# Adversarial Search

Robert Platt
Northeastern University

Some images and slides are used from:
1. CS188 UC Berkeley
2. RN, AIMA

# What is adversarial search?


Adversarial search: planning used to play a game such as chess or checkers

– algorithms are similar to graph search except that we plan under the assumption that our opponent will maximize his own advantage...

# Examples of adversarial search

Chess

Checkers

Tic-tac-toe

Go

# Examples of adversarial search

Chess Solved/unsolved?

Checkers Solved/unsolved?

Tic-tac-toe Solved/unsolved?

Go Solved/unsolved?

Outcome of game can be predicted from any initial state assuming both players play perfectly

# Examples of adversarial search

| | |
|---|---|
| Chess | Unsolved |
| Checkers | Solved |
| Tic-tac-toe | Solved |
| Go | Unsolved |

Outcome of game can be predicted from any initial state assuming both players play perfectly

# Examples of adversarial search

| | | |
|---|---|---|
| Chess | Unsolved | ~10^40 states |
| Checkers | Solved | ~10^20 states |
| Tic-tac-toe | Solved | Less than 9!=362k states |
| Go | Unsolved | ? |

Outcome of game can be predicted from any initial state assuming both players play perfectly

# Different types of games

Deterministic / stochastic

Two player / multi player?

Zero-sum / non zero-sum

Fully observable / partially observable

# What is a zero-sum game?

Zero-sum:

- Sum of utilities is zero
- In the case of a two player game: $U_A = -U_B$
- Pure competition

Not zero-sum:

- Agents have arbitrary utilities
- Might induce cooperation or competition

# A formal definition of a deterministic game

Problem:

State set: S (start at s0)

Players: P={1...N} (usually take turns)

Action set: A

Transition Function: SxA -> S

Terminal Test: S -> {t,f}

Terminal Utilities: SxP -> R

Solution:

Policy, S -> A

Objective:

Find an optimal policy

– a policy that maximizes utility assuming that adversary acts optimally.

# A formal definition of a deterministic game

Problem:

State set: S (start at s0)

Players: P={1...N} (usually take turns)

Action set: A

Transition Function: SxA -> S

Terminal Test: S -> {t,f}

Terminal Utilities: SxP -> R

How is this similar/different to the def'n of a standard search problem?

Solution:

Policy, S -> A

Objective:

Find an optimal policy

– a policy that maximizes utility assuming that adversary acts optimally.

# A formal definition of a deterministic game

Problem:

State set: S (start at s0)

Players: P={1...N} (usually take turns)

Action set: A

Transition Function: SxA -> S

Terminal Test: S -> {t,f}

Terminal Utilities: SxP -> R

How do we solve this problem?

Solution:

Policy, S -> A

Objective:

Find an optimal policy

– a policy that maximizes utility assuming that adversary acts optimally.

