CS 4700: Foundations of Artificial Intelligence

Spring 2020
Prof. Haym Hirsh

Lecture 7
February 5, 2020

Reminder: Technology Policy

No technology except for first four rows of left and right sides

Jupyter Notebooks

"Primer": Friday 5pm in Gates G01

Covers:
Jupyter Notebooks and .ipynb files
Google Colab (online Jupyter Notebook environment)
Useful Python features

Bring your laptop

Recap

- Uninformed search
  - Depth-first search
  - Breadth-first search
  - Iterative deepening search
- Informed search
  - Best-first search
  - A*
  - Hill climbing
  - Simulated annealing
- Genetic algorithms/evoluationary computation

Other Search Topics (not covered - yet)

- Continuous spaces / gradient-based methods
- Non-deterministic actions
  - Oops, I dropped the cup
- Partially observable states
  - What’s behind me?

Other Search Topics (not covered - yet)

- Continuous spaces / gradient-based methods
- Non-deterministic actions
  - Oops, I dropped the cup
- Partially observable states
  - What's behind me?

Will come up in machine learning

Other Search Topics (not covered - yet)

- Continuous spaces / gradient-based methods
- Non-deterministic actions
  - Oops, I dropped the cup
- Partially observable states
  - What's behind me?

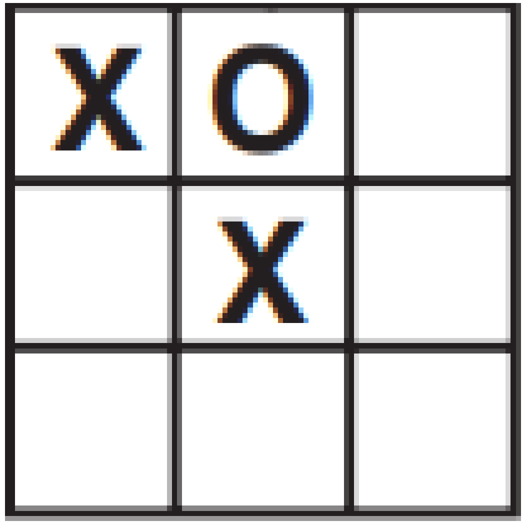
Will come up in machine learning

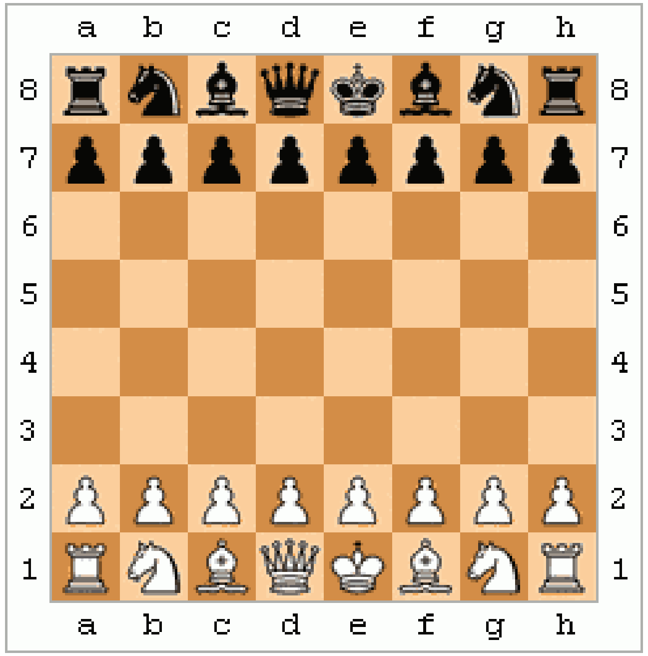
Will come up in reinforcement learning

Adversarial Search
(Playing Games)

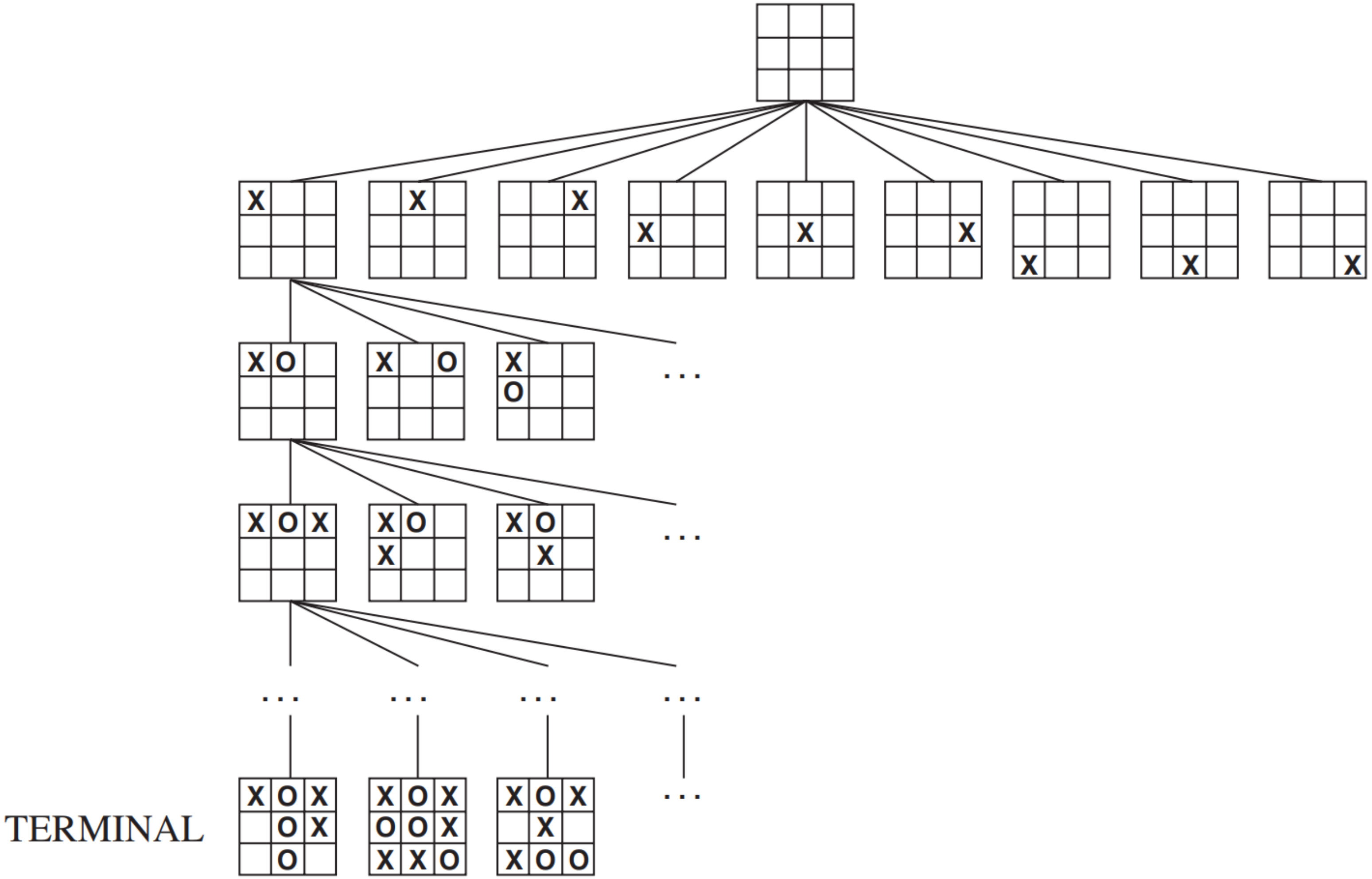
Chapter 5

Two Player Zero-Sum Games



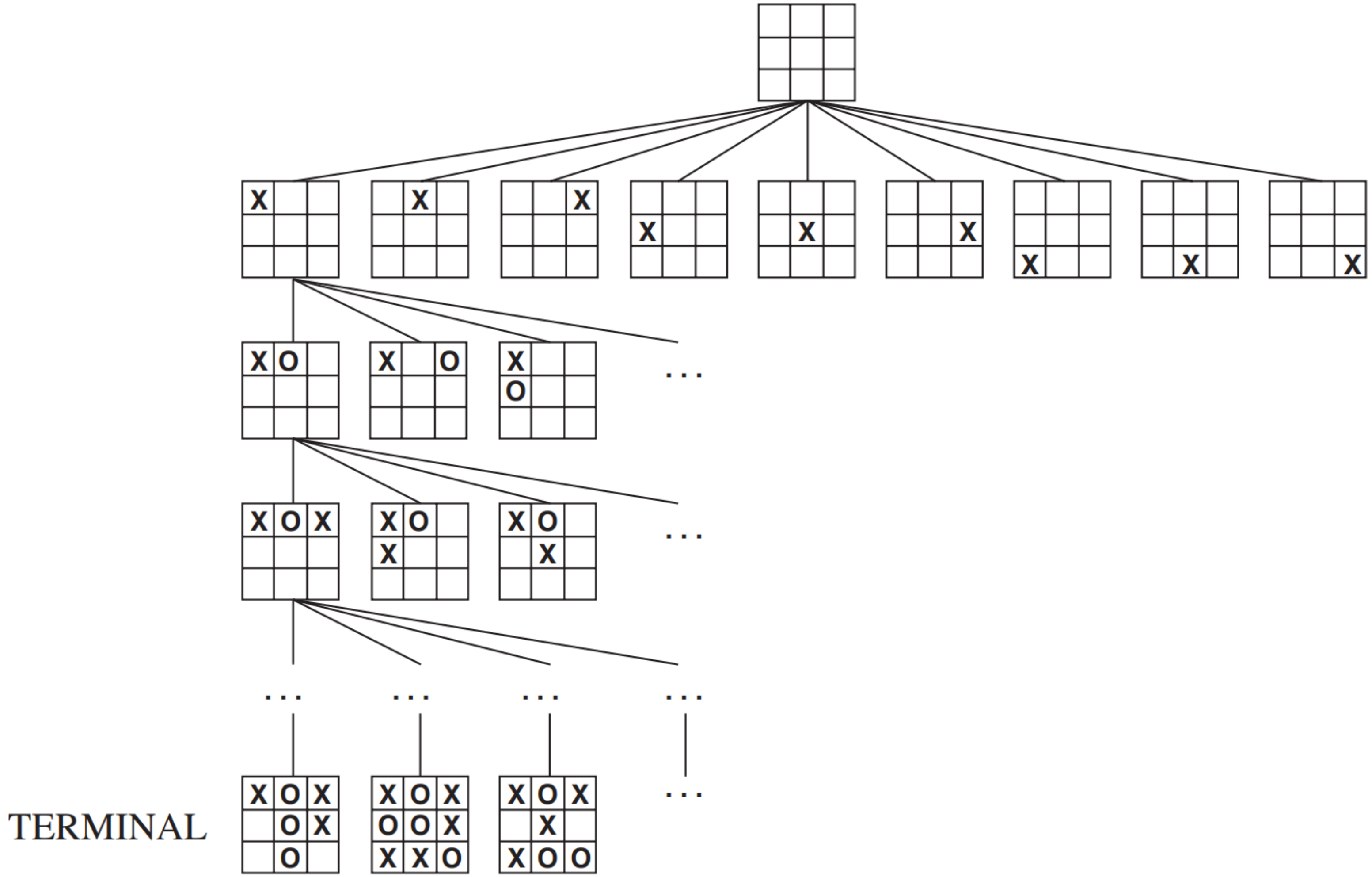


Tic Tac Toe

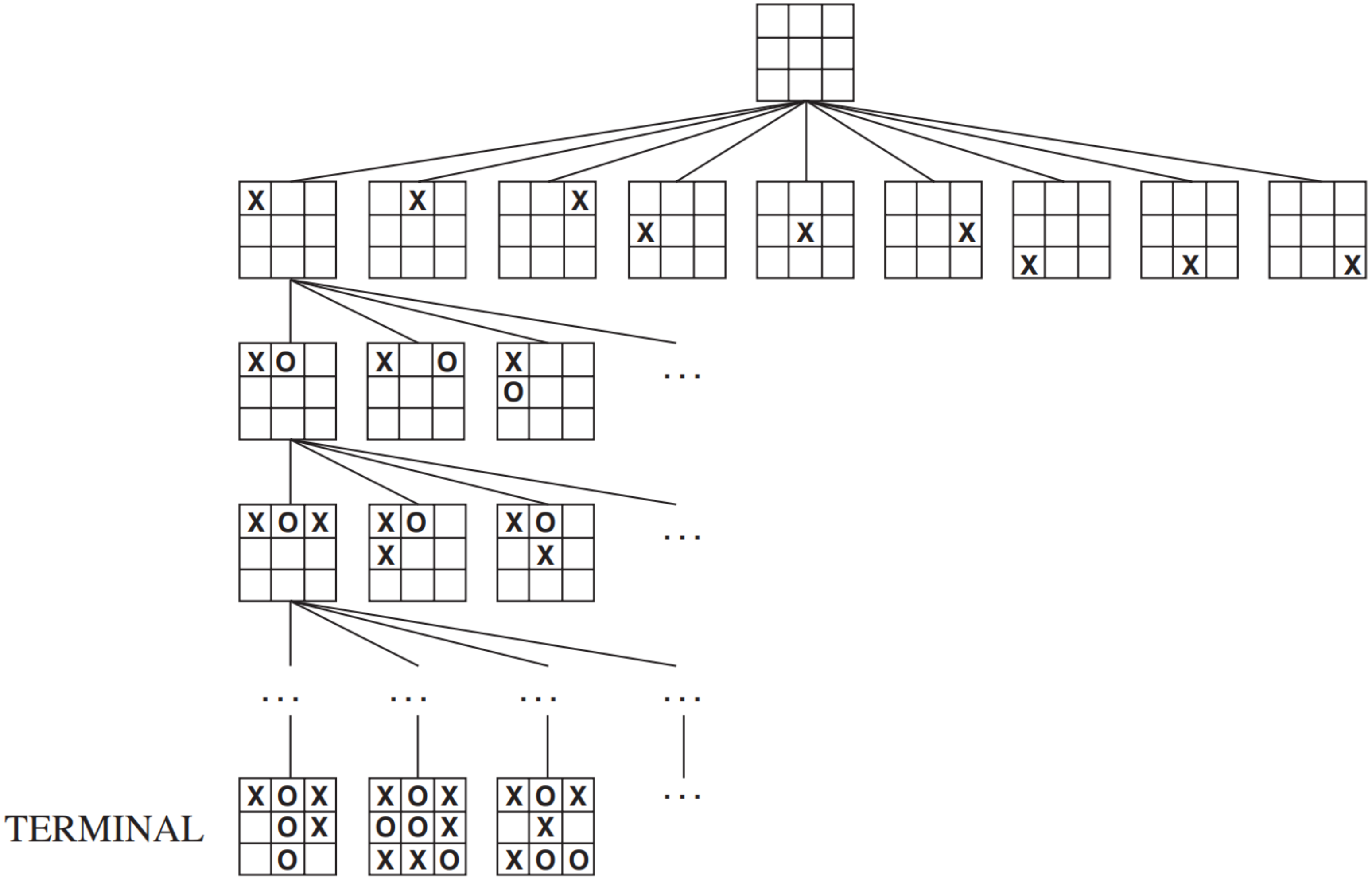


Tic Tac Toe

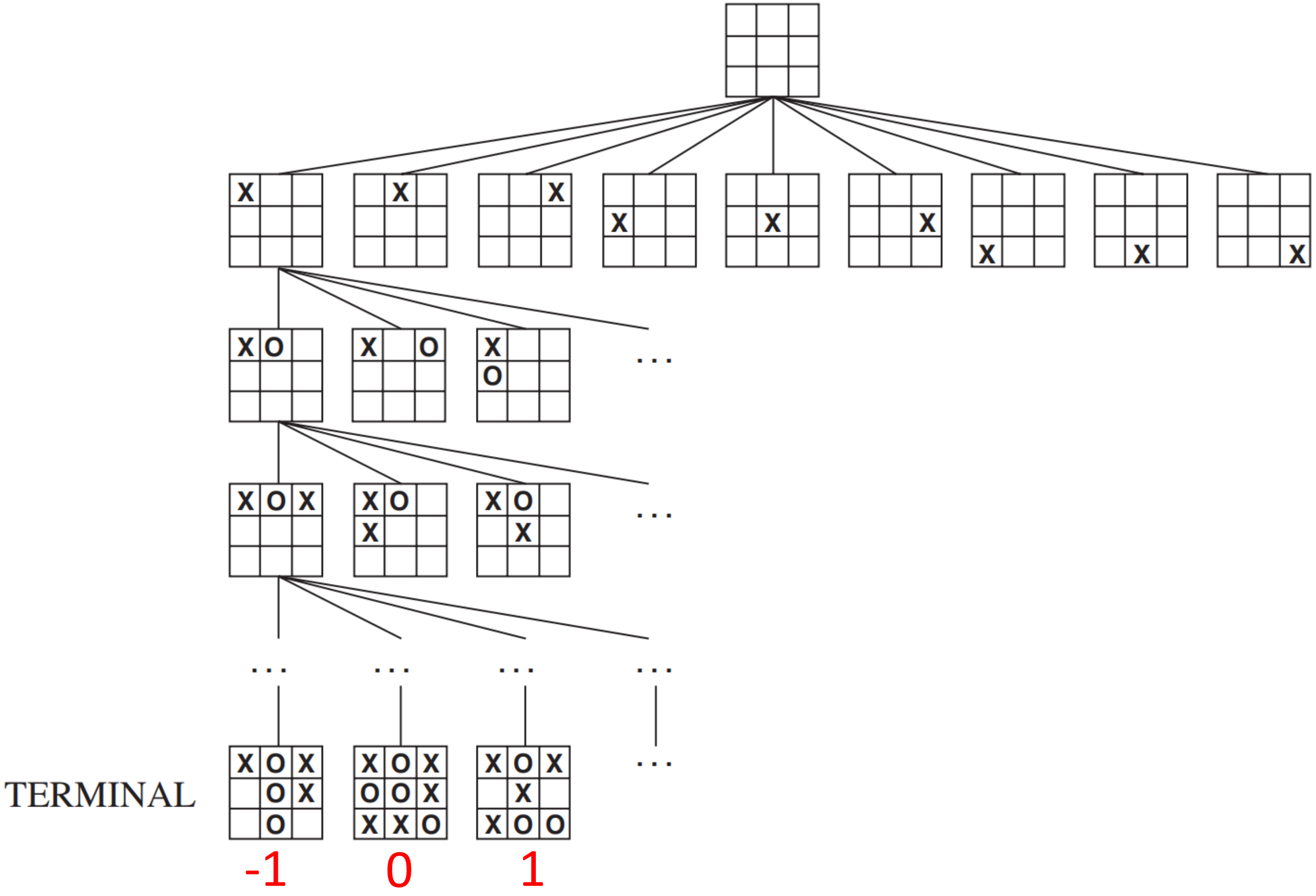
(Still using states and actions formulaton)



