Artificial Intelligence

Lec 10: Adversarial Search

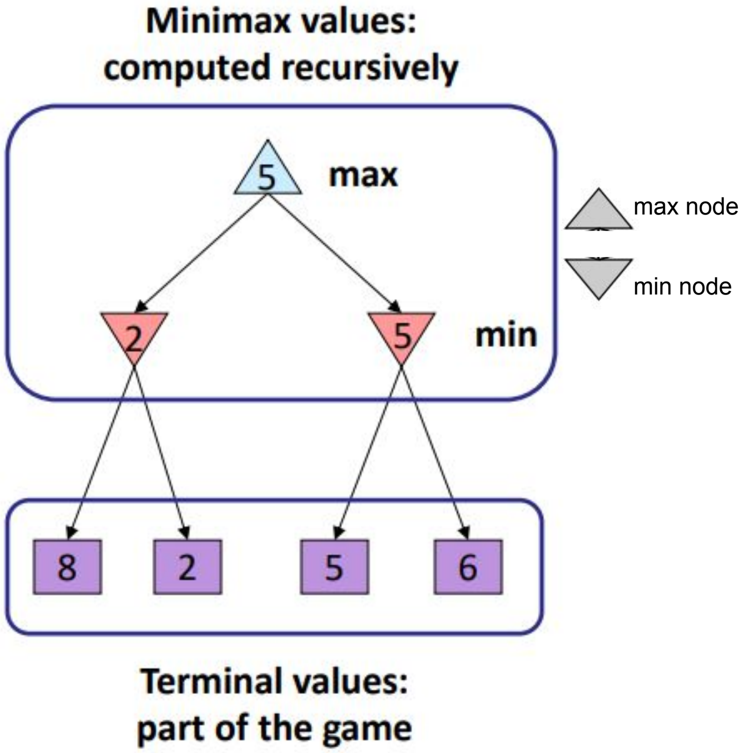
Pratik Mazumder

Content taken from various sources including
CS188 Intro to AI at UC Berkeley

Adversarial Search (Minimax)

- 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
5
max
2
5
min
8
2
5
6
Terminal values: part of the game
max node
min node

Adversarial Search (Minimax)

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

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

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

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

Adversarial Search (Minimax)

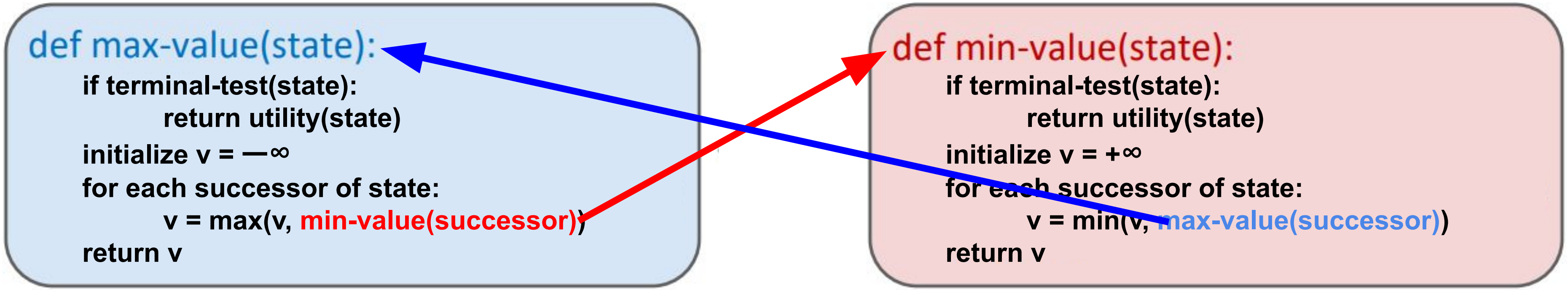
```
def max-value(state):
    if terminal-test(state):
        return utility(state)
    initialize v = —∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

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

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

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

Adversarial Search (Minimax)



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

```
def min-value(state):
    if terminal-test(state):
        return utility(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 Terminology

- move: a move by both players

- ply: a half move, i.e., action by one player

- backed-up value
  - Of a max position: the value of its largest successor
  - Of a min position: the value of its smallest successor

- Minimax procedure:
  - Search down several levels.
  - At the bottom level, apply the utility function.
  - Back-up values all the way up to the root node
  - Select a move starting from the root node [if you perform the first move of the game].

Minimax Example

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

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

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

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

▲ max node

▼ min node

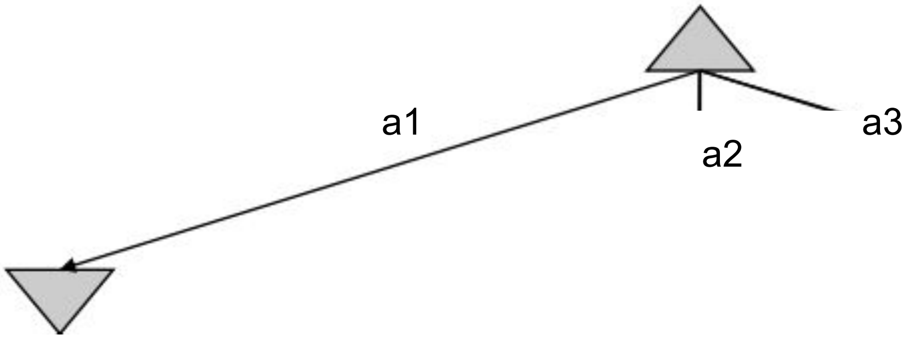
Minimax Example

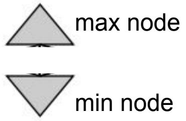
```
def max-value(state):
    if terminal-test(state):
        return utility(state)
    initialize v = −∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

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

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

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





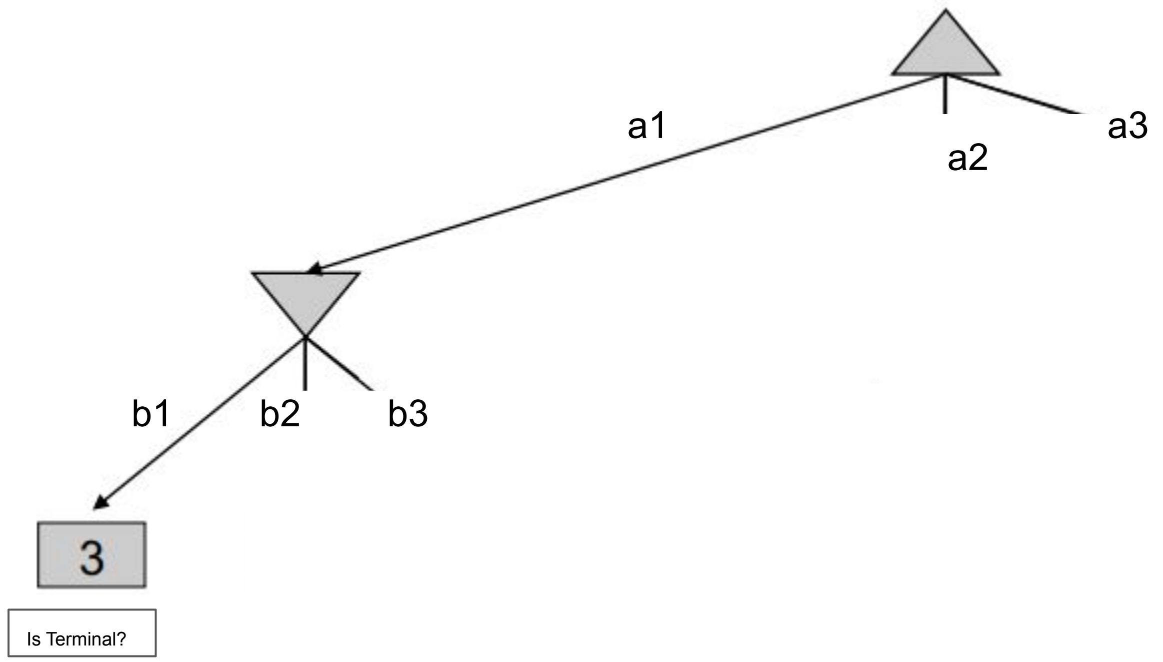
Minimax Example

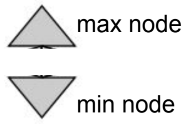
```
def max-value(state):
    if terminal-test(state):
        return utility(state)
    initialize v = —∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

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

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

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





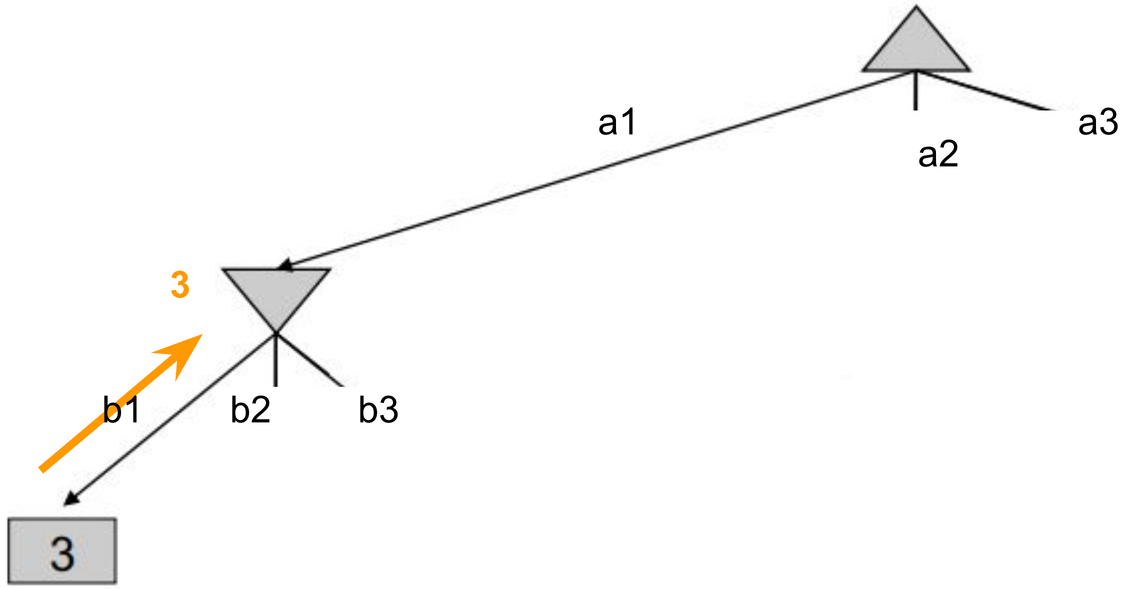
Minimax Example

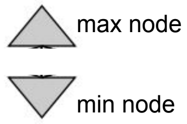
```
def max-value(state):
    if terminal-test(state):
        return utility(state)
    initialize v = −∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

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

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

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





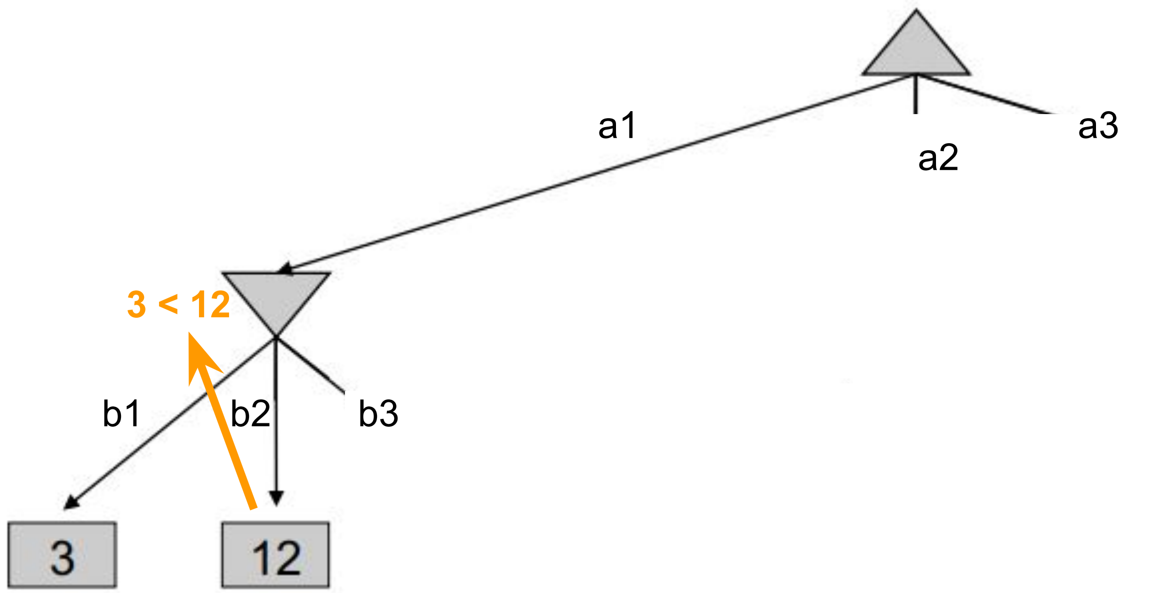
Minimax Example

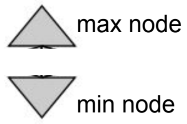
```
def max-value(state):
    if terminal-test(state):
        return utility(state)
    initialize v = —∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

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

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

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





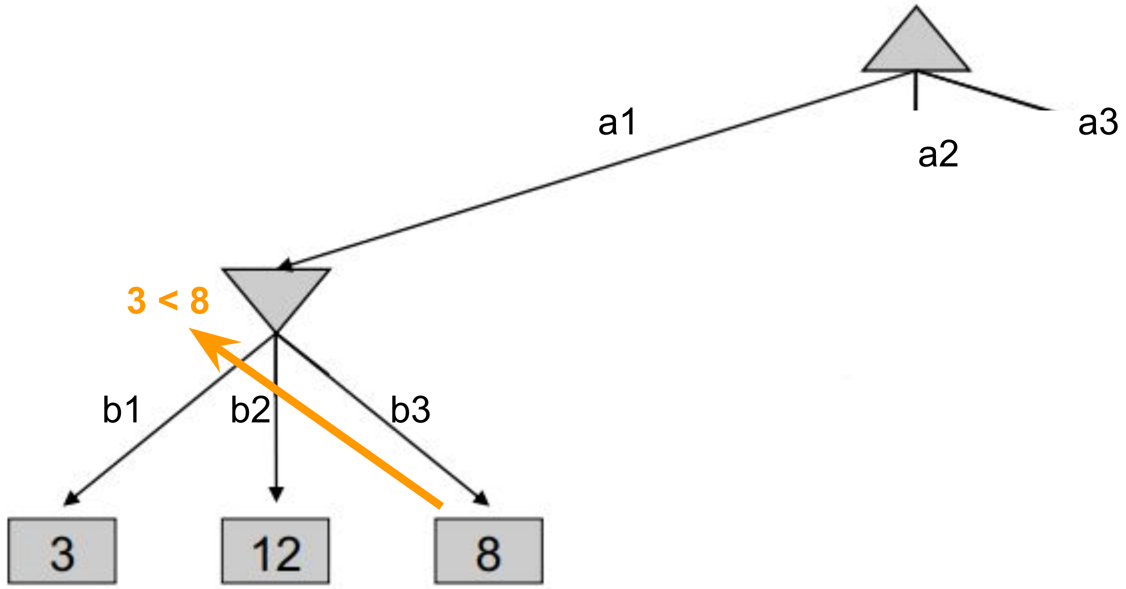
Minimax Example

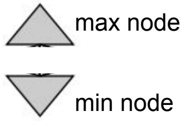
```
def max-value(state):
        if terminal-test(state):
                return utility(state)
        initialize v = —∞
        for each successor of state:
                v = max(v, min-value(successor))
        return v
```

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

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

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





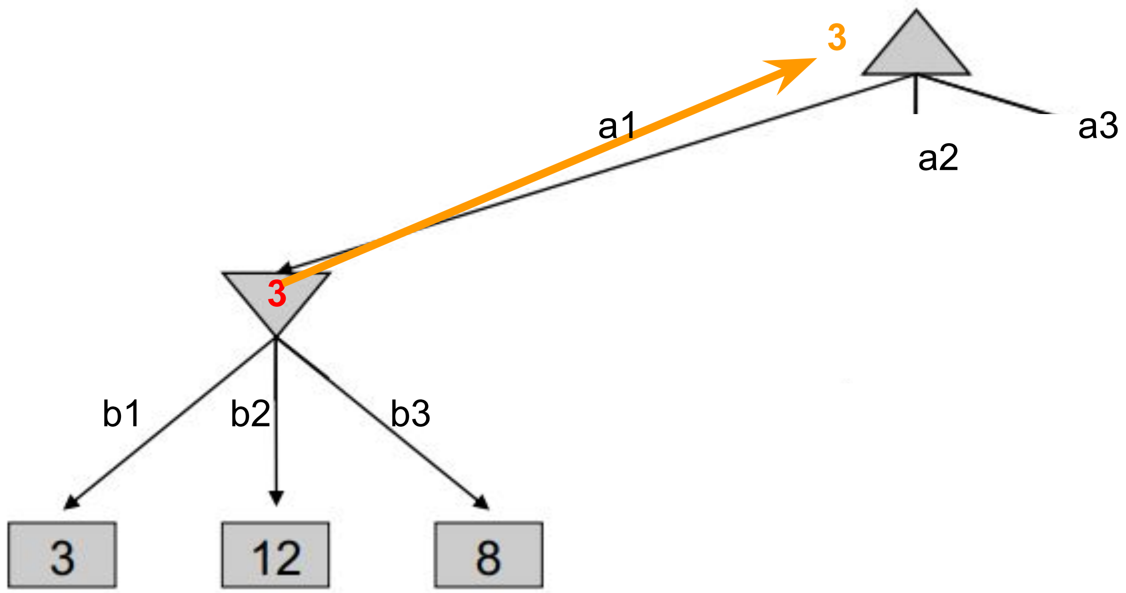
Minimax Example

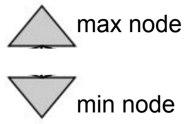
```
def max-value(state):
    if terminal-test(state):
        return utility(state)
    initialize v = —∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

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

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

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





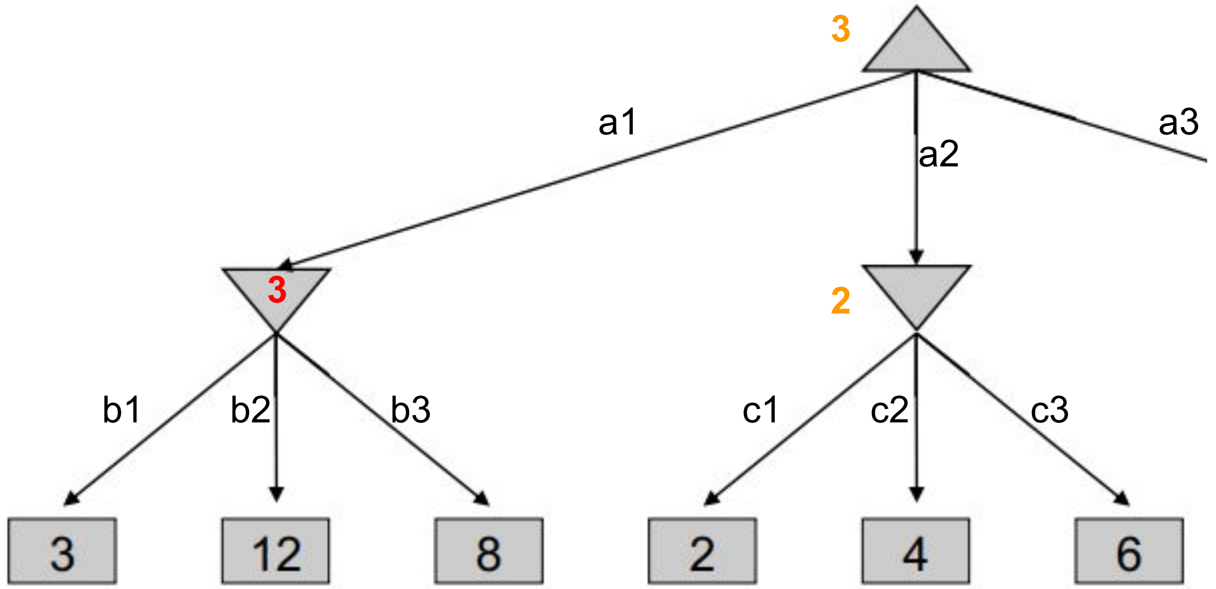
Minimax Example

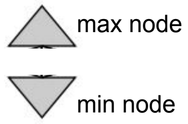
```
def max-value(state):
    if terminal-test(state):
        return utility(state)
    initialize v = —∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

