

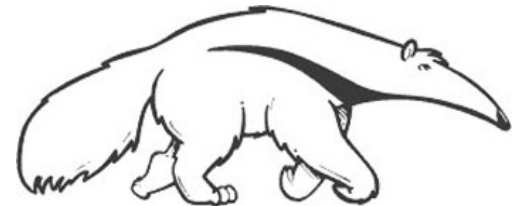
# Games and Adversarial Search A: Mini-max, Cutting Off Search

CS271P, Fall Quarter, 2019

Introduction to Artificial Intelligence

Prof. Richard Lathrop

Read Beforehand: R&N 5.1, 5.2, 5.4



# Outline

- **Computer programs that play 2-player games**
  - game-playing as search with the complication of an opponent
- **General principles of game-playing and search**
  - game tree
  - minimax principle; impractical, but theoretical basis for analysis
  - evaluation functions; cutting off search; static heuristic functions
  - alpha-beta-pruning
  - heuristic techniques
  - games with chance
  - Monte-Carlo tree search
- **Status of Game-Playing Systems**
  - in chess, checkers, backgammon, Othello, Go, etc., computers routinely defeat leading world players.

# Types of games

Deterministic:

Chance:

Perfect  
Information:

chess, checkers, go,  
othello

backgammon,  
monopoly

Imperfect  
Information:

battleship, Kriegspiel

Bridge, poker,  
scrabble, ...

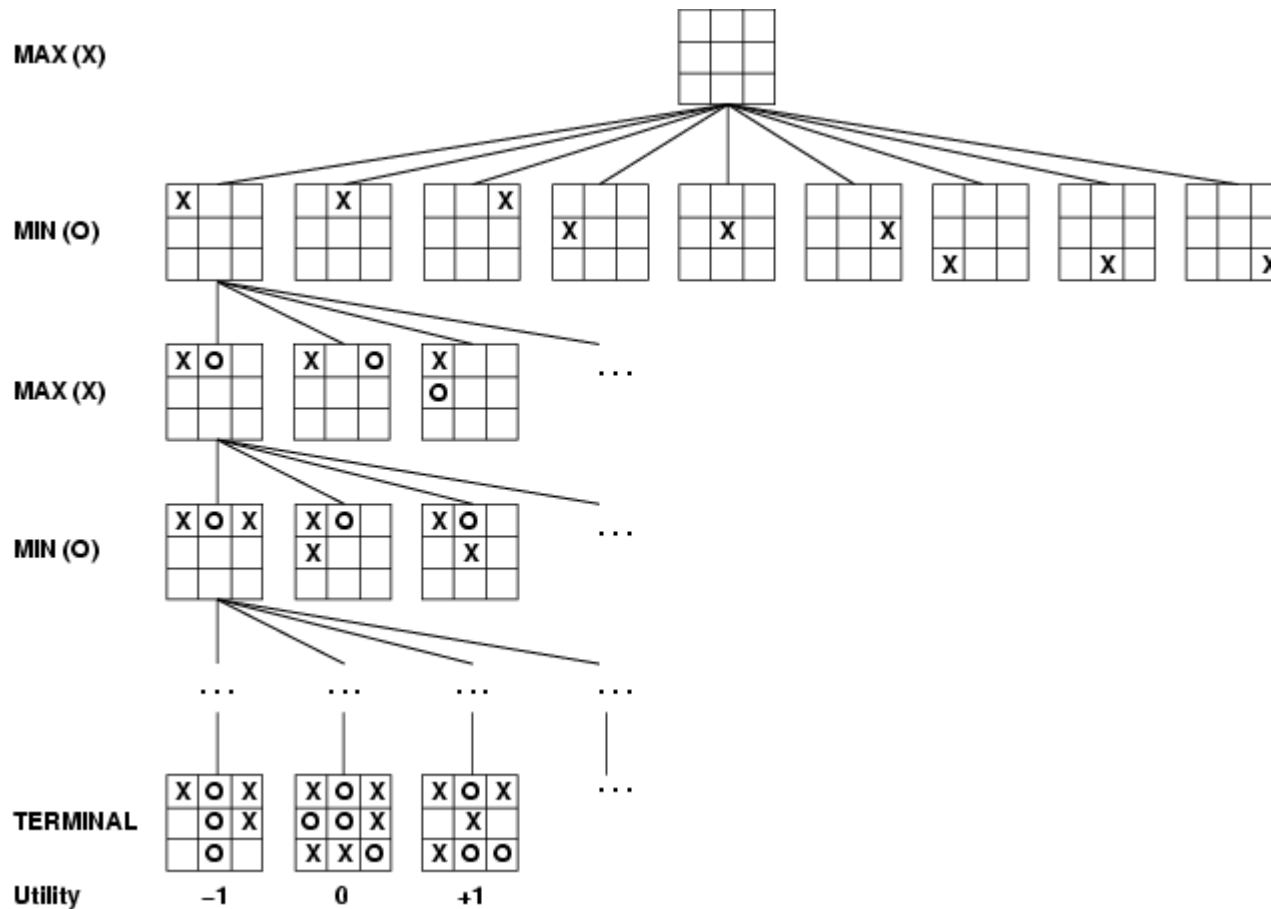
- Start with deterministic, perfect information games (easiest)
- Not considered:
  - Physical games like tennis, ice hockey, etc.
  - But, see “robot soccer,” <http://www.robocup.org/>

