COL333/671: Introduction to AI

Semester I, 2022-23

Adversarial Search

Rohan Paul

Outline

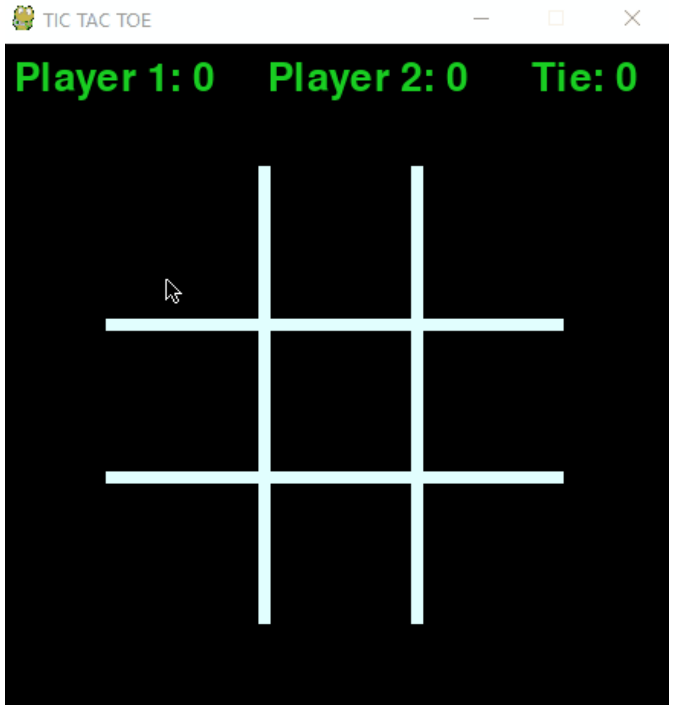
- Last Class
  - Constraint Satisfaction
- This Class
  - Adversarial Search
- Reference Material
  - AIMA Ch. 5 (Sec: 5.1-5.5)

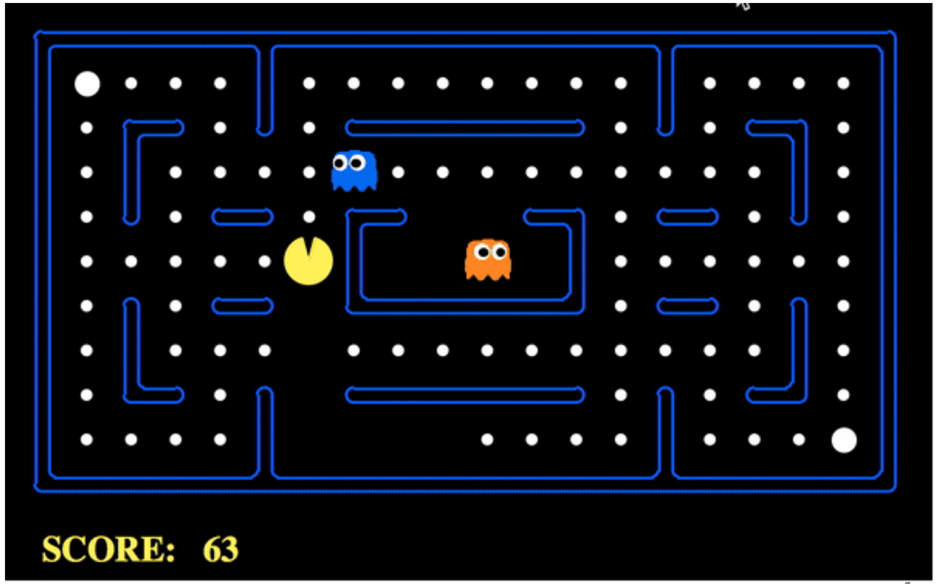
Acknowledgement

These slides are intended for teaching purposes only. Some material has been used/adapted from web sources and from slides by Doina Precup, Dorsa Sadigh, Percy Liang, Mausam, Dan Klein, Anca Dragan, Nicholas Roy and others.

Game Playing and AI

- **Games: challenging decision-making problems**
  - Incorporate the state of the other agent in your decision-making. Leads to a vast number of possibilities.
  - Long duration of play. Win at the end.
  - Time limits: Do not have time to compute optimal solutions.





Games: Characteristics

- Axes:
  - Players: one, two or more.
  - Actions (moves): deterministic or stochastic
  - States: fully known or not.

- Zero-Sum Games
  - Adversarial: agents have opposite utilities (values on outcomes)

- **Core: contingency problem**
  - The opponent's move is not known ahead of time. A player must respond with a move for every possible opponent reply.
- **Output**
  - Calculate a strategy (policy) which recommends a move from each state.

Playing Tic-Tac-Toe: *Essentially a search problem!*

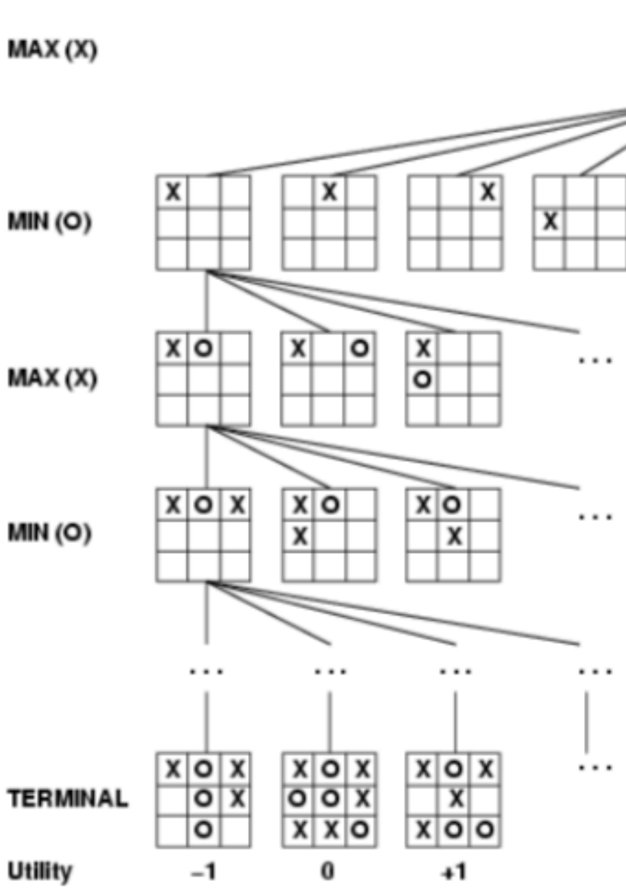
computer's turn

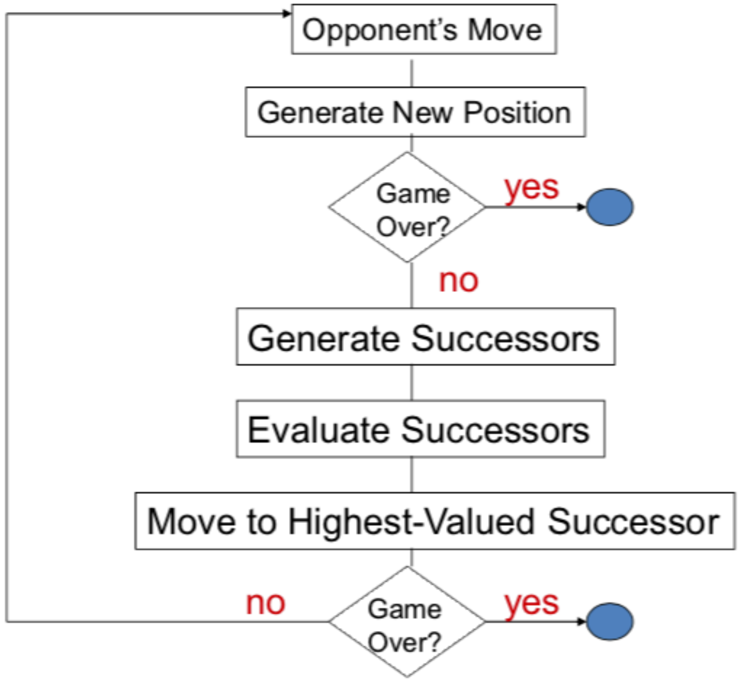
opponent's turn

computer's turn

opponent's turn

leaf nodes are evaluated

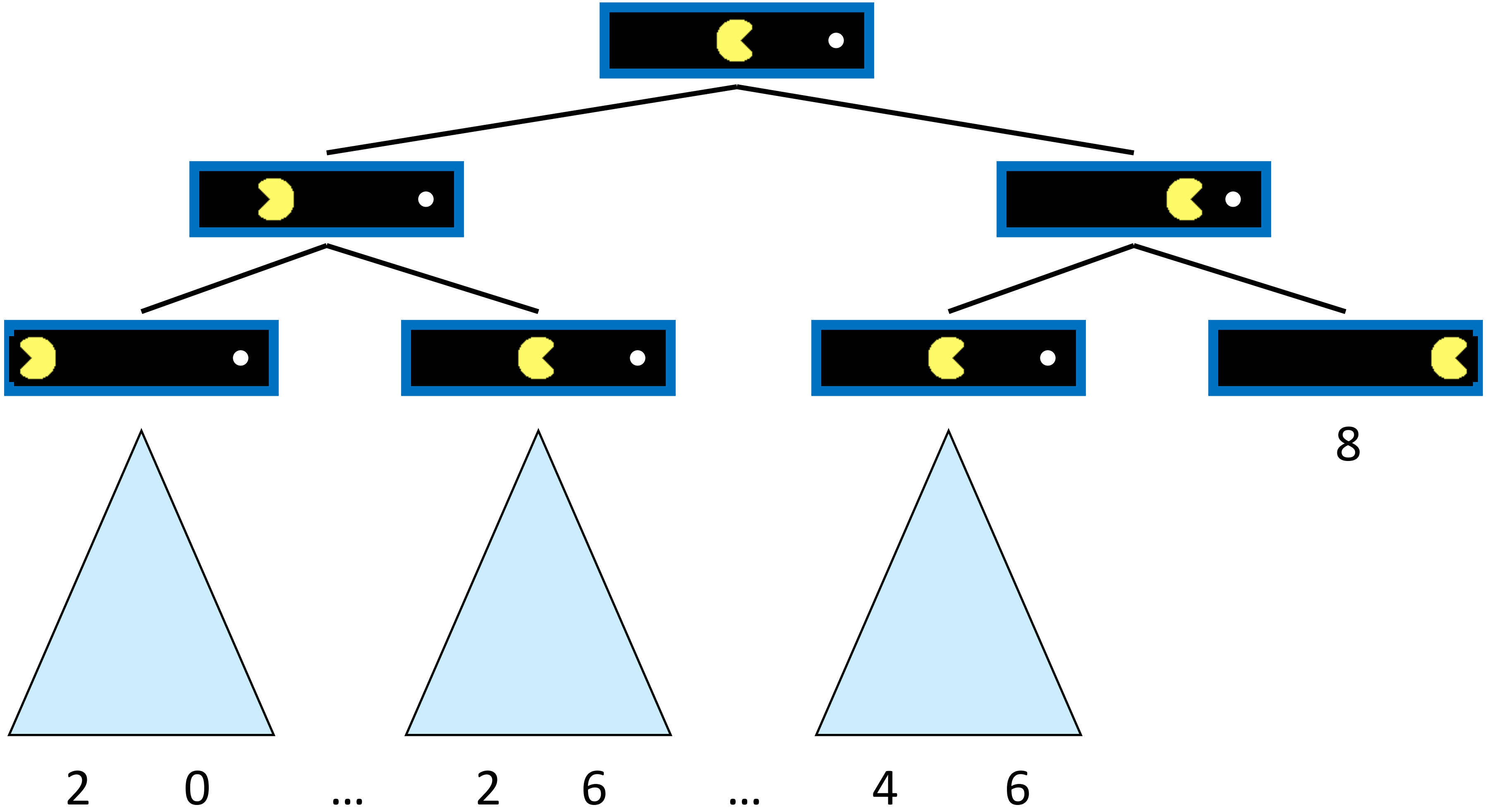




Terminal nodes we get -1, 0 or 1 for loss, tie or win. Think of this value as a "utility" of a state.

in and from Mausam

Single-Agent Trees

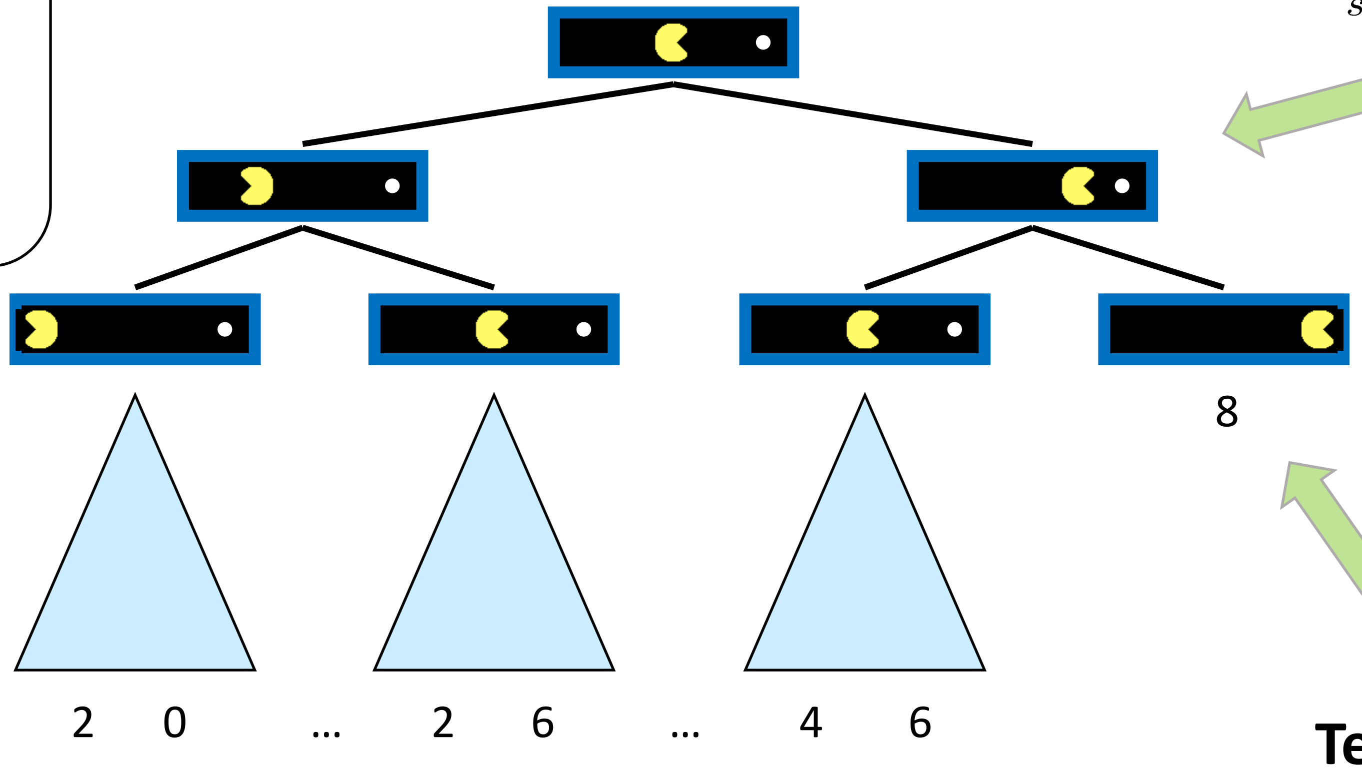


Computing "utility" of states to decide actions

Value of a state:
The best achievable outcome (utility) from that state

Non-Terminal States:

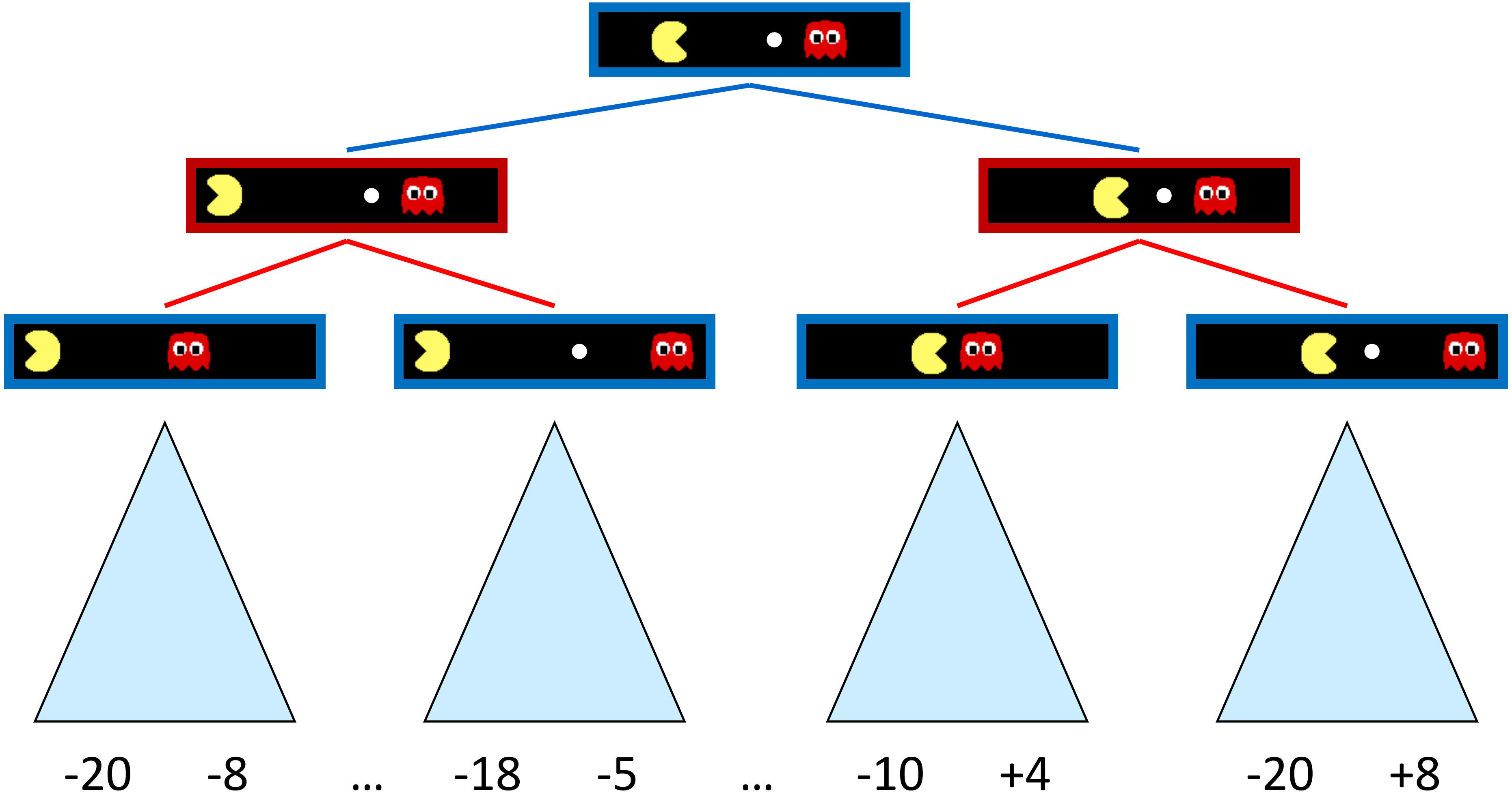
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Terminal States:

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

Game Trees: Presence of an Adversary



The adversary's actions are not in our control. Plan as a contingency considering all possible actions taken by the adversary.

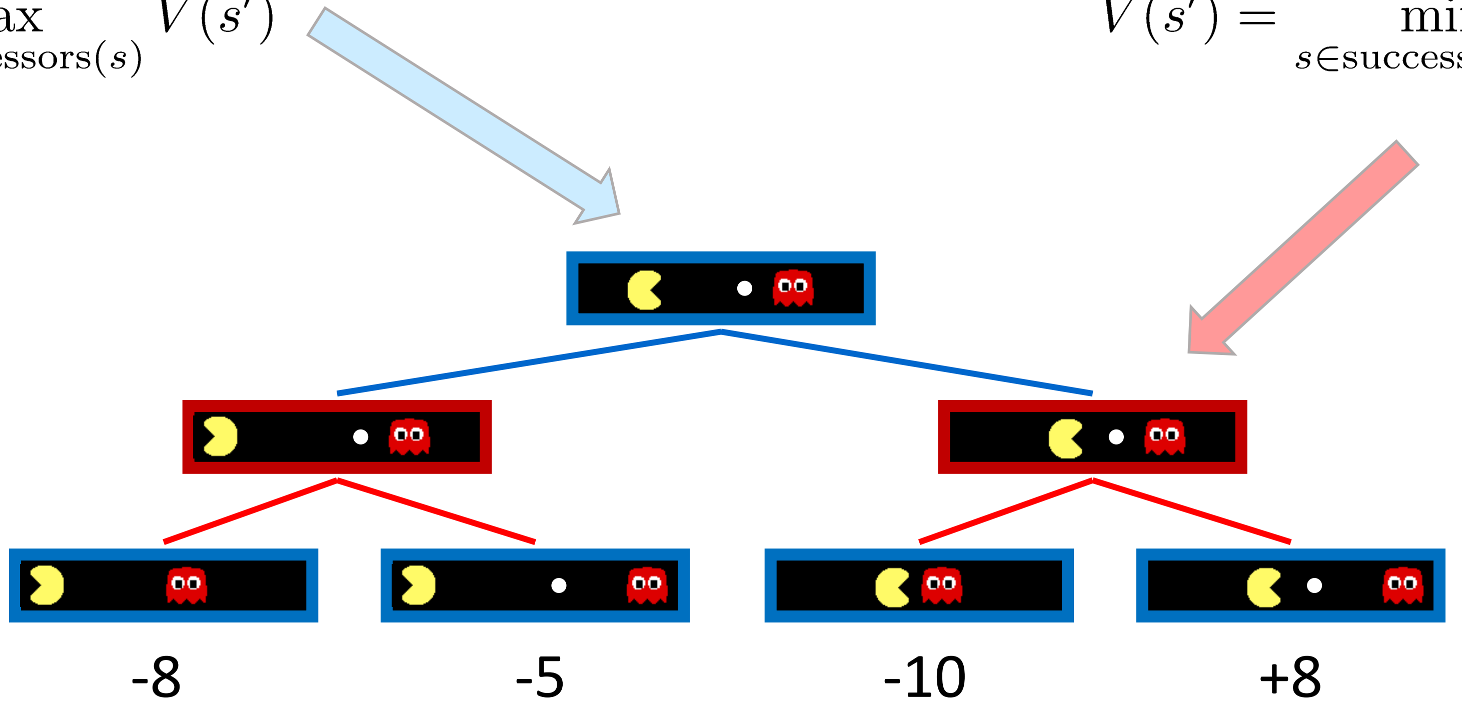
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

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



Terminal States:

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

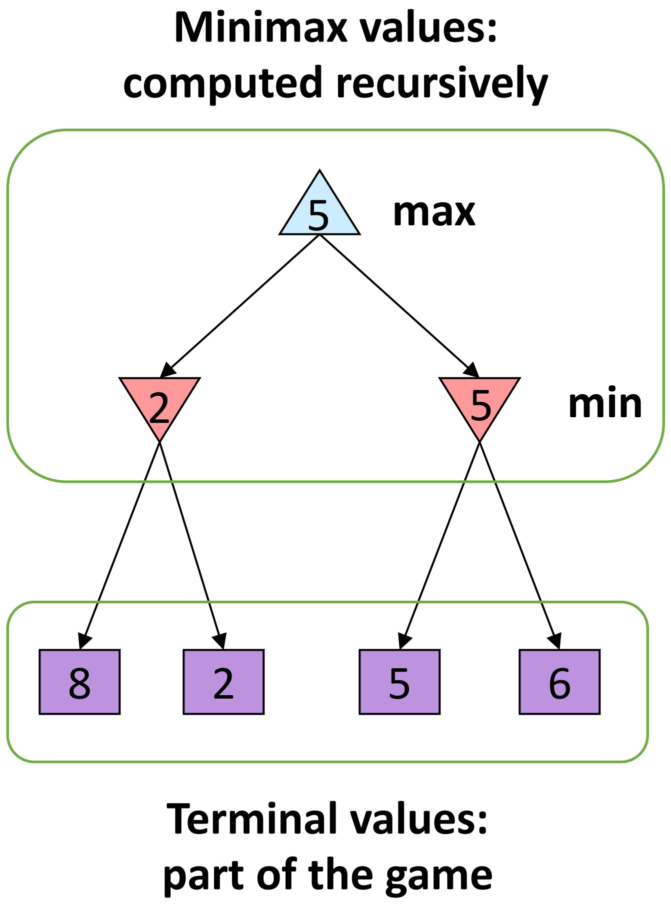
Adversarial Search (Minimax)

- Consider a deterministic, zero-sum game
  - Tic-tac-toe, chess etc.
  - One player maximizes result and the other minimizes result.

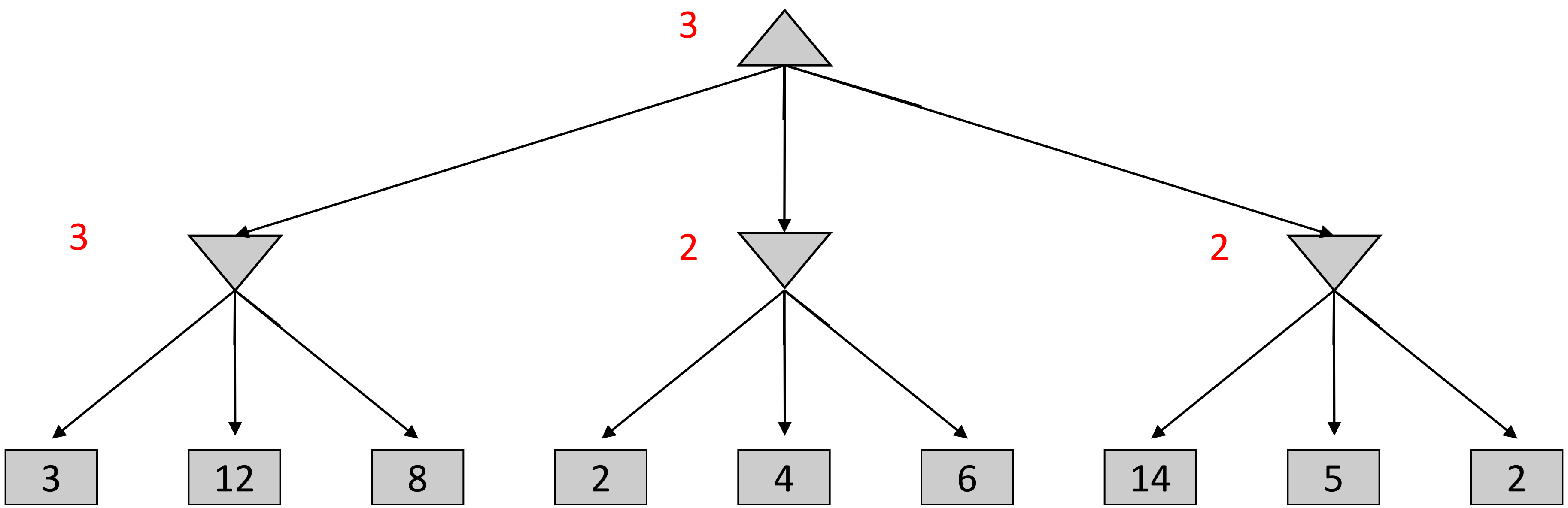
- Minimax Search
  - Search the game tree for best moves.
  - Select optimal actions that move to a position with the highest minimax value.
  - What is the minimax value?
    - It is the best achievable utility against the optimal (rational) adversary.
    - Best achievable payoff against the best play by the adversary.

Minimax Algorithm

- Ply and Move
  - Move: when action taken by both players.
  - Ply: is a half move.
- Backed-up value
  - of a MAX-position: the value of the largest successor
  - of a MIN-position: the value of its smallest successor.
- Minimax algorithm
  - Search down the tree till the terminal nodes.
  - At the bottom level apply the utility function.
  - Back up the values up to the root along the search path (compute as per min and max nodes)
  - The root node selects the action.



