Adversarial Search 2

Where choosing actions means respecting your opponent

R&N: Chap. 6

With some slides from Dan Klein and Stuart Russell

Minimax Algorithm

1. On each turn, expand the game tree uniformly (**BFS**) from the current state to depth **h**
2. Compute the evaluation function at **every leaf** of the tree
3. Back-up (propagate) the values from the leaves to the root of the tree as follows:
   a. A MAX node gets the maximum of the evaluation of its successors
   b. A MIN node gets the minimum of the evaluation of its successors
4. Select the move toward a MIN node that has the largest backed-up value

Minimax Algorithm: horizon

1. Expand the game tree uniformly from the current state (where it is MAX's turn to play) to depth **h**
2. Compute the evaluation function at every leaf of the tree
3. Back-up (propagate) the values from the leaves to the root of the tree as follows:
   a. A MAX node gets the maximum of the evaluation of its successors
   b. A MIN node gets the minimum of the evaluation of its successors
4. Select the move toward a MIN node that has the largest backed-up value

H=Horizon: Needed to return a decision within allowed time

Game Playing (for MAX)

Repeat until a terminal state is reached

1. Select move using Minimax
2. Execute move
3. Observe MIN's move

Note1: at each turn the large game tree built to horizon h is used to select **only one move**

Note 2: game tree re-built again for next turn **(a sub-tree of depth h-2 can be re-used)**

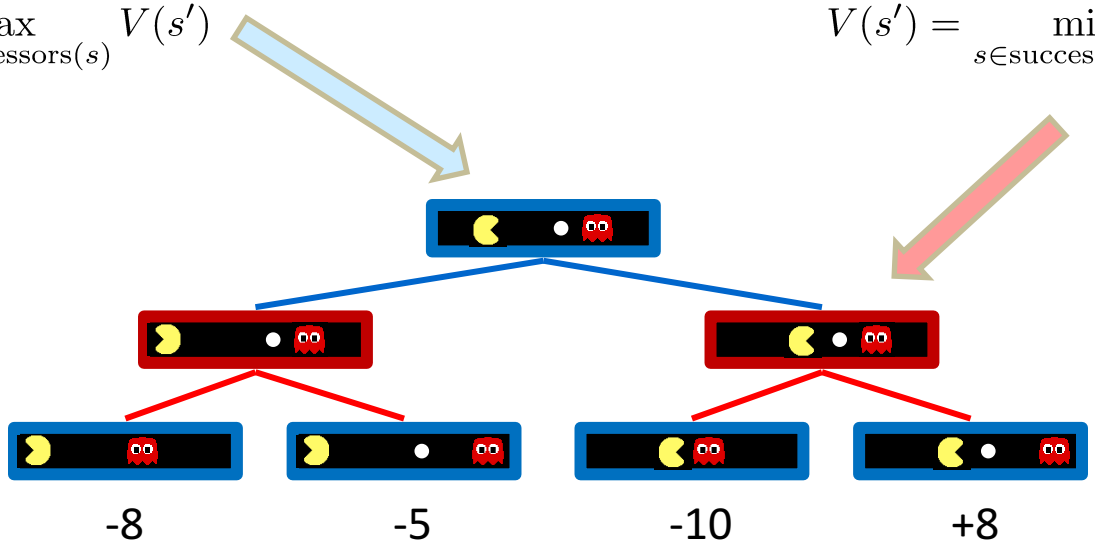
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

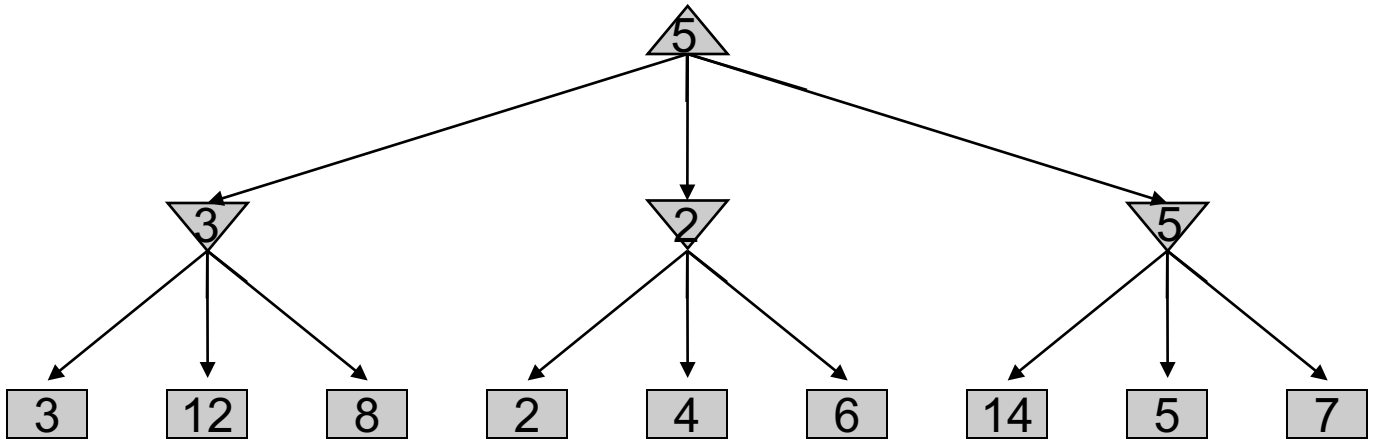
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

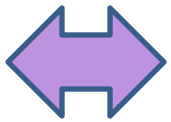
$$V(s) = \text{known}$$

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 (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
```

Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v←MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v,MIN-VALUE(s))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MIN(v,MAX-VALUE(s))
  return v
```

Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v←MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v,MIN-VALUE(s))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MIN(v,MAX-VALUE(s))
  return v
```

Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v←MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v,MIN-VALUE(s))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MIN(v,MAX-VALUE(s))
  return v
```

Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v←MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v,MIN-VALUE(s))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MIN(v,MAX-VALUE(s))
  return v
```

Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v←MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v,MIN-VALUE(s))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MIN(v,MAX-VALUE(s))
  return v
```

Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v←MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v,MIN-VALUE(s))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MIN(v,MAX-VALUE(s))
  return v
```

Minimax Code (Python)

http://aima.cs.berkeley.edu/python/games.html

```
def minimax_decision(state, game):
"""calculate the best move by searching forward all the way to
the terminal states. [Fig. 6.4]"""
    player = game.to_move(state)
    if game.terminal_test(state):
        return game.utility(state, player)
    return max_value(state, player)
```

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

```
def max_value(state, player):
    if game.terminal_test(state):
        return game.utility(state, player)
    v = -infinity
    for (a, s) in game.successors(state):
        v = max(v, min_value(s))
    return v
```

Minimax Code (Python): 2

```
def max_value(state, player):
    if game.terminal_test(state):
        return game.utility(state, player)
    v = -infinity
    for (a, s) in game.successors(state):
        v = max(v, min_value(s))
    return v
```

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