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

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

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





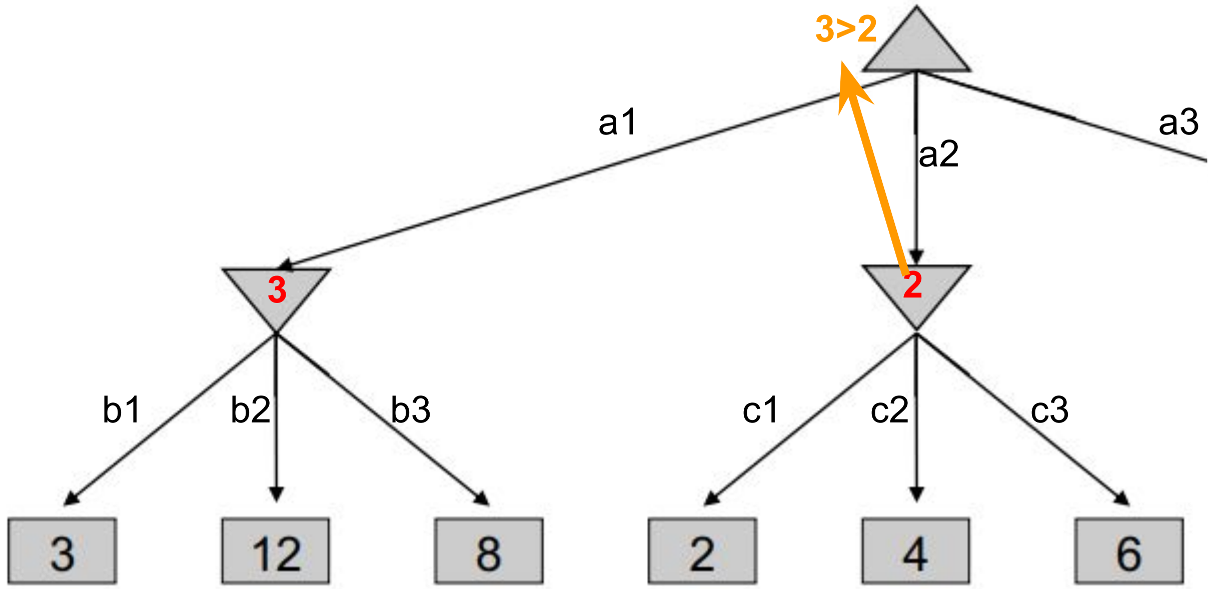
Minimax Example

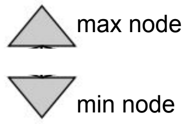
```
def max-value(state):
    if terminal-test(state):
        return utility(state)
    initialize v = —∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

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

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

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





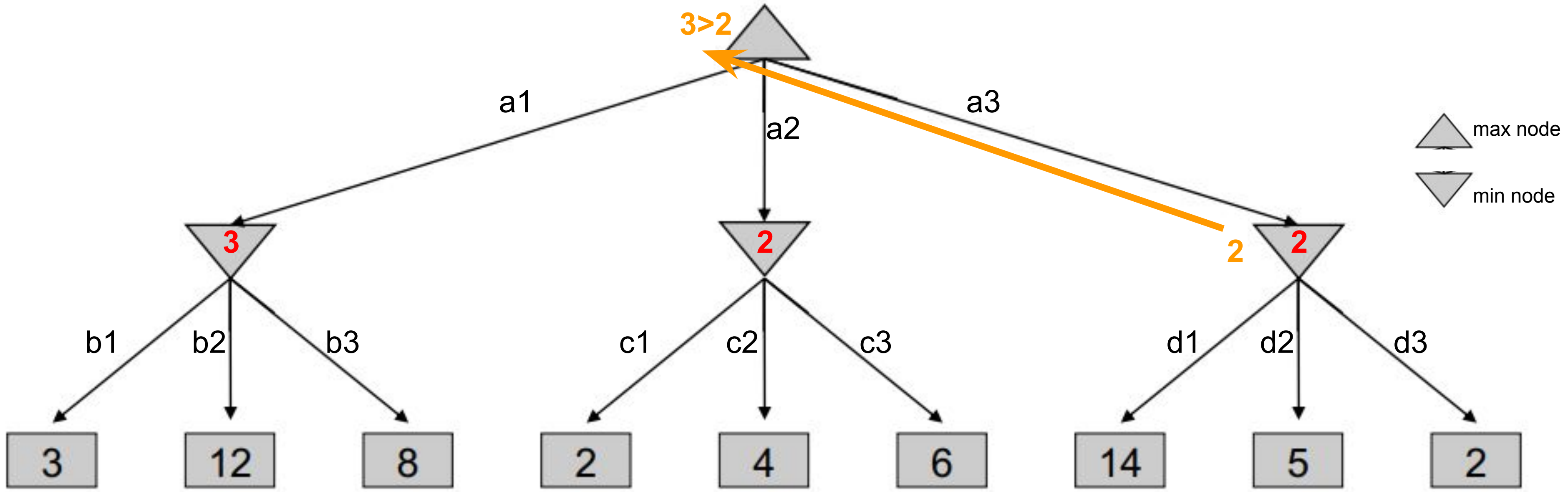
Minimax Example

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

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

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

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



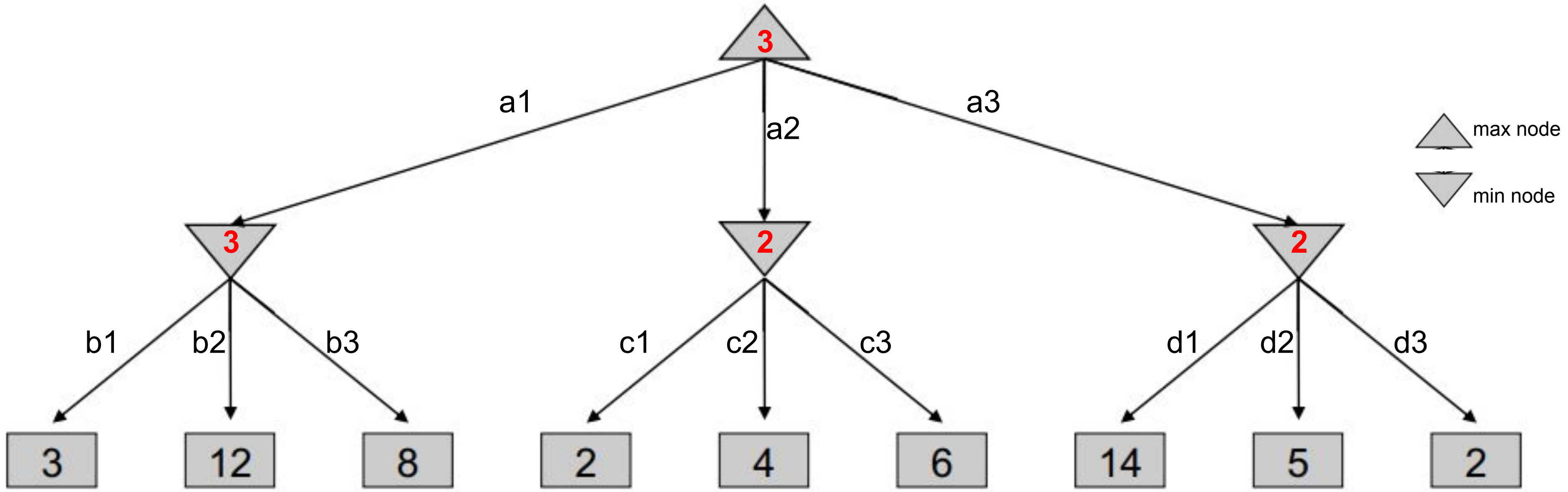
Minimax Example

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

```
def min-value(state):
    if terminal-test(state):
        return utility(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 Example

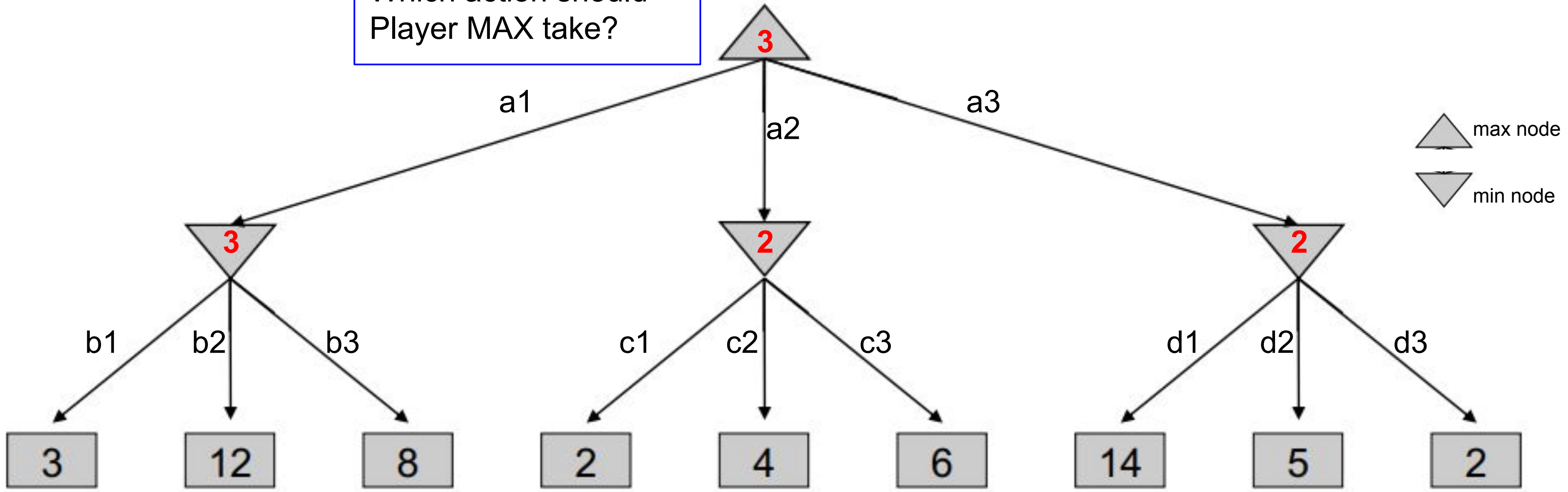
```
def max-value(state):
    if terminal-test(state):
        return utility(state)
    initialize v = —∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

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

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

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

Which action should Player MAX take?



Minimax Example

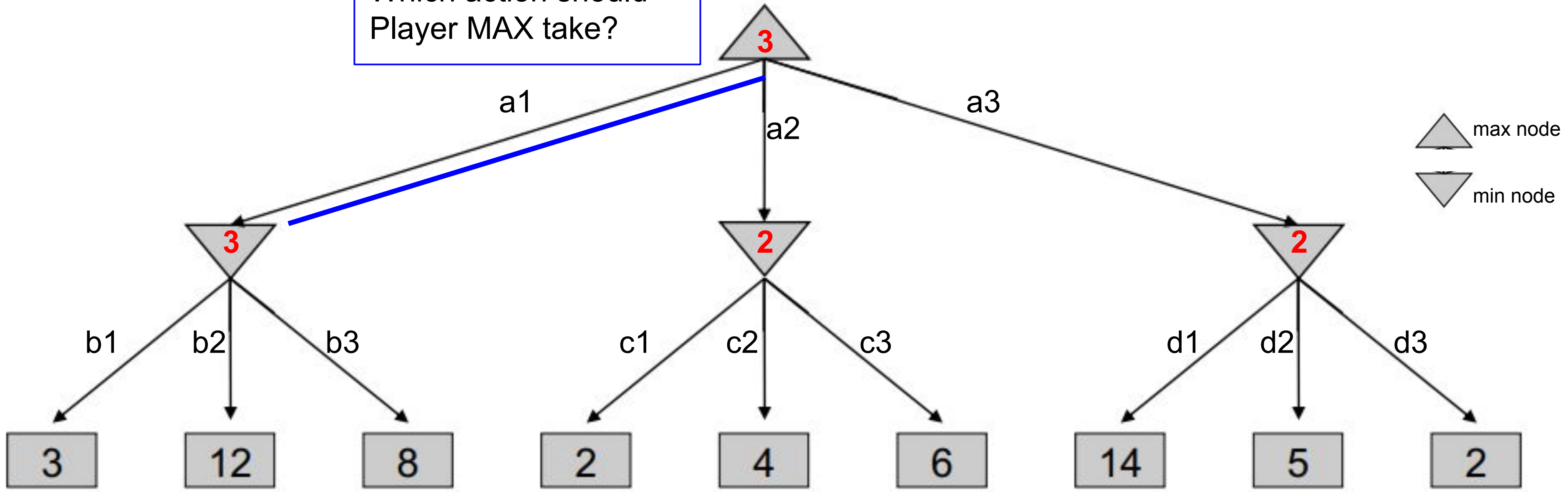
Which action should Player MAX take?

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

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

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

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



Minimax Example

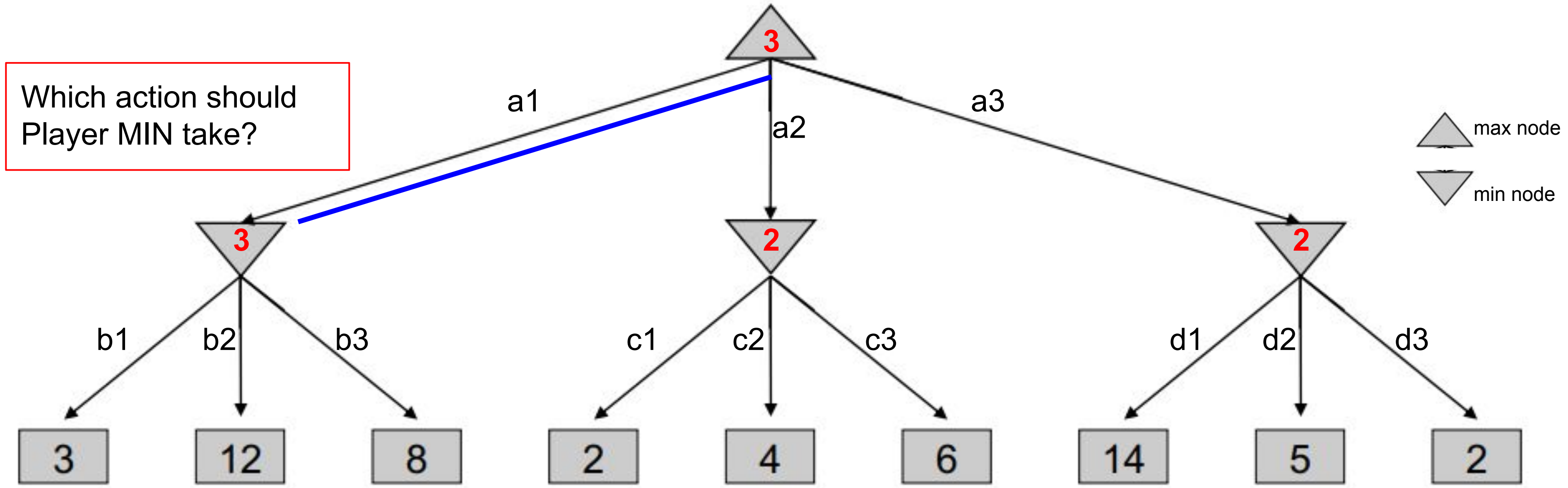
```
def max-value(state):
    if terminal-test(state):
        return utility(state)
    initialize v = —∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