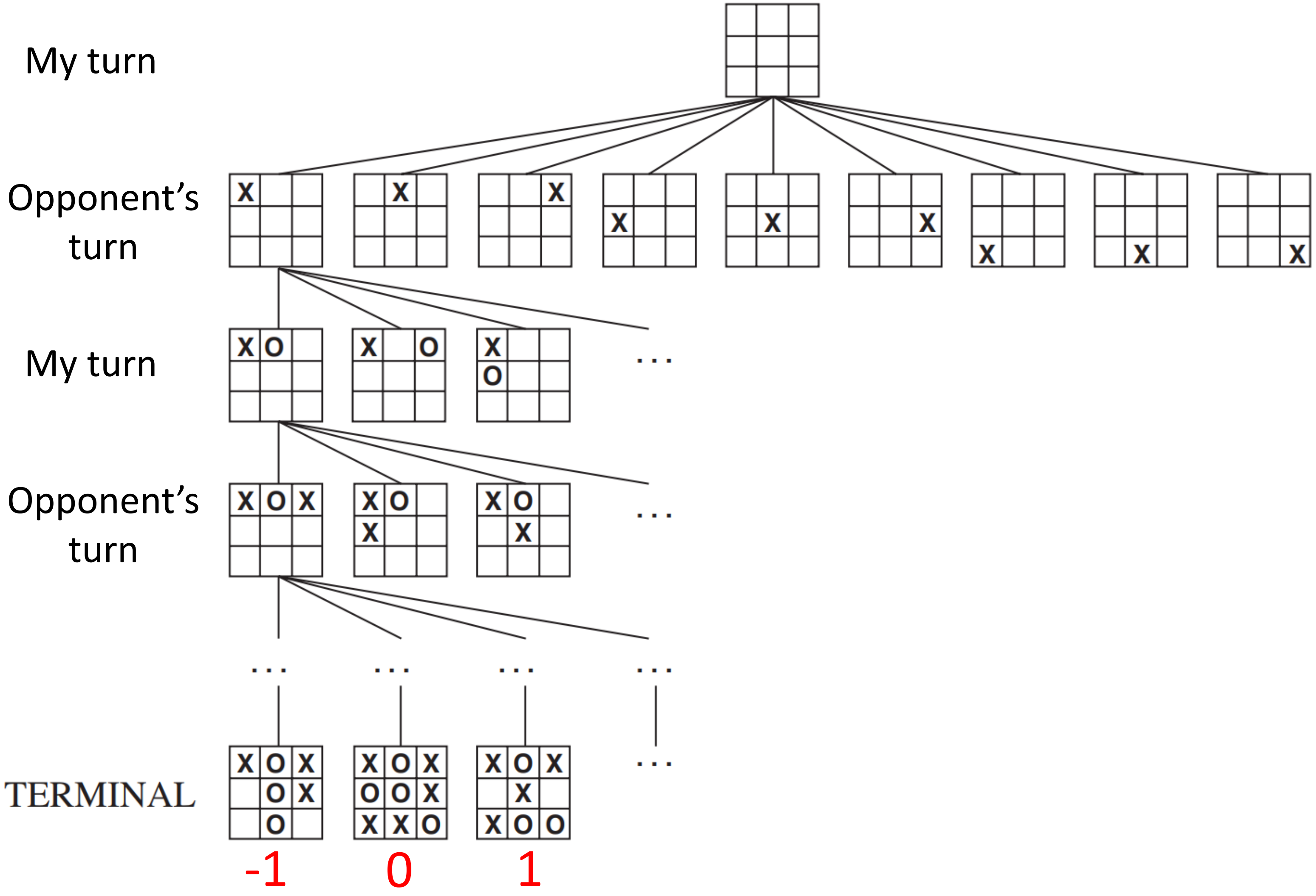
Tic Tac Toe



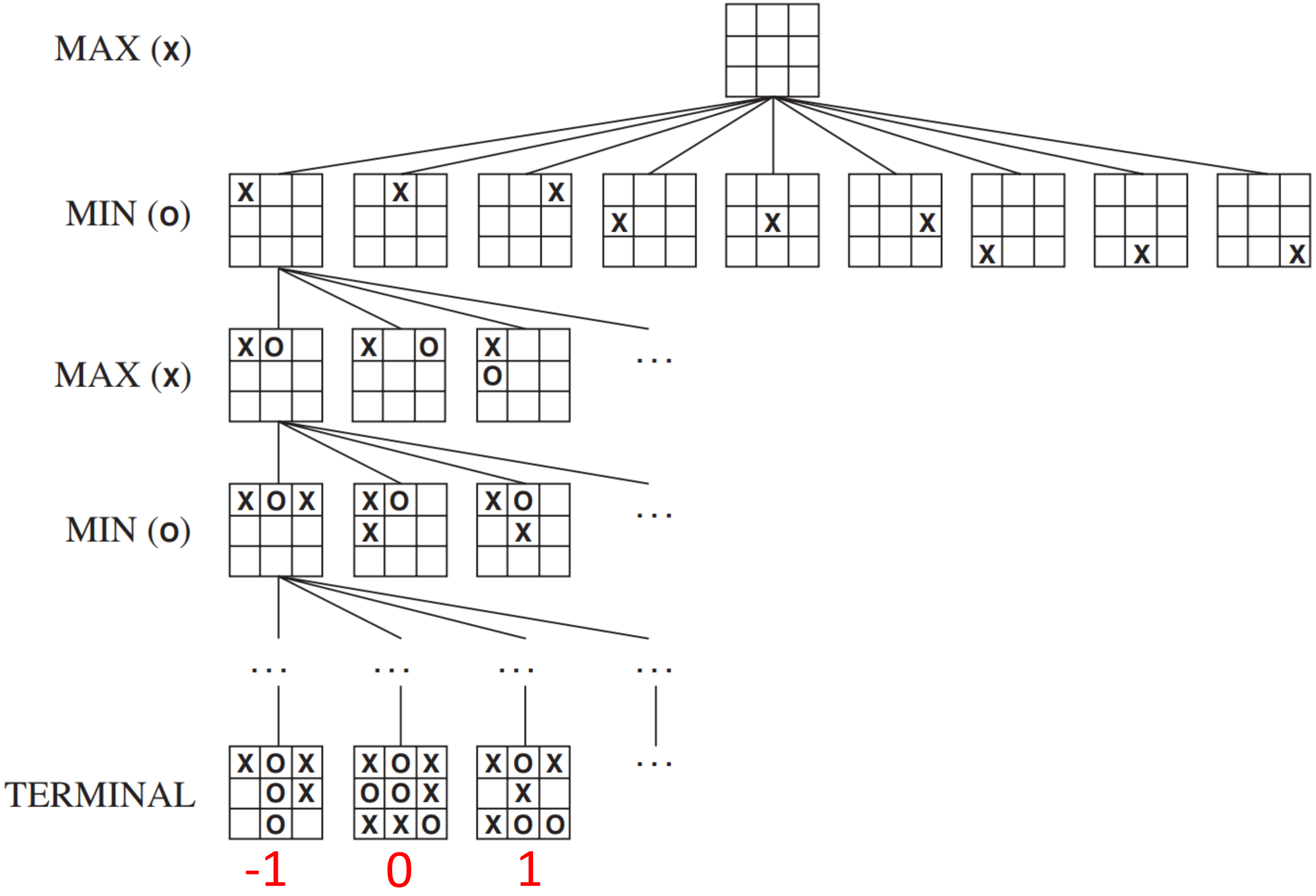
Tic Tac Toe



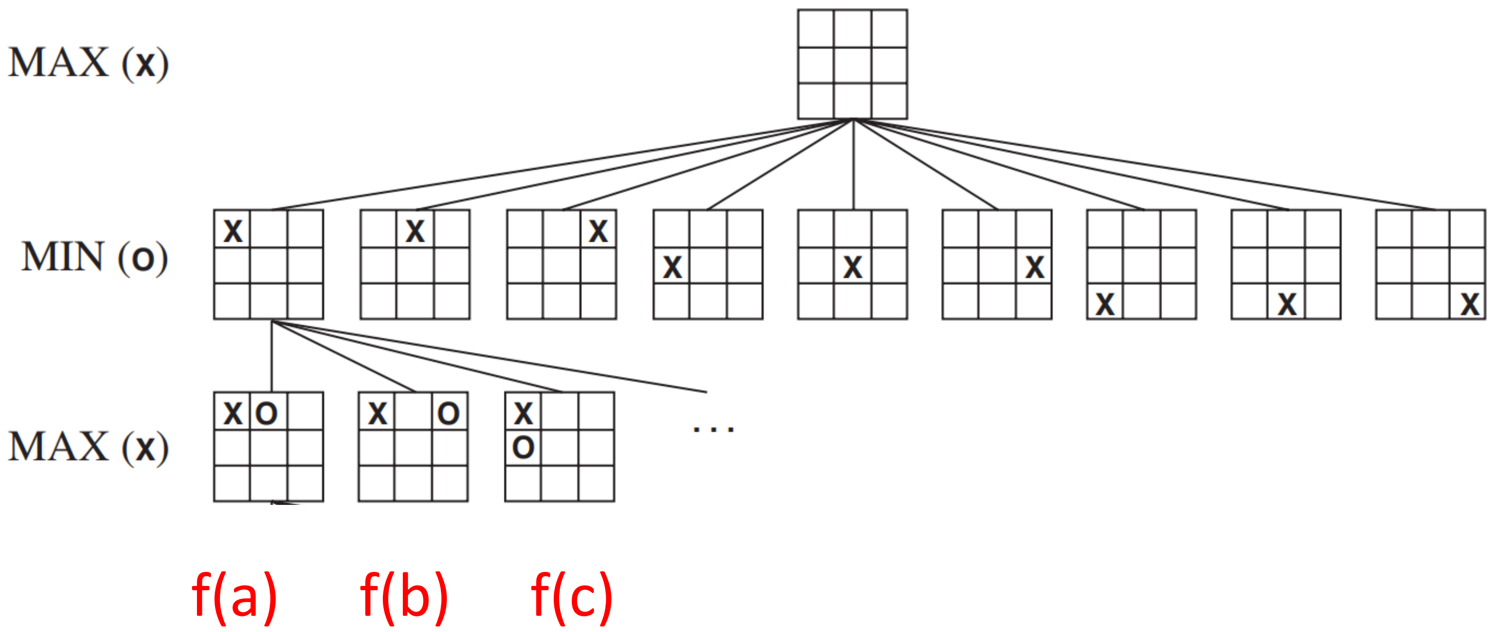
Tic Tac Toe



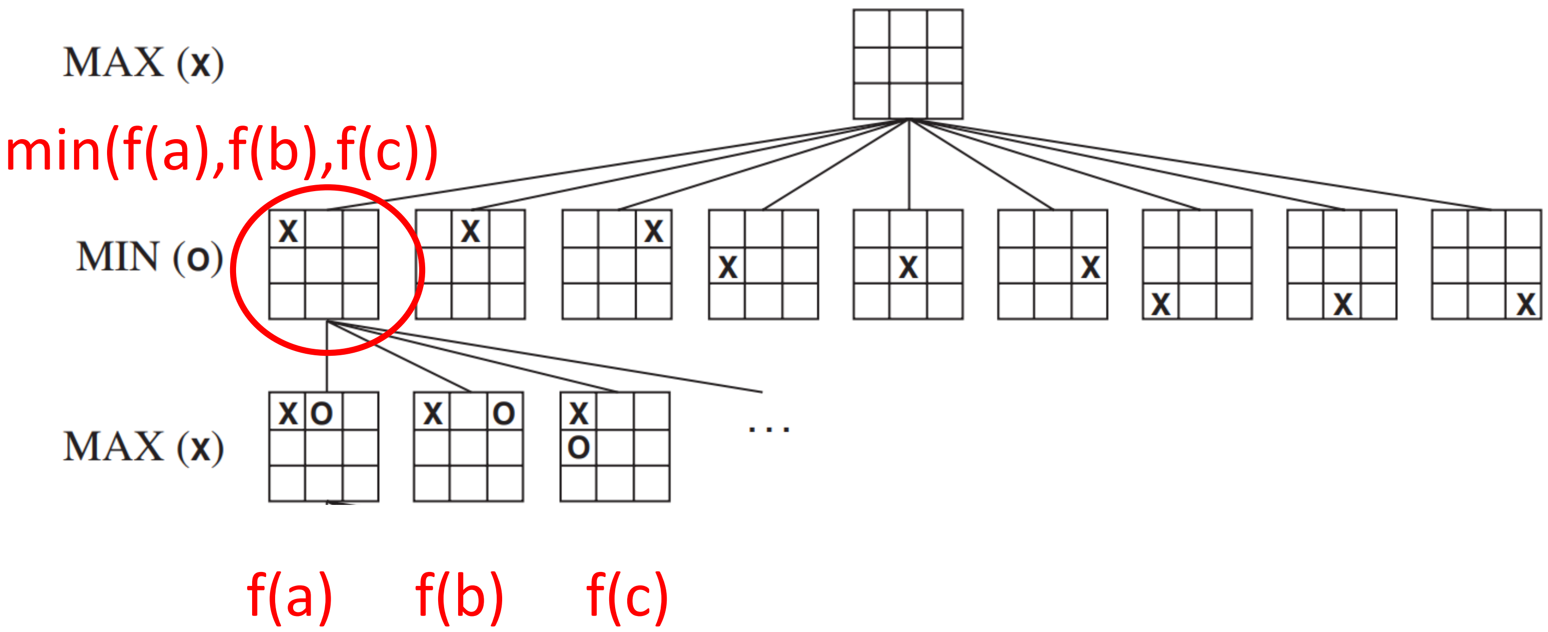
Tic Tac Toe



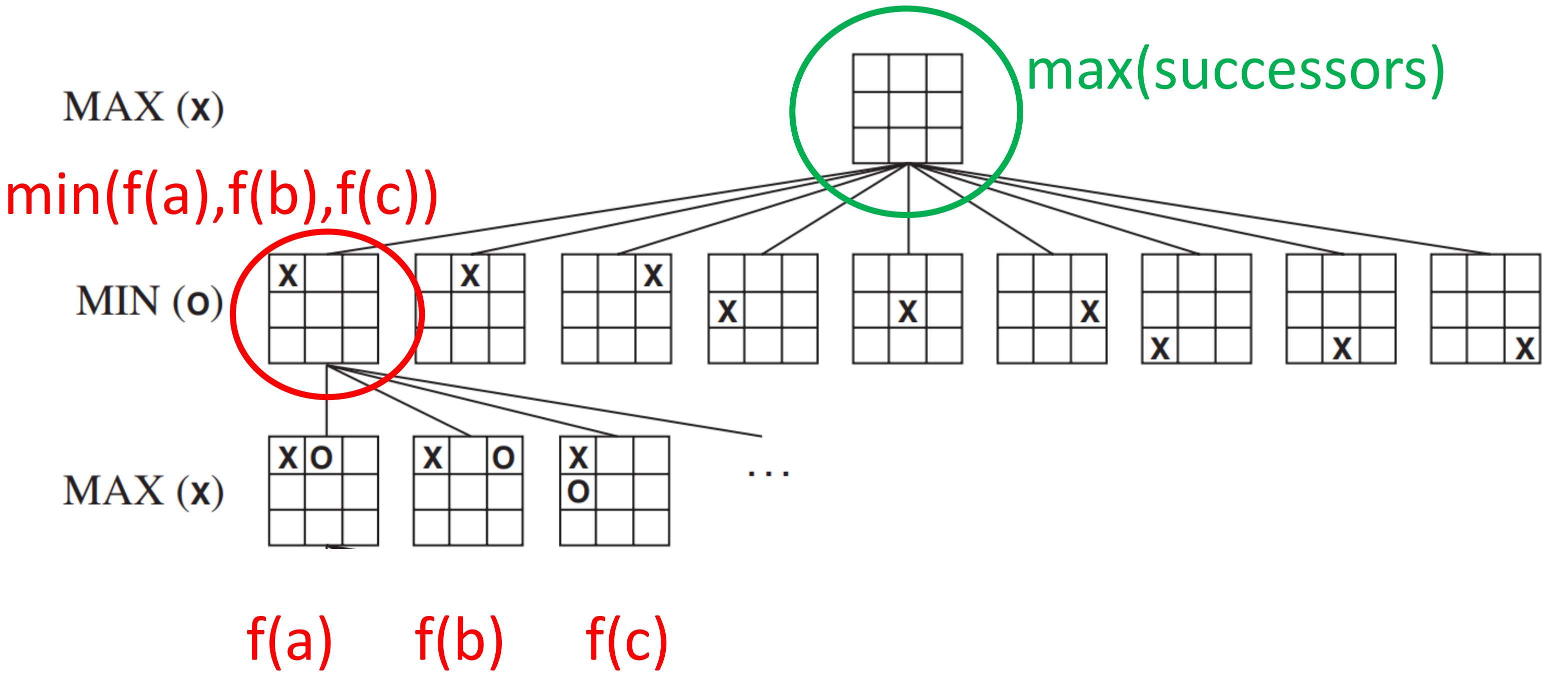
Tic Tac Toe



Tic Tac Toe



Tic Tac Toe



Adversarial Search

(Initial) assumptions:

- On my turn I consider all moves and pick the one that's best for me
- My opponent will consider all moves and the pick the one that's worst for me

$$f_{opponent}(s) = -f_{me}(s)$$

Adversarial Search

(Initial) assumptions:

- On my turn I consider all moves and pick the one that's best for me
- My opponent will consider all moves and the pick the one that's worst for me

$f_{opponent}(s) = - f_{me}(s)$ "zero sum"

Adversarial Search

(Initial) assumptions:

- On my turn I consider all moves and pick the one that's best for me
- My opponent will consider all moves and the pick the one that's worst for me

$f_{opponent}(s) = -f_{me}(s)$ "zero sum"

We'll use f(s) whenever we're talking about $f_{me}(s)$, and

-f(s) whenever we're talking about $f_{opponent}(s)$

Adversarial Search

(Initial) assumptions:

- On my turn I consider all moves and pick the one that's best for me
- My opponent will consider all moves and the pick the one that's worst for me

$f_{opponent}(s) = -f_{me}(s)$ "zero sum"

We'll use f(s) whenever we're talking about $f_{me}(s)$, and
-f(s) whenever we're talking about $f_{opponent}(s)$

My turn: max f(s) for all successors s (pick my best move)

Opponent's turn: min f(s) for all successors s (pick my worst move)

Minimax Value of a Game

"Ground truth"

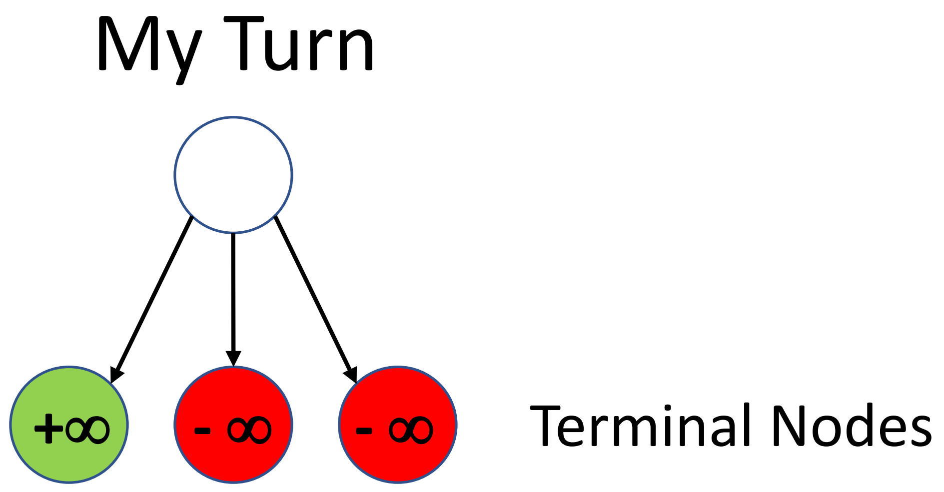
V(s): What the true value of a game state is

Given for terminal nodes

Computed from non-terminal nodes

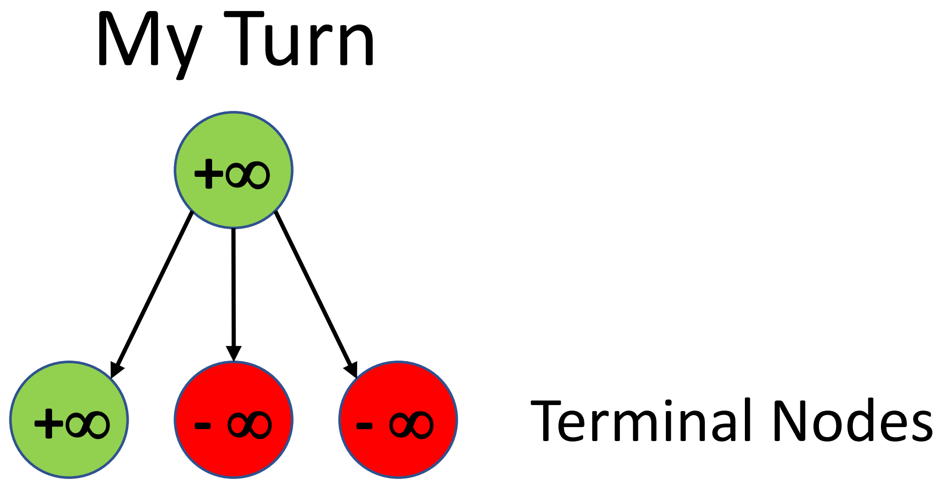
Minimax Value of a Game

- I win = $+\infty$
- I lose = $-\infty$



Minimax Value of a Game

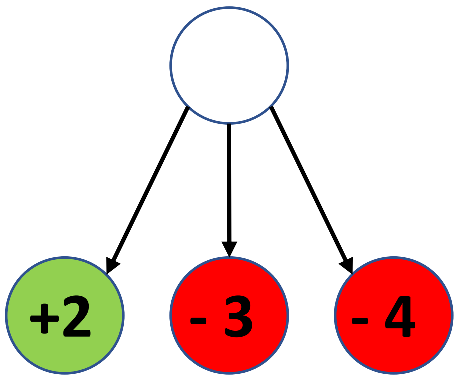
- I win = $+\infty$
- I lose = $-\infty$



Minimax Value of a Game

- V(s) = value of win (to me)

My Turn

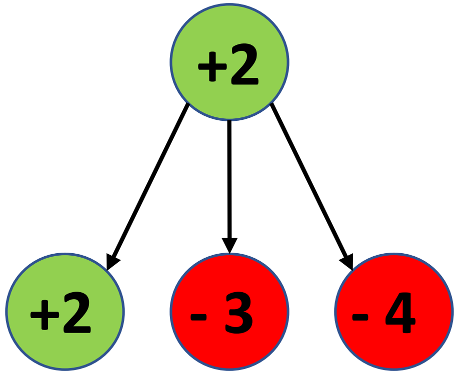


Terminal Nodes

Minimax Value of a Game

- V(s) = value of win (to me)

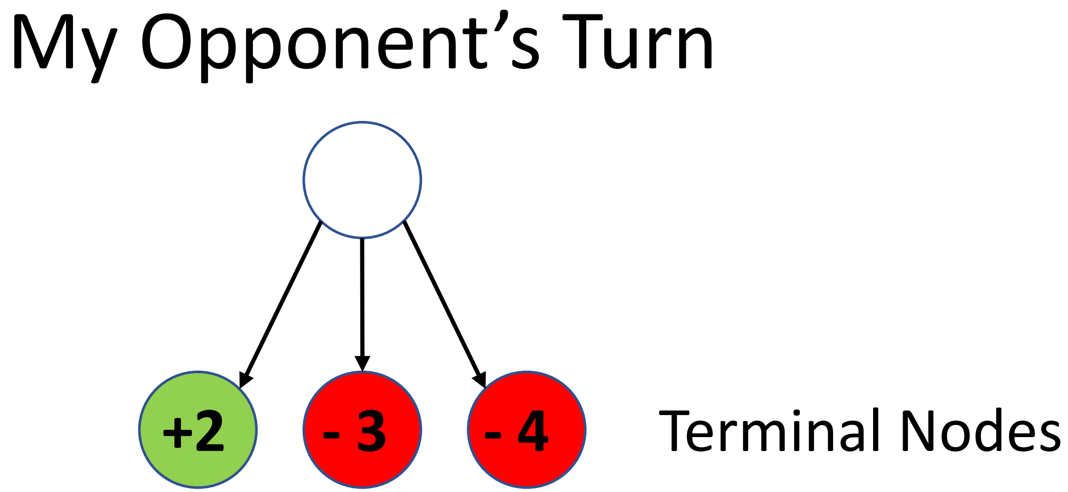
My Turn



Terminal Nodes

Minimax Value of a Game

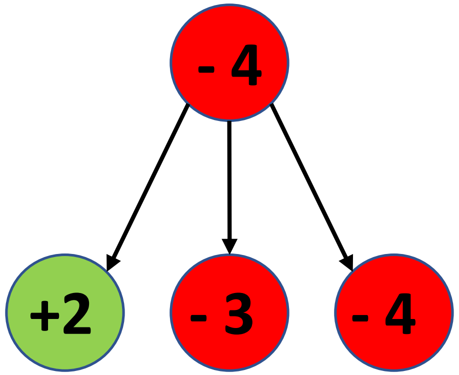
- V(s) = value of win (to me)



Minimax Value of a Game

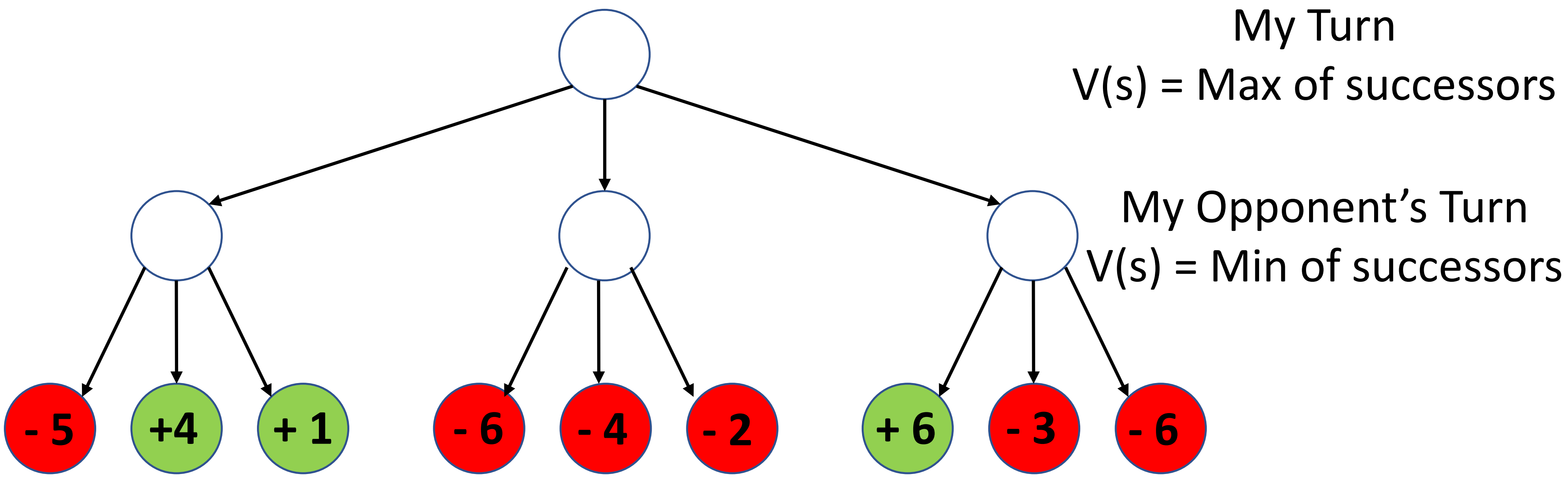
- V(s) = value of win (to me)

