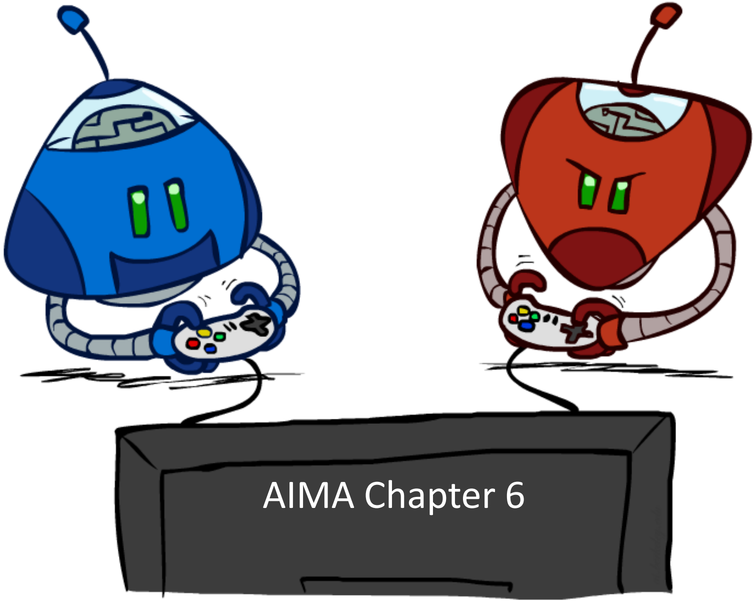
Announcement

- Programming Assignment 1B: adversarial search
  - Instructions at Blackboard -> “Programming Assignments”
  - Submission at AutoLab
  - Due: Mar 10, 11:59pm

Adversarial Search



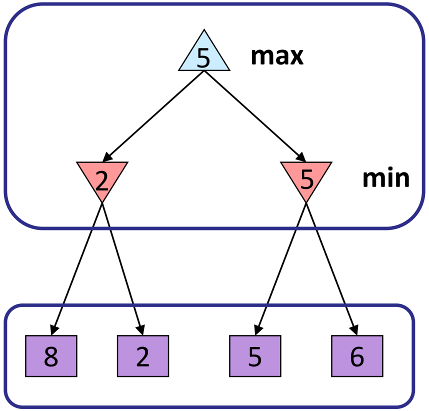
[Adapted from slides by Dan Klein and Pieter Abbeel at UC Berkeley (ai.berkeley.edu).]

Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - Players alternate turns
  - One player maximizes result
  - The other minimizes result

- Minimax search:
  - A state-space search tree
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary

Minimax values: computed recursively



Terminal values: part of the game

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)
```

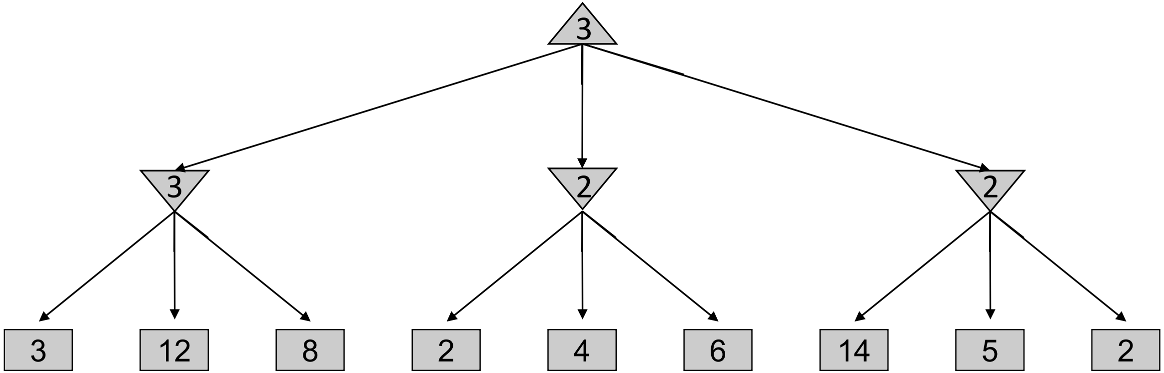
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

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

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

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

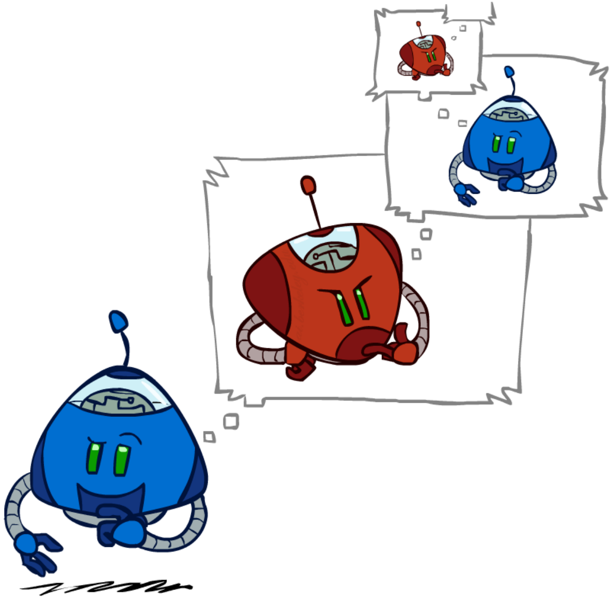
Minimax Example



Minimax Efficiency

- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$

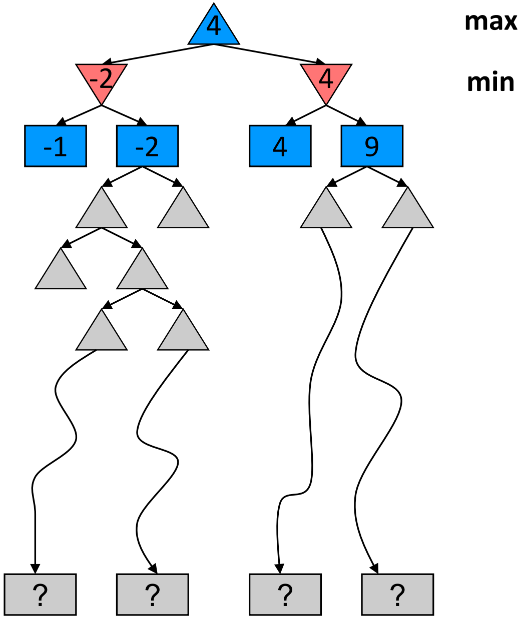
- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?



Resource Limits

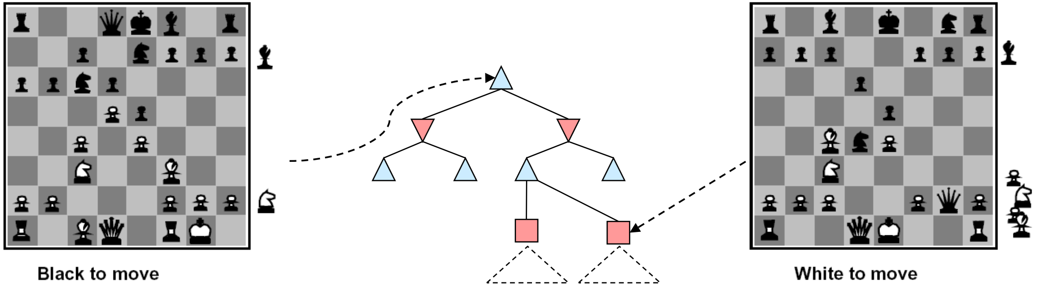
Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - α-β reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- More depth makes a BIG difference



