CS4725/CS6705

Chapter 5: Adversarial Search

Adversarial search = games

- Multi-agent environments in which the agents' goals are in conflict
- Normally focused on deterministic, turn-taking, two-player, **zero-sum** games of **perfect information**
  - e.g., one player wins (+1) and one player loses (-1)

Example: chess

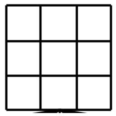
- Branching factor of about 35
- Games often last about 100 moves
- Search tree has 35^{100} (or 10^{154}) nodes (although the search graph has only 10^{40} distinct nodes)
- Calculating optimal decisions is infeasible

Optimal decisions in games

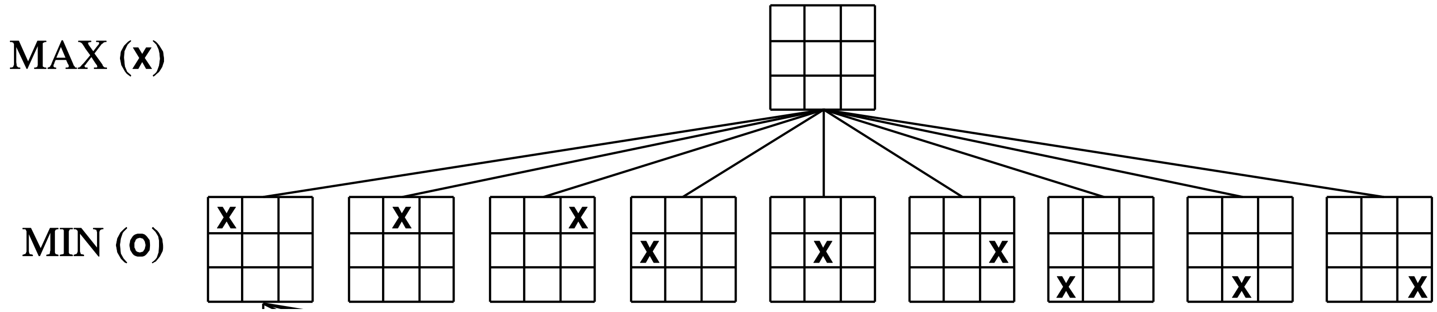
- Consider a two-player game, where MAX moves first, then MIN, then MAX, etc.
- Game definition:
  - **Initial state**: board position, whose move it is
  - **Successor function:** returns a list of (move,state) pairs for the current state
  - **Terminal test:** determines if the game is over
  - **Utility function:** provides a numerical value for terminal states (e.g., +1 for a win, -1 for a loss, 0 for a draw)
- Game tree (tic-tac-toe example on next slides)

Tic-tac-toe example from Russell&Norvig

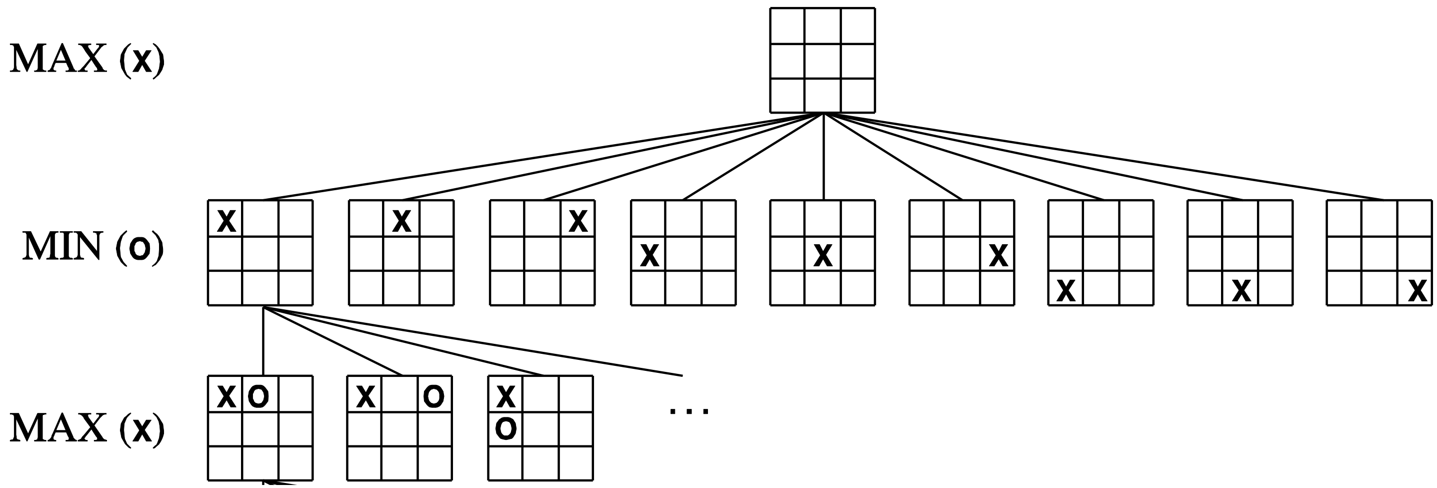
MAX (x)