```
def min_value(state, player):
```

Minimax Code (Python): 2

http://aima.cs.berkeley.edu/python/games.html

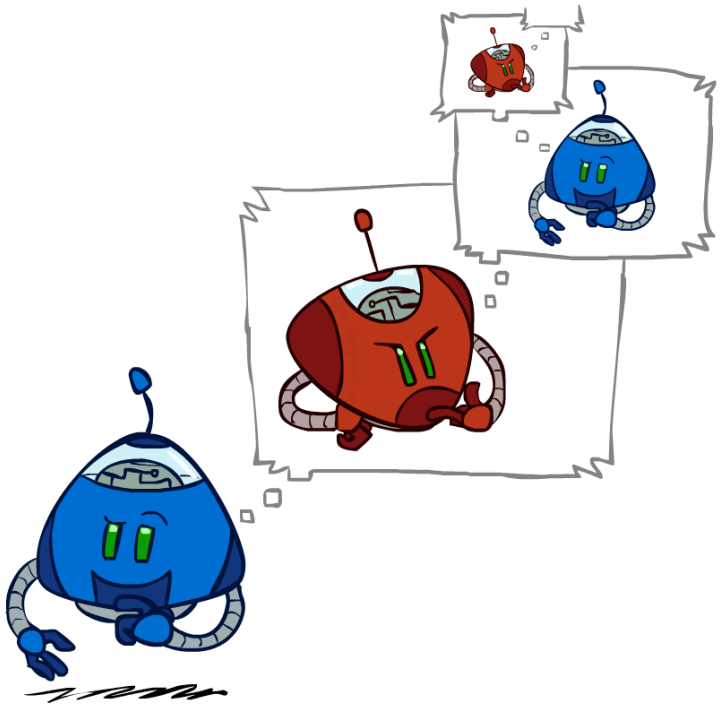
```
def max_value(state, player):
    if game.terminal_test(state):
        return game.utility(state, player)
    v = -infinity
    for (a, s) in game.successors(state):
        v = max(v, min_value(s))
    return v
```

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

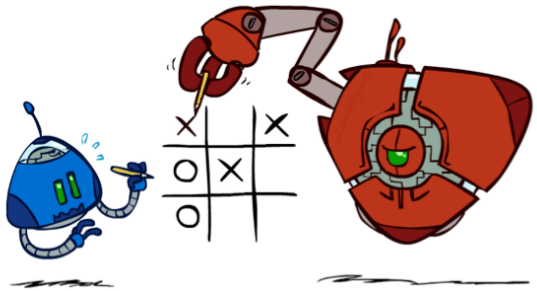
```
def min_value(state):
    if game.terminal_test(state):
        return game.utility(state, player)
    v = infinity
    for (a, s) in game.successors(state):
        v = min(v, max_value(s))
    return v
```

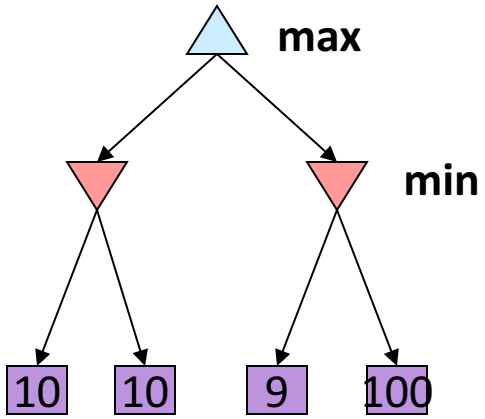
Minimax Efficiency

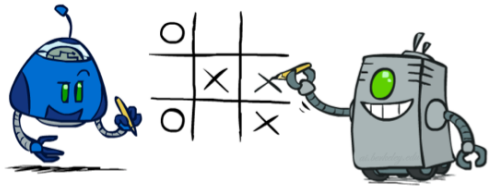
- How efficient is minimax?
  - Just like exhaustive **iterative DFS**
  - Time: $O(b^m)$
  - Space: O(bm)
- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?



Minimax Properties







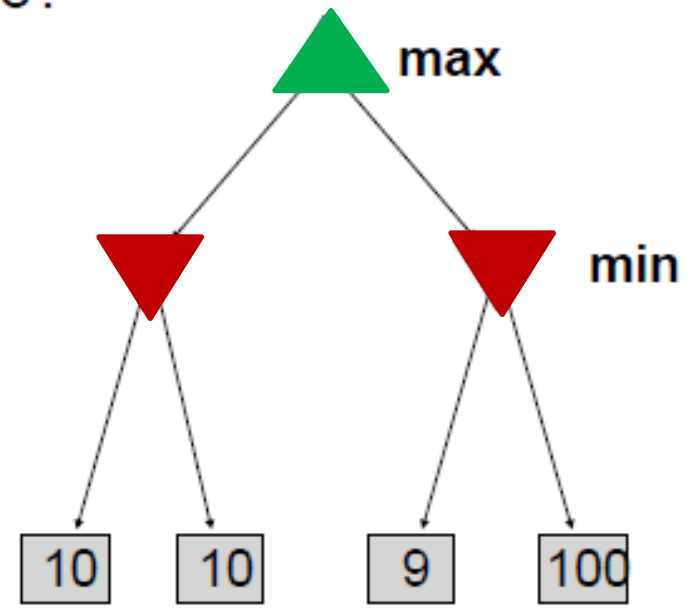
Optimal against a perfect player. Otherwise?

Pacman MiniMax 1

Pacman Minimax 2

Minimax Properties

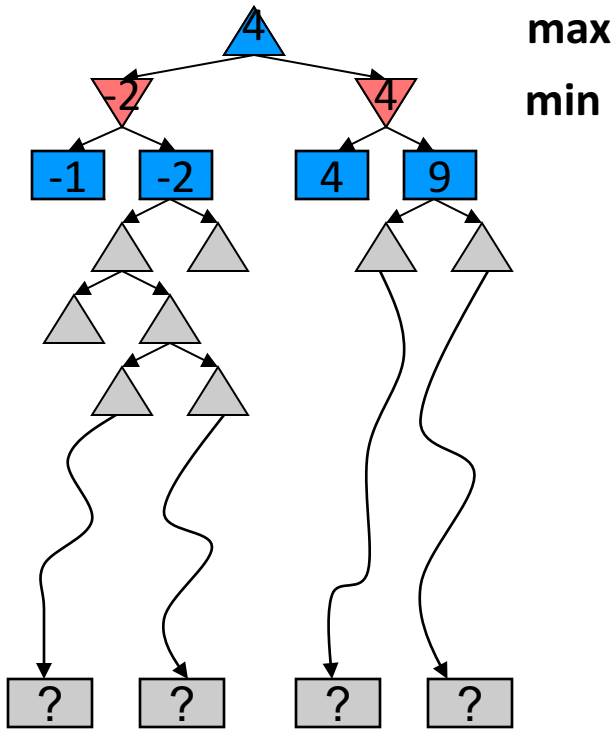
- Optimal?
  - Yes, against perfect player. Otherwise?
- Time complexity?
  - $O(b^m)$
- Space complexity?
  - O(bm)



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

Resource Limits: Depth Limit

- 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
  - Could reach 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