Evaluation Functions

- Evaluation functions score non-terminals in depth-limited search



Black to move

White slightly better

White to move

Black winning

- Ideal function: returns the actual minimax value of the position
- A simple solution in practice: weighted linear sum of features:

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

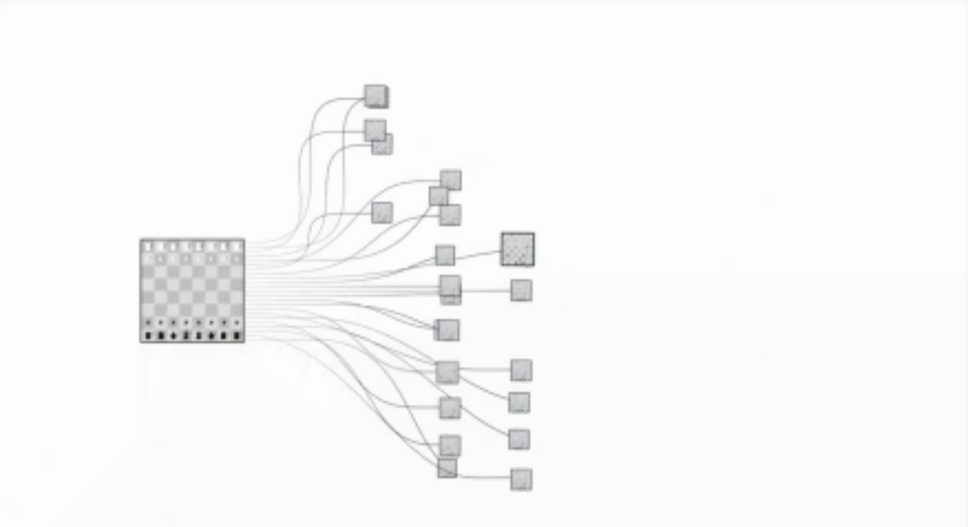
- e.g. $f_1(s)$ = (num white queens – num black queens), etc.

Evaluation Functions

- Recent advances
  - Monte Carlo Tree Search
    - Randomly choose moves until the end of game
    - Repeat for many many times
    - Evaluate the state based on these simulations, e.g., the winning rate
