Chapter 5: Adversarial Search

These slides are adapted from AI Berkeley course and Russell and Norvig textbook.

Adversarial Search (Games)

- Agents are in **competitive** environments (the agents' goals are in conflict).
- Types of Games
  - Deterministic vs. stochastic
  - One, two, or more players
  - Zero sum

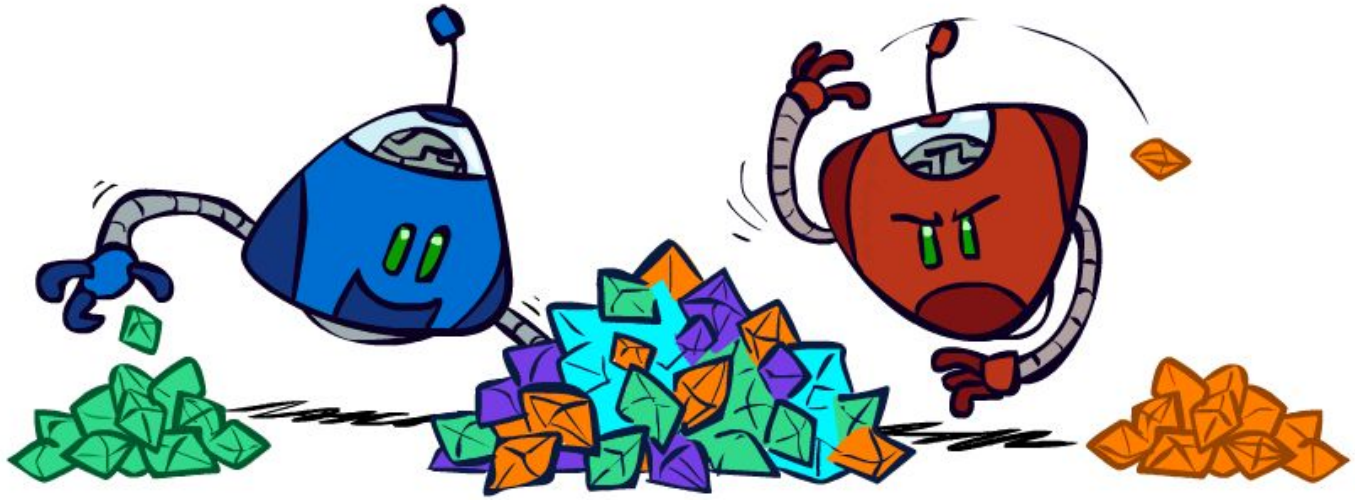
    The utility values at the end of the game are always equal and opposite like chess.
  - Perfect information (deterministic and fully-observable)

Game Formulation

- A game can be formally defined as a kind of search problem with the following elements:
  - Initial State S_0 which specifies how the game is set up at the start.
  - Players: P={1...N}: Defines which player has the move in a state.
  - Actions: A: Returns the set of legal moves in a state.
  - Transition Function: SxA $\longrightarrow$ S
  - Terminal Test: S $\longrightarrow$ {true if the game is over, otherwise false}
    - States where the game has ended are called **terminal states**.
  - Utilities: SxP $\longrightarrow$ R
    - A utility function defines the final numeric value for a game that ends in a terminal state s for a player p.
    - For example, in chess, the outcome is a win, loss, or draw, with values +1, 0, or 1/2.

- The game solution for a player is a strategy: S $\longrightarrow$ A to define its move for a given state.
- An **optimal** strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

Zero-Sum Games



Zero-Sum Games

- Agents have opposite utilities (values on outcomes)
- Lets us think of a single value that one maximizes and the other minimizes
- Adversarial, pure competition

General Games

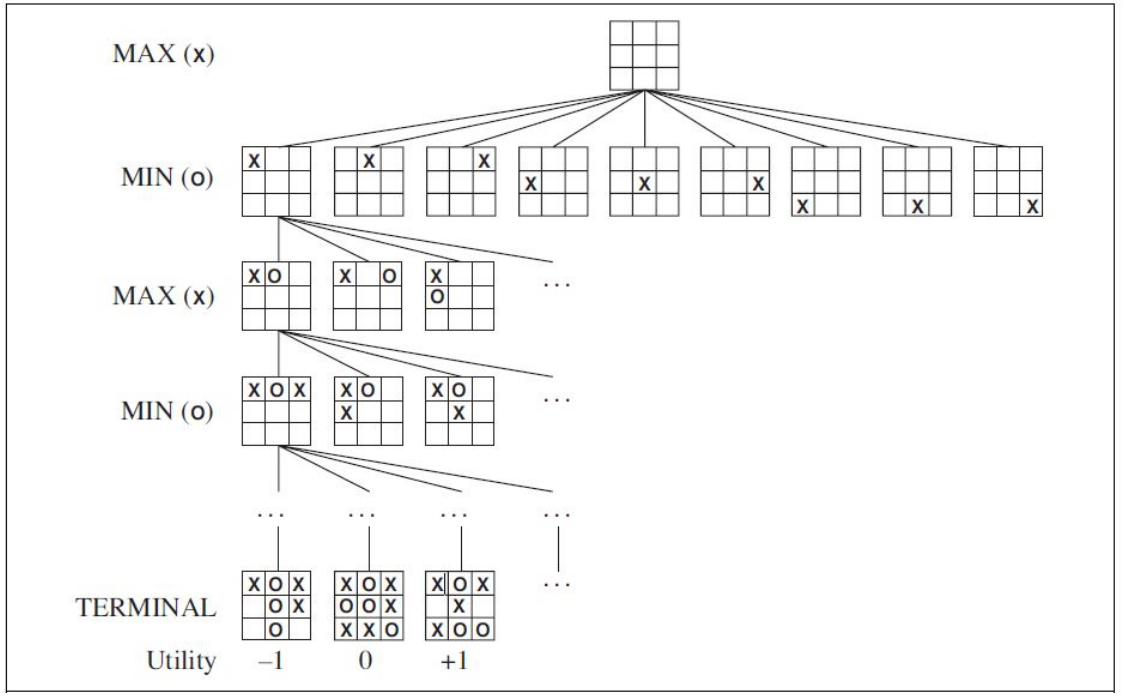
- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible

Deterministic Zero-sum Game Setup

- We first consider games with two players (MAX and MIN)
- MAX moves first, and then they take turns moving until the game is over.
- At the end of the game, points are awarded to the winning player and penalties are given to the loser.

Partial Game Tree Example

- Tic-tac-toe's game tree has $9! = 362,880$ terminal nodes.
- For chess there are over 10^{40} nodes!!



- How a player could choose an optimal move?!
  - Examine enough nodes to allow him to determine what move to make as we will see ☺️

Minimax values

Given a game tree, the optimal strategy can be determined from the **minimax value**

of each node, which we write as MINIMAX(n).

