COL333/671: Introduction to AI
Semester I, 2021

Adversarial Search

Rohan Paul

Outline

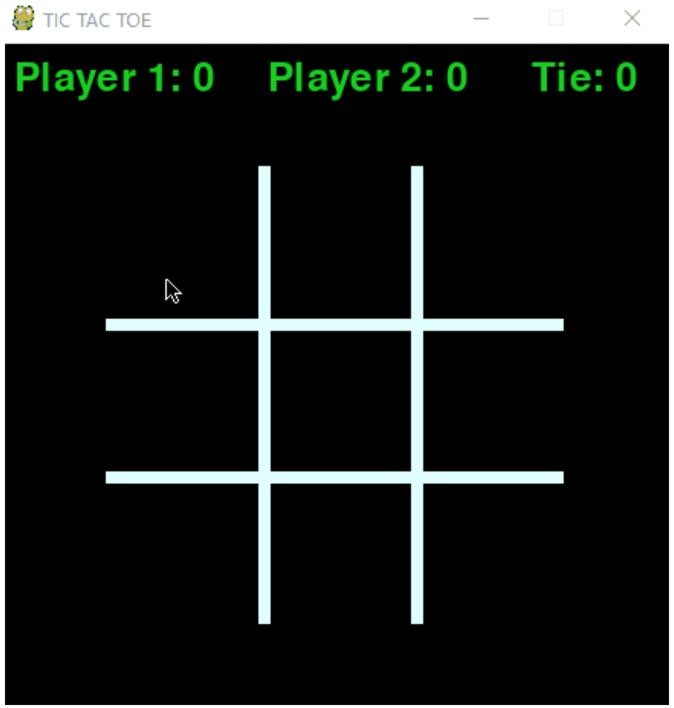
- Last Class
  - Local Search
- This Class
  - Adversarial Search
- Reference Material
  - AIMA Ch. 5 (Sec: 5.1-5.5)

