Introduction to Artificial Intelligence

Lecture: Adversarial Search

Outline

- Games
- Optimal Decisions in Games
- α-β Pruning
- Imperfect, Real-time Decisions
- Stochastic Games

Multiagent environments

- Each agent needs to consider the actions of other agents and how they affect its own welfare.
- The unpredictability of other agents introduce contingencies into the agent’s problem-solving process
- Game theory views any multiagent environment as a game.
  - The impact of each agent on the others is “significant,” regardless of whether the agents are cooperative or competitive.
- Types of games:
  - Perfect information vs Imperfect information
  - Deterministic vs Chance

A sequential game has perfect information if each player, when making any decision, is perfectly informed of all the events that have previously occurred, including the "initialization event" of the game.

Types of game

	Deterministic	Chance
Perfect information	Chess, Checkers, Go, Othello	Backgammon, Monopoly
Imperfect information		Bridge, poker, scrabble nuclear war





Adversarial search

- Adversarial search (known as games) covers competitive environments in which the agents' goals are in conflict.
- Zero-sum games of perfect information
    - Deterministic, fully observable environments, turn-taking, two-player
    - The utility values at the end are always equal and opposite.

Primary assumptions

- Two players only, called MAX and MIN.
  - MAX moves first, and then they take turns moving until the game ends
  - Winner gets reward, loser gets penalty.
- Both players have complete knowledge of the game’s state
- No element of chance
- Zero-sum games
  - The total payoff to all players is the same for every game instance.
- Rational players
  - Each player always tries to maximize his/her utility