# Adversarial search

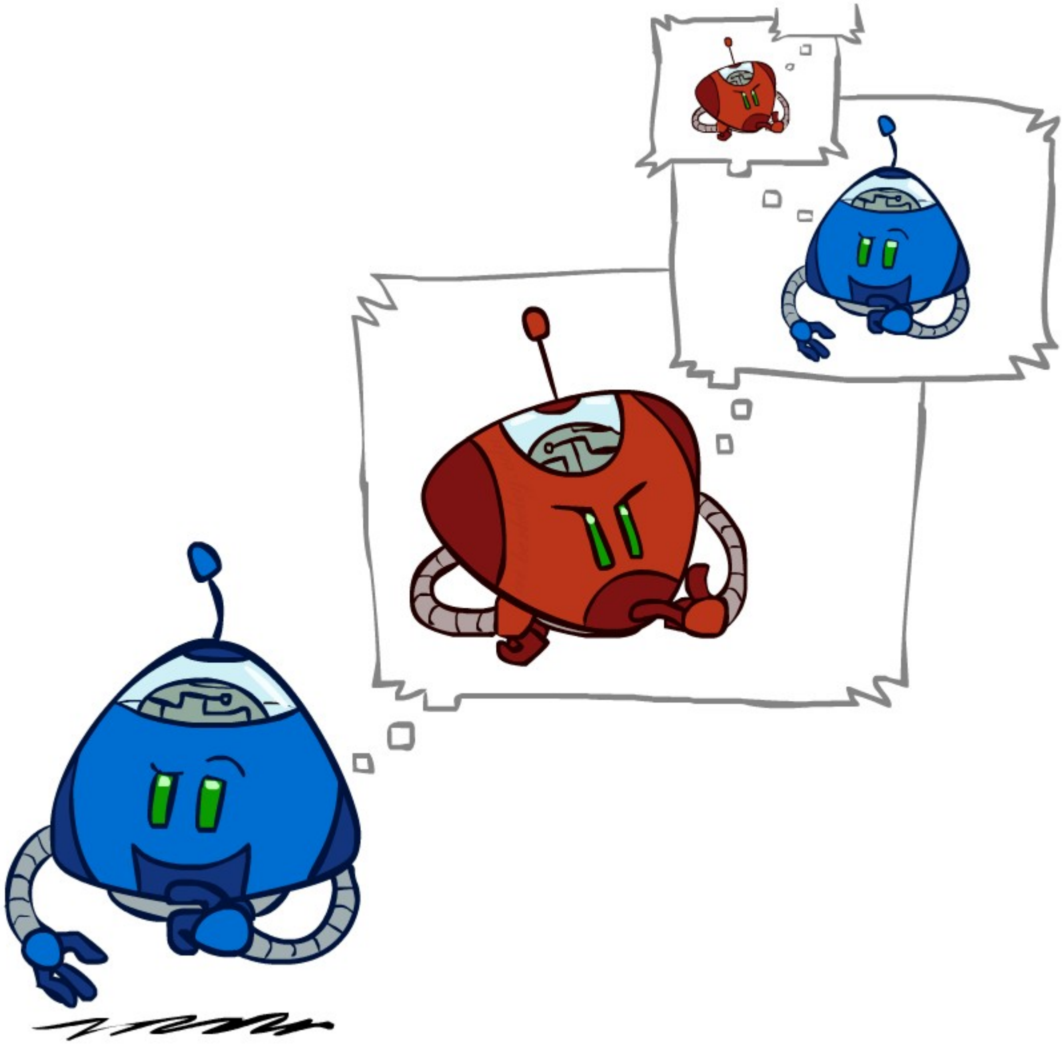
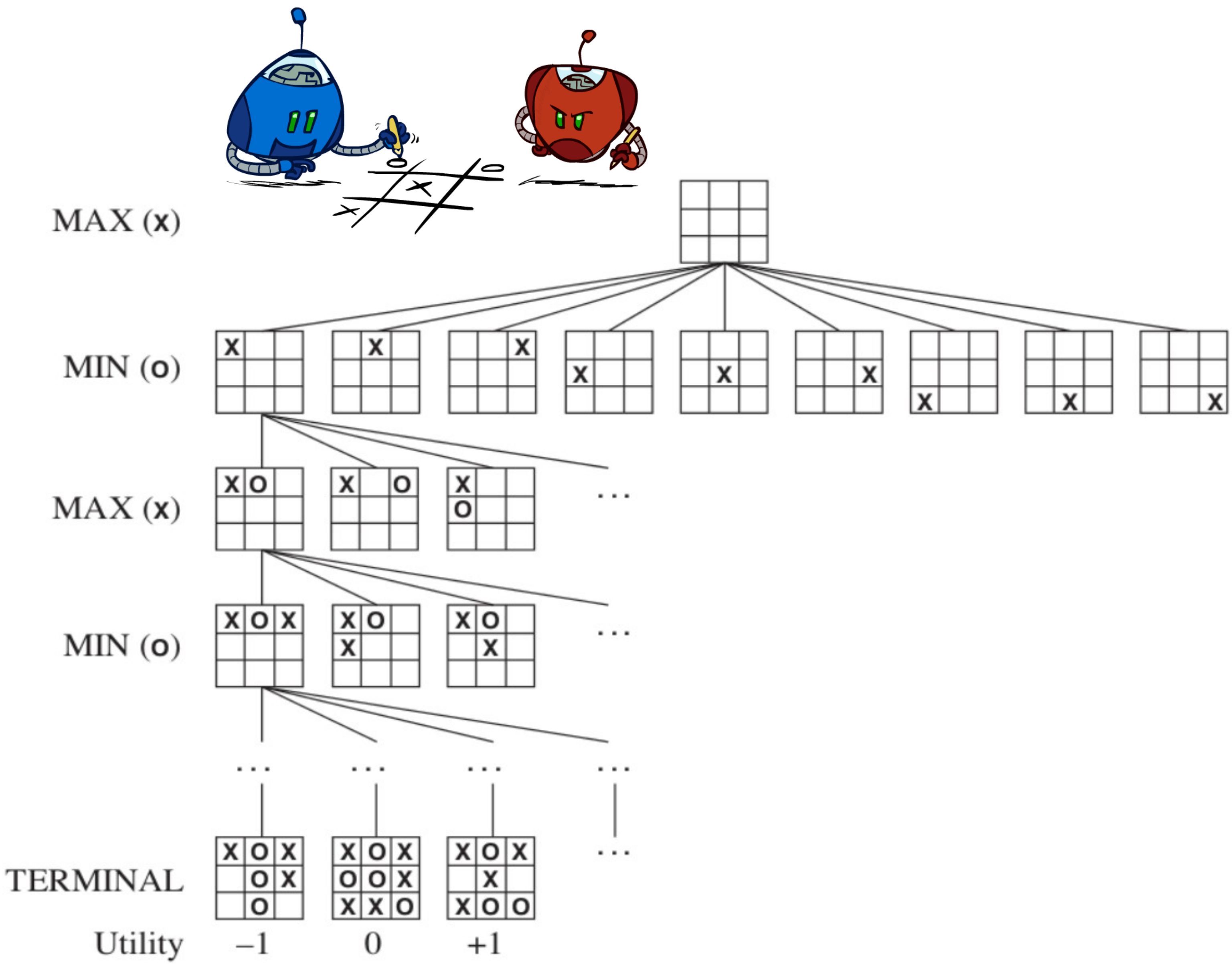


