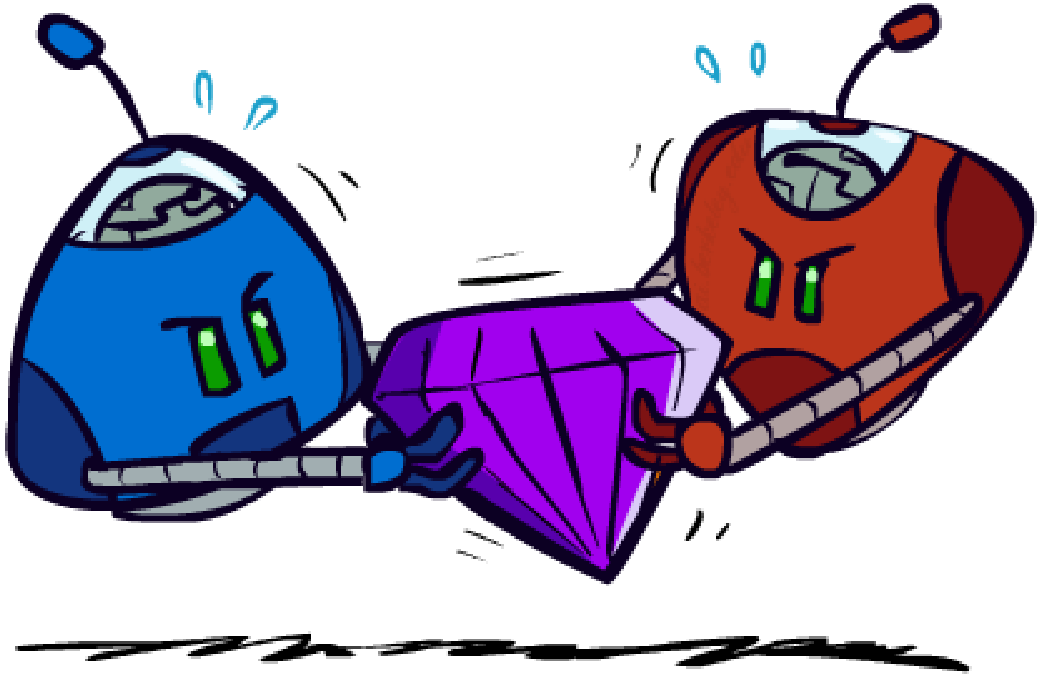
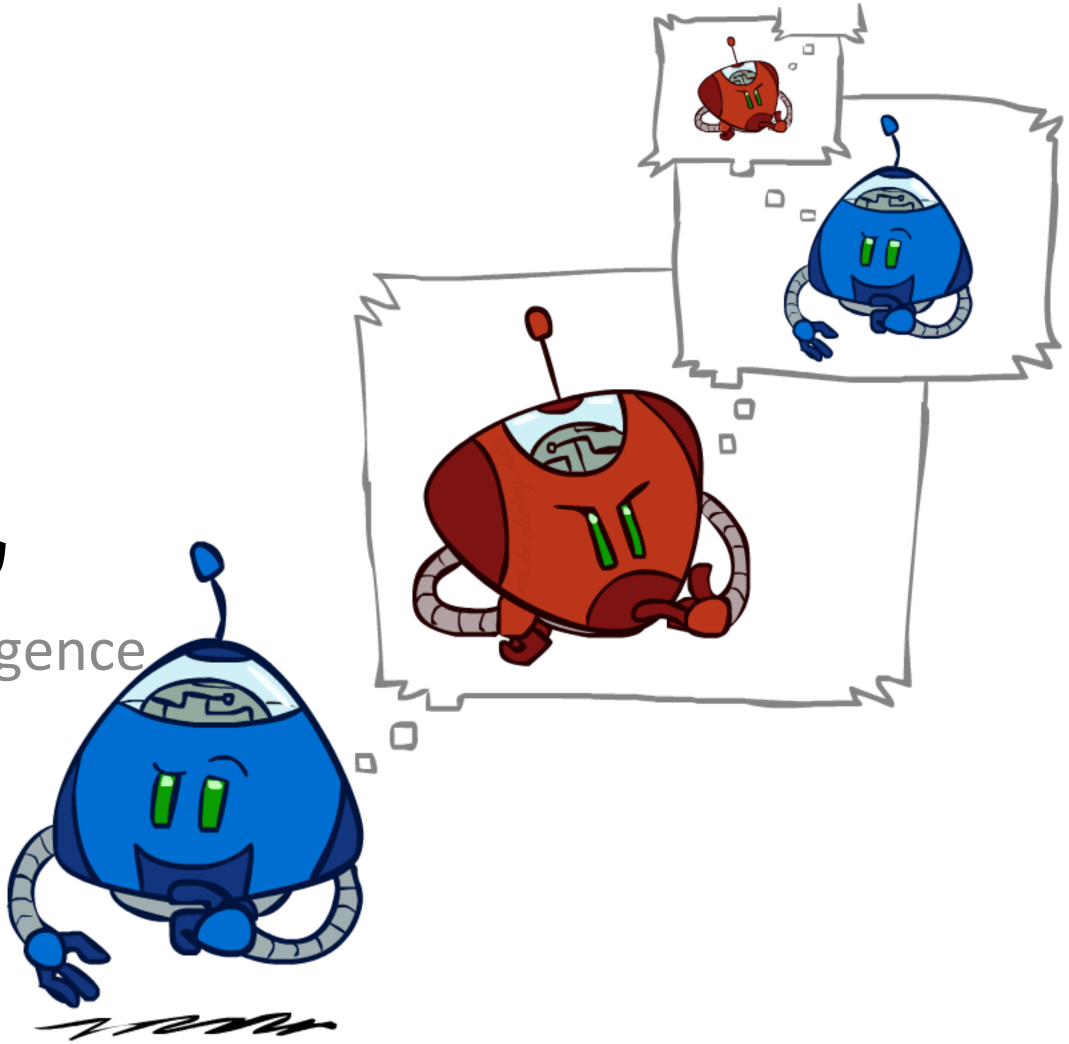
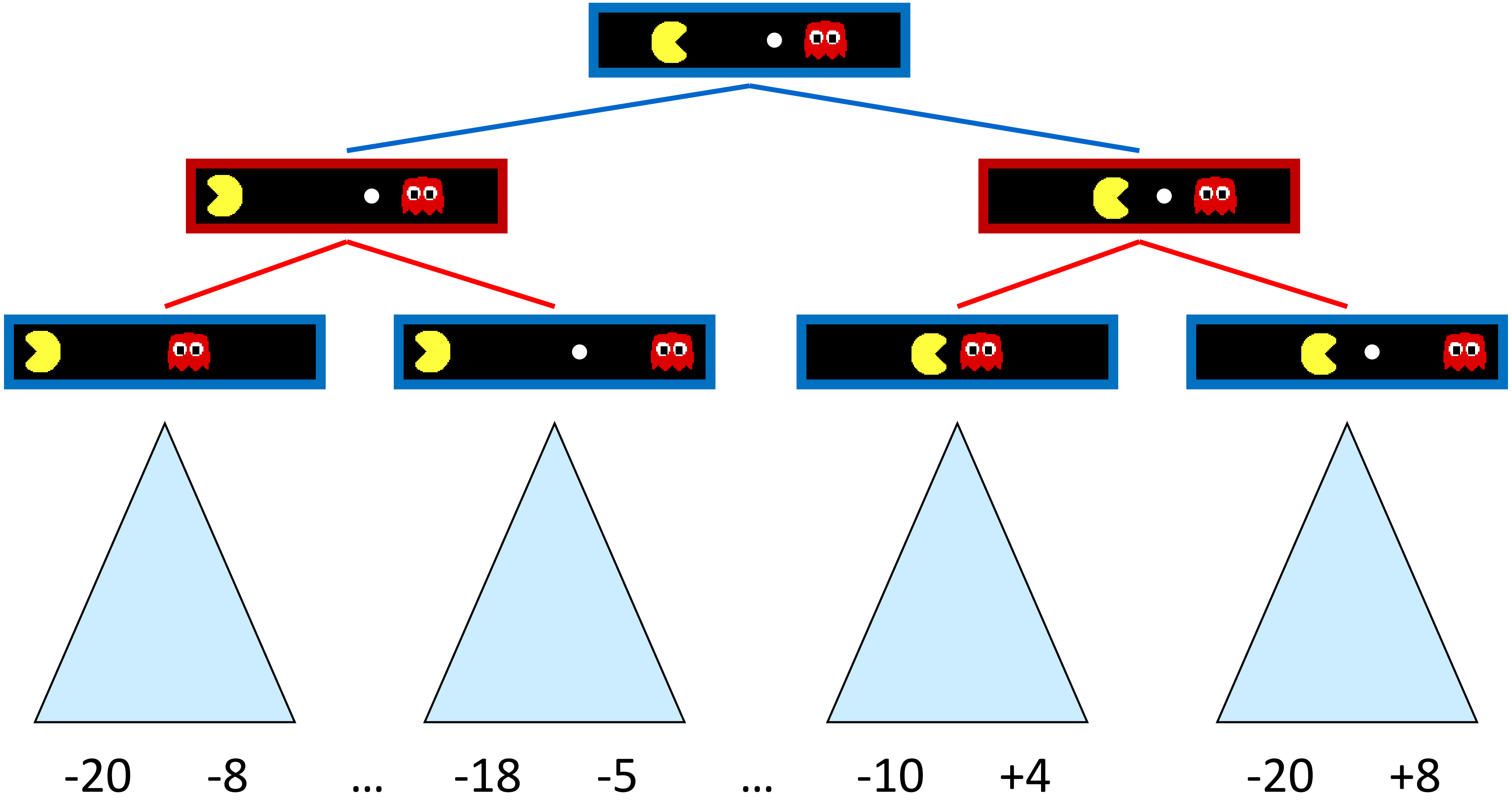