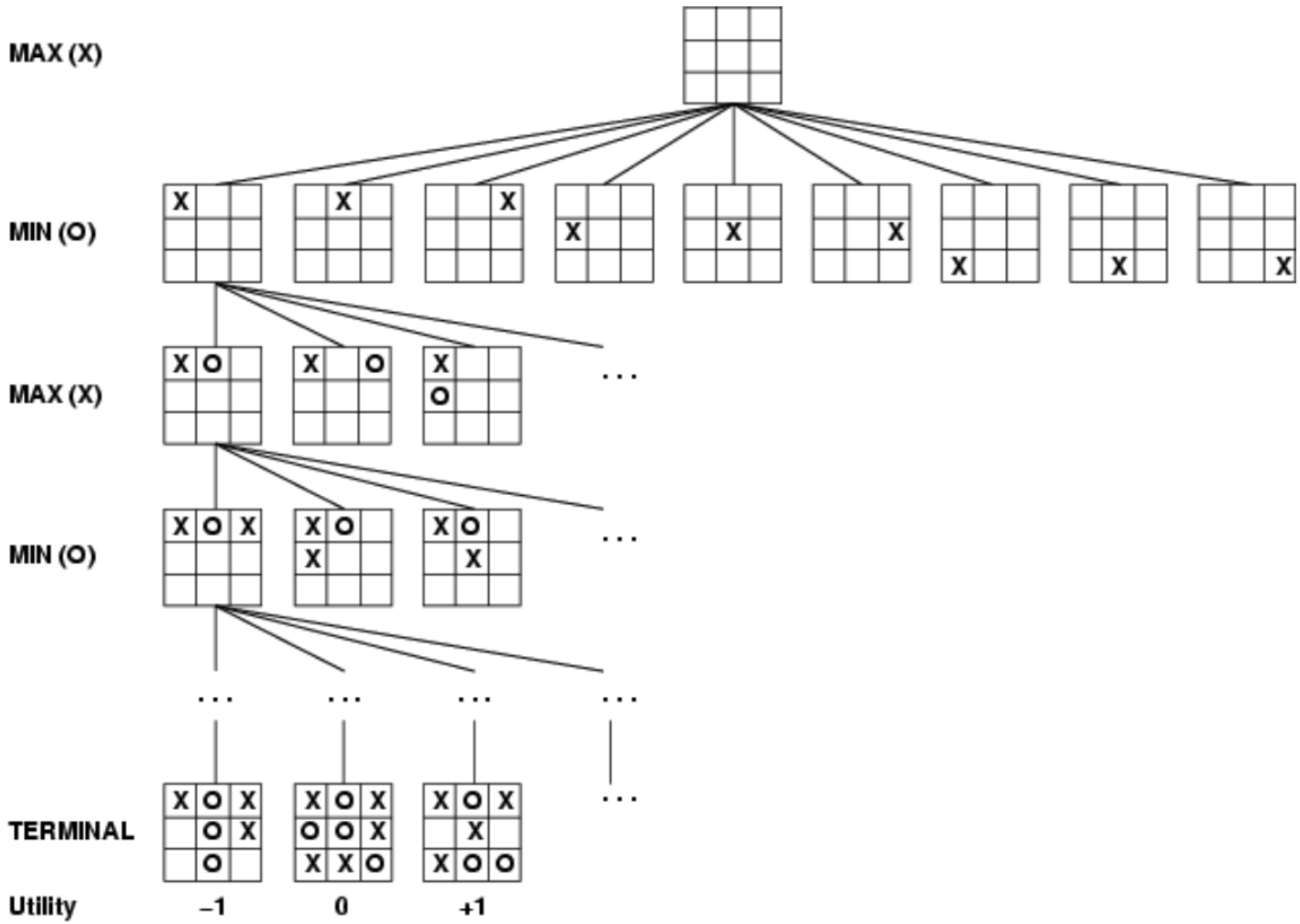
Games as search

- S_0 – Initial state: How the game is set up at the start 0.
- $PLAYER(s)$: Which player has the move in a state, MAX/MIN?
- $ACTIONS(s)$ – Successor function: A list of (move, state) pairs specifying legal moves.
- $RESULT(s, a)$ – Transition model: Result of move a on state s
- $TERMINAL - TEST(s)$: Is the game finished?
- $UTILITY(s,p)$ – Utility function: A numerical value of a terminal state s for a player p

Games vs. Search problems

- Complexity: games are too hard to be solved
- Time limits: make some decision even when calculating the optimal decision is infeasible
- Efficiency: penalize inefficiency severely
  - Several interesting ideas on how to make the best possible use of time are spawn.