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

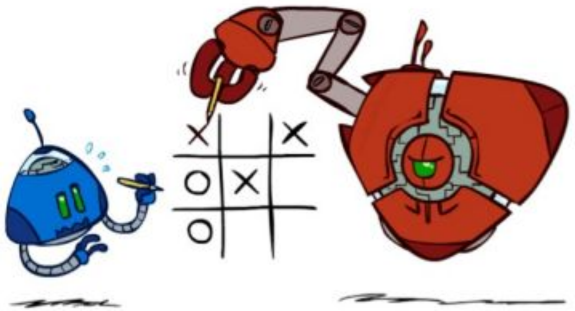
```
def min-value(state):
    if terminal-test(state):
        return utility(state)
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

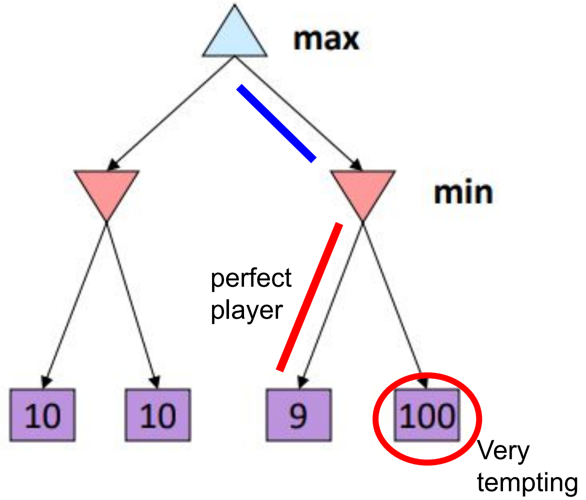
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Which action should Player MIN take?



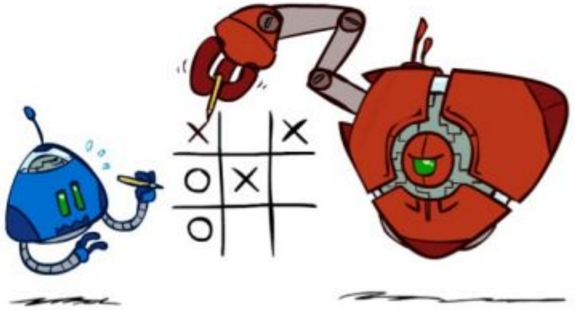
Minimax Properties

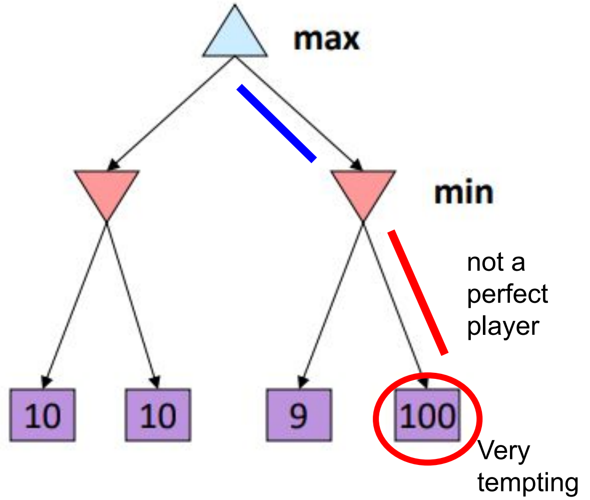


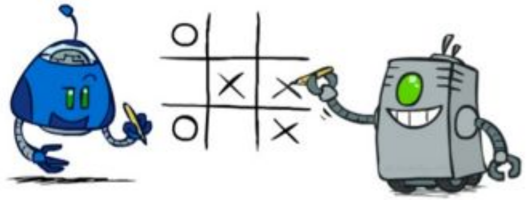


Optimal against a perfect player.

Minimax Properties



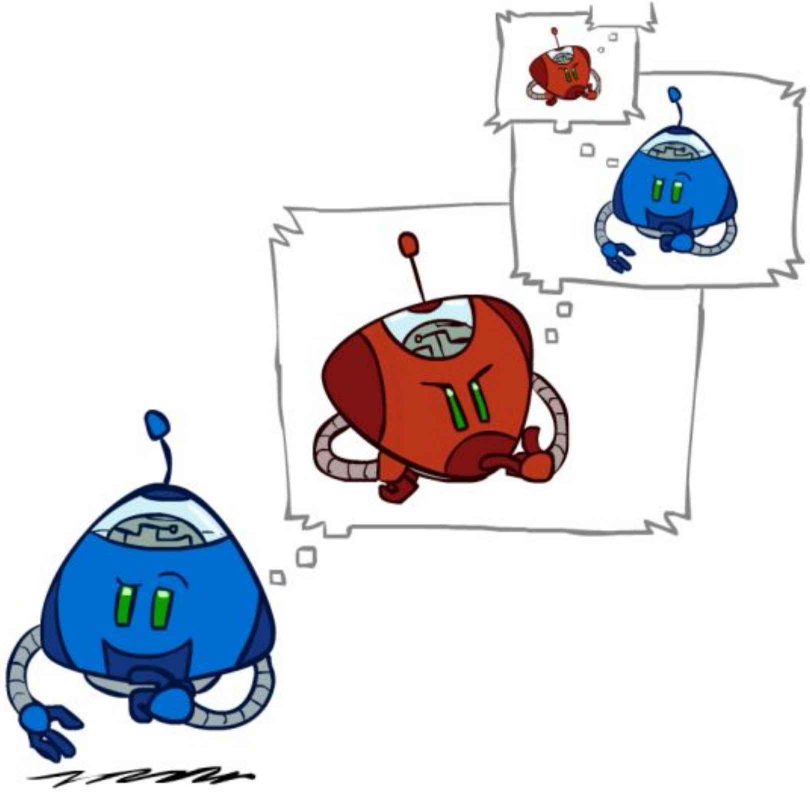




Optimal against a perfect player. Otherwise?

Minimax Efficiency

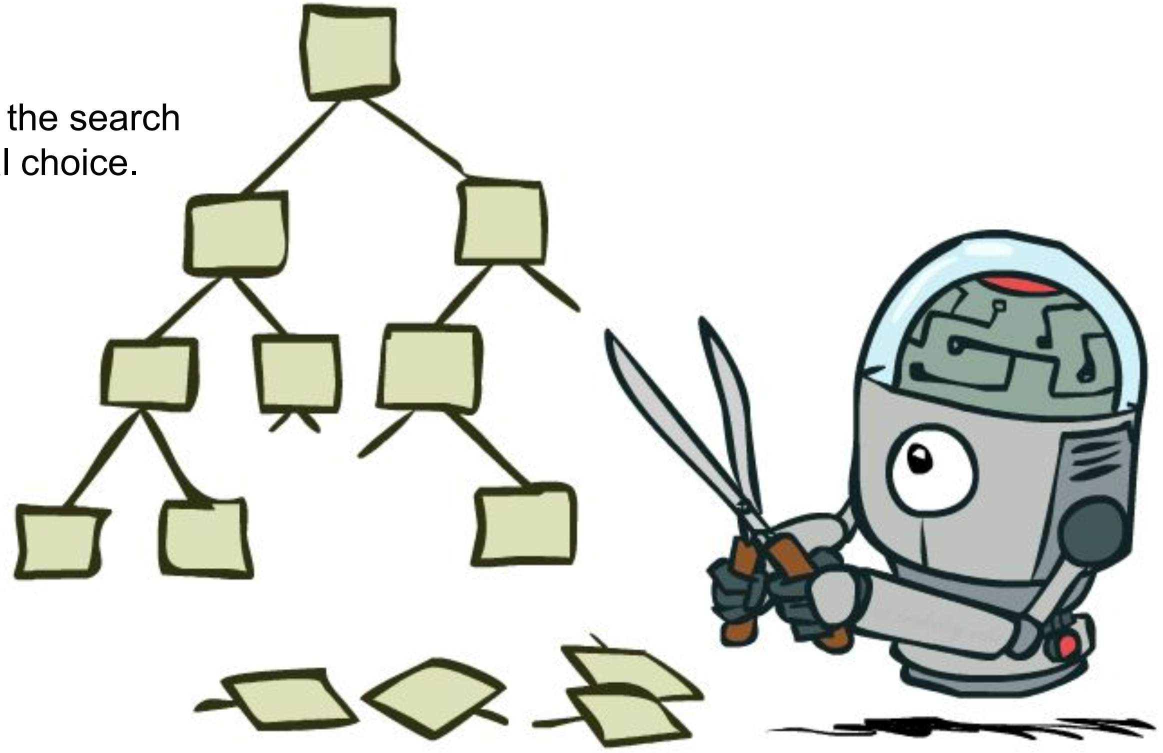
- How efficient is minimax?
  - Just like (exhaustive) DFS
- Example: For chess, branching factor b ≈ 35, solution depth/how many turns the game lasts typically m ≈ 100
  - Search space $b^m = 35^{100} \approx 10^{154}$
- Interesting Analogy: Universe
  - Number of atoms ≈ 10^{78}
  - Age ≈ 10^{18} seconds
  - Avg no of chemical reactions: 10^8/sec
  - 10^8 moves/sec x 10^{78} x 10^{18} = 10^{104}
- For Go, b ≈ 250-300, m ≈ 150
- Exact solution is not very feasible.
- But, do we need to explore the whole tree?



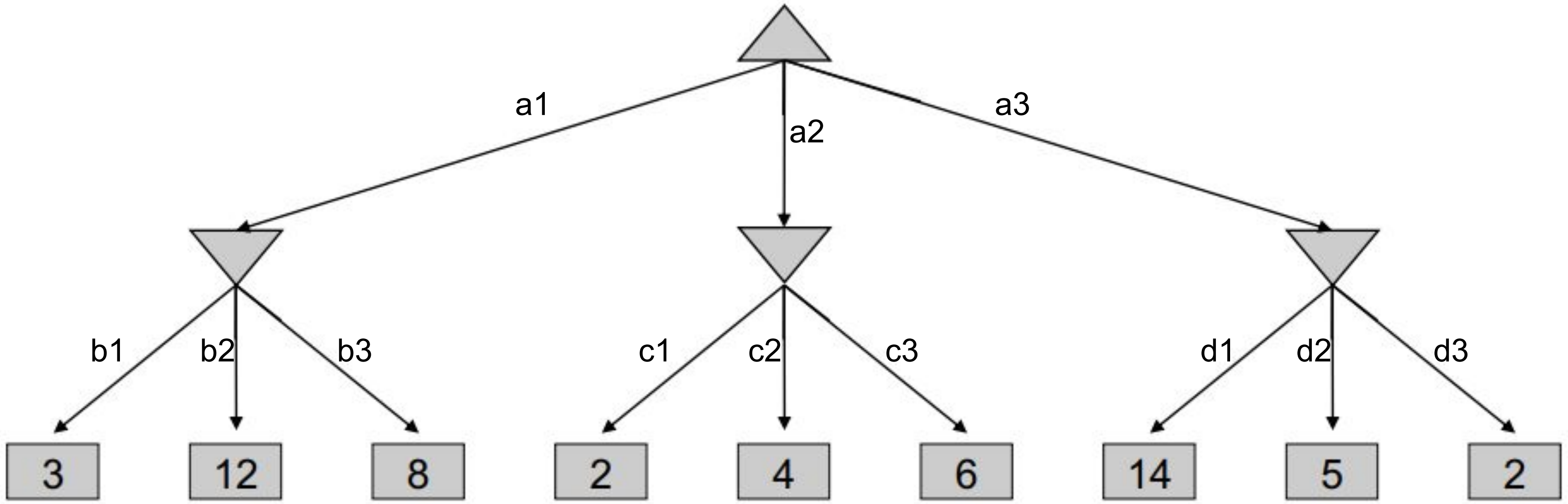
Overcoming Resource Limits

Overcoming Resource Limits: Game Tree Pruning

Pruning allows us to ignore portions of the search tree that make no difference to the final choice.

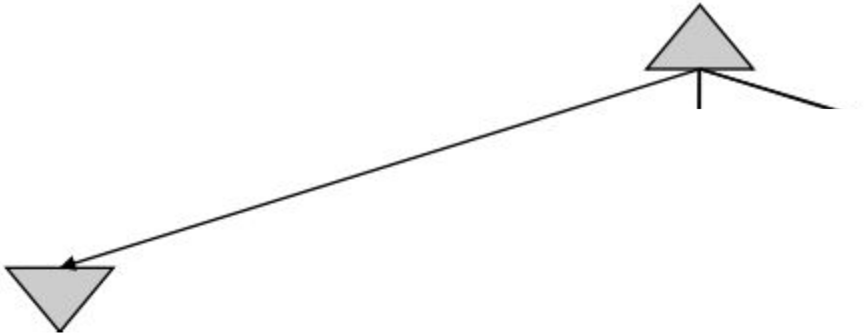


Minimax Example

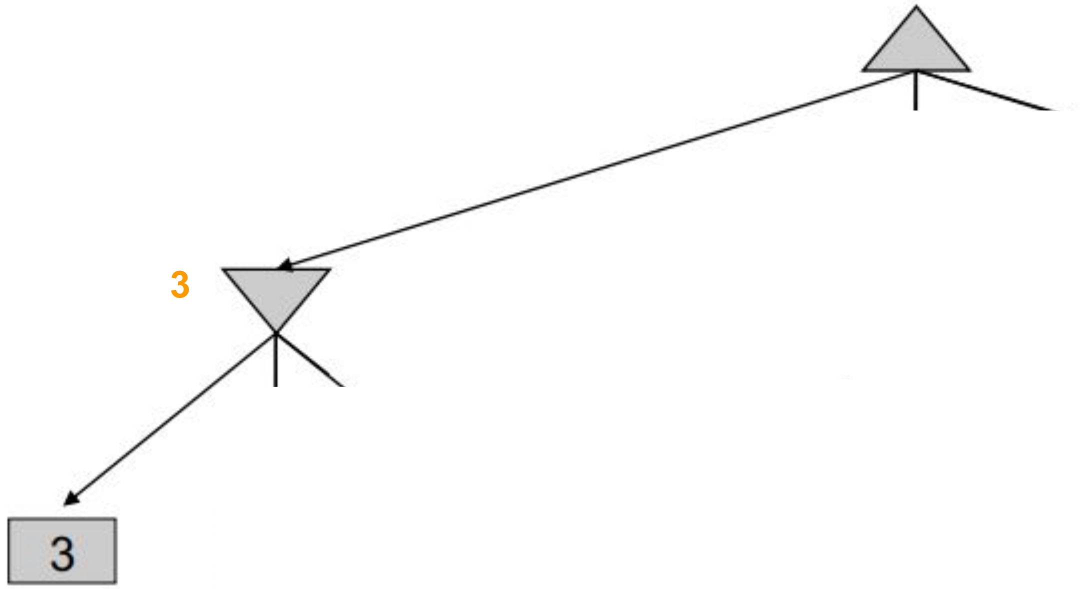


Pruning

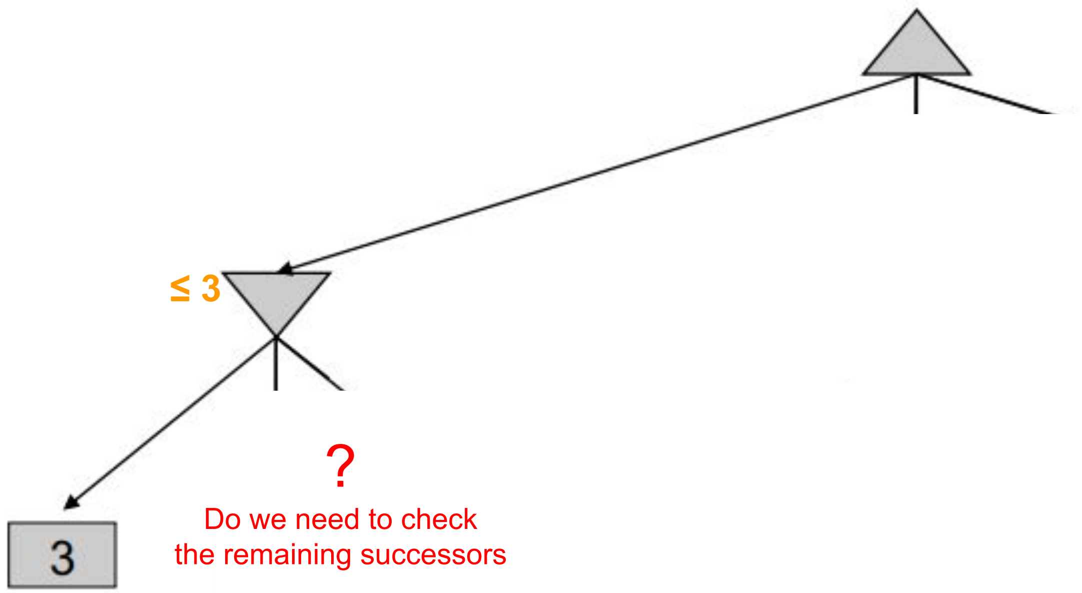
Pruning



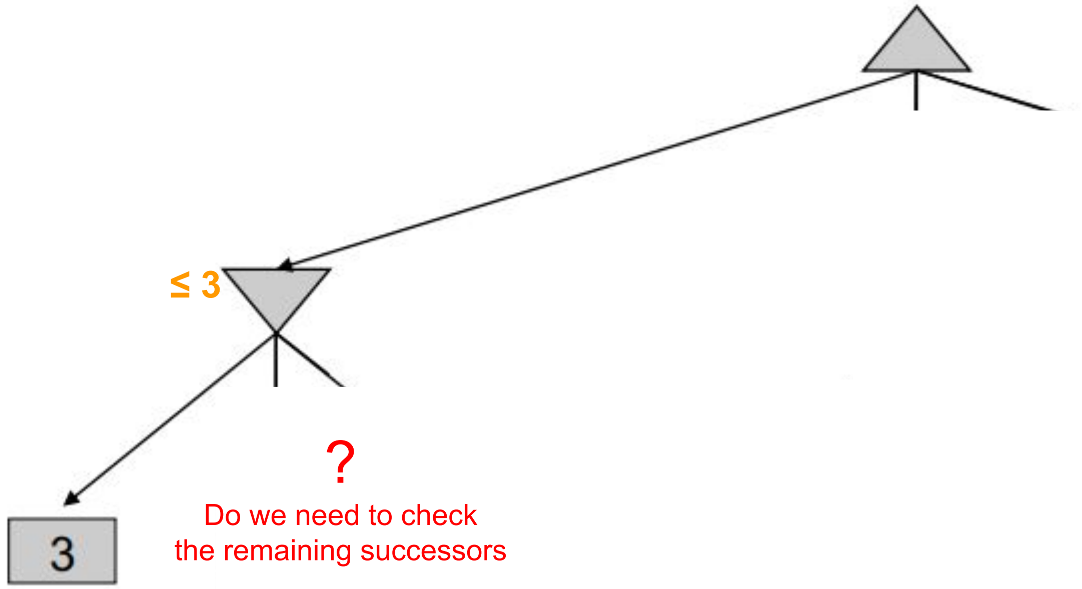
Pruning



Pruning

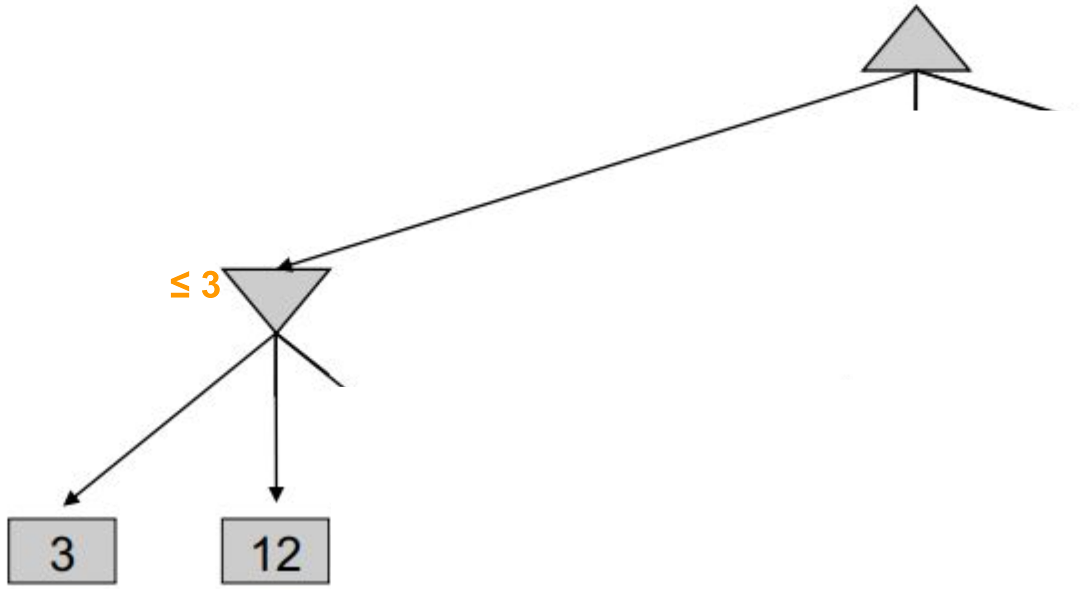


Pruning

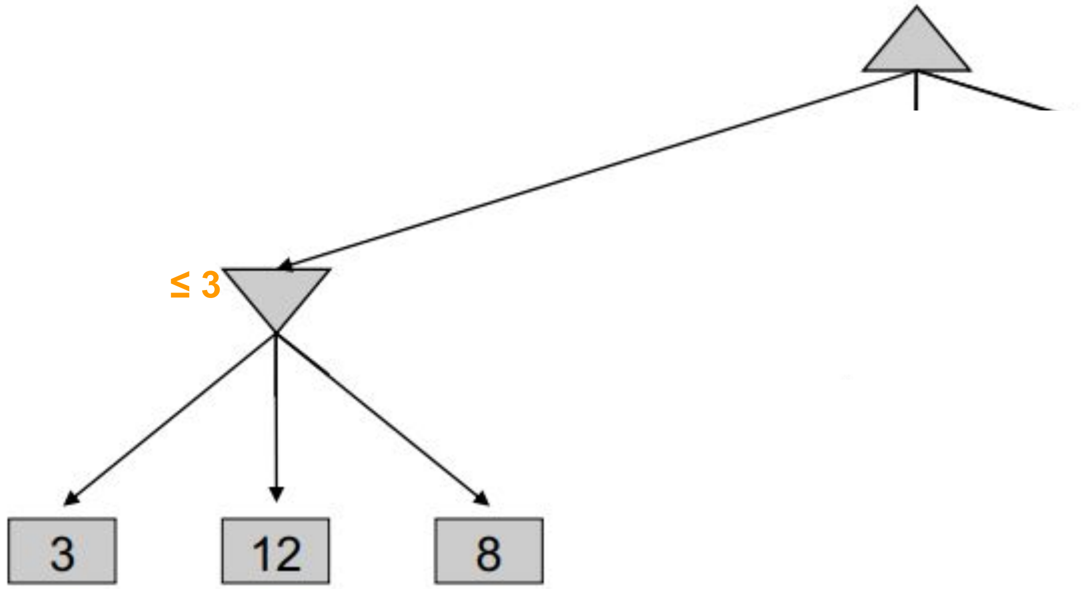


Do we need to check
the remaining successors

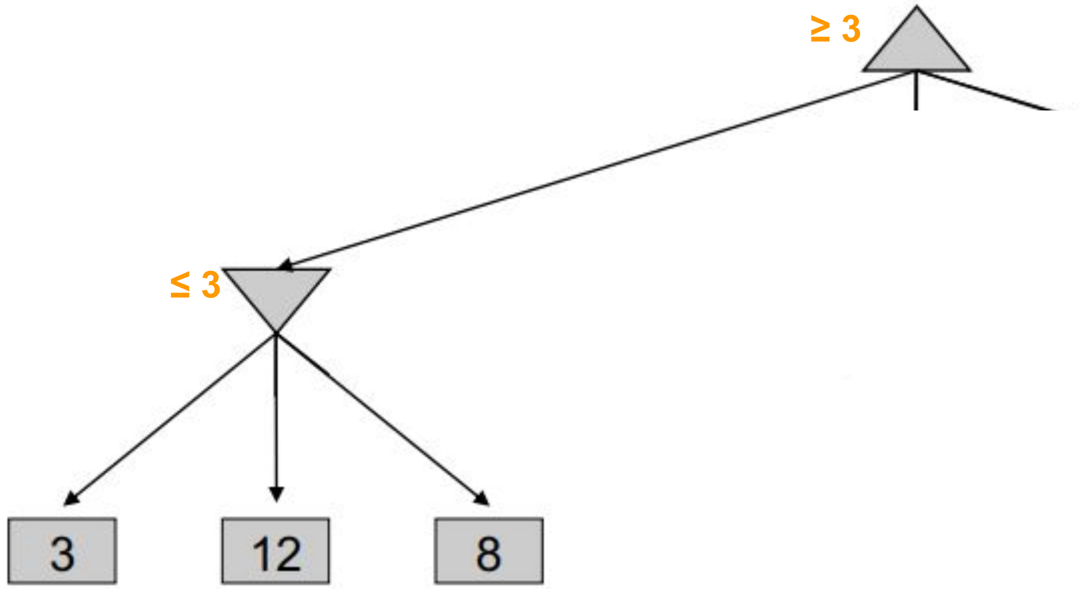
Pruning



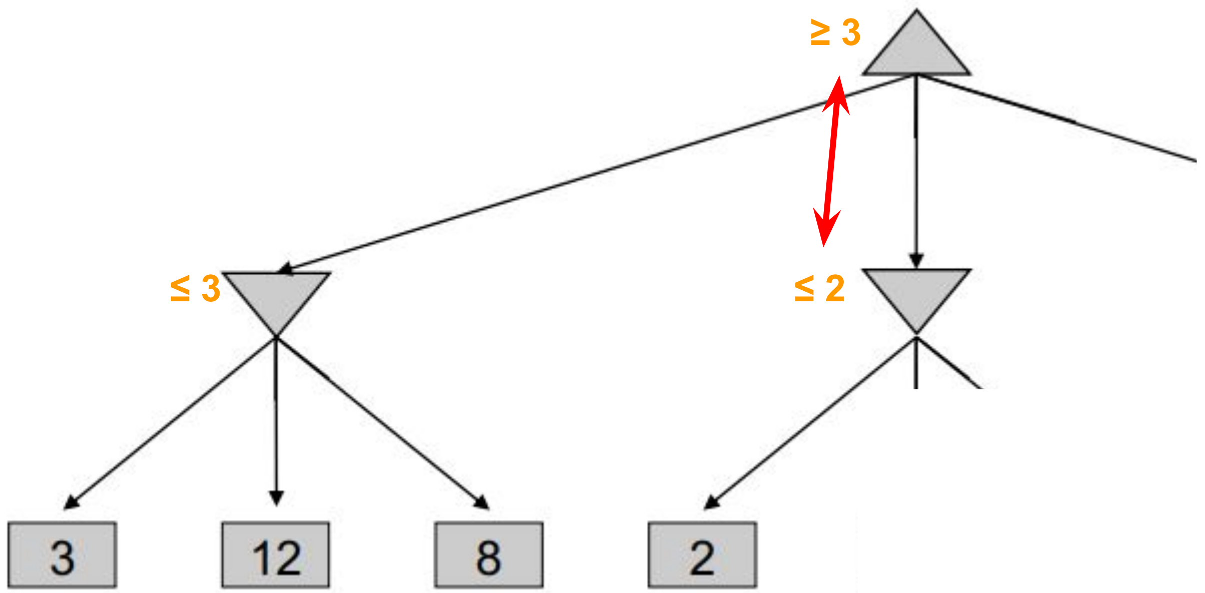
Pruning



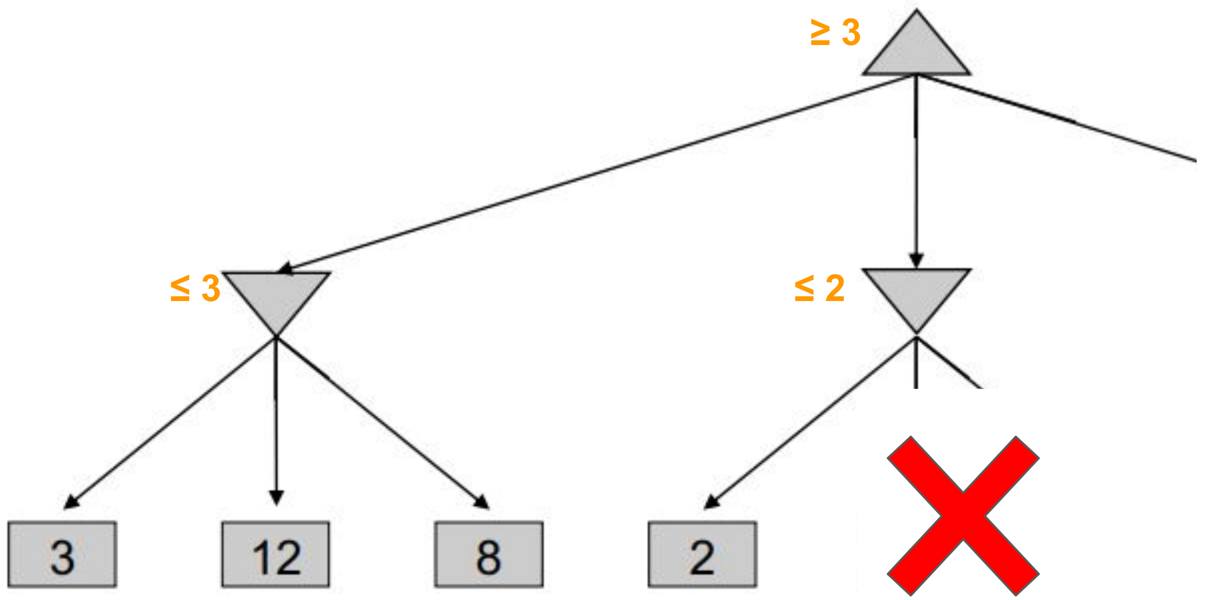
Pruning



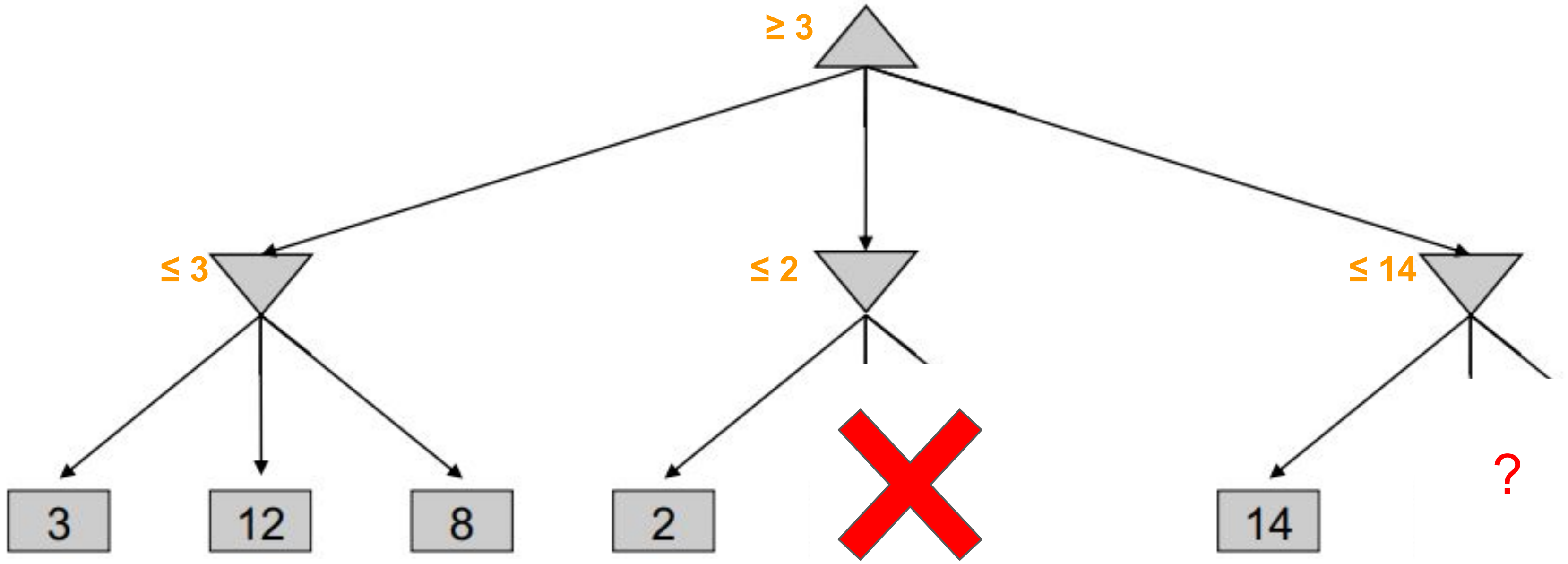
Pruning



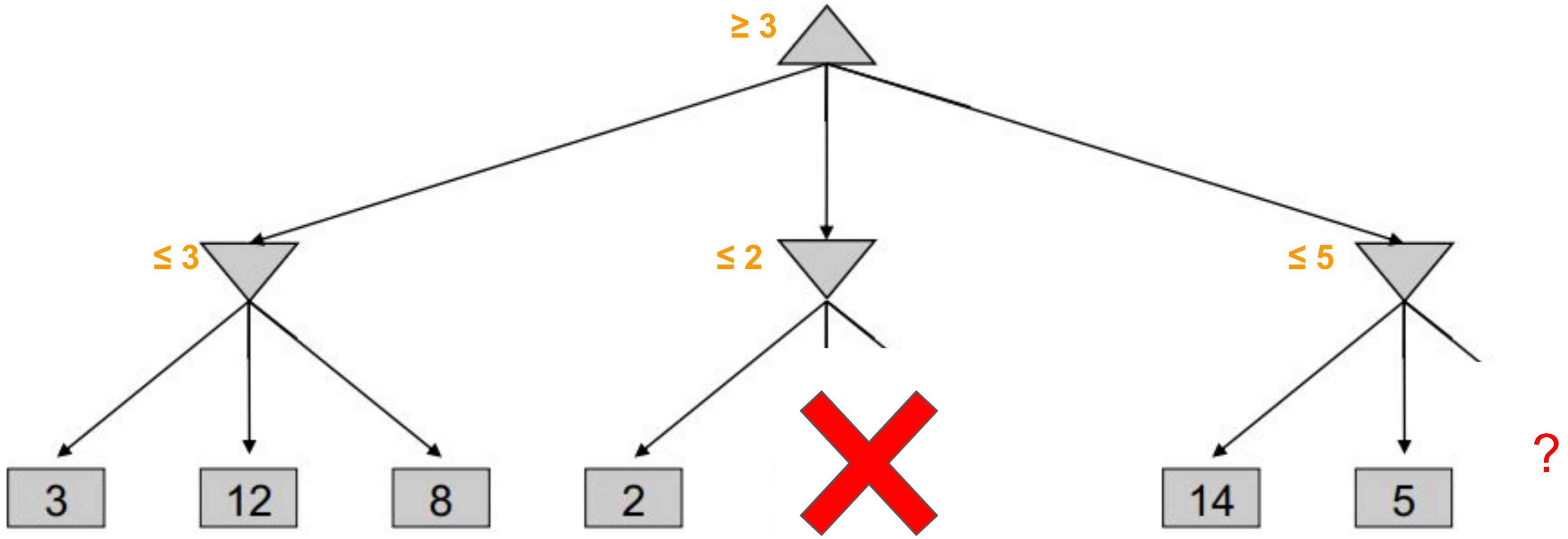
Pruning



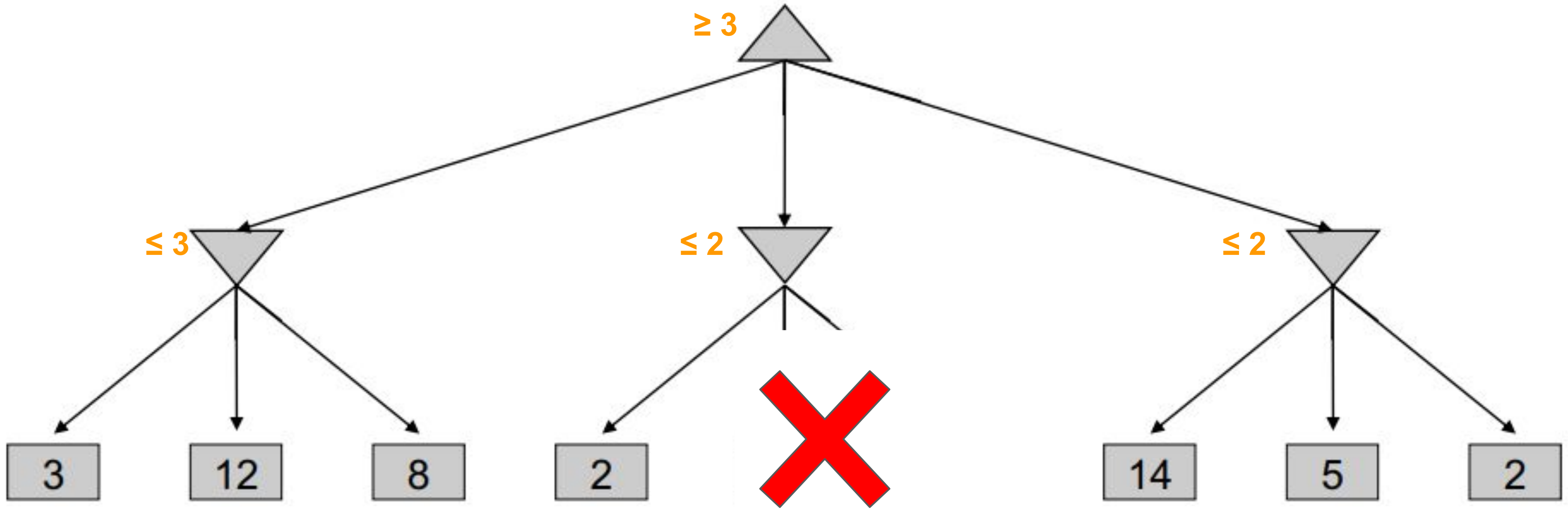
Pruning



Pruning

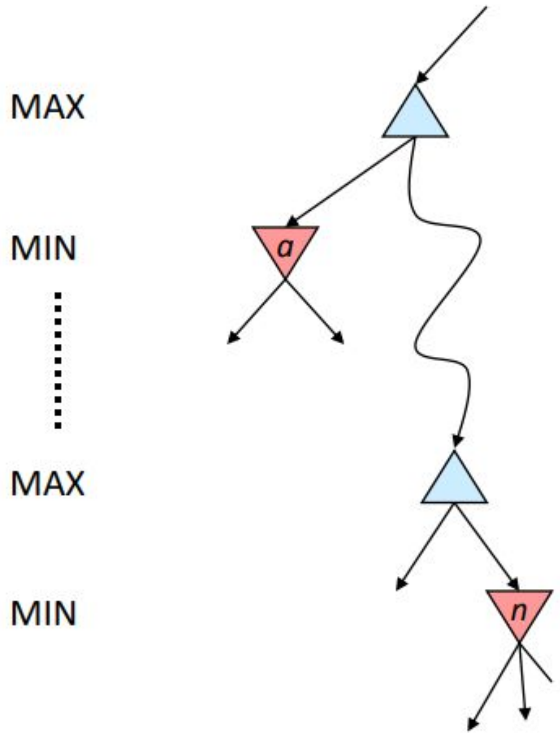


Pruning



Alpha-Beta Pruning

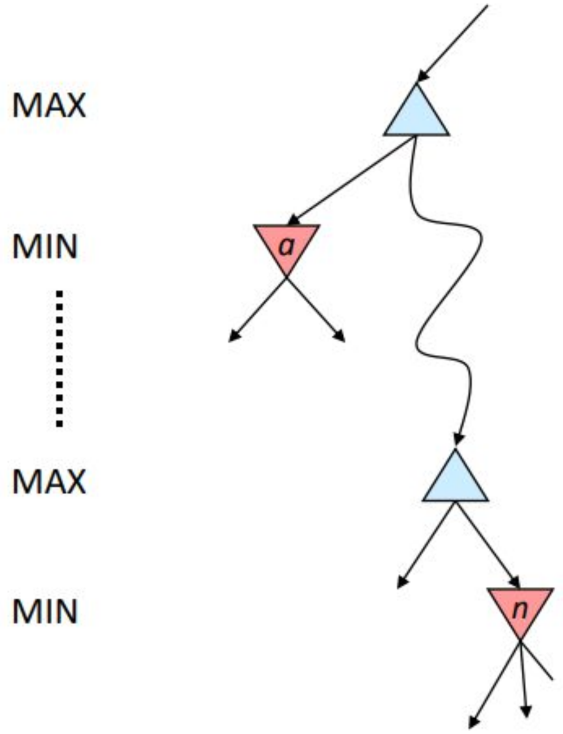
- The **problem with minimax** search is that the **number of game states** it has to examine is **exponential** in the **depth** of the tree.
- We can effectively **cut** the **search** and compute the correct minimax decision **without looking at every node** in the game tree.
- **Alpha–beta pruning** can be applied to trees of any depth, and it is often possible to **prune entire subtrees rather than just leaves**.
- Alpha–beta pruning gets its name from the following two parameters that describe **bounds on the backed-up values** that appear anywhere along the path:
  - **α** = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX .
  - **β** = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN .



Alpha-Beta Pruning

- Alpha–beta search **updates the values of α and β** as it goes along.

- It **prunes** the **remaining branches at a node** (i.e., terminates the recursive call) as soon **as the value** of the current node is known to be **worse than the current α or β** value depending on whether the current node is MAX or MIN.



Alpha-Beta Implementation

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

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

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

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

α: MAX's best option on path to root

β: MIN's best option on path to root