对抗搜索及其优化

Berkeley CS 188 | Introduction to Artificial Intelligence



Adversarial Game Trees



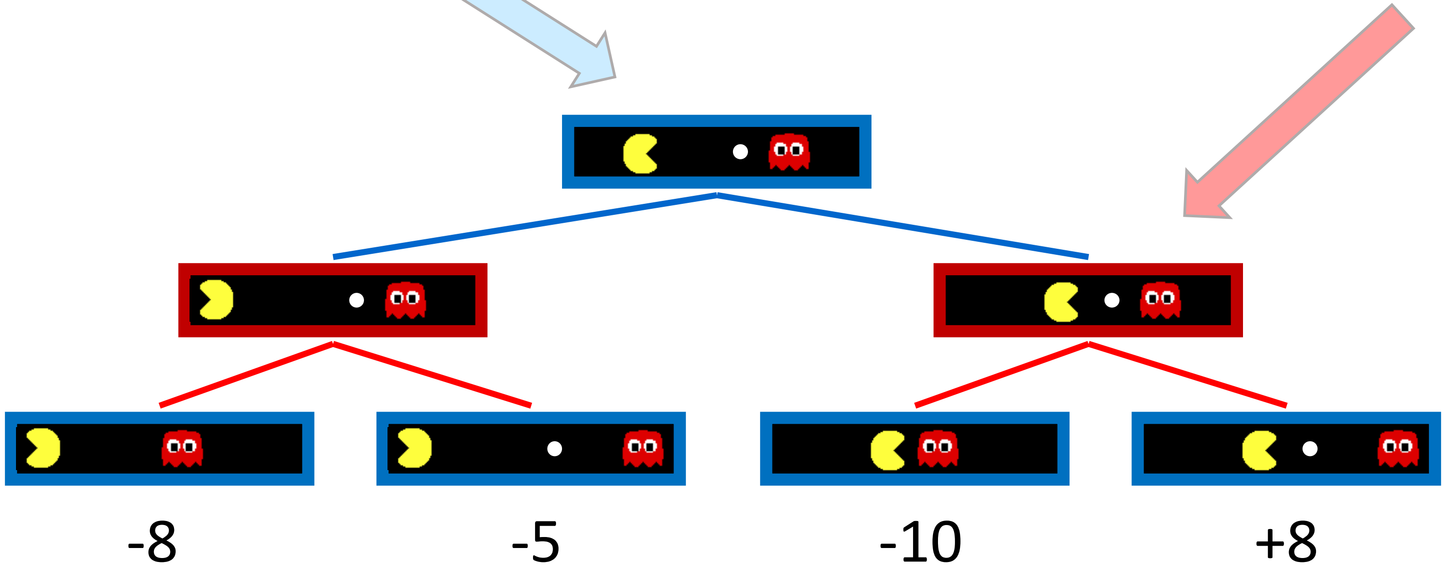
Adversarial Game Trees: Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