Search Tree of Tic-Tac-Toe



Optimal decision in games

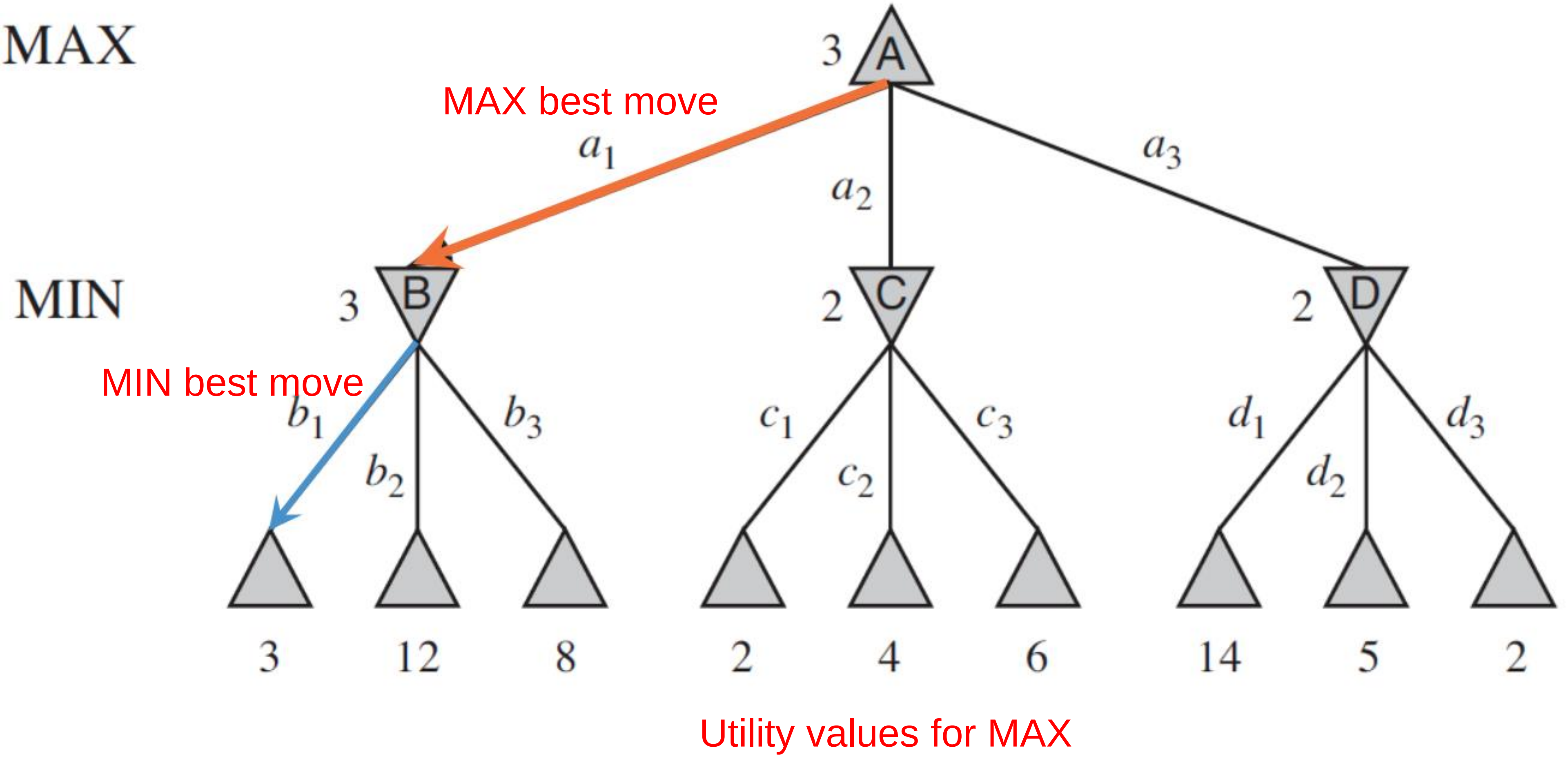
- Normal search problem
  - Optimal solution is a sequence of action leading to a goal state.
- Games
  - A search path that guarantee win for a player
  - The optimal strategy can be determined from the minimax value of each node
- MINIMAX(s) =

UTILITY(s) if TERMINAL-TEST(s)

$\max_{a \in Actions(s)}$ MINIMAX(RESULT(s, a)) if PLAYER(s) = MAX

min MINIMAX(RESULT(s, a)) if

Example



MAX
MIN
3
A
MAX best move
a_1
a_2
a_3
3
B
2
C
2
D
MIN best move
b_1
b_2
b_3
c_1
c_2
c_3
d_1
d_2
d_3
3
12
8
2
4
6
14
5
2
Utility values for MAX

The minimax algorithm

- Compute the minimax decision from the current state
- Use a simple recursive computation of the minimax values of each successor state
    - The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds.

The minimax algorithm

function MINIMAX-DECISION(state) **returns** an action
    return $\mathrm{argmax}_{a \in \mathrm{ACTIONS}(s)}$MIN-VALUE(RESULT(state, a))