My Opponent’s Turn

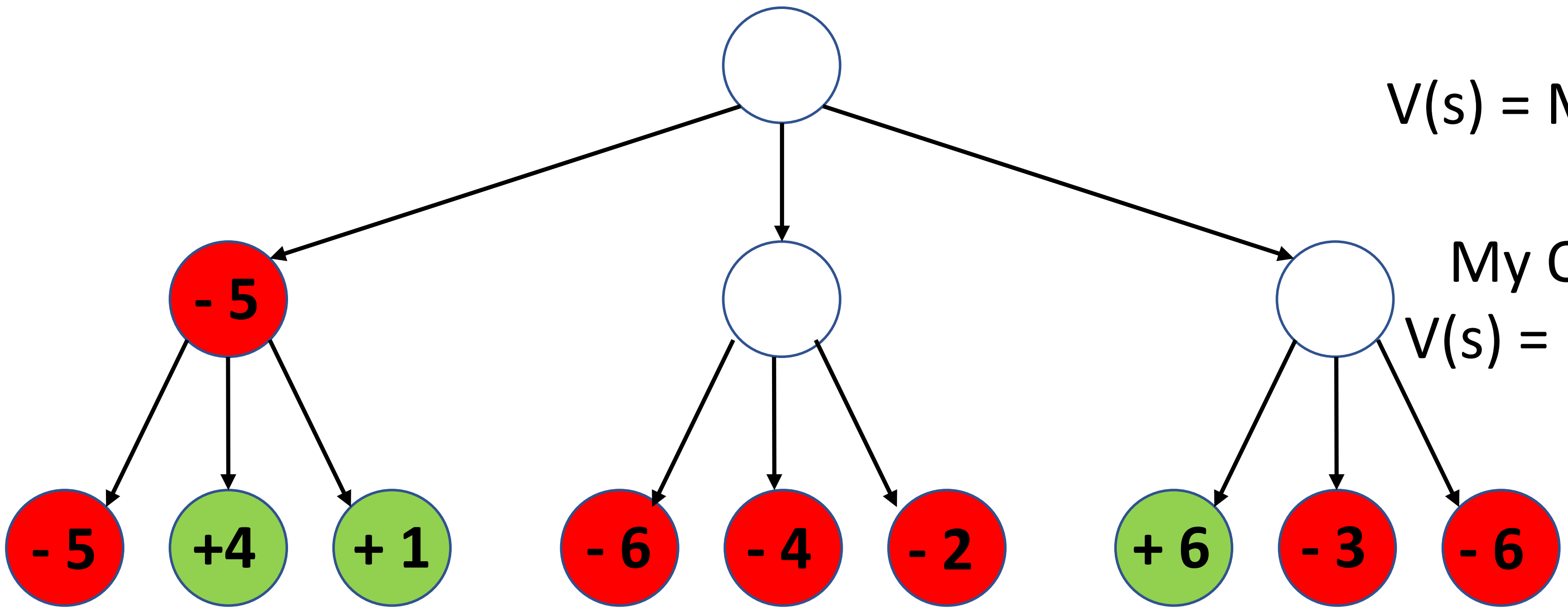


Terminal Nodes

Minimax Value of a Game



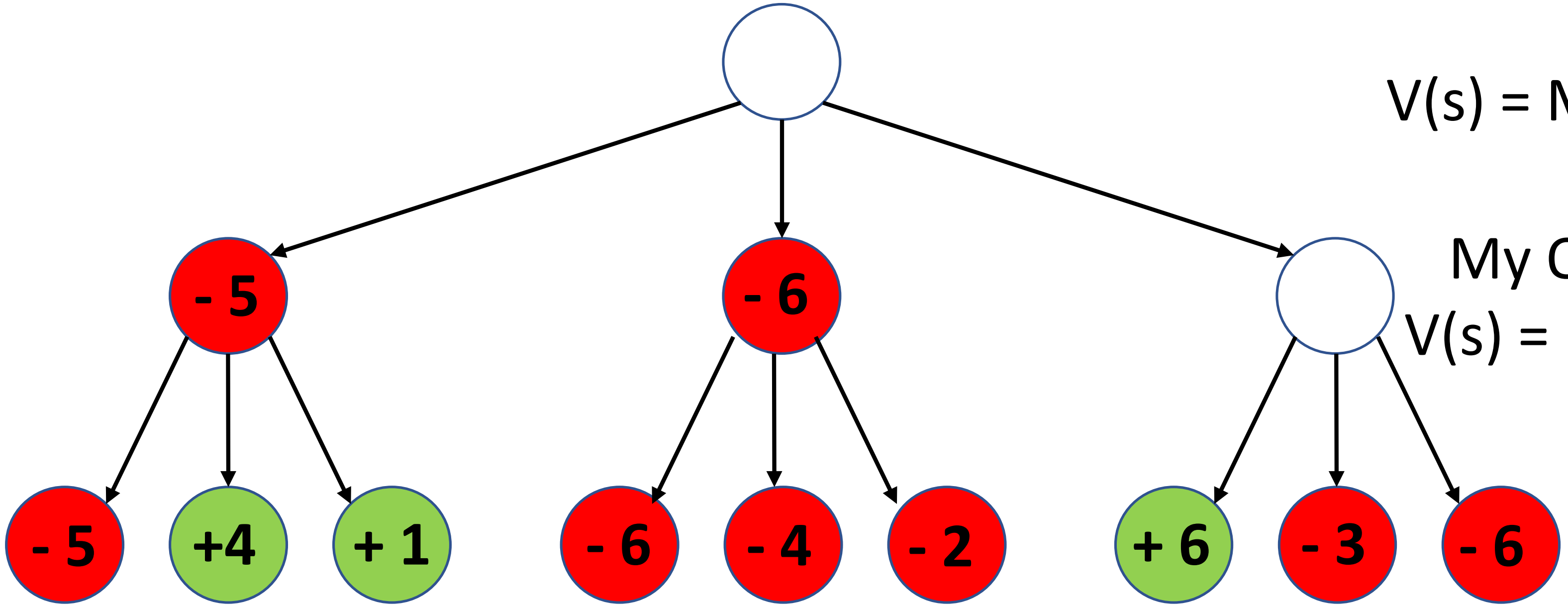
Minimax Value of a Game



My Turn
V(s) = Max of successors

My Opponent's Turn
V(s) = Min of successors

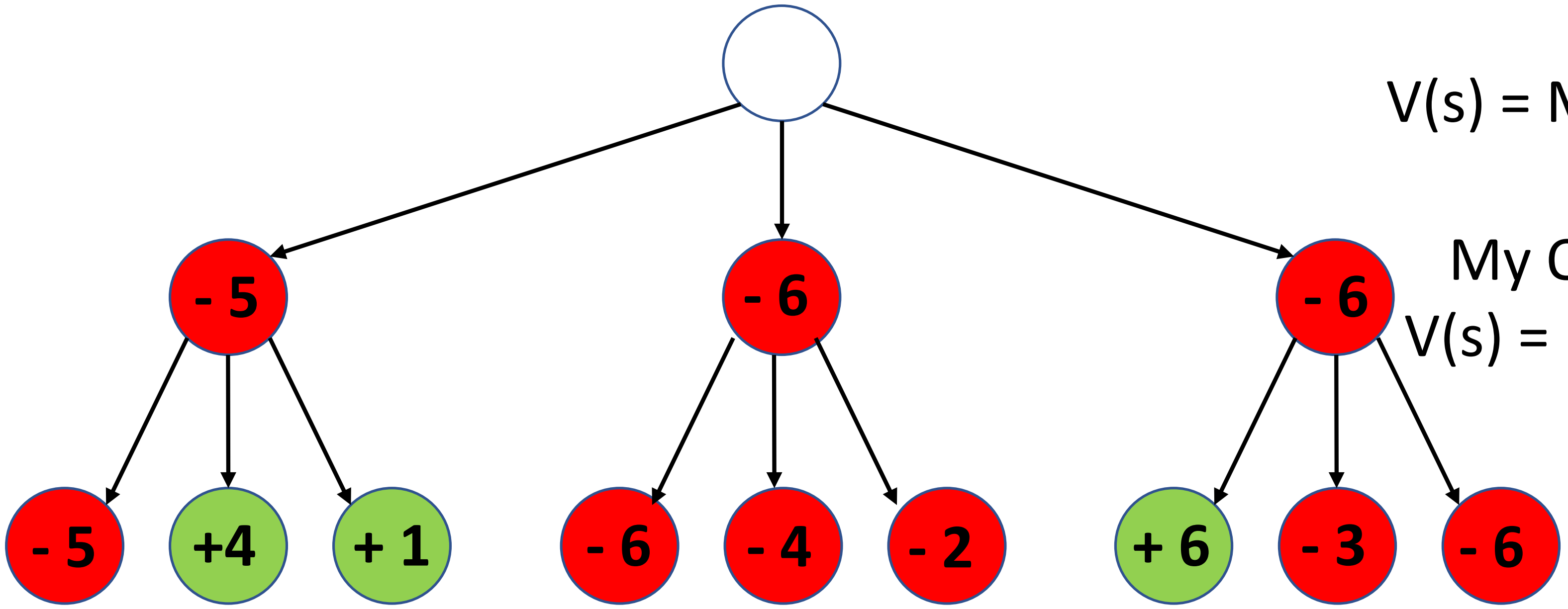
Minimax Value of a Game



My Turn
V(s) = Max of successors

My Opponent’s Turn
V(s) = Min of successors

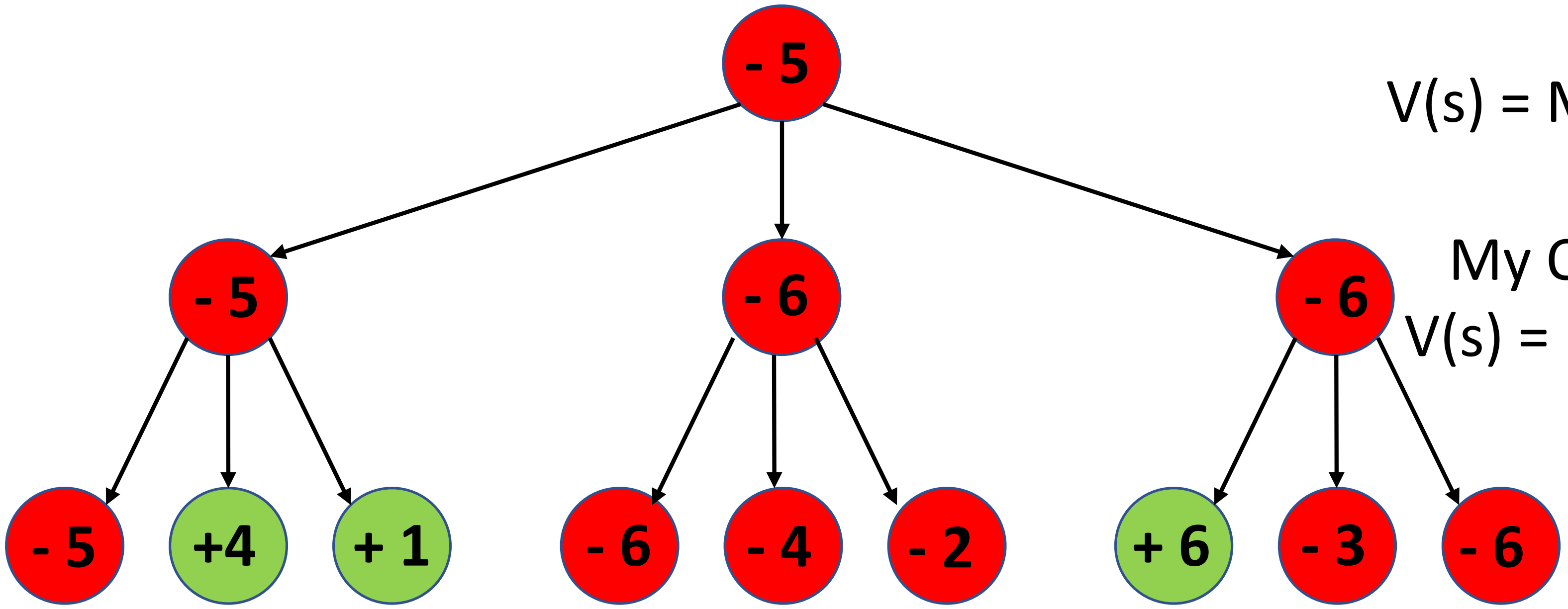
Minimax Value of a Game



My Turn
V(s) = Max of successors

My Opponent's Turn
V(s) = Min of successors

Minimax Value of a Game



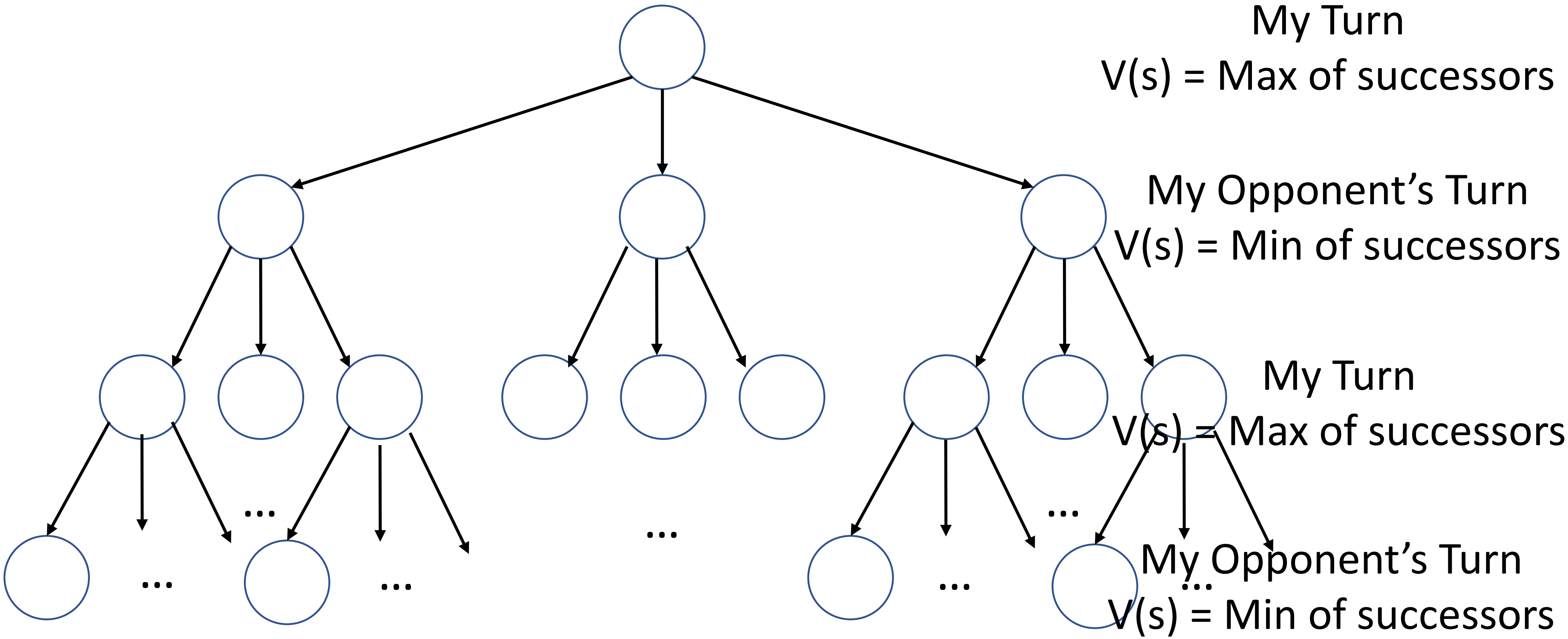
My Turn
V(s) = Max of successors

My Opponent's Turn
V(s) = Min of successors

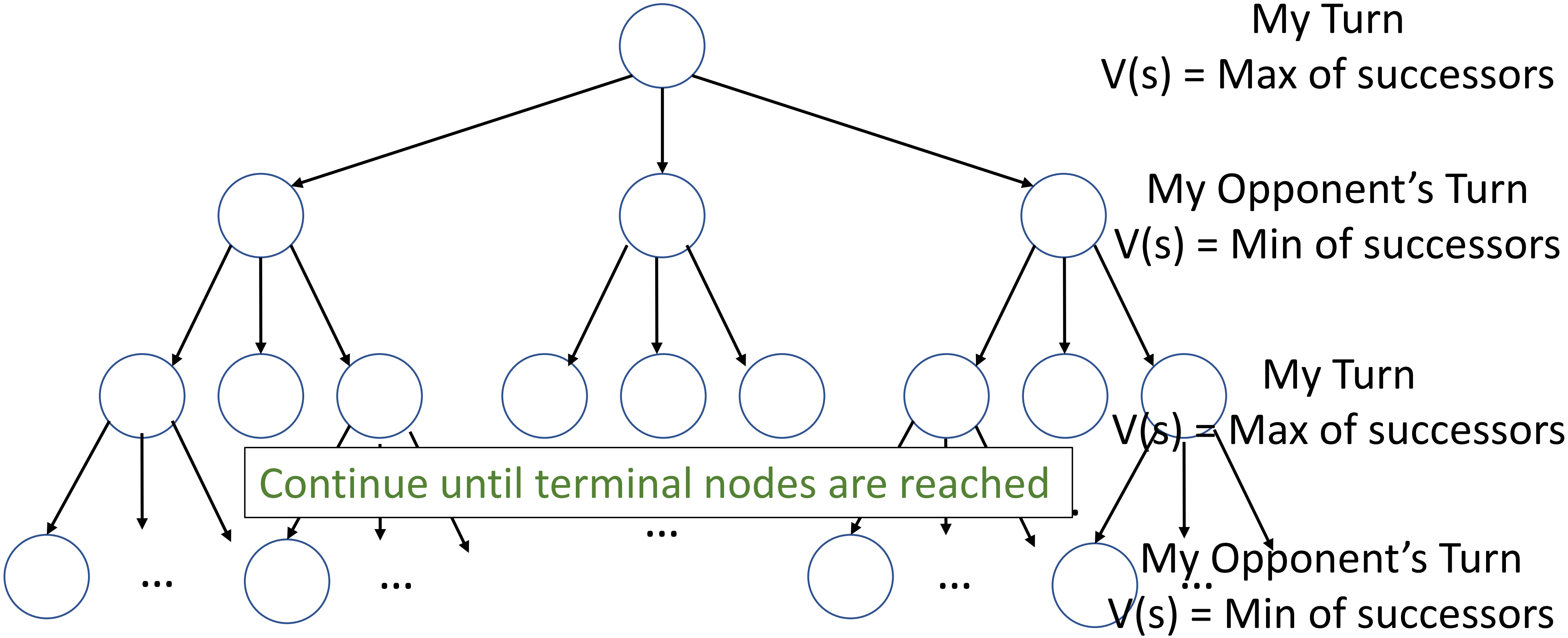
Minimax Value of a Game

- Current state: s
- Available operators: ops
- Value of a terminal state: V(s)
- My turn:
  - Value of state s = V(s) = $\max_{o \in ops} \{V(\text{apply}(s, o))\}$
  - Best move = $\operatorname{argmax}_{o \in ops} \{V(\text{apply}(s, o))\}$
- Opponent's turn:
  - Value of state s = V(s) = $\min_{o \in ops} \{V(\text{apply}(s, o))\}$
  - Best move = = $\operatorname{argmin}_{o \in ops} \{V(\text{apply}(s, o))\}$