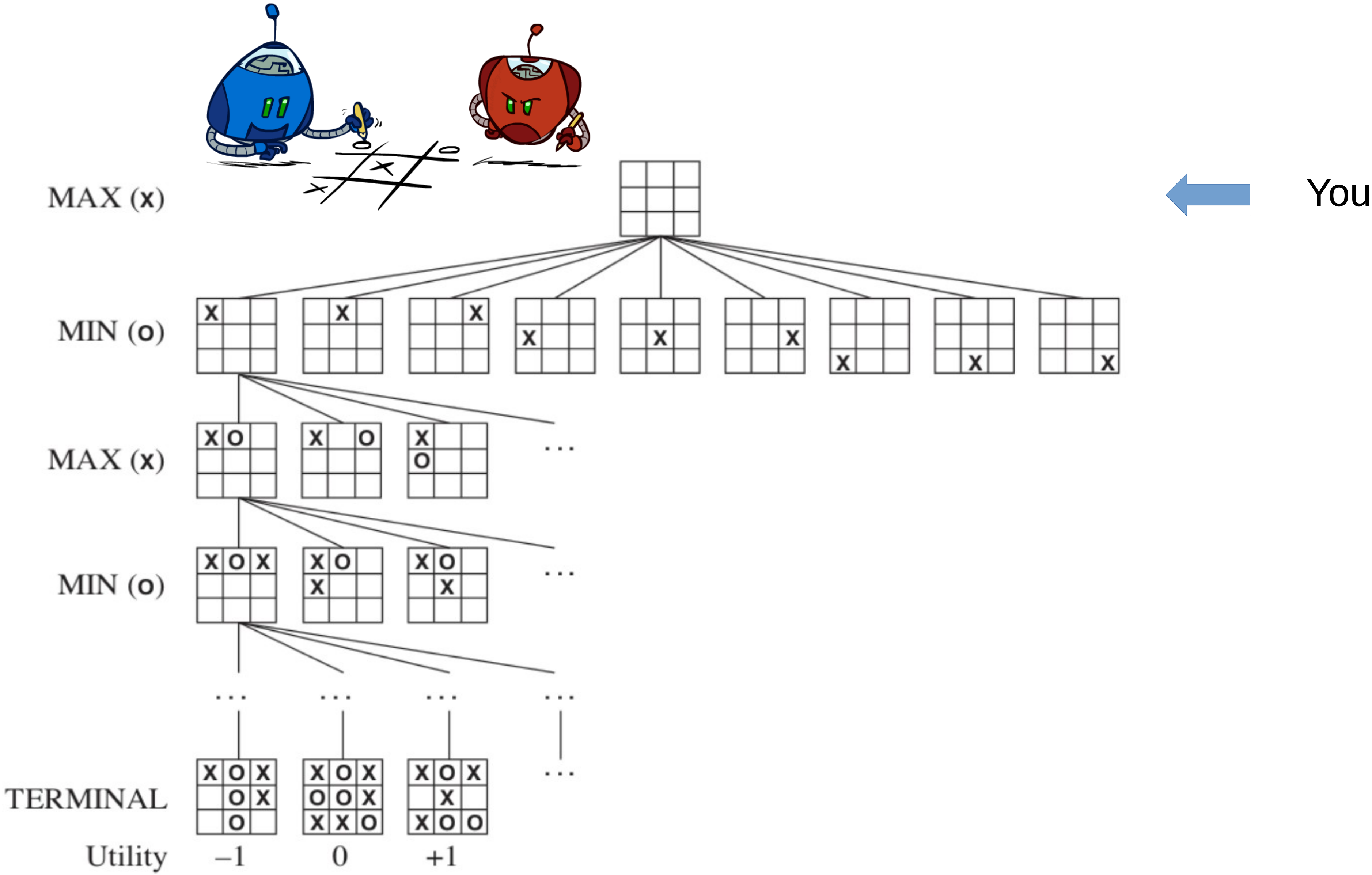
Image: Berkeley CS188 course notes (downloaded Summer 2015)