Minimax Example



Minimax Implementation

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

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

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

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

Minimax Implementation

```
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 MIN: return min-value(state)
```

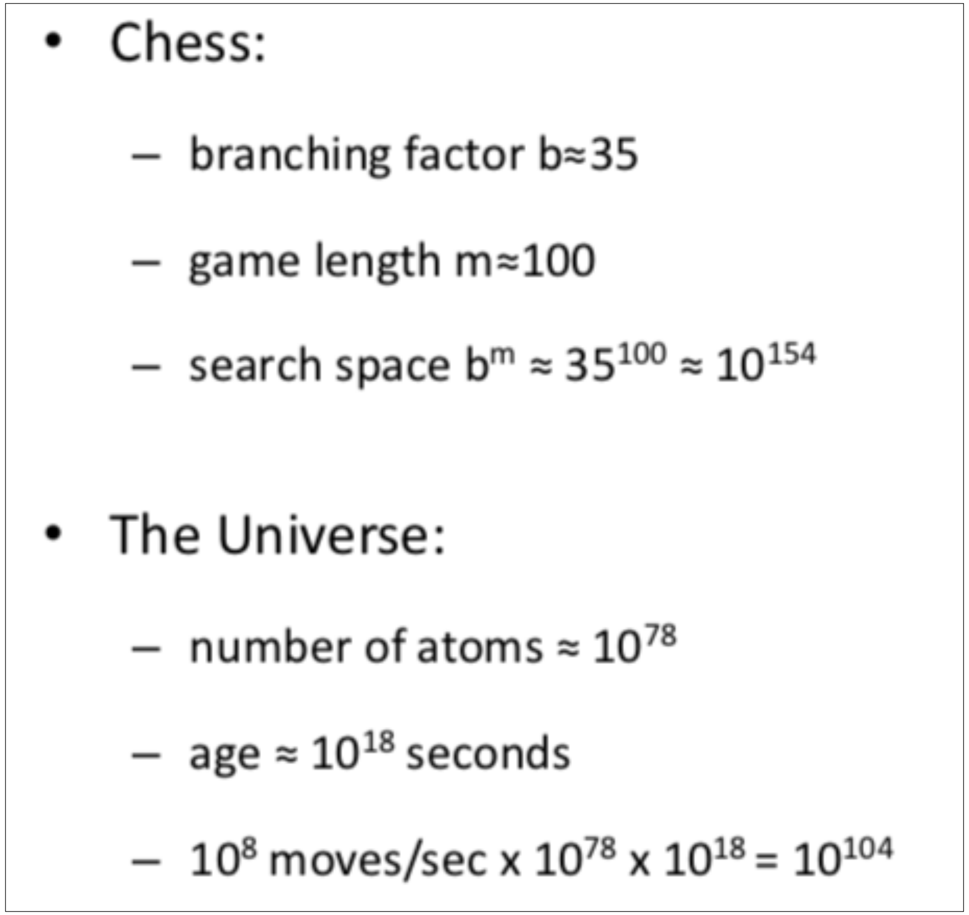
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

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

Useful, when there are multiple adversaries.

Minimax Properties

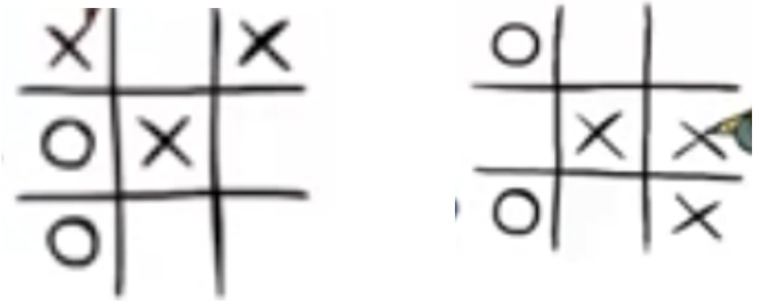
- Completeness
  - Yes
- Complexity
  - Time: $O(b^m)$
  - Space: $O(bm)$
- **Requires growing the tree till the terminal nodes.**
- **Not feasible in practice for a game like Chess.**

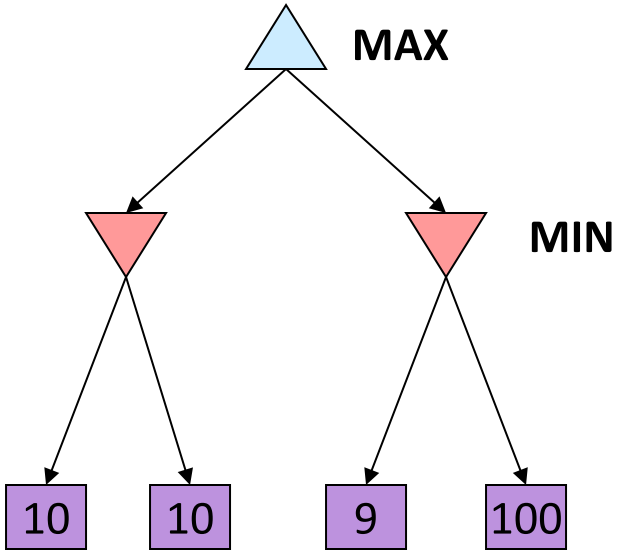


Minimax Properties

- Optimal
  - If the adversary is playing optimally (i.e., giving us the min value)
    - Yes
  - If the adversary is not playing optimally (i.e., not giving us the min value)
    - No. Why? It does not exploit the opponent's weakness against a suboptimal opponent).

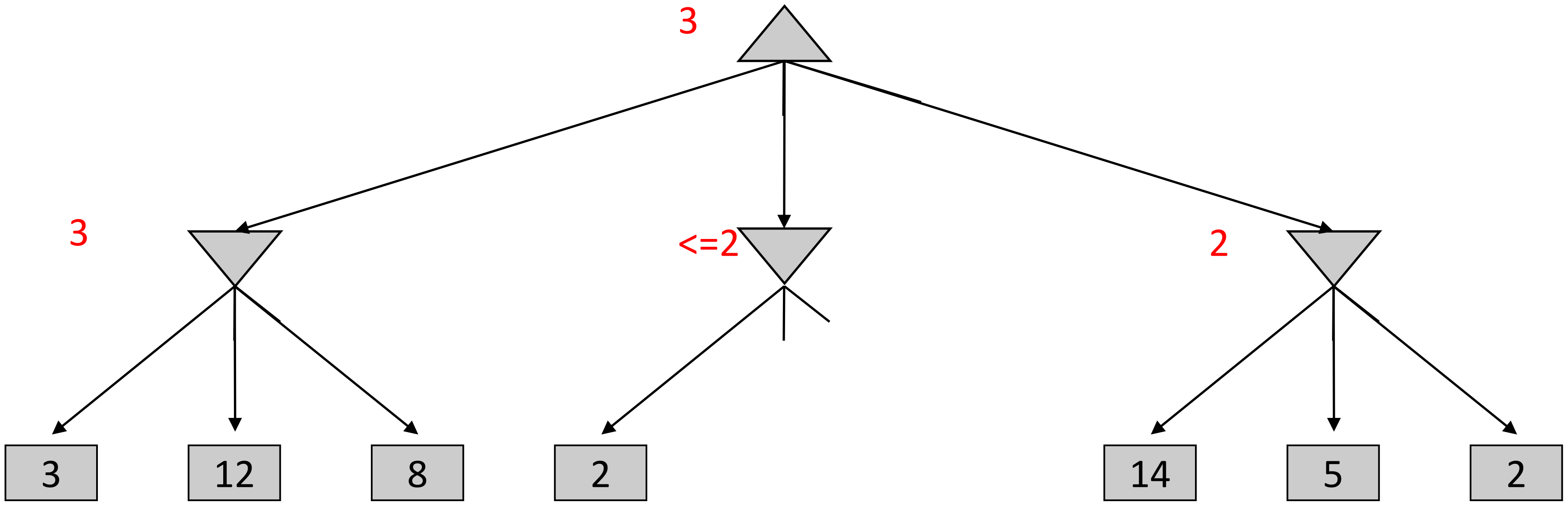
You: Cricle. Opponent: Cross





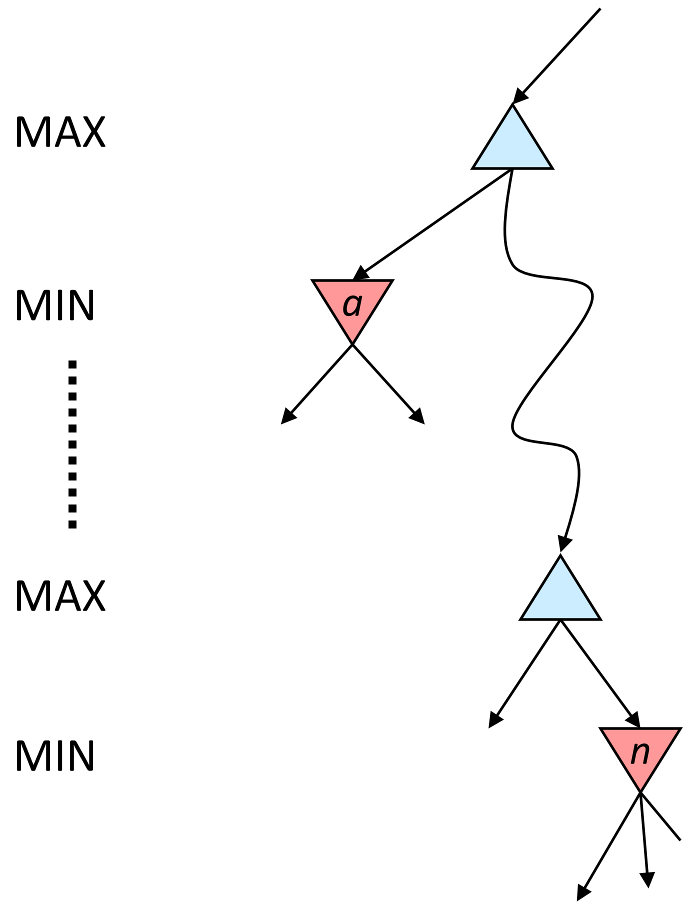
If min returns 9? Or 100?

Necessary to examine all values in the tree?



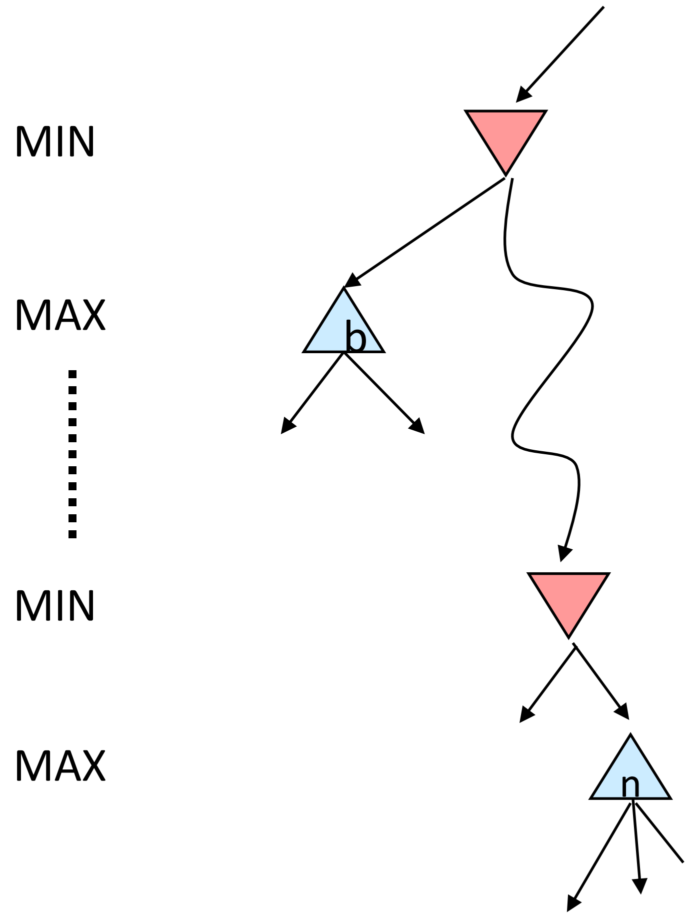
Alpha-Beta Pruning: General Idea

- **General Configuration (MIN version)**
  - Consider computing the MIN-VALUE at some node *n*, examining *n*'s children
  - *n*'s estimate of the childrens' min is reducing.
  - Who can use *n*'s value to make a choice? MAX
  - Let *a* be the best value that MAX can get at any choice point along the current path from the root
  - If the value at *n* becomes worse than *a*, MAX will not pick this option, so we can stop considering *n*'s other children (any further exploration of children will only reduce the value further)