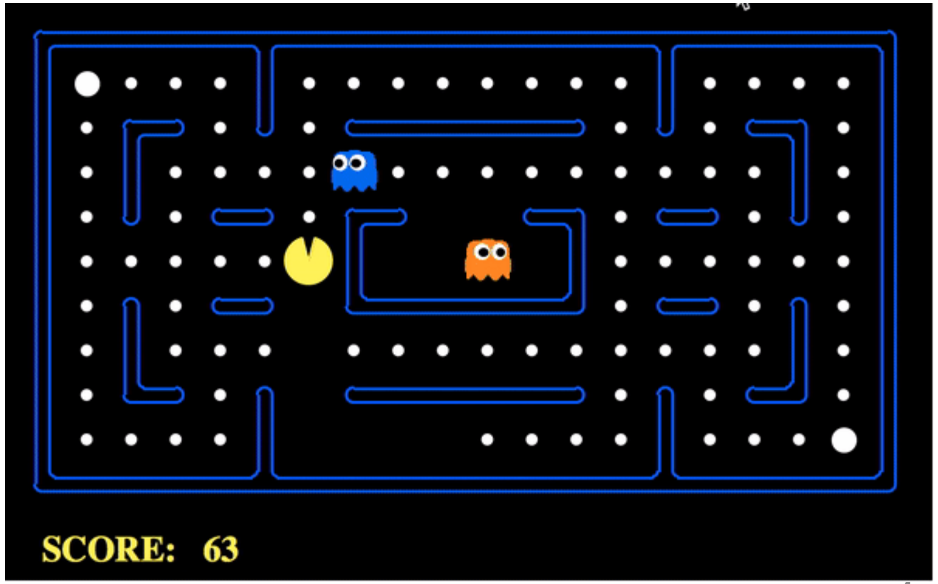
Acknowledgement

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!*

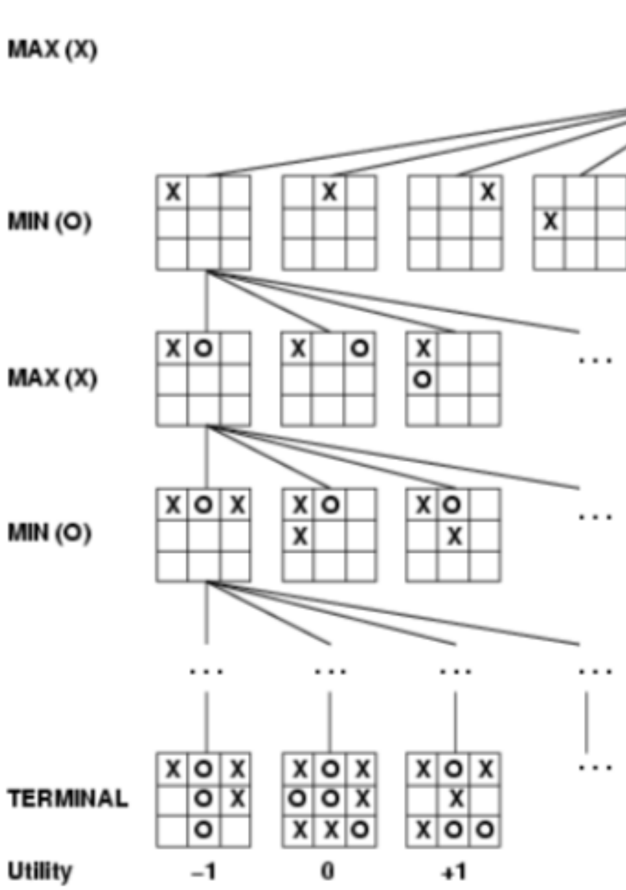
computer's turn

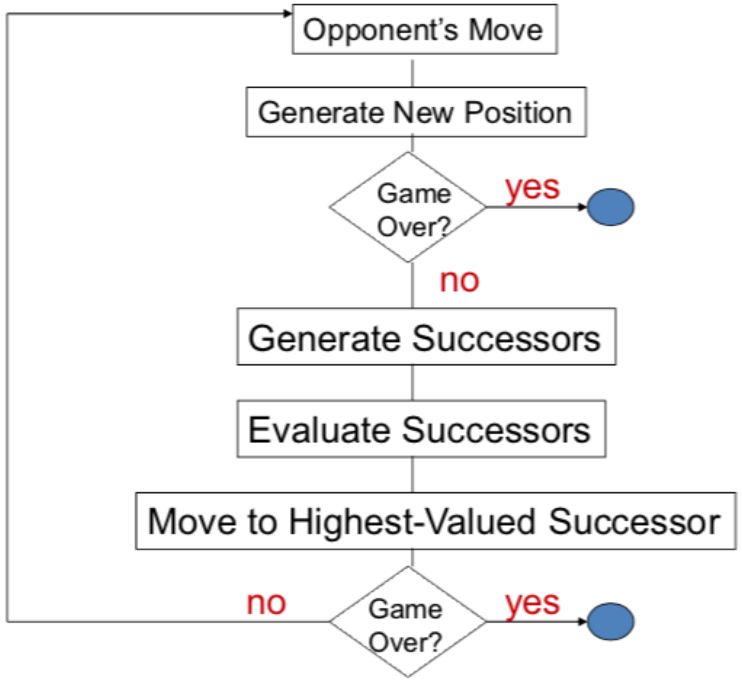
opponent's turn

computer's turn

opponent's turn

leaf nodes are evaluated

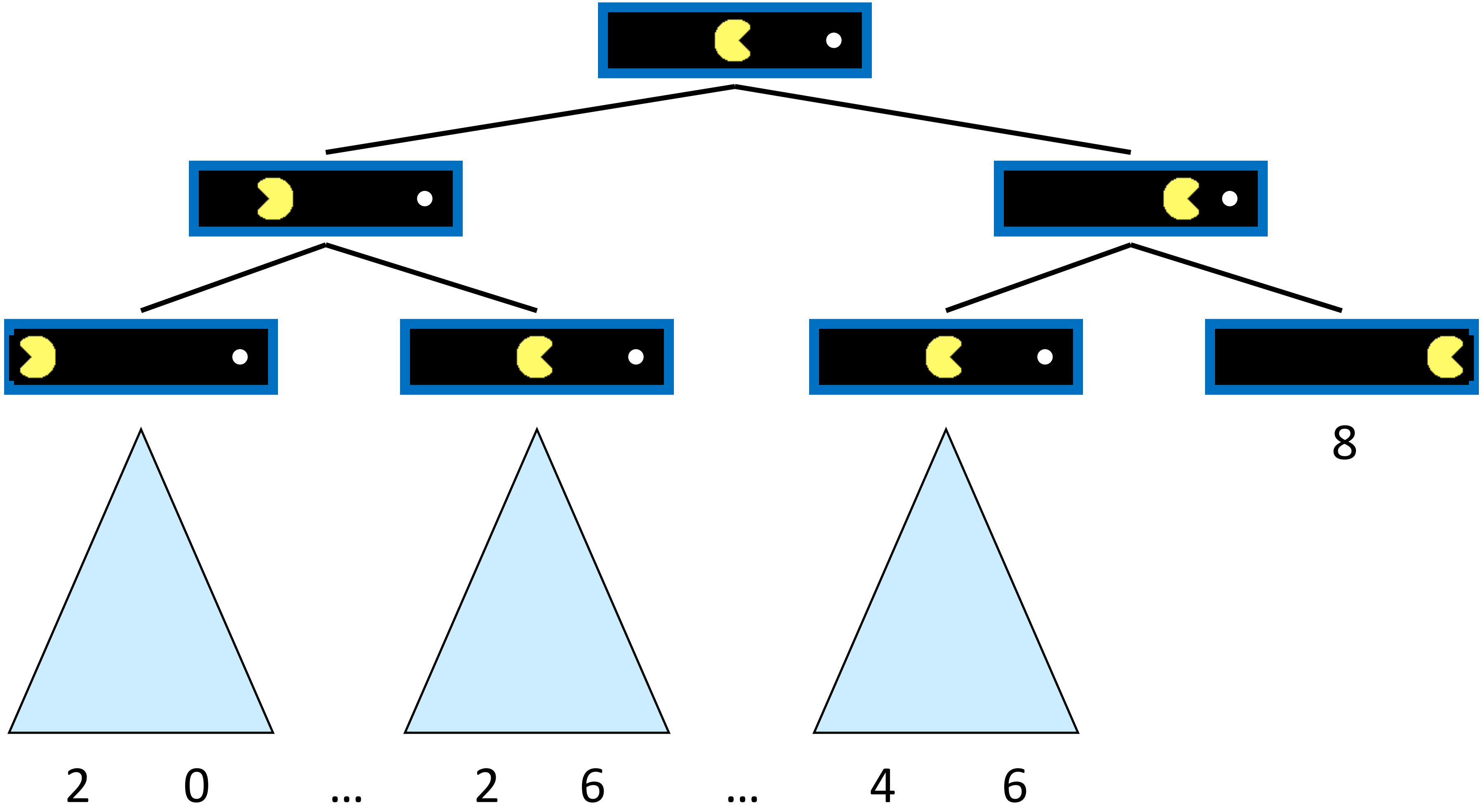




Terminal nodes we get -1, 0 or 1 for loss, tie or win. Think of this value as a ”utility” of a state.

in and from Mausam

Single-Agent Trees

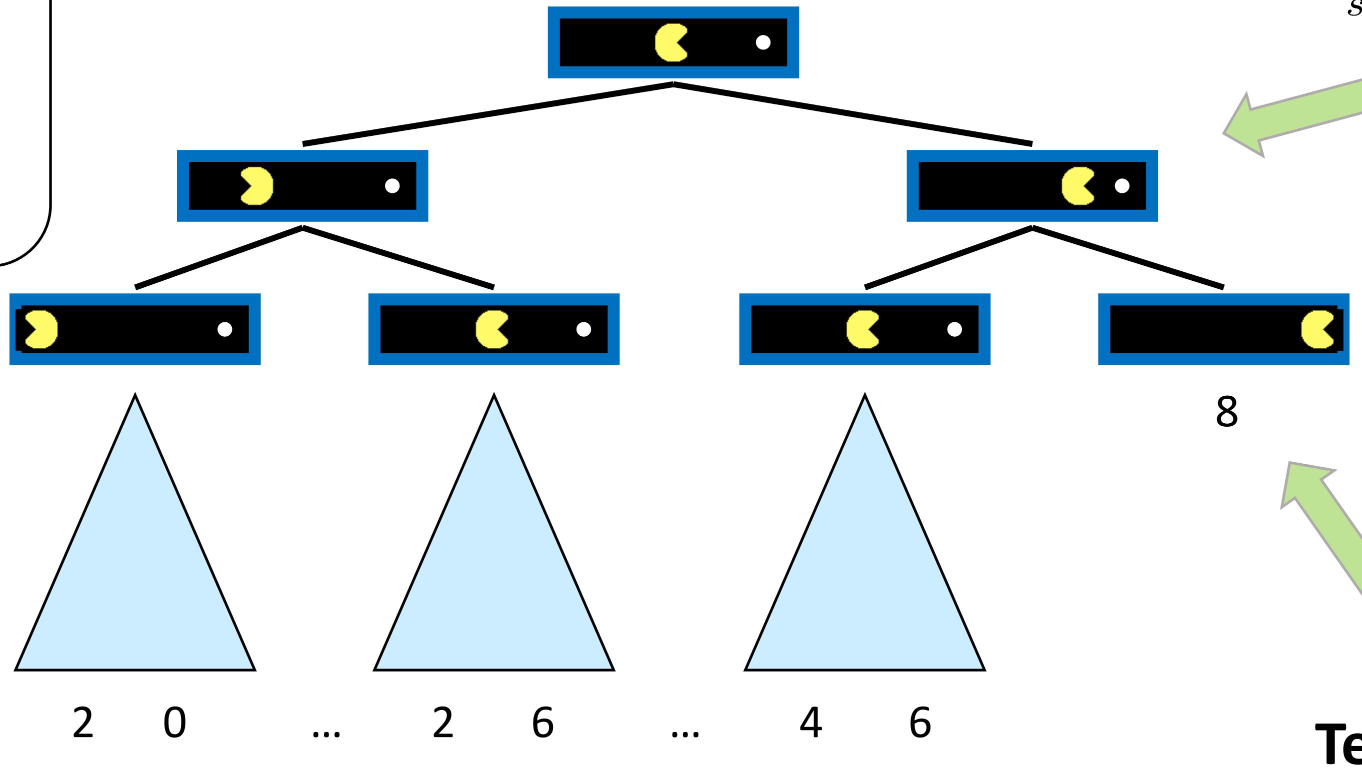


Computing "utility" of states to decide actions

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

Non-Terminal States:

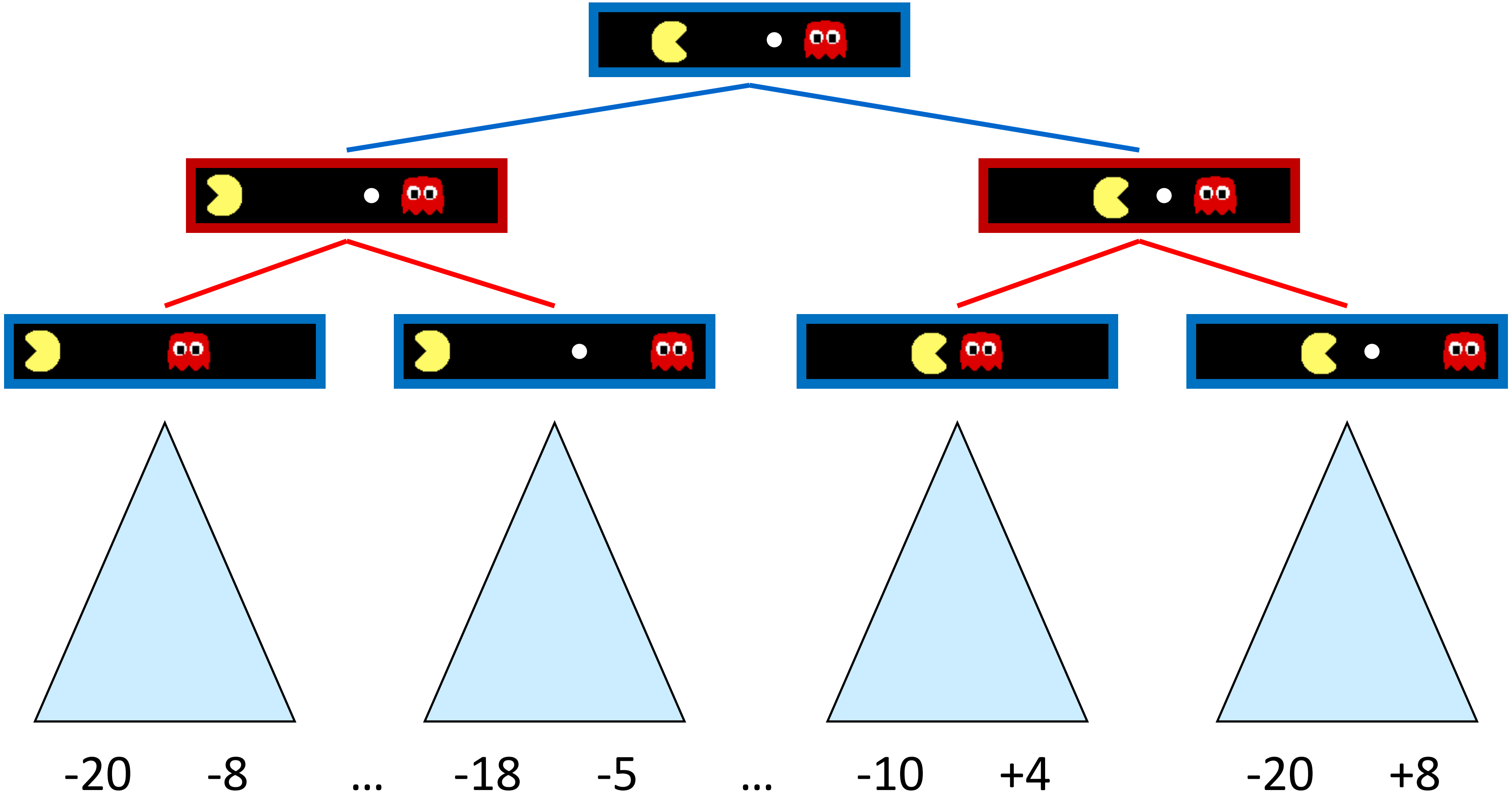
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



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.