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

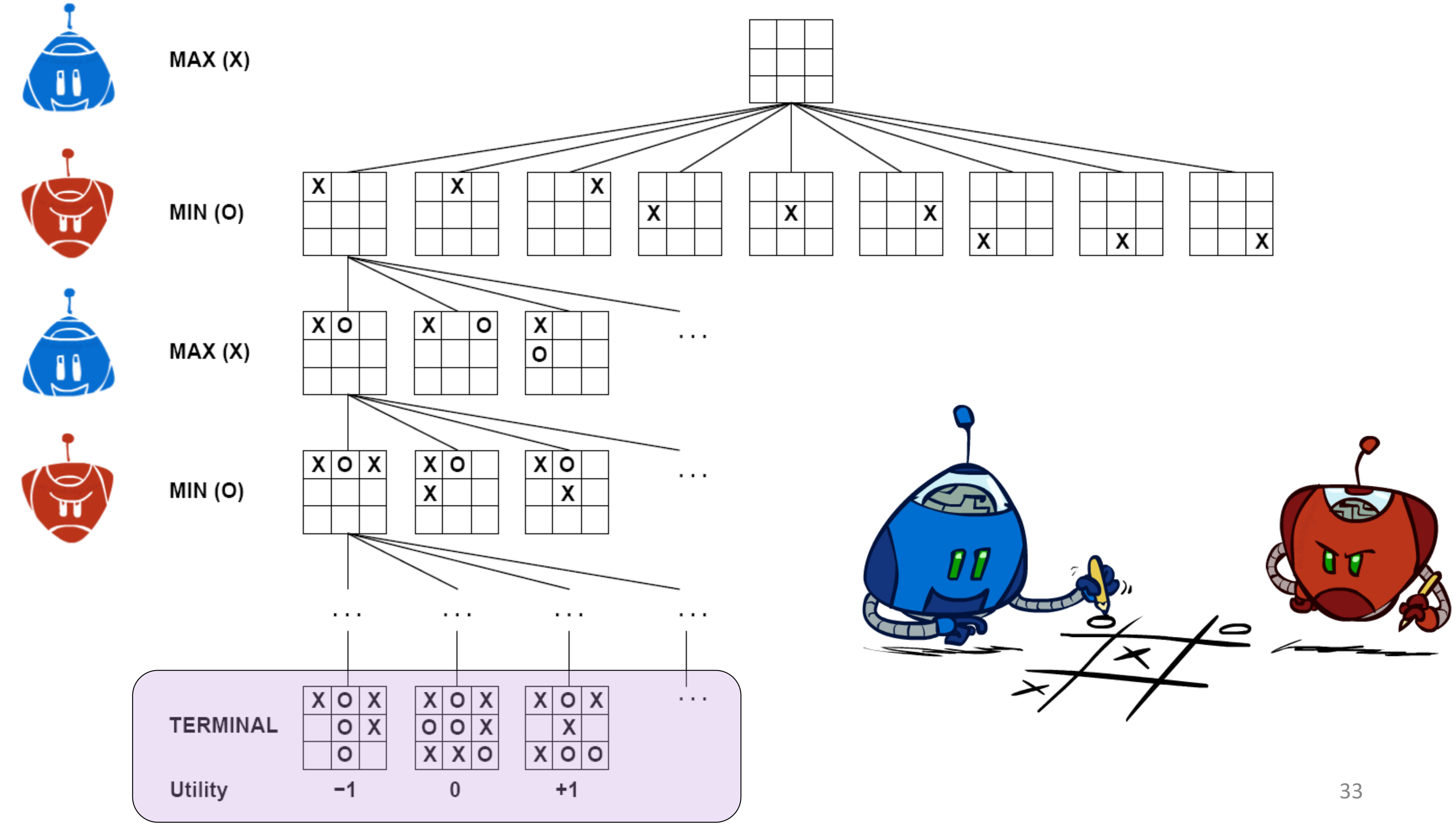


Terminal States:

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

Example: Tic-Tac-Toe Game Tree

- States
- Actions
- Values

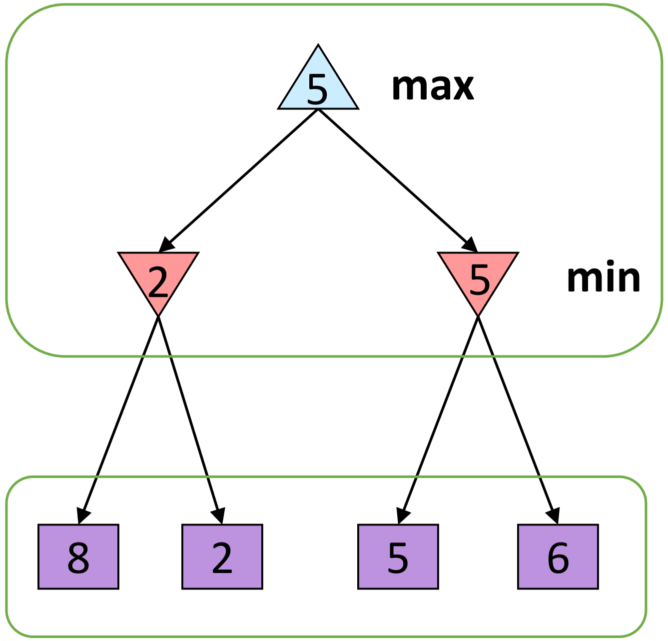


Minimax Search

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

- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - 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 (Dispatch)