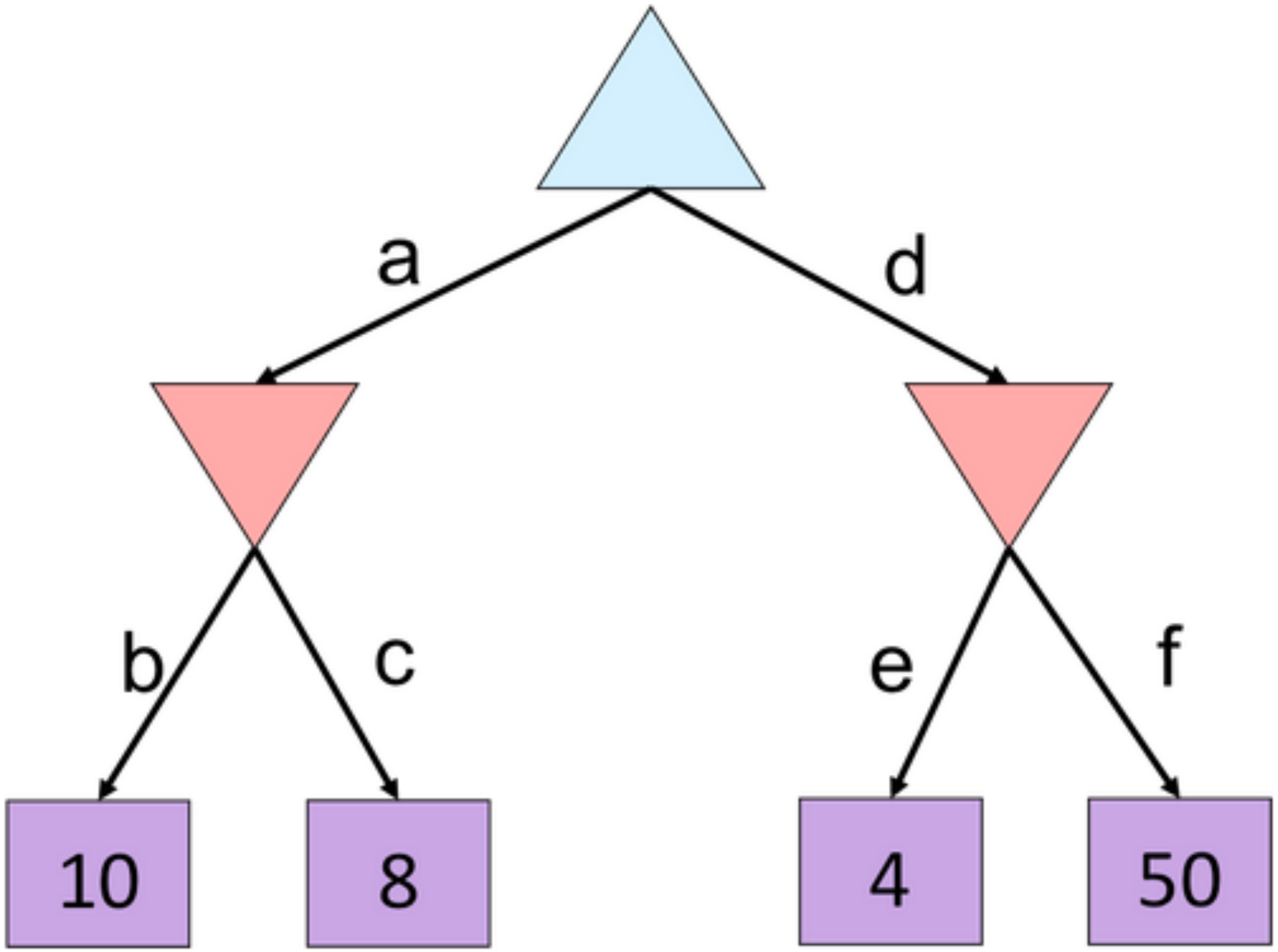


Alpha-Beta Pruning: General Idea

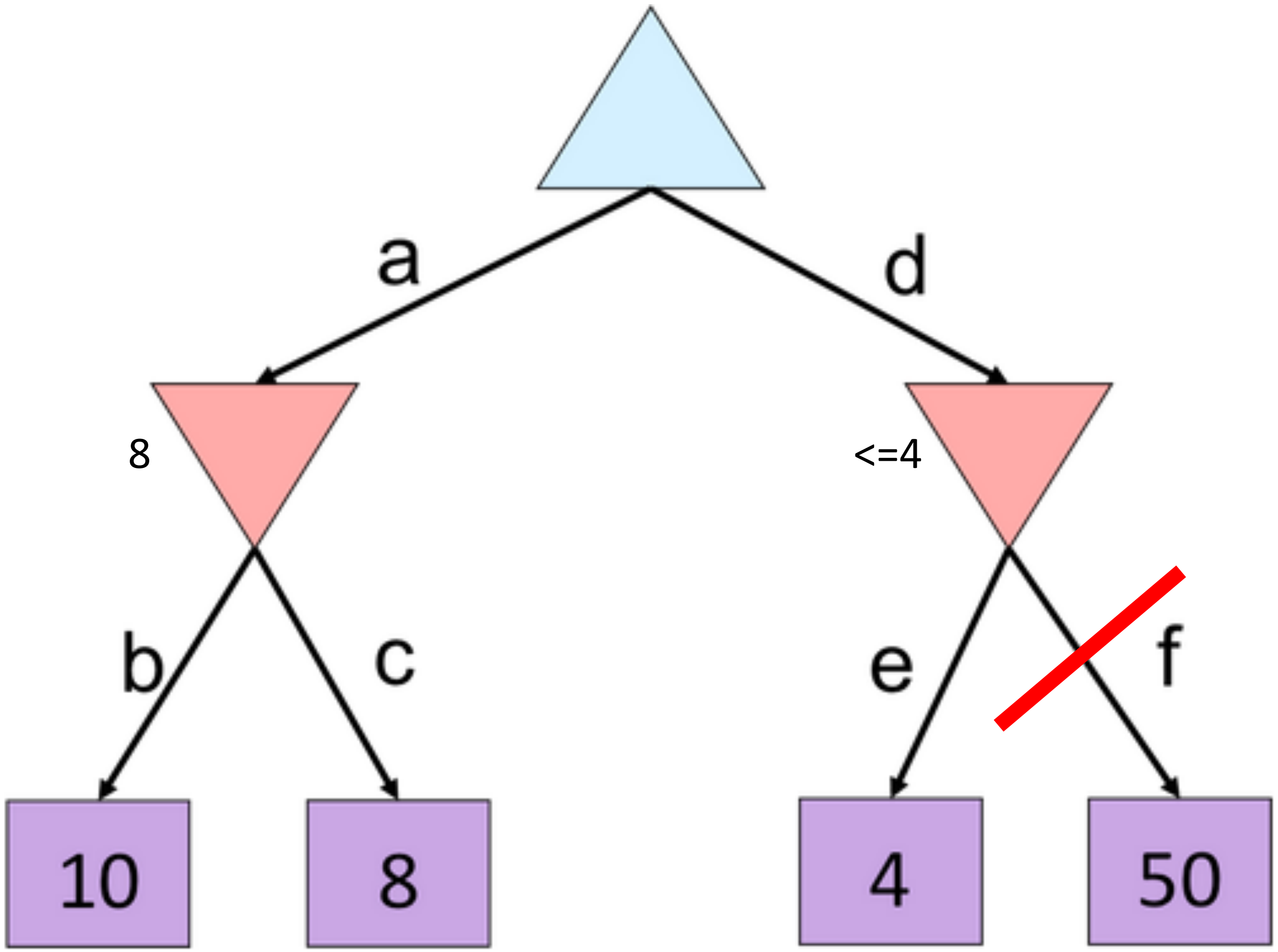
- **General Configuration (MAX version)**
  - Consider computing the MAX-VALUE at some node *n*, examining *n*'s children
  - *n*'s estimate of the childrens' max is increasing.
  - Who can use *n*'s value to make a choice? MIN
  - Let *b* be the lowest (best) value that MIN can get at any choice point along the current path from the root
  - If the value at *n* becomes higher than *b*, MIN will not pick this option, so we can stop considering *n*'s other children (any further exploration of children will only increase the value further)



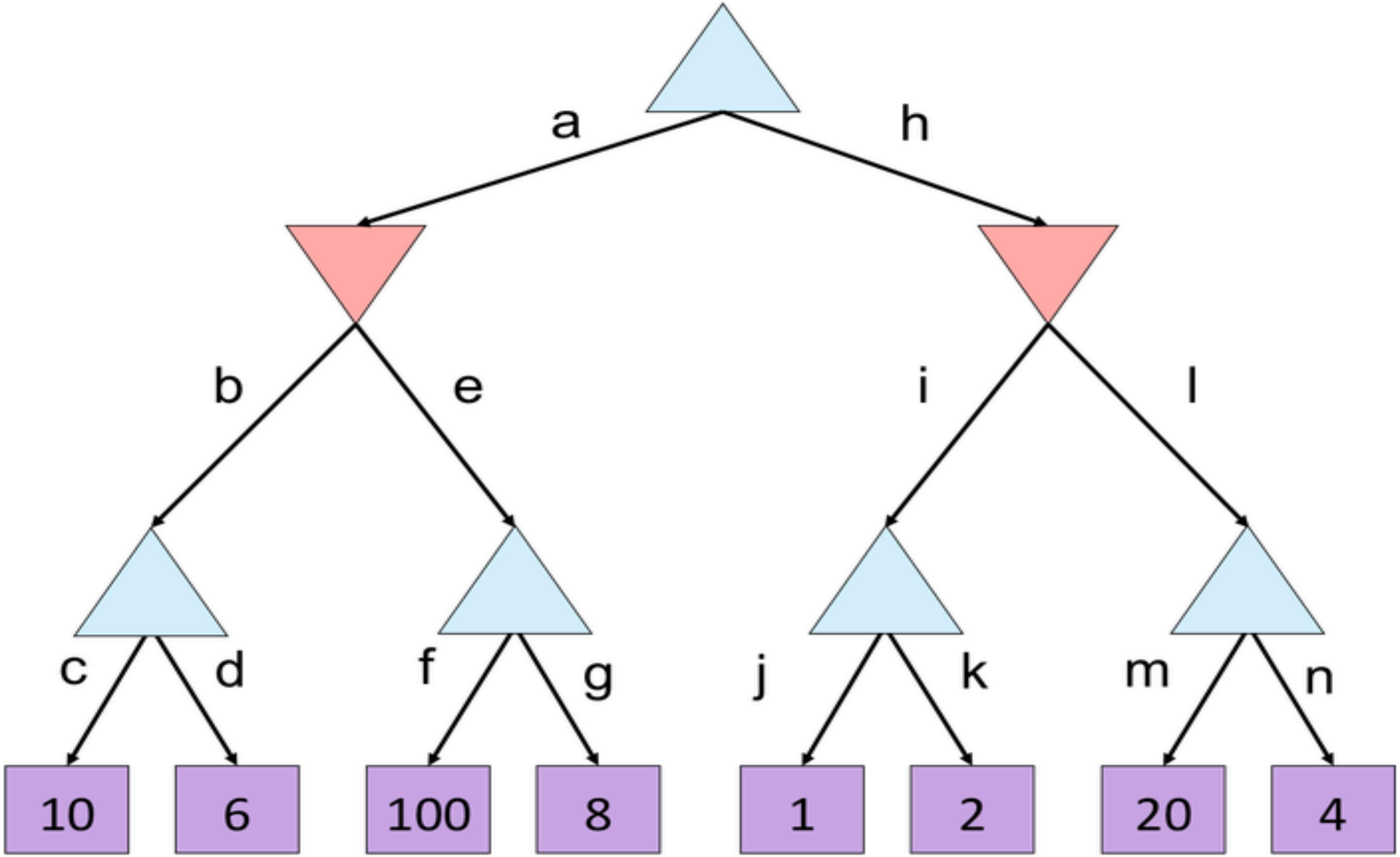
Pruning: Example



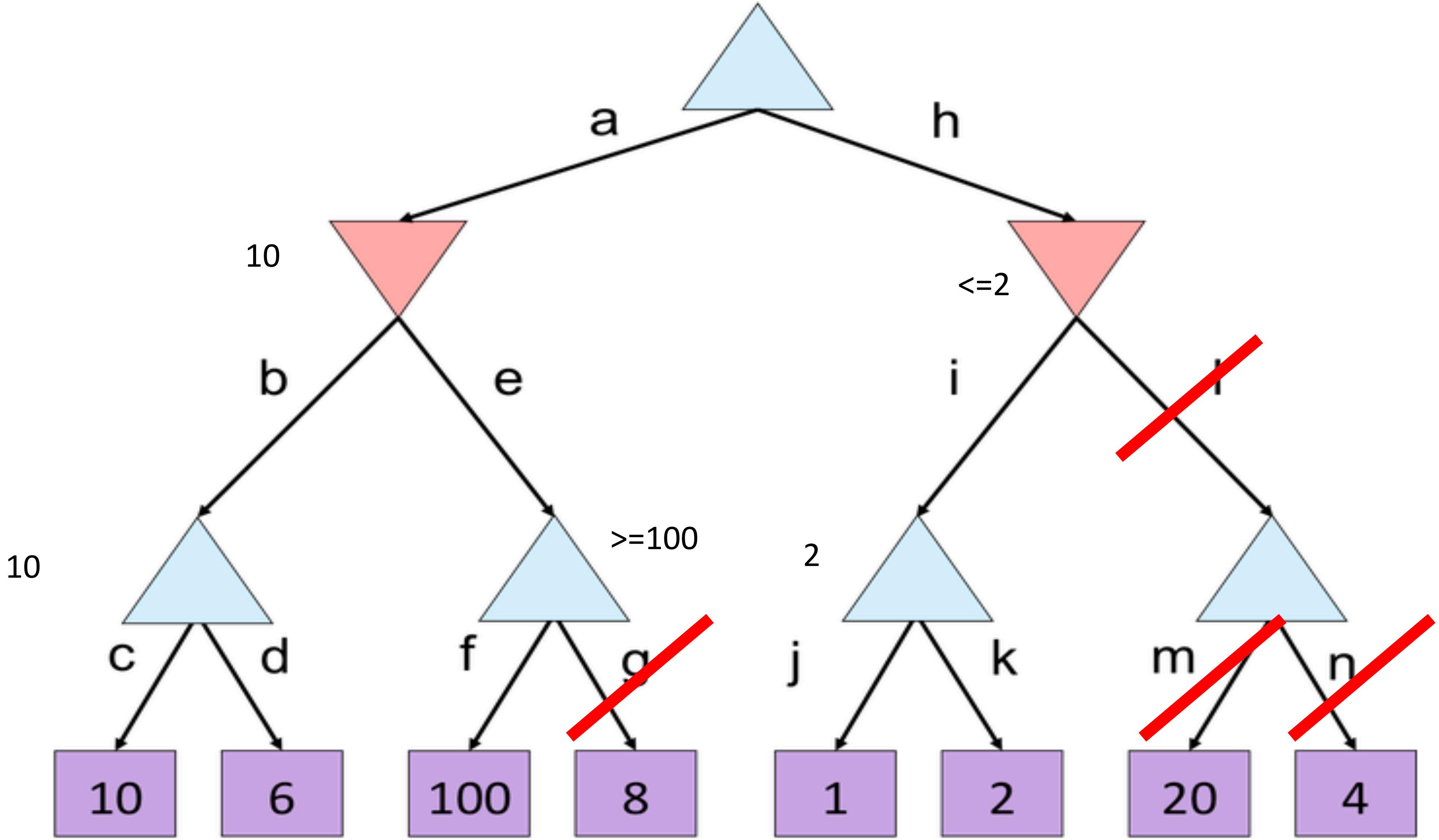
Pruning: Example



Pruning: Example



Pruning: Example



Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

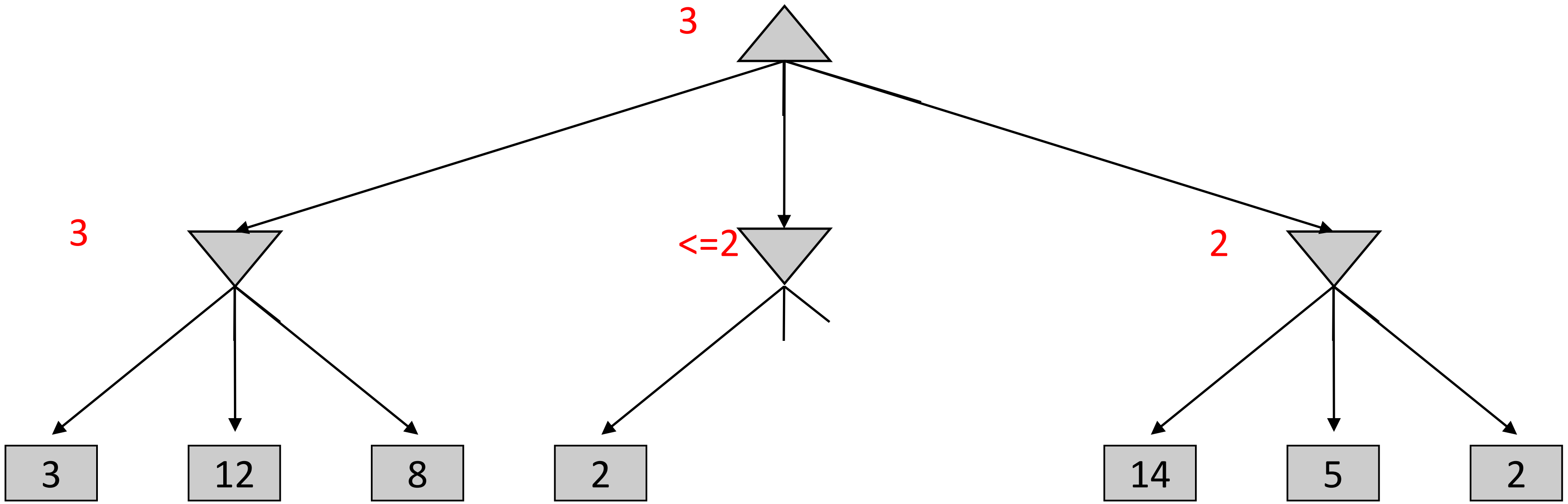
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