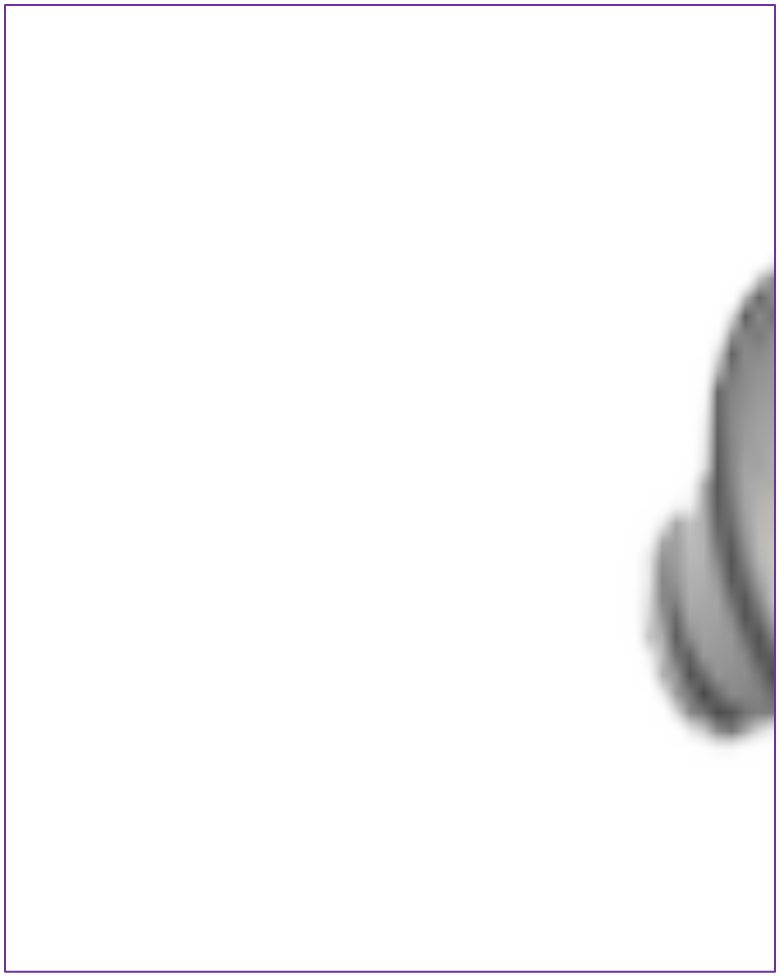
Depth Matters

- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the quality of the evaluation function matters less
- An important example of the tradeoff between complexity of features vs. complexity of computation

Video of Demo Limited Depth (2)

Below the horizon could lurk unseen dangers….

Video of Demo Limited Depth (10)

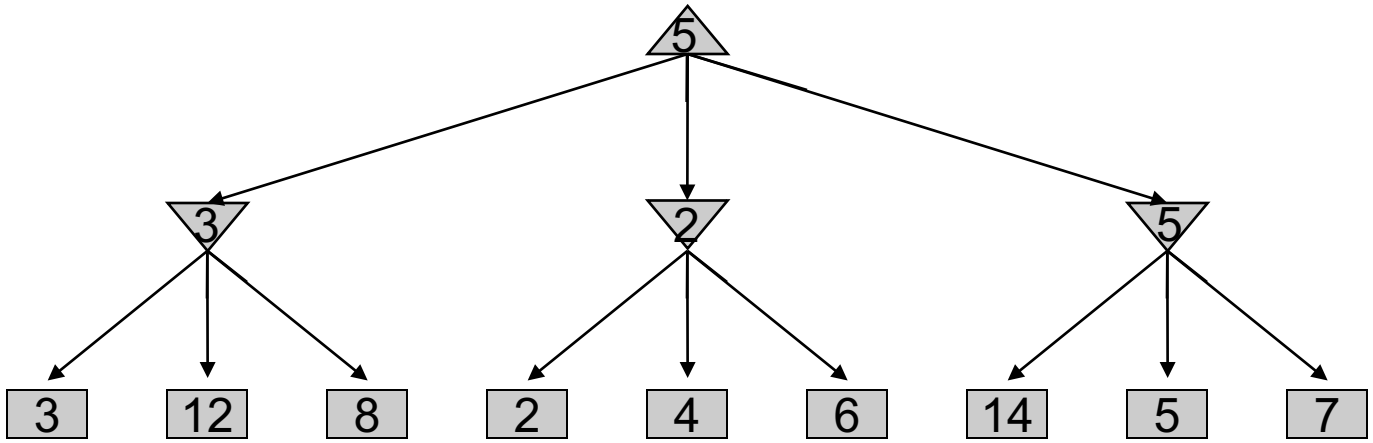


Project 2: Fightin' Pacman

- Due: **Wednesday Feb 28, at 8pm**

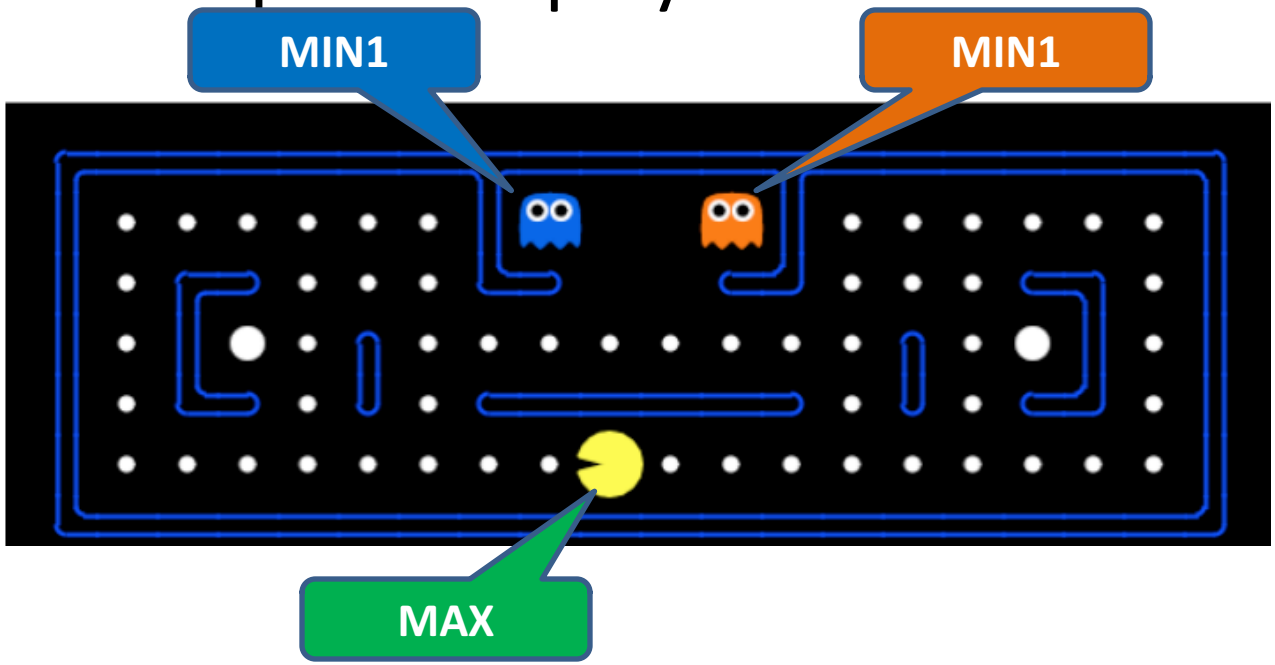
http://www.mathcs.emory.edu/~eugene/cs425/p2/

Minimax Example



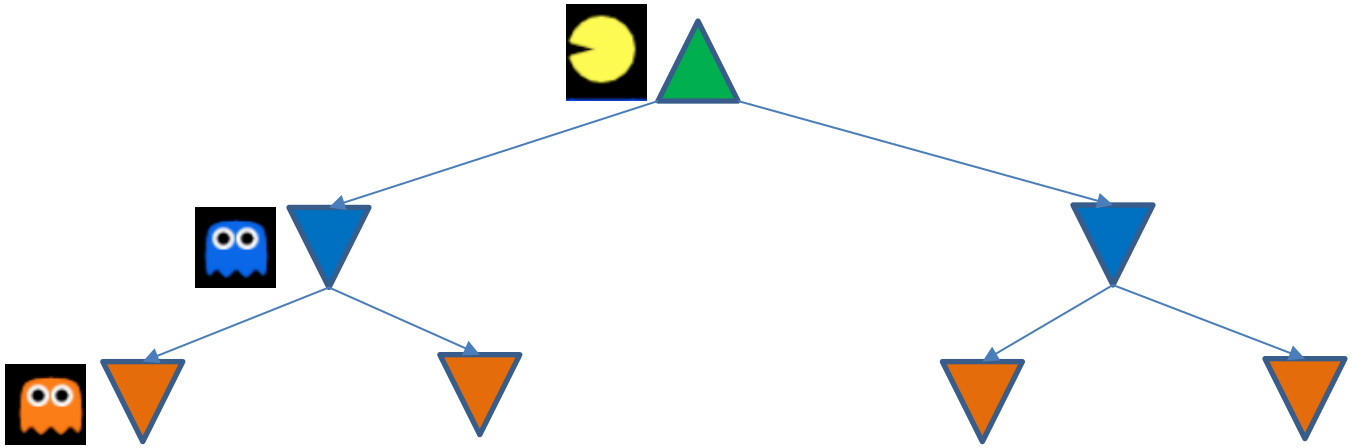
Multiplayer Games: 1 vs. All

- MAX vs. multiple MIN players

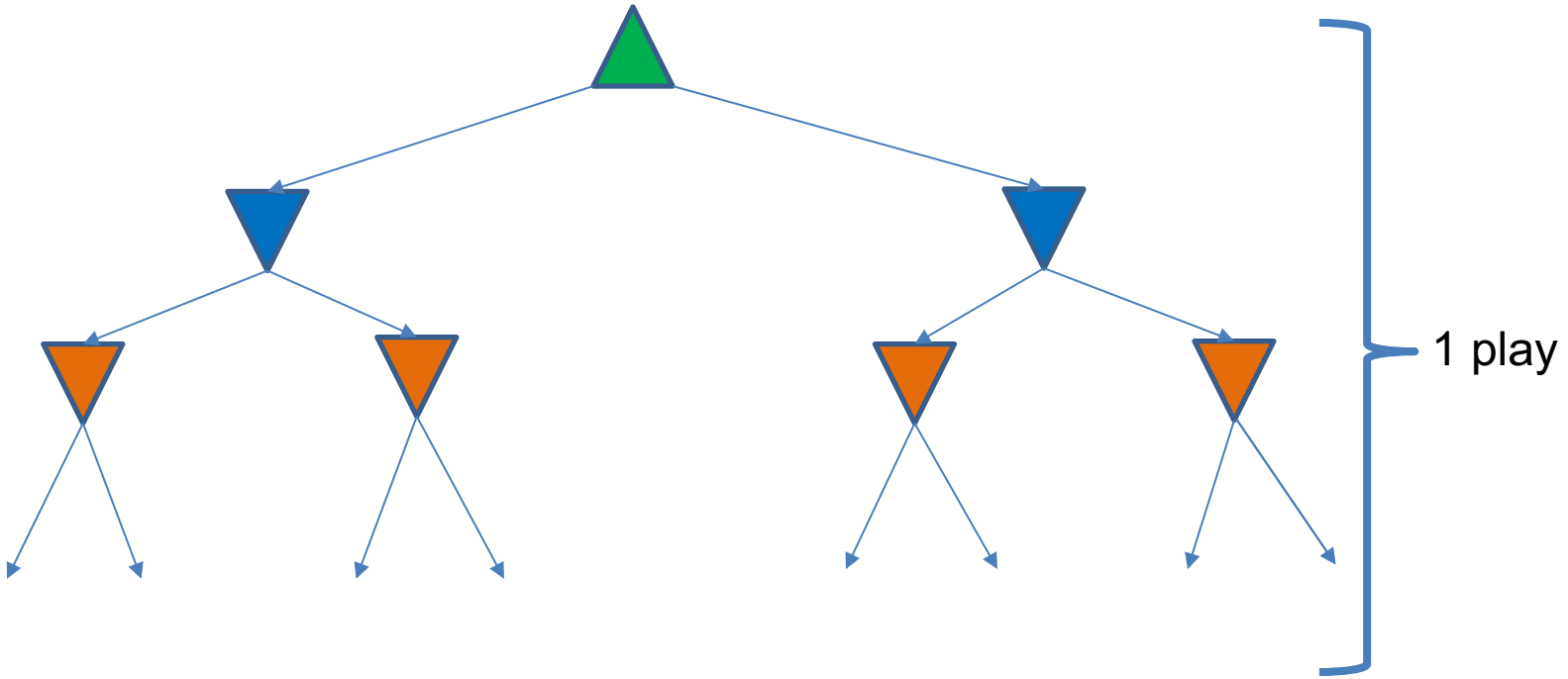


- How does the algorithm change?

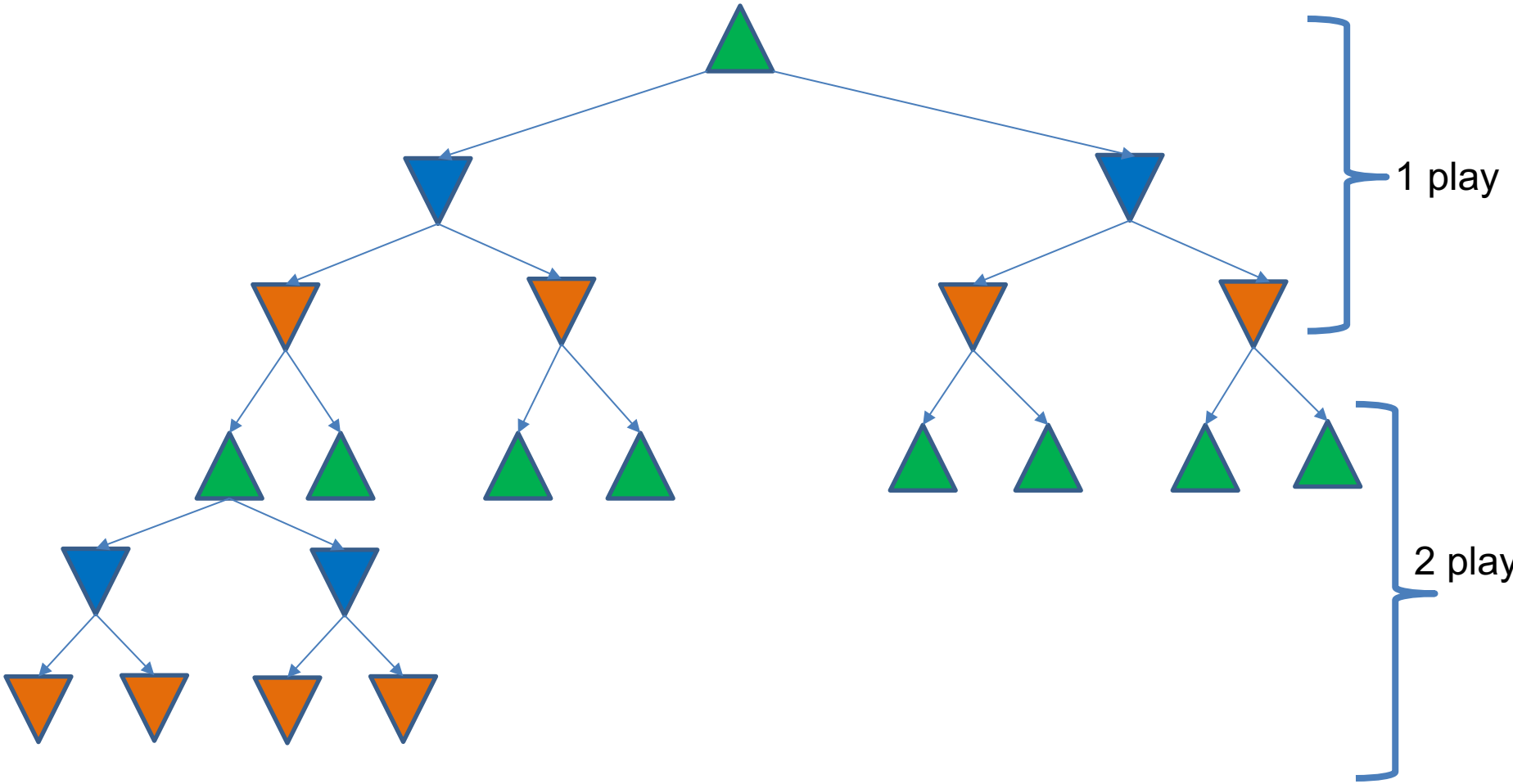
Multiplayer Game Tree: 1 Max, 2 MINs



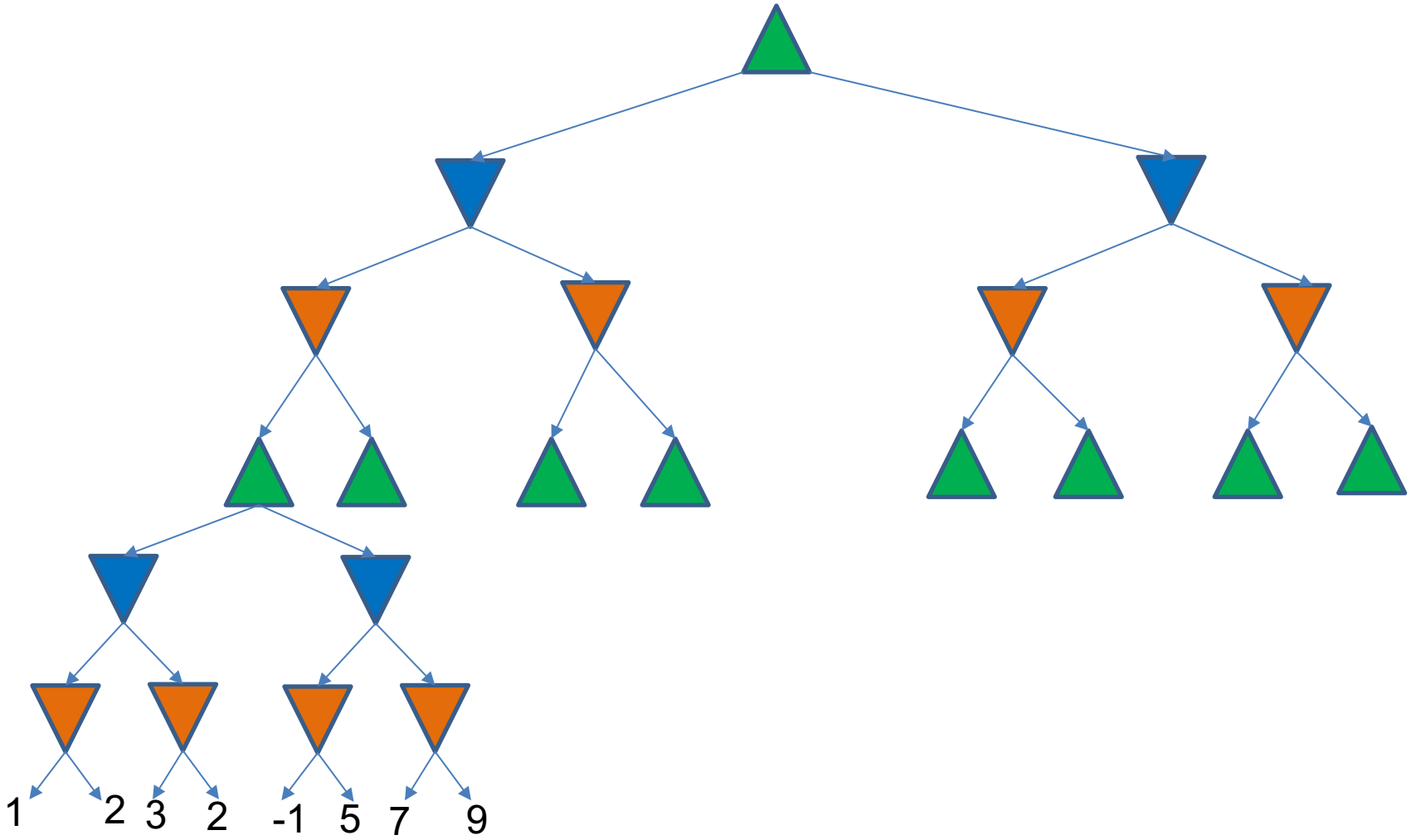
Multiplayer Game Tree: depth 1



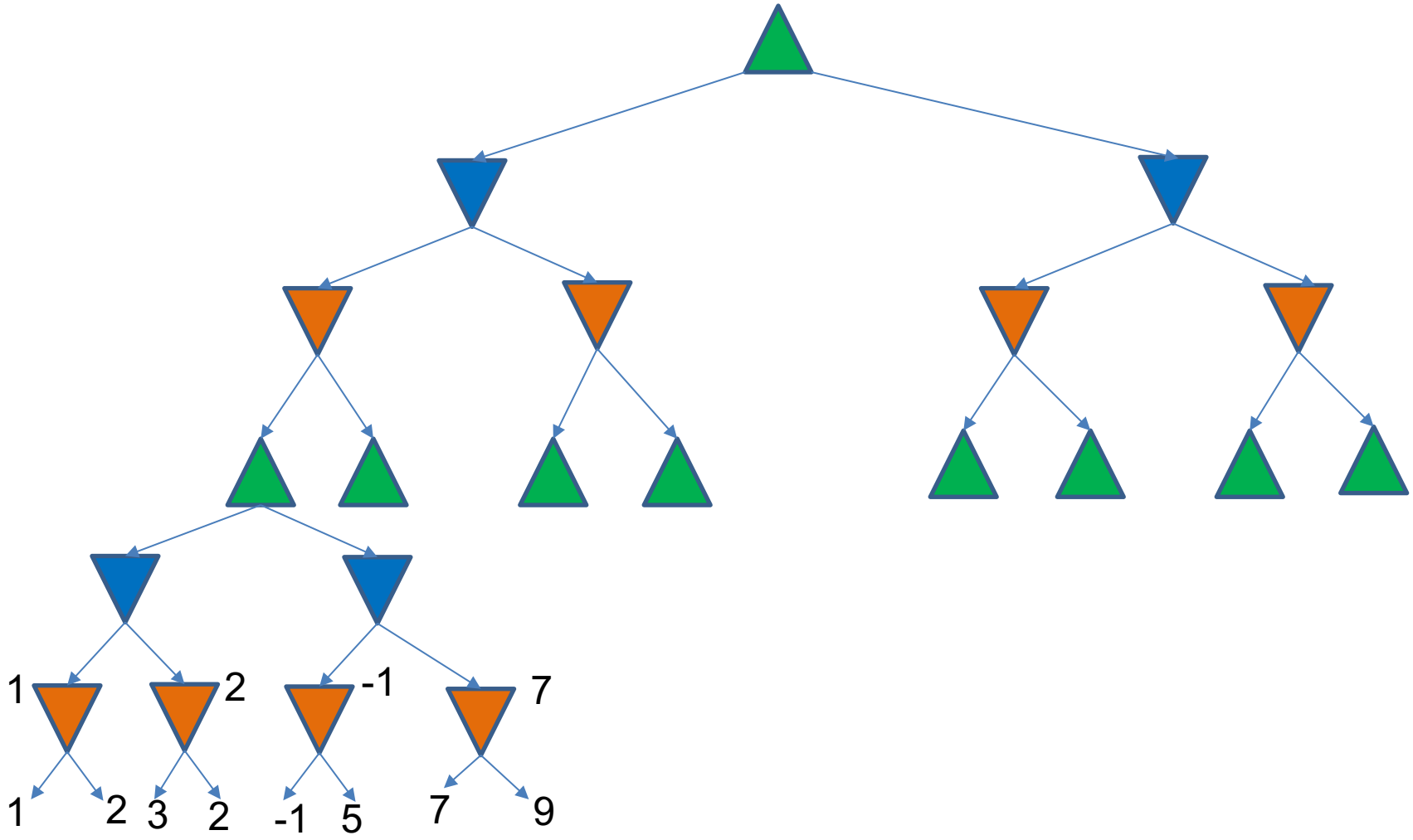
Multiplayer Game Tree: depth 2



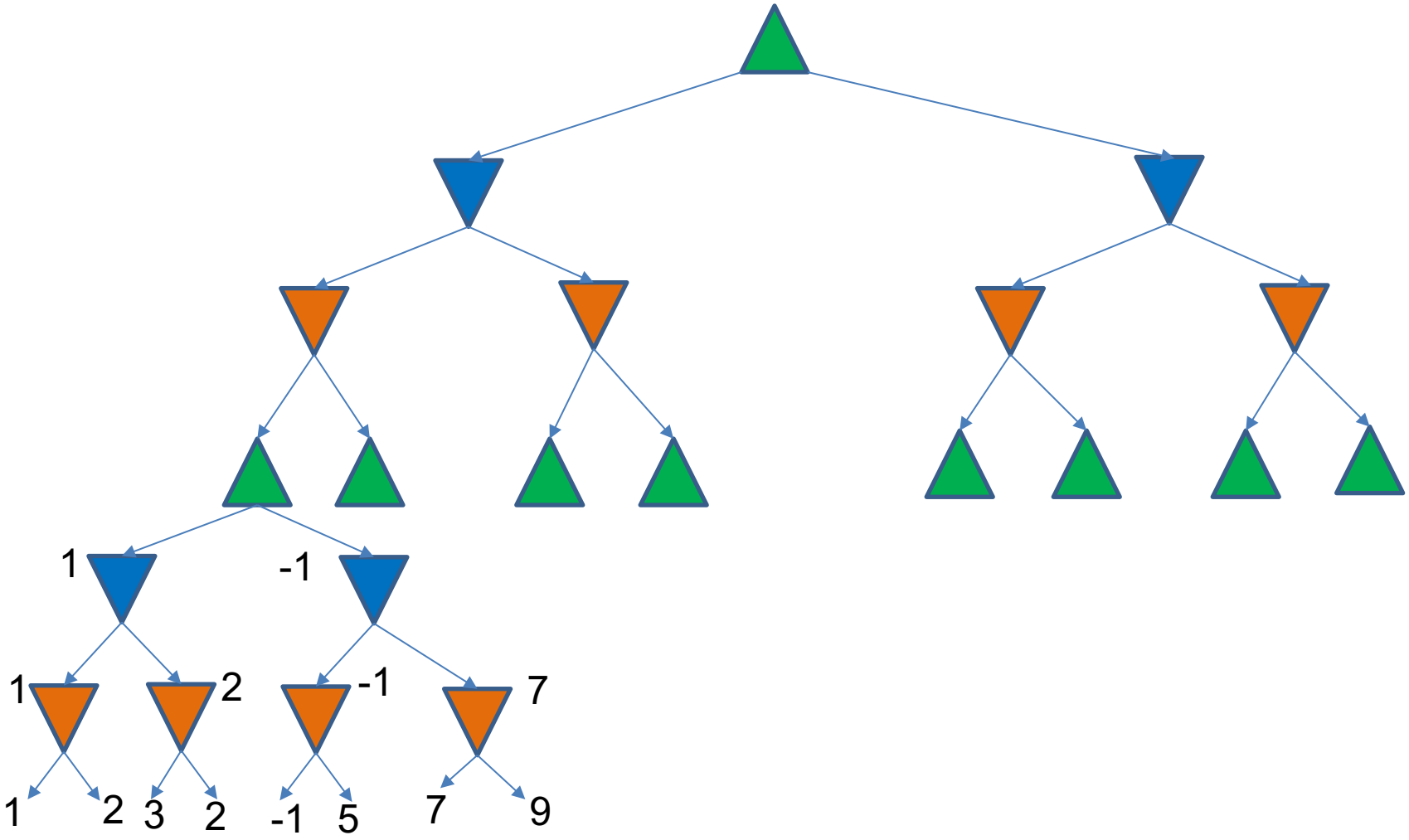
Multiplayer MiniMax: depth 2



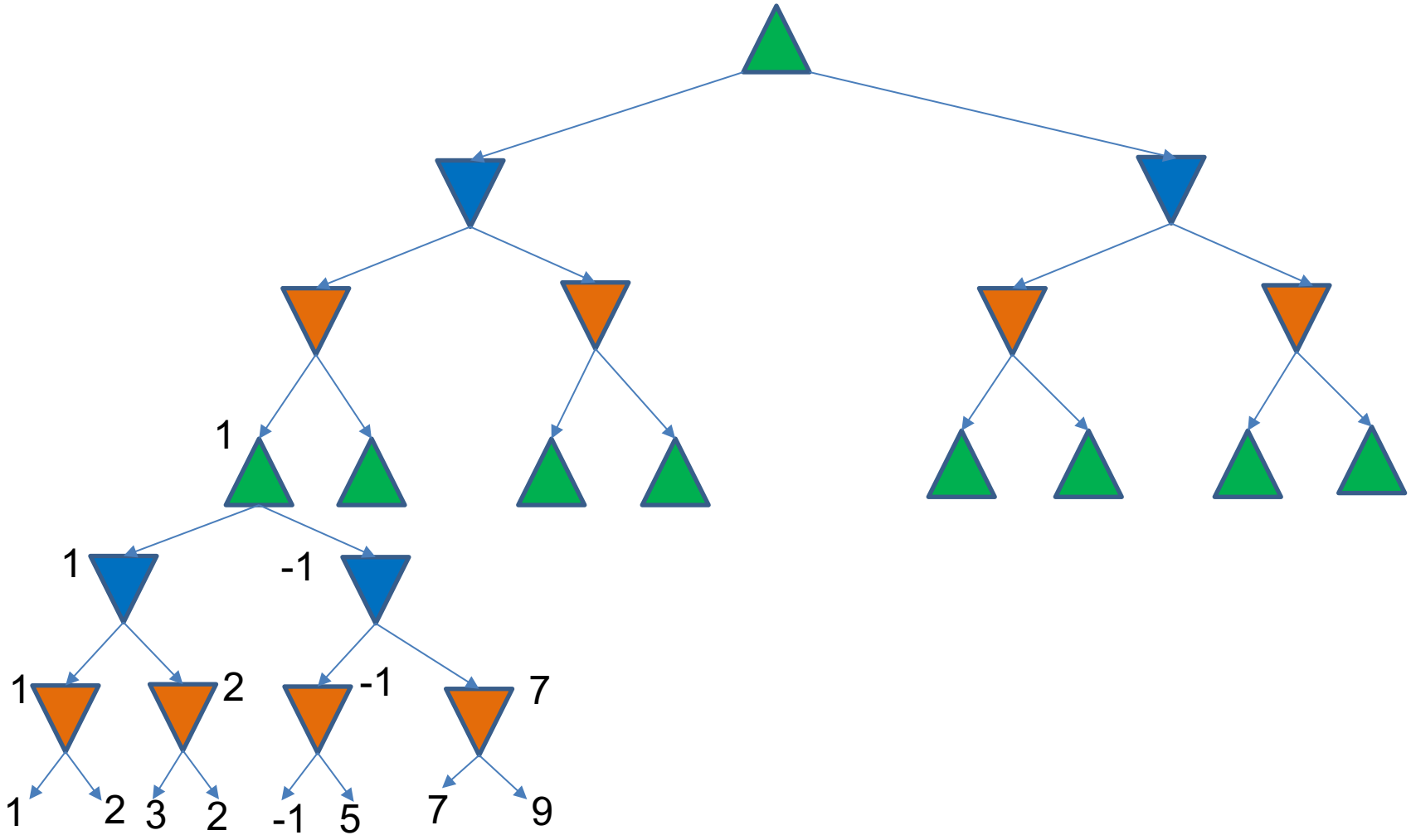
Multiplayer MiniMax: depth 2



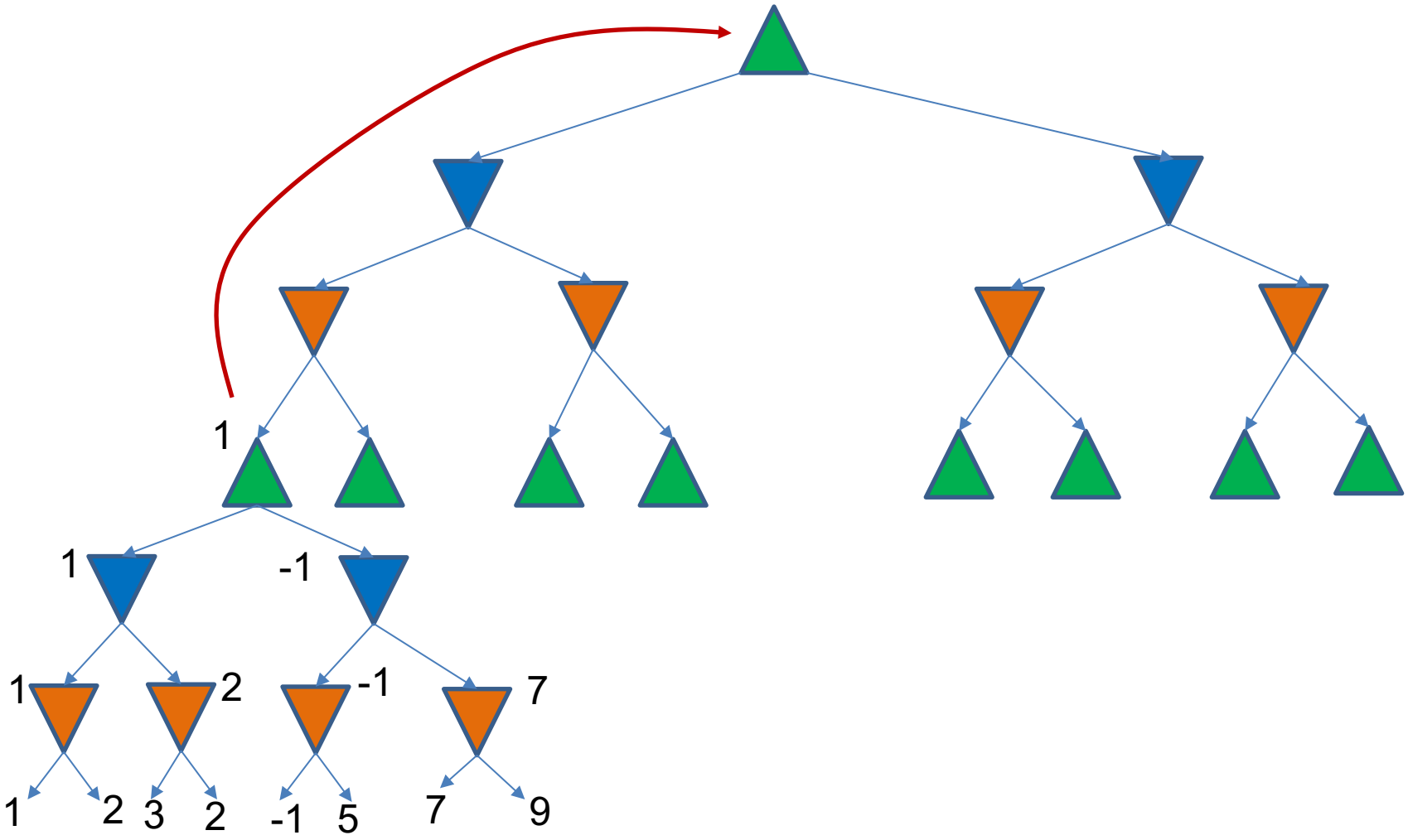
Multiplayer MiniMax: depth 2



Multiplayer MiniMax: depth 2



Multiplayer MiniMax: depth 2



MiniMax Algorithm: Multi-Min version

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v←MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, 1))
  return v
```