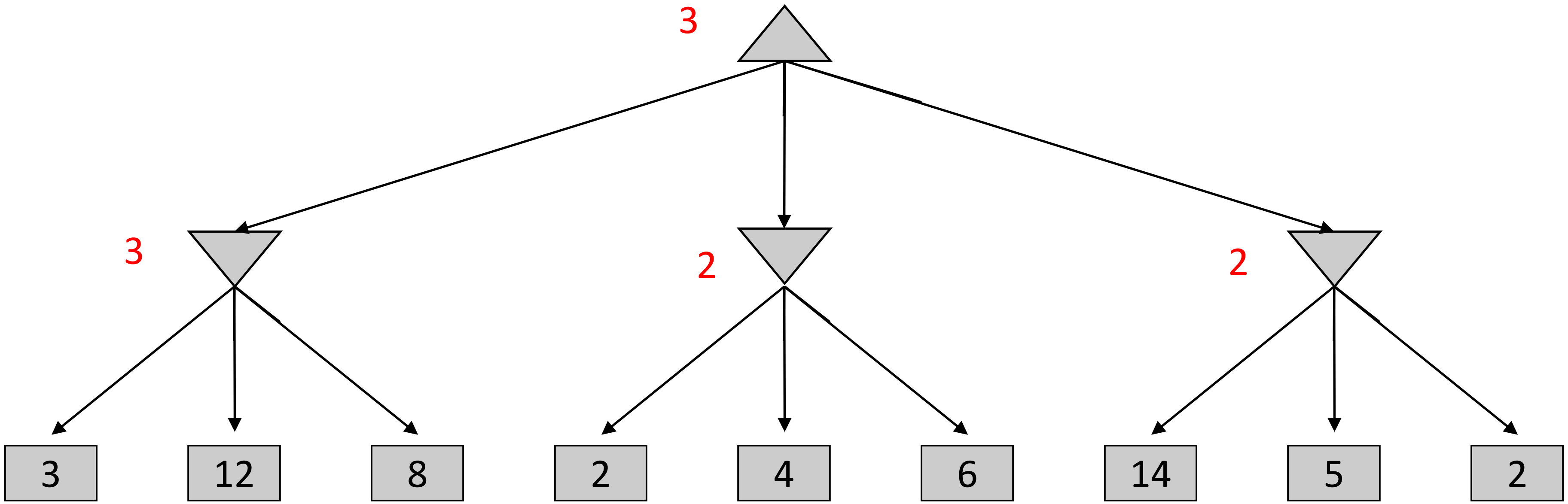
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
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
```

Example

- Actions?



Pseudocode for Generic Game Tree

```
function  minimax_decision( state )
        return  argmax a in state.actions  value( state.result(a) )

function  value( state )
    if  state.is_leaf
        return state.value

    if  state.player  is  MAX
        return  max a in state.actions  value( state.result(a) )