```
def max-value(state, α, β):
    if terminal-test(state):
            return utility(state)
    initialize v = —∞
    for each successor of state:
            v = max(v, min-value(successor, α, □))
            if v ≥ □ return v
            α = max(α,v)
    return v
```

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

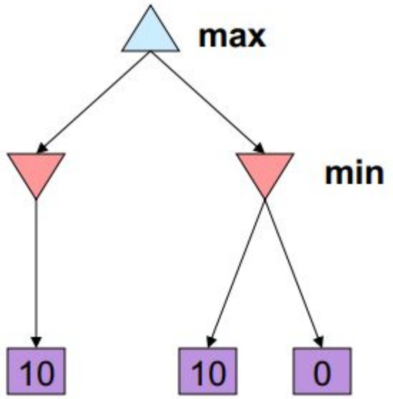
Alpha-Beta Pruning Properties

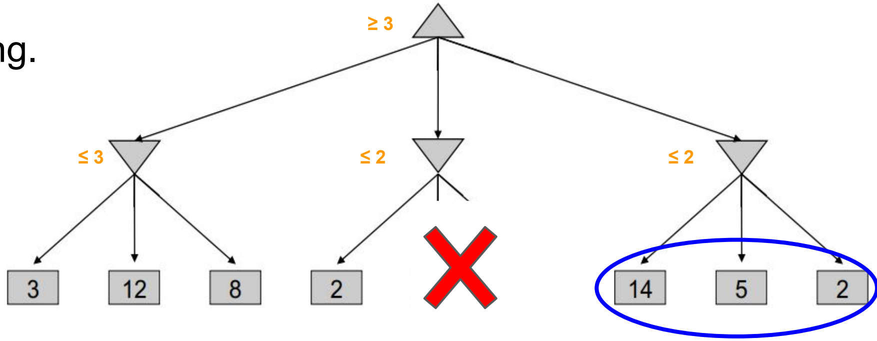
- This pruning has **no effect on the minimax value computed for the root**!

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

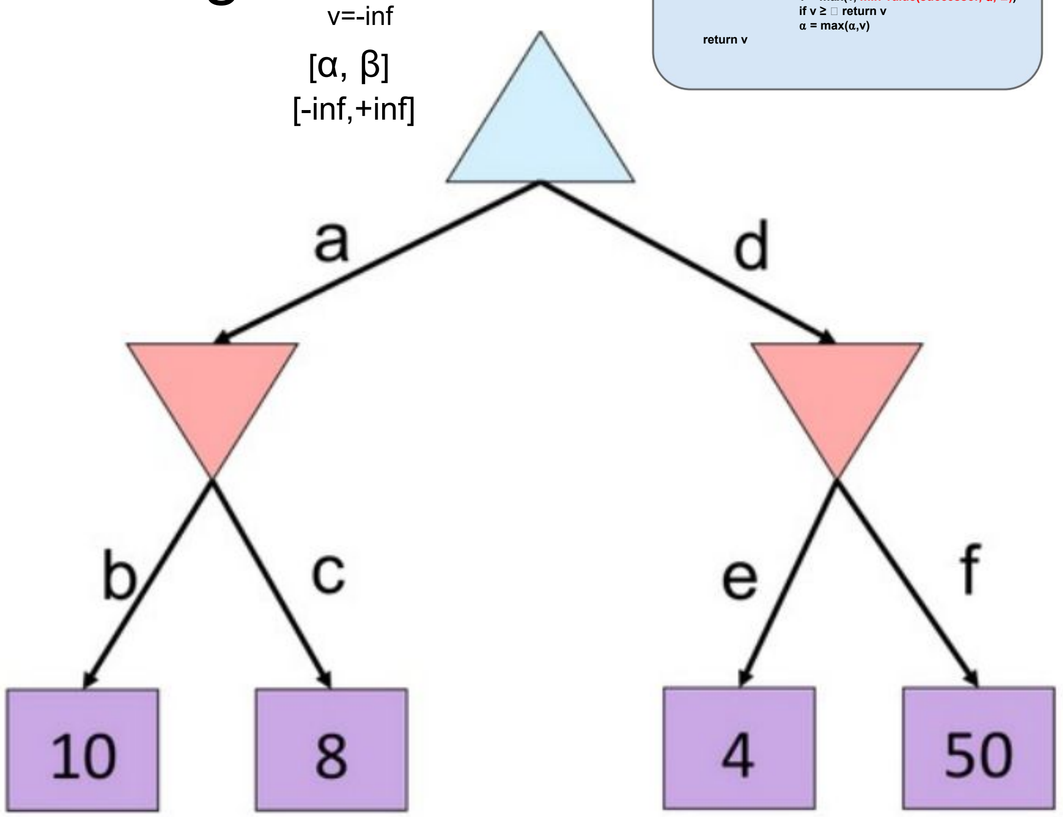
- Good child ordering improves the effectiveness of pruning.

- With "perfect ordering":
  - Time for search goes down.
  - Doubles solvable depth!
  - Full search of, e.g., chess, is still hopeless…





Alpha-Beta Pruning



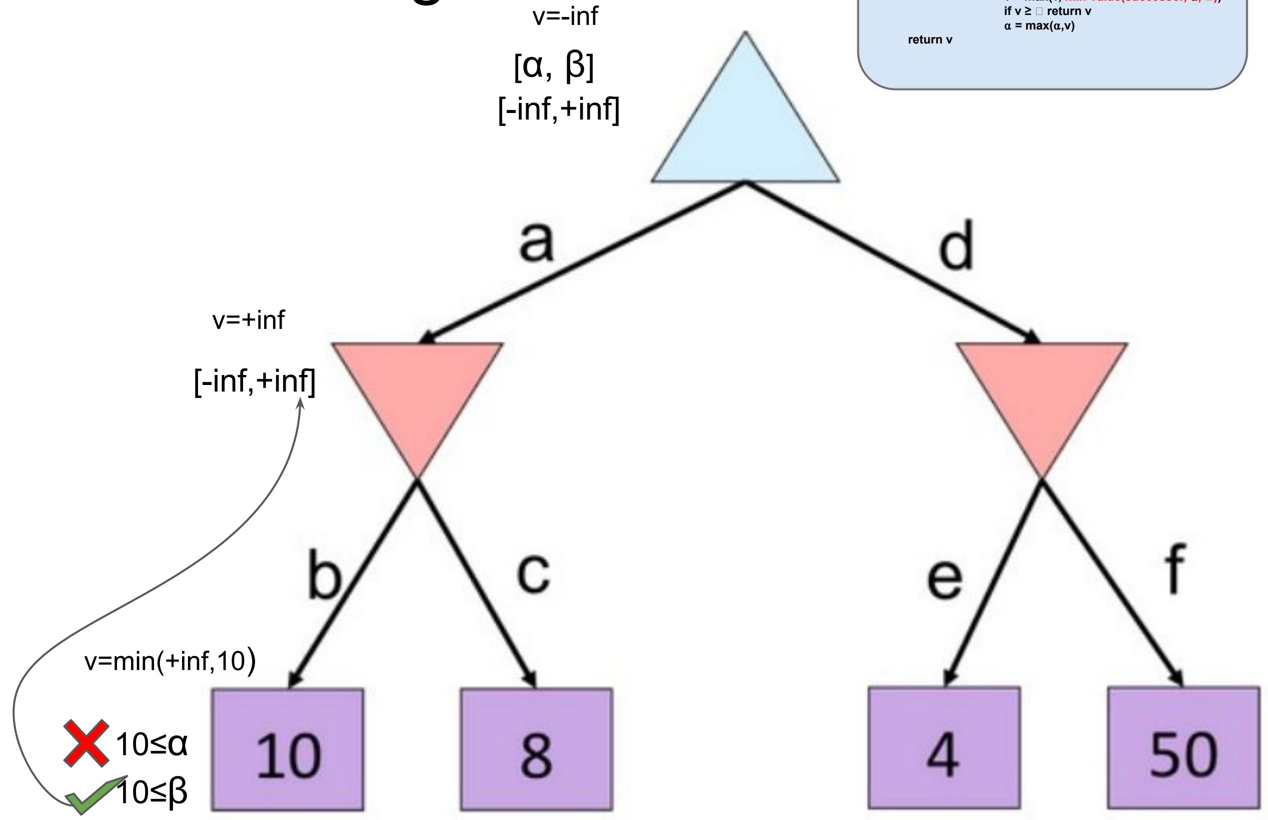
```
def max-value(state, α, β):