- The minimax value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally* from there to the end of the game.
- The minimax value for a state s is calculated as:

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

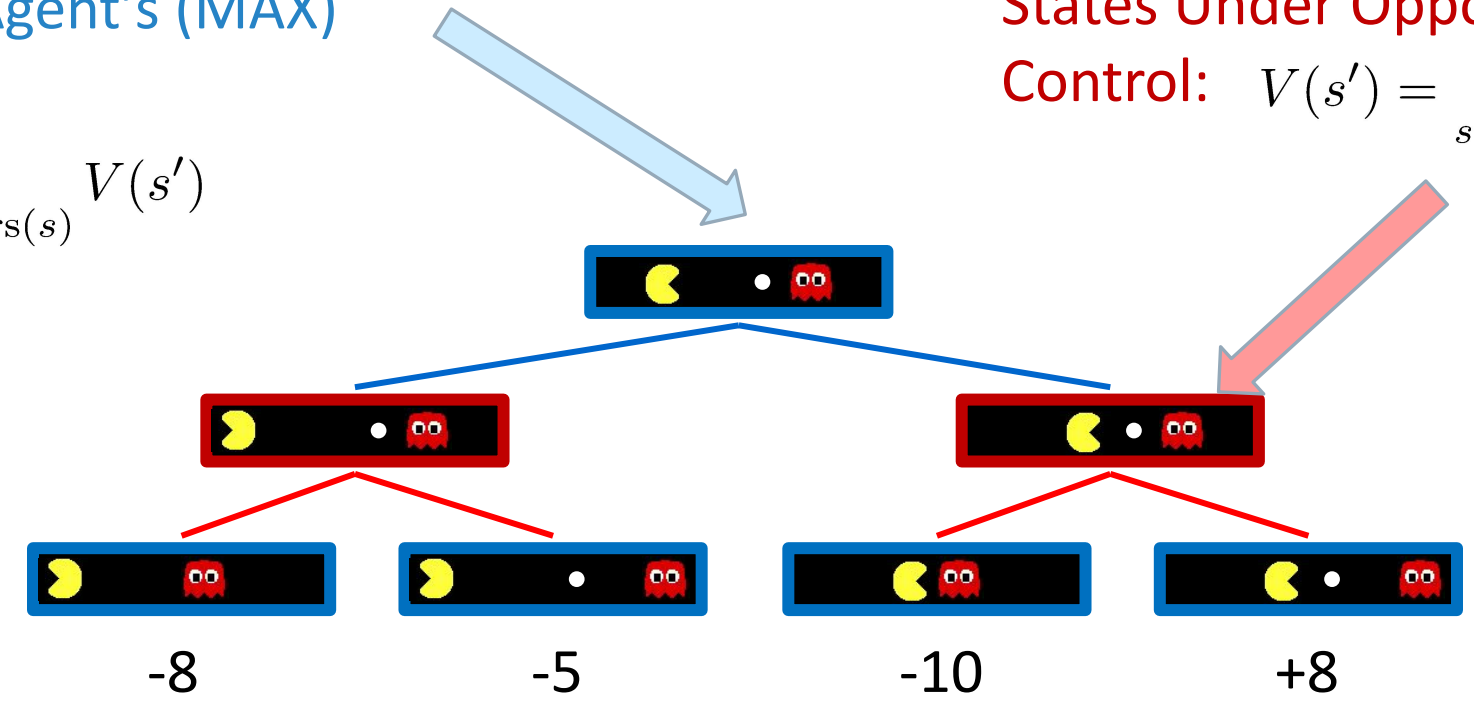
Minimax Values- Example Pacman Game

States Under Agent's (MAX) Control:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent's (MIN) Control:

