game trees represent alternate computer/opponent moves
- Minimax is a procedure which chooses moves by assuming that the opponent will always choose the move which is best for them
  - Basically, it avoids all worst-case outcomes for Max, to find the best.
  - If the opponent makes an error, Minimax will take optimal advantage of that error and make the best possible play that exploits the error.
- Cutting off search
  - In general, it is infeasible to search the entire game tree.
  - In practice, Cutoff-Test decides when to stop searching further.
  - Prefer to stop at quiescent positions.
  - Prefer to keep searching in positions that are still violently in flux.
- Static heuristic evaluation function
  - Estimate quality of a given board configuration for the Max player.
  - Called when search is cut off, to determine value of position found.