  - Convolutional Neural Network (value network in AlphaGo)
    - Trained from records of game plays to predict a score of the state

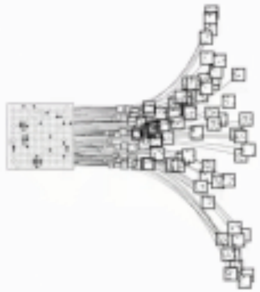
Branching Factor

- Chess



Branching Factor

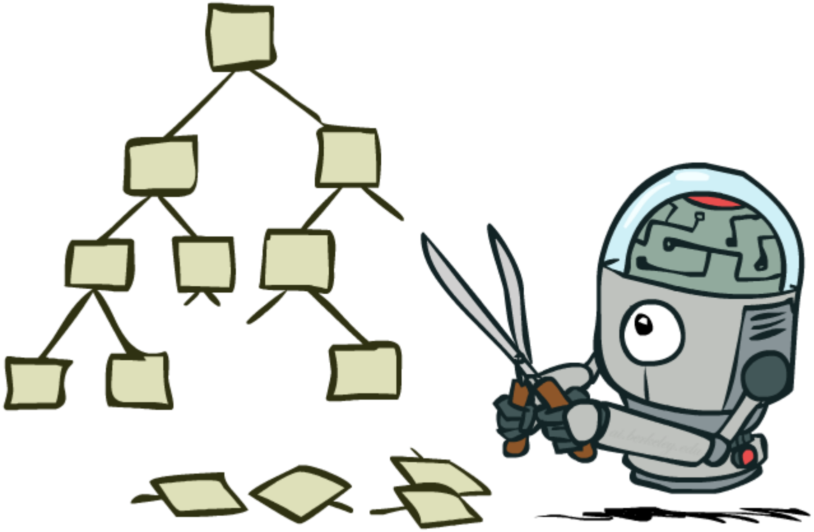
- Go



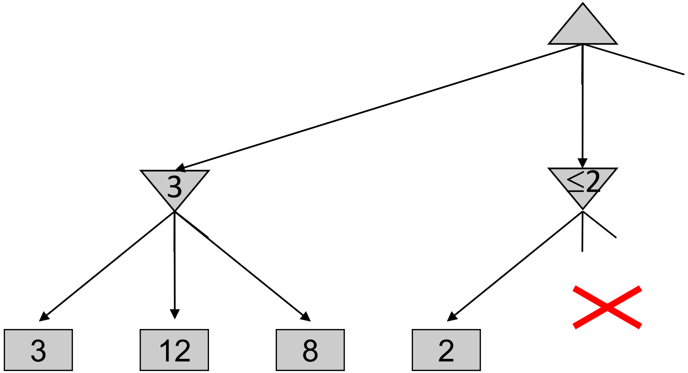
Branching Factor

- Go has a branching factor of up to 361
- Idea: limit the branching factor by considering only good moves
  - AlphaGo uses a Convolutional Neural Network (policy network)
    - Trained from records of game plays
    - Trained using reinforcement learning
      - AlphaGo Zero uses RL only

Game Tree Pruning

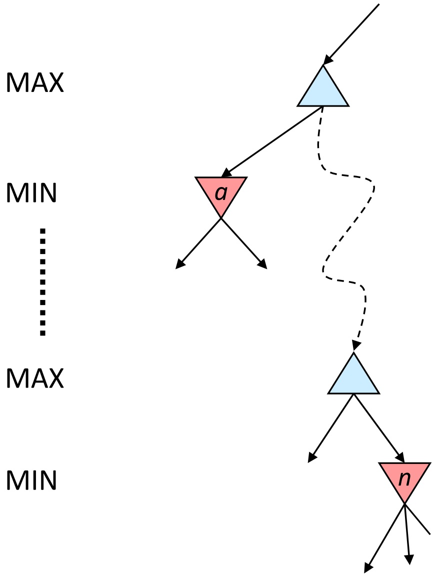


Minimax Pruning



Alpha-Beta Pruning

- General configuration (MIN version)
  - We're computing the MIN-VALUE at some node *n*