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

$$V(s) = \text{known}$$

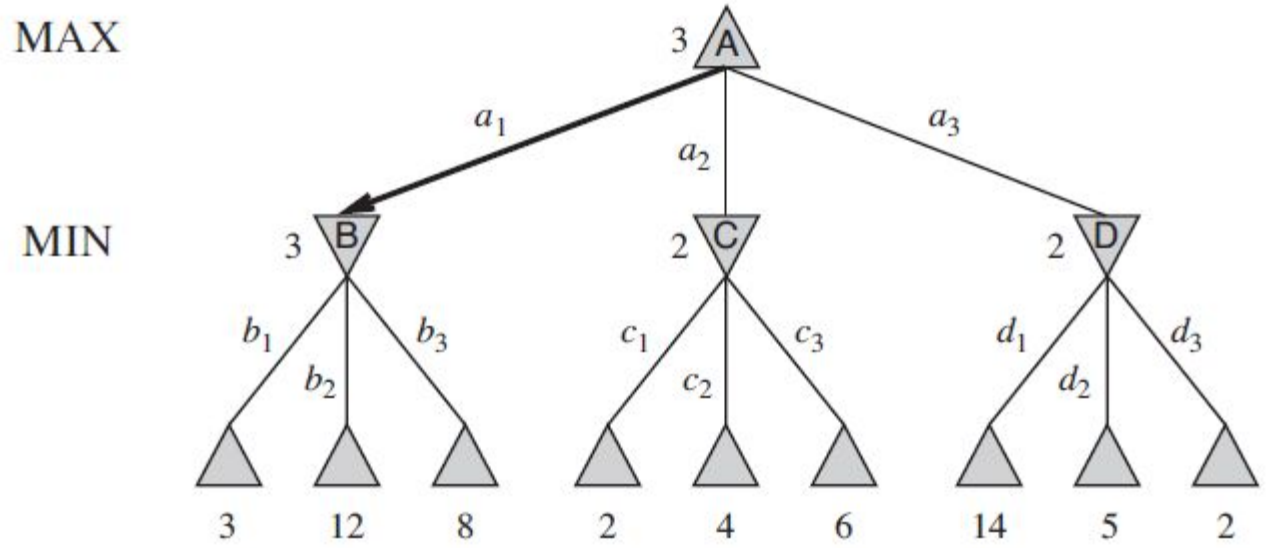
The Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ 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
```

Example



Minimax for Multi-player Games

- Terminals have utility tuples.
- Node values are utility tuples.
- Each player maximizes its own component.
- Can give rise to cooperation and competition dynamically.

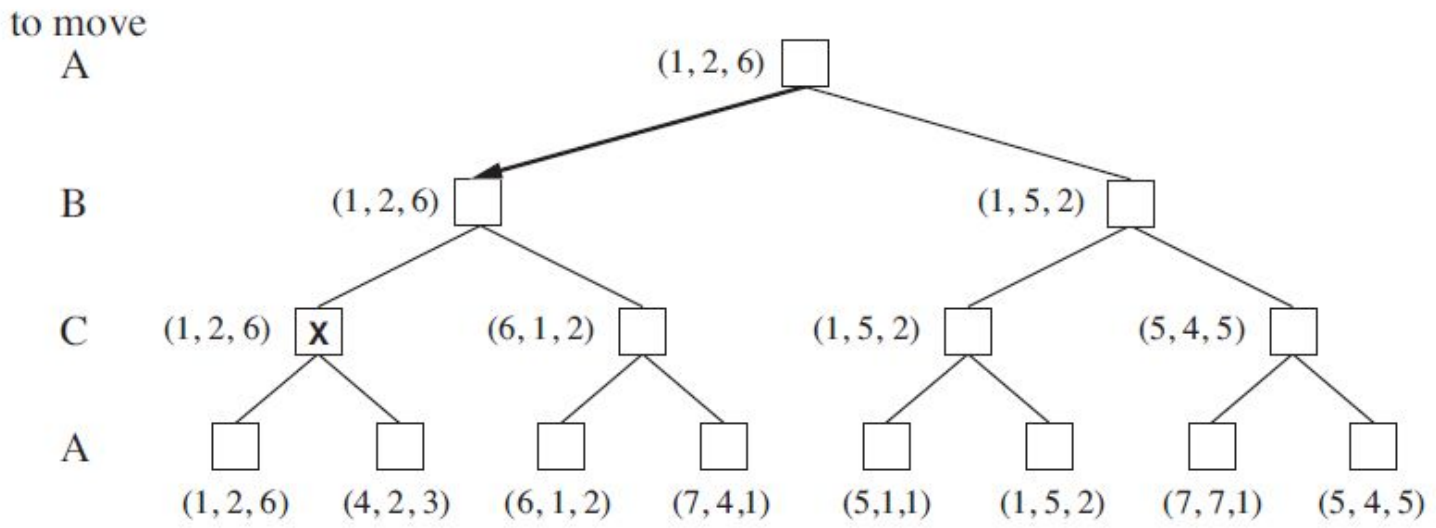


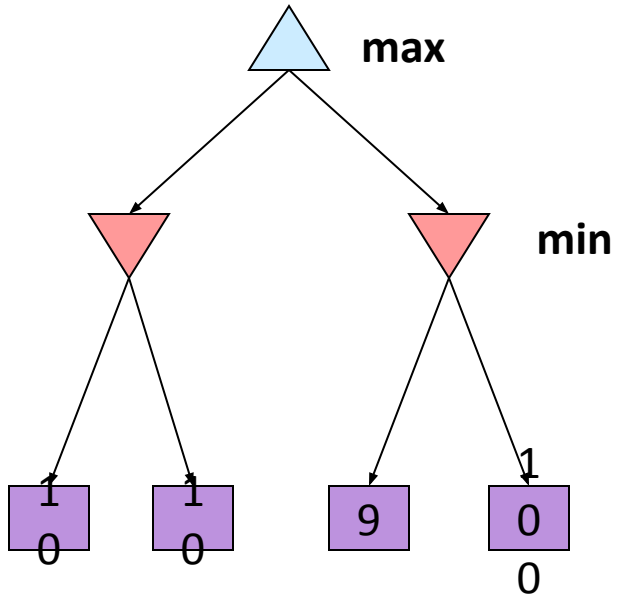
Figure 5.4 The first three plies of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Minimax Properties

- Complete: Yes, if the tree is finite
- Optimal: Yes, but against an optimal opponent.
- Time Complexity: $O(b^m)$ For chess, b=35, m=100, infeasible time!
- Space Complexity: O(bm) (depth-first exploration)

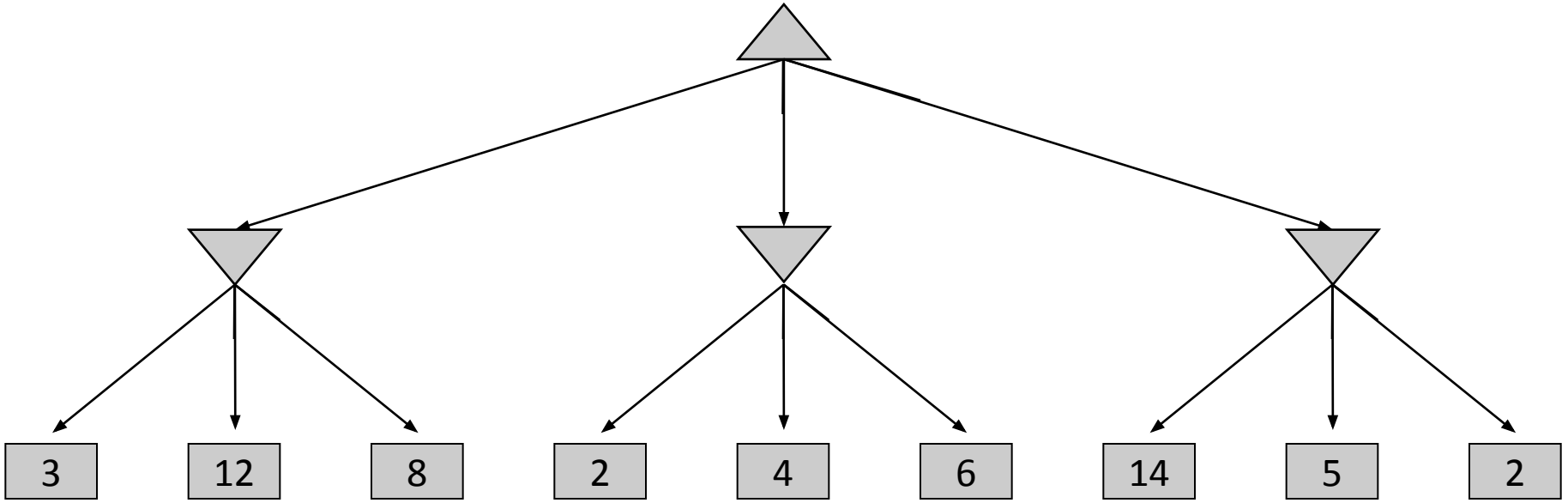