    if  state.player  is  MIN
        return  min a in state.actions  value( state.result(a) )
```

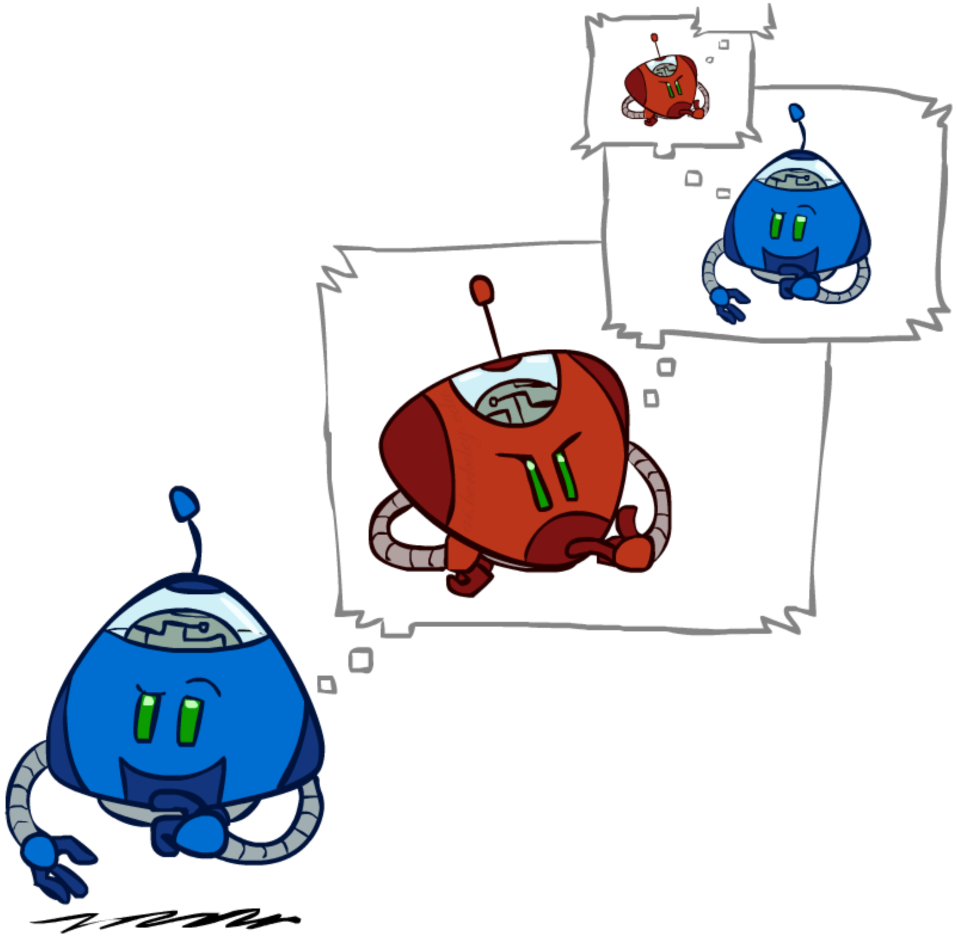
Quiz

- Minimax search belongs to which class?

A) BFS
B) DFS
C) UCS
D) A*

Minimax Efficiency

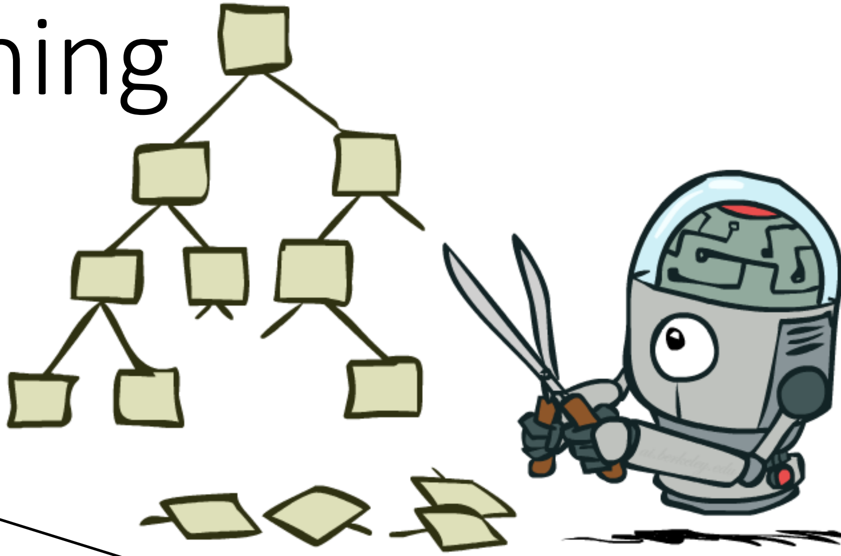
- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$

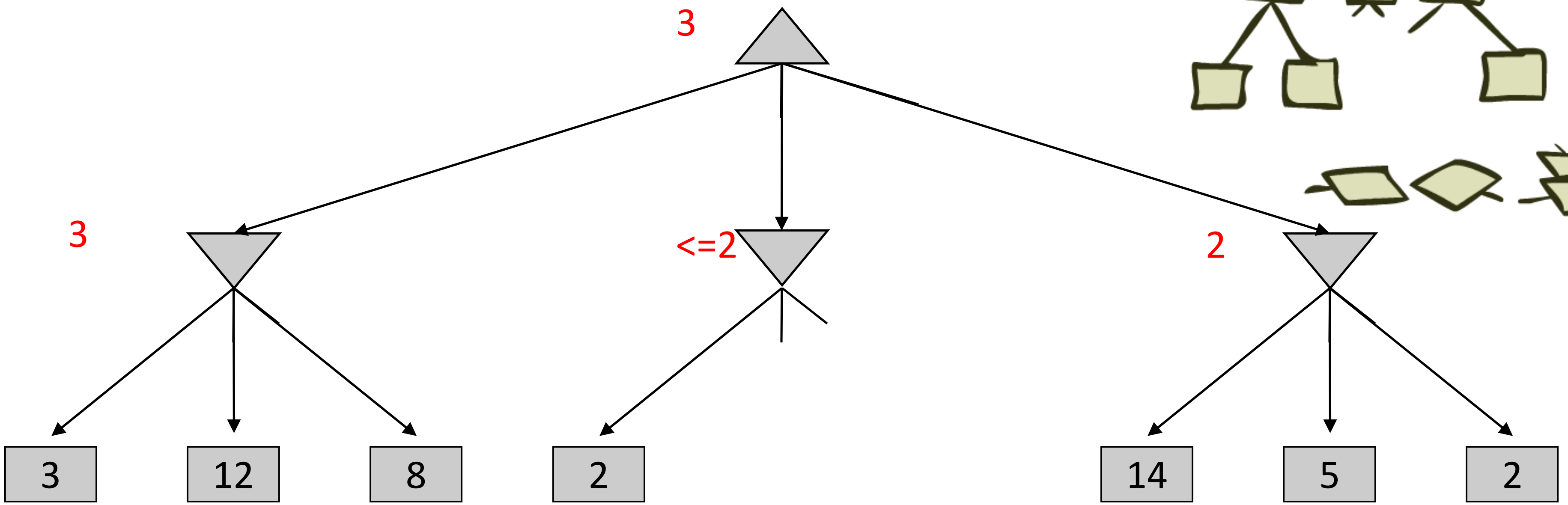


- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?
  - Humans can't do this either, so how do we play chess?
  - Bounded rationality – Herbert Simon

Resource Limits

Resource Limits: Game Tree Pruning



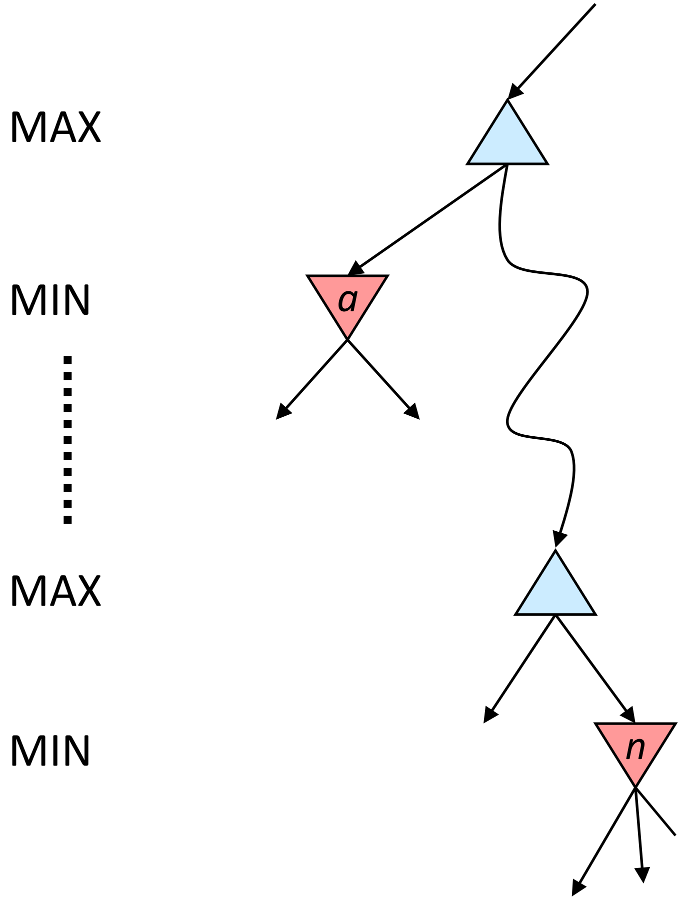


