Adversarial Search and Games

Game Theory

- For each state where we choose to stop searching, we ask who is winning?

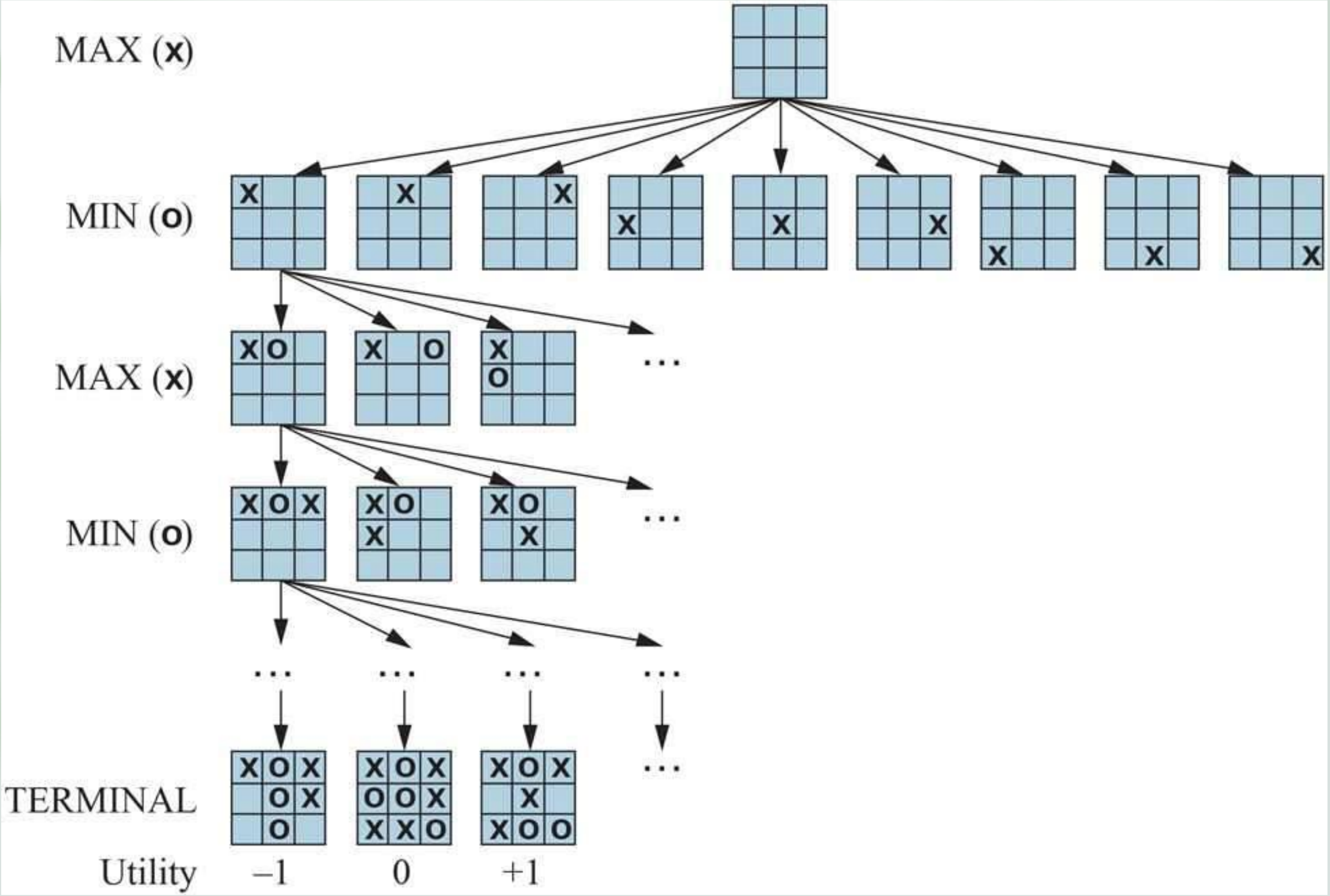
- Apply heuristic evaluation function to estimate who is winning based on features of the state.
- We can average the outcomes of many fast simulations of the game from that state all the way to the end.