# Typical assumptions

- Two agents, whose actions alternate
- Utility values for each agent are the opposite of the other
  - “Zero-sum” game; this creates adversarial situation
- Fully observable environments
- In game theory terms:
  - Deterministic, turn-taking, zero-sum, perfect information
- Generalizes: stochastic, multiplayer, non zero-sum, etc.
- Compare to e.g., Prisoner’s Dilemma” (R&N pp. 666-668)
  - Non-turn-taking, Non-zero-sum, Imperfect information

# Game Tree (tic-tac-toe)

- All possible moves at each step



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

# Search versus Games

- **Search:** no adversary
  - Solution is a path from start to goal, or a series of actions from start to goal
  - Search, Heuristics, and constraint techniques can find optimal solution
  - Evaluation function: estimate cost from start to goal through a given node
  - Actions have costs (sum of step costs = path cost)
  - Examples: path planning, scheduling activities, ...
- **Games:** adversary
  - Solution is a 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, Go

# Games as search

- Two players, “MAX” and “MIN”
- MAX moves first, and they take turns until game is over
  - Winner gets reward, loser gets penalty
  - “Zero sum”: sum of reward and penalty is constant
- Formal definition as a search problem:
  - **Initial state**: set-up defined by rules, e.g., initial board for chess
  - **Player(s)**: which player has the move in state  $s$
  - **Actions(s)**: set of legal moves in a state
  - **Result(s,a)**: transition model defines result of a move
  - **Terminal-Test(s)**: true if the game is finished; false otherwise
  - **Utility(s,p)**: the 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

# Min-Max: an optimal procedure

- Finds the optimal strategy or next best move for MAX:
  - Optimal strategy is a solution tree

## Brute Force:

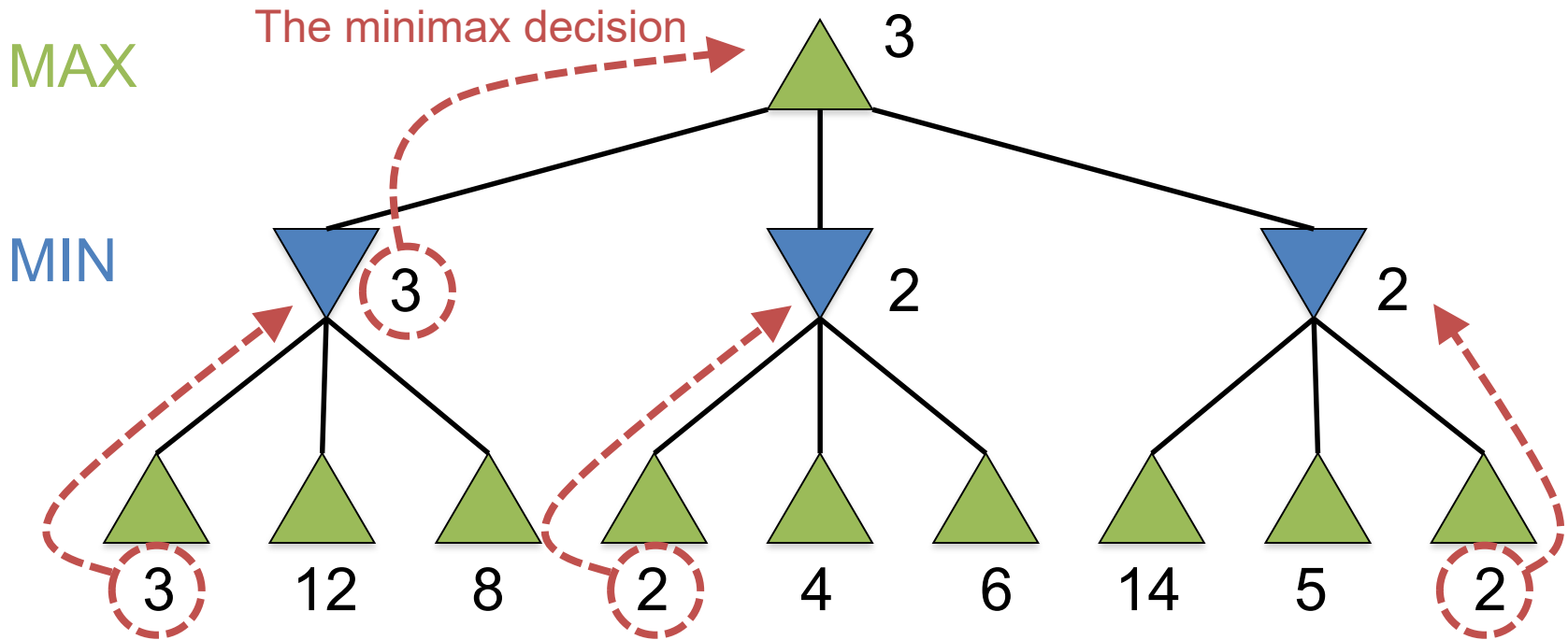
1. Generate the whole game tree to leaves
2. Apply utility (payoff) function to leaves
3. Back-up values from leaves toward the root:
  - a Max node computes the max of its child values
  - a Min node computes the min of its child values
4. At root: choose move leading to the child of highest value