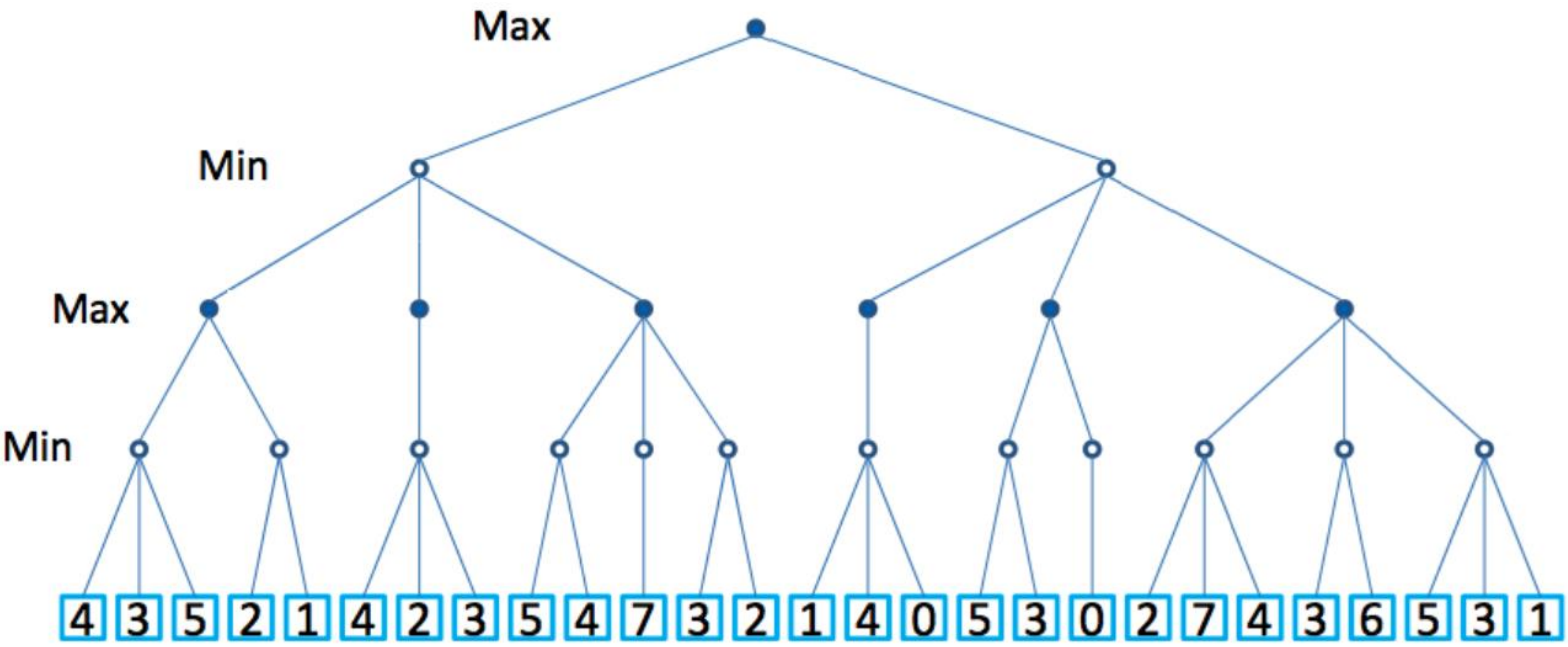
function MAX-VALUE(state) **returns** a utility value
    if TERMINAL-TEST(state) **then return** UTILITY(state)
    v ← -∞
    for each a **in** ACTIONS(state) **do**
        v ← MAX(v, MIN-VALUE(RESULT(s, a)))
    return v

function MIN-VALUE(state) **returns** a utility value
    if TERMINAL-TEST(state) **then return** UTILITY(state)
    v←∞
    for each a **in** ACTIONS(state) **do**
        v ← MIN(v, MAX-VALUE(RESULT(s, a)))
    return v

The minimax algorithm

- A complete depth-first exploration of the game tree
- Completeness
  - Yes (if tree is finite)
- Optimality
  - Yes (against an optimal opponent)
- Time complexity
  - $O(b^m)$
- Space complexity
  - $O(bm)$ (depth-first exploration)