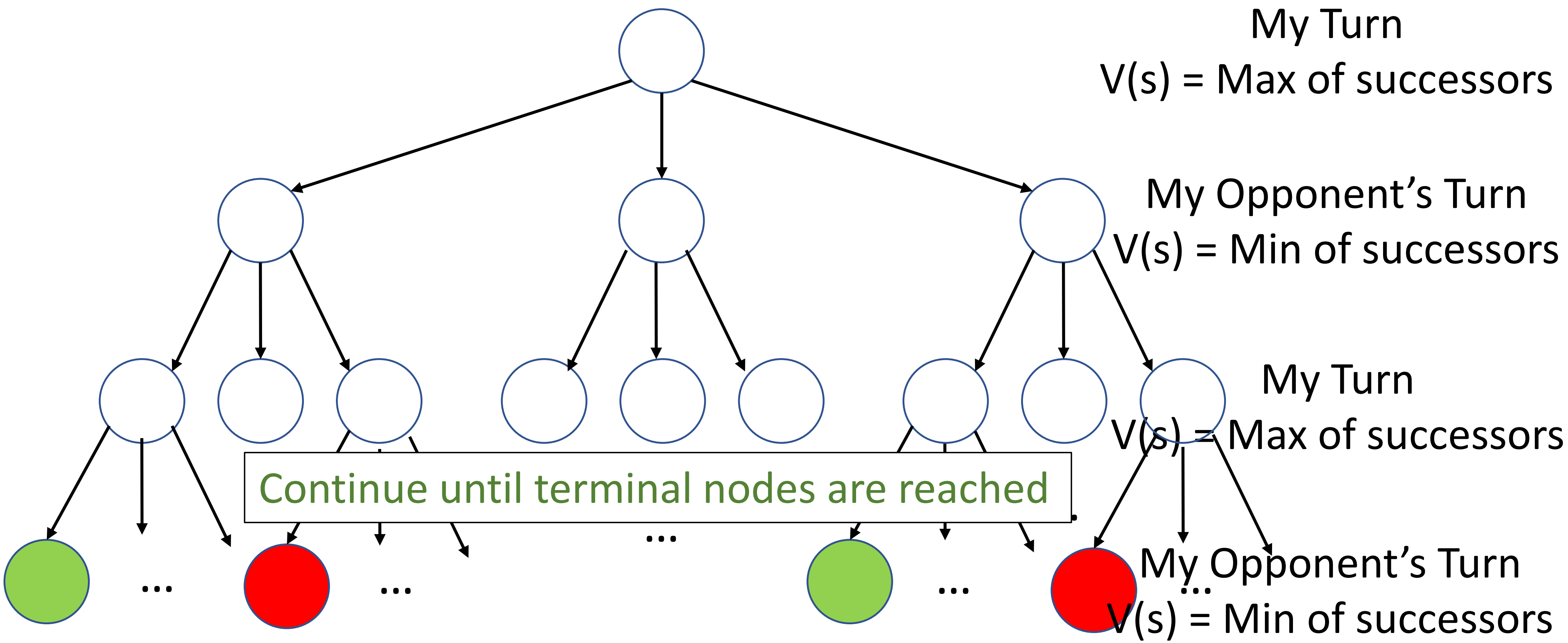
Minimax Value of a Game



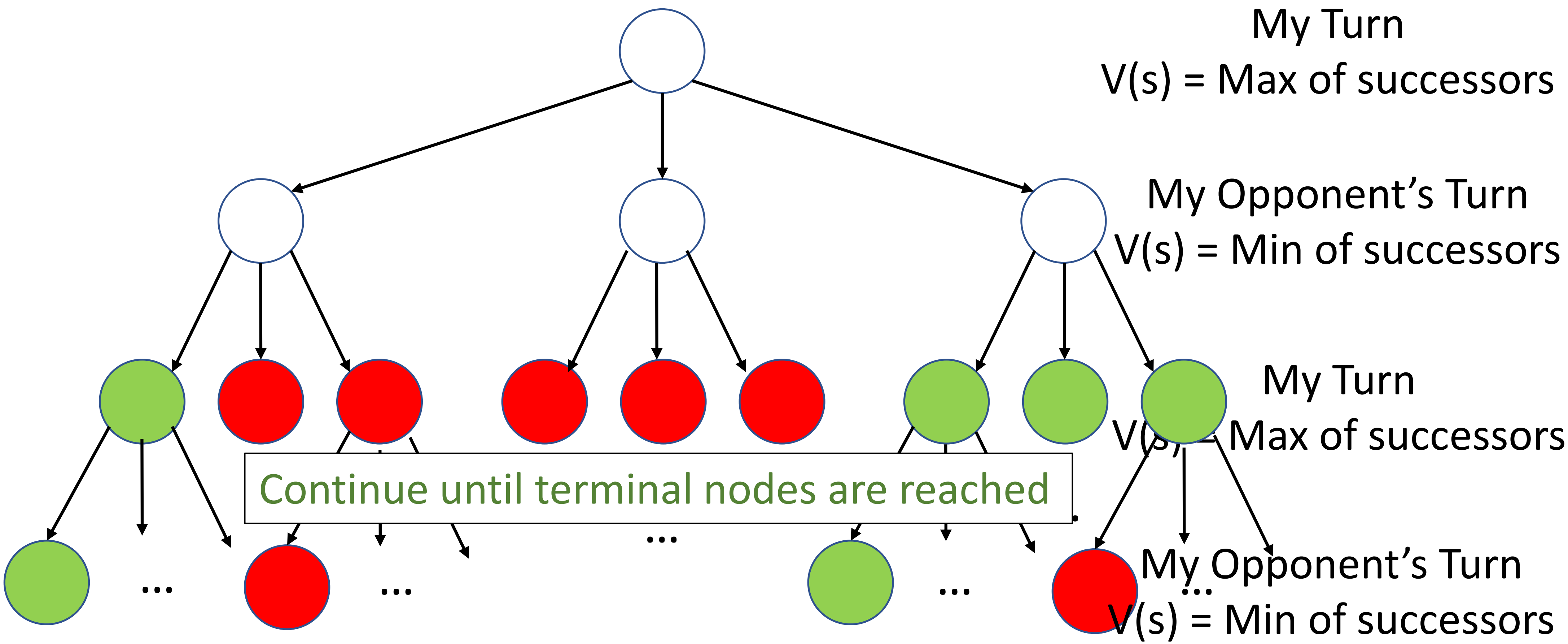
Minimax Value of a Game



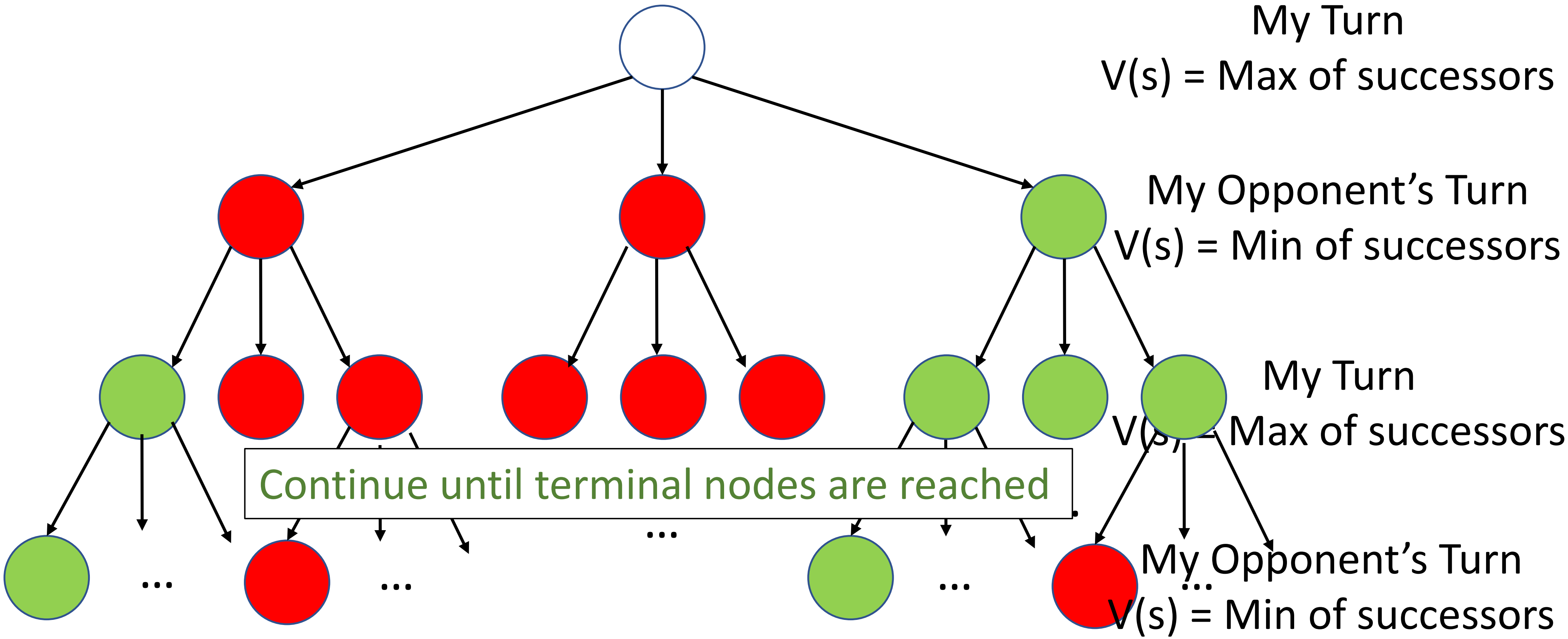
Minimax Value of a Game



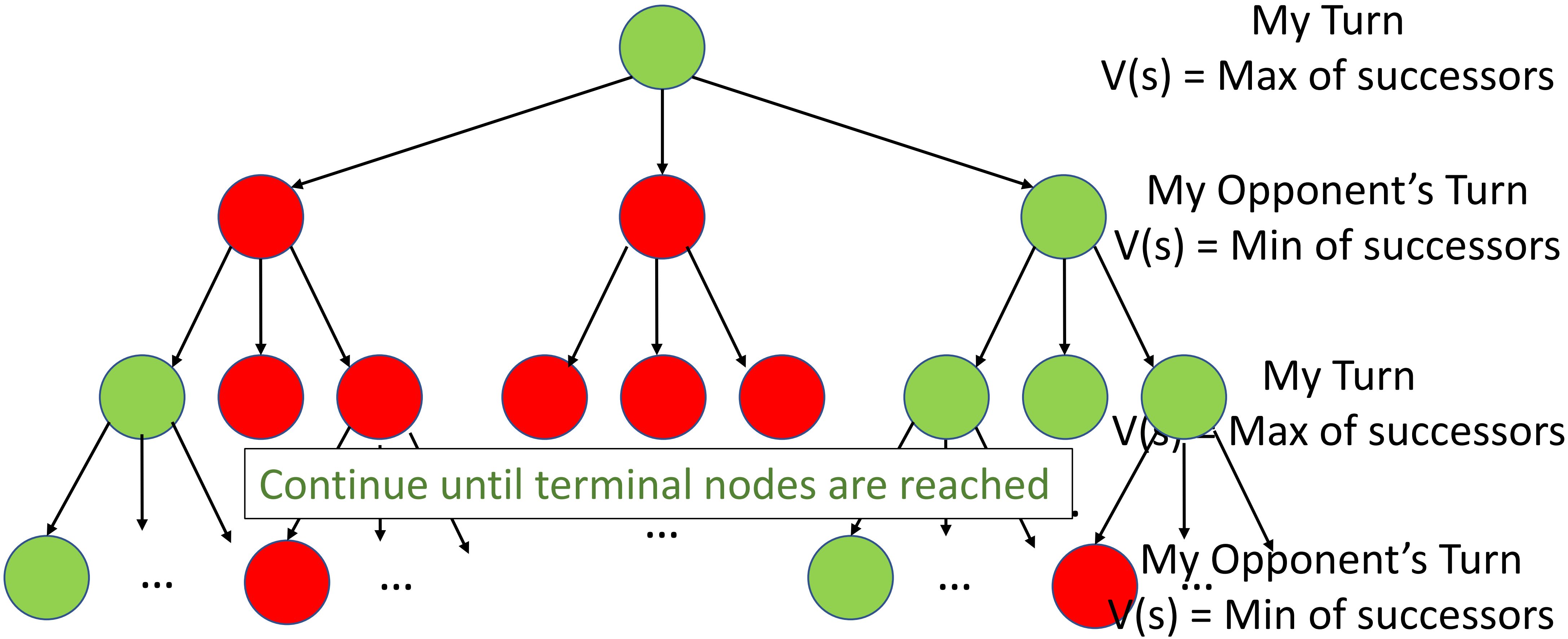
Minimax Value of a Game



Minimax Value of a Game



Minimax Value of a Game



Minimax Algorithm

Initial call:

- My turn: maxturn(initial-state,ops)
- Opponent's turn: minturn(initial-state,ops)

Minimax Algorithm

```
maxturn(s,ops):
    if terminal(s) then return V(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val’ ← minturn(apply(s,o),ops);
            if val’ > val then val ← val’; bestop ← o;
        return val
```

Minimax Algorithm

```
maxturn(s,ops):
    if terminal(s) then return V(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val’ ← minturn(apply(s,o),ops);
            if val’ > val then val ← val’; bestop ← o;
        return val
```

Shorthand for return a data structure with the value, the operator that took you here, possibly other info

Minimax Algorithm

```
maxturn(s,ops):
    if terminal(s) then return V(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val’ ← minturn(apply(s,o),ops);
            if val’ > val then
                val ← val’;
                bestop ← o;
        return val
```

```
minturn(s,ops):
    if terminal(s) then return V(s)
    else
        val ← +∞;
        foreach o ∈ ops
            val’ ← maxturn(apply(s,o),ops);
            if val’ < val then
                val ← val’;
                bestop ← o;
        return val
```

Minimax Algorithm (Complete Search)

```
maxturn(s,ops):
    if terminal(s) then return V(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val' ← minturn(apply(s,o),ops);
            if val' > val then
                val ← val';
                bestop ← o;
        return val
```

```
minturn(s,ops):
    if terminal(s) then return V(s)
    else
        val ← +∞;
        foreach o ∈ ops
            val' ← maxturn(apply(s,o),ops);
            if val' < val then
                val ← val';
                bestop ← o;
        return val
```

Minimax Algorithm (Complete Search)

```
maxturn(s,ops):
    if terminal(s) then return V(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val' ← minturn(apply(s,o),ops);
            if val' > val then
                val ← val';
                bestop ← o;
        return val
```

```
minturn(s,ops):
    if terminal(s) then return V(s)
    else
        val ← +∞;
        foreach o ∈ ops
            val' ← maxturn(apply(s,o),ops);
            if val' < val then
                val ← val';
                bestop ← o;
        return val
```

Minimax Search

- Complete search:
  Generally intractable to go all the way to terminal nodes

- Key idea:
  Use a heuristic evaluation function $f(s)$ that applies to intermediate states and returns a number that estimates the value of s

Minimax Algorithm (Complete Search)

```
maxturn(s,ops):

    if terminal(s) then return V(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val' ← minturn(apply(s,o),ops);
            if val' > val then
                val ← val';
                bestop ← o;
        return val
```

```
minturn(s,ops):

    if terminal(s) then return V(s)
    else
        val ← +∞;
        foreach o ∈ ops
            val' ← maxturn(apply(s,o),ops);
            if val' < val then
                val ← val';
                bestop ← o;
        return val
```

Initial call:

- If I go first: maxturn(initial-state,ops)
- If opponent goes first: minturn(initial-state,ops)

Minimax Algorithm (Heuristic Search)

```
maxturn(s,ops):
    if cutoff(s) then return f(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val' ← minturn(apply(s,o),ops);
            if val' > val then
                val ← val';
                bestop ← o;
        return val
```

```
minturn(s,ops):
    if cutoff(s) then return f(s)
    else
        val ← +∞;
        foreach o ∈ ops
            val' ← maxturn(apply(s,o),ops);
            if val' < val then
                val ← val';
                bestop ← o;
        return val
```

Initial call:

- If I go first: maxturn(initial-state,ops)
- If opponent goes first: minturn(initial-state,ops)

Minimax Algorithm (Heuristic Search)

```
maxturn(s,ops):
    if cutoff(s) then return f(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val' ← minturn(apply(s,o),ops);
            if val' > val then
                val ← val';
                bestop ← o;
        return val
```

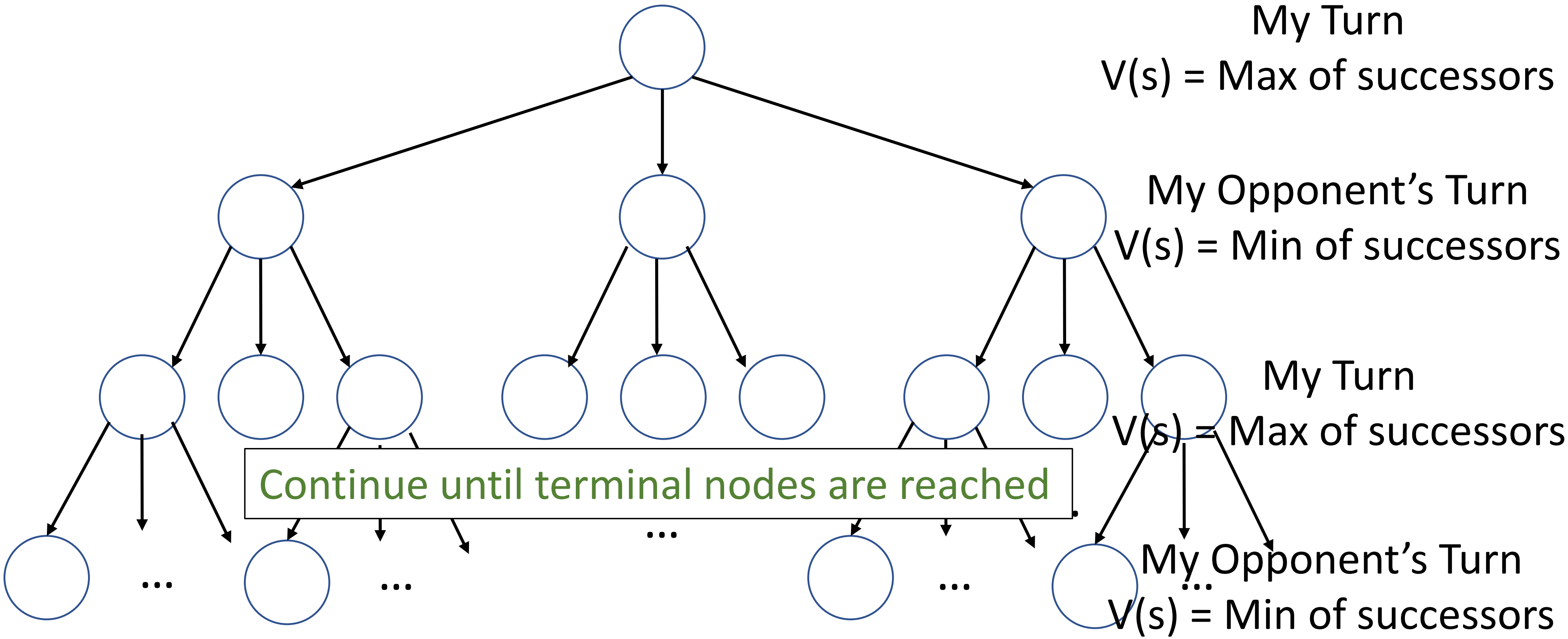
```
minturn(s,ops):
    if cutoff(s) then return f(s)
    else
        val ← +∞;
        foreach o ∈ ops
            val' ← maxturn(apply(s,o),ops);
            if val' < val then
                val ← val';
                bestop ← o;
        return val
```

Initial call:

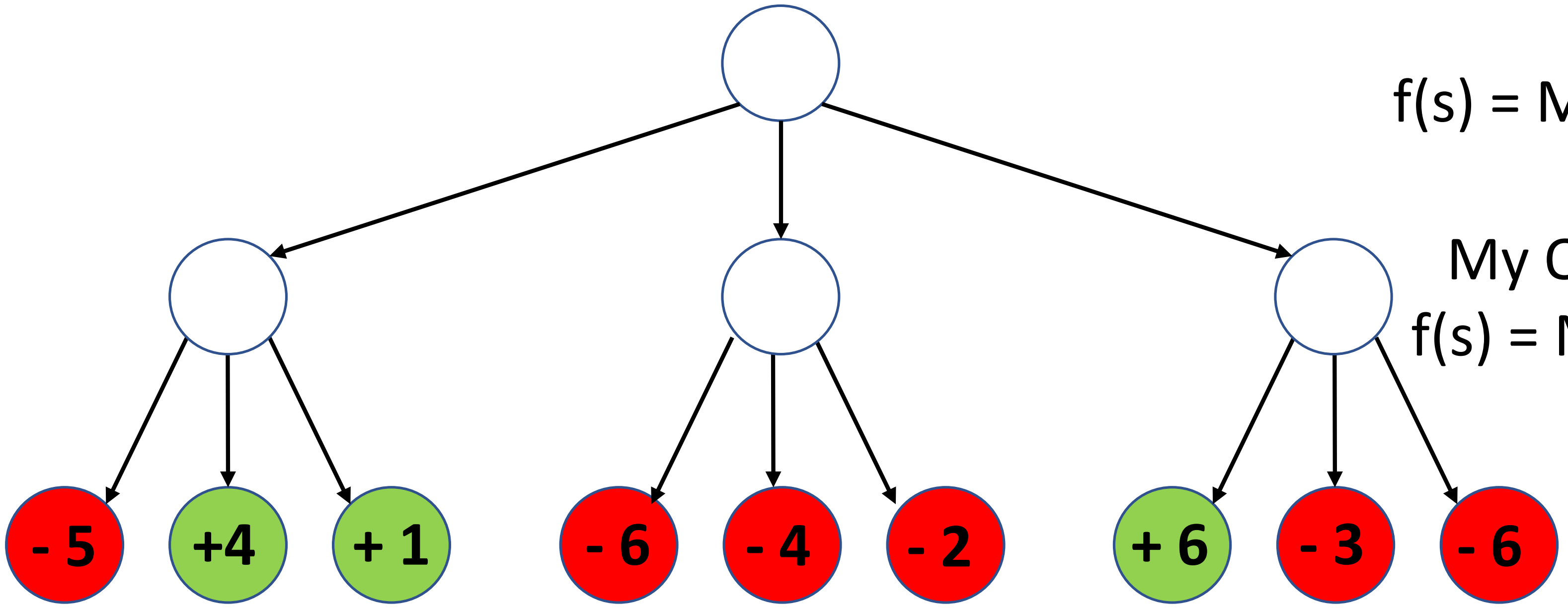
- If I go first: maxturn(initial-state,ops)
- If opponent goes first: minturn(initial-state,ops)

cutoff(s) could be a depth-bound or something more sophisticated

Minimax Value of a Game



Minimax Value of a Game

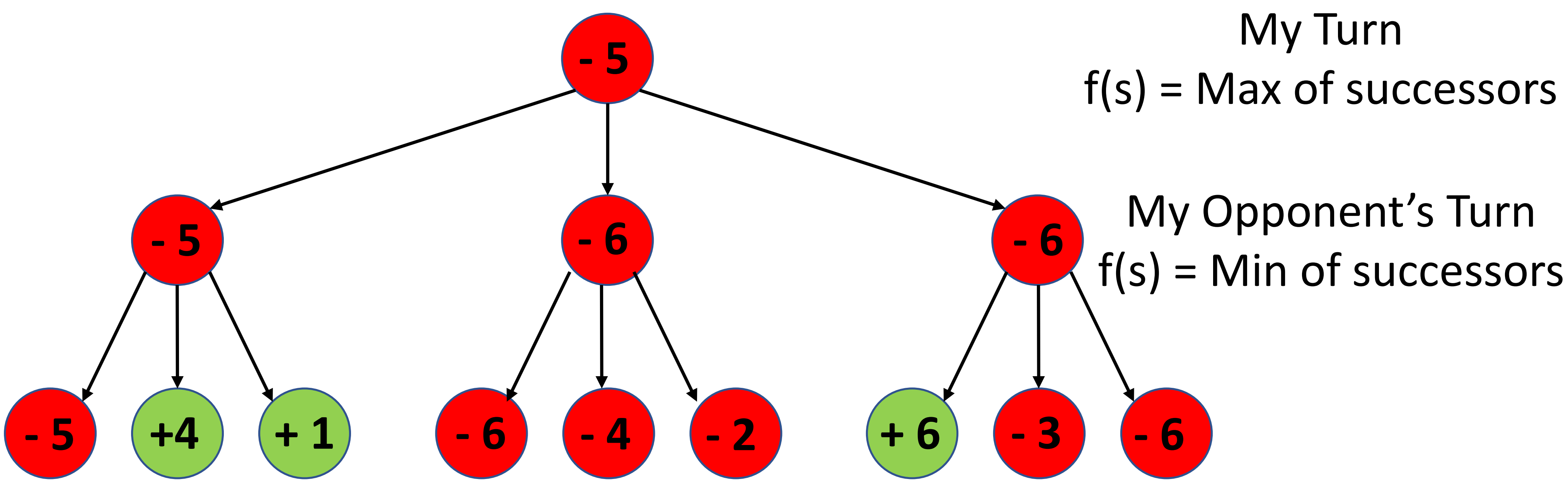


My Turn
f(s) = Max of successors

My Opponent's Turn
f(s) = Min of successors

Use heuristic evaluation function

Minimax Value of a Game



Use heuristic evaluation function

Minimax Algorithm (Heuristic Search)

```
maxturn(s,ops):
    if cutoff(s) then return f(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val' ← minturn(apply(s,o),ops);
            if val' > val then
                val ← val';
                bestop ← o;
        return val
```

```
minturn(s,ops):
    if cutoff(s) then return f(s)
    else
        val ← +∞;
        foreach o ∈ ops
            val' ← maxturn(apply(s,o),ops);
            if val' < val then
                val ← val';
                bestop ← o;
        return val
```