## Minimax:

Search the game tree using DFS to find the value (= best move) at the root



# Two-ply Game Tree



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

# Recursive min-max search

`minMaxSearch(state)`

return argmax( [ minValue( apply(state,a) ) for each action a ] )

Simple stub to call recursion f' ns

`maxValue(state)`

if (`terminal(state)`) return `utility(state)`;

v = -infty

for each action a:

v = max( v, `minValue( apply(state,a) ) )`

return v

If recursion limit reached, eval position

Otherwise, find our best child:

`minValue(state)`

if (`terminal(state)`) return `utility(state)`;

v = infy

for each action a:

v = min( v, `maxValue( apply(state,a) ) )`

return v

If recursion limit reached, eval position

Otherwise, find the worst child:

# Properties of minimax

- **Complete?** Yes (if tree is finite)
- **Optimal?**
  - Yes (against an optimal opponent)
  - Can it be beaten by a suboptimal opponent? (No – why?)
- **Time?**  $O(b^m)$
- **Space?**
  - $O(bm)$  (depth-first search, generate all actions at once)
  - $O(m)$  (backtracking search, generate actions one at a time)

# Game tree size

- Tic-tac-toe

- $B \approx 5$  legal actions per state on average; total 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 is quite reasonable

- Chess

- $b \approx 35$  (approximate average branching factor)
- $d \approx 100$  (depth of game tree for “typical” game)
- $b^d = 35^{100} \approx 10^{154}$  nodes!!!
- Exact solution completely infeasible

It is usually impossible to develop the whole search tree.

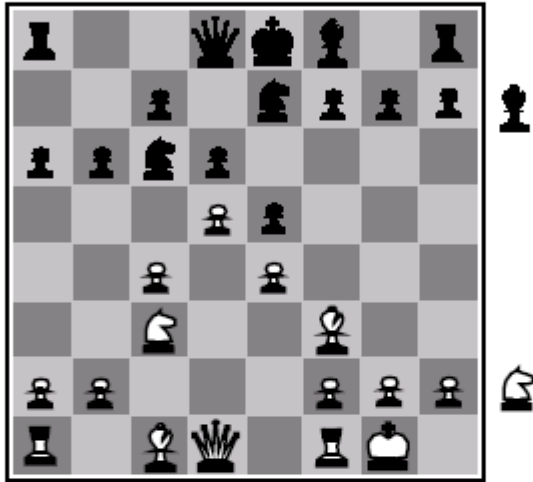
# Cutting off search

- One solution: cut off tree before game ends
- Replace
  - Terminal(s) with Cutoff(s) – e.g., stop at some max depth
  - Utility(s,p) with Eval(s,p) – estimate position quality
- Does it work in practice?
  - $b^m \approx 10^6$ ,  $b \approx 35 \rightarrow m \approx 4$
  - 4-ply look-ahead is a poor chess player
  - 4-ply  $\approx$  human novice
  - 8-ply  $\approx$  typical PC, human master
  - 12-ply  $\approx$  Deep Blue, human grand champion Kasparov
  - $35^{12} \approx 10^{18}$  (Yikes! but possible, with other clever methods)