b is the branching factor (legal moves at each point), m is the maximum depth of the tree.

Minimax Properties (cont.)

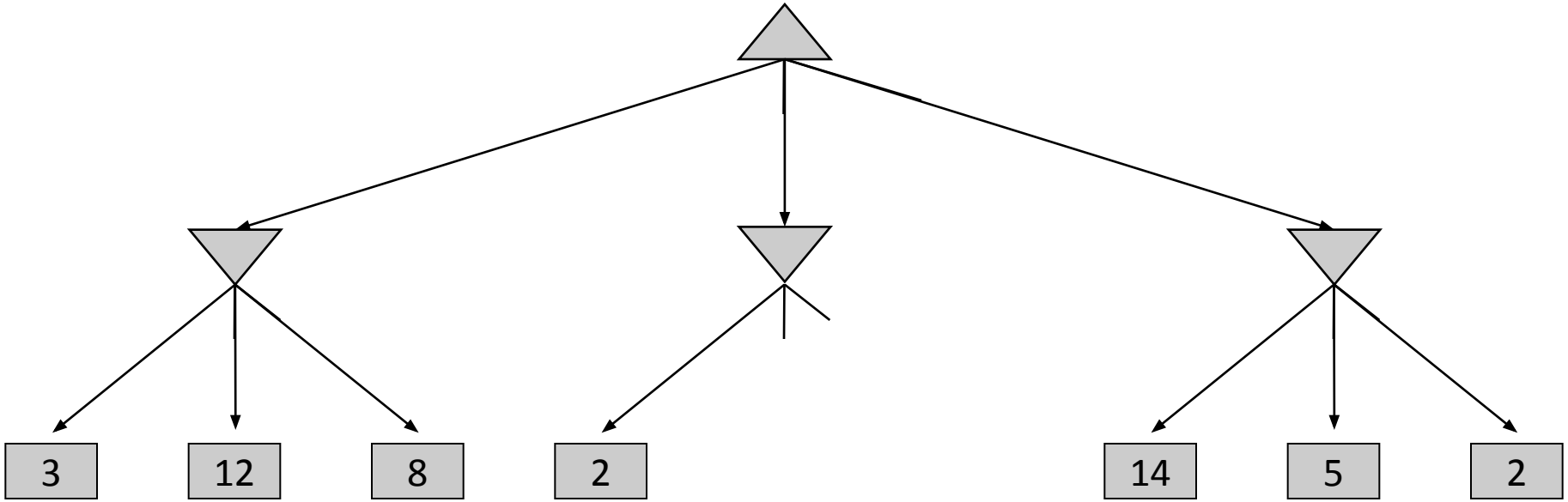


What if min plays non-optimally?

Minimax Example



Pruning?



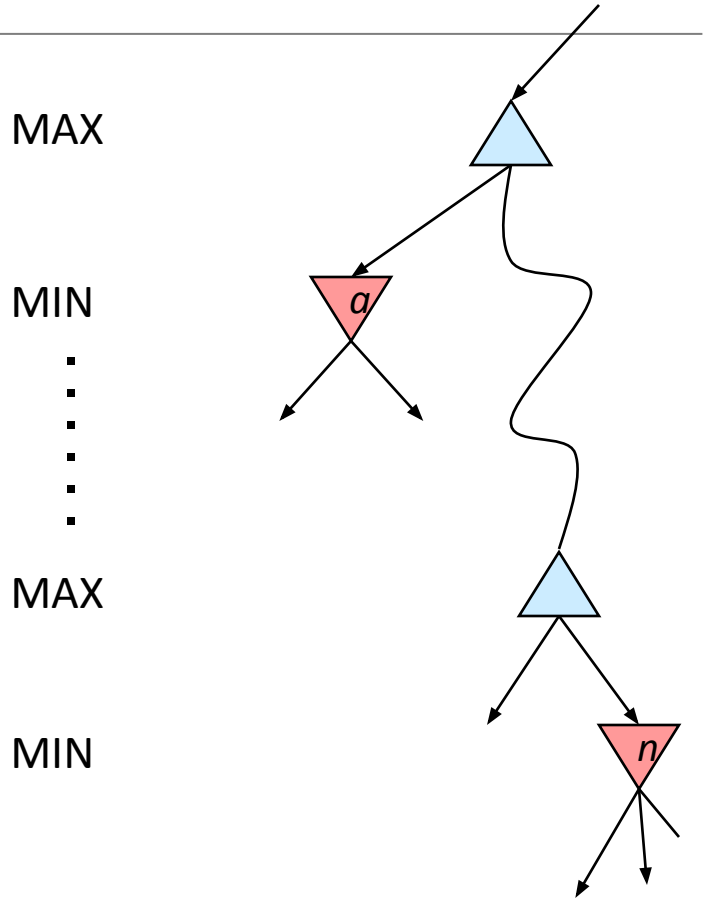
Alpha-Beta Pruning

- Alpha-beta pruning computes the correct minimax decision but prunes away branches that cannot possibly influence the final decision.

Alpha-Beta Pruning

General configuration

- We're computing the MIN-VALUE at some node *n*
- We're looping over *n*'s children
- *n*'s estimate of the childrens' min is dropping
- Who cares about *n*'s value? MAX
- Let *α* be the best value that MAX can get at any choice point along the current path from the root
- If *n* becomes worse than *α*, MAX will avoid it, so we can stop considering *n*'s other children (it's already bad enough that it won't be played)



Alpha-Beta Pruning Algorithm