The order of generation matters: more pruning is possible if good moves come first

Game Tree 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
  - *n*'s estimate of the childrens' min is dropping
  - Who cares about *n*'s value? MAX
  - 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*, MAX will avoid it, so we can stop considering *n*'s other children (it's already bad enough that it won't be played)

- MAX version is symmetric



Alpha-Beta Implementation

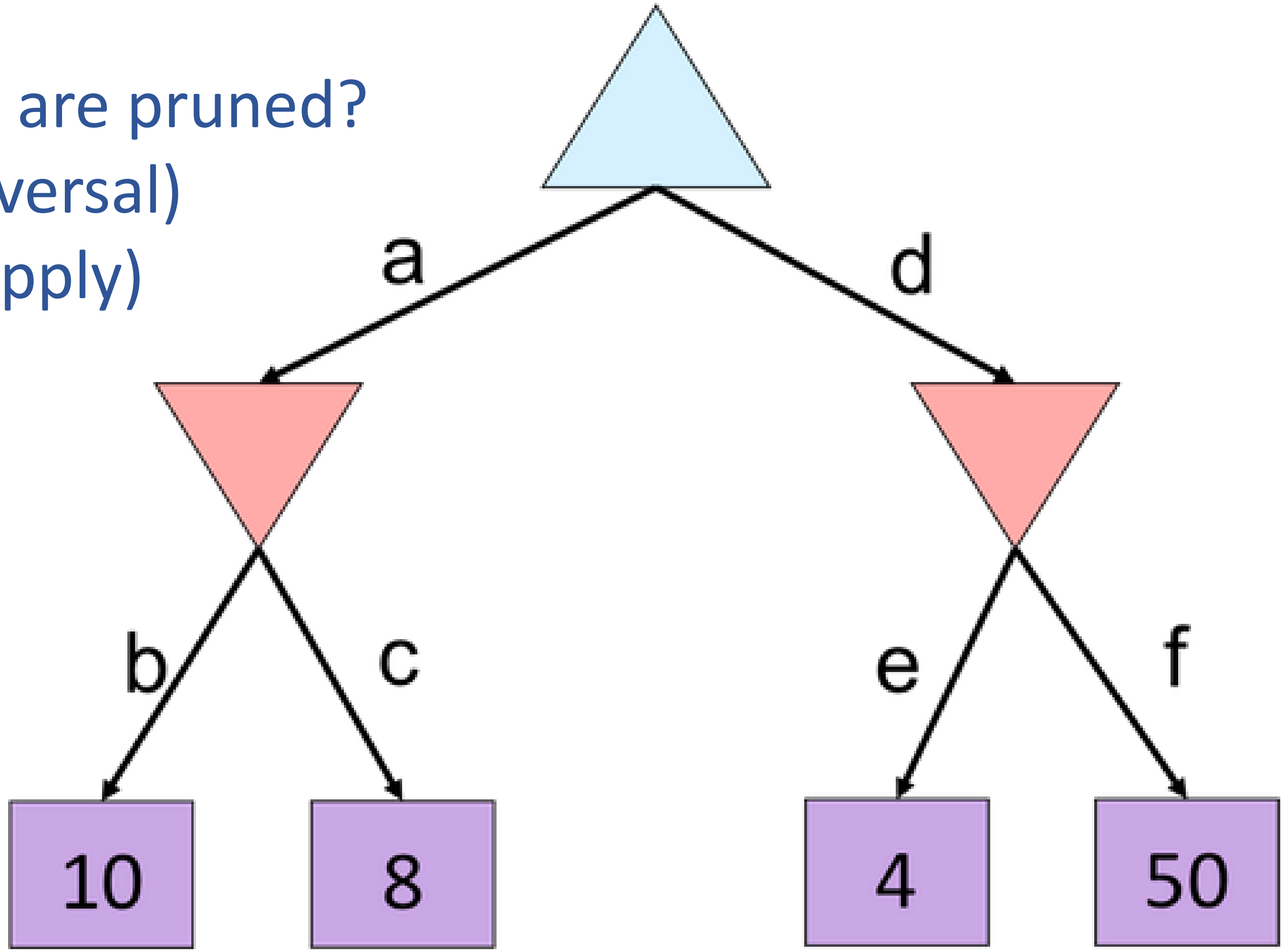
α: MAX's best option on path to root
β: MIN's best option on path to root