Two-player zero-sum games

- Deterministic, two-player, turn-taking, perfect information, zero-sum games.
- "Perfect information" is a synonym for "fully observable".
- "zero-sum" means that what is good for one player is just as bad for the other: there is no "win-win" outcome.

Defining a game

- S_o = The initial state, which specifies how the game is set up at the start.
- TO-MOVE(s): The player whose turn it is to move in state s.
- ACTIONS(s) : The set of legal moves in state s.
- RESULT(s,a) : The transition model, which defines the state resulting from taking action a in state s.
- IS-TERMINAL(s) : A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.
- UTILITY(s,p) : A utility function (also called an objective function or payoff function), which defines the final numeric value to player p when the game ends in terminal state s. In chess, the outcome is a win, loss, or draw, with values 1, 0, or ½.

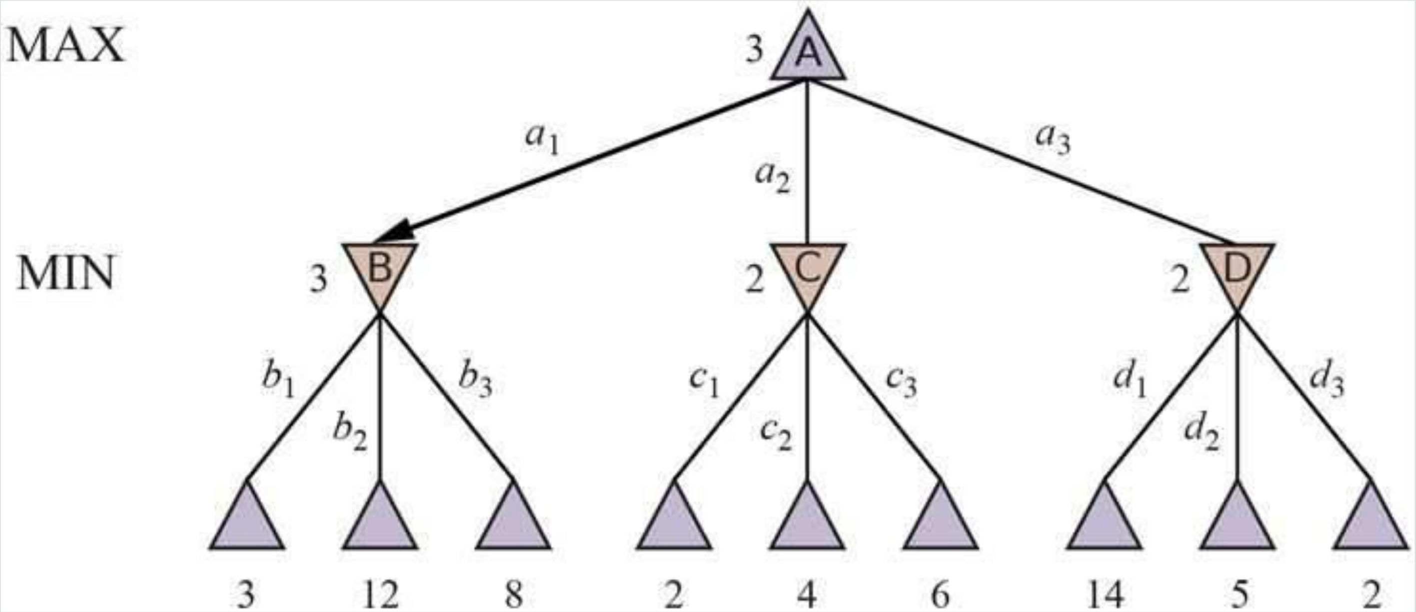


Tic-Tac-Toe

Optimal Decisions in Games

- Condition plan or Sequence of action?
- A contingent strategy specifying a response to each of MIN's possible moves.
- Identical to the definition of a solution for a nondeterministic planning problem.
- In games that have a binary outcome (win or lose), we could use AND–OR search.