```
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

- 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.
- 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.

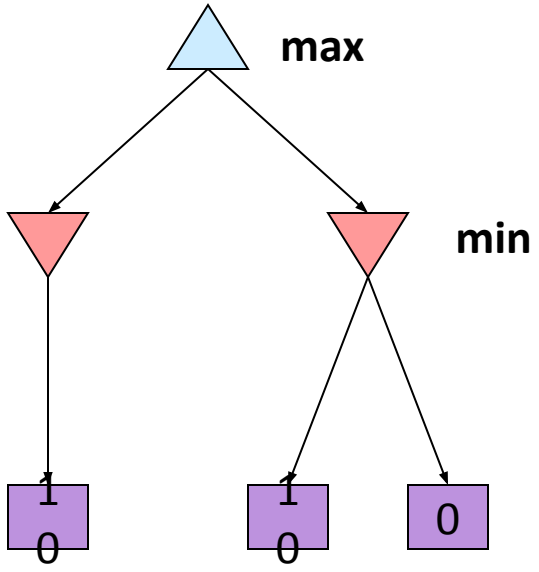
Alpha-Beta Pruning Properties

- The alpha-beta pruning has no effect on minimax value computed for the root!

- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value

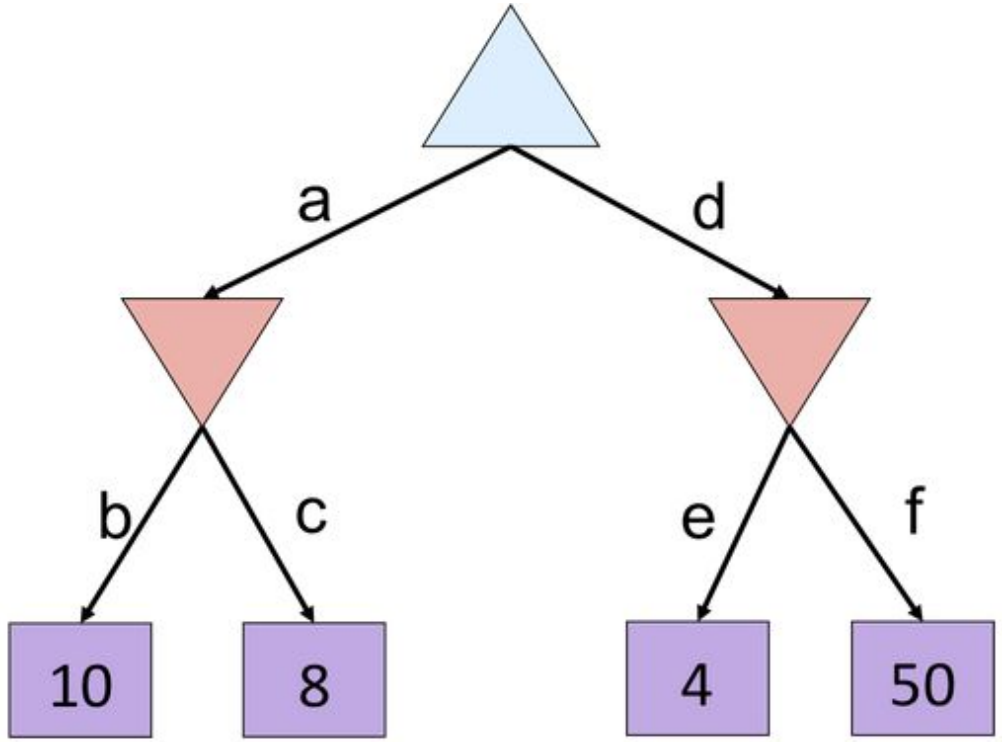
- Good child ordering improves effectiveness of pruning

- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…

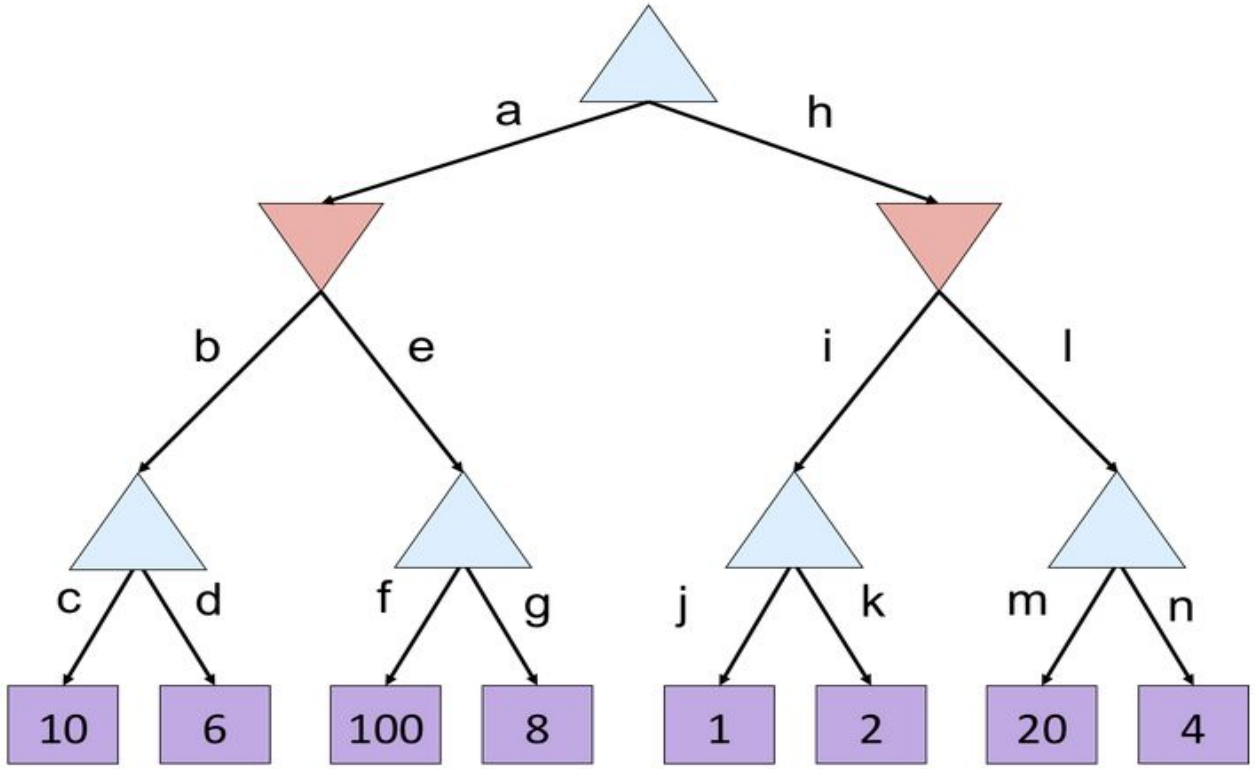


This is a simple example of metareasoning (computing about what to compute)

Alpha-Beta Example 1



Alpha-Beta Example 2



Evaluation Functions

- Apply evaluation function to **approximately estimate** the utility at **nonterminal** nodes turning them into terminal leaves, and cut-off the search depth.
- Alter minimax or alpha–beta in two ways:
  1. Replace the utility function by a heuristic evaluation function EVAL, which estimates the position's utility.
  2. Replace the terminal test by a **cutoff test** that decides when to apply EVAL.
- The heuristic minimax for state s and maximum depth d:

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

Evaluation Functions Design

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

- The evaluation function is typically a weighted linear sum of features:

e.g. For chess, $f_1(s)$ = (num white queens – num black queens), etc.

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

The weights of the evaluation function can be estimated by the machine learning techniques of Chapter 18.

Stochastic Games

- Many games has uncertainty through a random element, such as rolling dice.
- A game tree in stochastic games like backgammon must include **chance nodes** in addition to MAX and MIN nodes.