# Static (Heuristic) Evaluation Functions

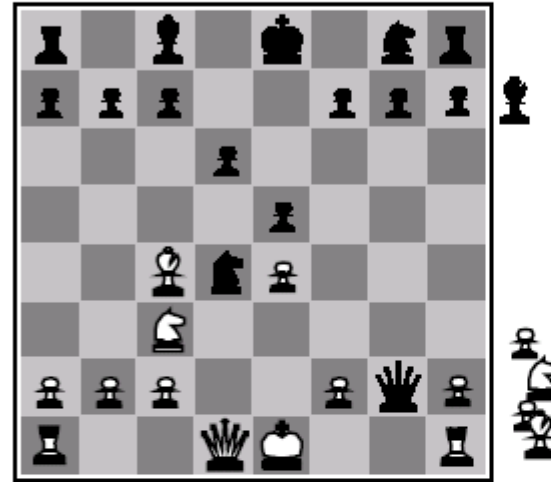
- **An Evaluation Function:**
  - Estimate how good the current board configuration is for a player.
  - Typically, evaluate how good it is for the player, and how good it is for the opponent, and subtract the opponent's score from the player's.
  - Often called "static" because it is called on a static board position
  - Ex: Othello: Number of white pieces - Number of black pieces
  - Ex: Chess: Value of all white pieces - Value of all black pieces
- Typical value ranges:  
[ -1, 1 ] (loss/win) or [ -1 , +1 ] or [ 0 , 1 ]
- Board evaluation: X for one player => -X for opponent
  - Zero-sum game: 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*

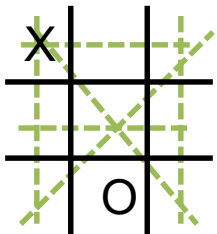
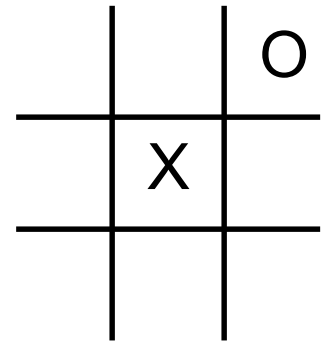
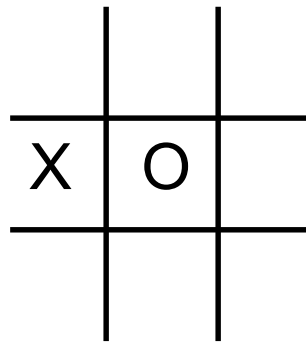
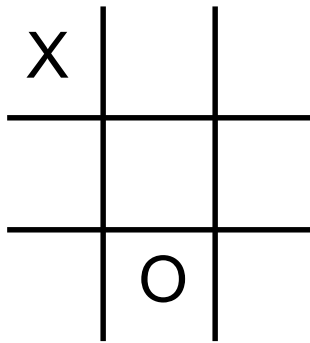
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$$

e.g.,  $w_1 = 9$  with

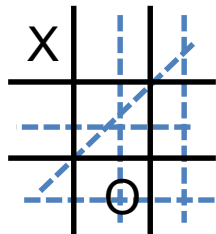
$f_1(s) = (\text{number of white queens}) - (\text{number of black queens}),$  etc.

# Applying minimax to tic-tac-toe

- The static heuristic evaluation function:
  - Count the number of possible win lines