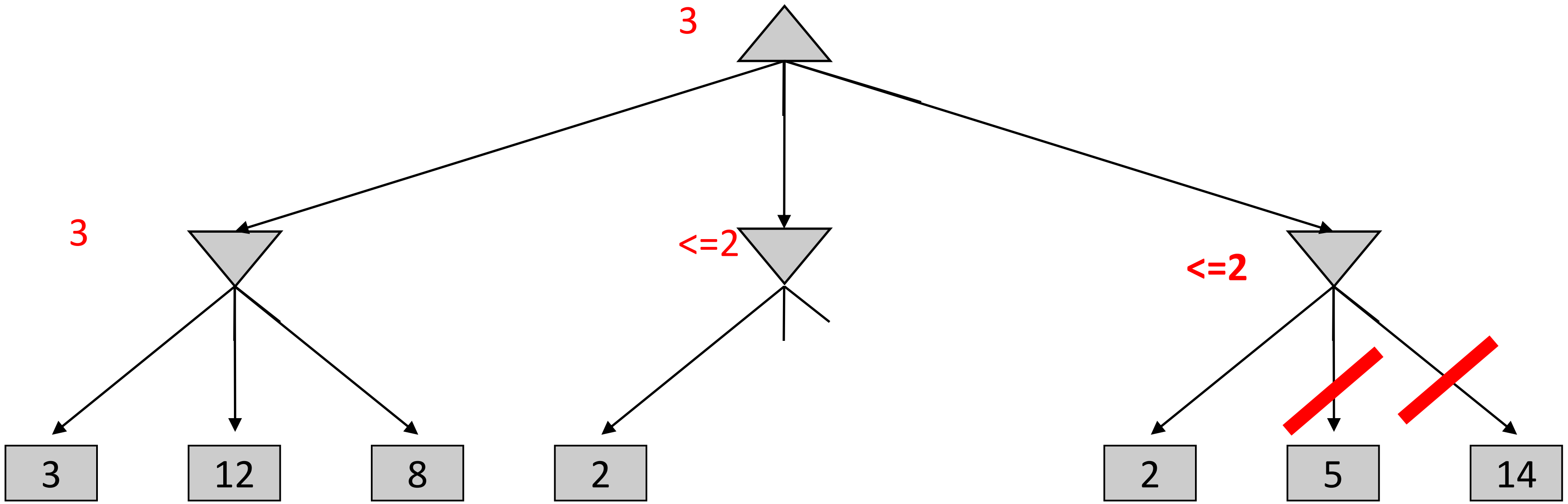
Alpha-Beta Pruning - Properties

1. Pruning has **no effect** on the minimax value at the root.
    - Pruning does not affect the final action selected at the root.
2. A form of **meta-reasoning** (computing what to compute)
    - Eliminates nodes that are irrelevant for the final decision.

Alpha-Beta Pruning – Order of nodes matters



Alpha-Beta Pruning – Order of nodes matters



Alpha-Beta Pruning - Properties

1. Pruning has **no effect** on the minimax value at the root.
    - Pruning does not affect the final action selected at the root.
2. A form of **meta-reasoning** (computing what to compute)
    - Eliminates nodes that are irrelevant for the final decision.
3. The alpha-beta search cuts the largest amount off the tree when we examine the **best move first**
    - However, best moves are typically **not** known. Need to make estimates.

Alpha-Beta Pruning – Order of nodes matters

```
                      4                    MAX
          +-----------+-----------+
          2           3           4        MIN
      +---+---+   +---+---+   +---+---+
      6   4   2   7   5   3   8   6   4    MAX
    +-+ +-+ +-+ +-+-+ +-+ +-+ +-+ +-+ +-+-+
    6 5 4 3 2 1 1 3 7 4 5 2 3 8 2 1 6 1 2 4
```

If the nodes were indeed encountered as "worst moves first" – then no pruning is possible

```
                      4                    MAX
          +-----------+-----------+
          4           3           2        MIN
      +---+---+   +---+---+   +---+---+
      4   6   8   3   x   x   2   x   x    MAX
    +-+ +-+ +-+ +-+         +-+-+
    4 2 6 x 8 x 3 2         1 2 1
```

If the nodes were encountered as "best moves first" – then pruning is possible

Note: In reality, we don't know the ordering.

Slide adapted from Prof. Mausam

Alpha-Beta Pruning - Properties

1. Pruning has **no effect** on the minimax value at the root.
   - Pruning does not affect the final action selected at the root.
2. A form of **meta-reasoning** (computing what to compute)
   - Eliminates nodes that are irrelevant for the final decision.
3. The alpha-beta search cuts the largest amount off the tree when we examine the **best move first**
   - Problem: However, best moves are typically **not** known.
   - Solution: Perform iterative deepening search and evaluate the states.
4. Time Complexity
   - **Best ordering - $O(b^{m/2})$.** Can double the search depth for the same resources. Effective branching factor becomes b^{1/2} instead of b.
   - On average – $O(b^{3m/4})$ if we expect to find the min or max after b/2 expansions.

Minimax for Chess

- Chess:
  - branching factor $b \approx 35$
  - game length $m \approx 100$
  - search space $b^m \approx 35^{100} \approx 10^{154}$
- The Universe:
  - number of atoms $\approx 10^{78}$
  - age $\approx 10^{18}$ seconds
  - 10^8 moves/sec x 10^{78} x 10^{18} = 10^{104}

Alpha-Beta for Chess

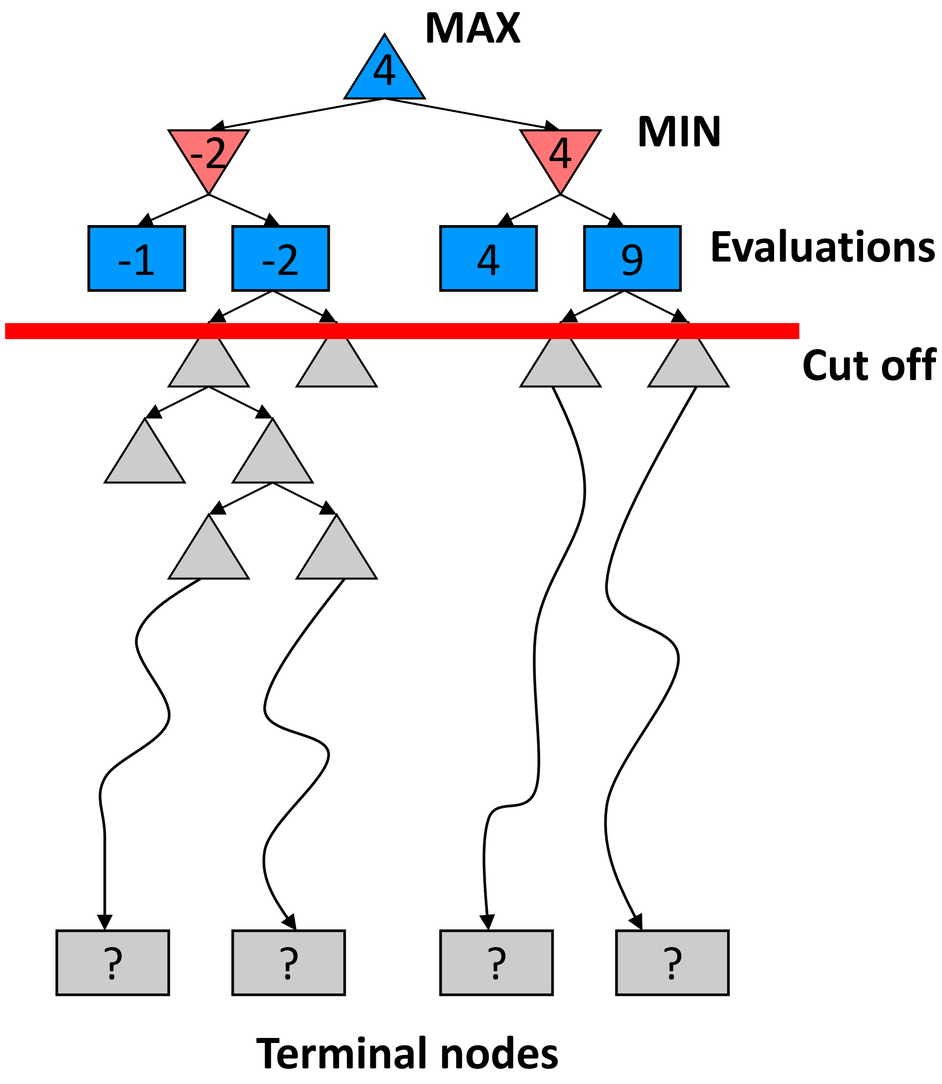
- Chess:
  - branching factor $b \approx 35$
  - game length $m \approx 100$
  - search space $b^{m/2} \approx 35^{50} \approx 10^{77}$

Slide adapted from Prof. Mausam

Cutting-off Search

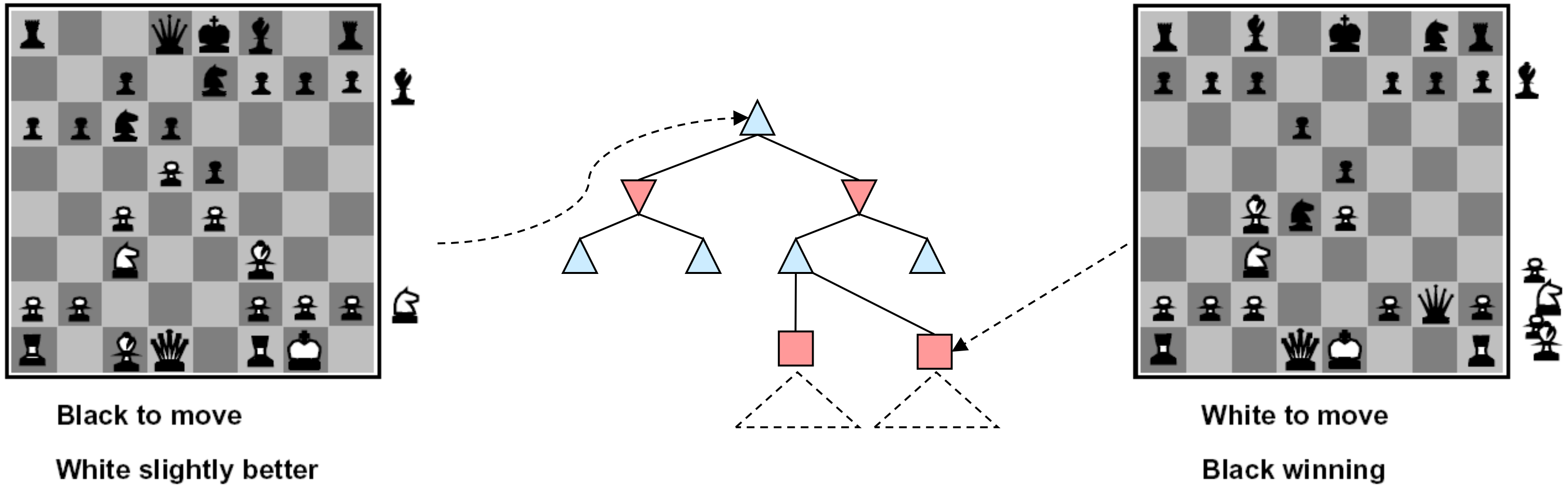
- Problem (Resource costraint):
  - Minimax search: full tree till the terminal nodes.
  - Alpha-beta prunes the tree but still searches till the terminal nodes.
  - We can’t search till the terminal nodes.
- Solution:
  - Depth-limited Search (H-Minimax)
  - Search only to a limited depth (cutoff) in the tree
  - **Replace the terminal utilities with an evaluation function for non-terminal positions.**

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



Evaluation Functions

- Evaluation functions score non-terminals in depth-limited search.
- Estimate the chances of winning.



- Ideal function: returns the actual **minimax** value of the position
- In practice: typically weighted linear sum of features:

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

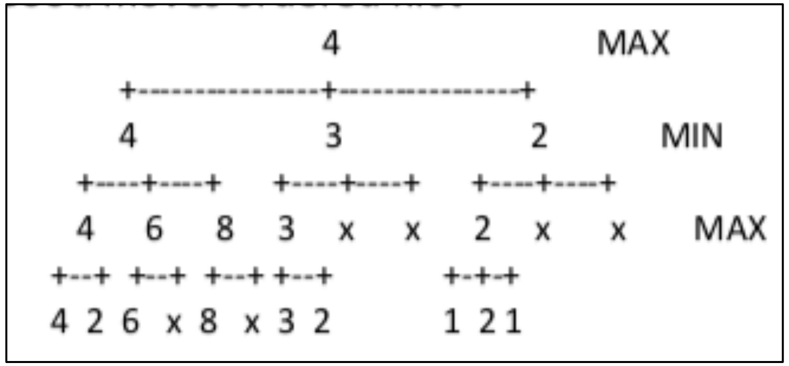
- e.g. $f_i(s)$ = (number of pieces of type i), each weight w_i etc.

Evaluation Functions

- Evaluation functions take a state and output an estimate of the true minimax value of that node.
  - Typically, “better” states will be assigned higher values by a good evaluation function in comparison to ”worse” states. Evaluation functions serve a similar purpose as heuristics in classical search.
- Depth-limited search applies evaluation function at the maximum solvable depth
  - Gives them mock terminal utilities by the evaluation function.
- Evaluation functions require features (some aspect of the current state).
  - Functions may or may not be linear. Require considerable thought and experimentation for designing.
- The better the evaluation function is, the closer the agent will come to behaving optimally.
  - Going deeper into the tree before using the evaluation function also tends to give better results. Reduces the compromise of optimality.