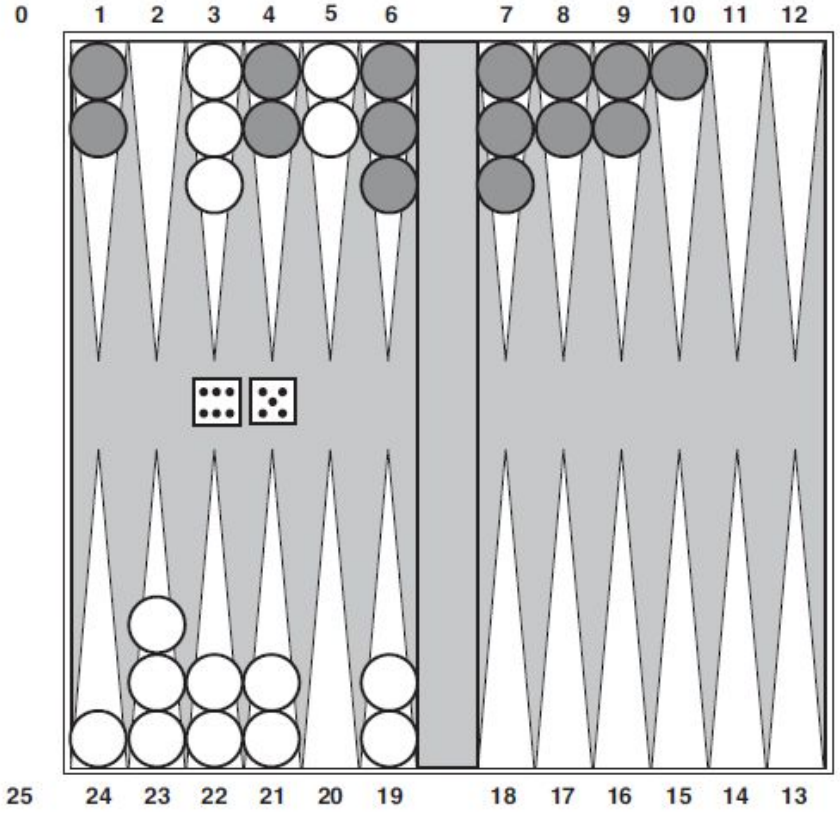
- Sources of uncertainty:
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts in a pacman game respond randomly
  - Actions can fail: when moving a robot, wheels might slip

Example: Backgammon Game

- Although White knows what his or her own legal moves are, White does not know what Black is going to roll and thus does not know what Black's legal moves will be.

- That means White cannot construct a standard game tree of the sort we saw in chess and tic-tac-toe.

- A game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes.

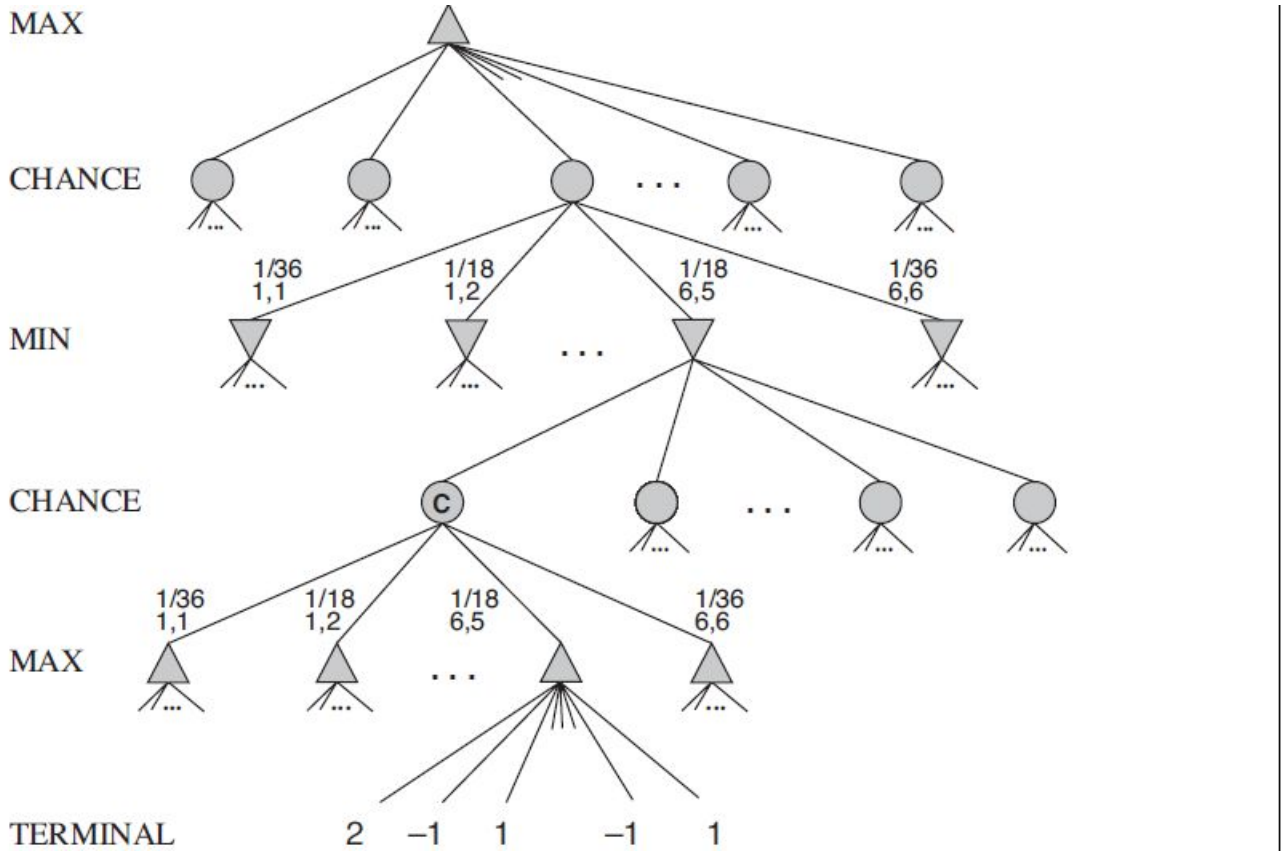


Example: Backgammon Game Tree

- The branches leading from each chance node denote the possible dice rolls; each branch is labeled with the roll and its probability.

- There are 36 ways to roll two dice, each equally likely; but because a 6–5 is the same as a 5–6,
- there are only 21 distinct rolls.

- The six doubles (1–1 through 6–6) each have a probability of
- 1/36, so we say P(1–1) = 1/36. The other 15 distinct rolls each have a 1/18 probability