Note:
MAX=Agent1
MIN1=Agent2
MIN2=Agent3
….
MINK=AgentN

```
function MIN-VALUE(state, agentIndex) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(agentIndex, state) do
      if (agentIndex >= numAgents) then
          v_temp = MAX-VALUE(s)
      else #another ghost plays
          v_temp = MIN-VALUE(s, agentIndex+1))
      v ← MIN (v, v_temp)
return v
```

problem: collusion possible

- Previous slide (standard minimax analysis) assumes that each player operates to maximize only their own utility
- In practice, players **could make alliances**
  - Ex: C strong, A and B both weak
  - May be best for A and B to attack C rather than each other
- If game is not zero-sum (i.e., utility(A) = - utility(B) then alliances can be useful even with 2 players
  - e.g., both cooperate to maximum the sum of the utilities
- Ignore this, **assume non-cooperative opponents**

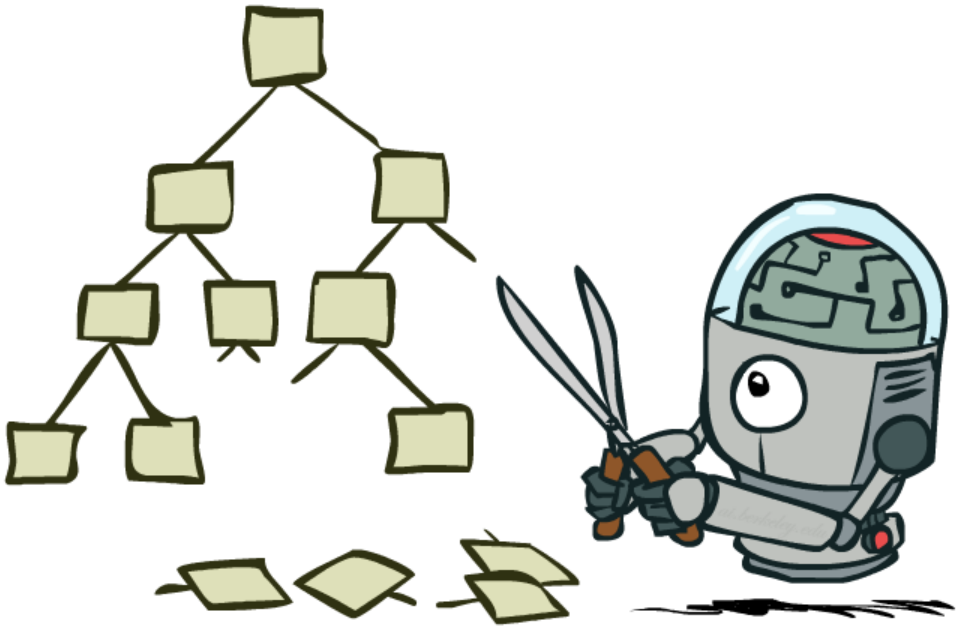
Minimax Algorithm: Summary

- Complete depth-first exploration of the game tree

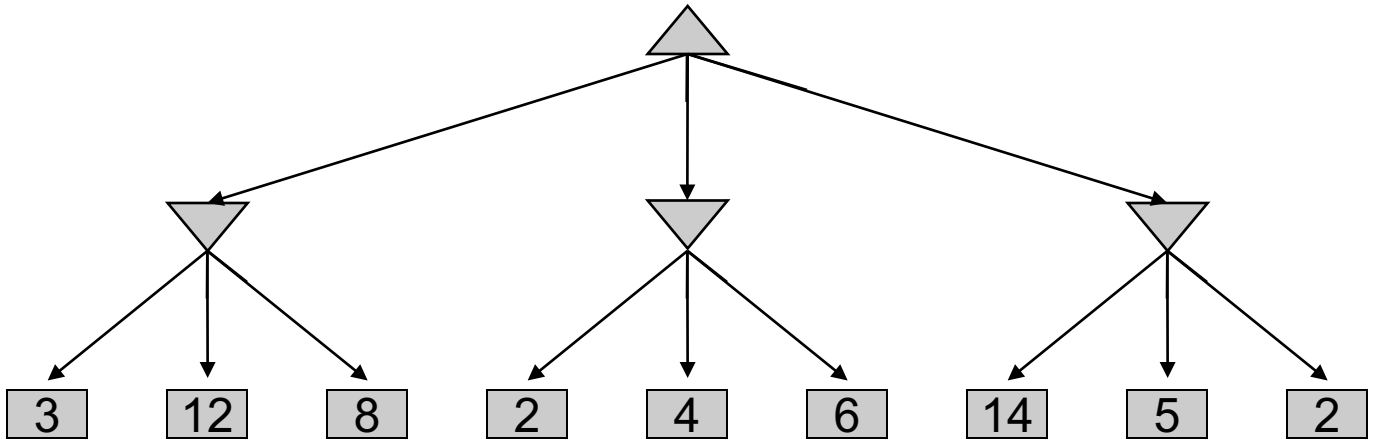
- Complexity:
  - Max depth = d, b legal moves at each point
  - E.g., Chess: d ~ 100, b ~35

Criterion	Minimax
Time	$O(b^m)$ ☹
Space	O(bm) ☺

Game Tree Pruning

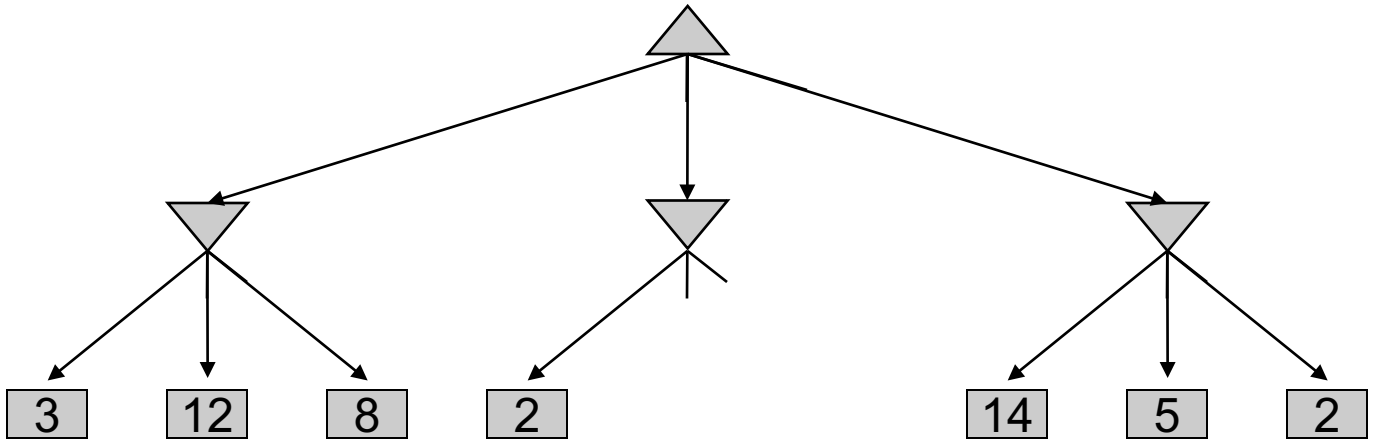


Minimax Example



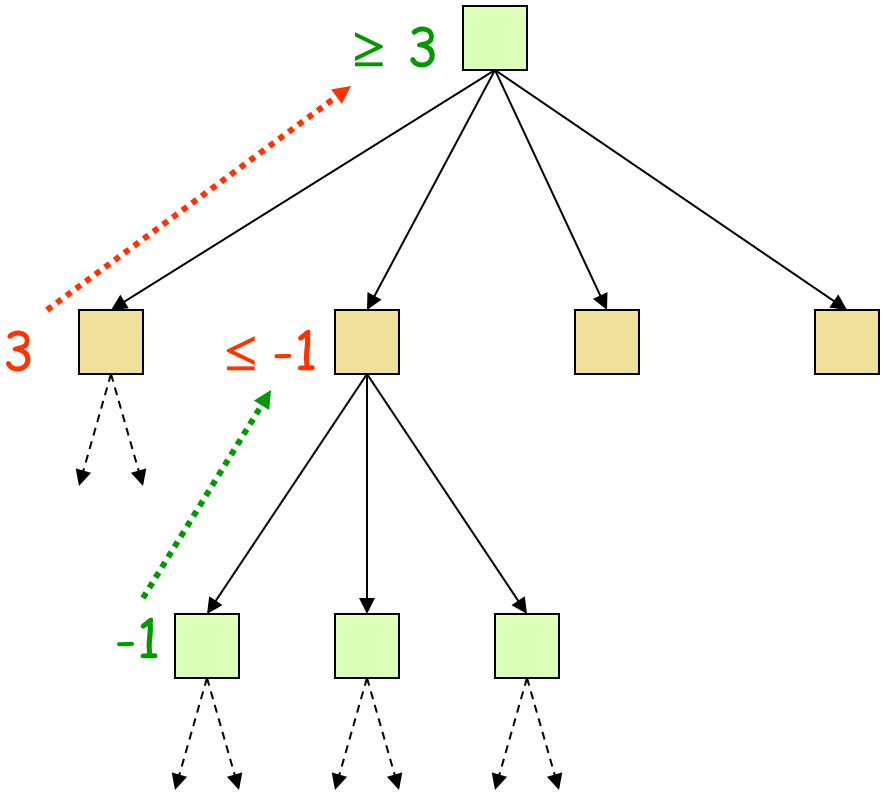
Do we have to explore all the nodes?

Minimax Pruning



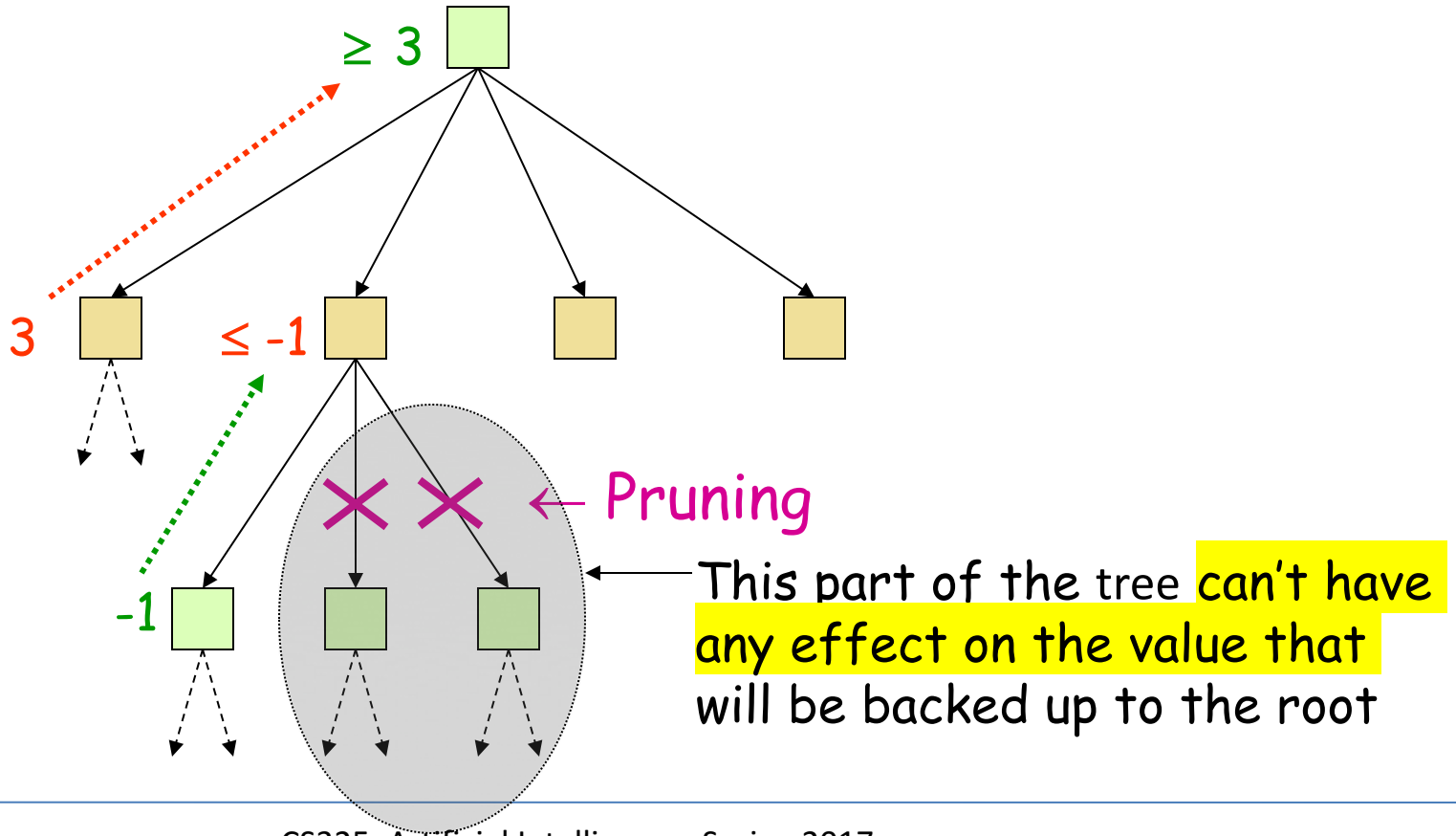
Can we do better?

Yes ! Much better !



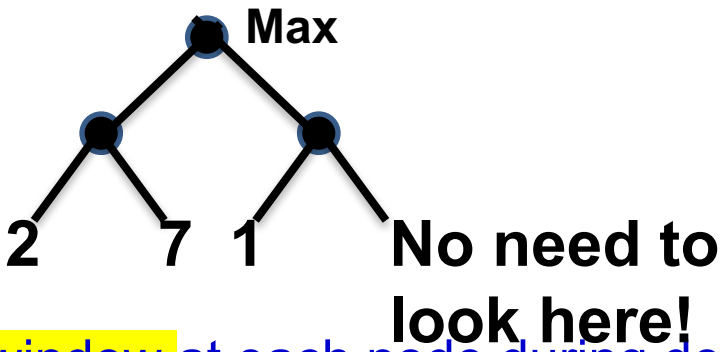
Can we do better?

Yes ! Much better !