The minimax algorithm



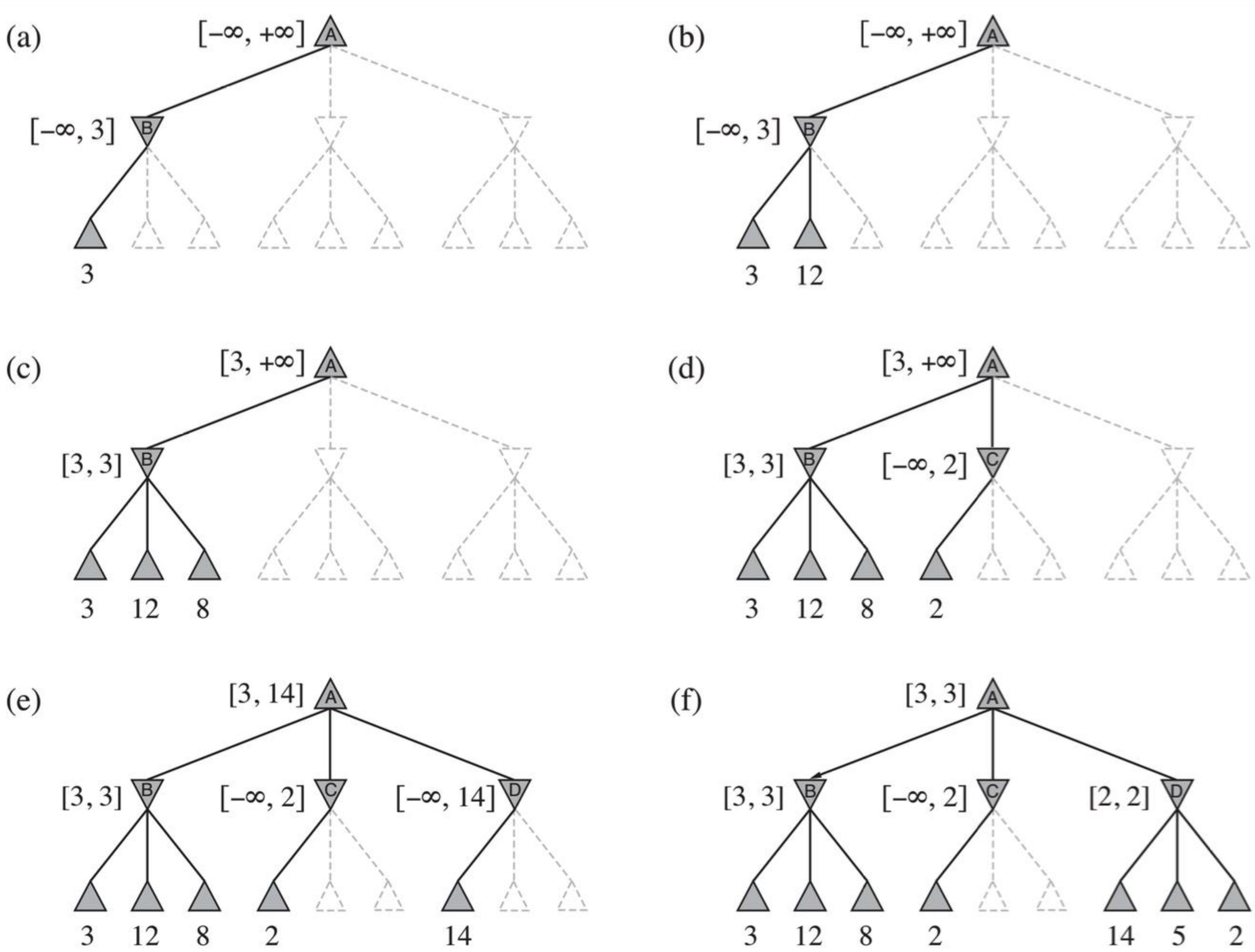
Problem with minimax search

- The number of game states is exponential in the tree's depth

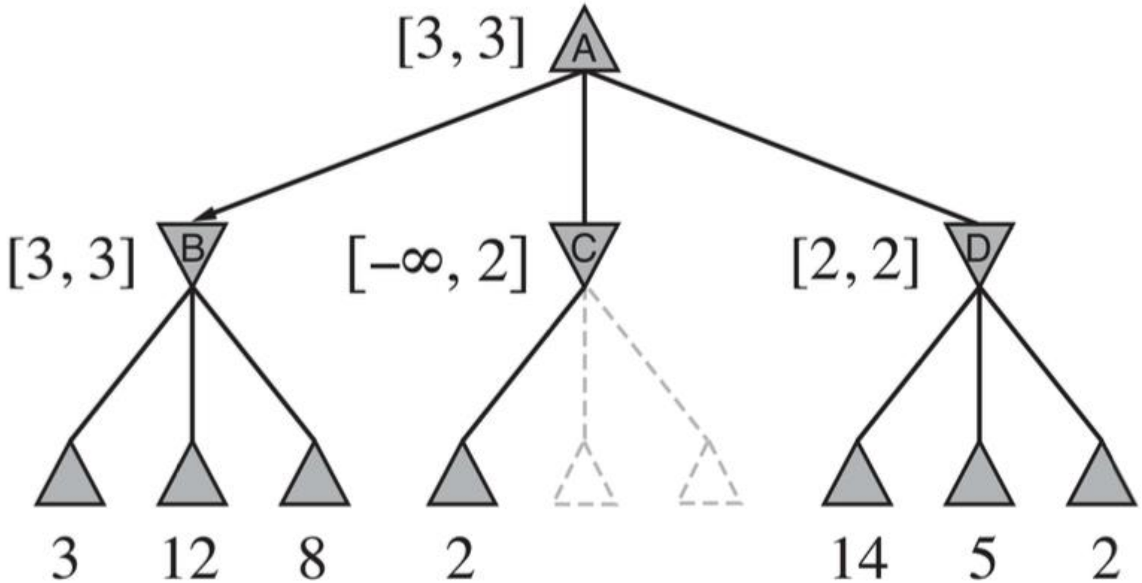
  → Do not examine every node

- Alpha-beta pruning: Prune away branches that cannot possibly influence the final decision
- Bounded lookahead
  - Limit depth for each search
  - This is what chess players do: look ahead for a few moves and see what looks best