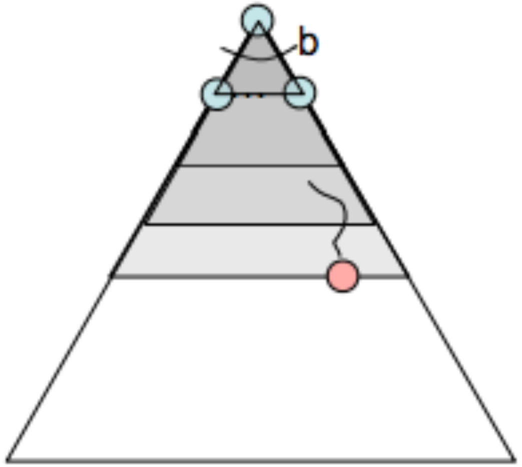
Determining “good” node orderings

- The ordering of nodes helps alpha-beta pruning.
  - Worst ordering $O(b^m)$. Best ordering $O(b^{m/2})$.



- How to find good orderings
  - Problem: we only know them when we evaluate the nodes.

- One approach – iterative deepening to determine evaluations for nodes
  - What if we can do iterative deepening to a certain depth. Use the evaluation function at the set depth and then compute the values for the nodes in the tree that is generated.
  - Next time, use the evaluations of the previous search to order the nodes. Use them for pruning.
  - Use evaluations of the previous search for order.

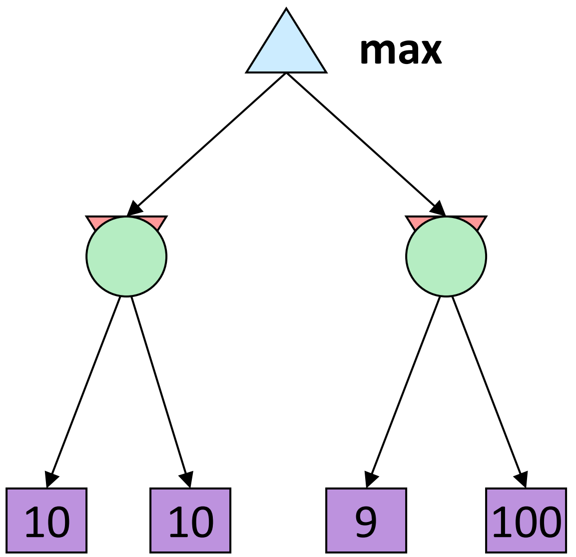


Game of Chance: Expectimax

- When the result of an action is not exactly known. Need a notion of uncertainty or chance in action selection.

- Explicit randomness in the opponent's action selection
  - Unpredictable opponents: the ghosts move randomly in Pacman.
  - Rolling dice by a player in a game.

- Pessimistic assumption is not valid for the adversary
  - The adversary may not be that bad. May not provide the worst value. Optimal response may not be guaranteed.



Expectimax:

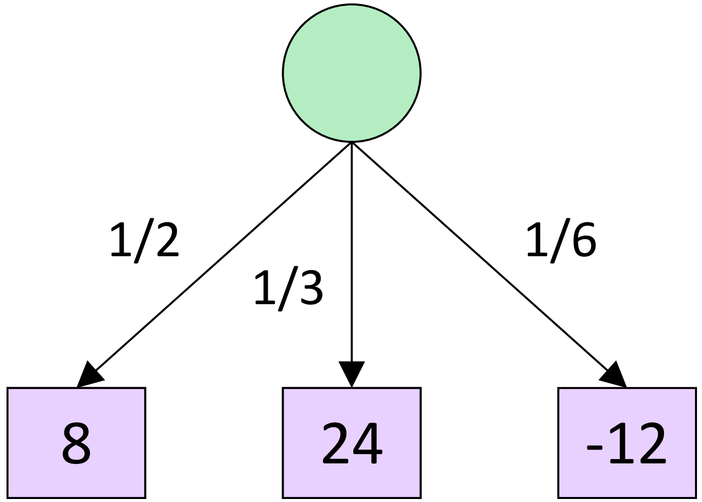
At chance nodes the outcome is uncertain. Calculate the ***expected utilities:*** weighted average (expectation) of children

Expectimax Search

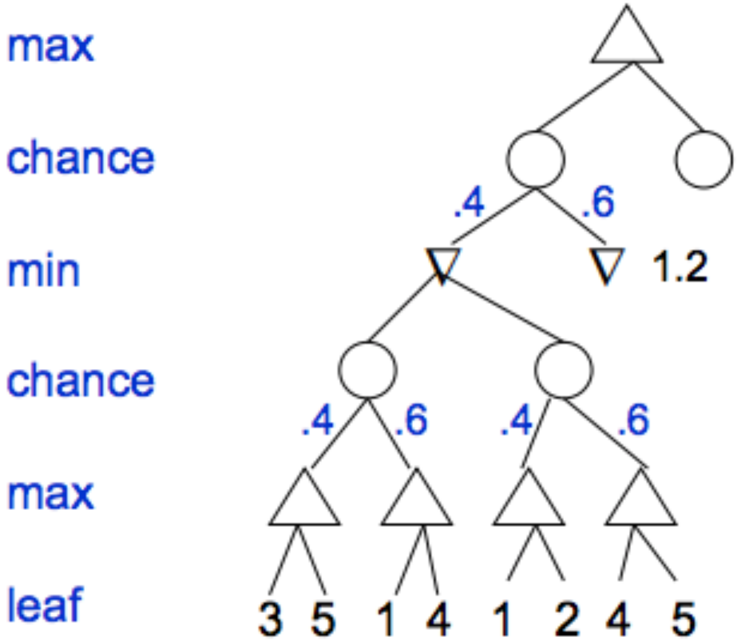
$$\forall \text{ agent-controlled states}, V(s) = \max_{s' \in successors(s)} V(s')$$

$$\forall \text{ chance states}, V(s) = \sum_{s' \in successors(s)} p(s'|s)V(s')$$

$$\forall \text{ terminal states}, V(s) = \text{ known}$$

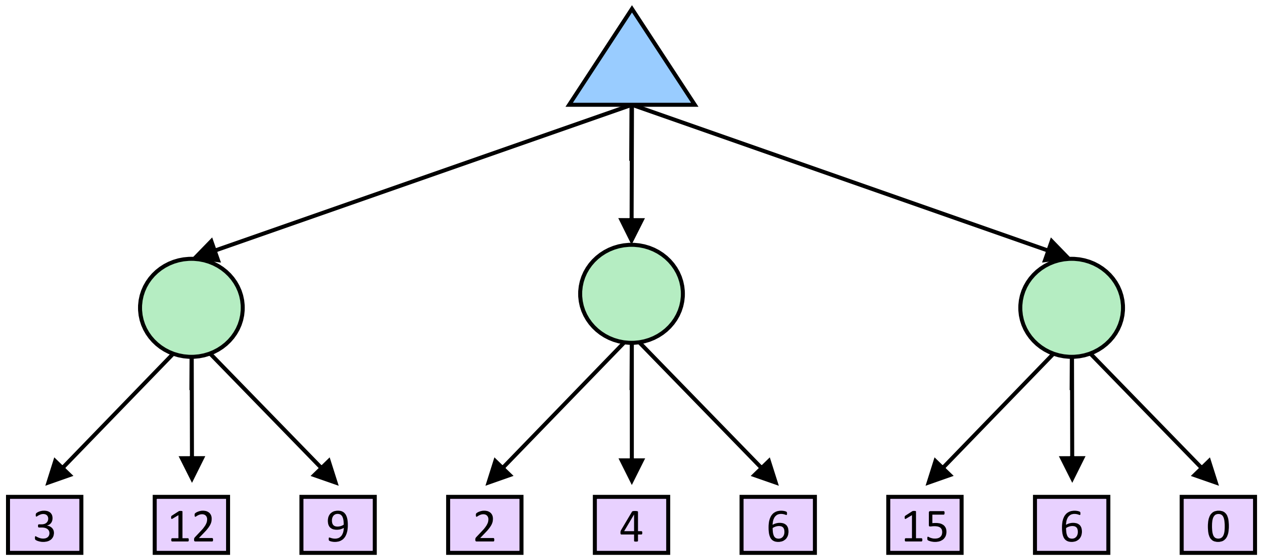


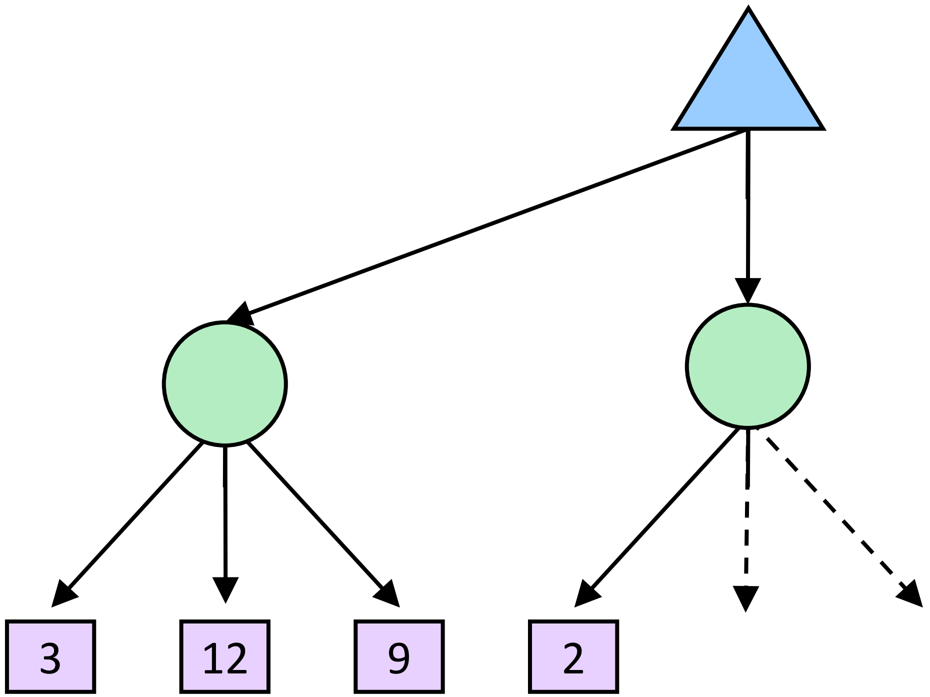
v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10



Mixed-type layers in a game tree are also possible. More than two agents.

Expectimax Search





Can we perform pruning?