    if terminal-test(state):
                return utility(state)
    initialize v = —∞
    for each successor of state:
                v = max(v, min-value(successor, α, □))
                if v ≥ □ return v
                α = max(α,v)
    return v
```

```
def min-value(state , α, β):

    if terminal-test(state):
                return utility(state)
    initialize v = +∞
    for each successor of state:
                v = min(v, max-value(successor, α, □))
                if v ≤ α return v
                □ = min(□,v)
    return v
```

Alpha-Beta Pruning

```
def max-value(state, α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = —∞
    for each successor of state:
            v = max(v, min-value(successor, α, □))
            if v ≥ □ return v
            α = max(α,v)
    return v
```

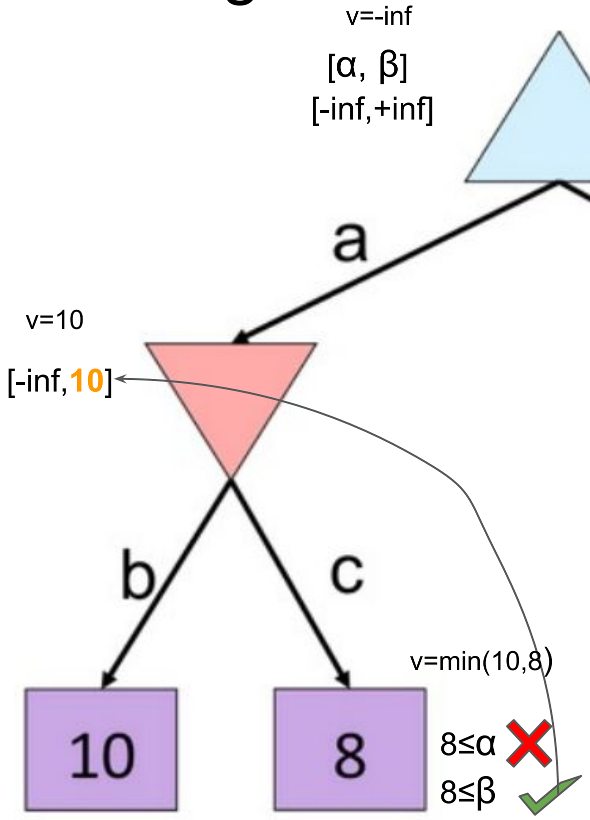
```
def min-value(state , α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = +∞
    for each successor of state:
            v = min(v, max-value(successor, α, □))
            if v ≤ α return v
            □ = min(□,v)
    return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

    if terminal-test(state):
                return utility(state)
    initialize v = —∞
    for each successor of state:
                v = max(v, min-value(successor, α, □))
                if v ≥ □ return v
                α = max(α,v)
    return v
```

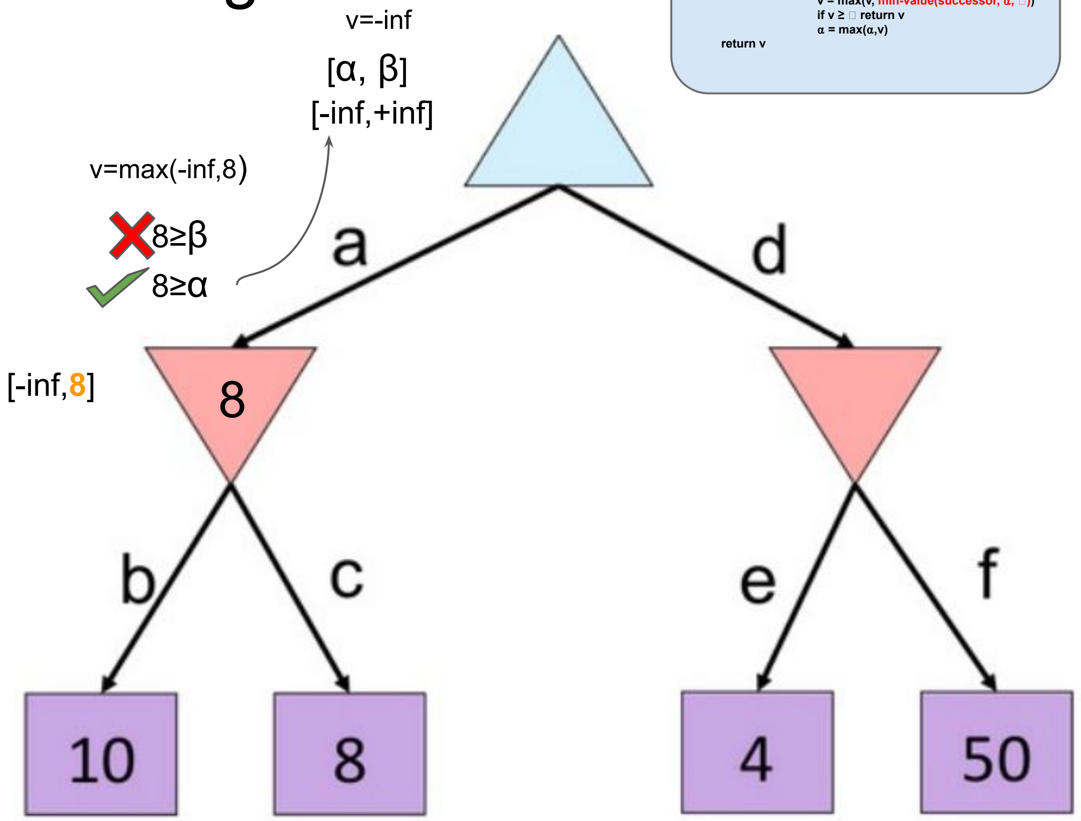
```
def min-value(state , α, β):

    if terminal-test(state):
                return utility(state)
    initialize v = +∞
    for each successor of state:
                v = min(v, max-value(successor, α, □))
                if v ≤ α return v
                □ = min(□,v)
    return v
```



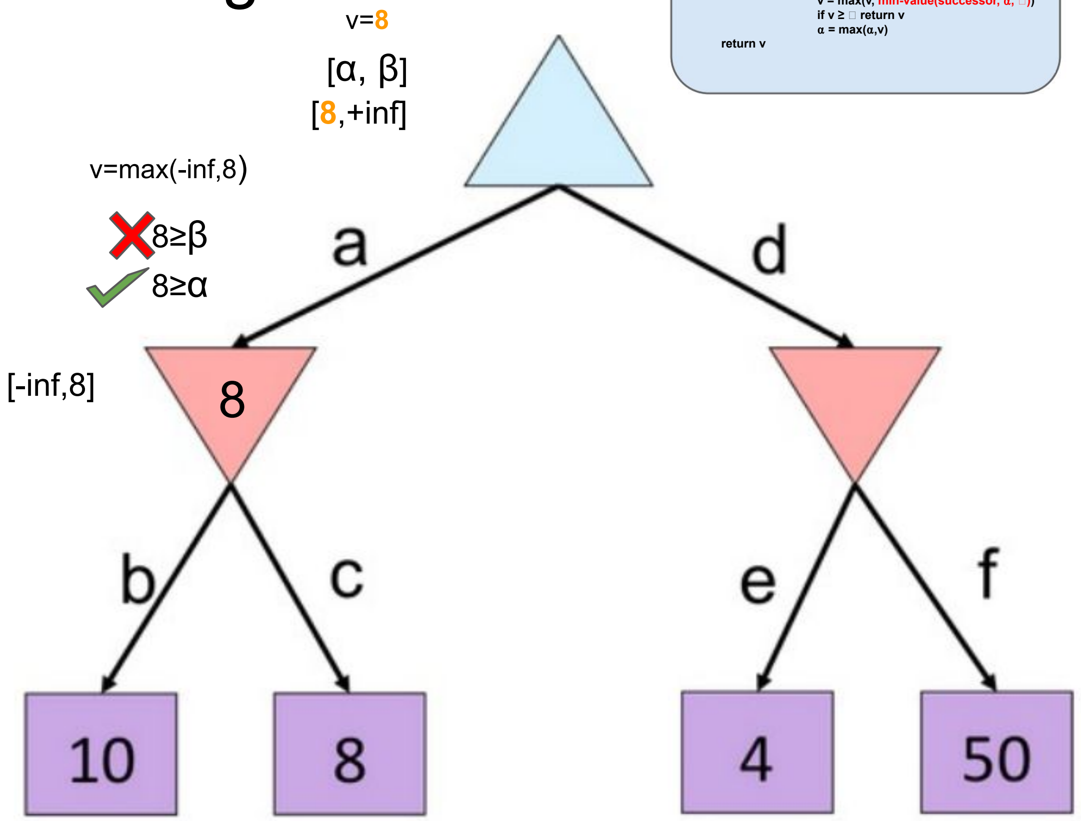
Alpha-Beta Pruning

```
def max-value(state, α, β):
    if terminal-test(state):
        return utility(state)
    initialize v = —∞
    for each successor of state:
        v = max(v, min-value(successor, α, □))
        if v ≥ □ return v
        α = max(α,v)
    return v
```

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



Alpha-Beta Pruning



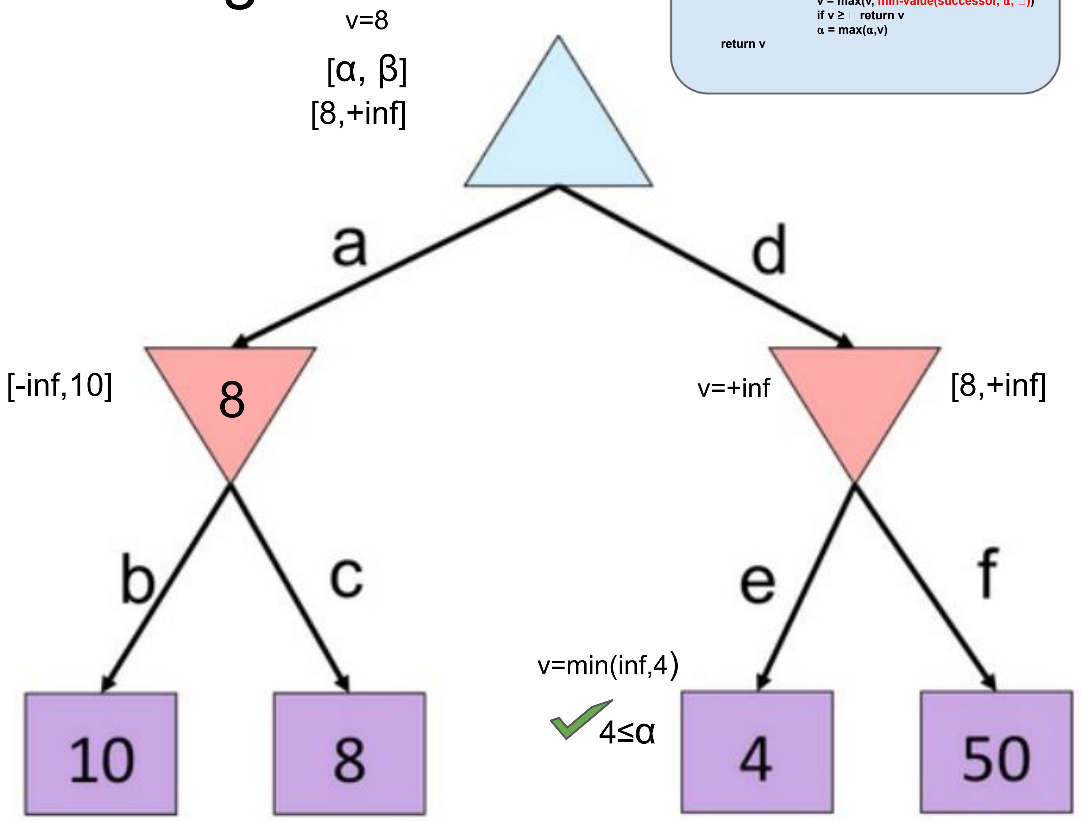
```
def max-value(state, α, β):

    if terminal-test(state):
                    return utility(state)
    initialize v = —∞
    for each successor of state:
                    v = max(v, min-value(successor, α, □))
                    if v ≥ □ return v
                    α = max(α,v)
    return v
```

```
def min-value(state , α, β):

    if terminal-test(state):
                    return utility(state)
    initialize v = +∞
    for each successor of state:
                    v = min(v, max-value(successor, α, □))
                    if v ≤ α return v
                    □ = min(□,v)
    return v
```

Alpha-Beta Pruning

```
def max-value(state, α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = —∞
    for each successor of state:
            v = max(v, min-value(successor, α, □))
            if v ≥ □ return v
            α = max(α,v)
    return v
```

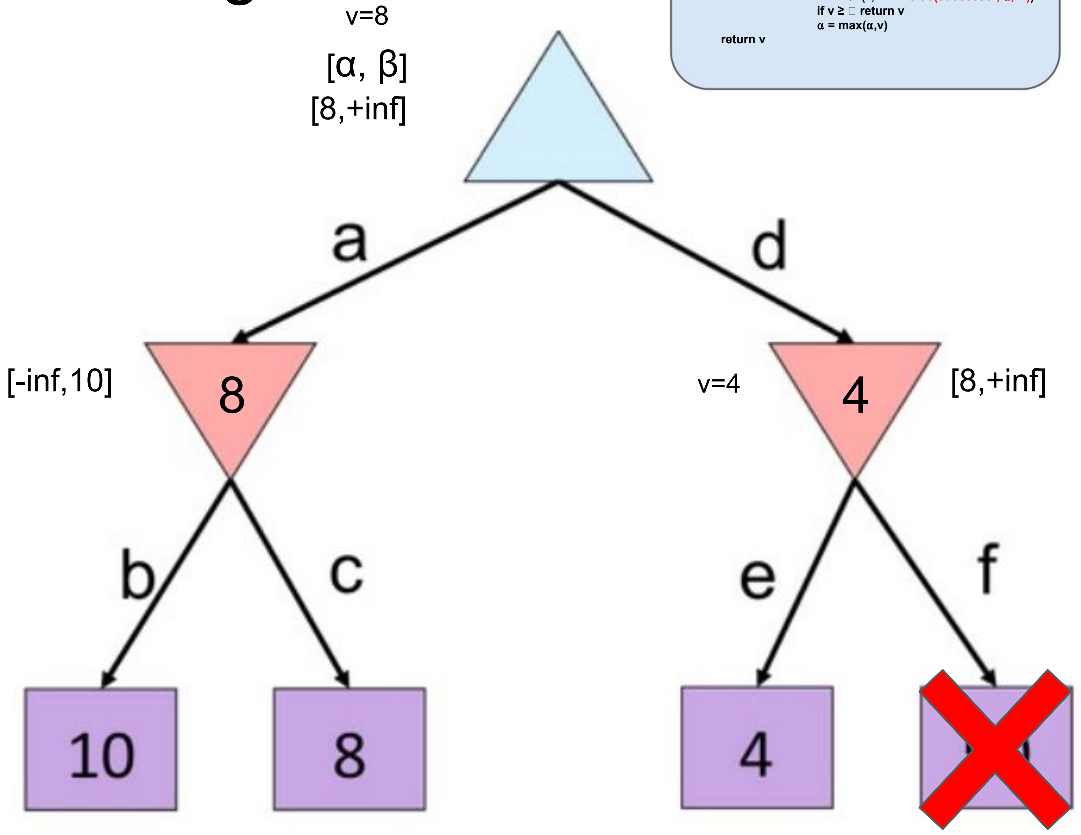
```
def min-value(state , α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = +∞
    for each successor of state:
            v = min(v, max-value(successor, α, □))
            if v ≤ α return v
            □ = min(□,v)
    return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