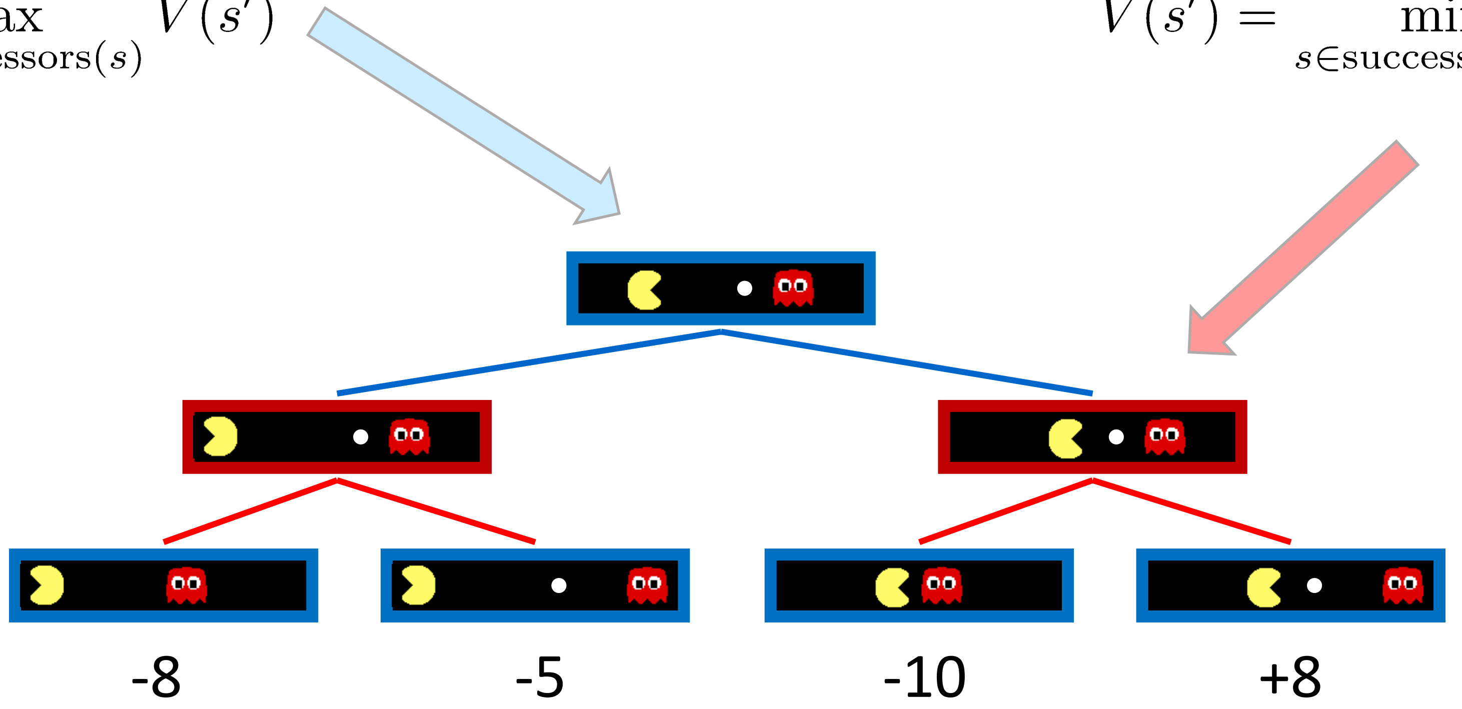
Minimax Values

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:

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

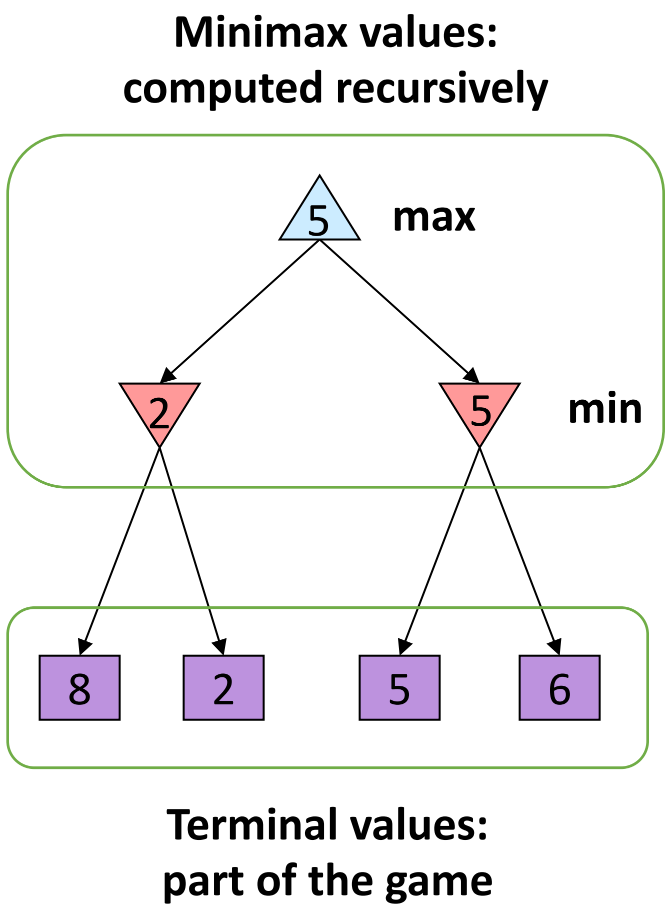
Adversarial Search (Minimax)

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

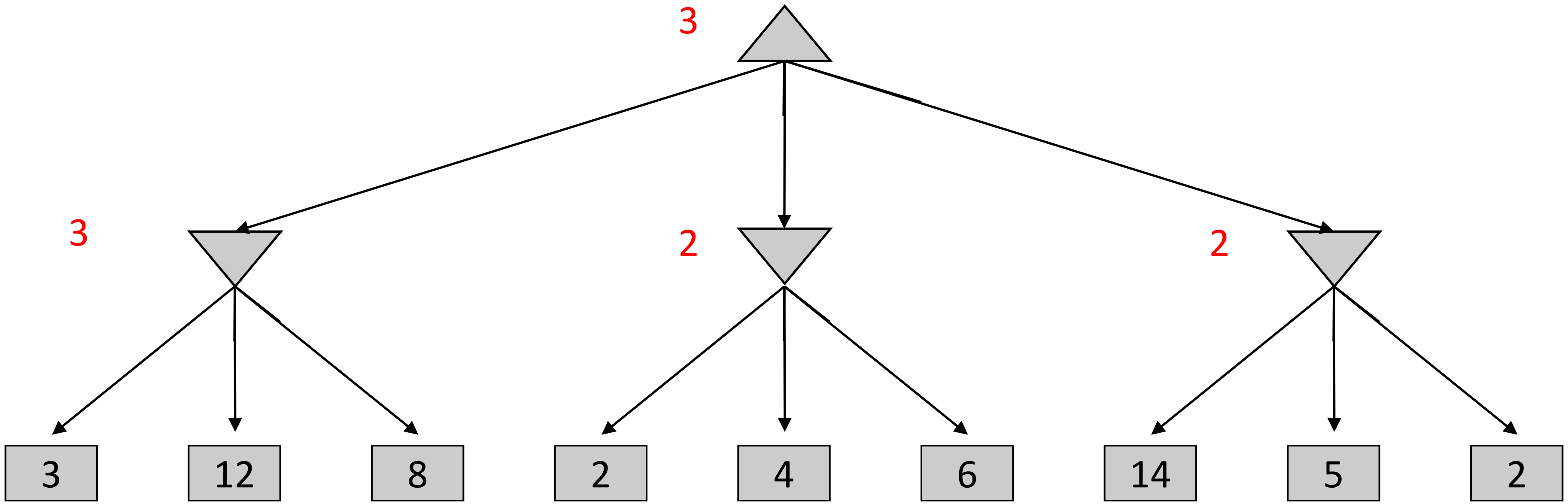
- Minimax Search
  - Search the game tree for best moves.
  - Select optimal actions that move to a position with the highest minimax value.
  - What is the minimax value?
    - It is the best achievable utility against the optimal (rational) adversary.
    - Best achievable payoff against the best play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

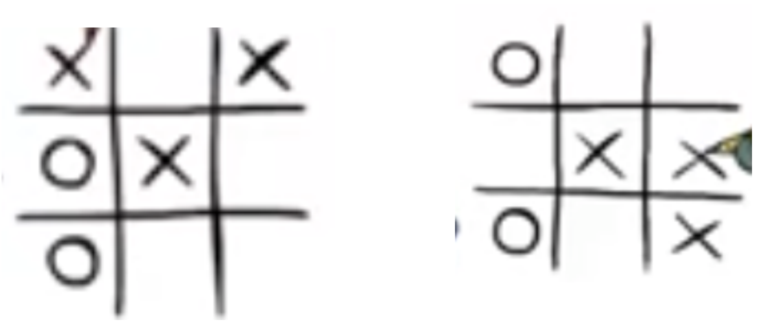
- **Requires growing the tree till the terminal nodes.**
- **Not feasible in practice for a game like Chess.**

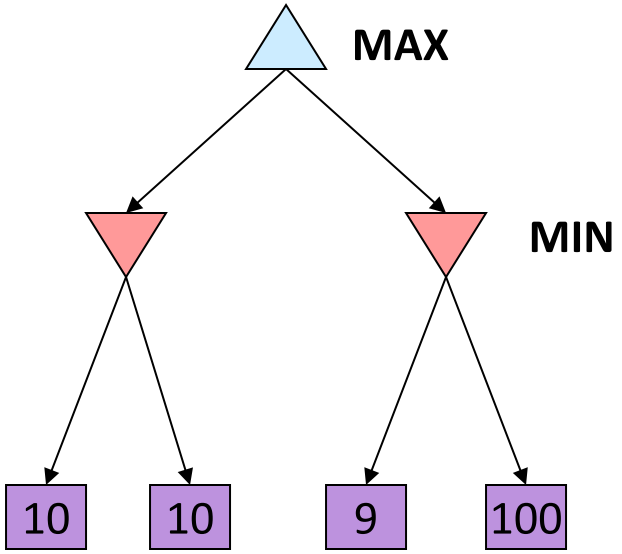
- Chess:
  - branching factor $b \approx 35$
  - game length $m \approx 100$
  - search space $b^m \approx 35^{100} \approx 10^{154}$
- The Universe:
  - number of atoms $\approx 10^{78}$
  - age $\approx 10^{18}$ seconds
  - 10^8 moves/sec x 10^{78} x 10^{18} = 10^{104}

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., <u>not</u> giving us the min value)
    - No. Why? It does not exploit the opponent’s weakness against a suboptimal opponent).