Initial call:

- If I go first: maxturn(initial-state,ops)
- If opponent goes first: minturn(initial-state,ops)

Alpha-Beta Pruning

New idea to improve efficiency:

Can prune branches that are guaranteed never to be used

Analogy

Given AND(p,q,r)

If p is FALSE don’t need to evaluate q or r

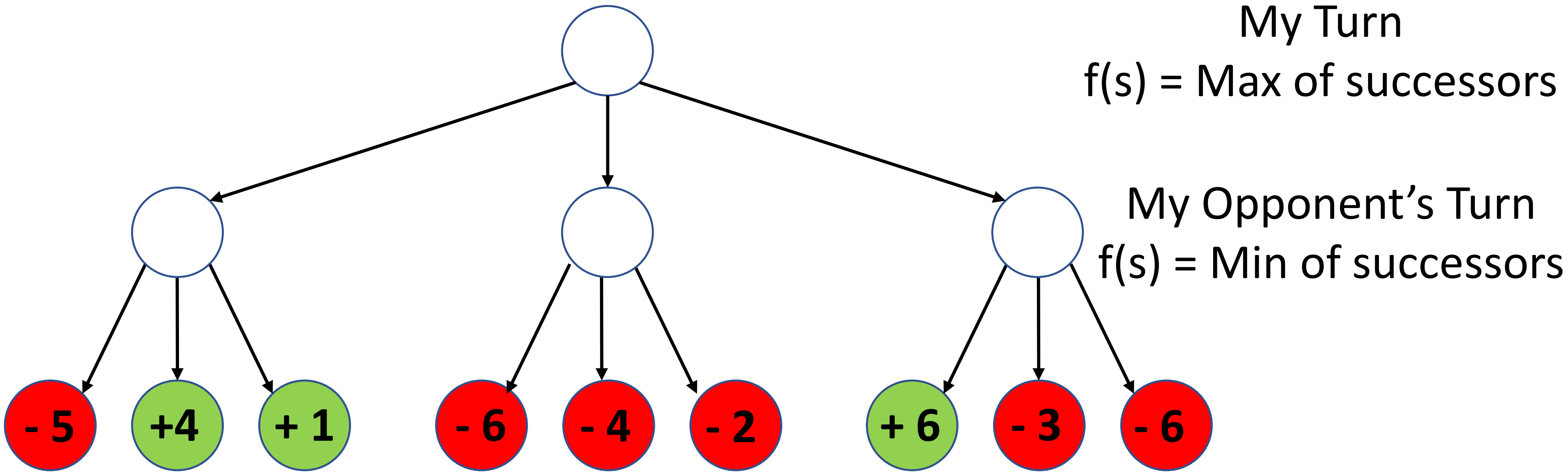
Given OR(p,q,r)