Alpha-beta pruning



Alpha-beta pruning



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

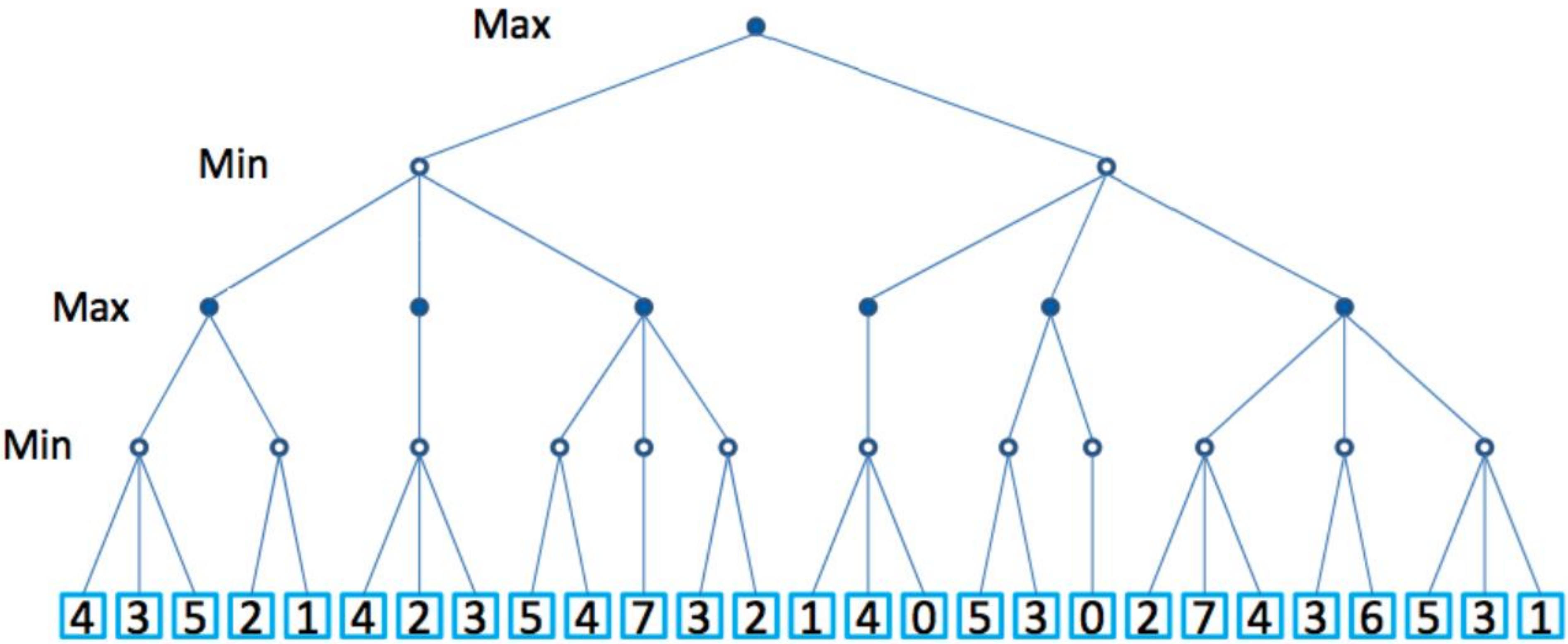
Alpha-beta pruning

```
function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state,-∞,+∞)
    return the action in ACTIONS(state) with value v

function MAX-VALUE(state,α,β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← -∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s,a),α,β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v

function MIN-VALUE(state,α,β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s,a) ,α,β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

Alpha-beta pruning

- Pruning does not affect the result
- Good move ordering improves effectiveness of pruning
- Killer move heuristic
- Transposition table avoids re-evaluation a state

Alpha-beta pruning



Heuristic minimax

- Both minimax and alpha-beta pruning search all the way to terminal states.
  - This depth is usually impractical because moves must be made in a reasonable amount of time (~ minutes).
- Cut off the search earlier with some depth limit
- Use an evaluation function

$$\text{H-MINIMAX}(s, d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s, d) \\ \max_{a \in \text{ACTIONS}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{ACTIONS}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d+1) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Evaluation Functions

- The evaluation function should order the terminal states in the same way as the true utility function does
  - States that are wins must evaluate better than draws, which in turn must be better than losses.
- The computation must not take too long!
- For nonterminal states, their orders should be strongly correlated with the actual chances of winning.

Cutting off search

- Minimax Cutoff is identical to Minimax Value except
  - $Terminal$? is replaced by $Cutoff$?
  - $Utility$ is replaced by $Eval$

if CUTOFF-TEST(state, depth) **then return** EVAL(state)

Stochastic Games

- Uncertain outcomes controlled by chance, not an adversary!