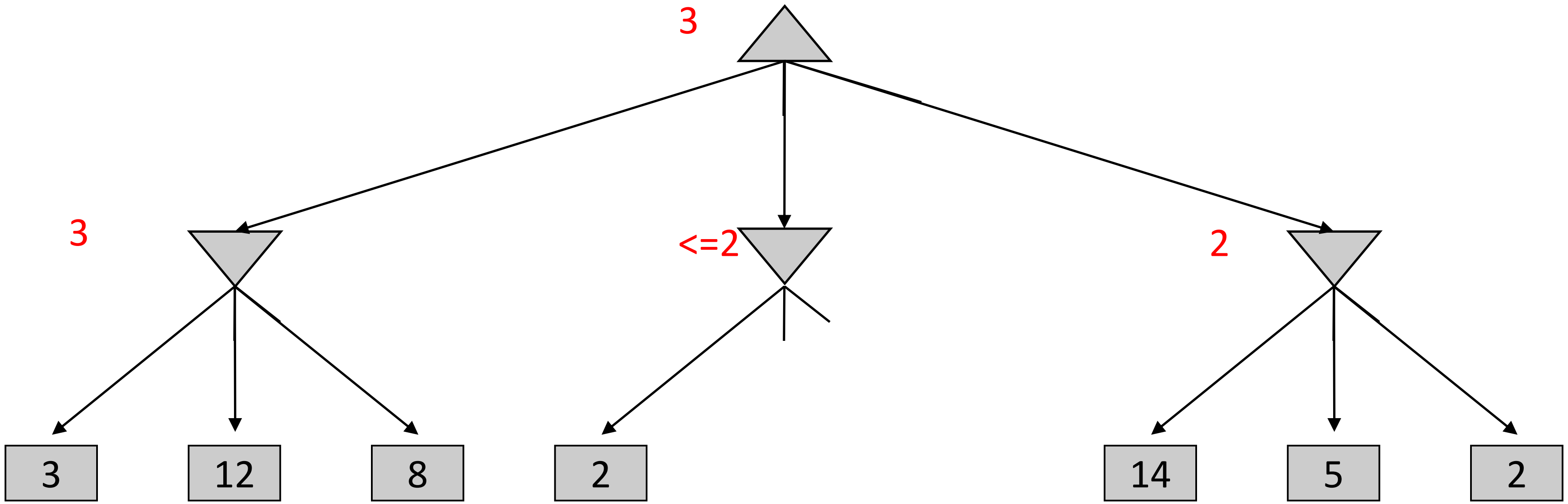
You: Cricle. Opponent: Cross





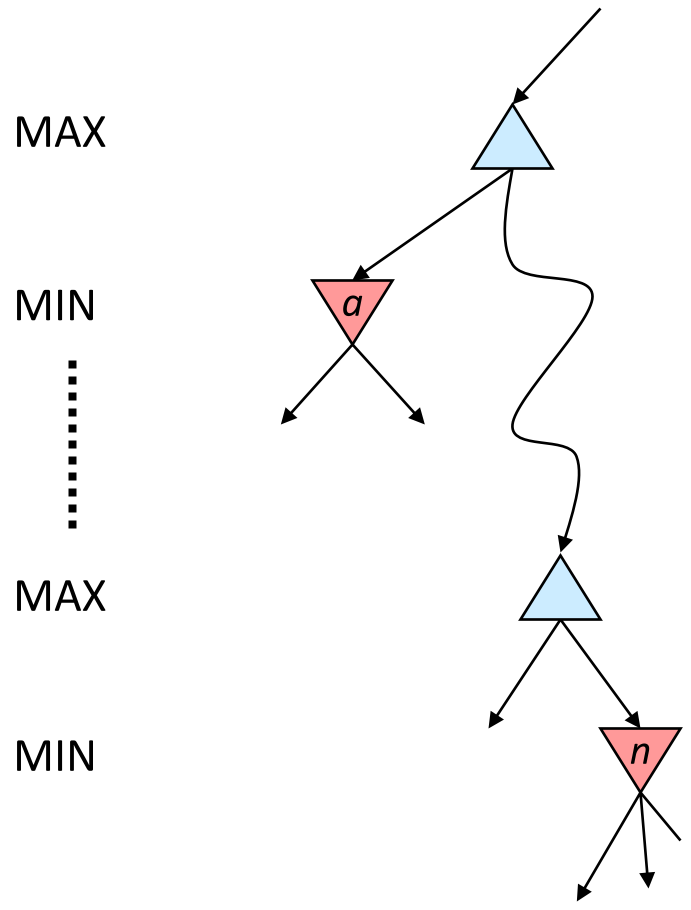
If min returns 9? Or 100?

Necessary to examine all values in the tree?



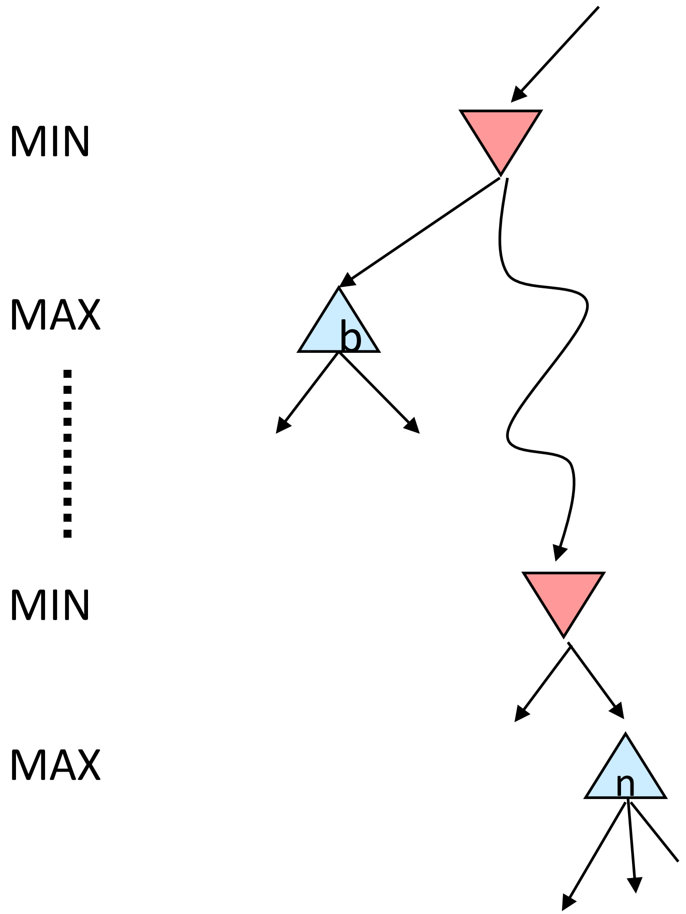
Alpha-Beta Pruning: General Idea

- **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)