  - We're looping over *n*'s children, so *n*'s estimate is decreasing
  - Let *a* be the best value that MAX can get at any choice point along the current path from the root
  - If *n* becomes worse than *a*, then we can stop considering *n*'s other children
  - Reason: if *n* is eventually chosen, then the nodes along the path shall all have the value of *n*, but *n* is worse than *a* and hence the path shall not be chosen at the MAX



- MAX version is symmetric

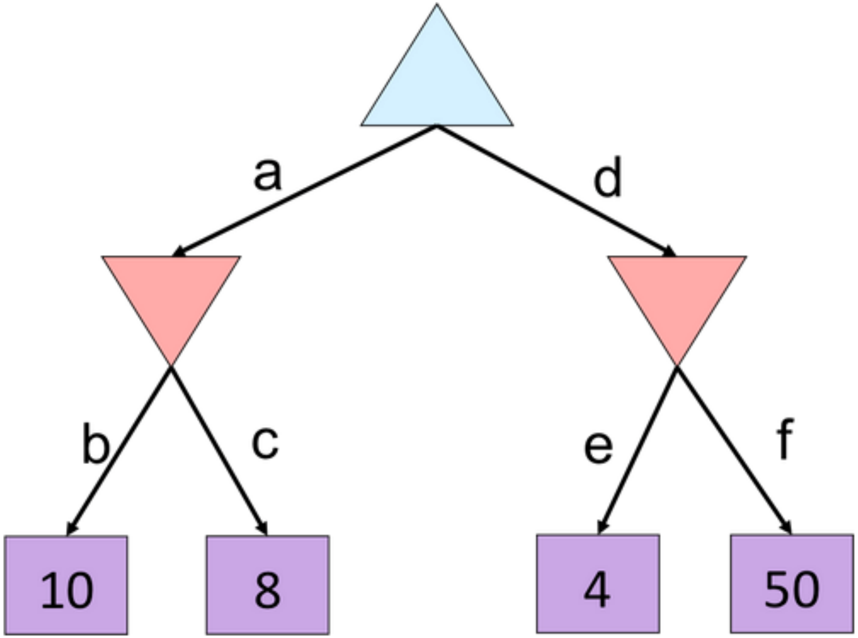
Alpha-Beta Implementation

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

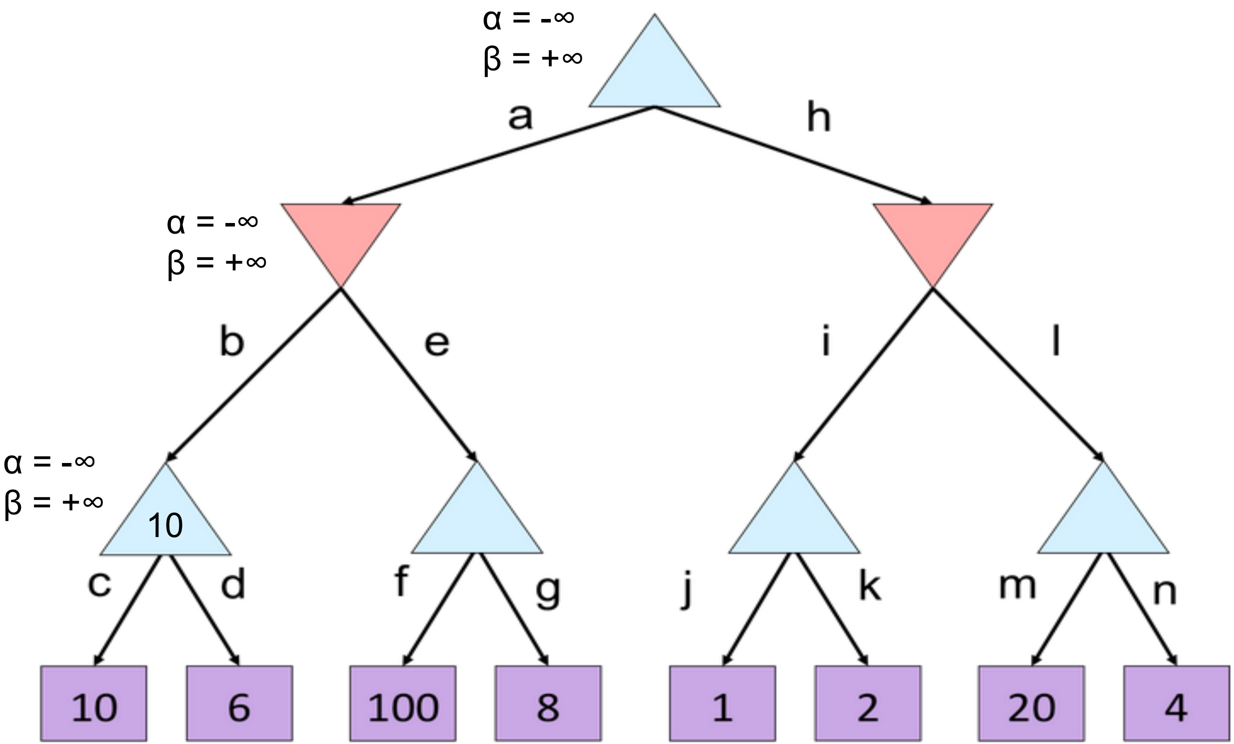
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
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
```