X has 6  
possible win  
paths



O has 5  
possible win  
paths

$$E(s) = 6 - 5 = 1$$

X has 4 possible wins  
O has 6 possible wins

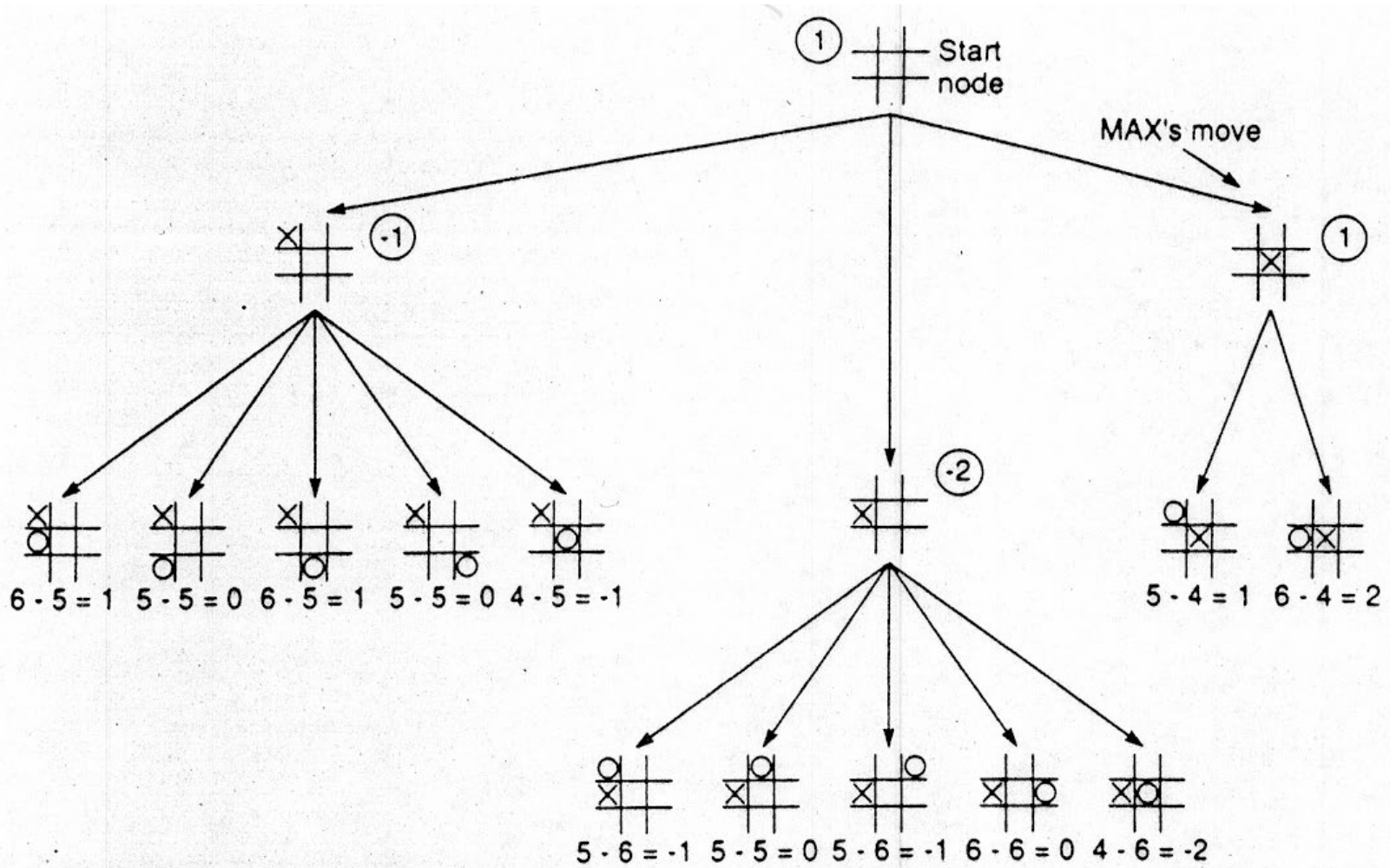
$$E(n) = 4 - 6 = -2$$

X has 5 possible wins  