    if terminal-test(state):
                return utility(state)
    initialize v = —∞
    for each successor of state:
                v = max(v, min-value(successor, α, □))
                if v ≥ □ return v
                α = max(α,v)
    return v
```

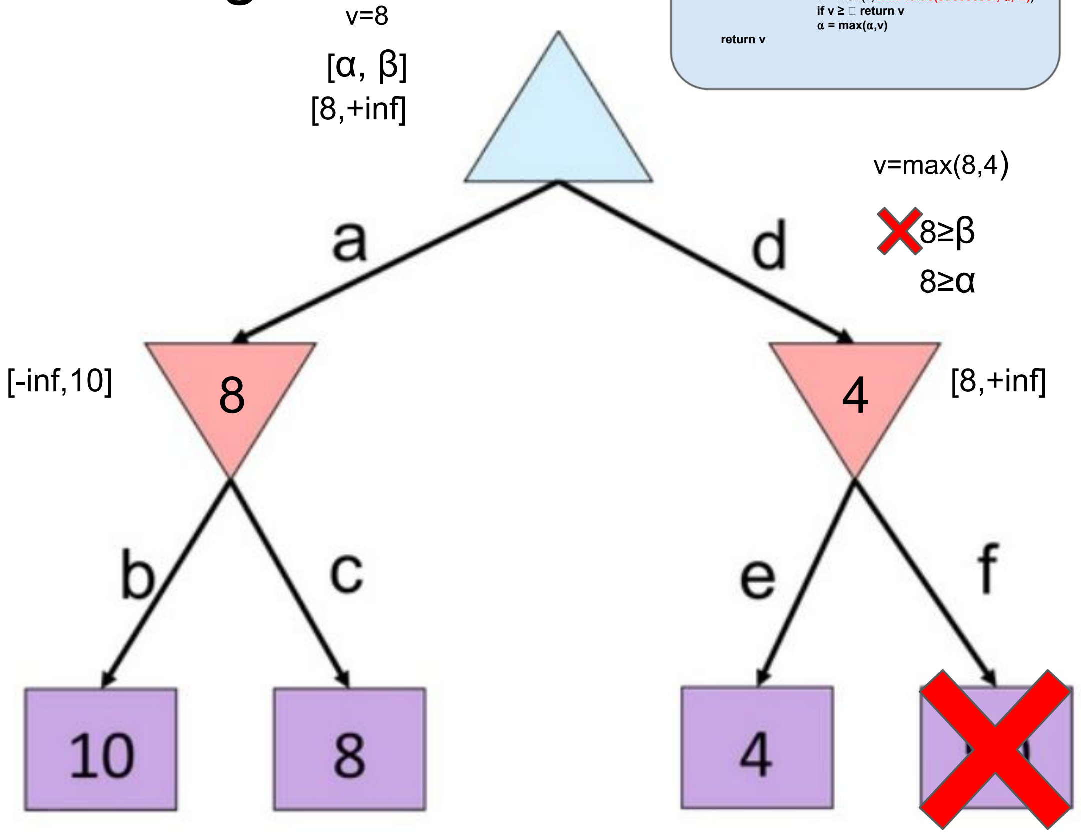
```
def min-value(state , α, β):

    if terminal-test(state):
                return utility(state)
    initialize v = +∞
    for each successor of state:
                v = min(v, max-value(successor, α, □))
                if v ≤ α return v
                □ = min(□,v)
    return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = —∞
    for each successor of state:
            v = max(v, min-value(successor, α, □))
            if v ≥ □ return v
            α = max(α,v)
    return v
```

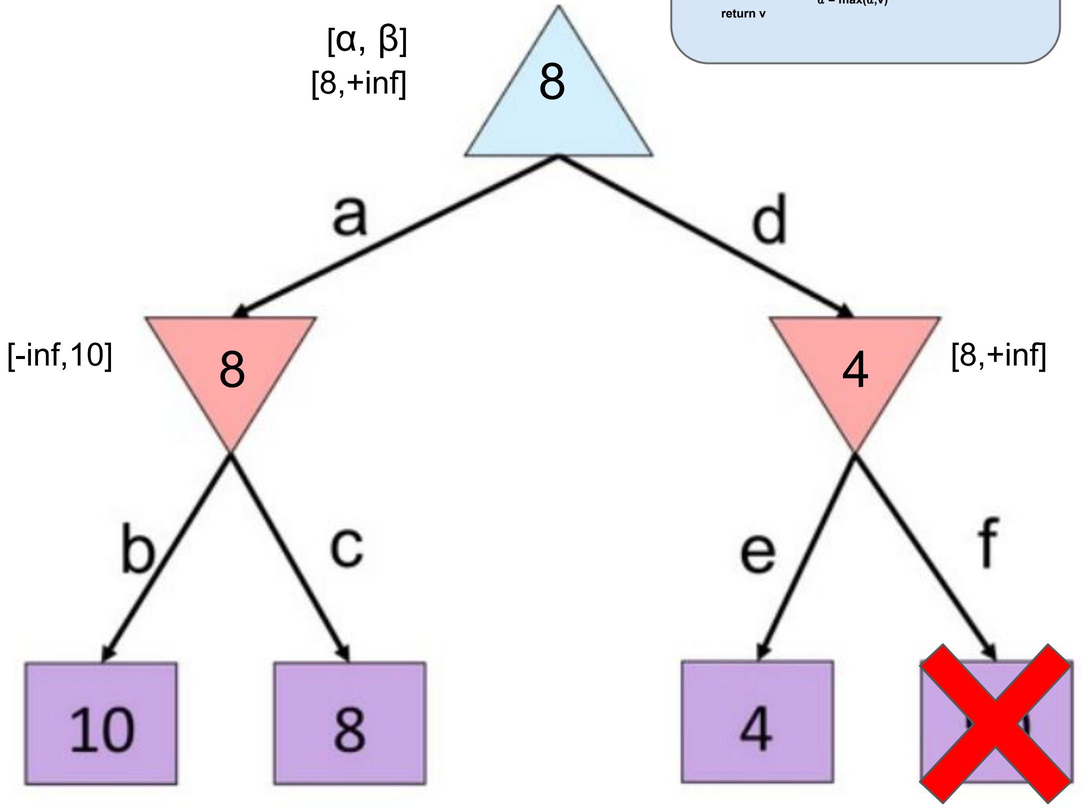
```
def min-value(state , α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = +∞
    for each successor of state:
            v = min(v, max-value(successor, α, □))
            if v ≤ α return v
            □ = min(□,v)
    return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

    if terminal-test(state):
                  return utility(state)
    initialize v = —∞
    for each successor of state:
                  v = max(v, min-value(successor, α, □))
                  if v ≥ □ return v
                  α = max(α,v)
    return v
```

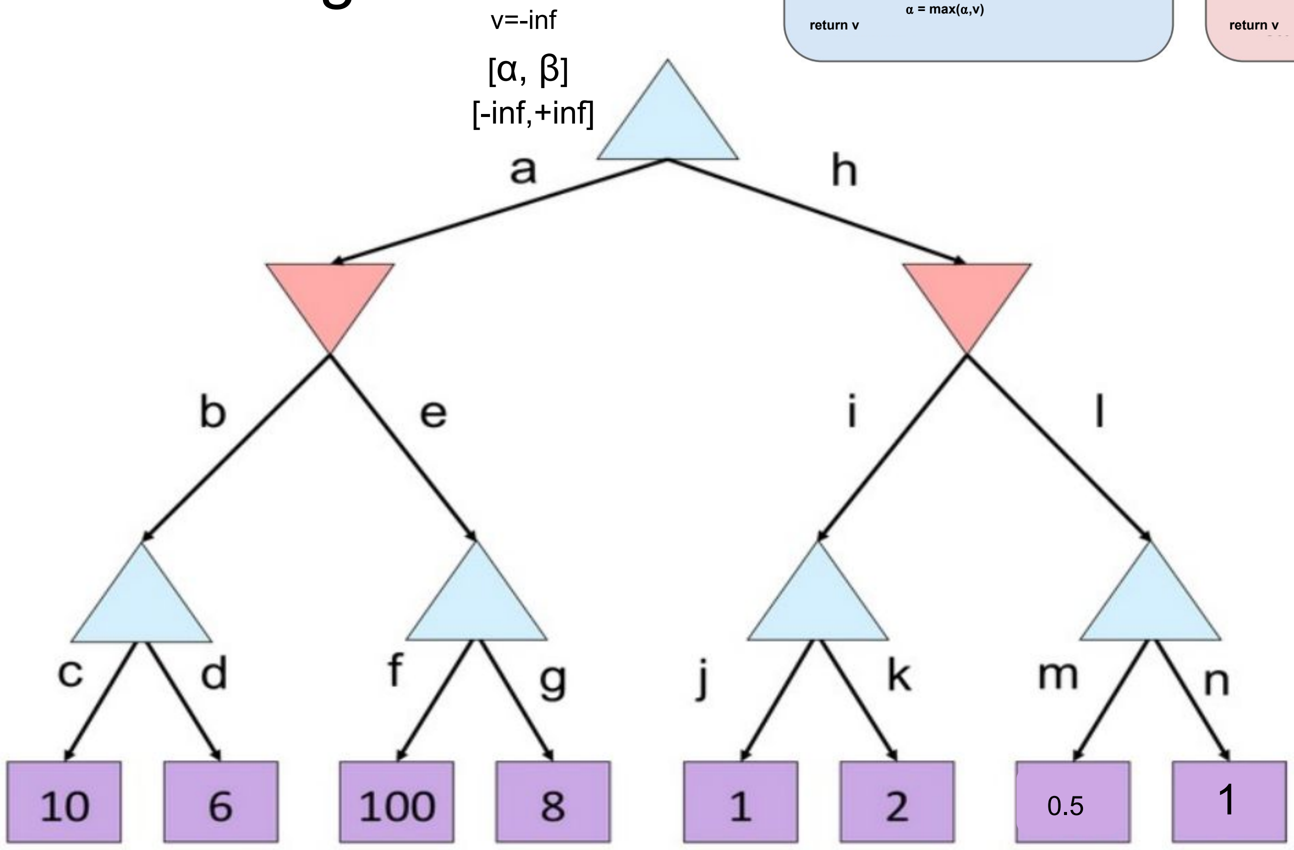
```
def min-value(state , α, β):

    if terminal-test(state):
                  return utility(state)
    initialize v = +∞
    for each successor of state:
                  v = min(v, max-value(successor, α, □))
                  if v ≤ α return v
                  □ = min(□,v)
    return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = —∞
    for each successor of state:
            v = max(v, min-value(successor, α, □))
            if v ≥ □ return v
            α = max(α,v)
    return v
```

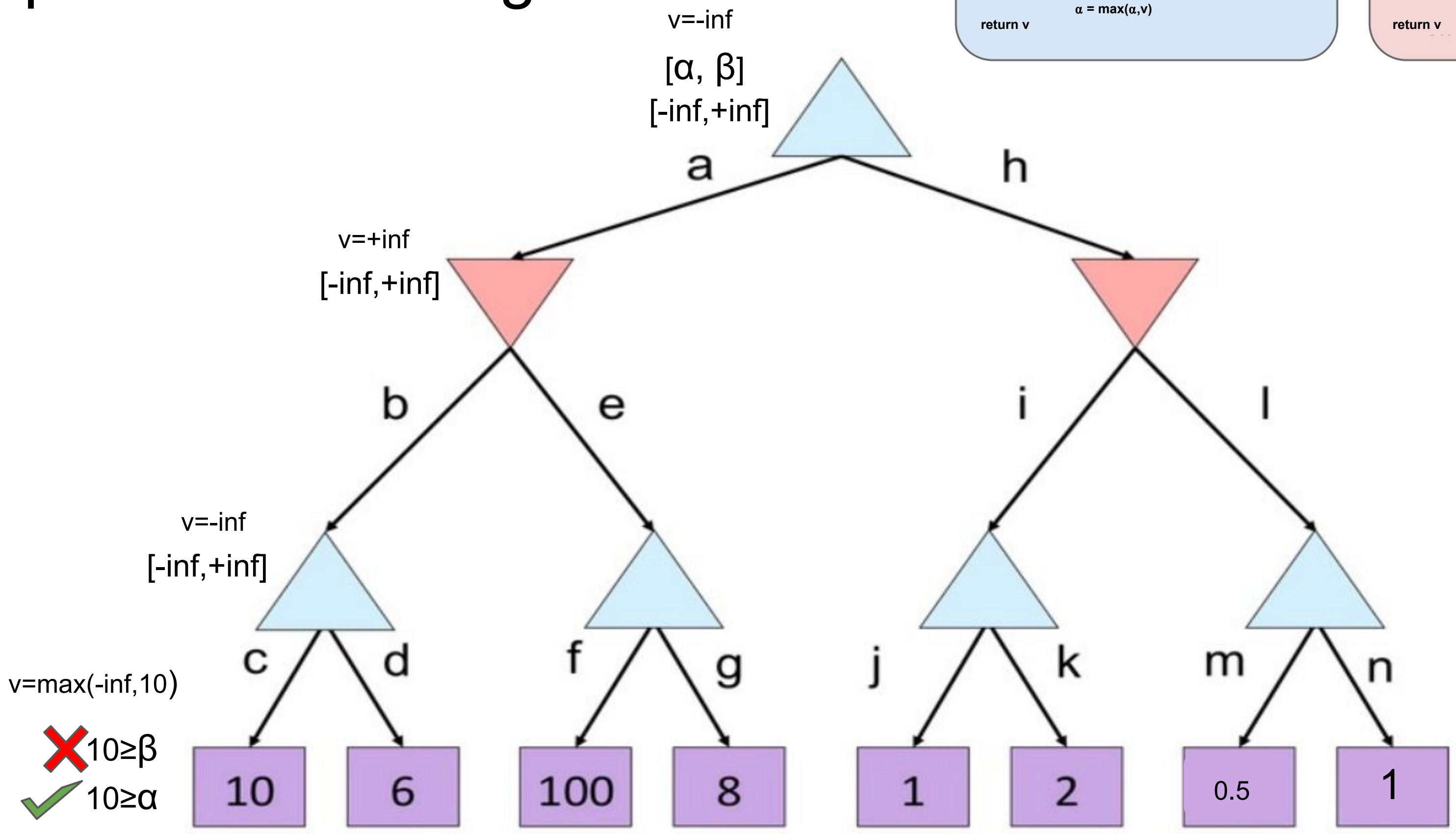
```
def min-value(state , α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = +∞
    for each successor of state:
            v = min(v, max-value(successor, α, □))
            if v ≤ α return v
            □ = min(□,v)
    return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = —∞
    for each successor of state:
            v = max(v, min-value(successor, α, □))
            if v ≥ □ return v
            α = max(α,v)
    return v
```

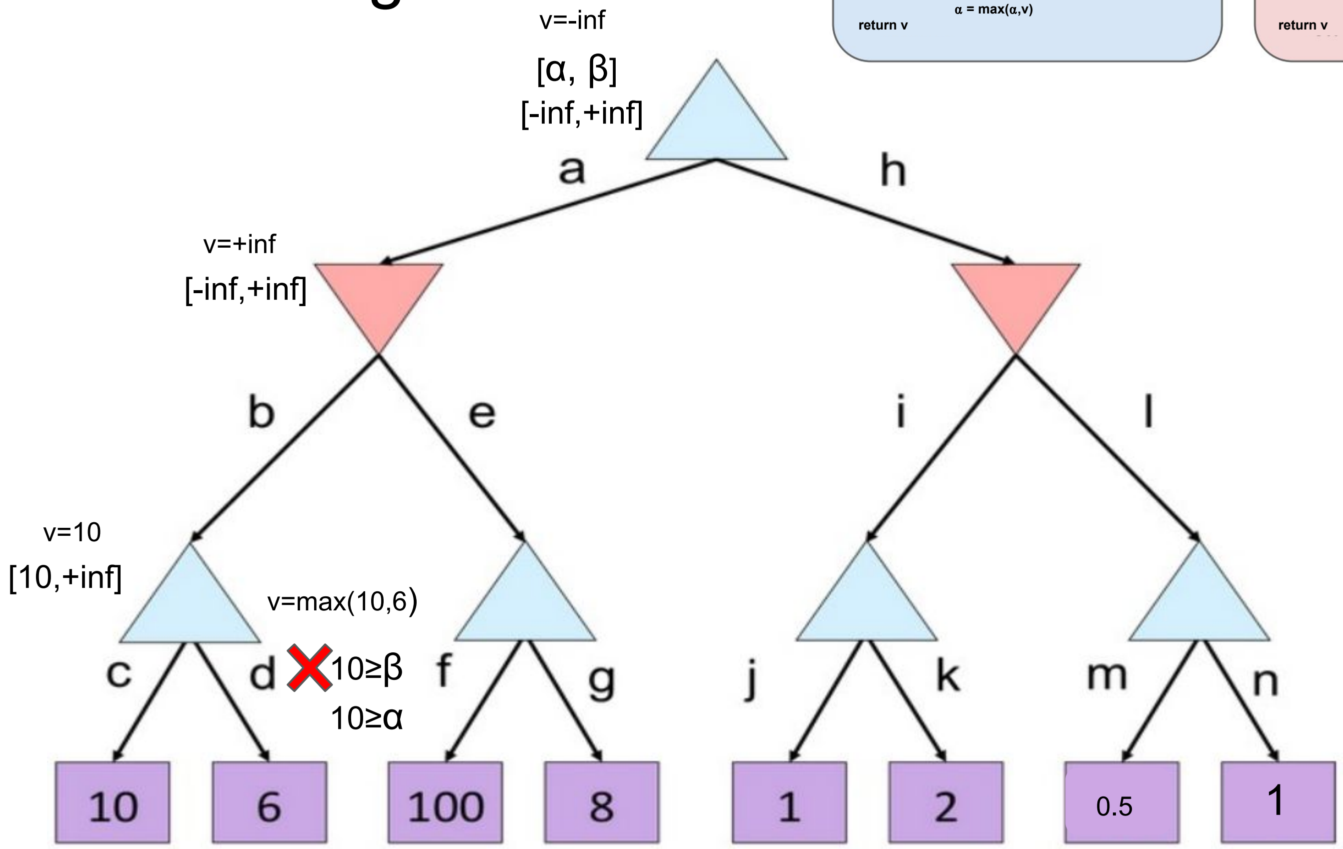
```
def min-value(state , α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = +∞
    for each successor of state:
            v = min(v, max-value(successor, α, □))
            if v ≤ α return v
            □ = min(□,v)
    return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

  if terminal-test(state):
            return utility(state)
  initialize v = —∞
  for each successor of state:
            v = max(v, min-value(successor, α, □))
            if v ≥ □ return v
            α = max(α,v)
  return v
```

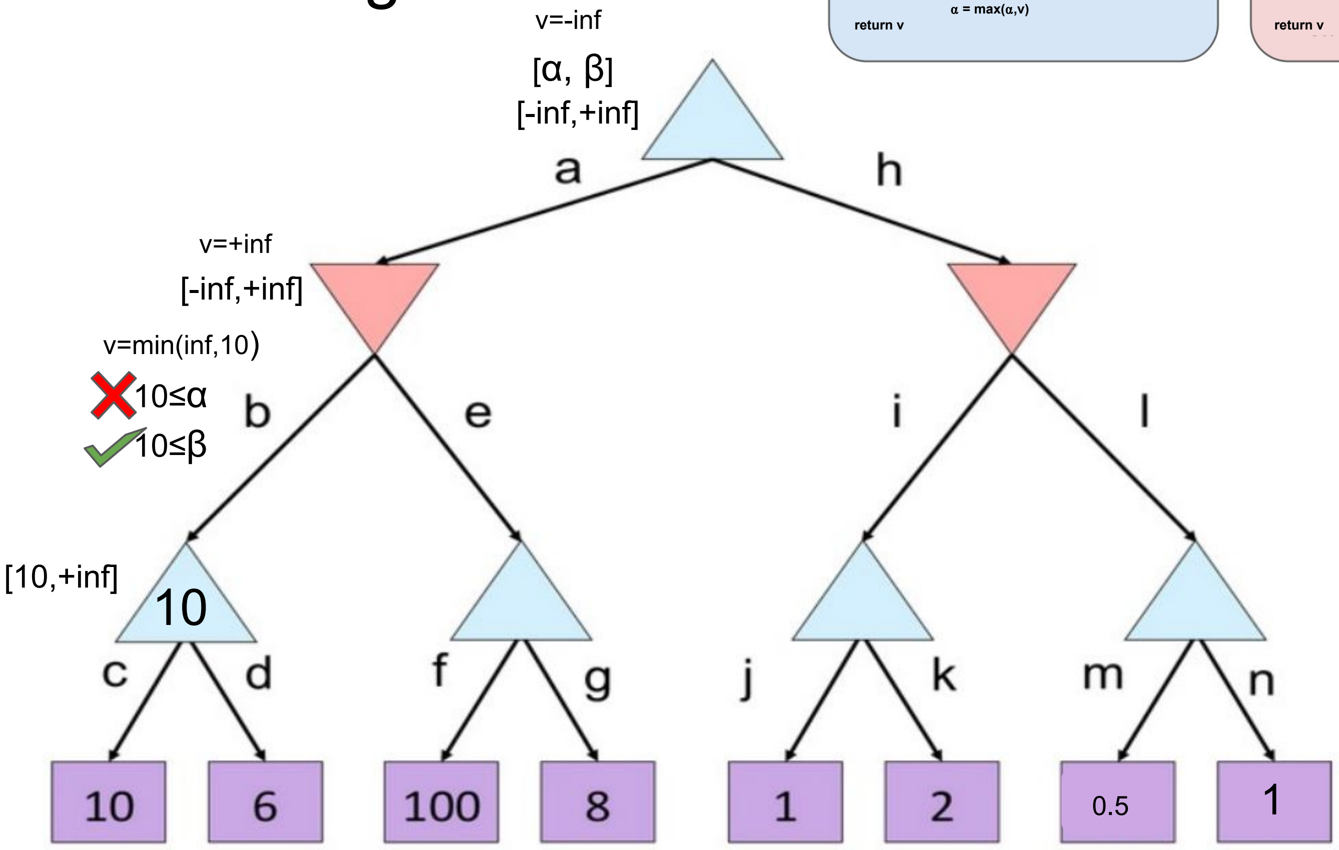
```
def min-value(state , α, β):