Expectimax Search

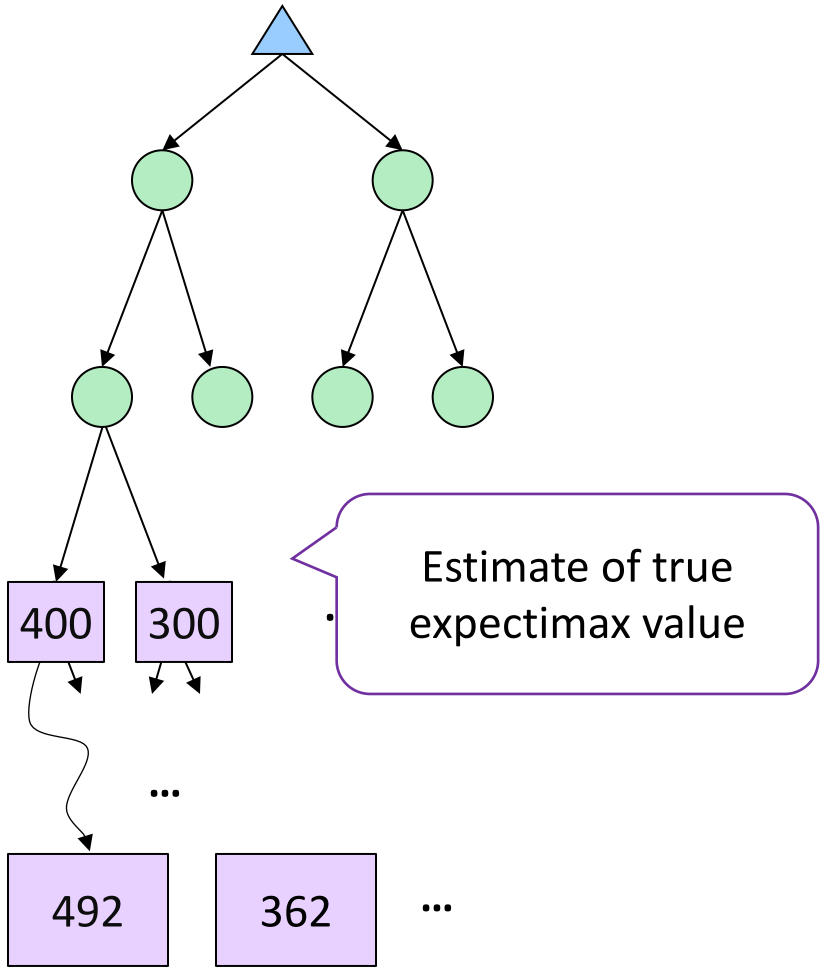
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
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
```

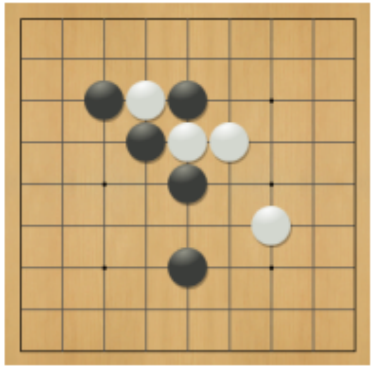
Depth-Limited Expectimax

- Depth-limit can also be applied in Expectimax search.
- Use heuristics to estimate the values at the depth limit.



Example: Game of Go

- The game of Go originated in China more than 2000 years ago.
- Usually played on 19x19, also 13x13 or 9x9 board
- Black and white place down stones alternately.
- Surrounded stones are captured and removed.
- The player with more territory wins the game.
- Complex strategy for capturing and creating a territory.
- Grand challenge in AI game playing because of its complexity.



Example: Game of Go

Significantly higher branching factor compared to Chess.
- Alpha-beta pruning/minimax does not scale. Not easy to evaluate all the action outcomes.

Design of a heuristic function is difficult
- Most positions are in a flux till the end game. Value not a strong indicator of winning.

Alternate approach, Monte-Carlo Tree Search.

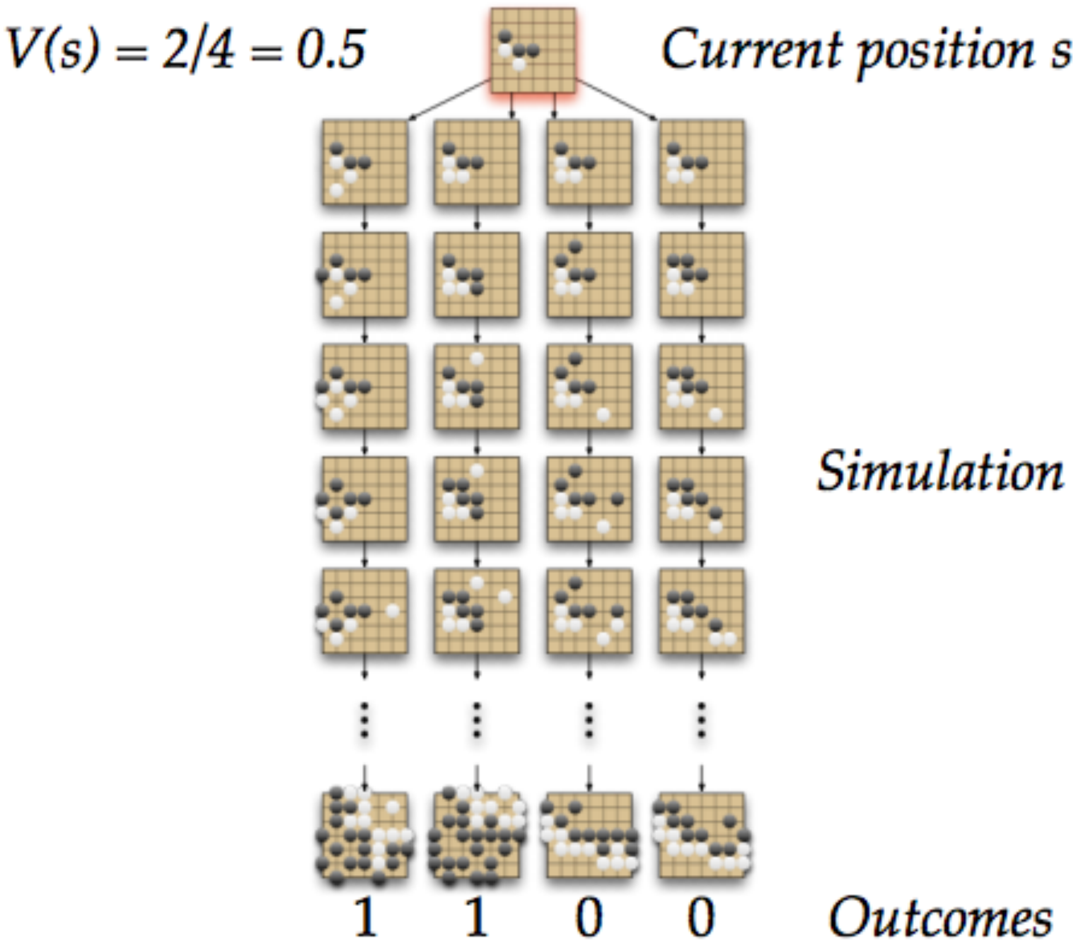
Popularized by Alpha Go
https://www.deepmind.com/research/highlighted-research/alphago

	Chess	Go
Size of board	8 x 8	19 x 19
Average no. of moves per game	100	300
Avg branching factor per turn	35	235
Additional complexity		Players can pass

Monte Carlo Tree Search (MCTS)

1. Simulations/Rollouts

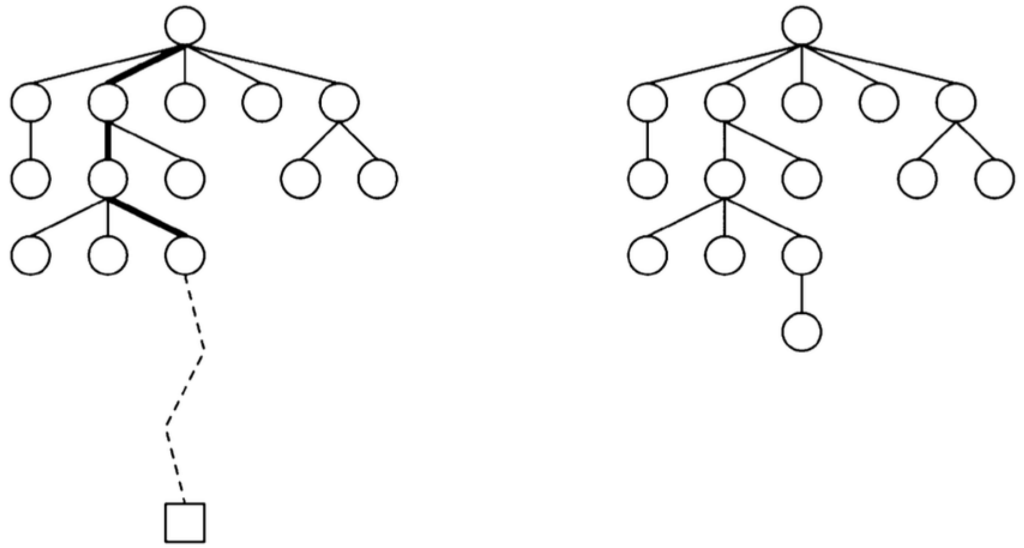
- Evaluation of a state V(s) using roll outs or simulating what will happen from this state on wards.
  - From state s play many times using a policy (e.g., random) and count wins and losses.
- For games in which the only outcomes are a win or a loss,
  - The “win percentage” approximates the “average utility”.

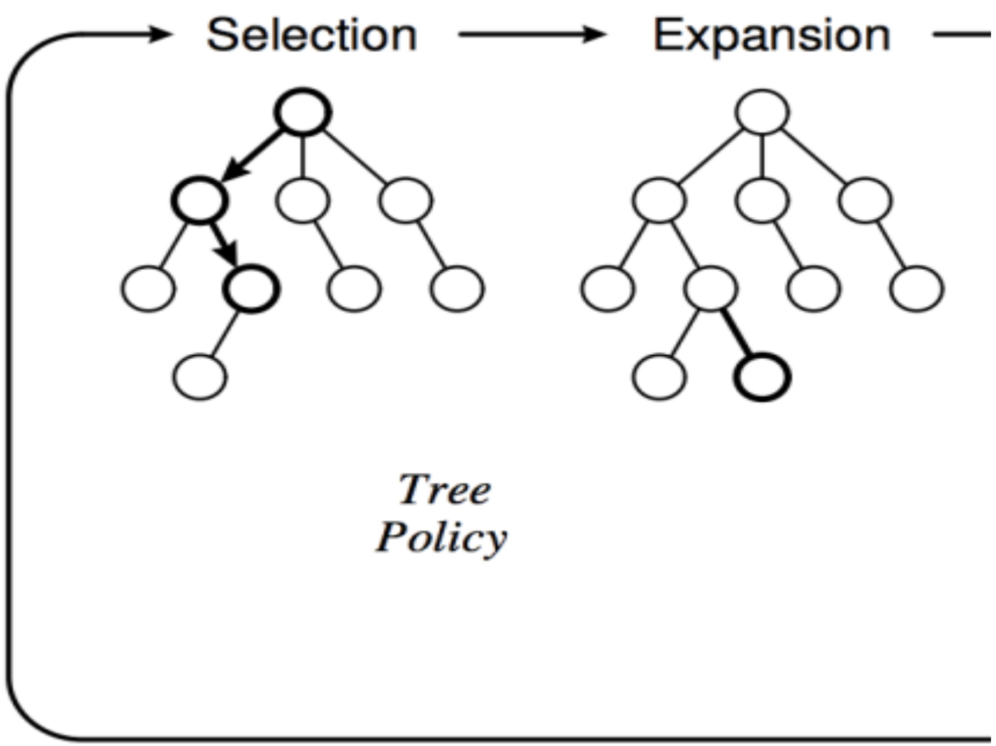


Monte Carlo Tree Search (MCTS)

2. Selective Search

- May not evaluate all states.
  - Be selective with evaluations on more promising actions/states.
- Explore parts of the tree (without an explicit depth for exploration) that will
  - Improve the decision at the root (improve the estimation of the value function)
  - Grow the tree of states as needed to improve the value estimates of a state.



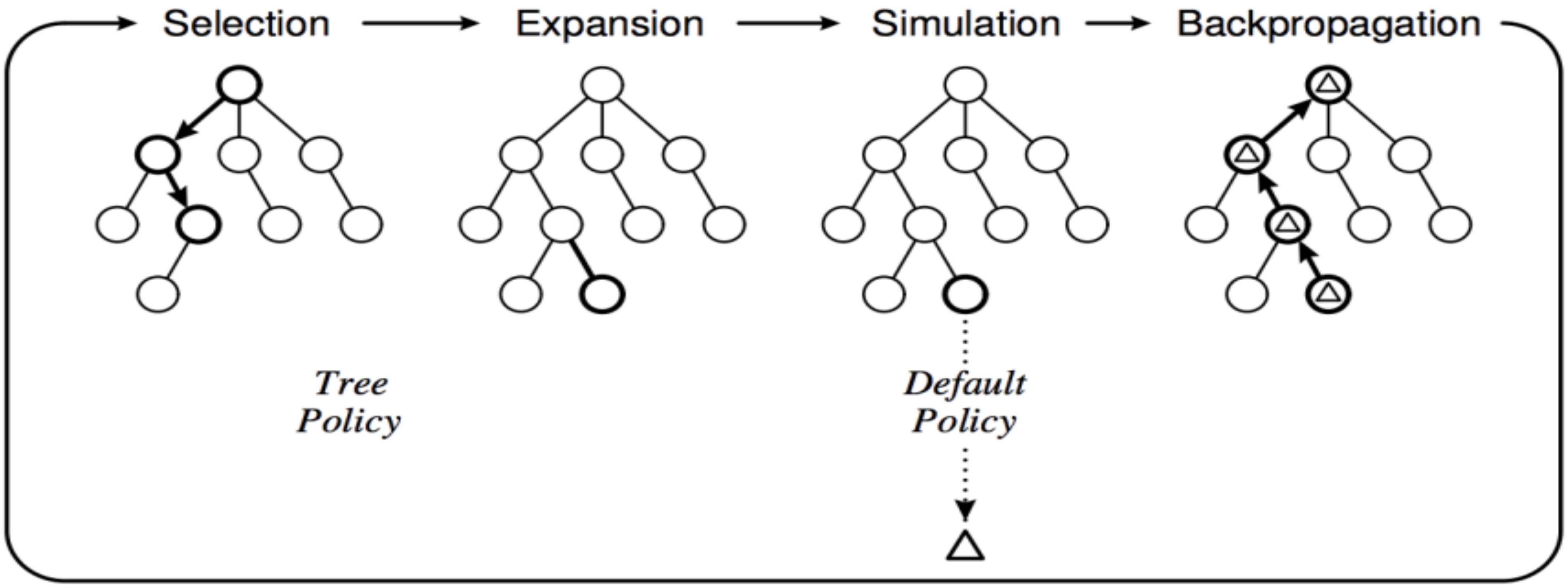


Selection

- Start from the root and select a move (via a selection/tree policy).
- Used for nodes we *have* seen before

Expansion

- When we reach the frontier, grow the search tree by generating a new child node of the node selected from the frontier.



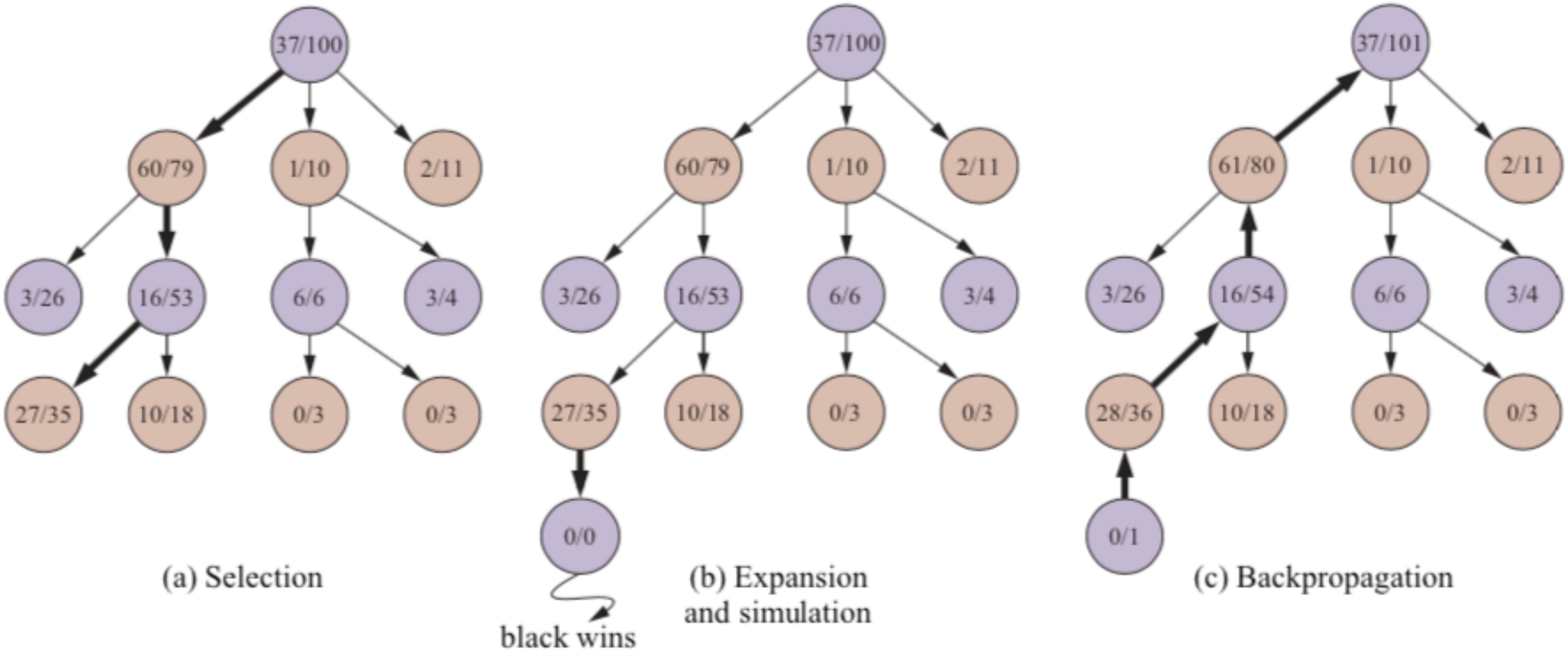
Selection
- Start from the root and select a move (via a selection/tree policy).
- Used for nodes we *have* seen before

Expansion
- When we reach the frontier, grow the search tree by generating a new child node of the node selected from the frontier.

Simulation
- Perform playout from the newly generated child node.
- Select moves for both players according to a playout policy (also called default policy) such as random action selection.
- Do not record the nodes in the tree.

Backpropagation
- After reaching a terminal node
- Update value and visits for states expanded in selection and expansion

Example



(a) Selection

(b) Expansion and simulation

(c) Backpropagation

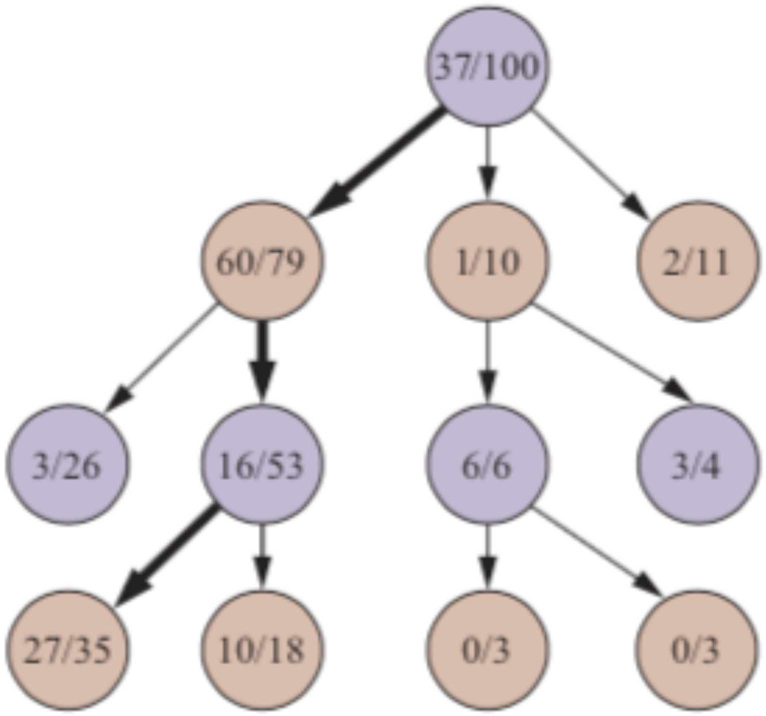
MCTS Procedure

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
  tree ← NODE(state)
  while IS-TIME-REMAINING() do
    leaf ← SELECT(tree)
    child ← EXPAND(leaf)
    result ← SIMULATE(child)
    BACK-PROPAGATE(result, child)
  return the move in ACTIONS(state) whose node has highest number of playouts
```