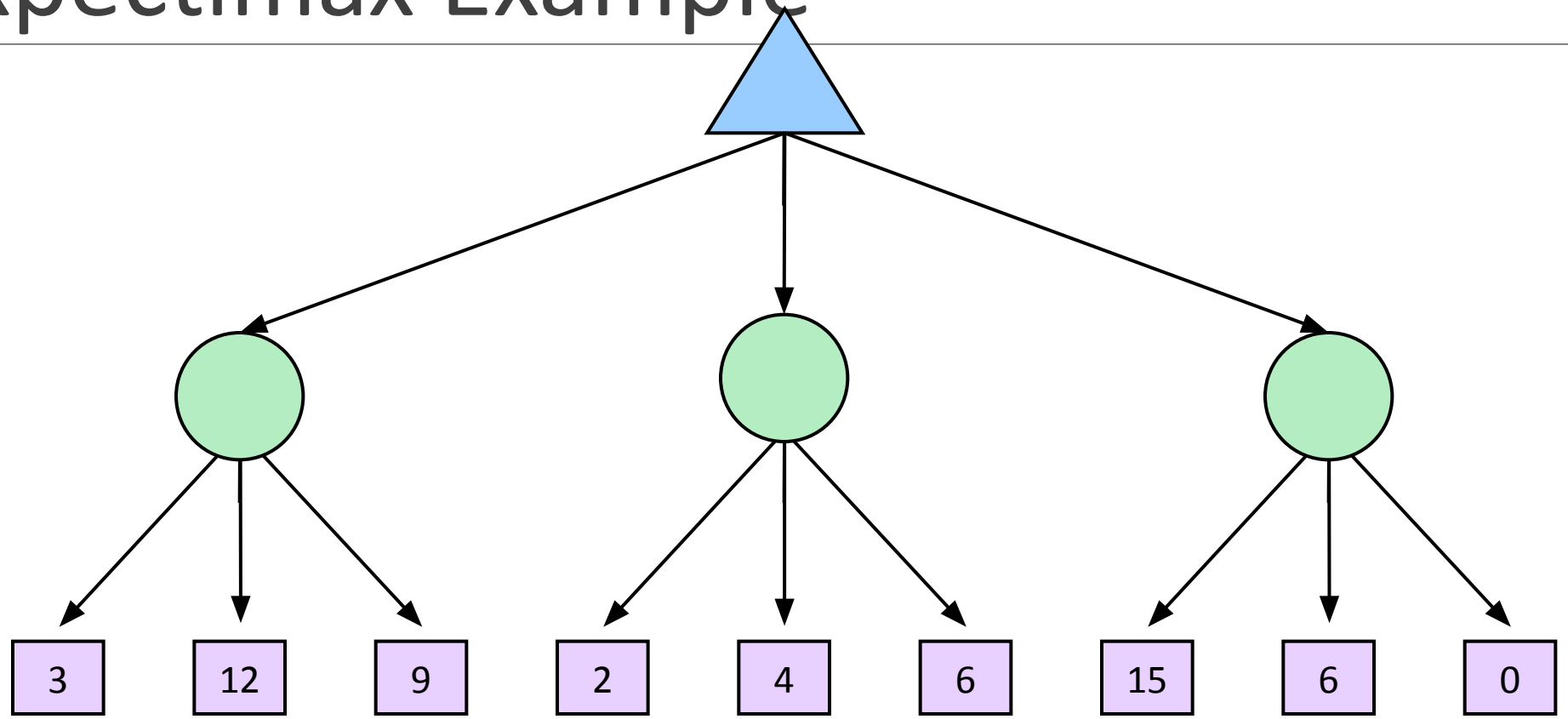
Expectiminimax Algorithm

- For chance nodes, expectiminimax computes the expected value, which is the sum of the value over all outcomes, weighted by the probability of each chance action.
- The terminal nodes and MAX and MIN nodes work exactly the same way as the minimax algorithm.

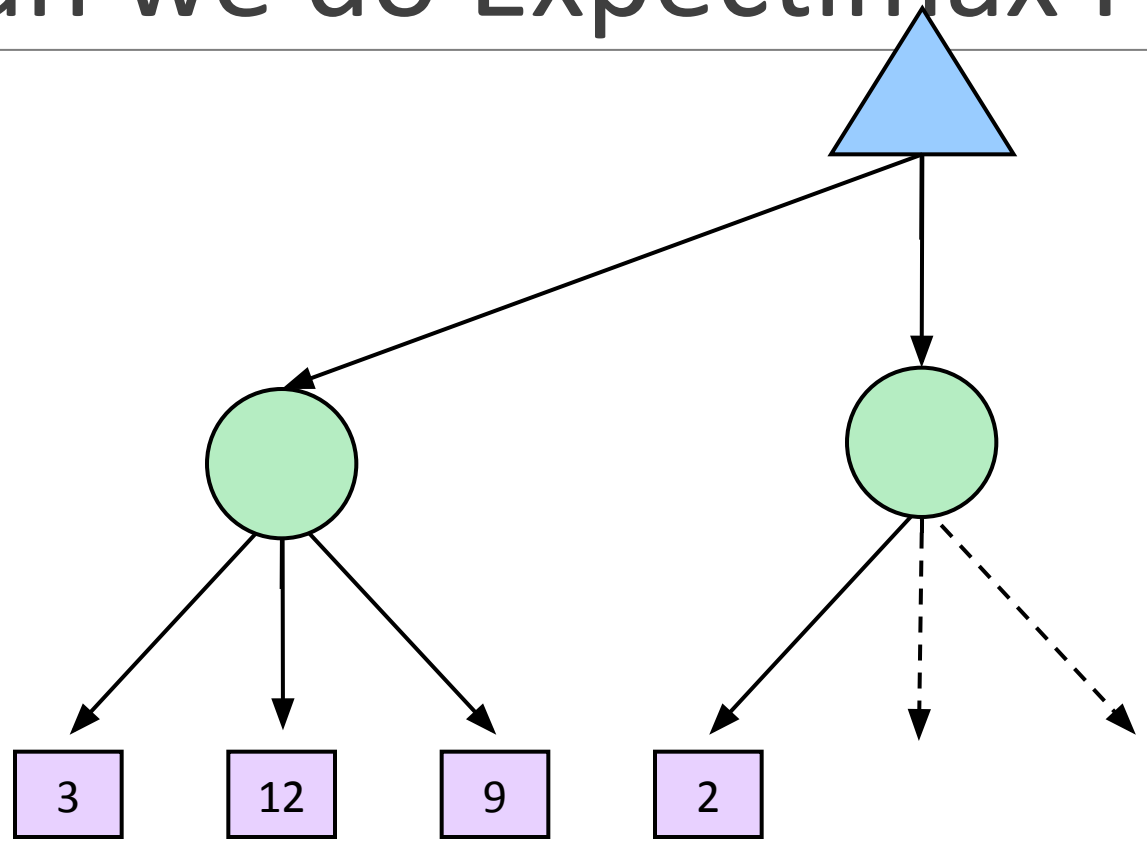
$$\textsc{Expectiminimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \\ \sum_r P(r)\textsc{Expectiminimax}(\textsc{Result}(s,r)) & \text{if } \textsc{Player}(s) = \textsc{chance} \end{cases}$$

where r represents a possible dice roll (or other chance event) and $\textsc{Result}(s,r)$ is the same state as s, with the additional fact that the result of the dice roll is r.

Expectimax Example



Can we do Expectimax Pruning? Why?