If q is TRUE don’t need to evaluate q or r

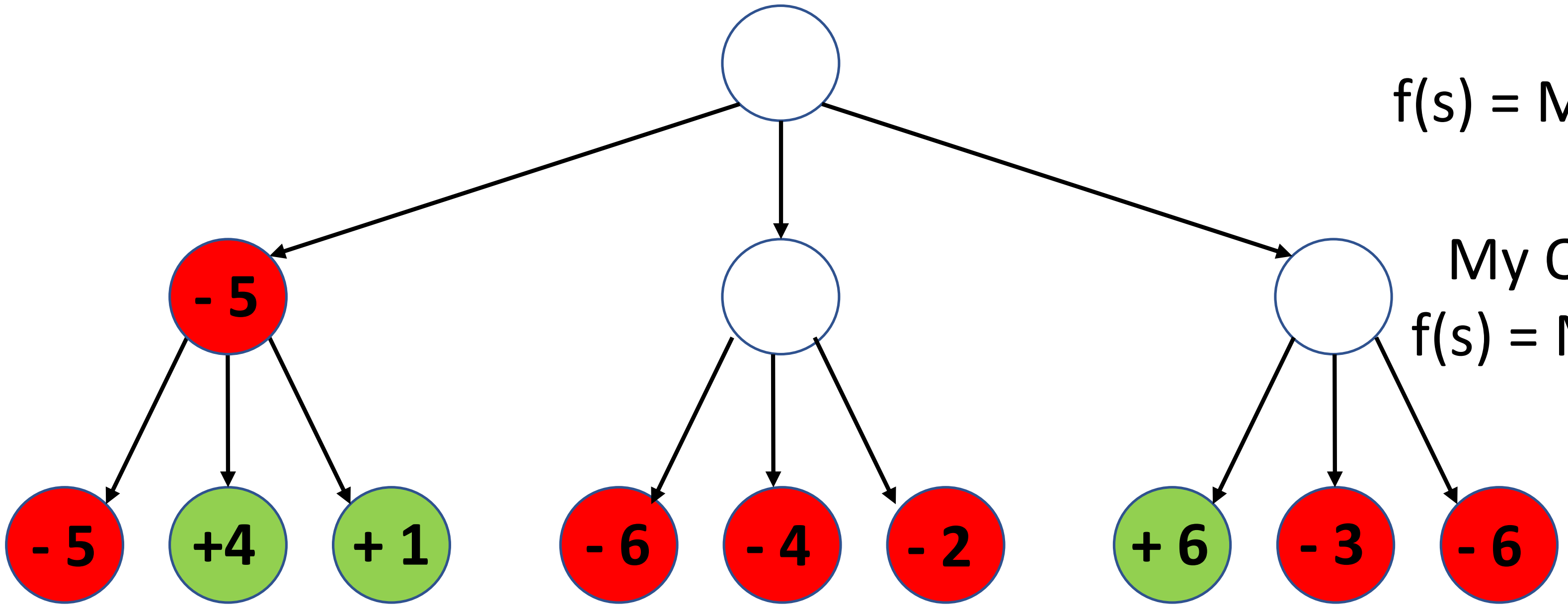
Alpha-Beta Pruning

Don’t explore branches that will have no effect on the outcome

Alpha-Beta Pruning



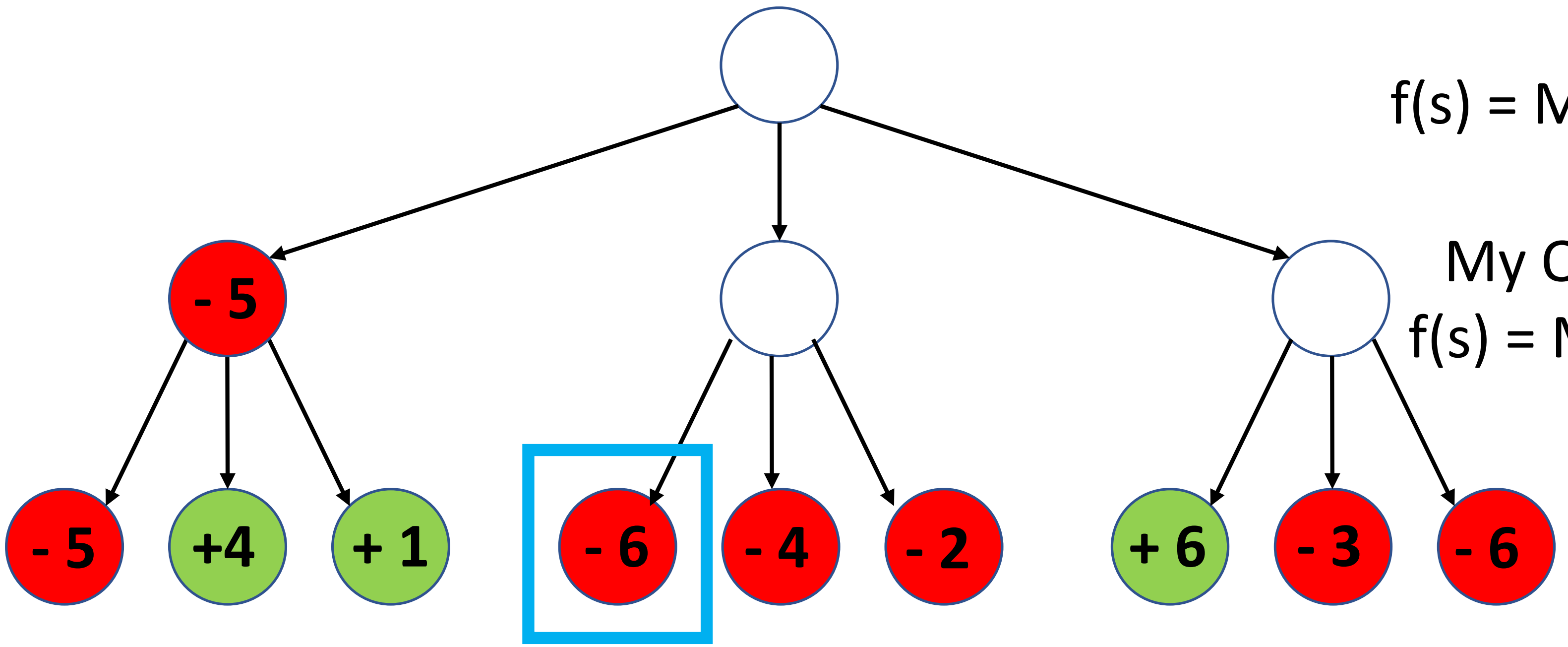
Alpha-Beta Pruning



My Turn
f(s) = Max of successors

My Opponent’s Turn
f(s) = Min of successors

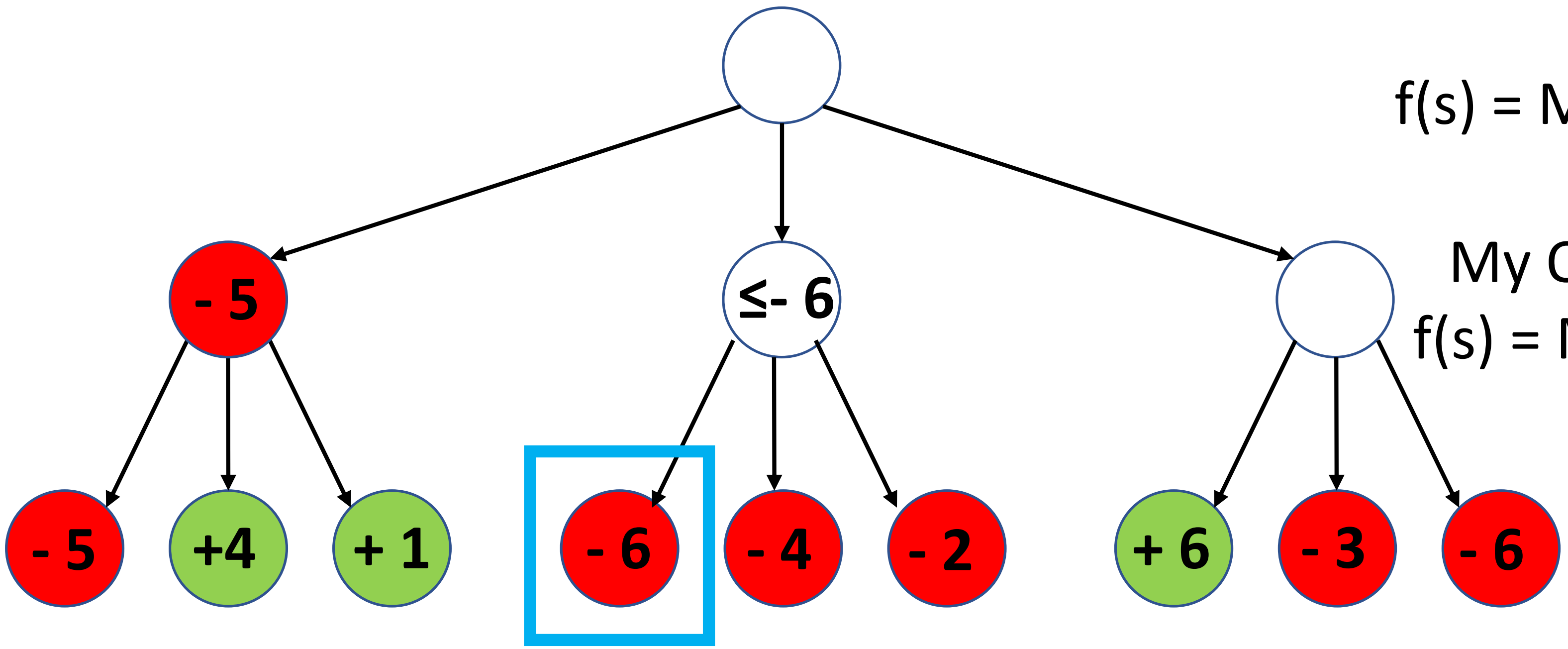
Alpha-Beta Pruning



My Turn
f(s) = Max of successors

My Opponent's Turn
f(s) = Min of successors

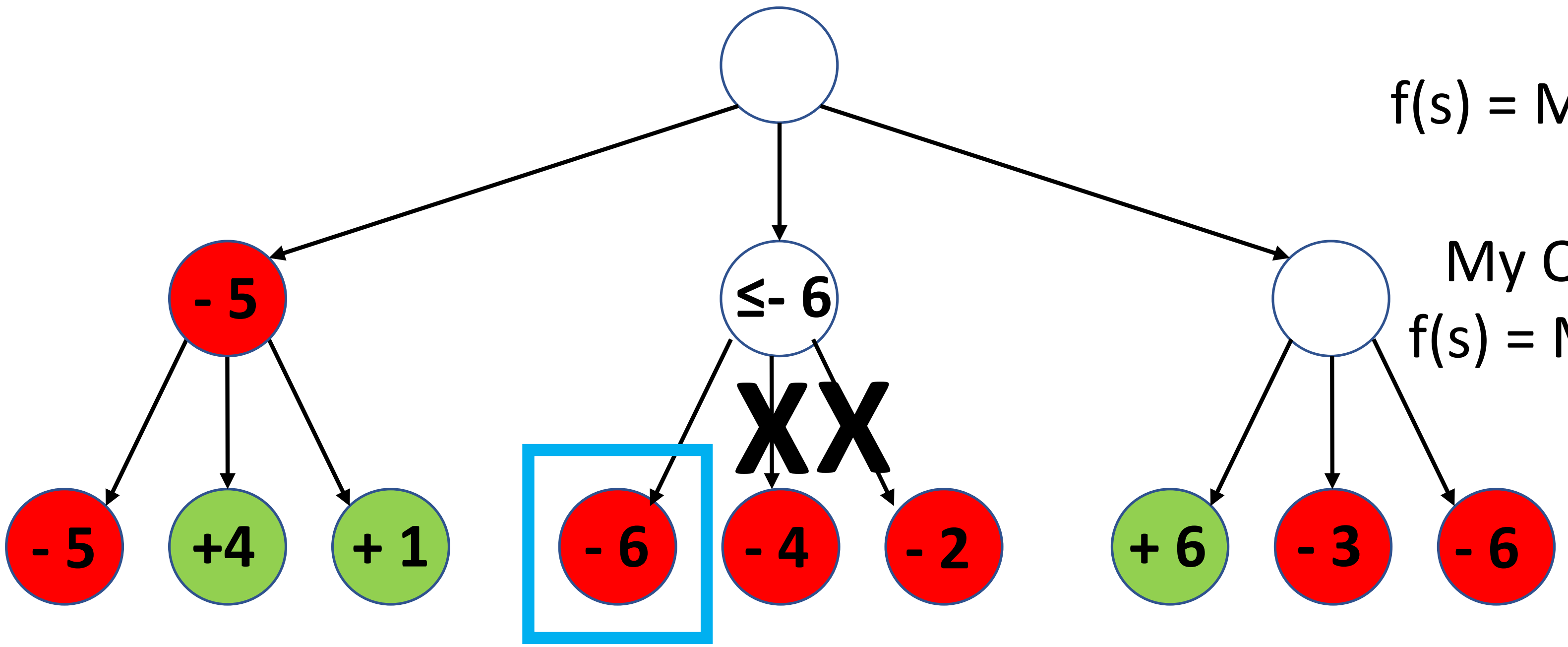
Alpha-Beta Pruning



My Turn
f(s) = Max of successors

My Opponent's Turn
f(s) = Min of successors

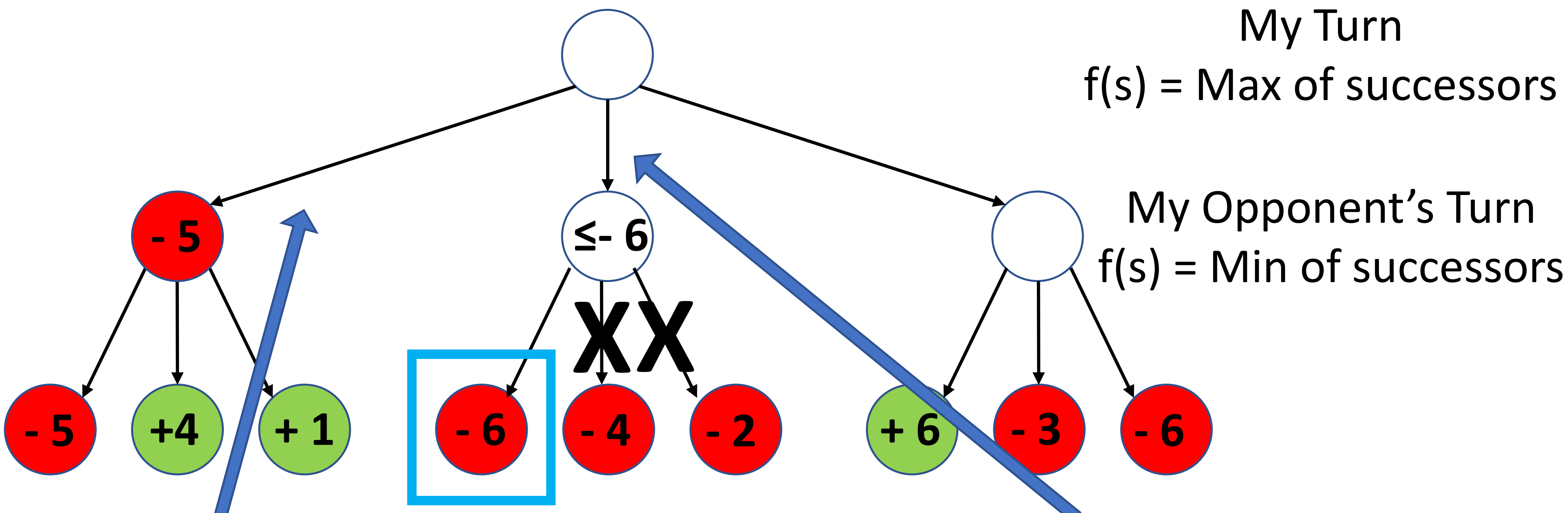
Alpha-Beta Pruning



My Turn
f(s) = Max of successors

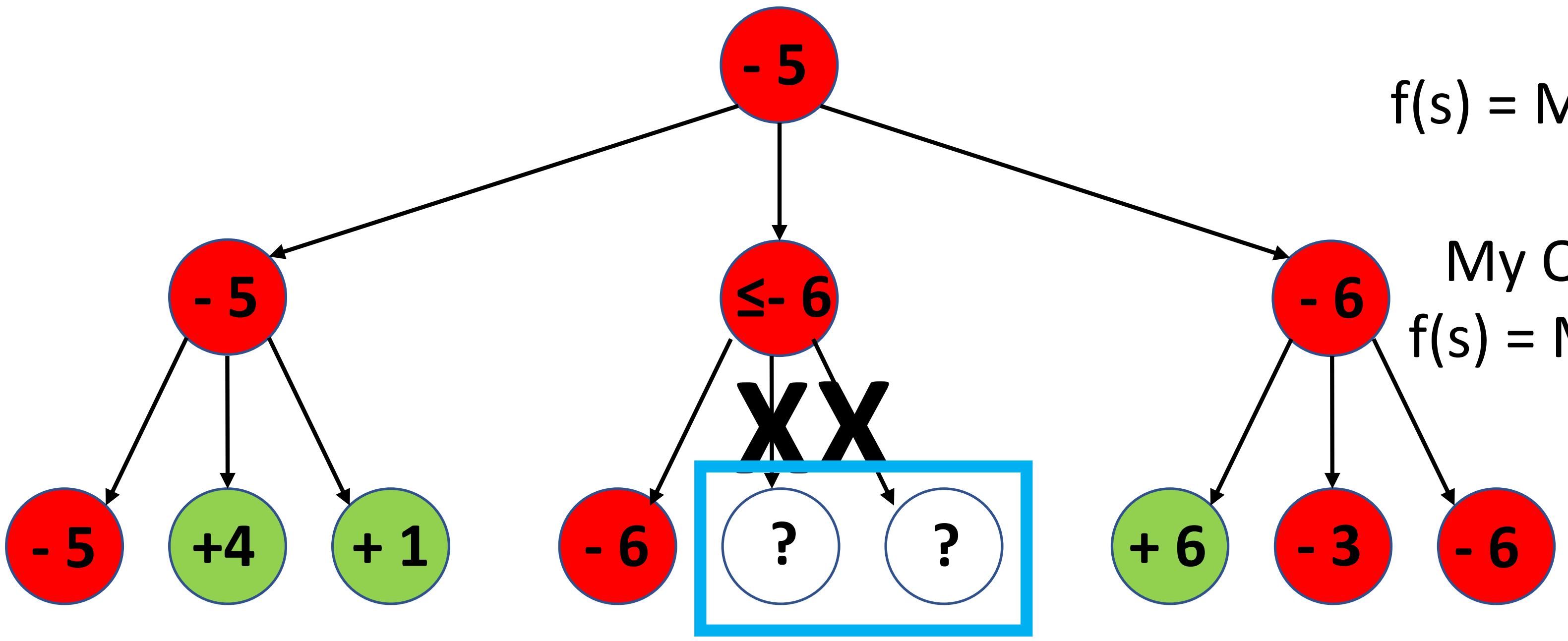
My Opponent's Turn
f(s) = Min of successors

Alpha-Beta Pruning



This is guaranteed to be a better move than this regardless of what results from the other actions

Alpha-Beta Pruning

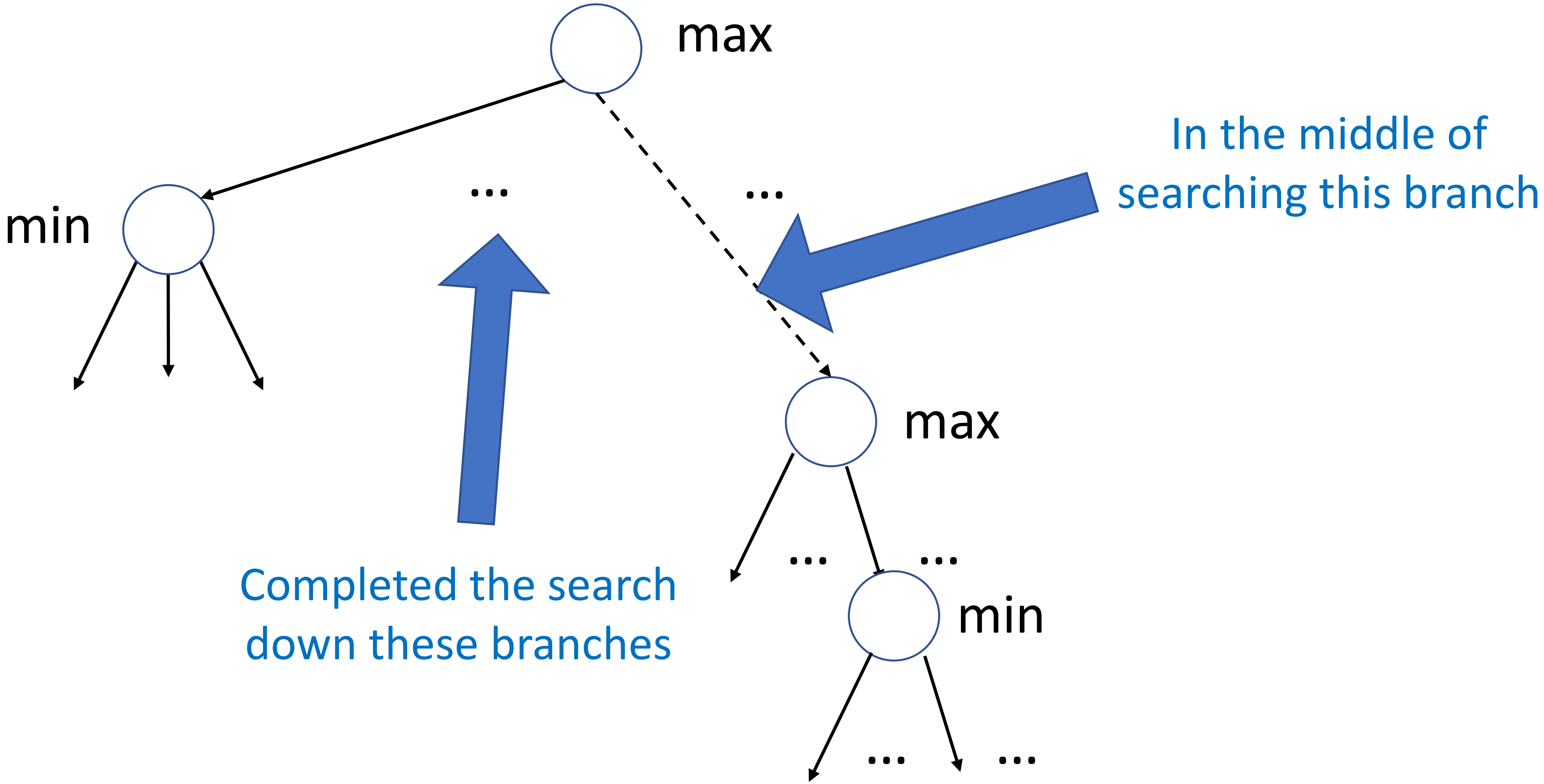


My Turn
f(s) = Max of successors

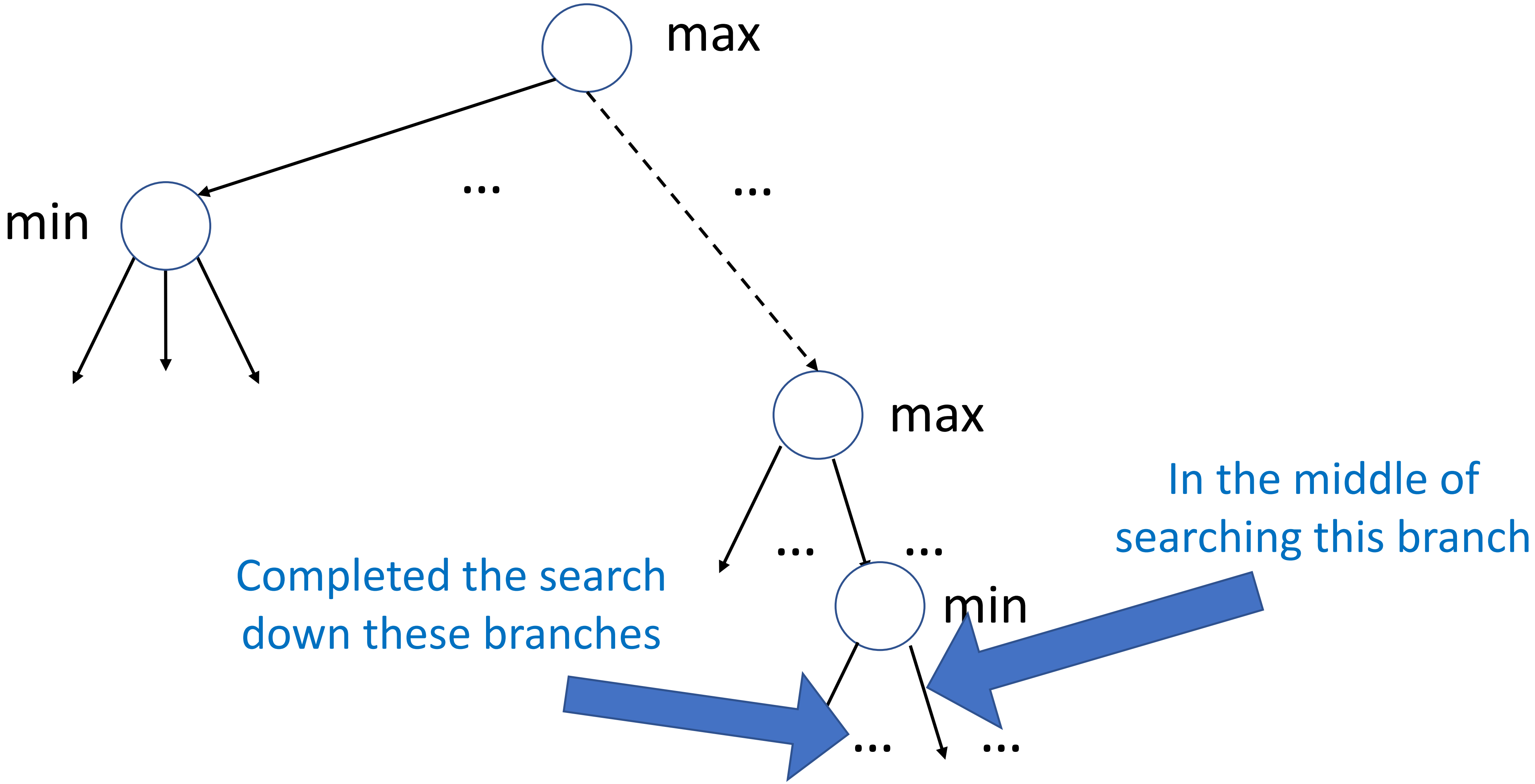
My Opponent's Turn
f(s) = Min of successors

There is no way to set a value to either of these two states that would change the outcome