O has 4 possible wins

$$E(n) = 5 - 4 = 1$$

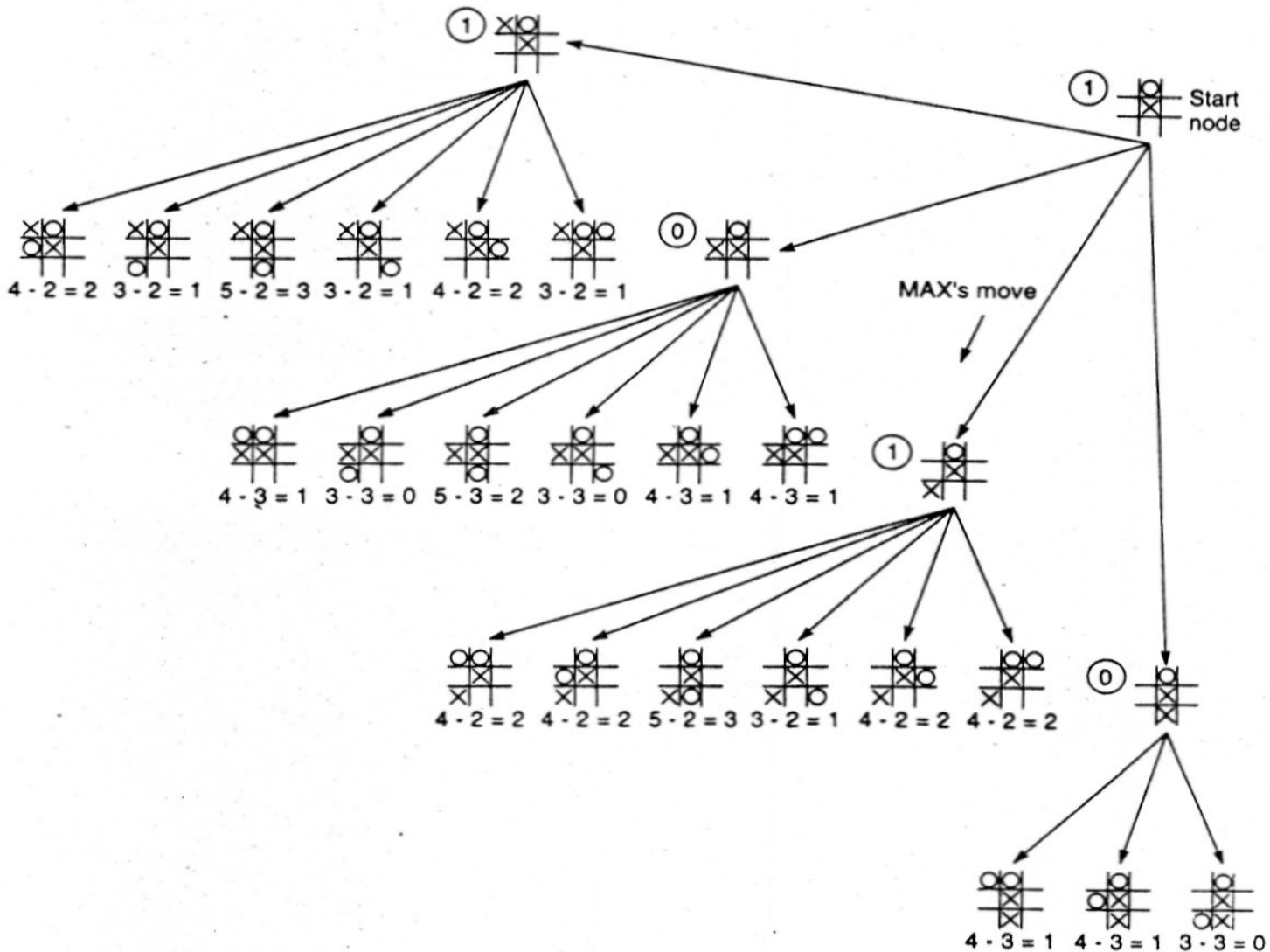


# Minimax values (two ply)



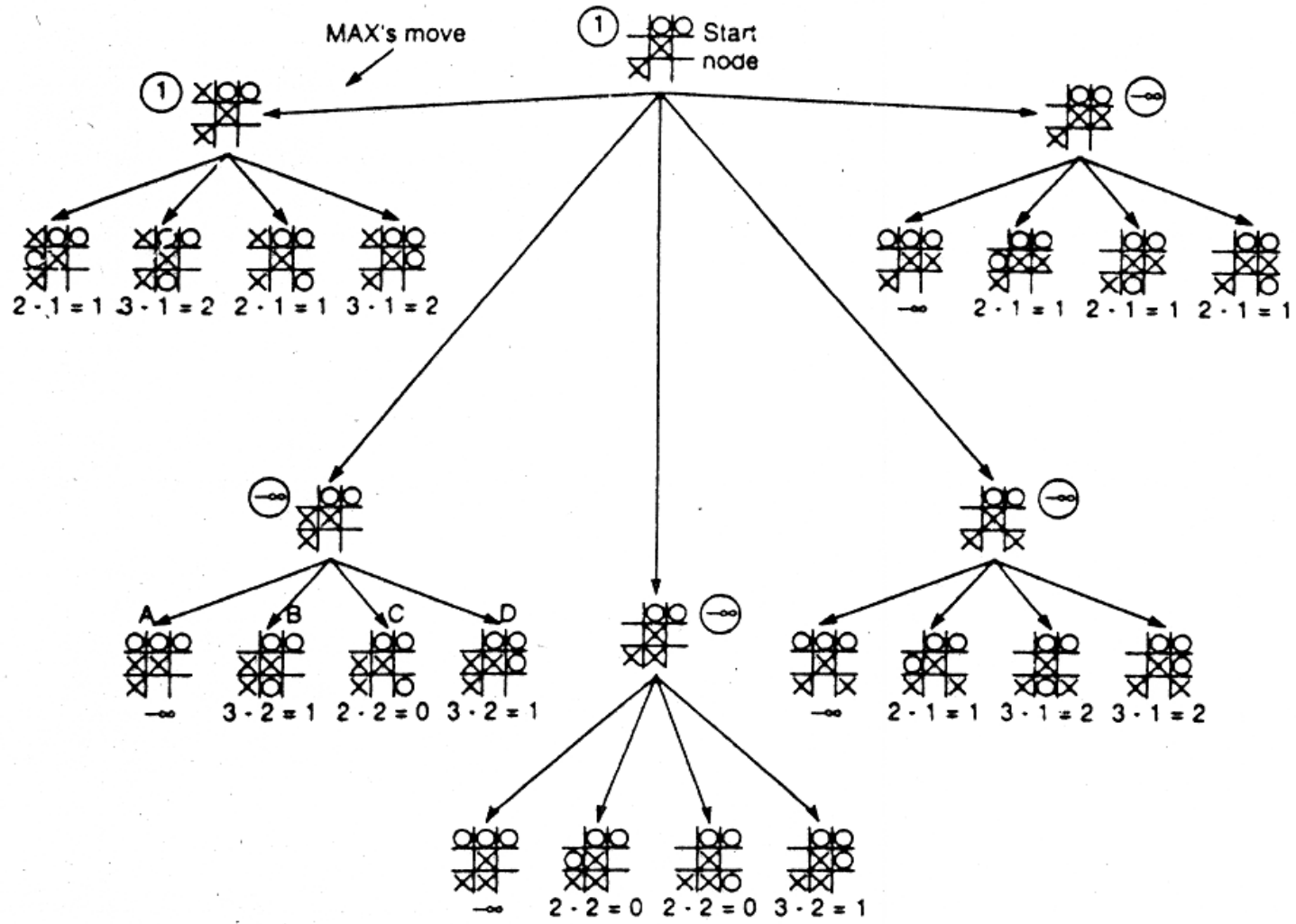
**Figure 4.17** Two-ply minimax applied to the opening move of tic-tac-toe.