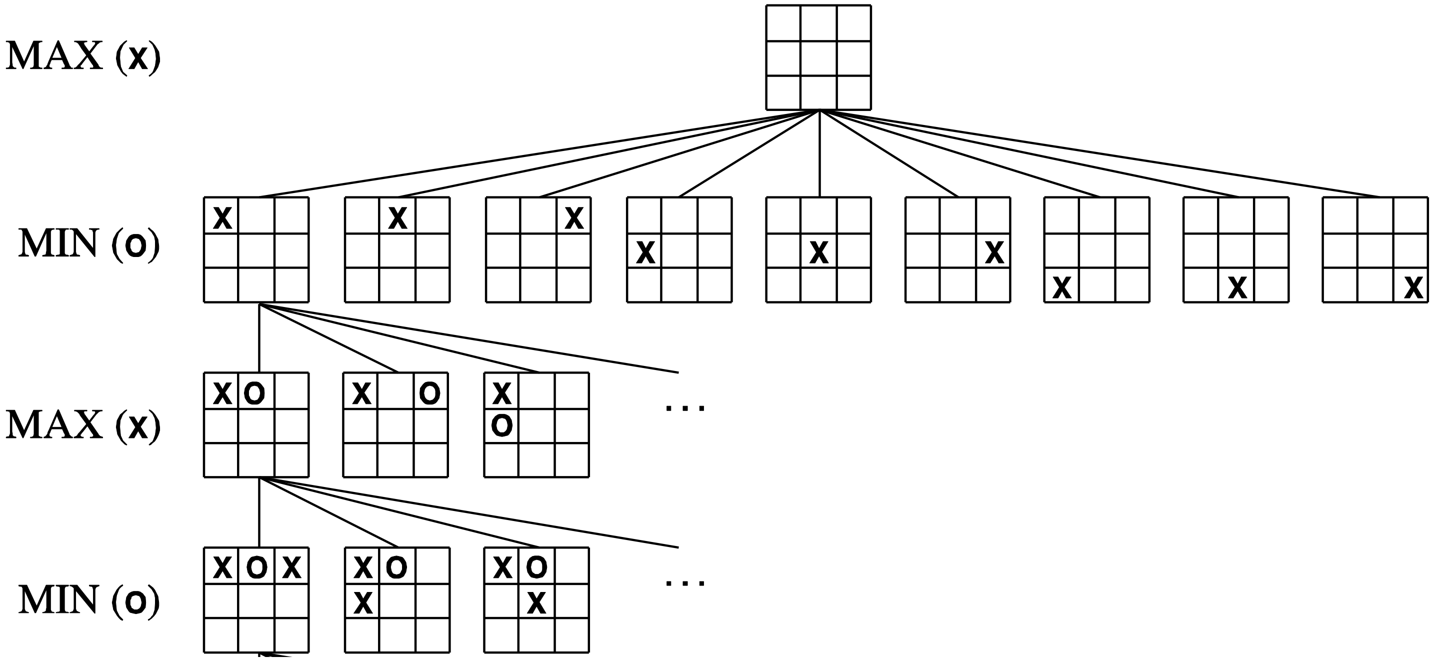
Tic-tac-toe example from Russell&Norvig



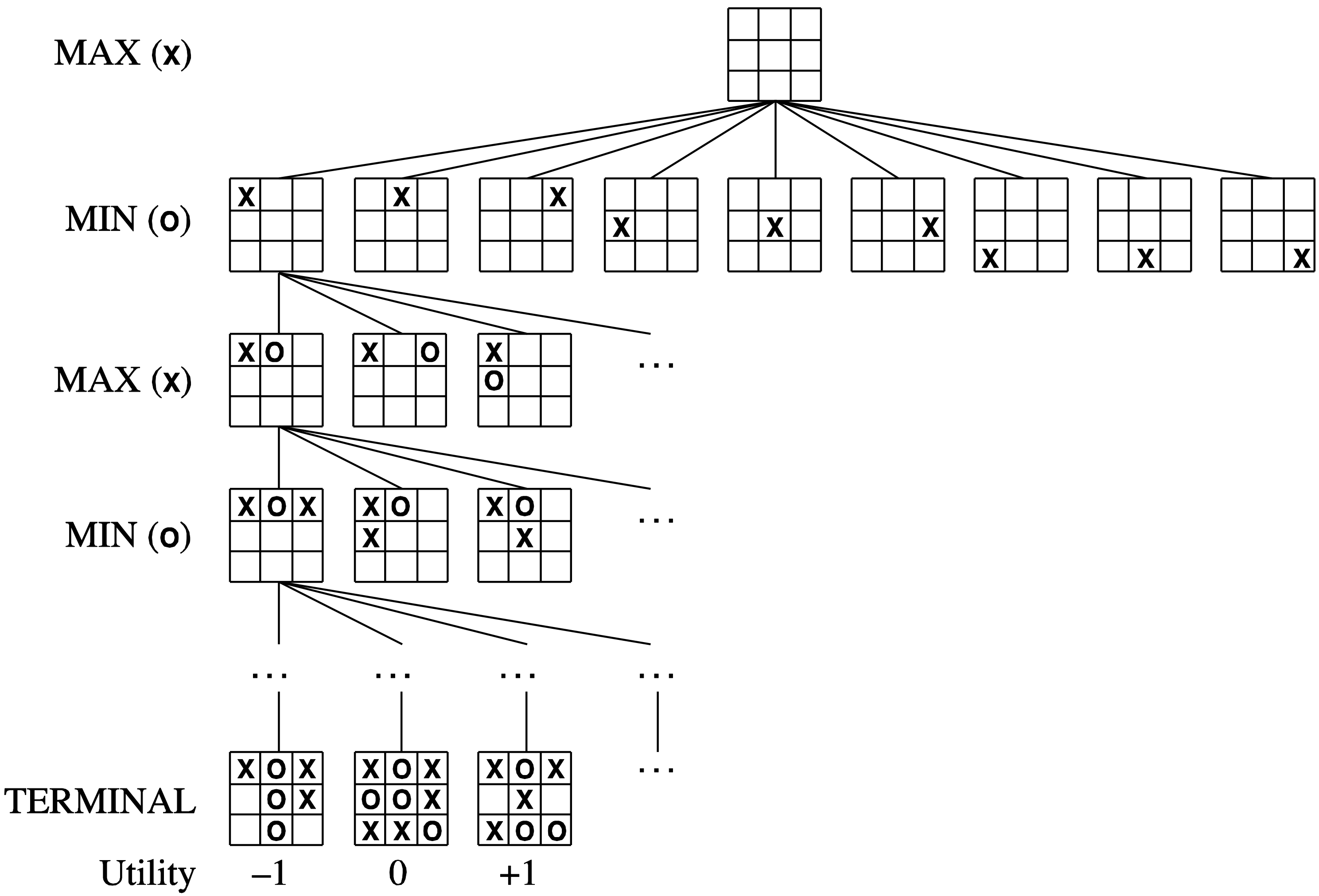
Tic-tac-toe example from Russell&Norvig



Tic-tac-toe example from Russell&Norvig



Tic-tac-toe example from Russell&Norvig



Optimal strategies

- We can’t just search for a sequence of moves that will lead us to a good solution.
- We must consider the fact that, in between our moves, the *opponent* will be making moves, working *against* our goal.