Alpha-Beta Pruning

- Typically can only look 3-4 ply in allowable chess time
- Alpha-beta pruning simplifies search space while keeping optimality
  - By applying common sense
  - If one route allows queen to be captured and a better move is available
  - Then don't search further down bad path
  - If one route would be bad for opponent, ignore that route also



Maintain [alpha, beta] window at each node during depth-first search

alpha = lower bound, change at max levels
beta = upper bound, change at min levels

Alpha-Beta Pruning

- **Explore** the game tree to depth **h** in depth-first manner
- **Back** up *alpha* and *beta* values whenever possible
- **Prune** branches that can't lead to changing the final decision

Alpha-beta Algorithm

- Depth first search – only considers nodes along a single path at any time

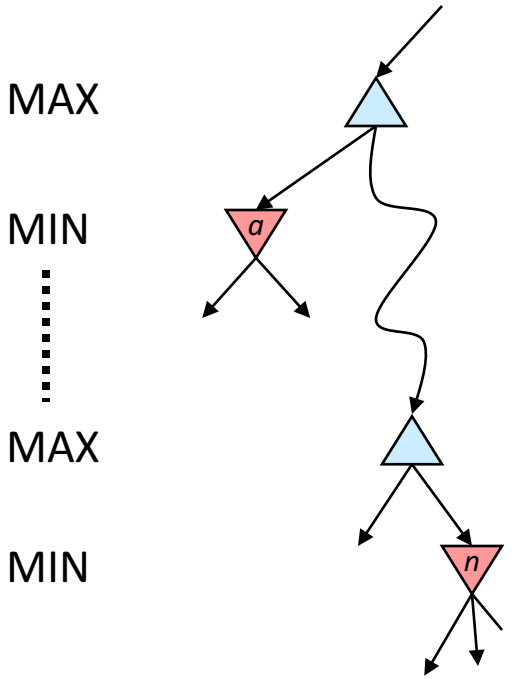
→ α = highest-value choice we have found at any choice point along the path for MAX

→ β = lowest-value choice we have found at any choice point along the path for MIN

- update values of α and β during search, and prune remaining branches when the value is worse than the current α or β value for MAX or MIN

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

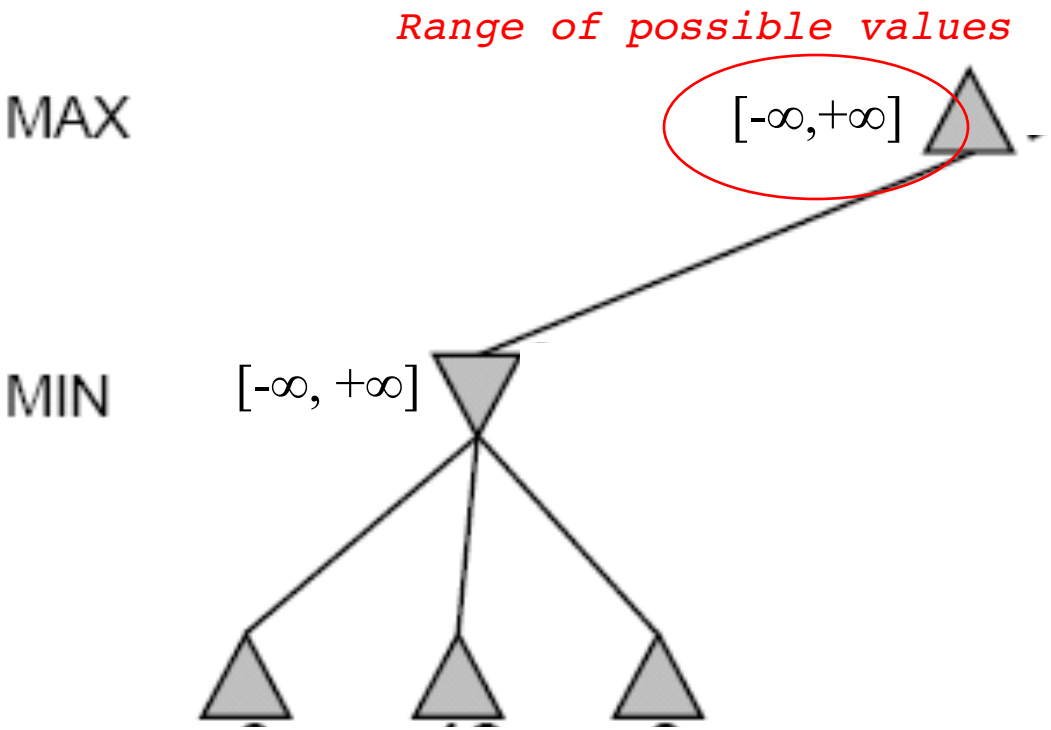
α: MAX's best option on path to root
β: MIN's best option on path to root

```
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
```

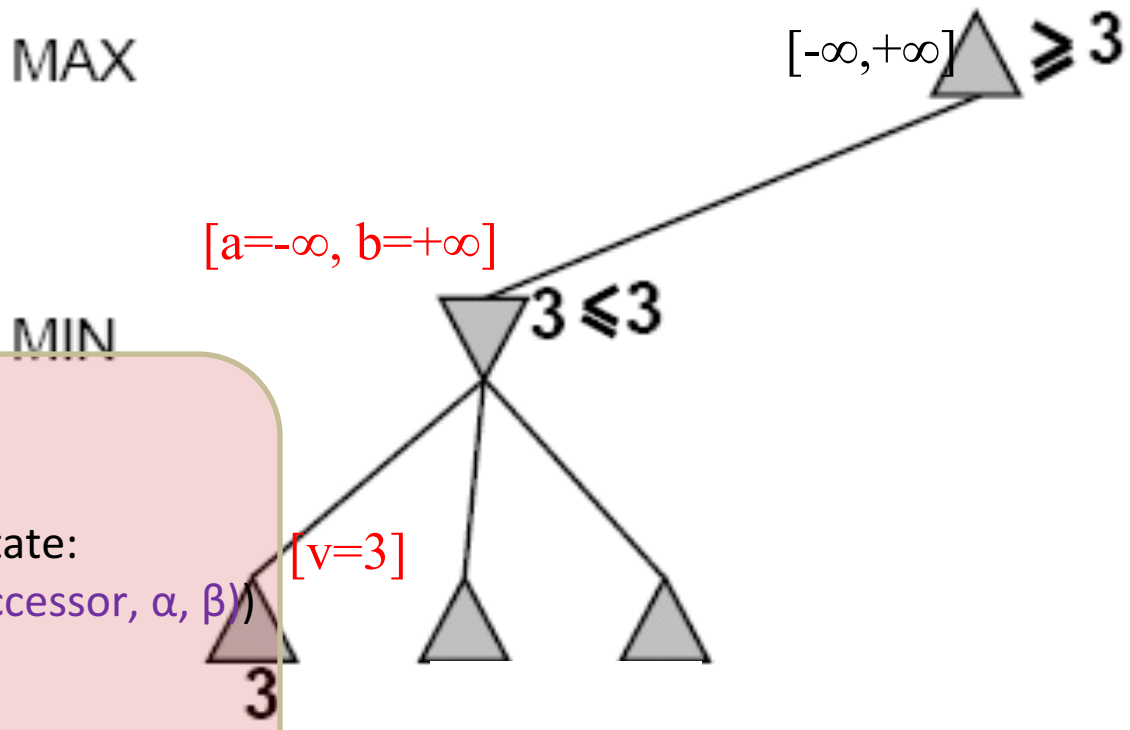
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Alpha-Beta Example

Do DF-search until first leaf

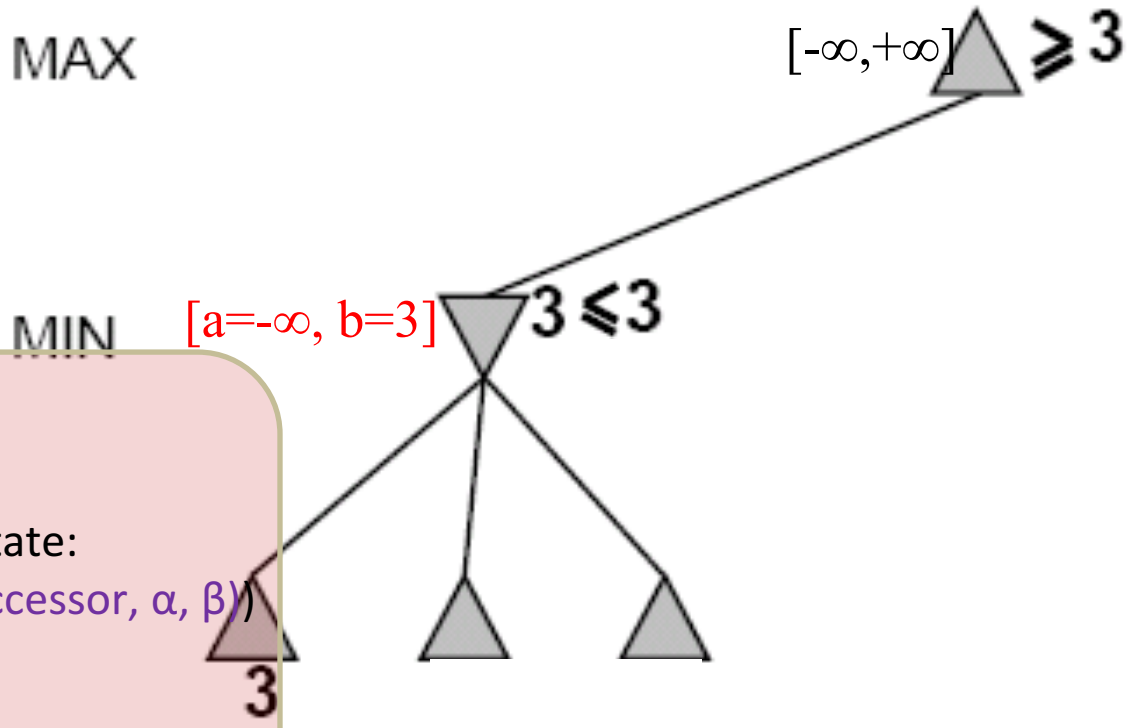


Alpha-Beta Example (continued)



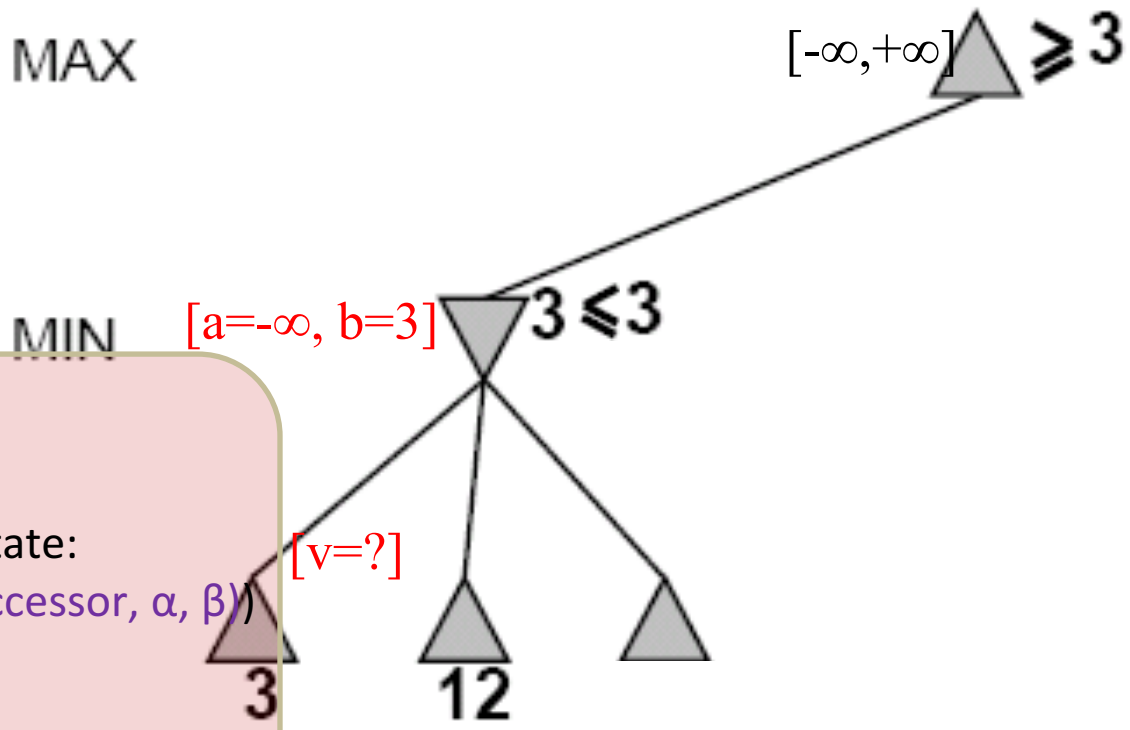

```
f min-value(state , α, β):
  initialize v = +∞
  for each successor of state:
      v = min(v, value(successor, α, β))
      if v ≤ α return v
      β = min(β, v)
  return v
```

Alpha-Beta Example (continued)



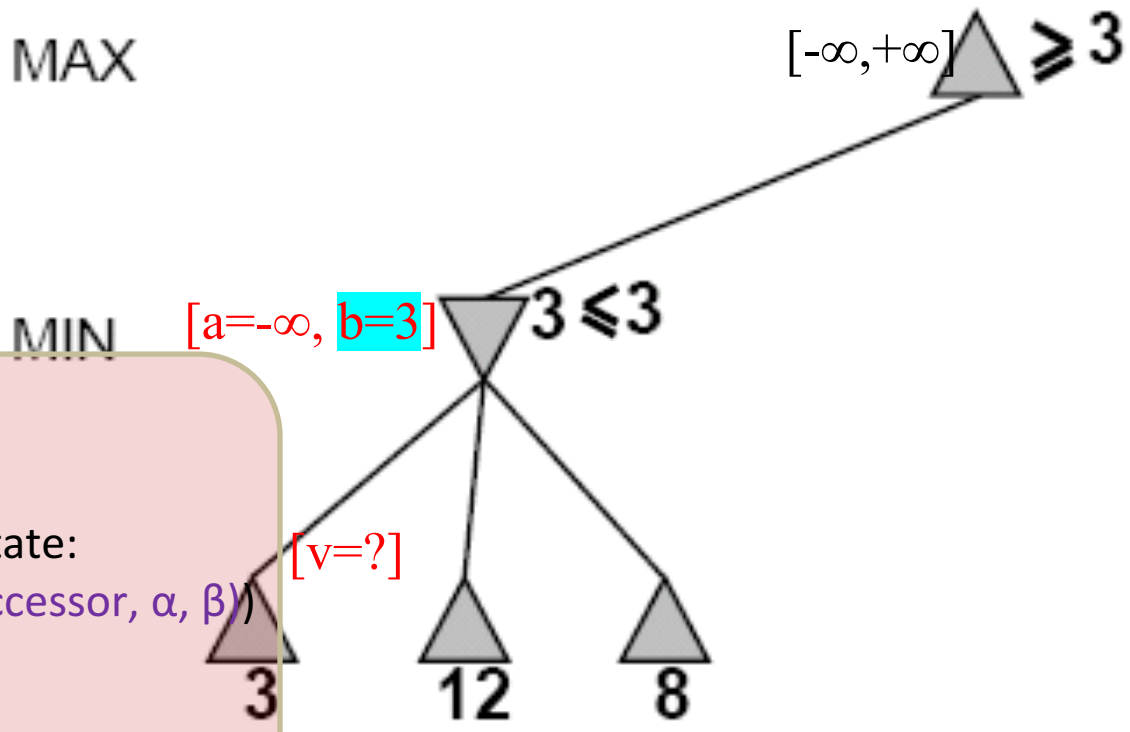

```
f min-value(state , α, β):
  initialize v = +∞
  for each successor of state:
      v = min(v, value(successor, α, β))
      if v ≤ α return v
      β = min(β, v)
  return v
```

Alpha-Beta Example (continued)