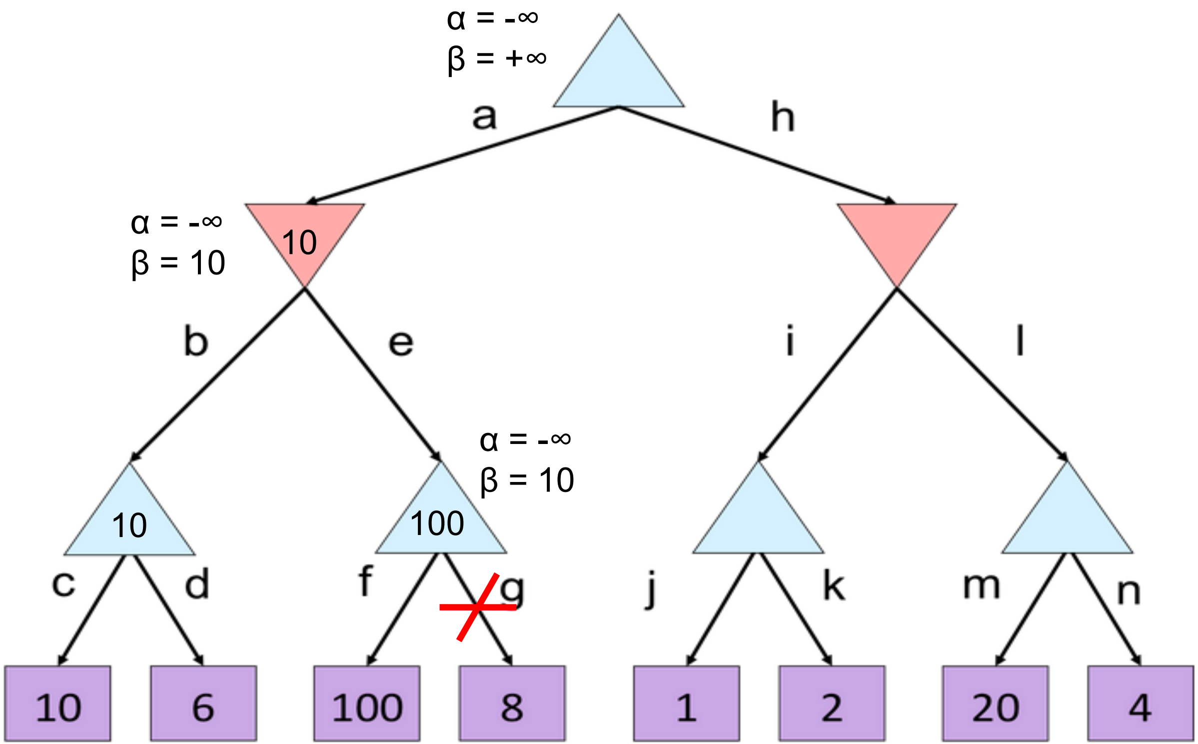
Alpha-Beta Example



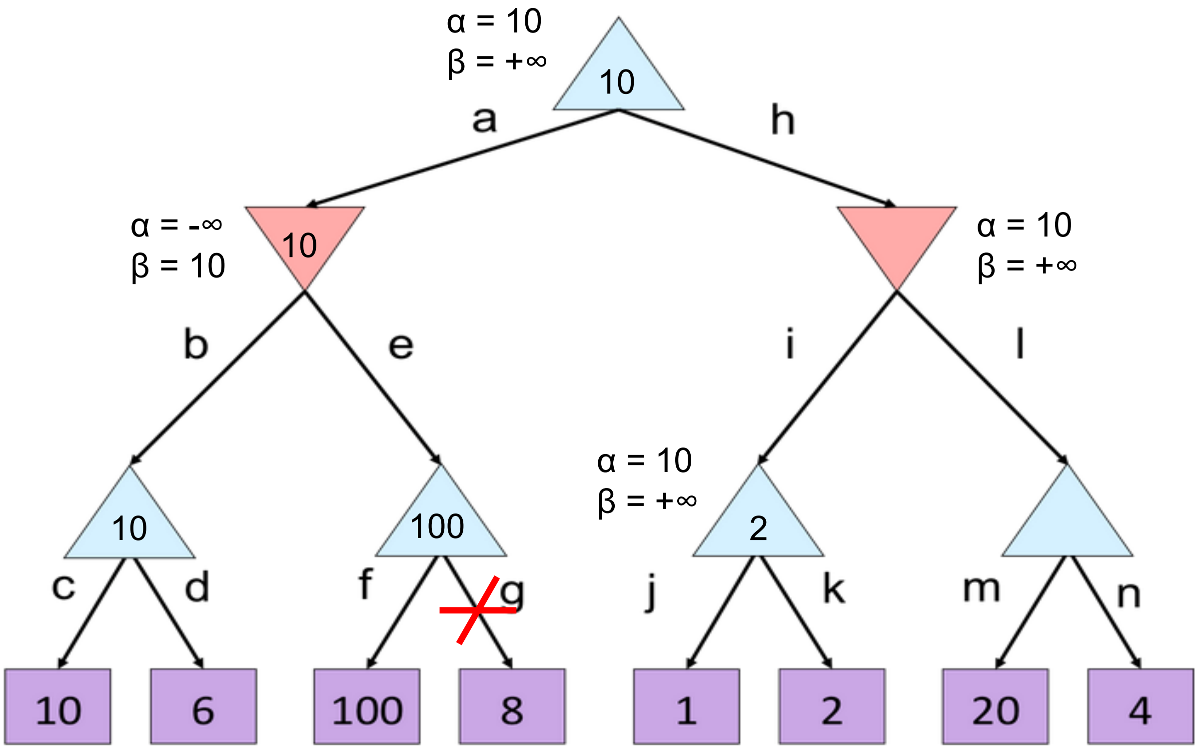
Alpha-Beta Example 2



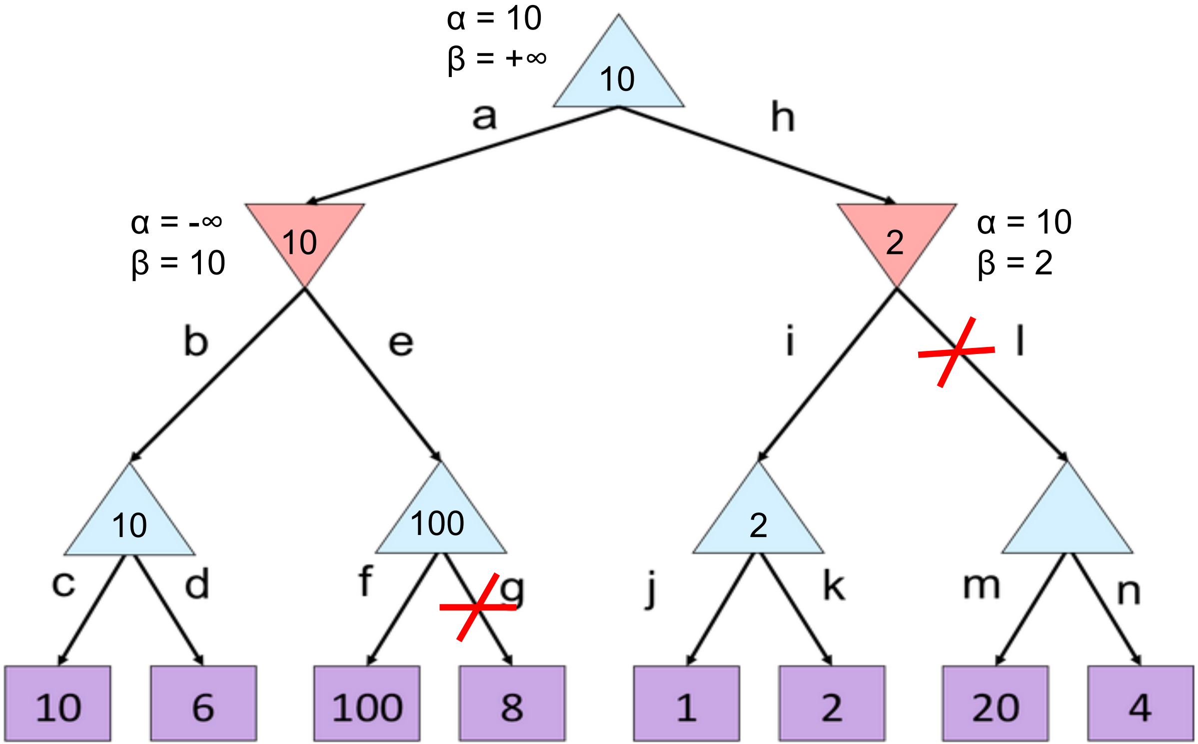
Alpha-Beta Example 2



Alpha-Beta Example 2