# Minimax values (two ply)



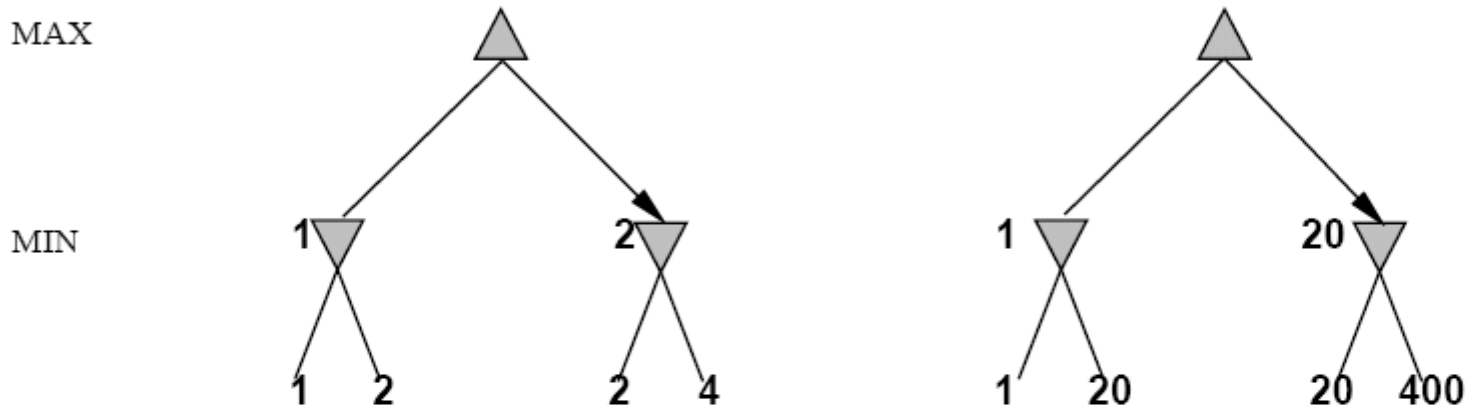
**Figure 4.18** Two-ply minimax applied to X's second move of tic-tac-toe.