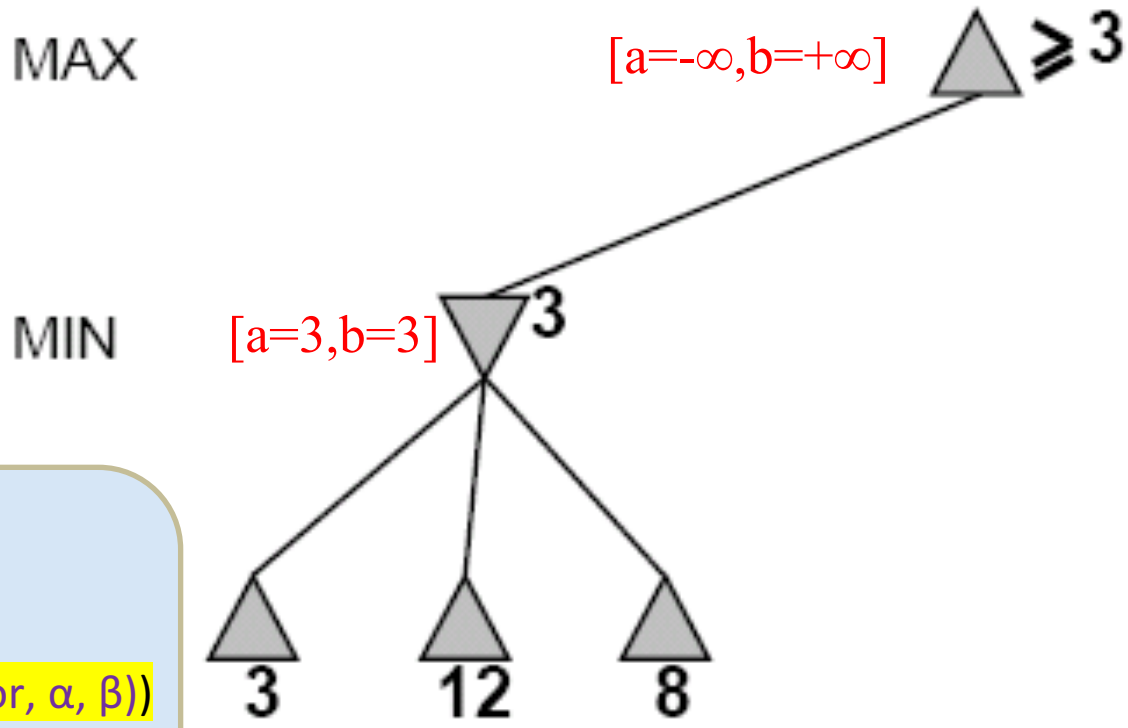

```
f min-value(state , α, β):
  initialize v = +∞
  for each successor of state:
      v = min(v, value(successor, α, β))
      if v ≤ α return v
      β = min(β, v)
  return v
```

Alpha-Beta Example (continued)



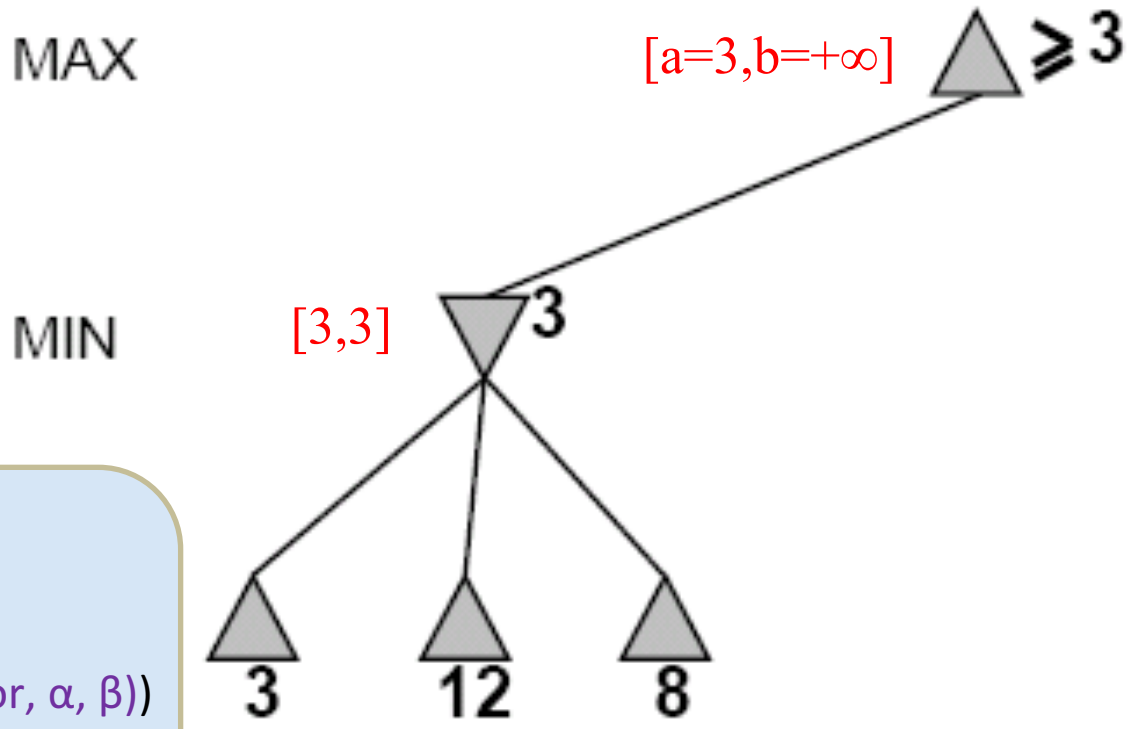

```
f min-value(state , α, β):
  initialize v = +∞
  for each successor of state:
      v = min(v, value(successor, α, β))
      if v ≤ α return v
      β = min(β, v)
  return v
```

Alpha-Beta Example (continued)



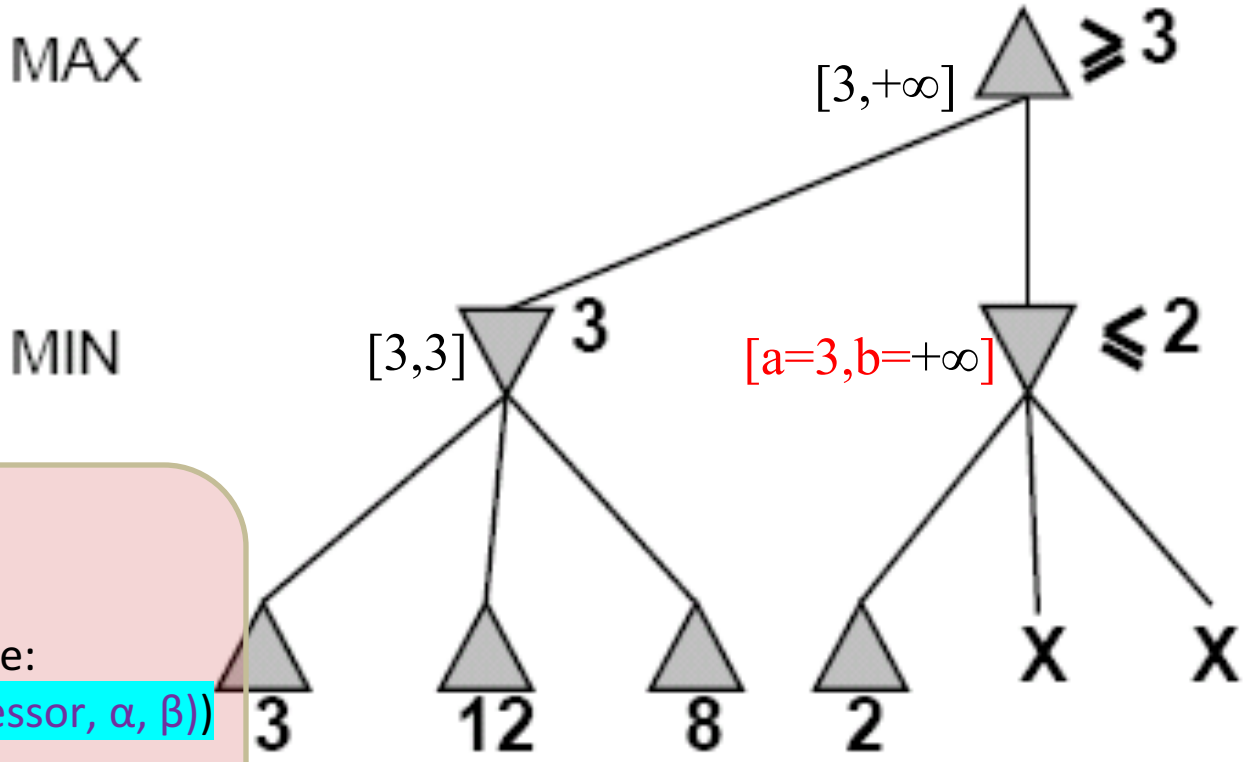

```
f max-value(state, α, β):
 initialize v = -∞
 for each successor of state:
      v = max(v, value(successor, α, β))
      if v ≥ β return v
      α = max(α, v)
 return v
```

Alpha-Beta Example (continued)



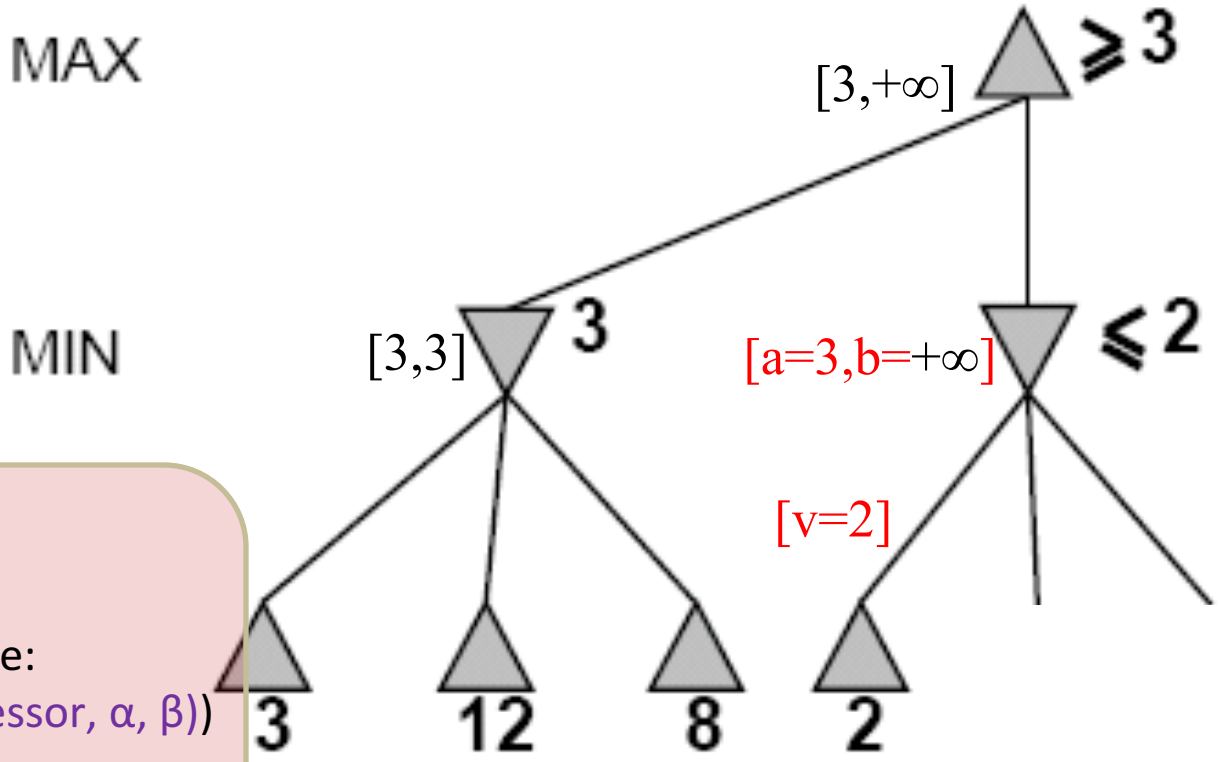

```
f max-value(state, α, β):
 initialize v = -∞
 for each successor of state:
      v = max(v, value(successor, α, β))
      if v ≥ β return v
      α = max(α, v)
 return v
```

Alpha-Beta Example (continued)



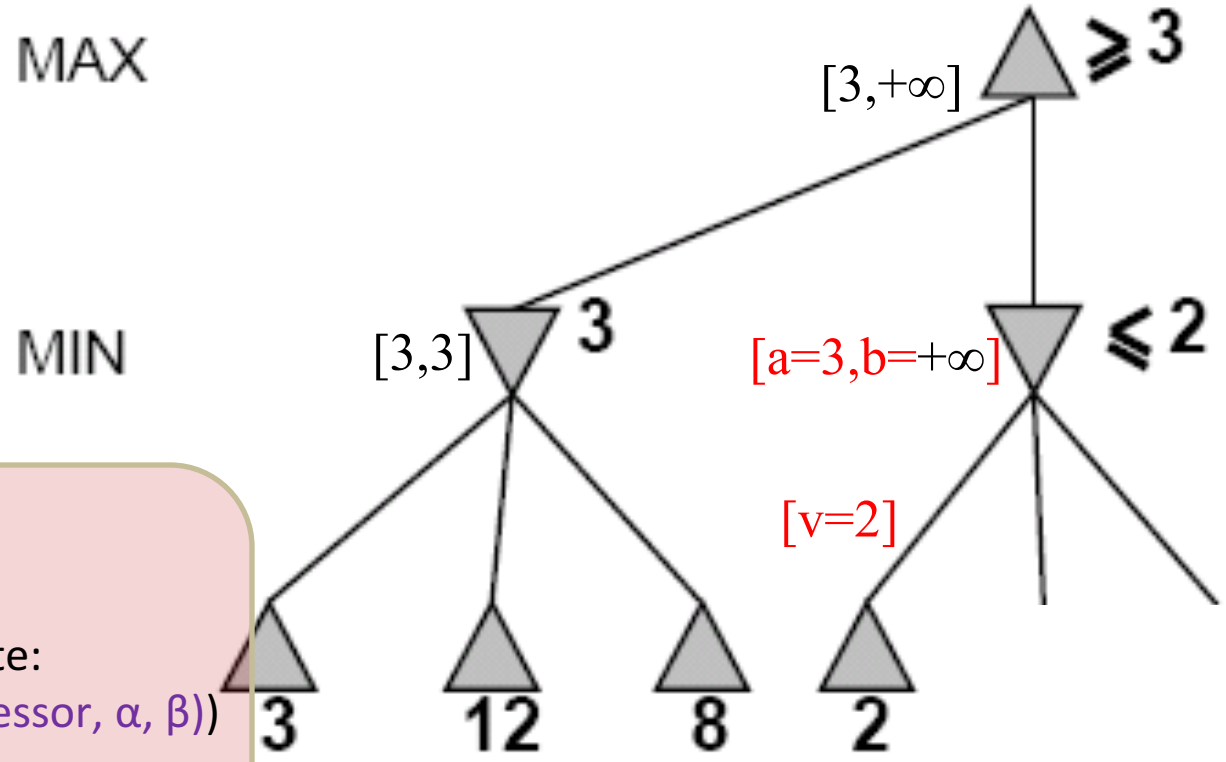

```
f min-value(state , α, β):
 initialize v = +∞
 for each successor of state:
      v = min(v, value(successor, α, β))
      if v ≤ α return v
      β = min(β, v)
 return v
```

Alpha-Beta Example (continued)