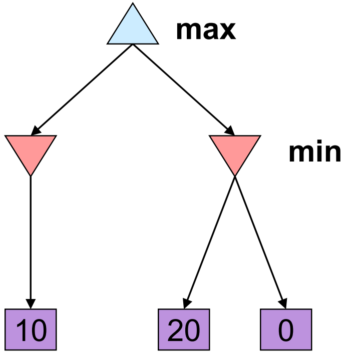
Alpha-Beta Example 2



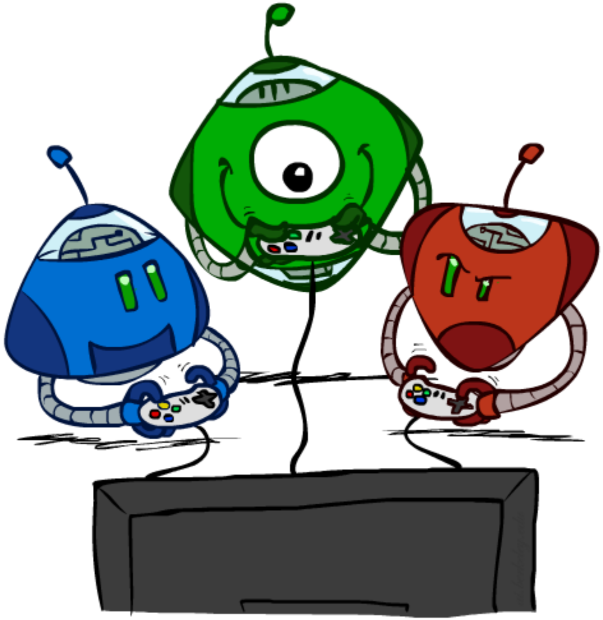
Alpha-Beta Pruning Properties

- Good child ordering improves effectiveness of pruning

- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!