```
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
```

Quiz

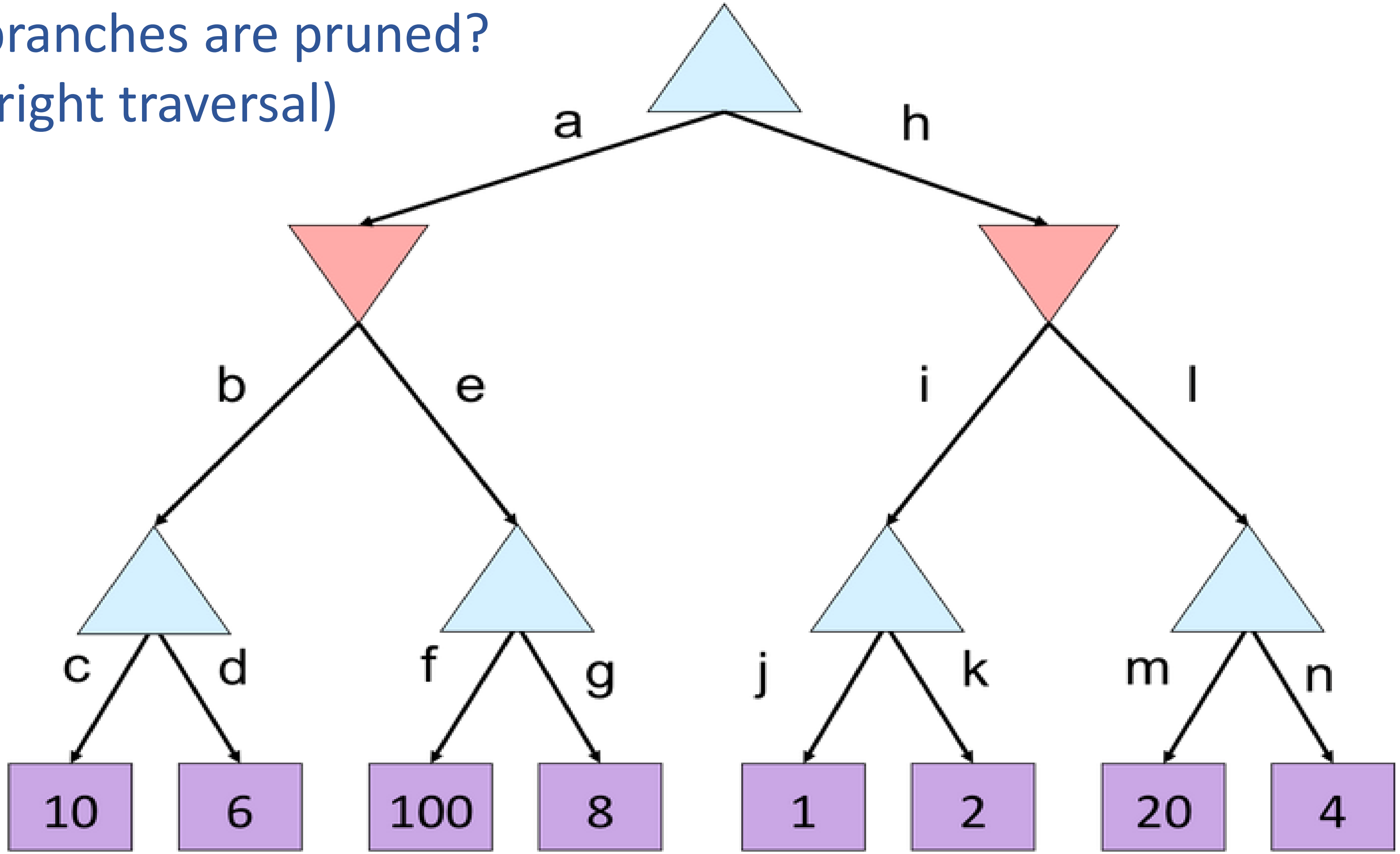
Which branches are pruned?
(Left to right traversal)
(Select all that apply)



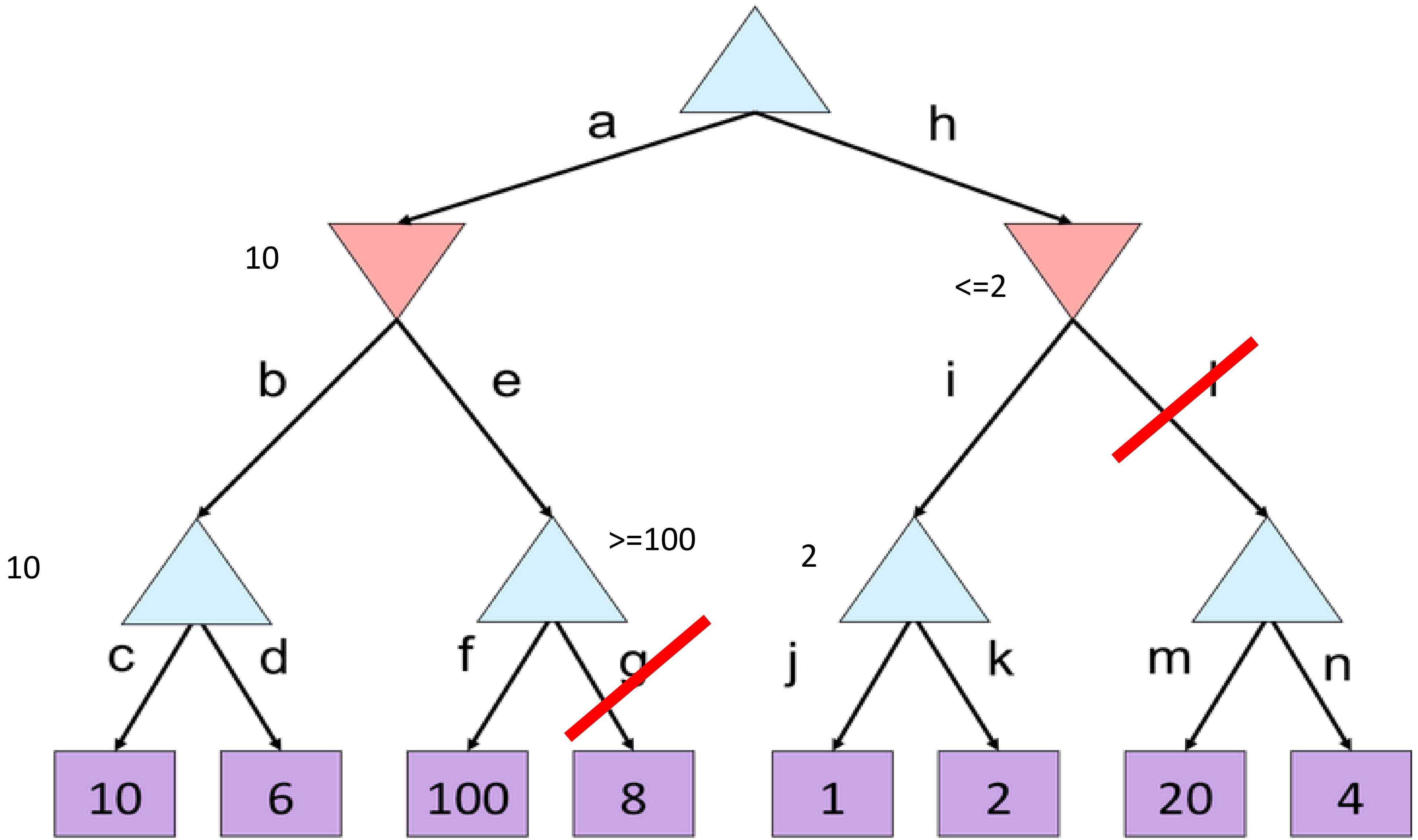
Quiz 2

Which branches are pruned?
(Left to right traversal)

A) e, l
B) g, l
C) g, k, l
D) g, n



Quiz 2



Alpha-Beta Pruning Properties

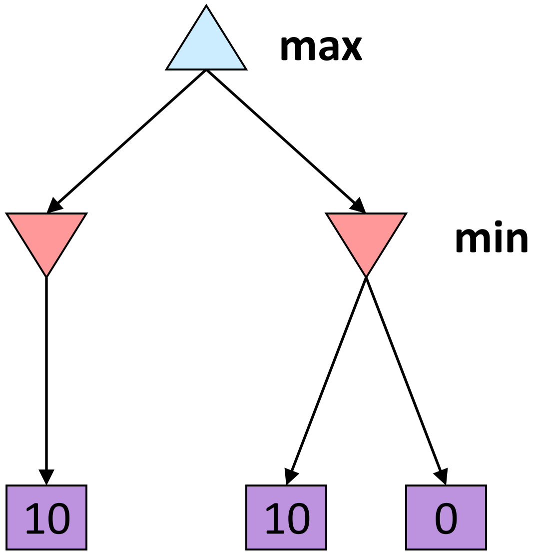
- This pruning has no effect on minimax value computed for the root!

- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection

- Good child ordering improves effectiveness of pruning

- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Chess: 1M nodes/move => depth=8, respectable
  - Full search of complicated games, is still hopeless…

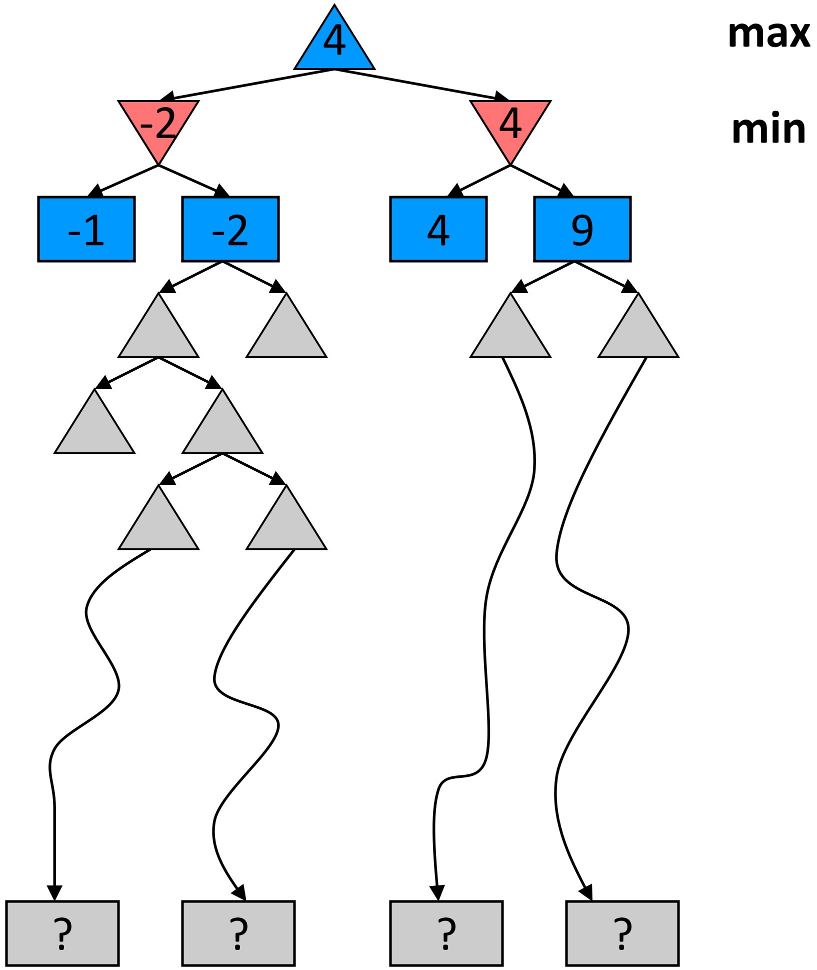
- This is a simple example of metareasoning (computing about what to compute)



Resource Limits II
Bounded lookahead

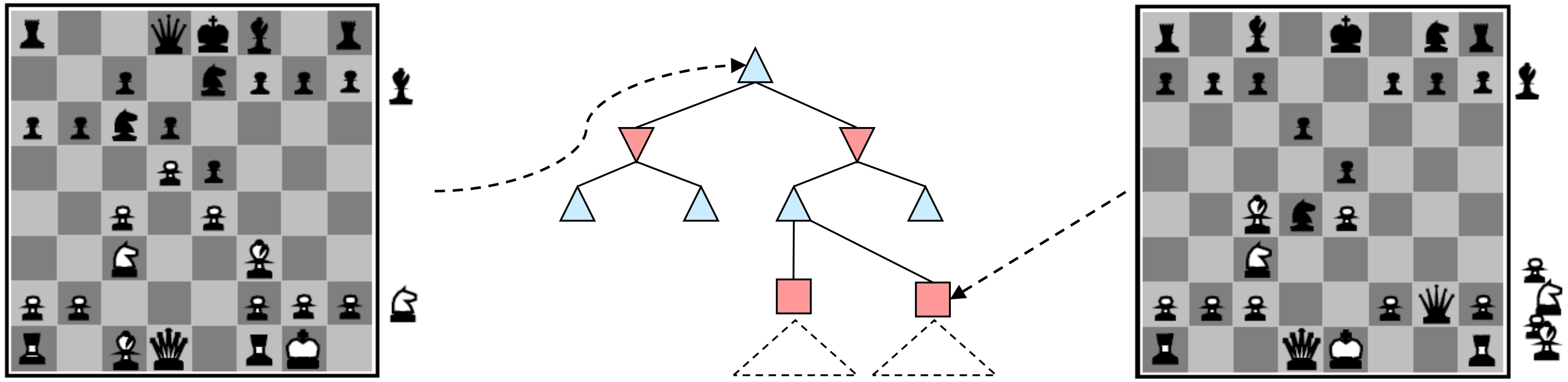