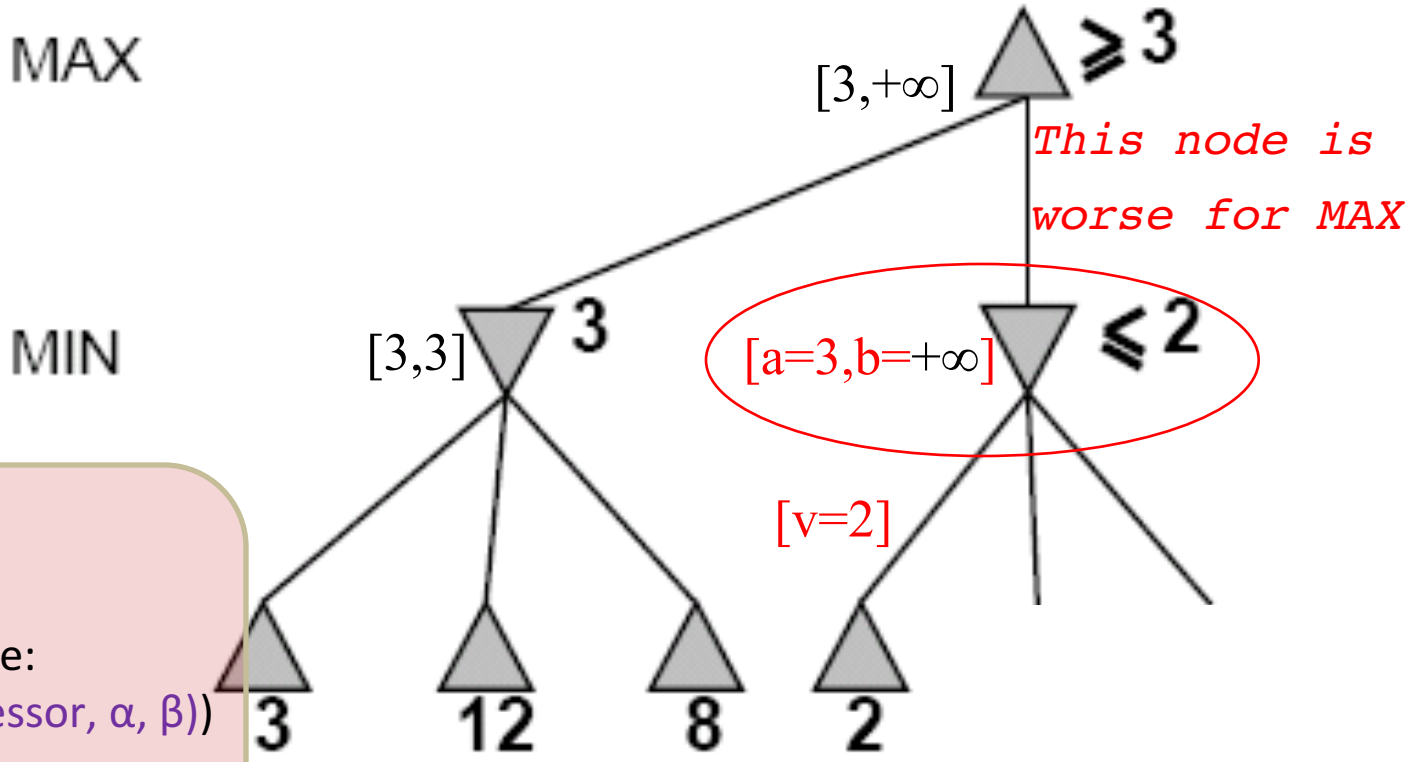

```
f min-value(state , α, β):
  initialize v = +∞
  for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
  return v
```

Alpha-Beta Example (continued)



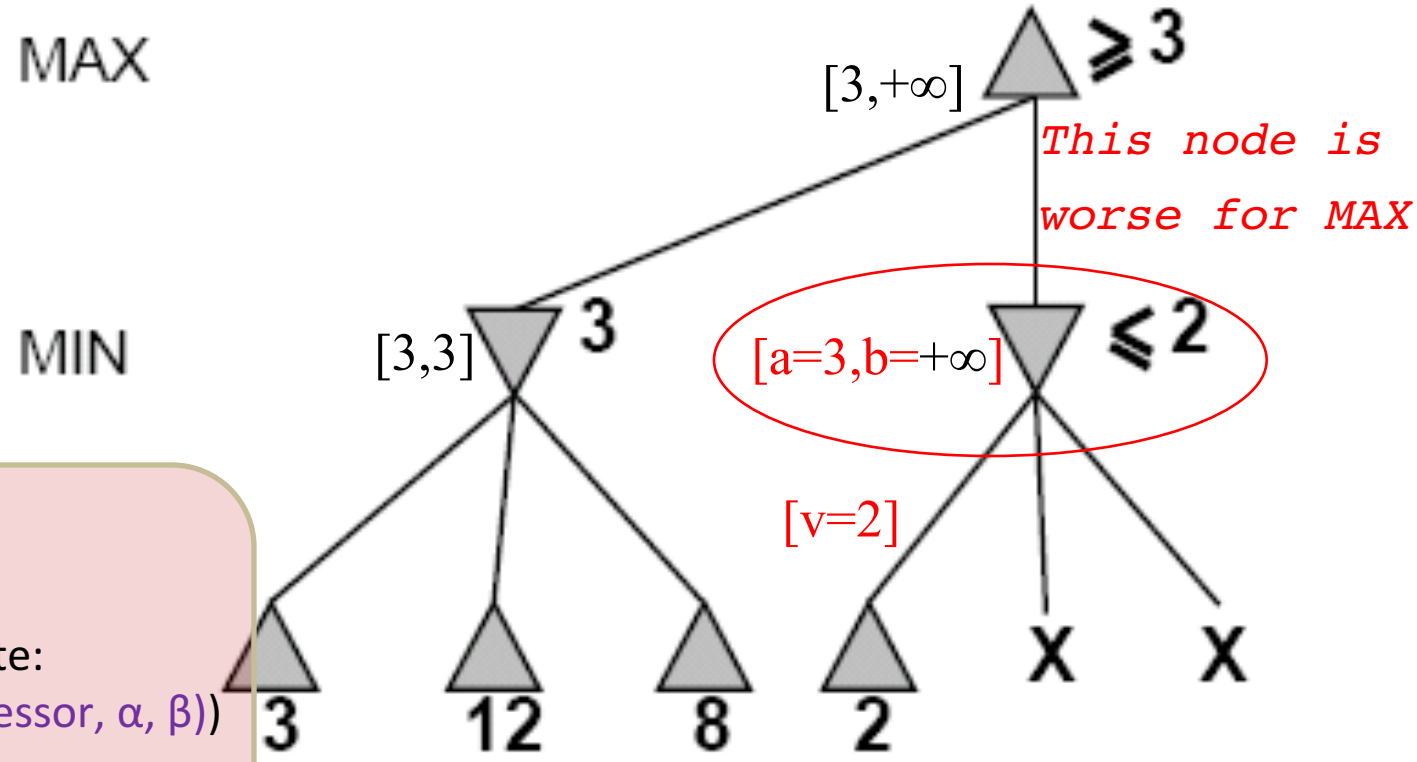

```
f min-value(state , α, β):
  initialize v = +∞
  for each successor of state:
      v = min(v, value(successor, α, β))
      if v ≤ α return v
      β = min(β, v)
  return v
```

Alpha-Beta Example (continued)



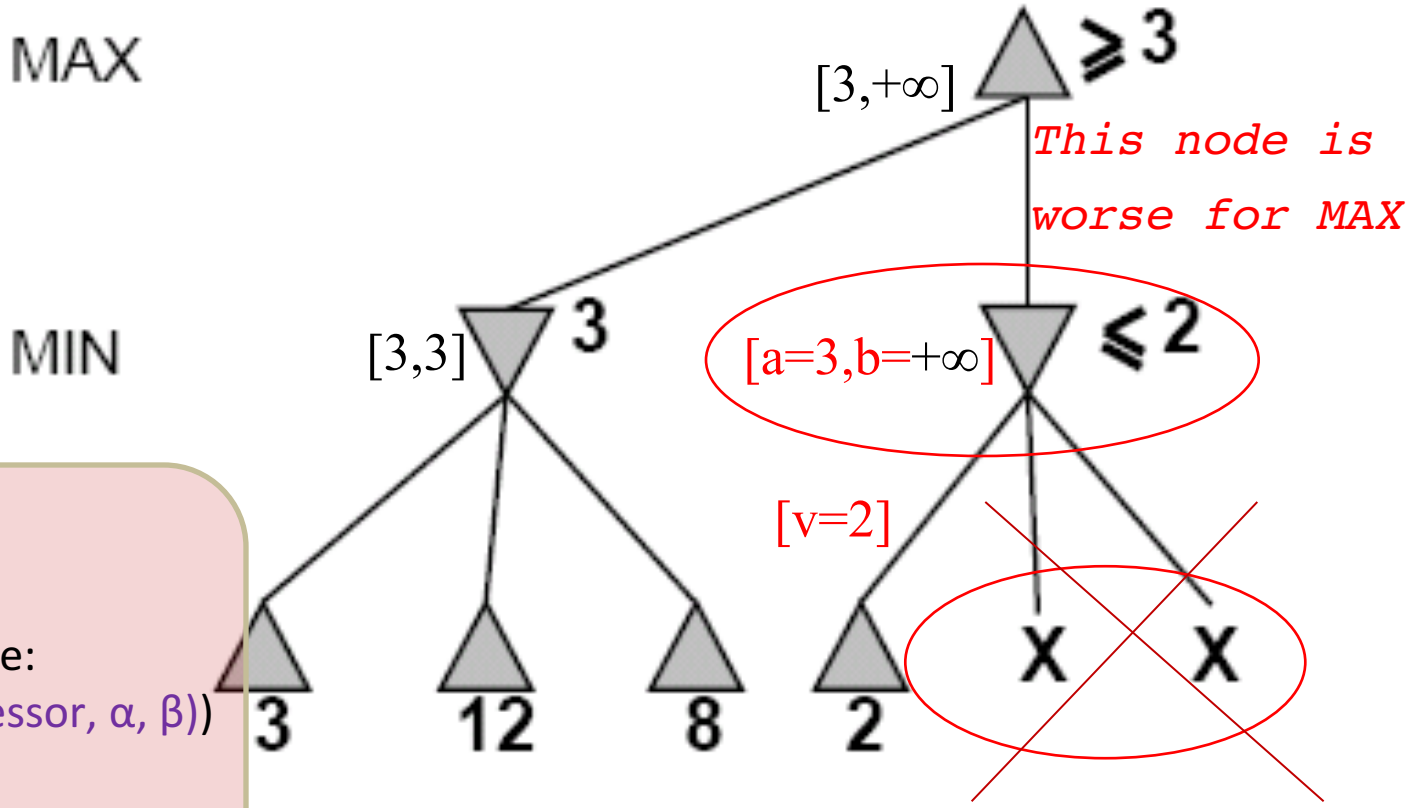

```
f min-value(state , α, β):
  initialize v = +∞
  for each successor of state:
      v = min(v, value(successor, α, β))
      if v ≤ α return v
      β = min(β, v)
  return v
```

Alpha-Beta Example (continued)



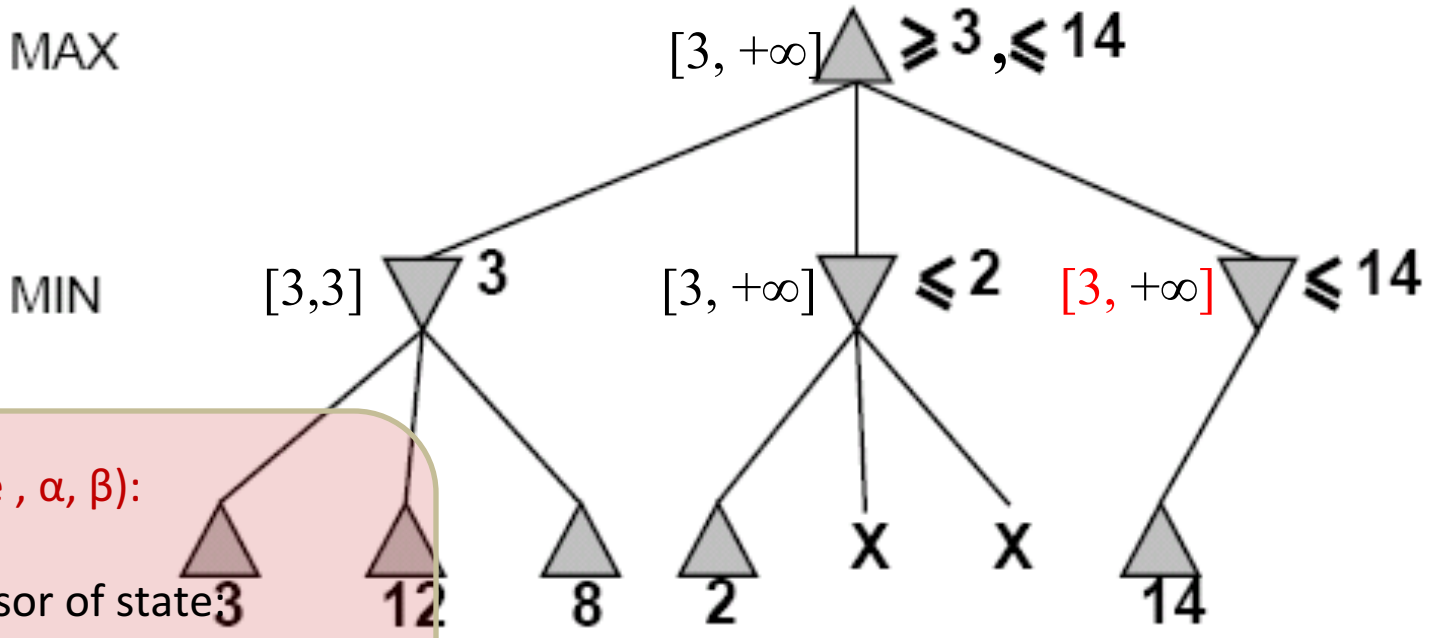

```
f min-value(state , α, β):
  initialize v = +∞
  for each successor of state:
      v = min(v, value(successor, α, β))
      if v ≤ α return v
      β = min(β, v)
  return v
```

Alpha-Beta Example (continued)



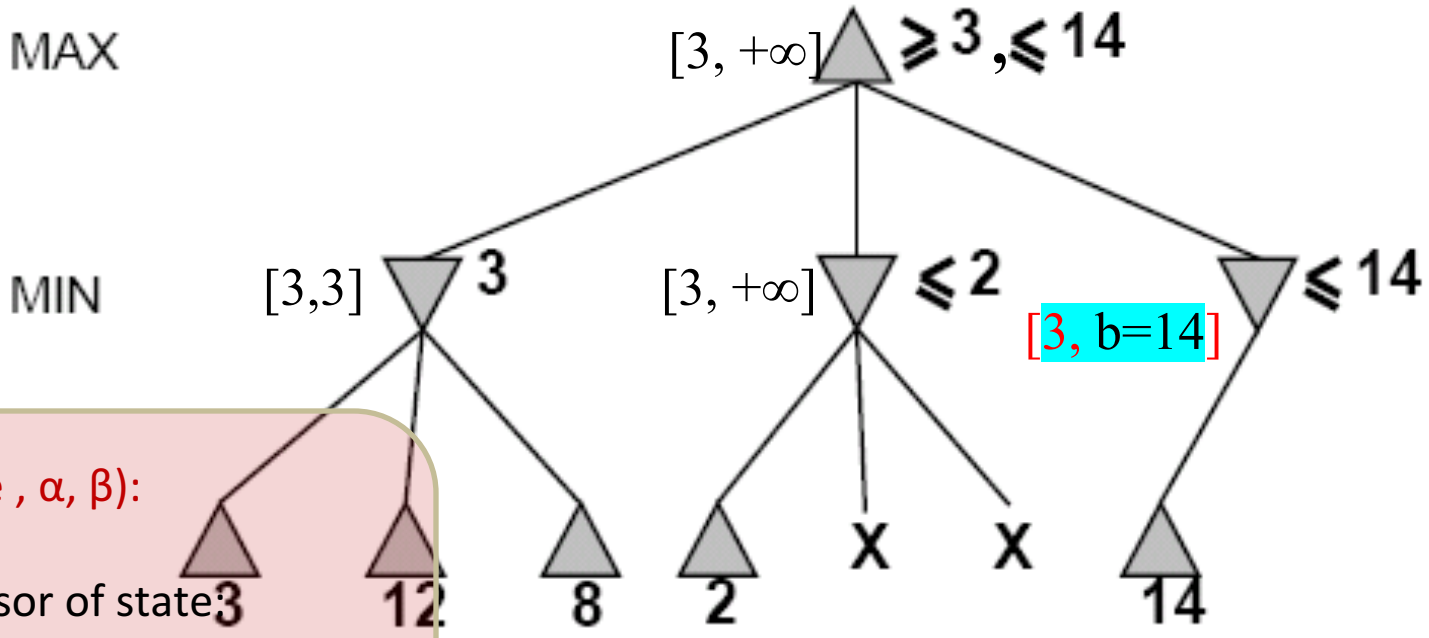

```
f min-value(state , α, β):
  initialize v = +∞
  for each successor of state:
      v = min(v, value(successor, α, β))
      if v ≤ α return v
      β = min(β, v)
  return v
```