# This is a game tree for tic-tac-toe



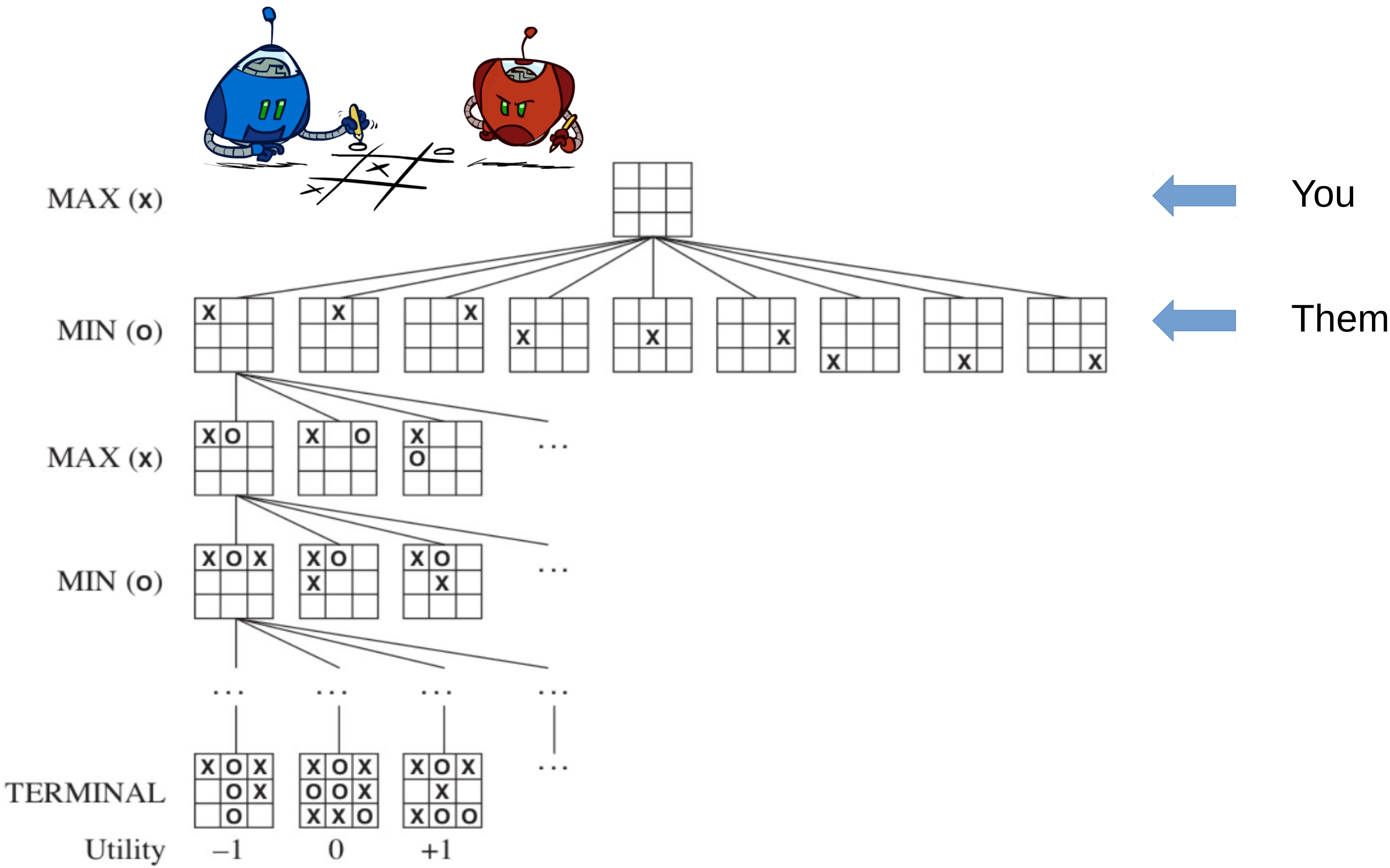

Images: AIMA, Berkeley CS188 course notes (downloaded Summer 2015)