Minimax search



$$
\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s, \textsc{max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,\ a)) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,\ a)) & \text{if } \textsc{To-Move}(s) = \textsc{min} \end{cases}
$$

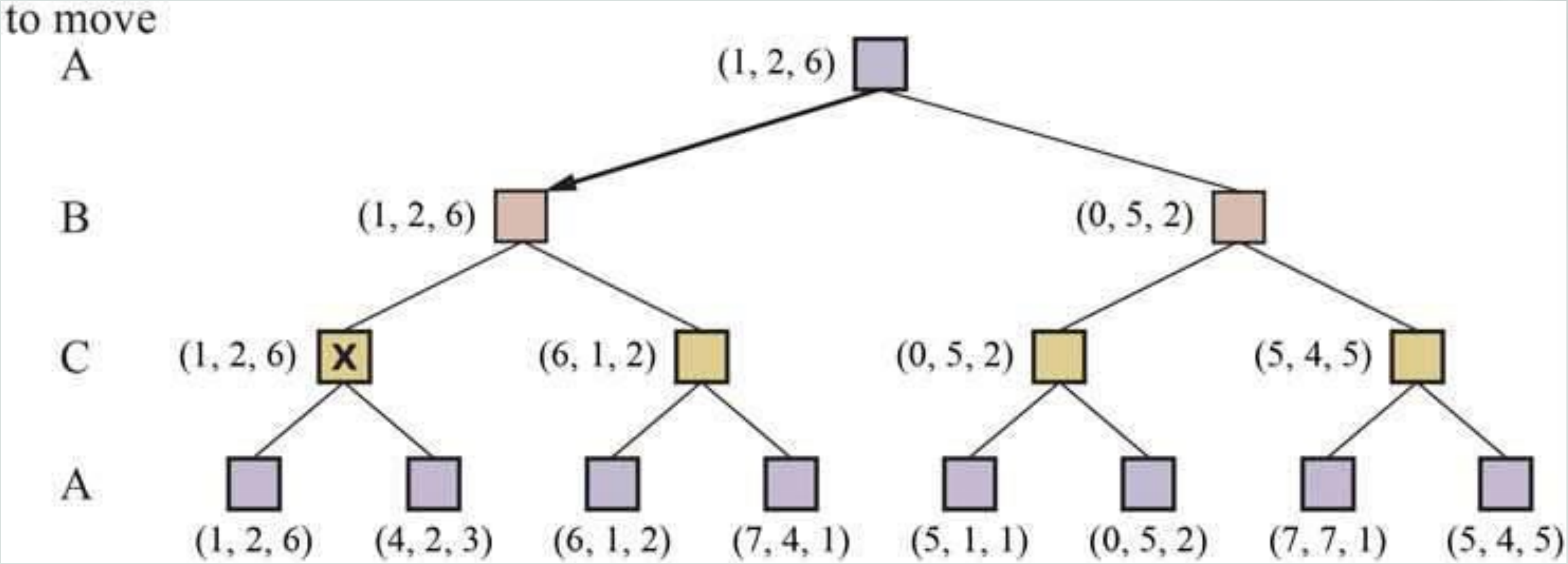
Minimax search algorithm

```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Optimal decisions in multiplayer games

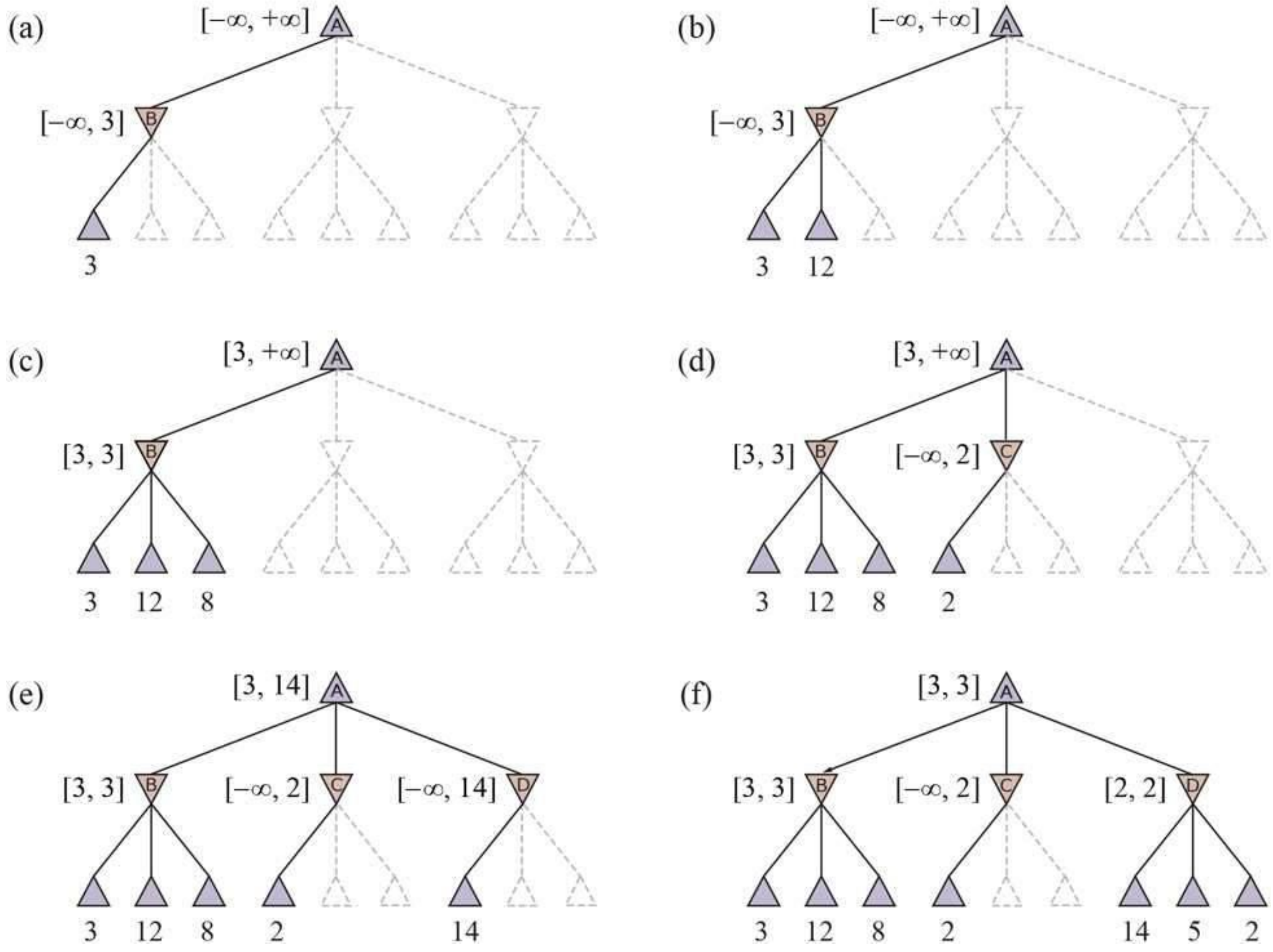


Alpha–Beta Pruning

- MAX-VALUE (state,α,β)
- α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX. Think: α = "at least."
- β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN. Think: β = "at most."

Alpha–Beta Pruning

$$
\begin{aligned}
\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3,z,2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\
&= 3.
\end{aligned}
$$



Alpha–Beta Pruning

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```