Alpha-Beta Example (continued)



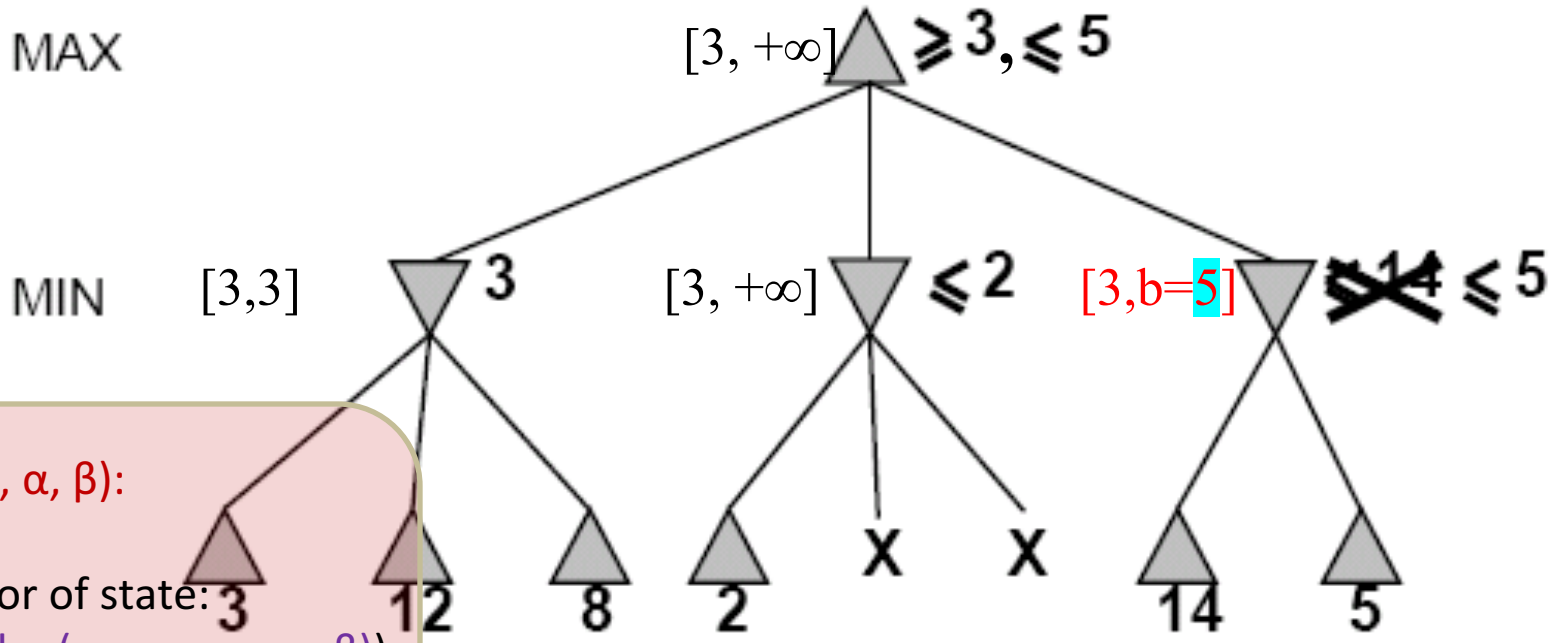

```
f min-value(state , α, β):
  initialize v = +∞
  for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
  return v
```

Alpha-Beta Example (continued)



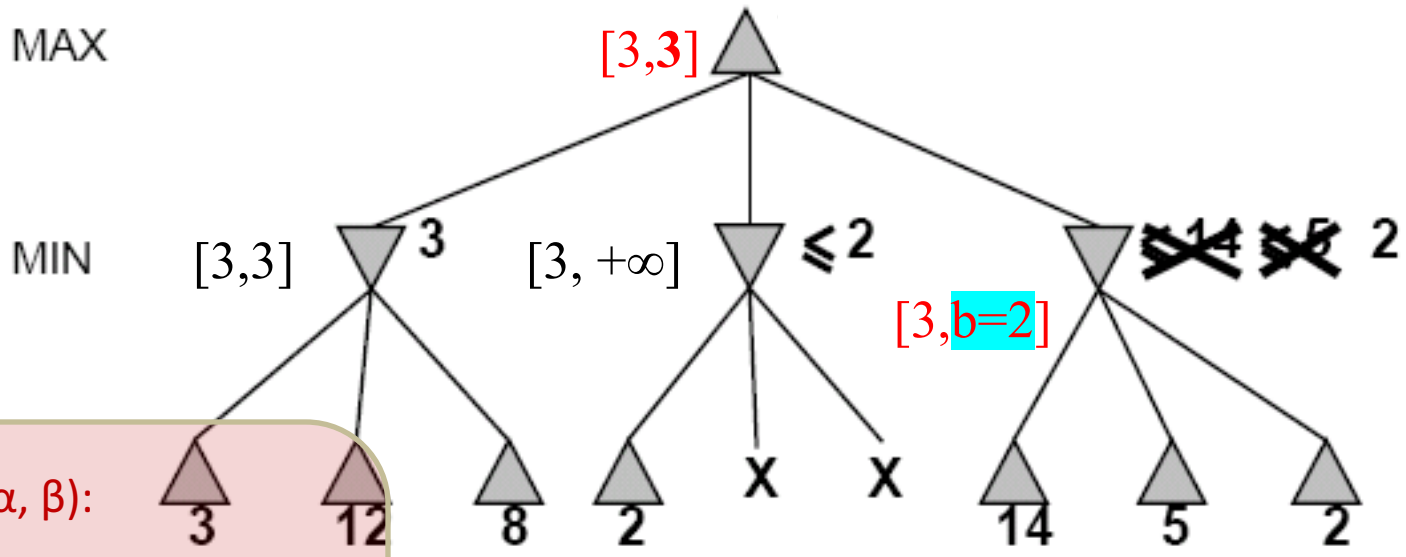

```
f min-value(state , α, β):
  initialize v = +∞
  for each successor of state:
      v = min(v, value(successor, α, β))
      if v ≤ α return v
      β = min(β, v)
  return v
```

Alpha-Beta Example (continued)



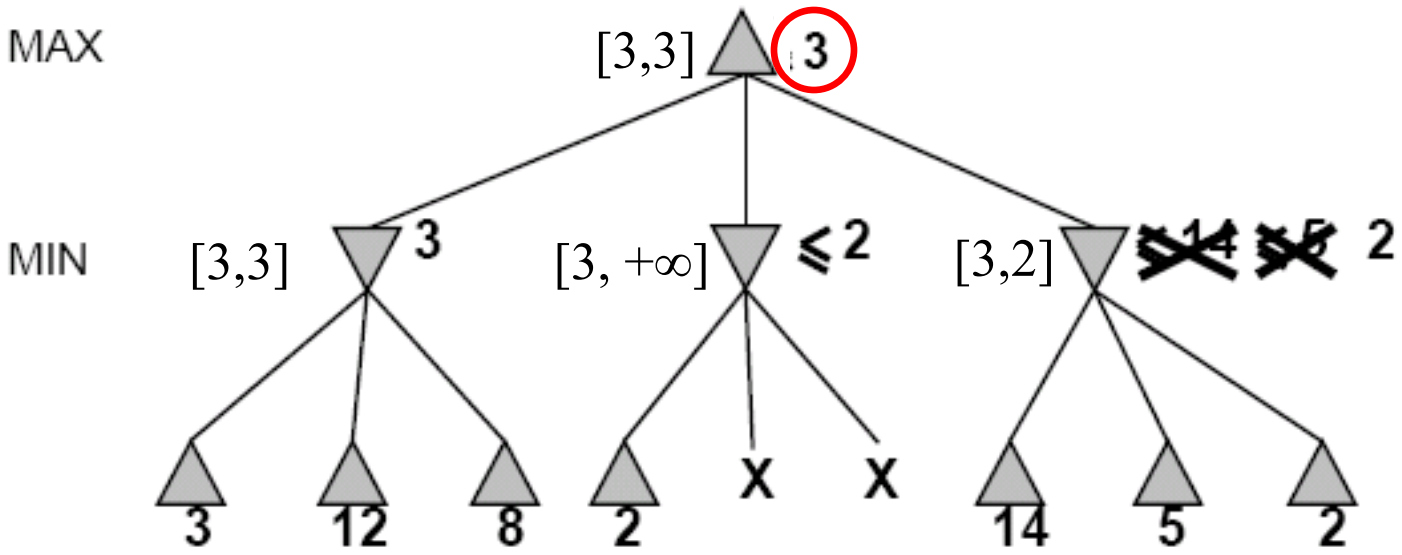

```
min-value(state , α, β):
initialize v = +∞
for each successor of state:
    v = min(v, value(successor, α, β))
    if v ≤ α return v
    β = min(β, v)
return v
```

Alpha-Beta Example (continued)

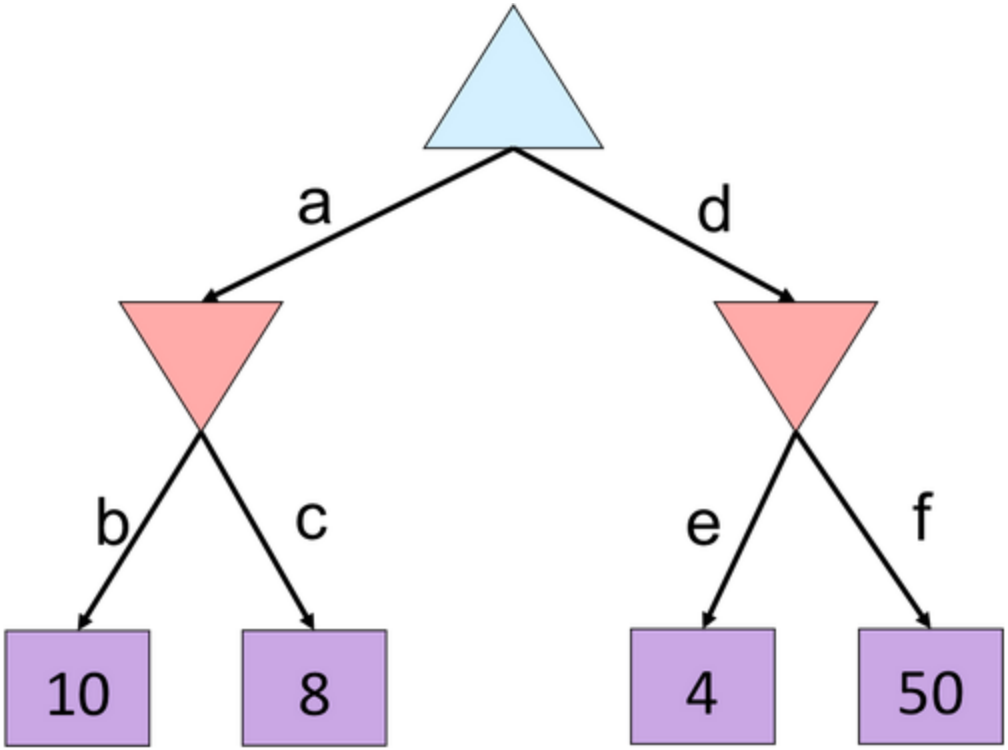



```
min-value(state , α, β):
initialize v = +∞
for each successor of state:
    v = min(v, value(successor, α, β))
    if v ≤ α return v
    β = min(β, v)
return v
```

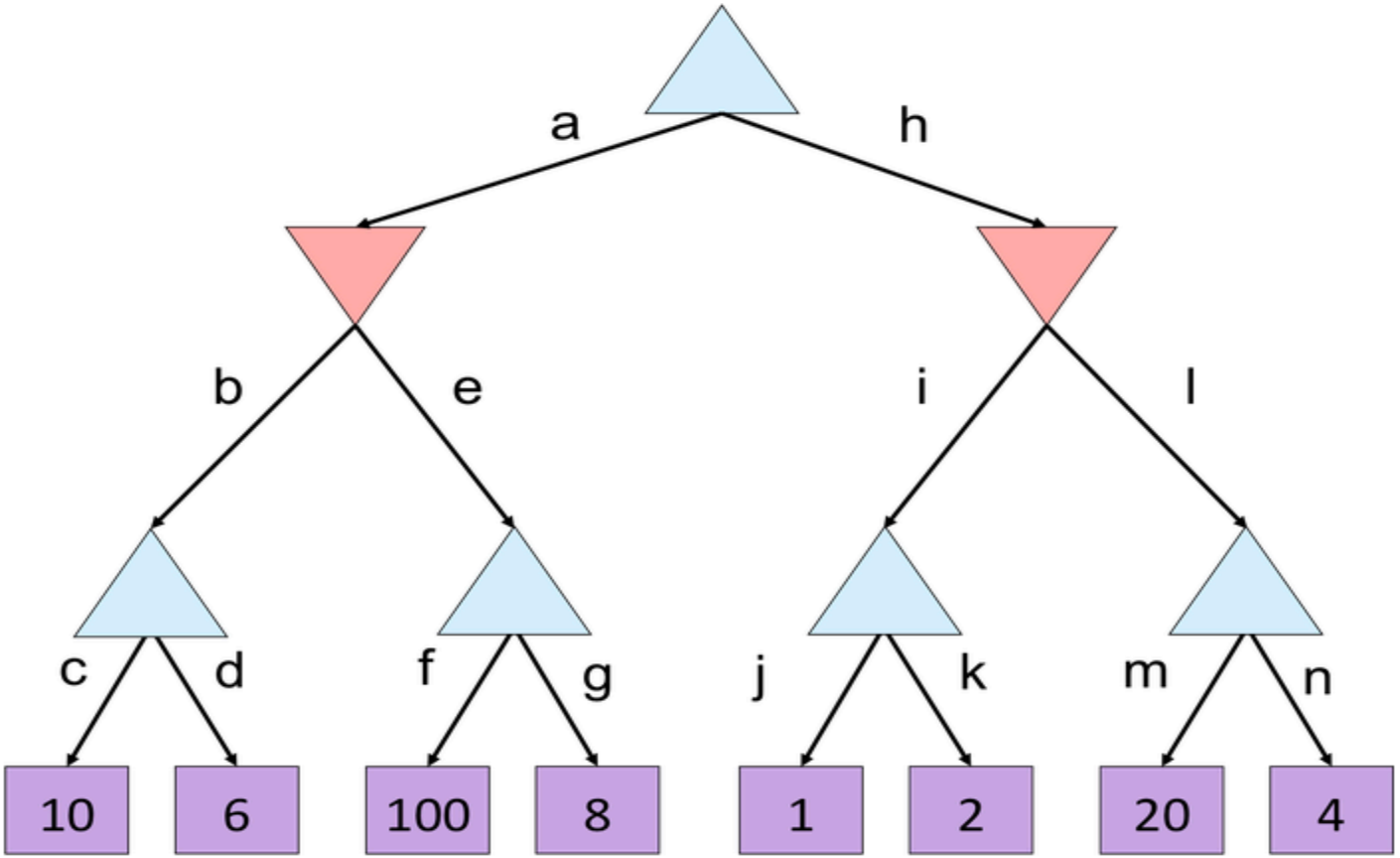
Alpha-Beta Example (continued)



Alpha-Beta Quiz: which nodes pruned?



Alpha-Beta Quiz 2: which nodes pruned?



More Alpha-Beta Pruning Examples

- Generic game tree visualization: http://homepage.ufp.pt/jtorres/ensino/ia/alfabeta.html
- Connect Four: https://gimu.org/connect-four-js/