Depth-limited search

- 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
  - For chess, $b \approx 35$ so reaches about depth 4 – not so good
  - α-β reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm



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
- In practice: typically weighted linear sum of features:

  EVAL(s) = $w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$

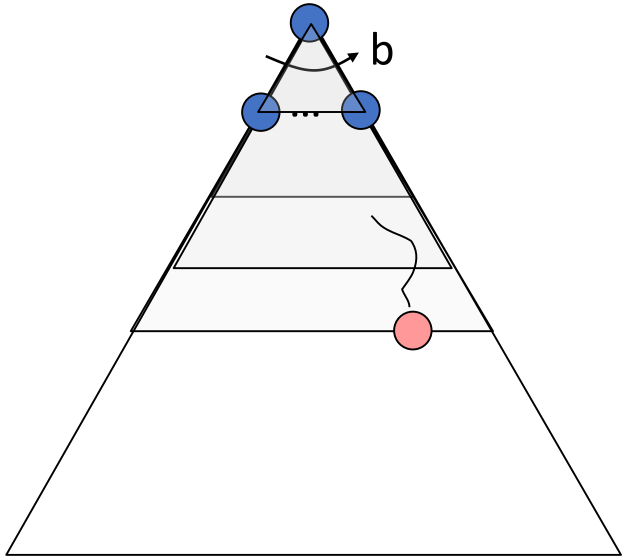
  - e.g. $w_1 = 9$, $f_1(s)$ = (num white queens – num black queens), etc.

Iterative Deepening

Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less. (DFS gives up on any path of length 2)
2. If “1” failed, do a DFS which only searches paths of length 2 or less.
3. If “2” failed, do a DFS which only searches paths of length 3 or less.

....and so on.



Why do we want to do this for multiplayer games?

Note: wrongness of eval functions matters less and less the deeper the search goes!