# This is a game tree for tic-tac-toe



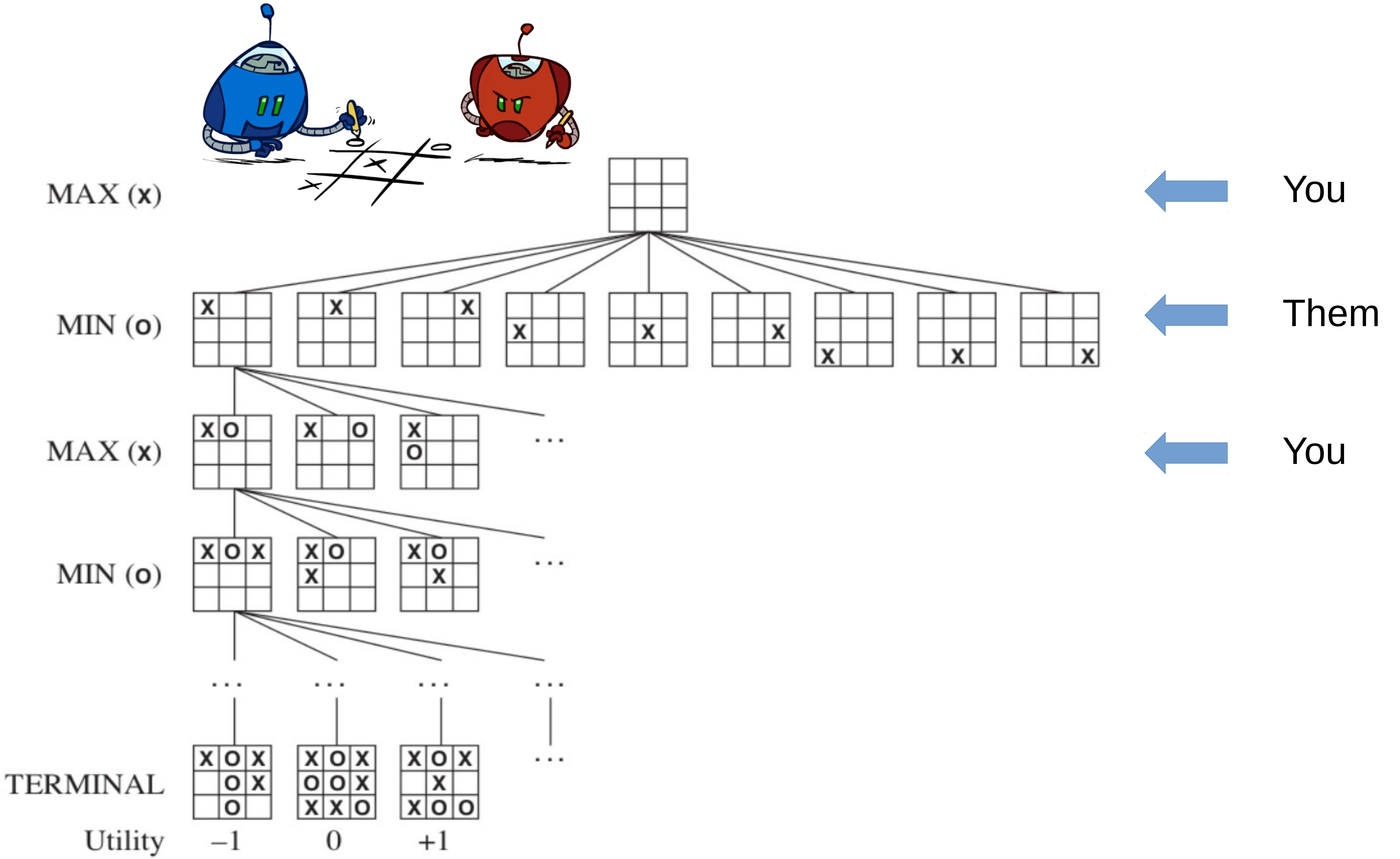

Images: AIMA, Berkeley CS188 course notes (downloaded Summer 2015)