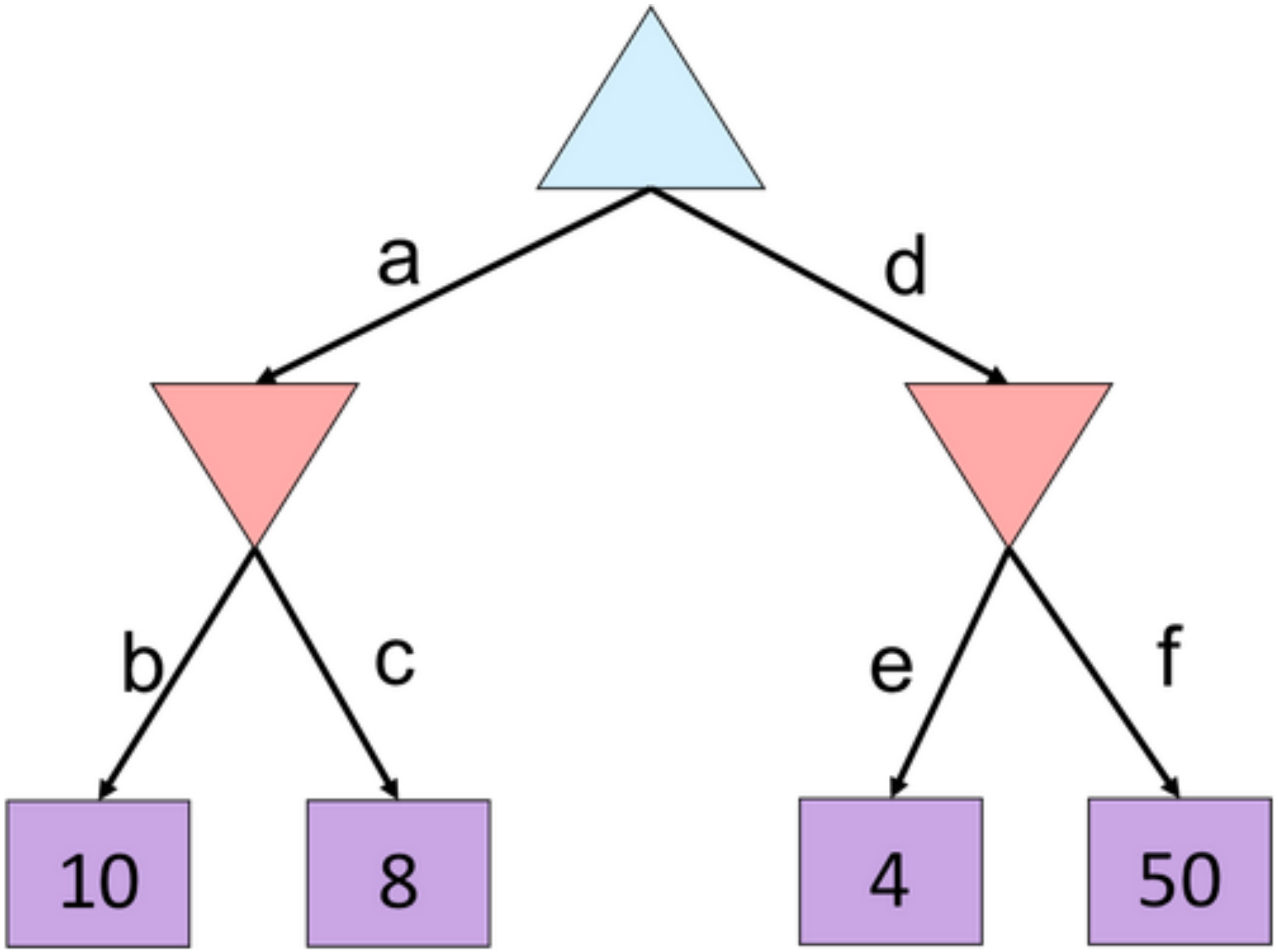


Alpha-Beta Pruning: General Idea

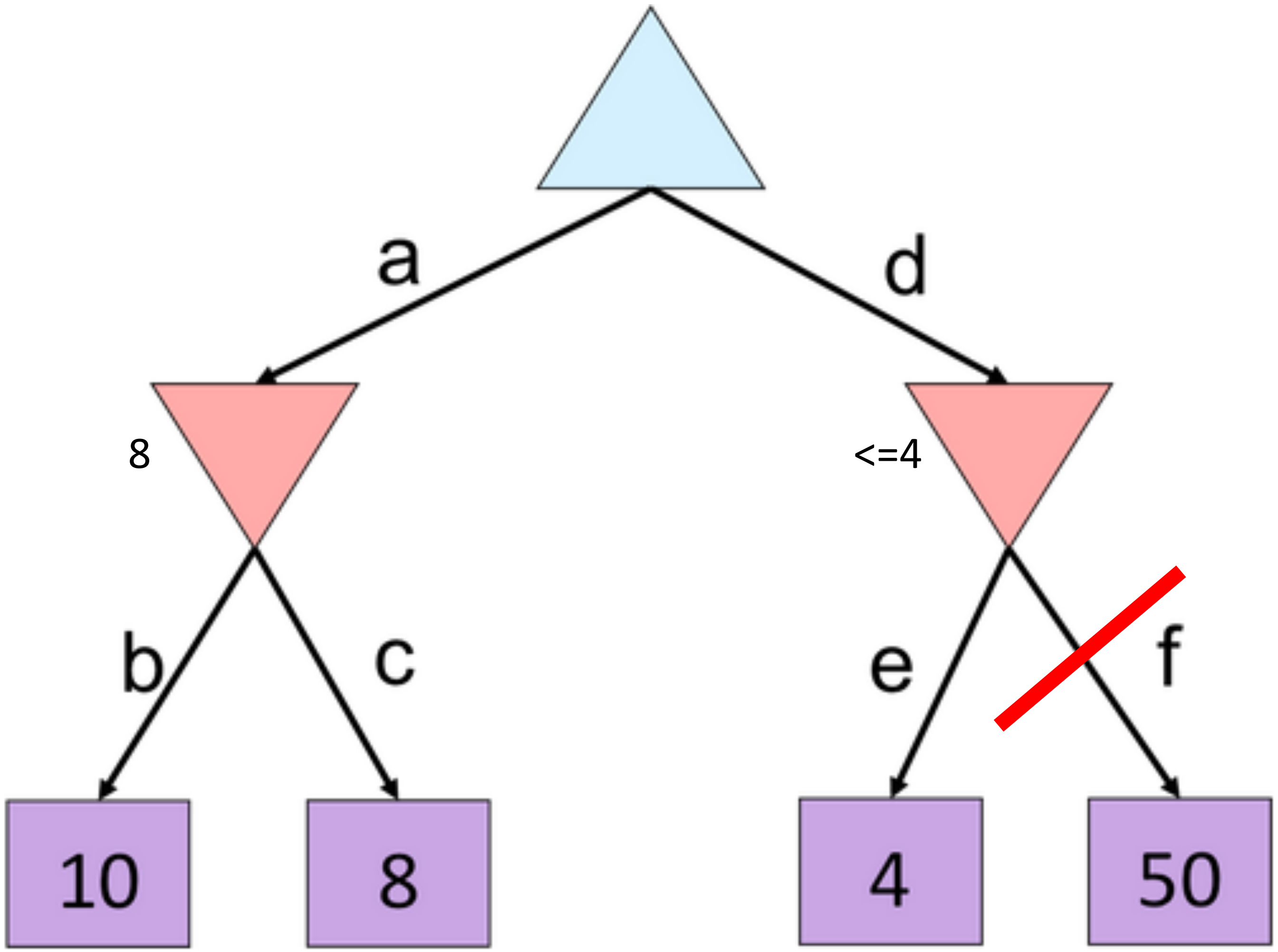
- **General Configuration (MAX version)**
  - Consider computing the MAX-VALUE at some node *n*, examining *n*'s children
  - *n*'s estimate of the childrens' min is increasing.
  - Who can use *n*'s value to make a choice? MIN
  - Let *b* be the lowest (best) value that MIN can get at any choice point 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)