  if terminal-test(state):
            return utility(state)
  initialize v = +∞
  for each successor of state:
            v = min(v, max-value(successor, α, □))
            if v ≤ α return v
            □ = min(□,v)
  return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

  if terminal-test(state):
        return utility(state)
  initialize v = —∞
  for each successor of state:
        v = max(v, min-value(successor, α, □))
        if v ≥ □ return v
        α = max(α,v)
  return v
```

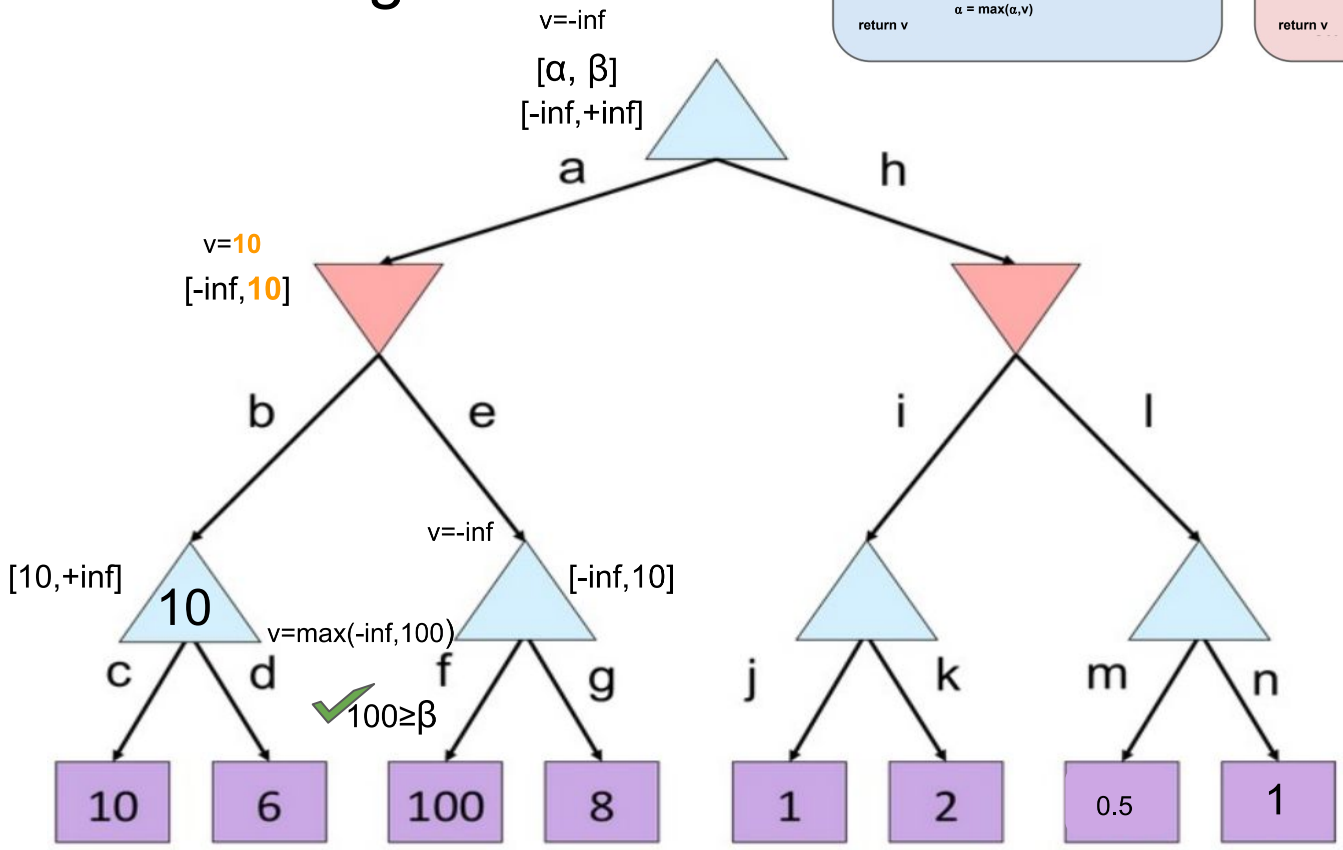
```
def min-value(state , α, β):

  if terminal-test(state):
        return utility(state)
  initialize v = +∞
  for each successor of state:
        v = min(v, max-value(successor, α, □))
        if v ≤ α return v
        □ = min(□,v)
  return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

  if terminal-test(state):
          return utility(state)
  initialize v = —∞
  for each successor of state:
          v = max(v, min-value(successor, α, □))
          if v ≥ □ return v
          α = max(α,v)
  return v
```

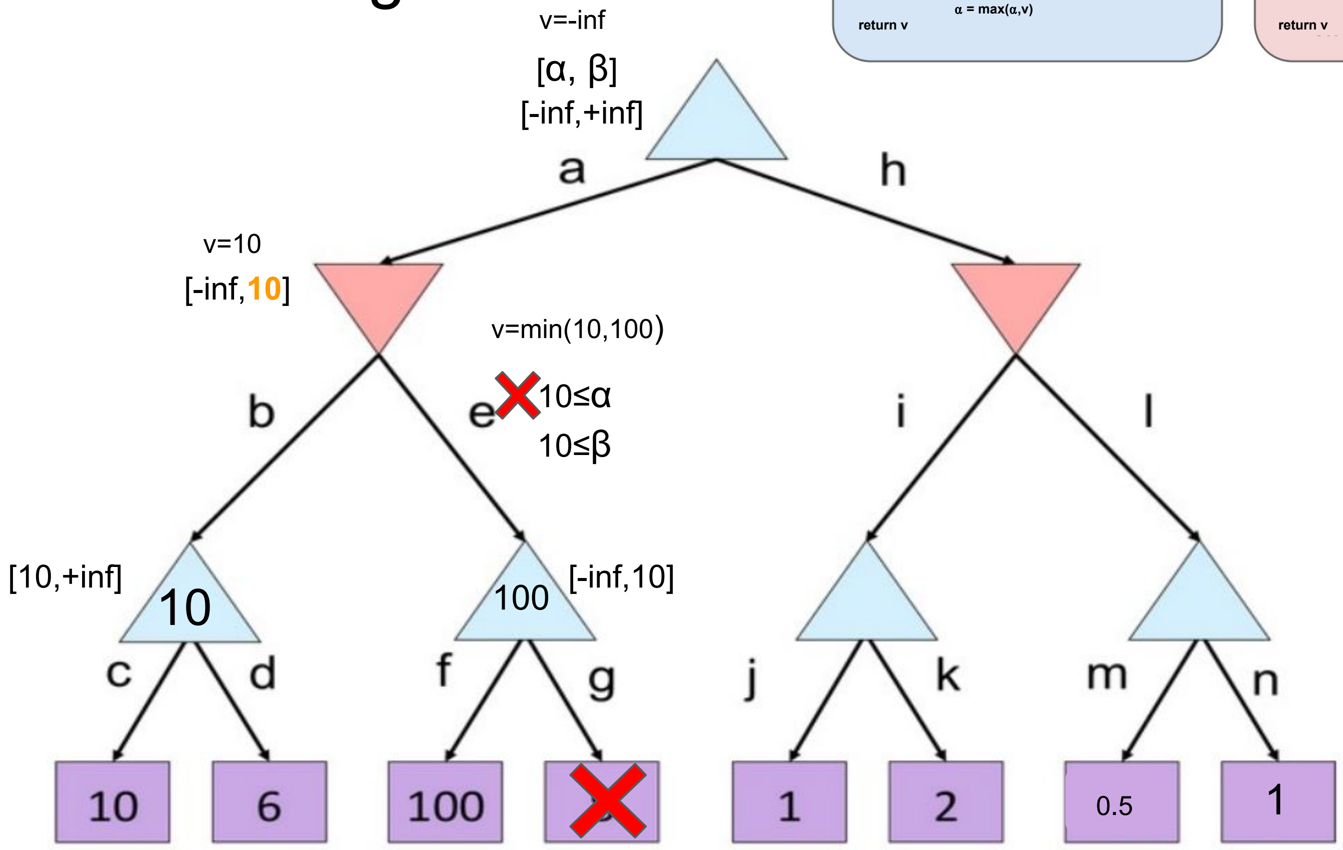
```
def min-value(state , α, β):

  if terminal-test(state):
          return utility(state)
  initialize v = +∞
  for each successor of state:
          v = min(v, max-value(successor, α, □))
          if v ≤ α return v
          □ = min(□,v)
  return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

if terminal-test(state):
                    return utility(state)
initialize v = —∞
for each successor of state:
                    v = max(v, min-value(successor, α, □))
                    if v ≥ □ return v
                    α = max(α,v)
return v
```

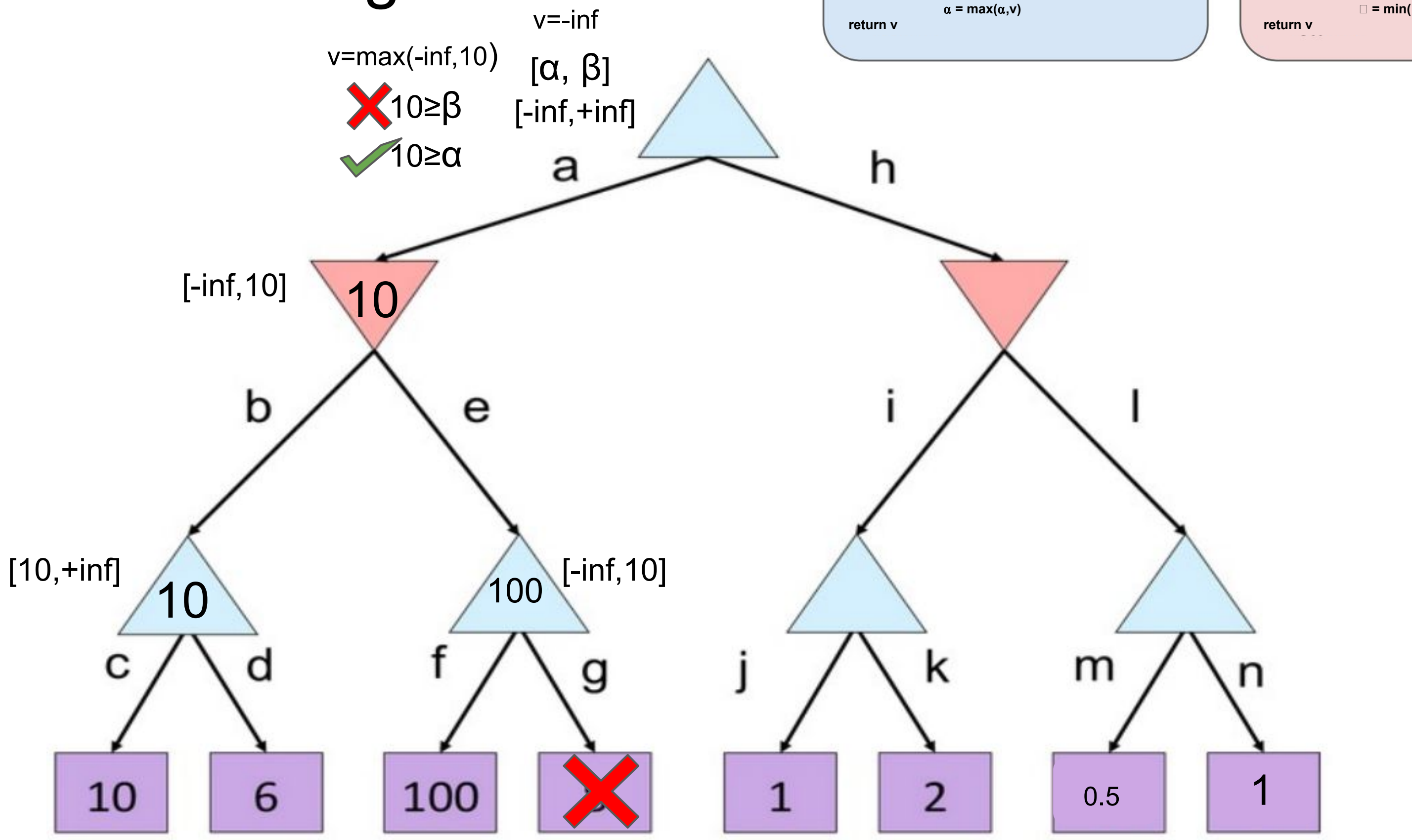
```
def min-value(state , α, β):

if terminal-test(state):
                    return utility(state)
initialize v = +∞
for each successor of state:
                    v = min(v, max-value(successor, α, □))
                    if v ≤ α return v
                    □ = min(□,v)
return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = —∞
    for each successor of state:
            v = max(v, min-value(successor, α, □))
            if v ≥ □ return v
            α = max(α,v)
    return v
```

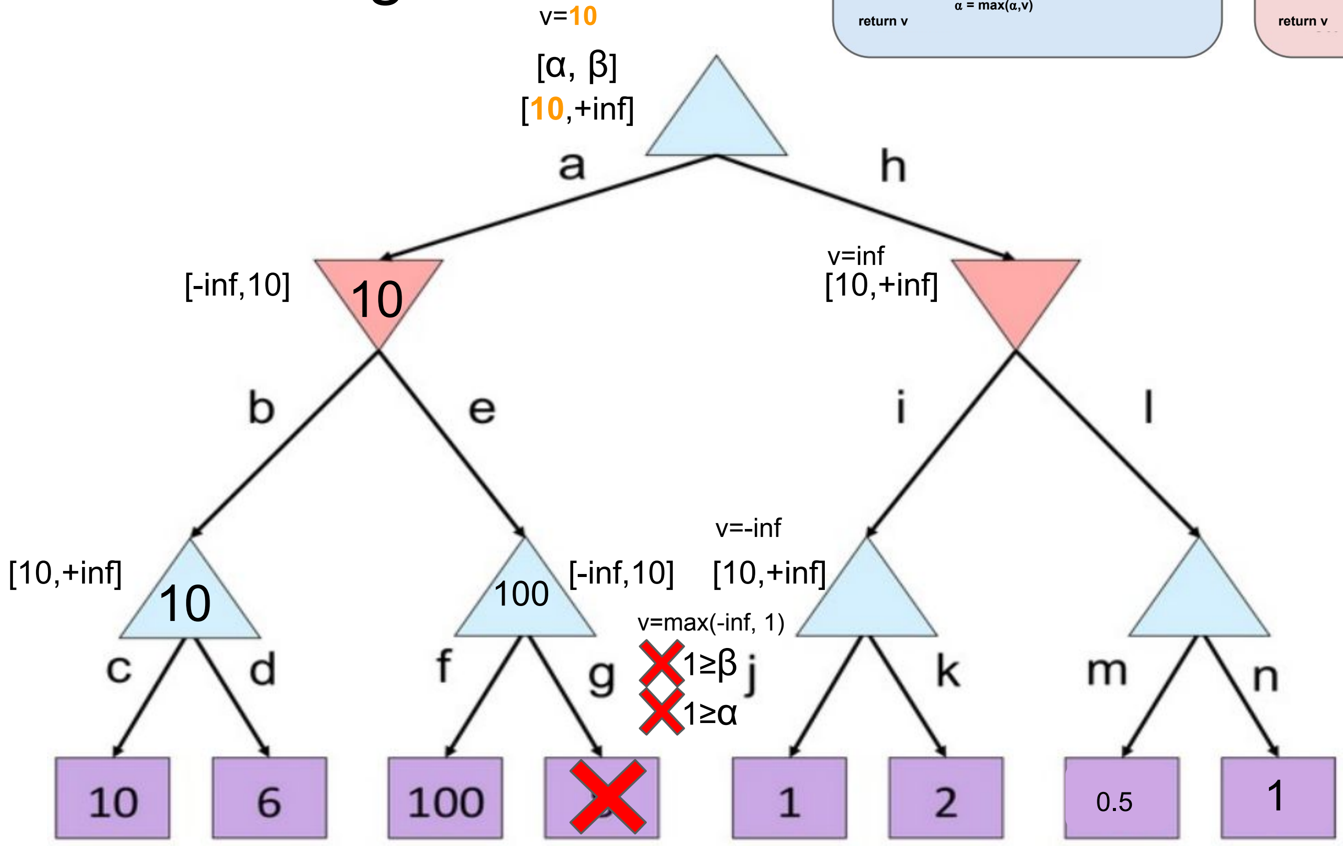
```
def min-value(state , α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = +∞
    for each successor of state:
            v = min(v, max-value(successor, α, □))
            if v ≤ α return v
            □ = min(□,v)
    return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = —∞
    for each successor of state:
            v = max(v, min-value(successor, α, □))
            if v ≥ □ return v
            α = max(α,v)
    return v
```

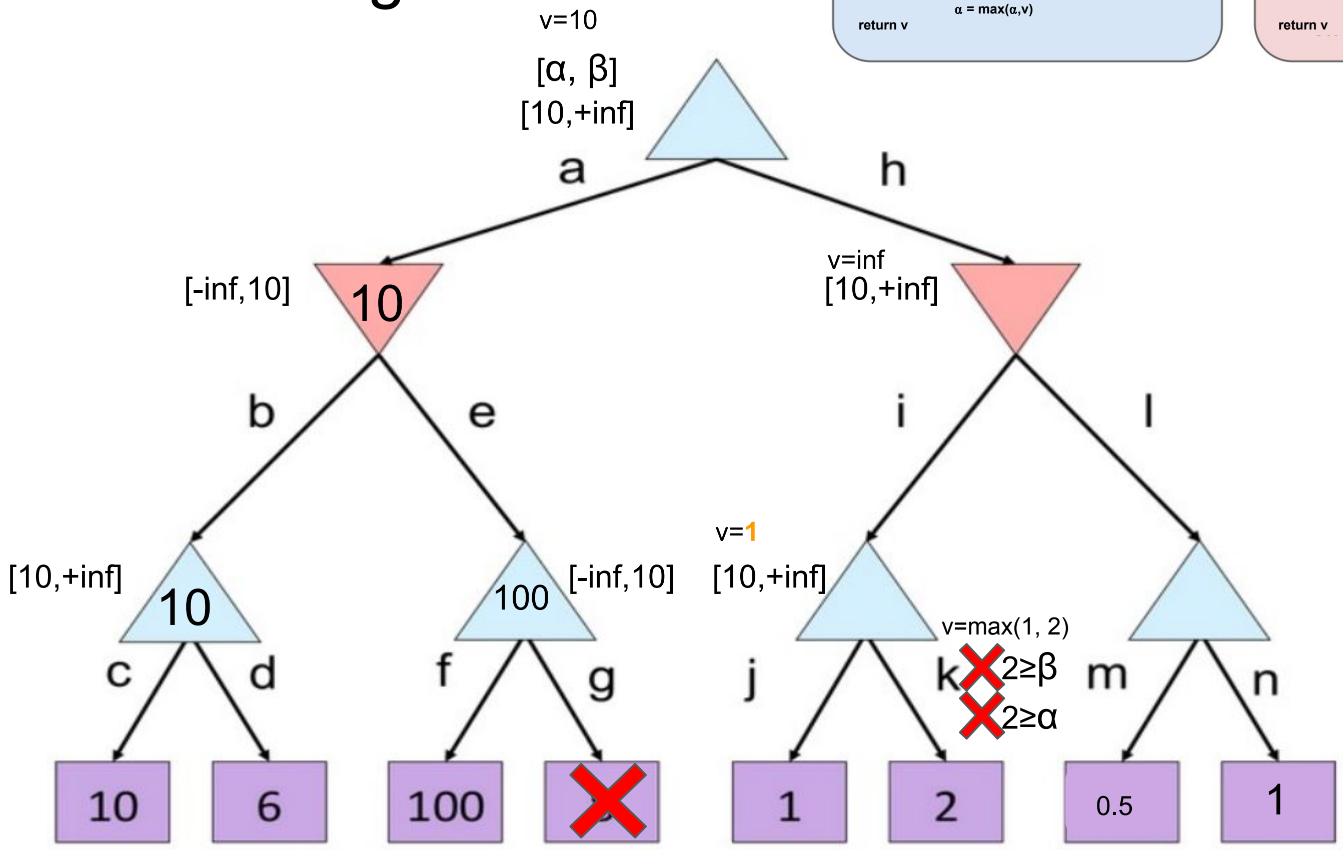
```
def min-value(state , α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = +∞
    for each successor of state:
            v = min(v, max-value(successor, α, □))
            if v ≤ α return v
            □ = min(□,v)
    return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = —∞
    for each successor of state:
            v = max(v, min-value(successor, α, □))
            if v ≥ □ return v
            α = max(α,v)
    return v
```

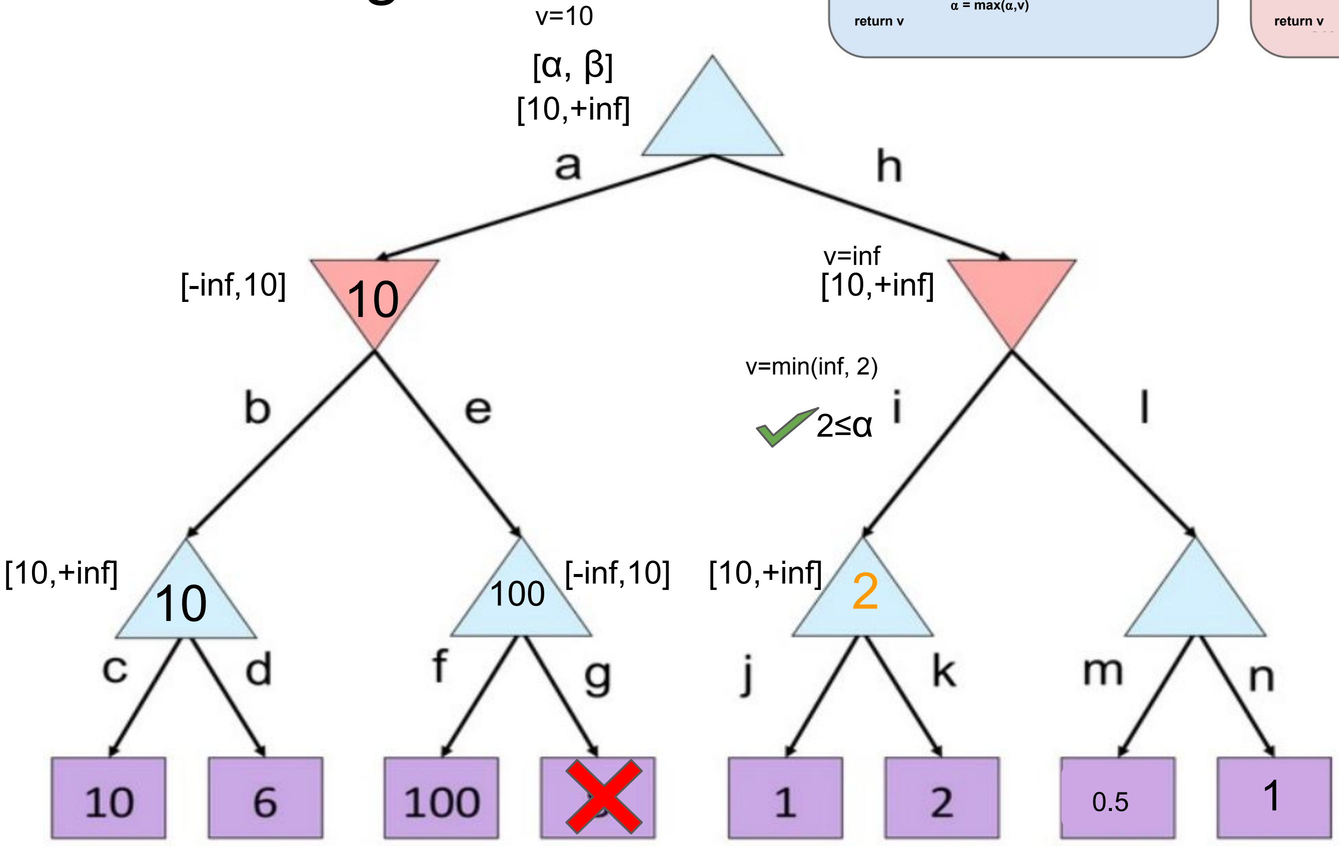
```
def min-value(state , α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = +∞
    for each successor of state:
            v = min(v, max-value(successor, α, □))
            if v ≤ α return v
            □ = min(□,v)
    return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