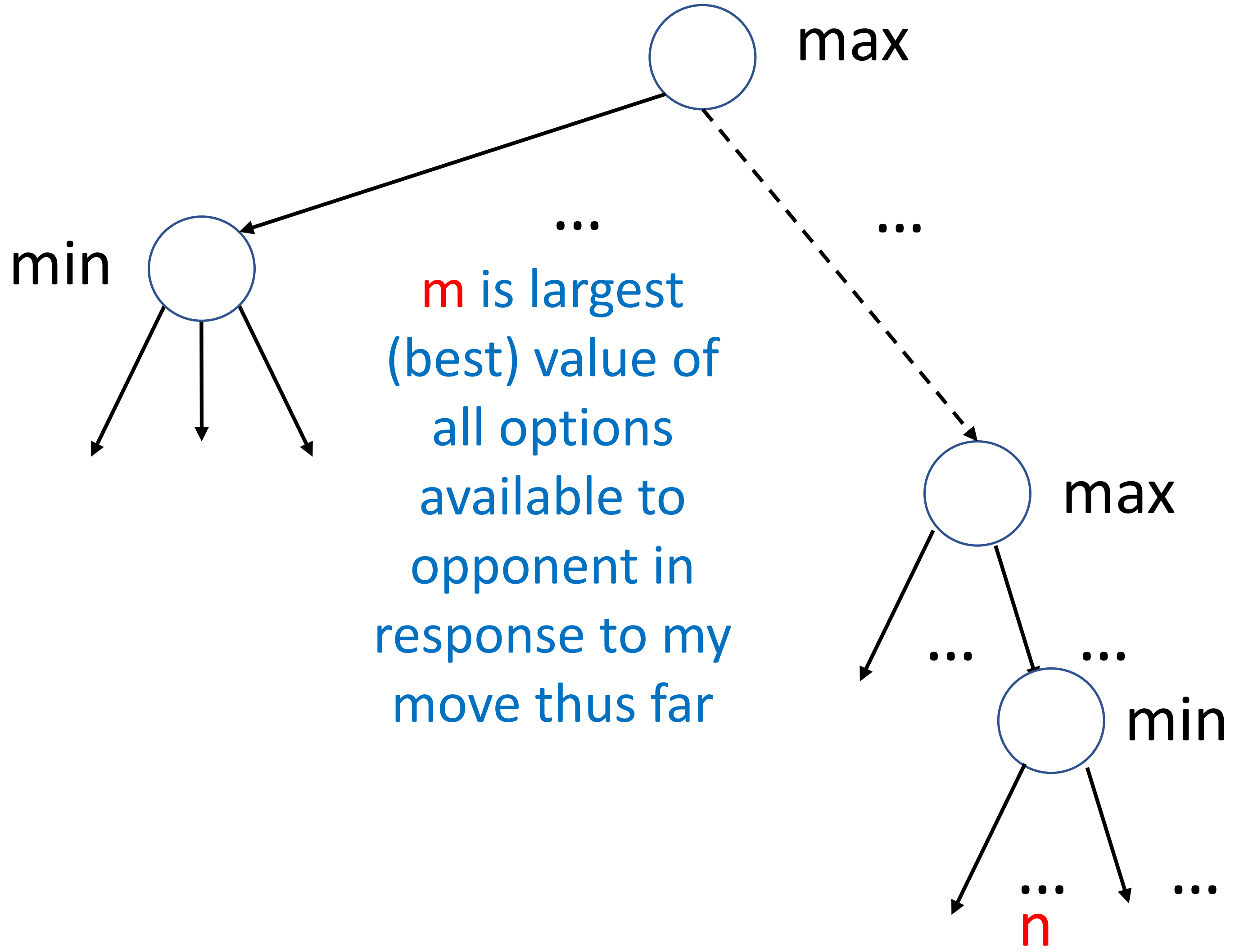
Alpha-Beta Pruning: General Case



Alpha-Beta Pruning: General Case

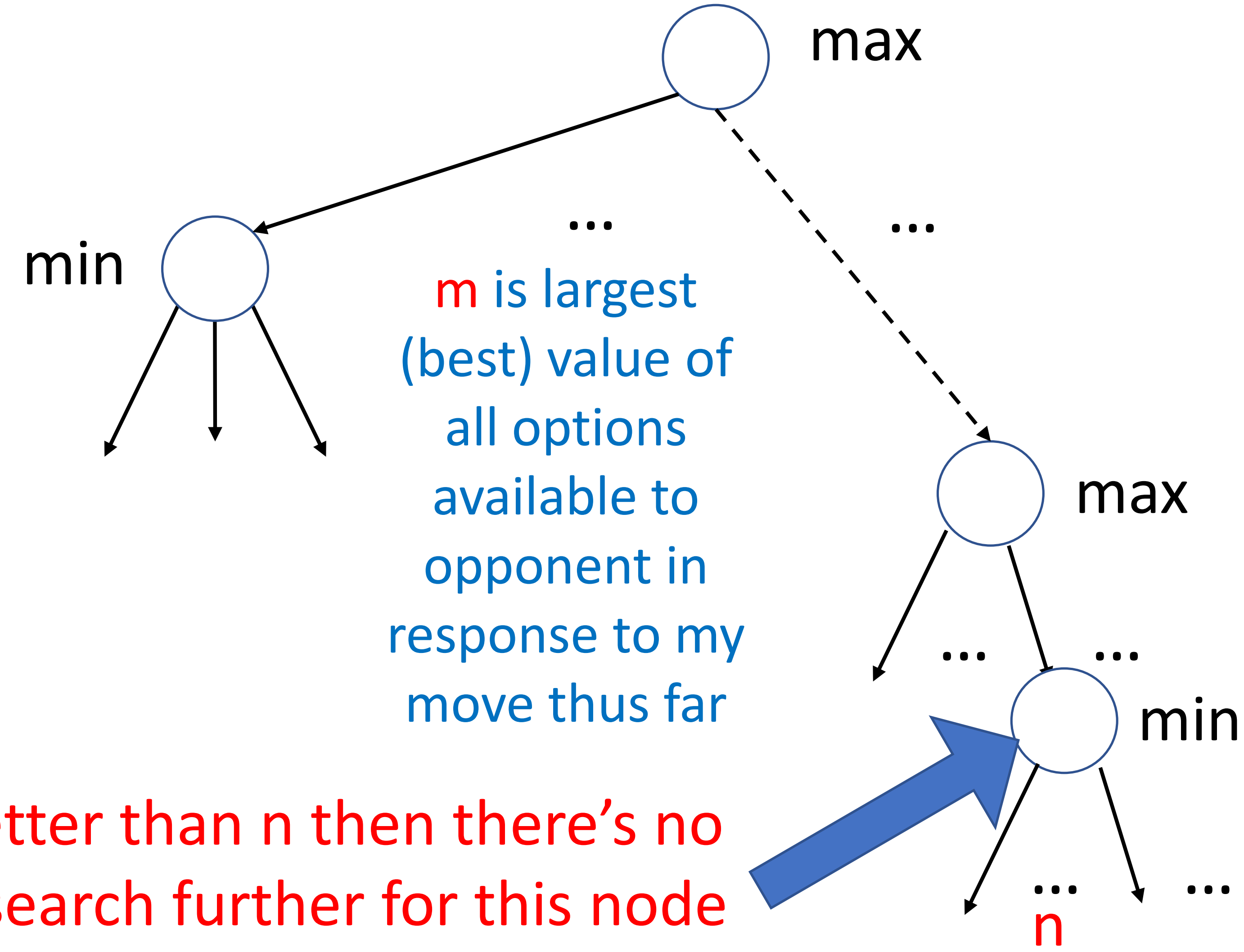


Alpha-Beta Pruning: General Case



n is lowest (worst) value of all options available to opponent in response to my move thus far

Alpha-Beta Pruning: General Case

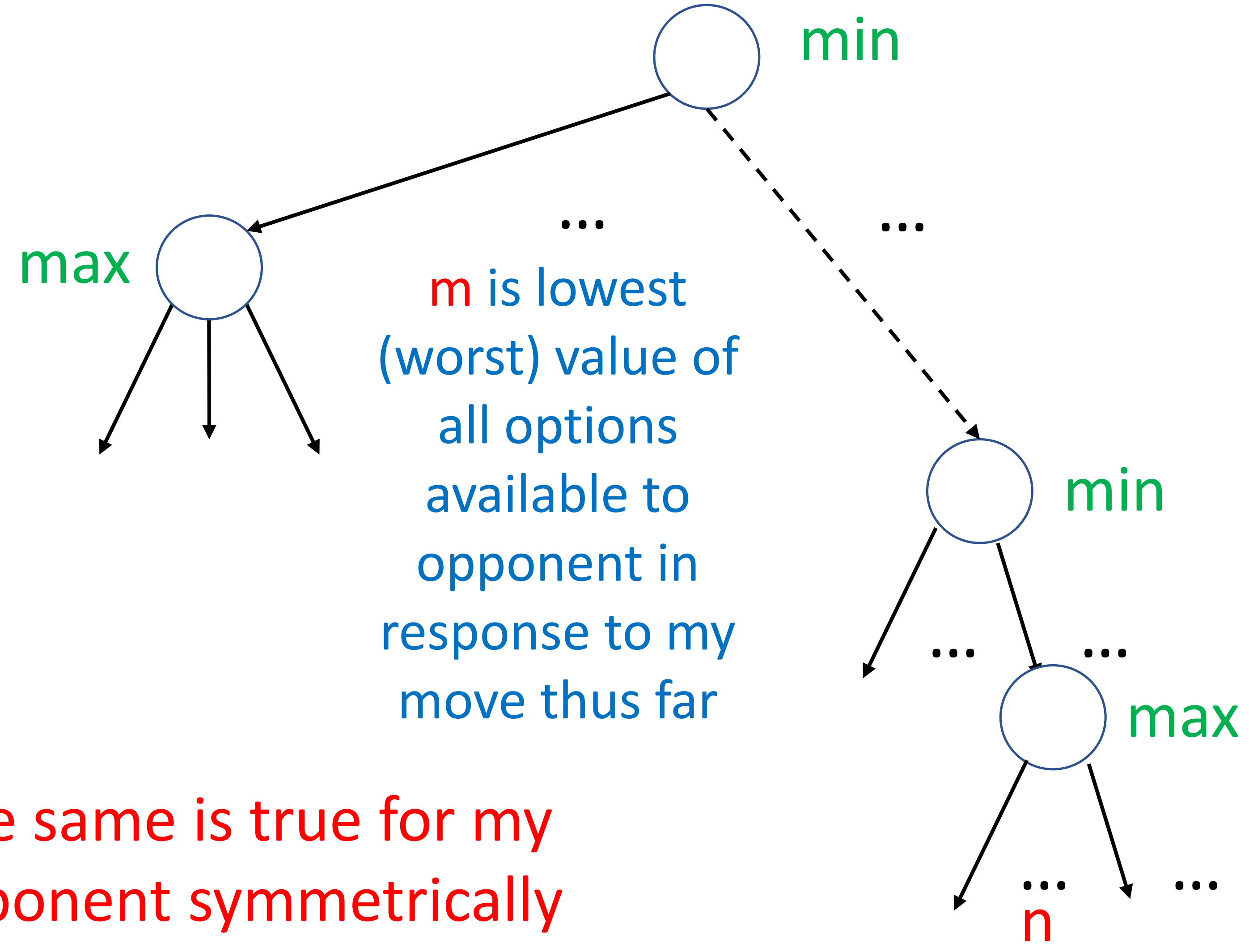


m is largest (best) value of all options available to opponent in response to my move thus far

n is lowest (worst) value of all options available to opponent in response to my move thus far

If m is better than n then there's no need to search further for this node

Alpha-Beta Pruning: General Case



m is lowest (worst) value of all options available to opponent in response to my move thus far

n is biggest (best) value of all options available to opponent in response to my move thus far

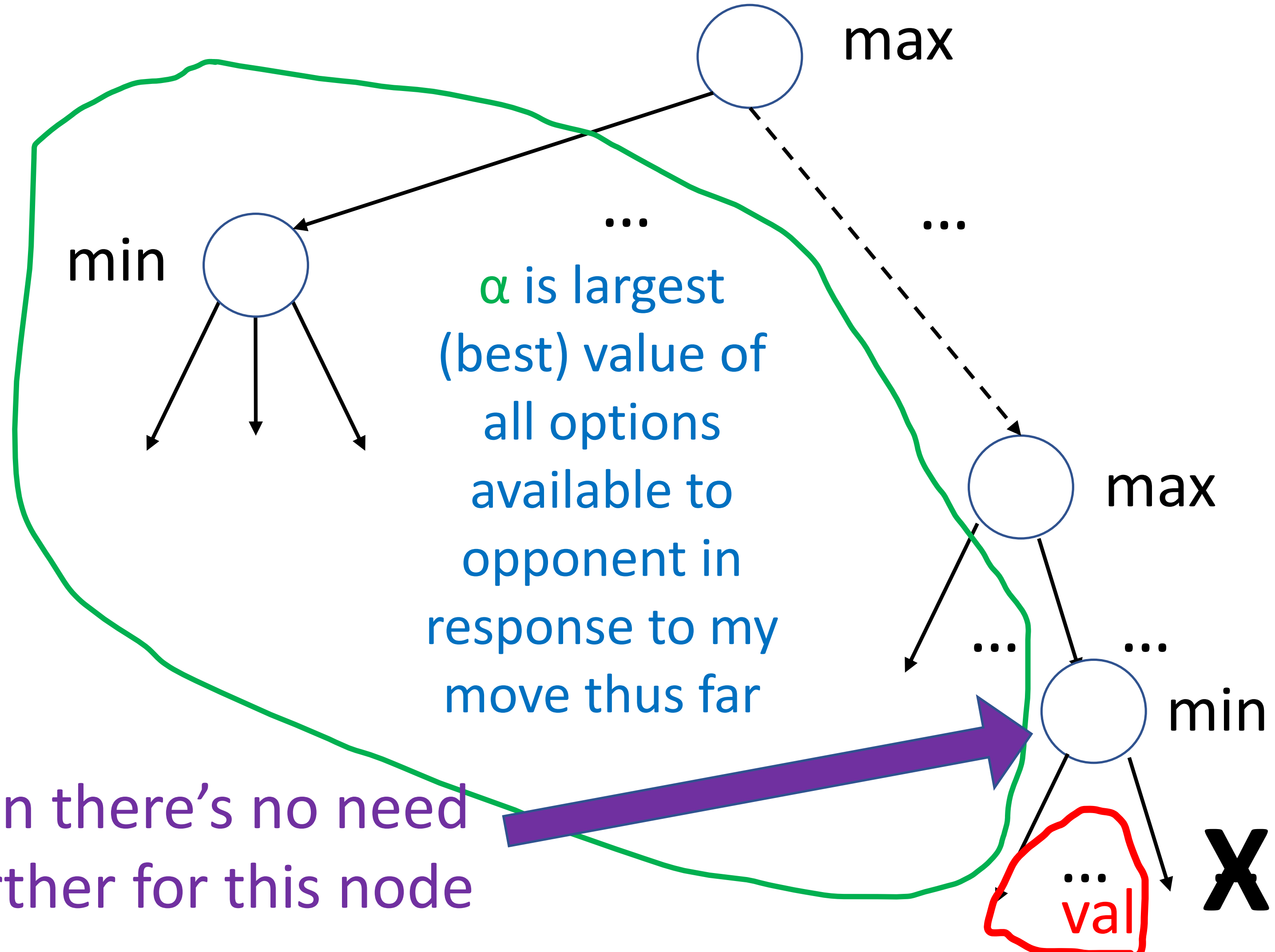
The same is true for my opponent symmetrically

Alpha-Beta Pruning: Implementation

- Key idea:
  - At each node keep track of two parameters, α and β (I'll use a and b in the algorithm)
  - α is the best value I can guarantee achieving via some earlier move
  - β is the worst value my opponent can guarantee achieving via some earlier move

  - Don't explore further nodes at this level if
    - You're at a min node and it's already guaranteed that this will be worse than some other move I can force (worst so far means $< \alpha$)
    - You're at a max node and it's already guaranteed that this will be better than some other move my opponent can force on me (best so far means $< \beta$)

Alpha-Beta Pruning: General Case

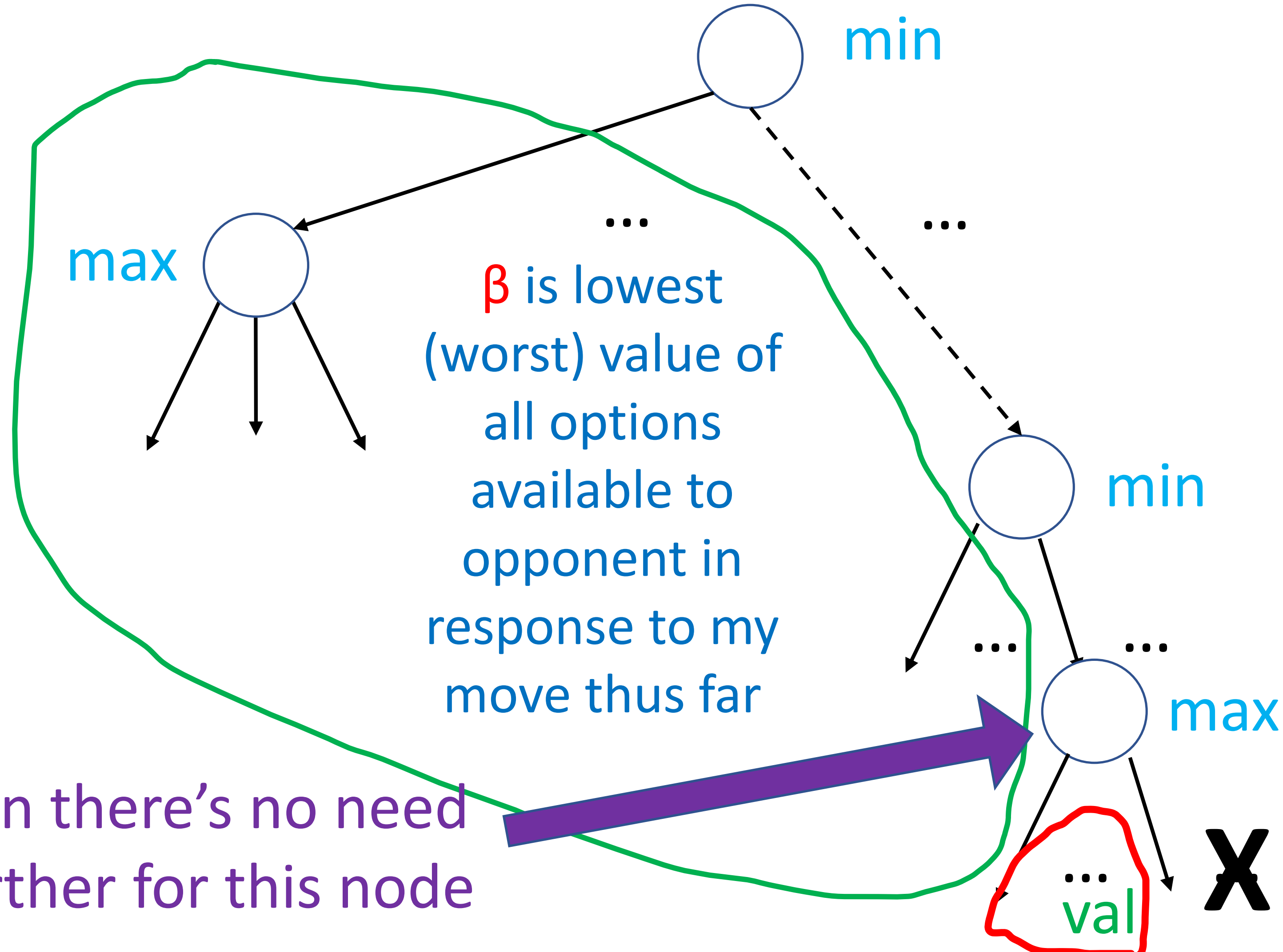


α is largest (best) value of all options available to opponent in response to my move thus far

val is lowest (worst) value of all options available to opponent in response to my move thus far

If $\alpha \geq$ val then there's no need to search further for this node

Alpha-Beta Pruning: General Case



val is biggest (best) value of all options available to opponent in response to my move thus far

If β ≤ val then there's no need to search further for this node

Minimax Algorithm

```
maxturn(s,ops):
    if cutoff(s) then return f(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val' ← minturn(apply(s,o),ops);
            if val' > val then
                val ← val';
                bestop ← o;
        return val
```

```
maxturn(s,ops):
    if cutoff(s) then return f(s)
    else
        val ← +∞;
        foreach o ∈ ops
            val' ← minturn(apply(s,o),ops);
            if val' < val then
                val ← val';
                bestop ← o;
        return val
```

Initial call:

- If I go first: maxturn(initial-state,ops,0)
- If opponent goes first: minturn(initial-state,ops,0)

Minimax Algorithm with Alpha-Beta Pruning

```
maxturn(s,ops,a,b):     {my turn}

    if cutoff(s,depth) then return f(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val' ← minturn(apply(s,o),ops,a,b);
            if val' > val then
                val ← val';
                bestop ← o;
            if val ≥ b then return val;
            a ← max(a,val)
        return val
```

```
maxturn(s,ops,a,b):     {opponent's turn}

    if cutoff(s,depth) then return f(s)
    else
        val ← +∞;
        foreach o ∈ ops
            val' ← minturn(apply(s,o),ops,a,b);
            if val' < val then
                val ← val';
                bestop ← o;
            if val ≤ a then return val;
            b ← min(b,val)
        return val
```

Initial call:

- If I go first: maxturn(initial-state,ops,-∞,+∞)
- If opponent goes first: minturn(initial-state,ops,-∞,+∞)