# Minimax values (two ply)



**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

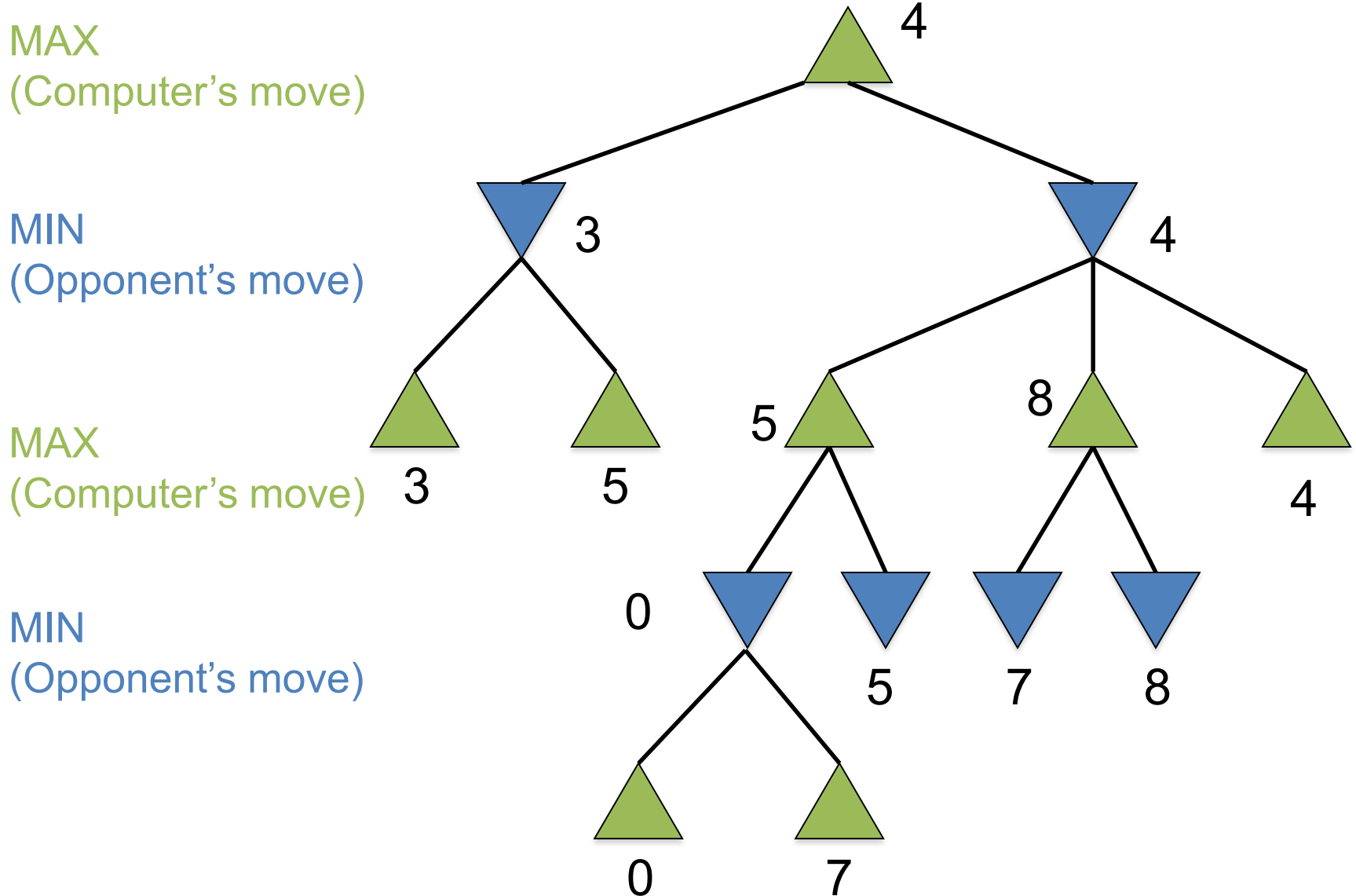
# Iterative deepening

- In real games, there is usually a time limit  $T$  to make a move
- How do we take this into account?
- Minimax cannot use “partial” results with any confidence, unless the full tree has been searched
  - Conservative: set small depth limit to guarantee finding a move in time  $< T$
  - But, we may finish early – could do more search!
- In practice, iterative deepening search (IDS) is used
  - IDS: depth-first search with increasing depth limit
  - When time runs out, use the solution from previous depth
  - With alpha-beta pruning (next), we can sort the nodes based on values from the previous depth limit in order to maximize pruning during the next depth limit => search deeper!

# Limited horizon effects

- The Horizon Effect
  - Sometimes there's a major "effect" (such as a piece being 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".
  - There are heuristics to try to follow certain branches more deeply to detect such important events
  - This helps to avoid catastrophic losses due to "short-sightedness"
- Heuristics for Tree Exploration
  - Often better to explore some branches more deeply in the 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  $10^{150} / 10^{40} \approx 10^{100}$  times
  - Vastly redundant search effort
- Can't remember all nodes (too many)
  - Can't eliminate all redundant nodes
- 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-KR4leads to the same position as
  1. P-QR4 P-KR4;      2. P-KR4 P-QR4



# Summary

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