  if terminal-test(state):
          return utility(state)
  initialize v = —∞
  for each successor of state:
          v = max(v, min-value(successor, α, □))
          if v ≥ □ return v
          α = max(α,v)
  return v
```

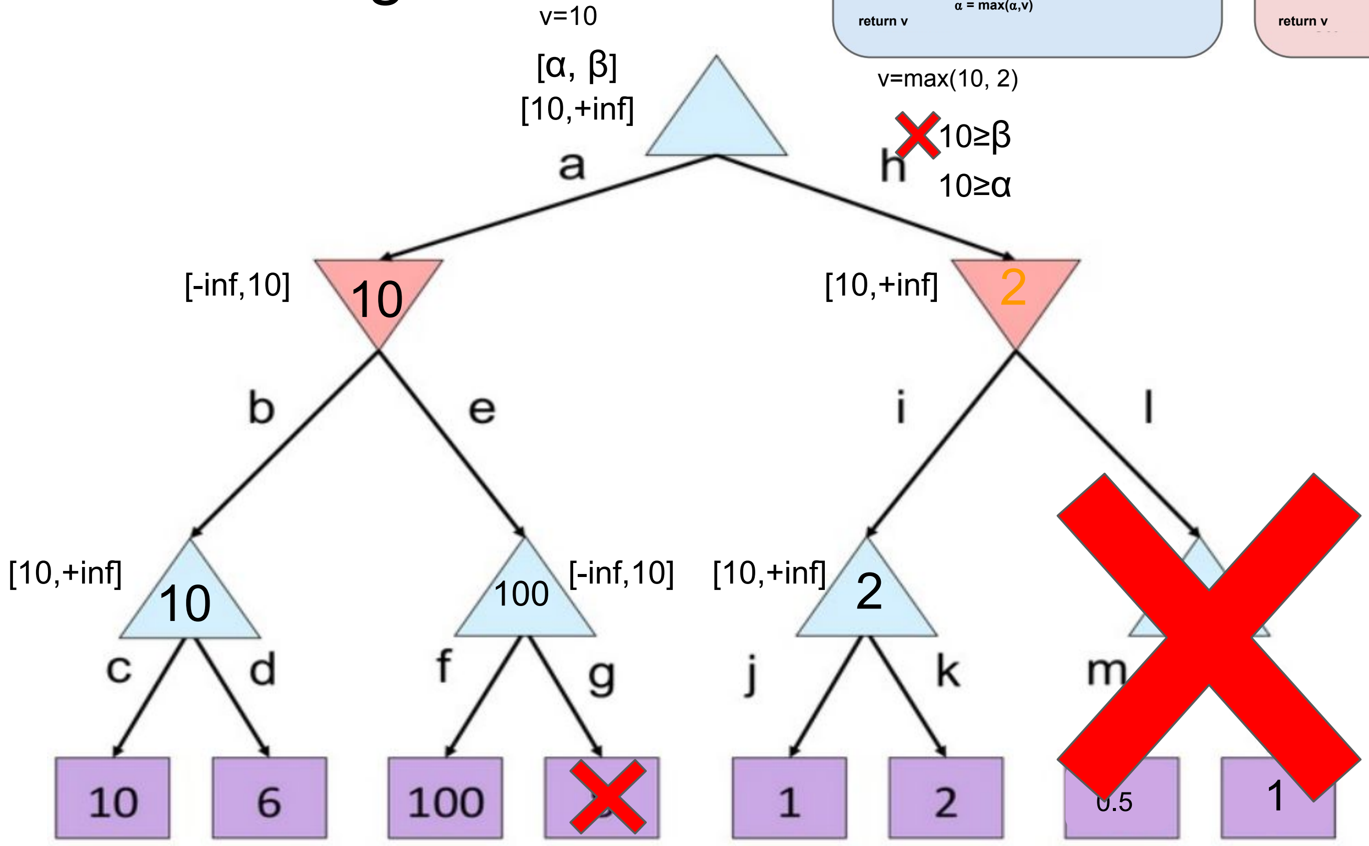
```
def min-value(state , α, β):

  if terminal-test(state):
          return utility(state)
  initialize v = +∞
  for each successor of state:
          v = min(v, max-value(successor, α, □))
          if v ≤ α return v
          □ = min(□,v)
  return v
```



Alpha-Beta Pruning

```
def max-value(state, α, β):

    if terminal-test(state):
                return utility(state)
    initialize v = —∞
    for each successor of state:
                v = max(v, min-value(successor, α, □))
                if v ≥ □ return v
                α = max(α,v)
    return v
```

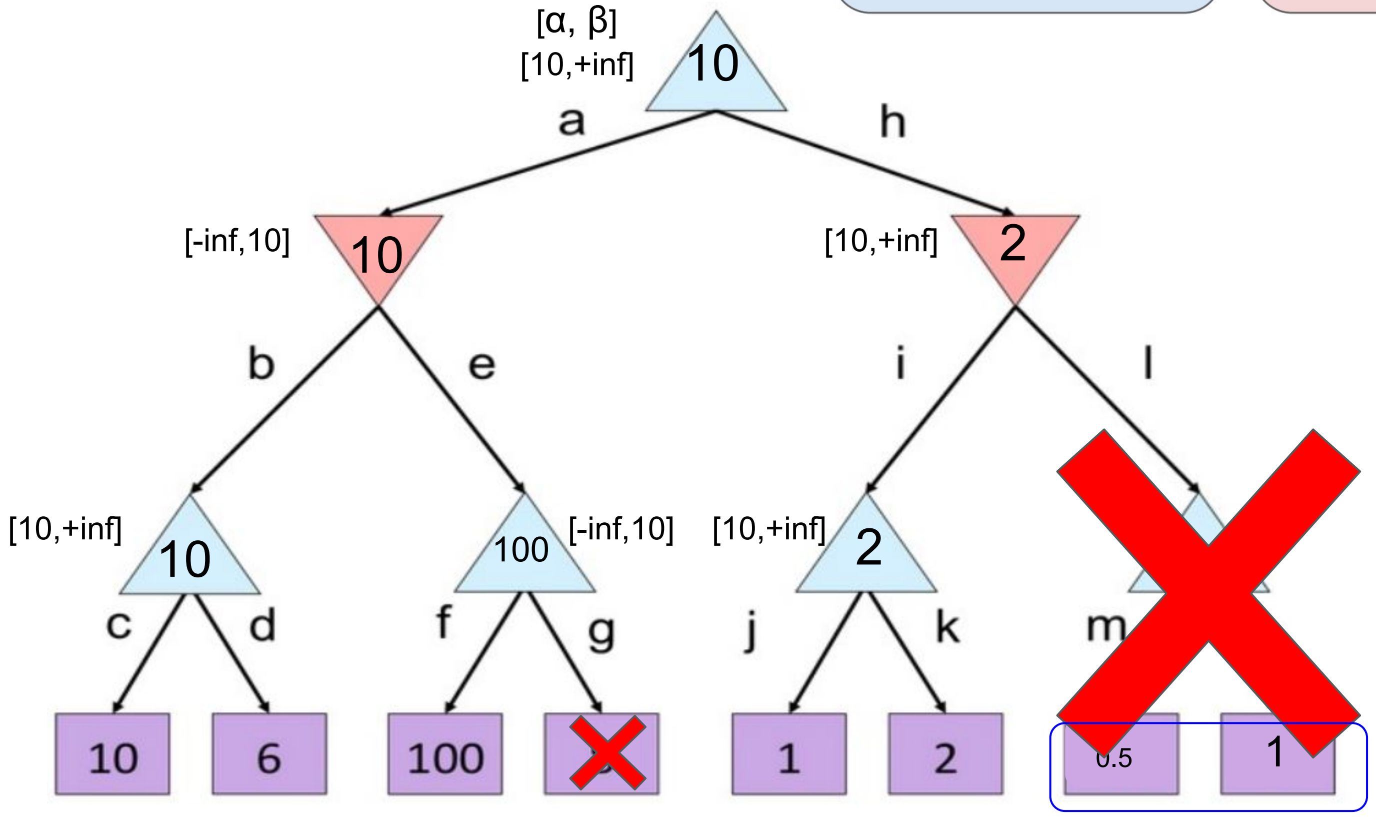
```
def min-value(state , α, β):

    if terminal-test(state):
                return utility(state)
    initialize v = +∞
    for each successor of state:
                v = min(v, max-value(successor, α, □))
                if v ≤ α return v
                □ = min(□,v)
    return v
```



Alpha-Beta Pruning

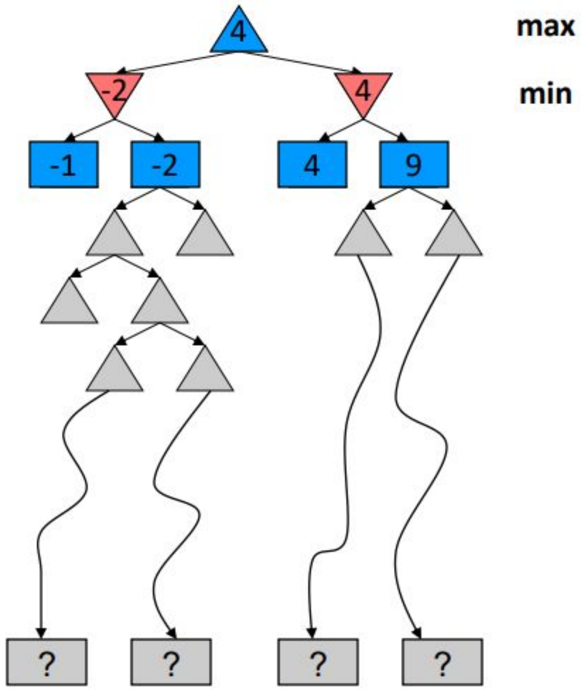
```
def max-value(state, α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = —∞
    for each successor of state:
            v = max(v, min-value(successor, α, □))
            if v ≥ □ return v
            α = max(α,v)
    return v
```

```
def min-value(state , α, β):

    if terminal-test(state):
            return utility(state)
    initialize v = +∞
    for each successor of state:
            v = min(v, max-value(successor, α, □))
            if v ≤ α return v
            □ = min(□,v)
    return v
```

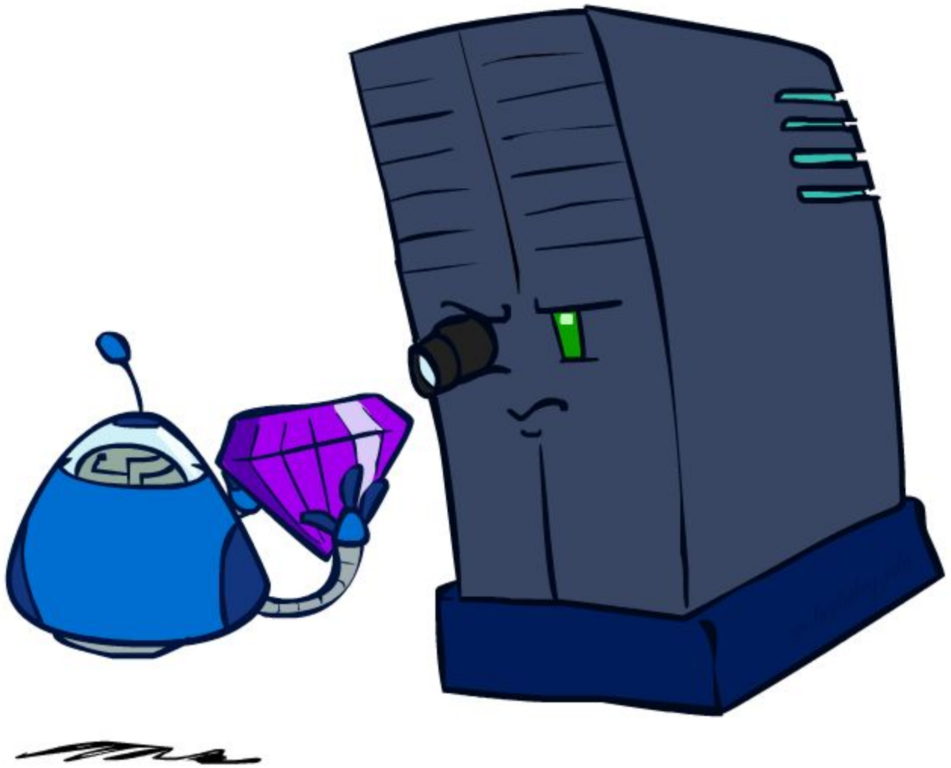


Overcoming Resource Limits: Limiting Depth

- Problem: In realistic games, you cannot search upto leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Use an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds for a move, and can explore 10K nodes per second.
  - So can check 1M nodes per move
- Guarantee of optimal play is gone.
- More plies/moves makes a BIG difference.
- Use iterative deepening.

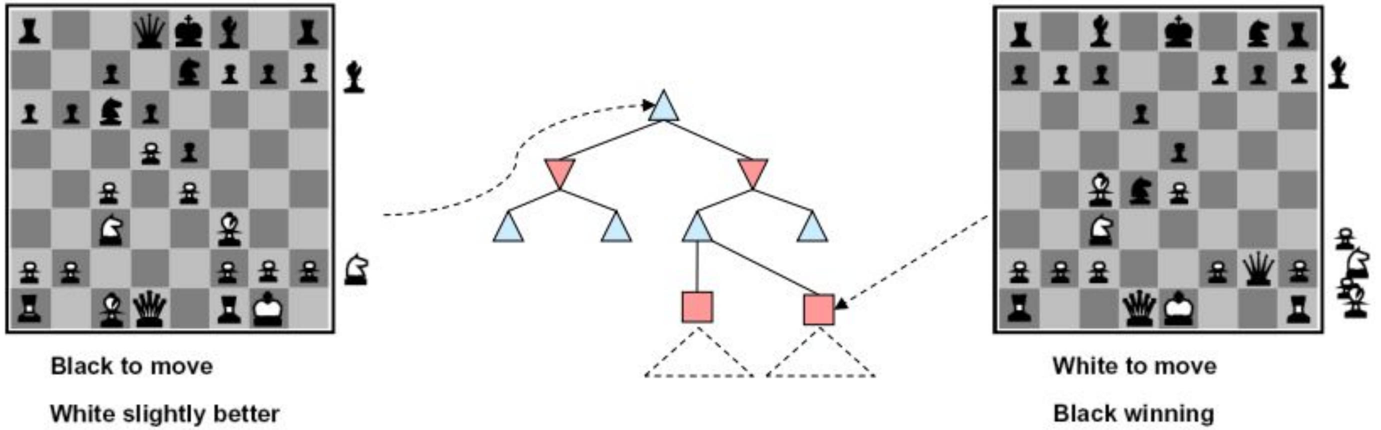


Evaluation Function



Evaluation Function

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



- 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. f1(s) = (num white queens – num black queens), etc.

Evaluation Function

- Evaluation functions are always imperfect.

- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters.

- Tradeoff between complexity of features and complexity of computation