Exploration vs. Exploitation

Selection Strategy

- How to select moves/actions in the tree?
- Bias the moves towards those providing higher value.
- But we may not know about the value of certain states or may be very uncertain about them. Hence, sometimes we should explore too.
- Fundamental trade-off between exploration and exploitation.



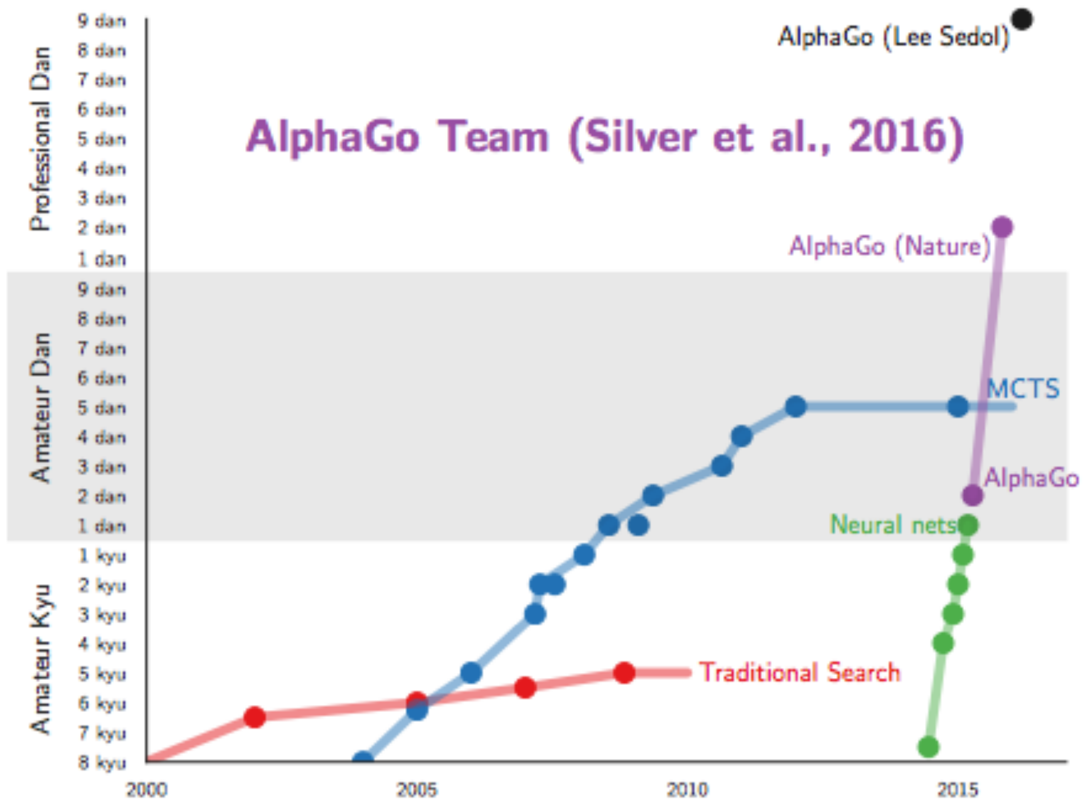
How to select the moves balancing exploration and exploitation.

Upper Confidence Bound applied to Trees

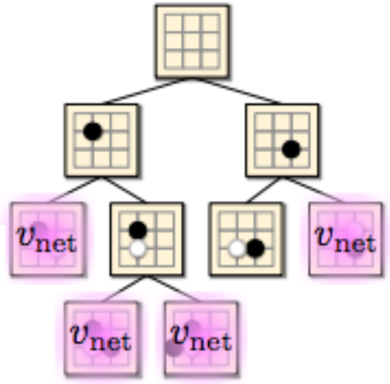
$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\textsc{Parent}(n))}{N(n)}}$$

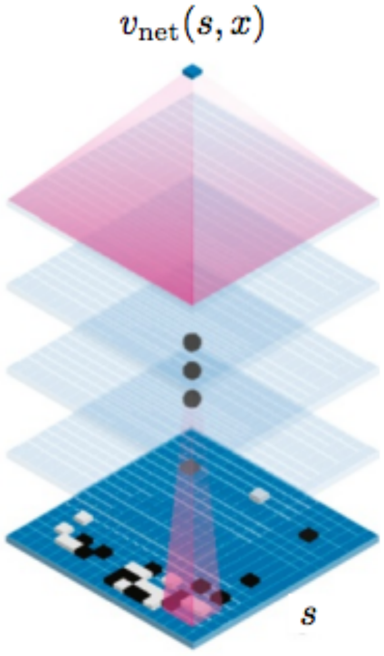
- $N(n)$ = number of rollouts from node n
- $U(n)$ = total utility of rollouts (e.g., # wins) for Player(Parent(n))
- C is the tunable parameter.

- The first term is the exploitation term: the average utility of node n.
- The second term is the exploration term: how uncertain we are about the node’s utility.
  - The denominator is the number of visits to the states, so states visited less often are preferred.
  - The numerator is the log of the number of times the parent is explored.
  - If we are selecting n for some non-zero percentage of times then the exploration term goes to zero as the counts increase.
- We will revisit this concept in the discussion on Reinforcement Learning later.



adapted from Sylvain Gelly & David Silver, Test of Time Award ICML 2017



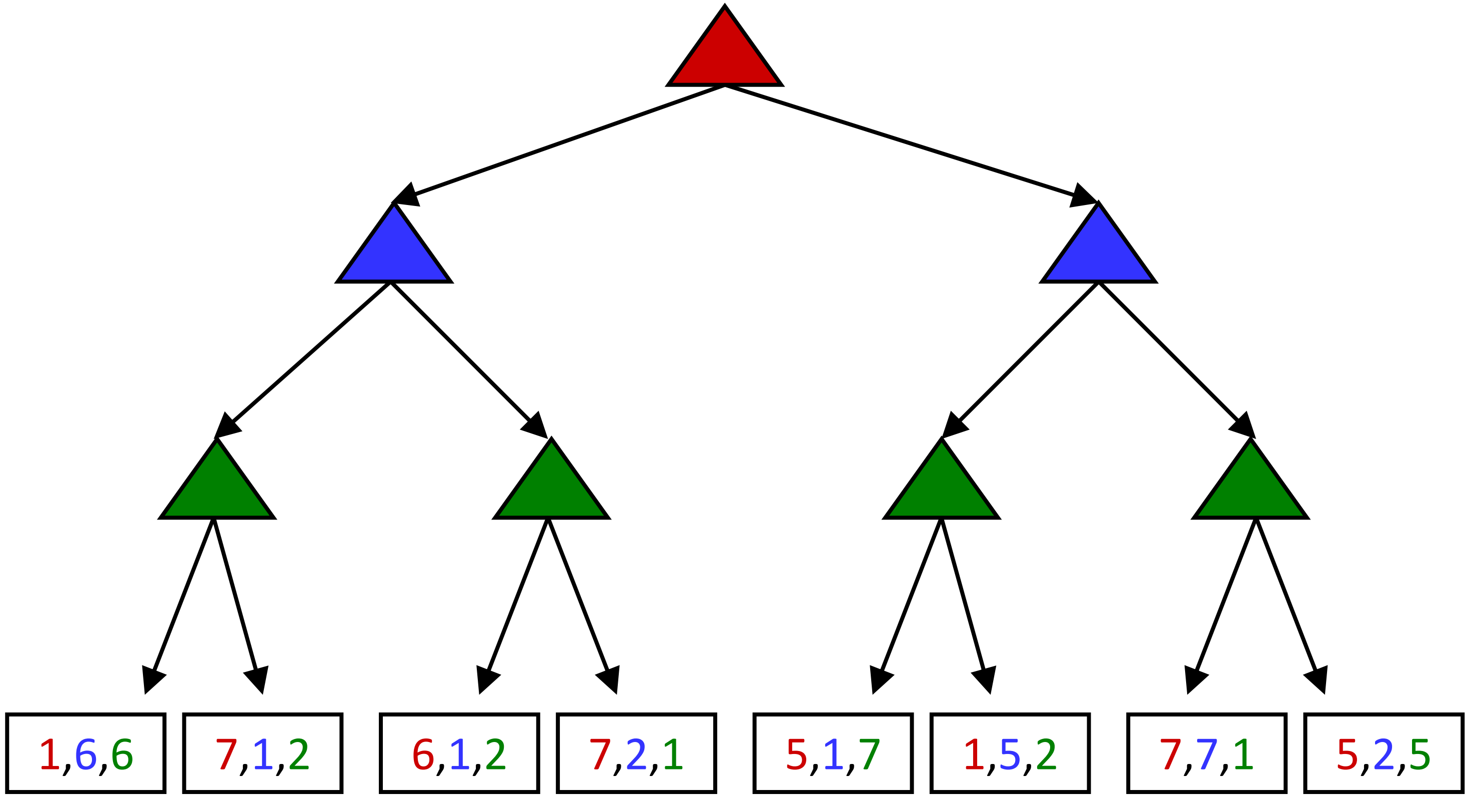


(Silver et al., 2016)

Alpha Go combined learning with MCTS (used a NN to predict values/utilities of states). Employed self play etc.

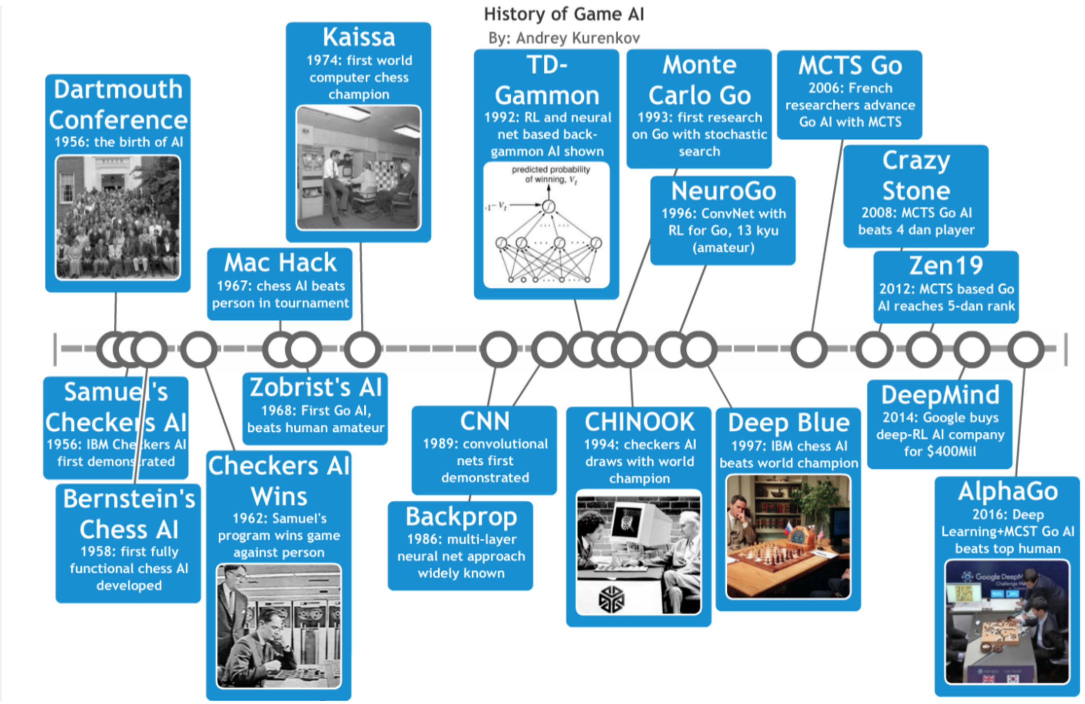
Multiple players and other games

- Not all games are zero sum.
  - Loss for one agent may not be win for the other agent.
  - Different agents may have different tasks in the game that don't directly involve strictly competing against each other.
- Multi-agent utilities.
  - Generalization of minimax.
  - Each player maximizes its own utility at each node they control and ignore the utilities of the other agents.
- General gams with multi-agent utilities
  - Can invoke cooperation
  - The utility selected at the root tends to yield a reasonable utility for all participating agents.



Game Playing AI: Wrap up

- Game playing domains
  - Very large amount of contingency reasoning.
- Exact decision making is nearly impossible.
  - Approximate evaluation functions etc.
  - Force efficient use of computation (alpha-beta pruning.)
- An important test bed for AI algorithms.
  - We play games intuitively, used to reasoning.
  - Easy to compare human and computer performance.
- Game playing has produced important research ideas
  - Reinforcement learning (checkers)
  - Iterative deepening (chess)
  - Monte Carlo tree search (chess, Go)
  - Solution methods for partial-information games in economics (poker)



"Games are to AI as grand prix is to automobile design"
Games viewed as an indicator of intelligence.