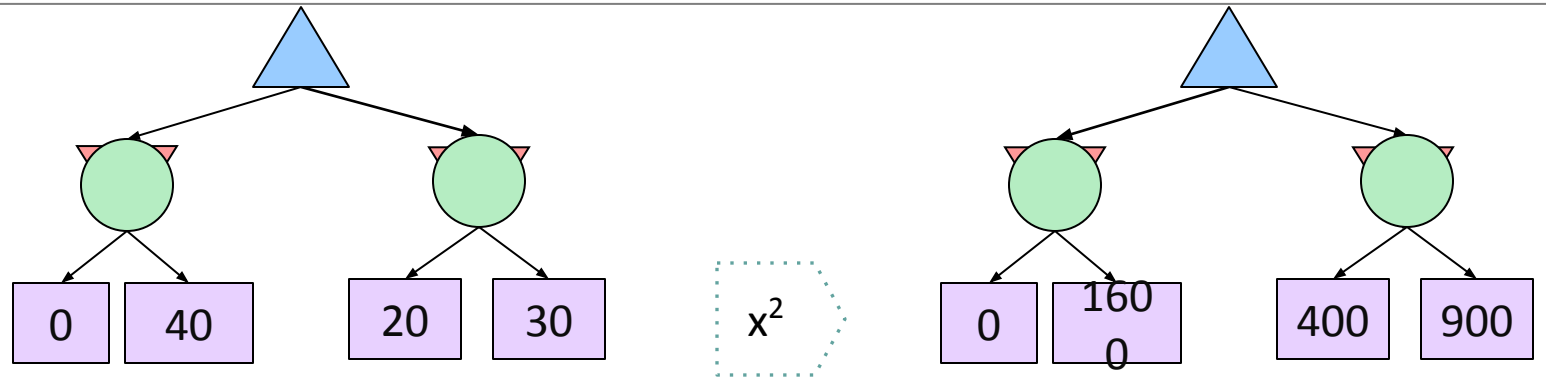


- It might seem impossible because the value of chance node is the *average* of its children's values, and in order to compute the average of a set of numbers, we must look at all the numbers.

- But if we put bounds on the possible values of the utility function, then we can arrive at bounds for the average without looking at every number.

- For example, say that all utility values are between −2 and +2; then the value of leaf nodes is bounded, and in turn we *can* place an upper bound on the value of a chance node without looking at all its children.

Utility-Scale

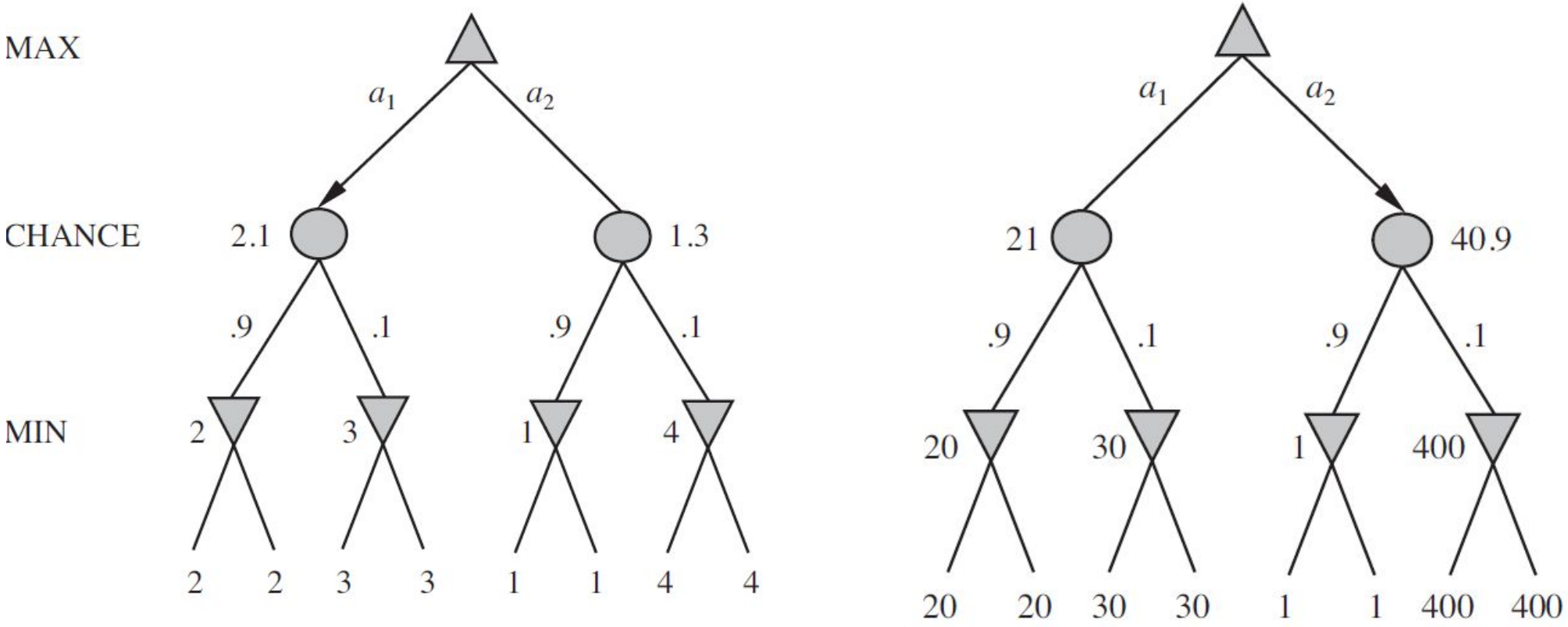


For worst-case minimax reasoning, terminal function scale doesn't matter.

- We just want better states to have higher evaluations (get the ordering right).
- We call this insensitivity to monotonic transformations.

For average-case expectimax reasoning, we need *magnitudes* to be meaningful.

Expectimax Example



- An order-preserving transformation on leaf values changes the best move!

Partially Observable Games

- For example, card games where the opponent's cards are unknown.
- Consider all possible deals of the invisible cards; solve each one as if it were a fully observable game; and then choose the move that has the best outcome averaged over all the deals.
- Suppose that each deal s occurs with probability P(s); then the move to play would be:

$$\operatorname*{argmax}_a \sum_s P(s)\,\textsc{Minimax}(\textsc{Result}(s,a))\ .$$

Partially Observable Games- Card games

- In most card games, the number of possible deals is rather large.

- For example, in bridge 10, 400, 600 possible deals!!

- Apply Monte Carlo approximation by taking a *random sample* of N deals,

where the probability of deal s appearing in the sample is proportional to P(s):

$$\underset{a}{\operatorname{argmax}} \frac{1}{N} \sum_{i=1}^{N} \textsc{Minimax}(\textsc{Result}(s_i, a)) \ .$$

- As N increases, the sum over the random sample tends to the exact value.

- P(s) does not appear explicitly in the summation, because the samples are already drawn according to P(s).

Summary

- Adversarial Search
- Minimax Algorithm
- Alpha beta pruning
- Stochastic games
- Expectimax algorithm
- Evaluation functions and cutting-off the search
- Partially observable games