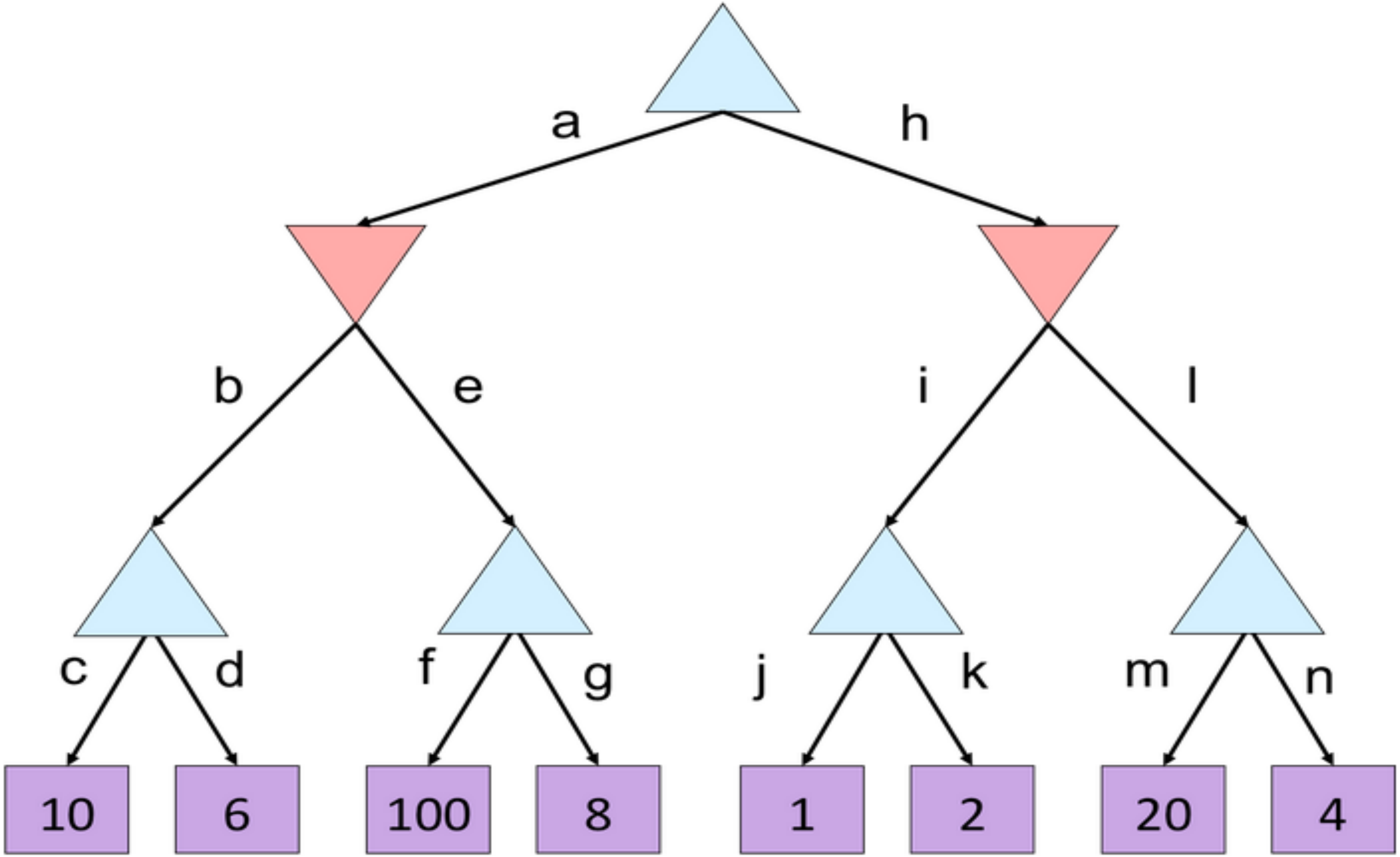
Pruning: Example



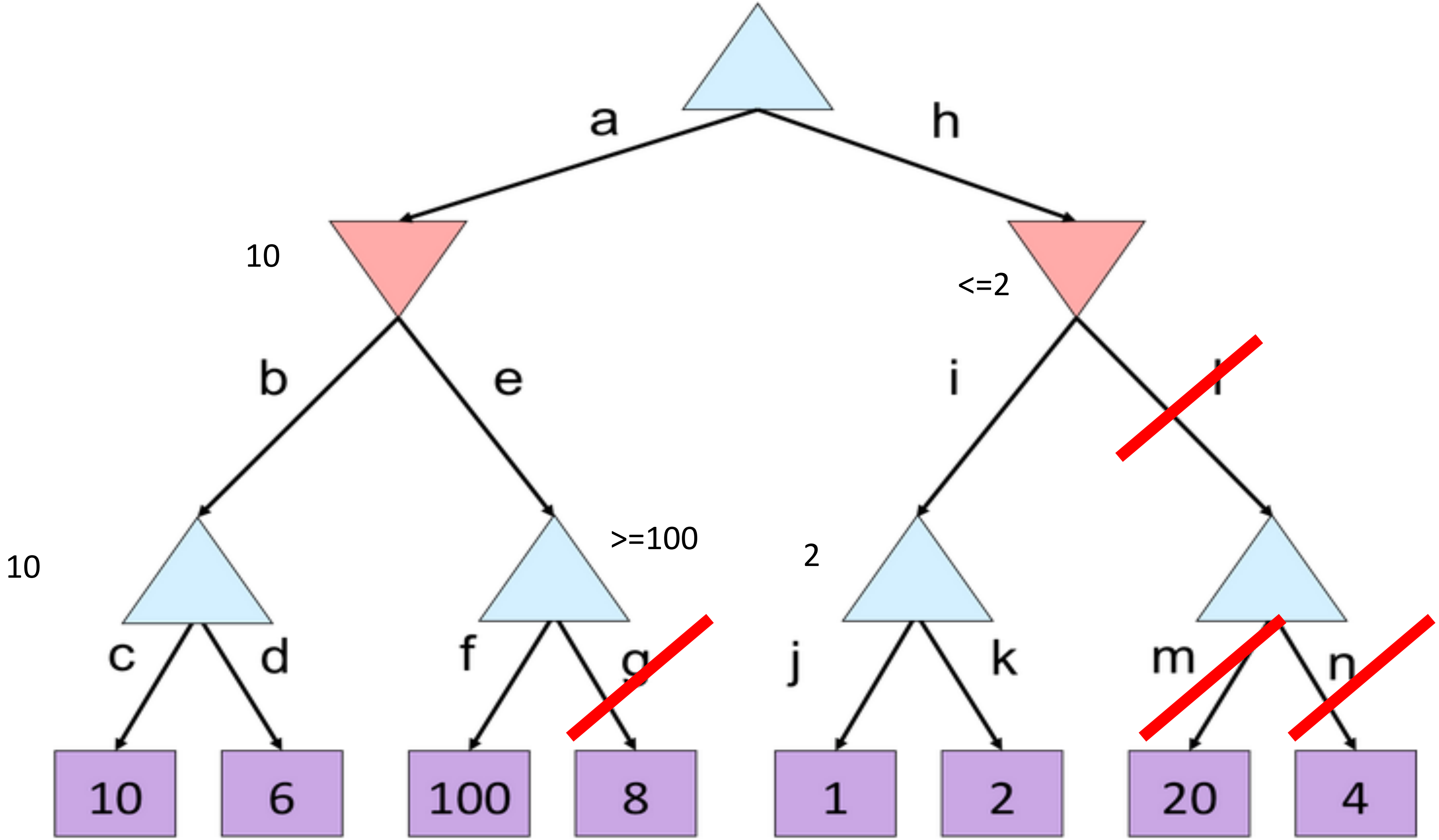
Pruning: Example



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

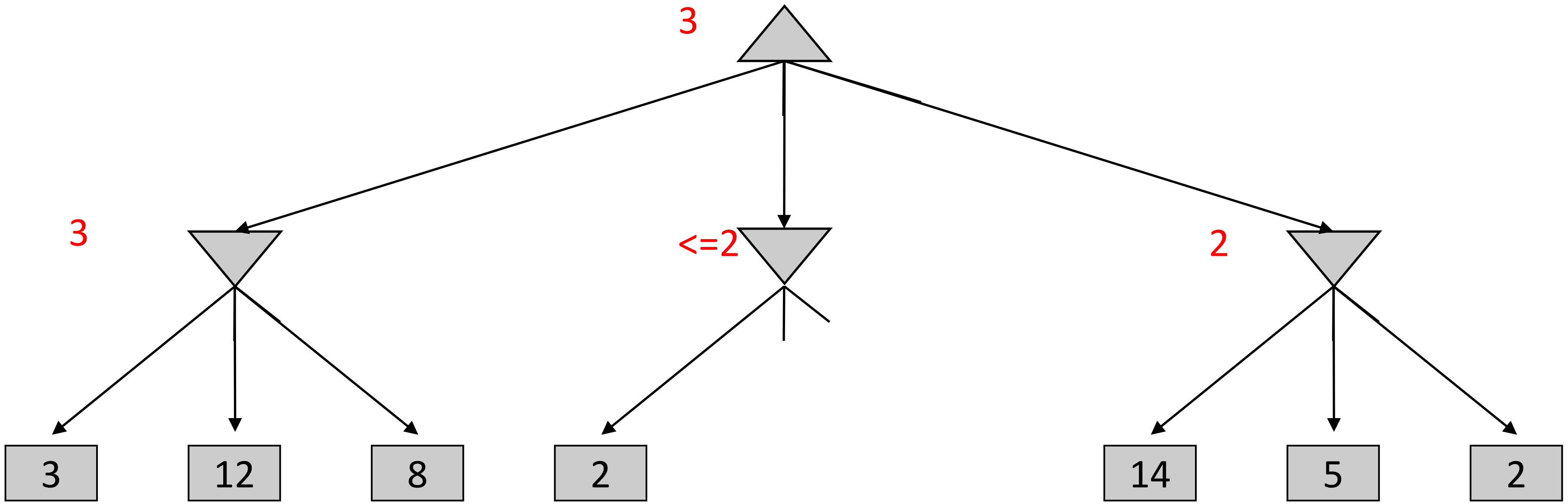
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
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
```

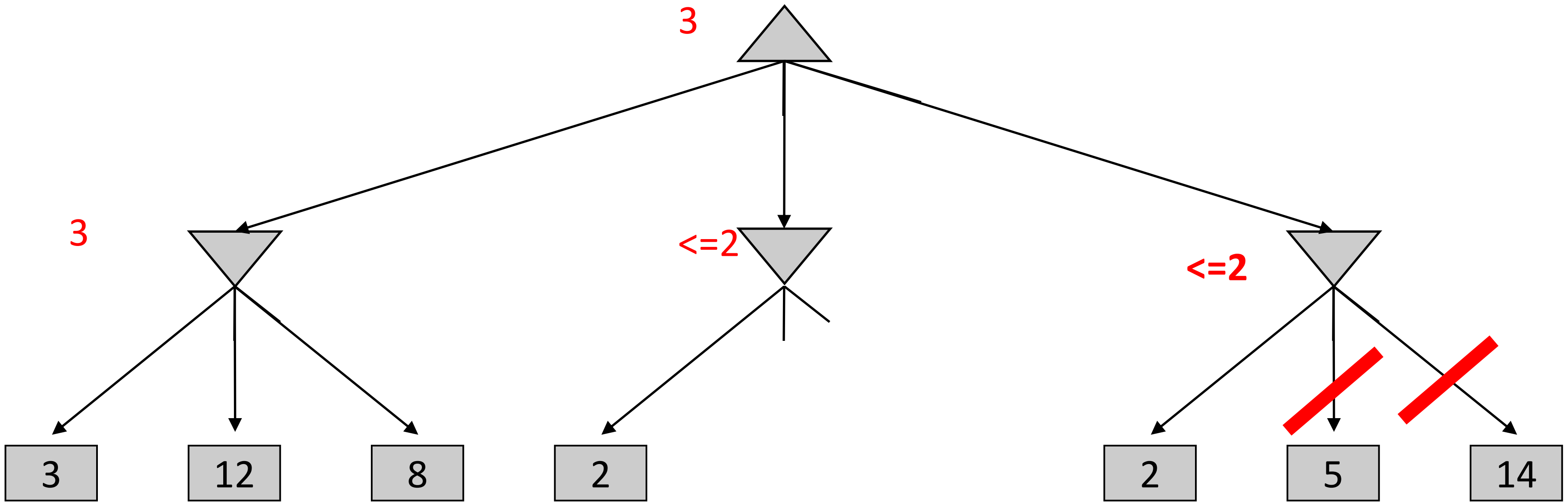
Alpha-Beta Pruning - Properties

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

- Bad: Worst moves encountered first