- Why wouldn’t we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Actions can fail: when moving a robot, wheels might slip

Expectimax search

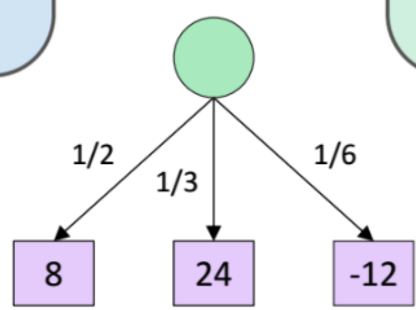
- Values reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes
- Expectimax search: compute the average score under optimal play
  - Max nodes as in minimax search
  - Chance nodes are like min nodes, but the outcome is uncertain
  - Calculate expected utilities, i.e. take weighted average of children
- The underlying uncertain-result problems can be formulated as Markov Decision Processes

Expectimax search

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```



$v = (1/2)\ (8) + (1/3)\ (24) + (1/6)\ (-12) = 10$

Expectimax search

- It is possible to perform pruning in expectimax search.
- Common techniques for pruning in expectimax include:
  - **Depth Limiting**: Limiting the depth of the search tree can effectively prune branches that are too deep to be practically explored.
  - **Evaluation Function**: Using an evaluation function to estimate the value of a state without fully exploring its subtree. If the evaluation function indicates that further exploration is unlikely to yield significant improvements, you can prune the subtree.
  - **Probabilistic Pruning**: In scenarios where probabilities are involved, you might prune branches that have very low probabilities of occurring, as they contribute little to the overall expectation.
  - **Iterative Deepening**: Iterative deepening can be combined with pruning techniques to explore deeper parts of the tree only when necessary, based on the current state of the search.

References

- Stuart Russell and Peter Norvig. 2009. Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.
- Lê Hoài Bắc, Tô Hoài Việt. 2014. Giáo trình Cơ sở Trí tuệ nhân tạo. Khoa Công nghệ Thông tin. Trường ĐH Khoa học Tự nhiên, ĐHQG-HCM.
- Nguyễn Ngọc Thảo, Nguyễn Hải Minh. 2020. Bài giảng Cơ sở Trí tuệ Nhân tạo. Khoa Công nghệ Thông tin. Trường ĐH Khoa học Tự nhiên, ĐHQG-HCM.