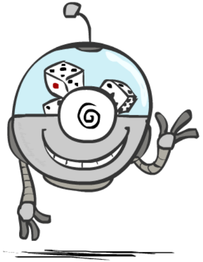
Other Game Types

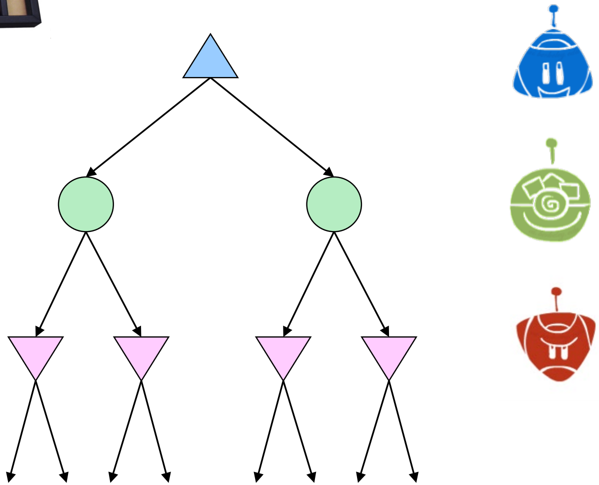


Mixed Layer Types

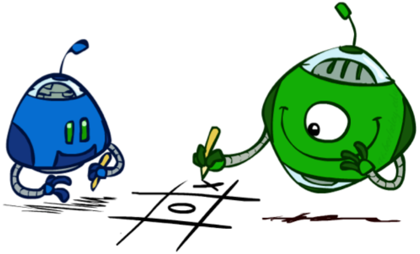
- Backgammon

- Expectiminimax
  - Environment is an extra "random agent" player that moves after each min/max agent
  - Each node computes the appropriate combination of its children



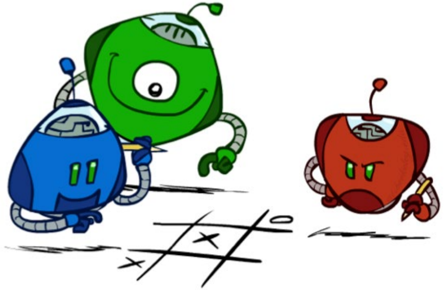


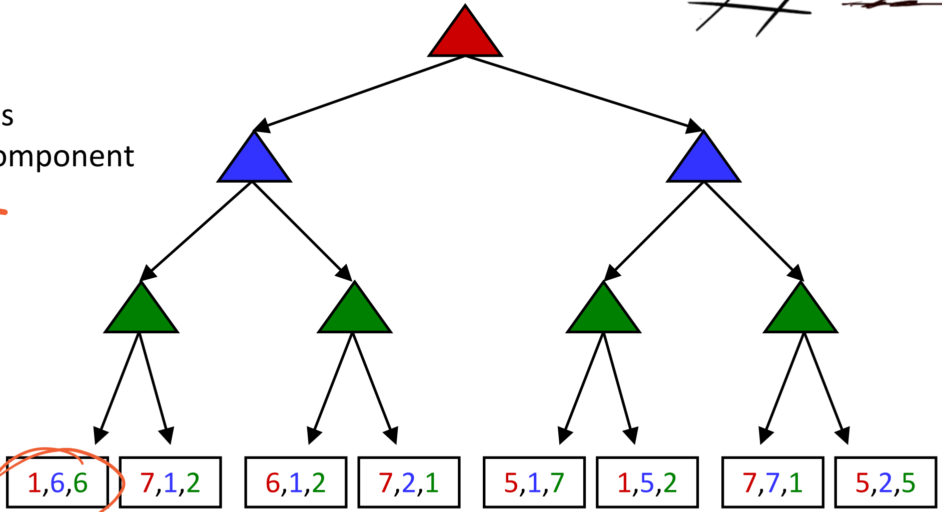
Multi-Agent Utilities



- What if the game is not zero-sum, or has multiple players?

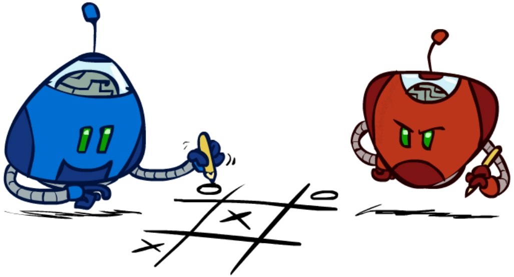
- Generalization of minimax:
  - Terminals have utility tuples
  - Node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to cooperation and competition dynamically…





tuple.

Summary



- Adversarial Games
- Adversarial Search
  - Minimax
- Resource Limits
  - Depth-limited search, limiting branching factor
- Game Tree Pruning (alpha-beta pruning)
- Uncertain Outcomes
  - Expectimax
- Other Game Types