```
                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
```

- Good: Good moves ordered first

```
                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
```

Ordering moves with good moves first can benefit alpha-beta pruning.

Slide adapted from Prof. Mausam

Alpha-Beta Pruning – $O(b^{m/2})$

Let $T(m)$ be time complexity of search for depth m

Normally:

$T(m) = b.T(m-1) + c \rightarrow T(m) = O(b^m)$

With ideal α-β pruning:

$T(m) = T(m-1) + (b-1)T(m-2) + c \rightarrow T(m) = O(b^{m/2})$

We are cutting off the branching at every other level.

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.
   - 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

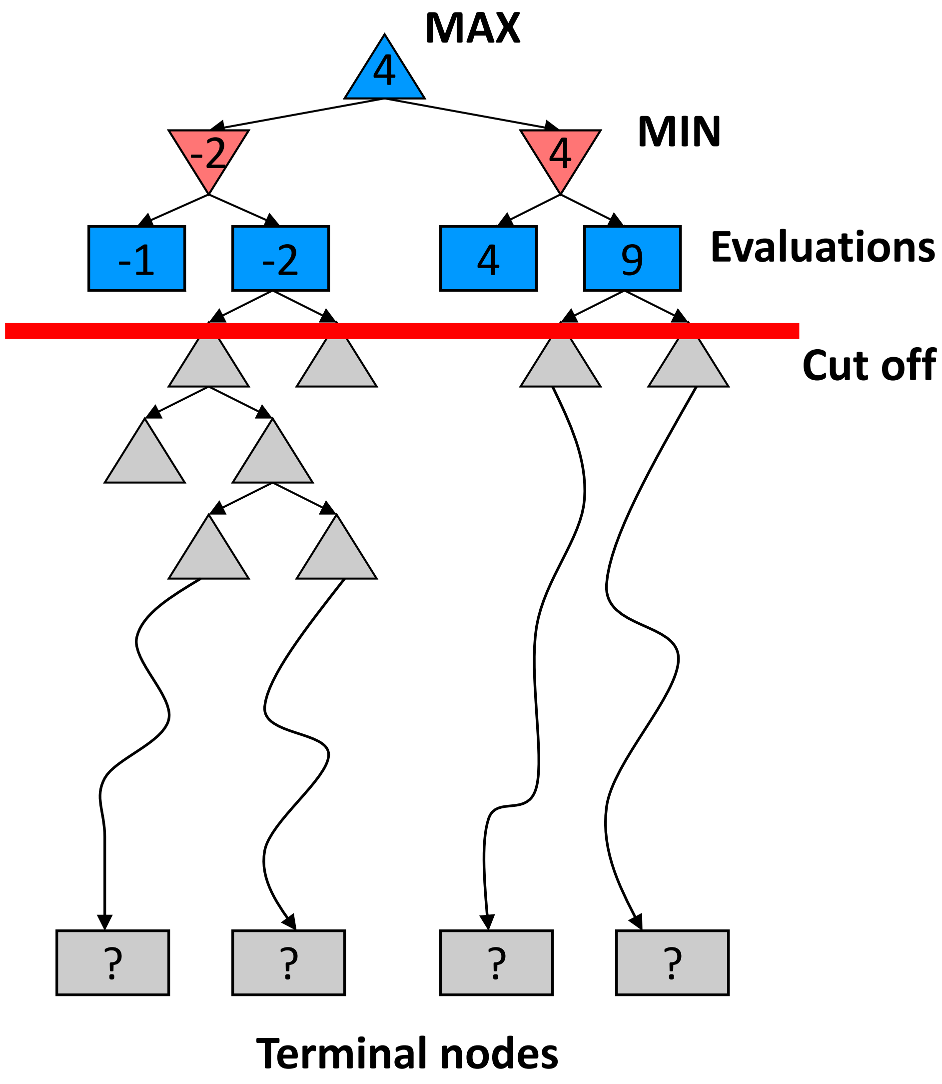
- Chess:
  - branching factor $b \approx 35$
  - game length $m \approx 100$
  - search space $b^{m/2} \approx 35^{50} \approx 10^{77}$

Slide adapted from Prof. Mausam

Cutting-off Search

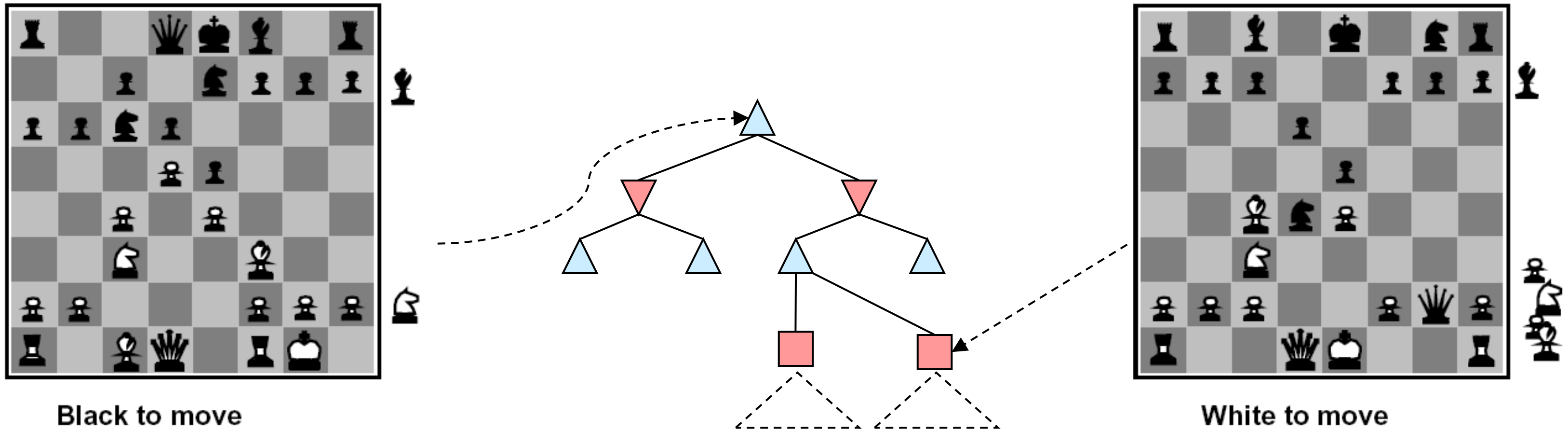
- Problem:
  - Minimax search: full tree till the terminal nodes.
  - Alpha-beta prunes the tree but still searches till the terminal nodes.
  - Still difficult to search till the leaves.
- Solution:
  - Depth-limited Search (H-Minimax)
  - Search only to a limited depth (cutoff) in the tree
  - Replace the terminal utilities with an evaluation function for non-terminal positions

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



Evaluation Functions

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



Black to move

White slightly better

White to move

Black 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.

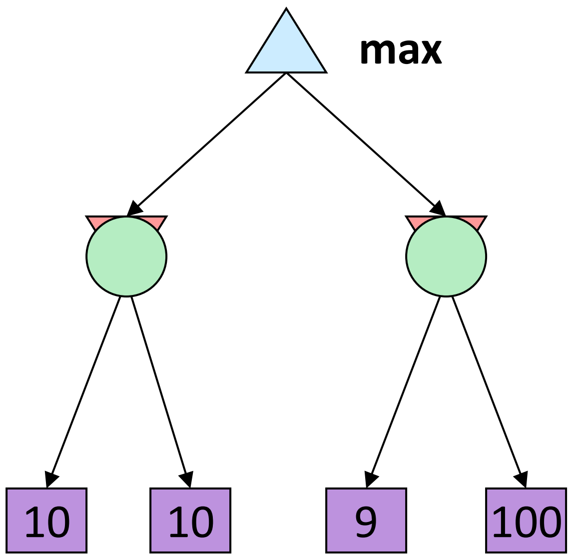
Incorporating Chance: Expectimax Search

- Till now
  - Assumed that the opponent provides us with the *worst-case* outcome.

- Incorporate a notion of chance
  - Include chance nodes
    - Unpredictable opponents: the ghosts move randomly in Pacman
    - Explicit randomness: rolling dice

- Computing values at nodes
  - Not worst-case (minimax) outcomes
  - Reflect average-case (expectimax) outcomes

- **Expectimax search:**
  - Compute the average score under optimal play
  - Max nodes as in minimax search
  - At chance nodes the outcome is uncertain
  - Calculate ***expected utilities:*** weighted average (expectation) of children



Expectimax Search

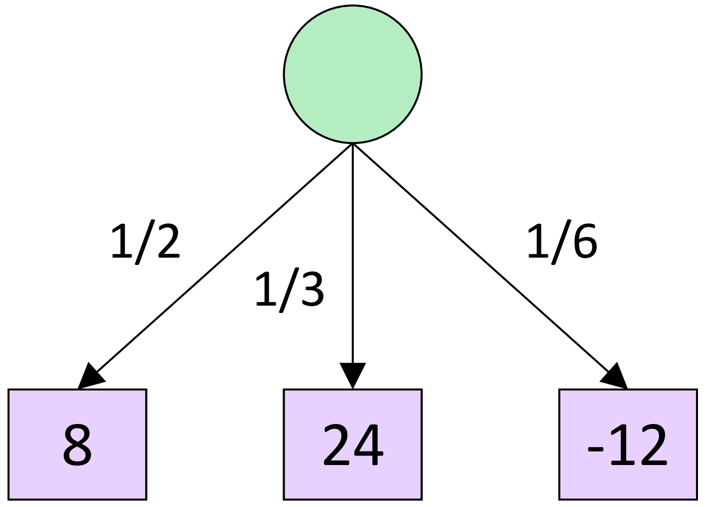
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
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
```