Minimax value

- Minimax value of a node: the utility (for MAX) of being in the current state, *assuming that both players play optimally* for the rest of the game
- Minimax value of node n=
  - Utility(n) if n is a terminal state
  - Max. minimax value of n's successors if n is a MAX node
  - Min. minimax value of n's successors if n is a MIN node
- [Simple game example on board]

Minimax algorithm

- Algorithm used to compute the minimax value of the current state, based on the definition on the previous slide
- Start from the leaves of the tree and back up
- Completely impractical for anything but the simplest games, but an important basis for the development of more practical algorithms

Alpha-beta pruning

- Minimax search: number of states is exponential in the number of moves
- Alpha-beta pruning: [simple example on board]
- In this example, we just saved the work of looking at 1 leaf node; in general, we can prune entire subtrees and save a significant amount of work

Alpha-beta pruning

- General idea:
  - Consider a node n in the tree that the player might move to.
  - If a player has a better choice m either at the parent node of n or at any choice point further up, then the player will never choose to go to node n.
  - [Diagram on board]

Alpha-beta pruning

- [Depth-first search] Consider all nodes along the current path in the tree
- Definitions:
  - α: the value of the best (highest) choice we have found so far for MAX at any choice point along the path
  - β: the value of the best (lowest) choice we have found so far for MIN at any choice point along the path

Alpha-beta pruning

- Some notation: For a given *state*,
  - Let ACTIONS(*state*) denote the set of actions available in that state.
  - Let RESULT(*state*, *action*) be the state that results from performing *action* in *state.*
  - TERMINAL-TEST(*state*) is true if *state* is a terminal state (has no successors / represents the end of the game).

Alpha-beta pruning (continued)

- Perform the following algorithm:

function ALPHA-BETA-SEARCH(*state*) **returns** an action

  v ← MAX-VALUE(*state*, -∞, +∞)

  return the *action* in ACTIONS(*state*) that has value *v*

(See the next two slides for MAX-VALUE and MIN-VALUE pseudocode.)

Alpha-beta pruning (continued)

function MAX-VALUE(*state*, α, β) **returns** a *utility value*

  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

  $v \leftarrow -\infty$

  for each *action* in ACTIONS(*state*) do

    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(*state,action*), α, β))

    if $v \geq \beta$ **then return** v // ignore/prune all other actions

    $\alpha \leftarrow$ MAX(α, v)

return v

Alpha-beta pruning (continued)

function MIN-VALUE(*state*, α, β) **returns** a *utility value*

  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

  $v \leftarrow +\infty$

  for each *action* in ACTIONS(*state*) do

    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(*state,action*), α, β))

    if $v \leq \alpha$ **then return** v // ignore/prune all other actions

    $\beta \leftarrow$ MIN(β, v)

return v

Alpha-beta pruning (continued)

- [Detailed example on board]

Effectiveness of alpha-beta pruning

- Depends on the order in which successors are checked
- If we do a good job of ordering successors:
  - Only about $O(b^{d/2})$ nodes have to be checked instead of $O(b^d)$
  - Therefore, we can look ahead about twice as far in the same amount of time
  - Example: effective branching factor for chess becomes 6 instead of 35

Imperfect, real-time decisions

- Even with alpha-beta pruning, the search space for most real games is far too large to find perfect game strategies.
- One approach: cut off the search tree at a reasonable depth and apply an **evaluation function** to leaf nodes:
  - An estimate of how "good" each state is
  - Chess example

Games involving chance

- Game tree must be adjusted to include chance nodes
- [Backgammon example on board]
- Instead of finding definite minimax values for nodes, we have to calculate the **expected value** over all possible results of the chance event (e.g., rolling dice)
- **expectiminimax value**

Expectiminimax value

- Expectiminimax value of node n =
  - Utility(n) if n is a terminal state
  - Max. expectiminimax value of n's successors if n is a MAX node
  - Min. expectiminimax value of n's successors if n is a MIN node
  - If n is a chance node... sum (over all successors s of n) of:

    Probability(s) x expectiminimax(s)

- [Simple example on board]