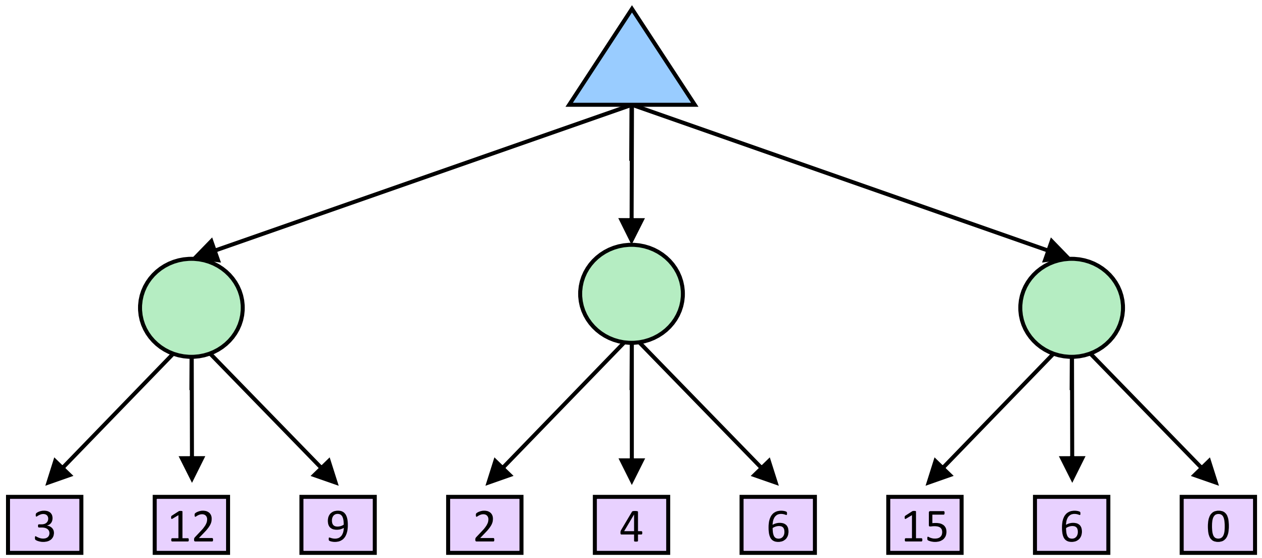
Expectimax Search

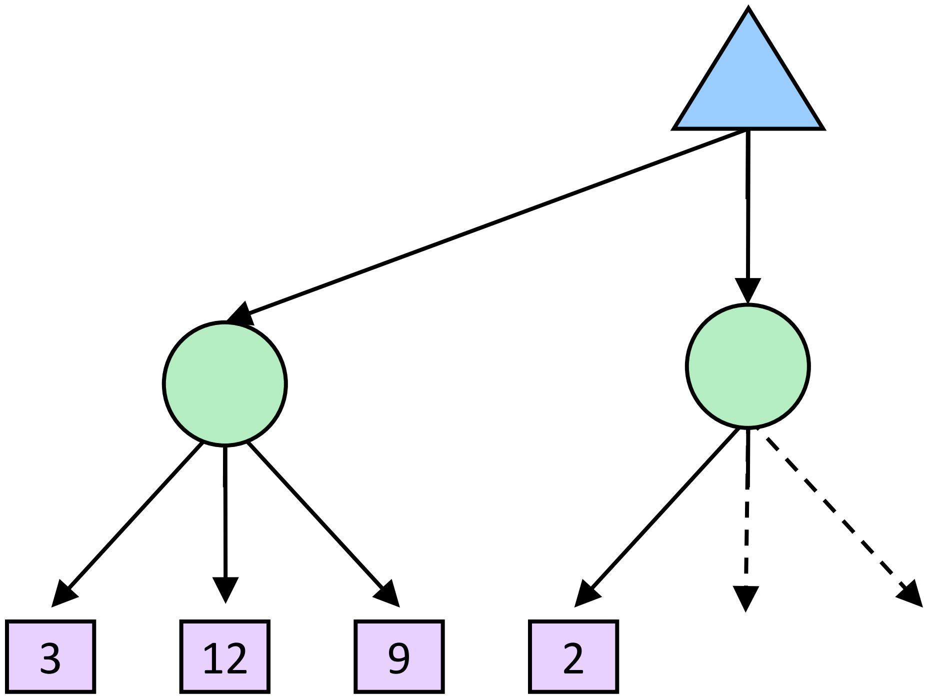
```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```



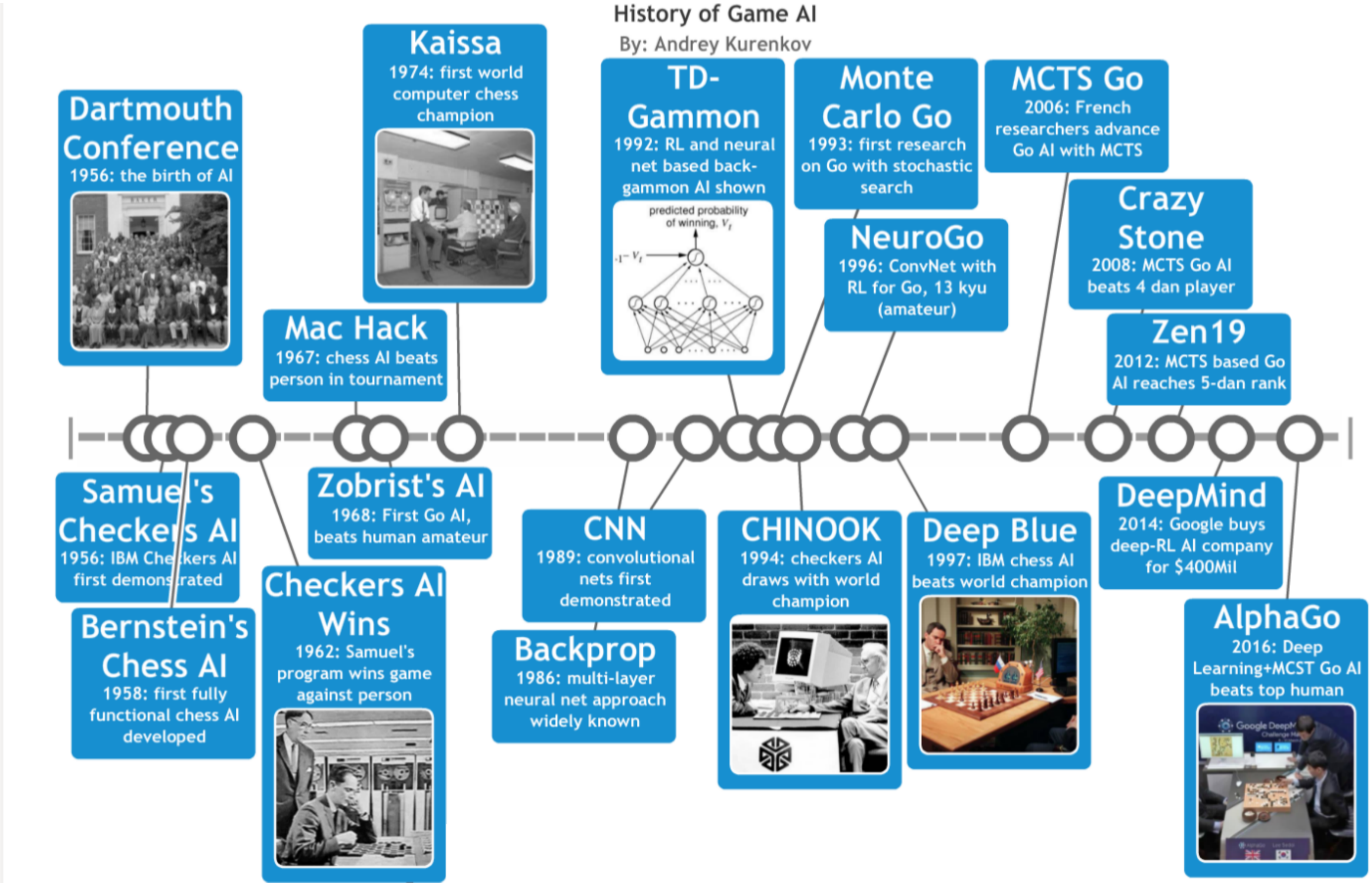
$$v = (1/2)\ (8) + (1/3)\ (24) + (1/6)\ (-12) = 10$$

Expectimax Search





Can we perform pruning?



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