# This is a game tree for tic-tac-toe



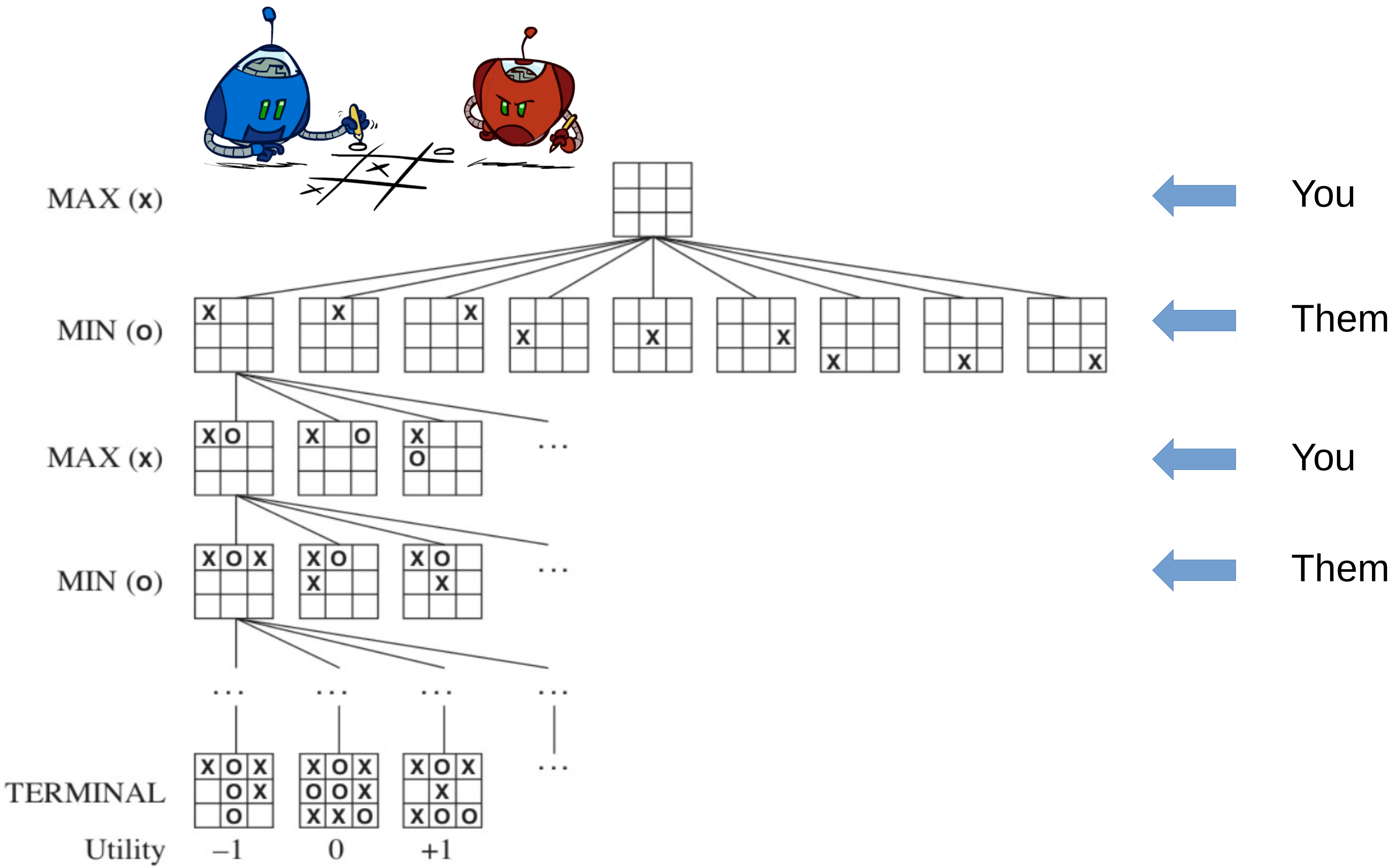

Images: AIMA, Berkeley CS188 course notes (downloaded Summer 2015)

# This is a game tree for tic-tac-toe



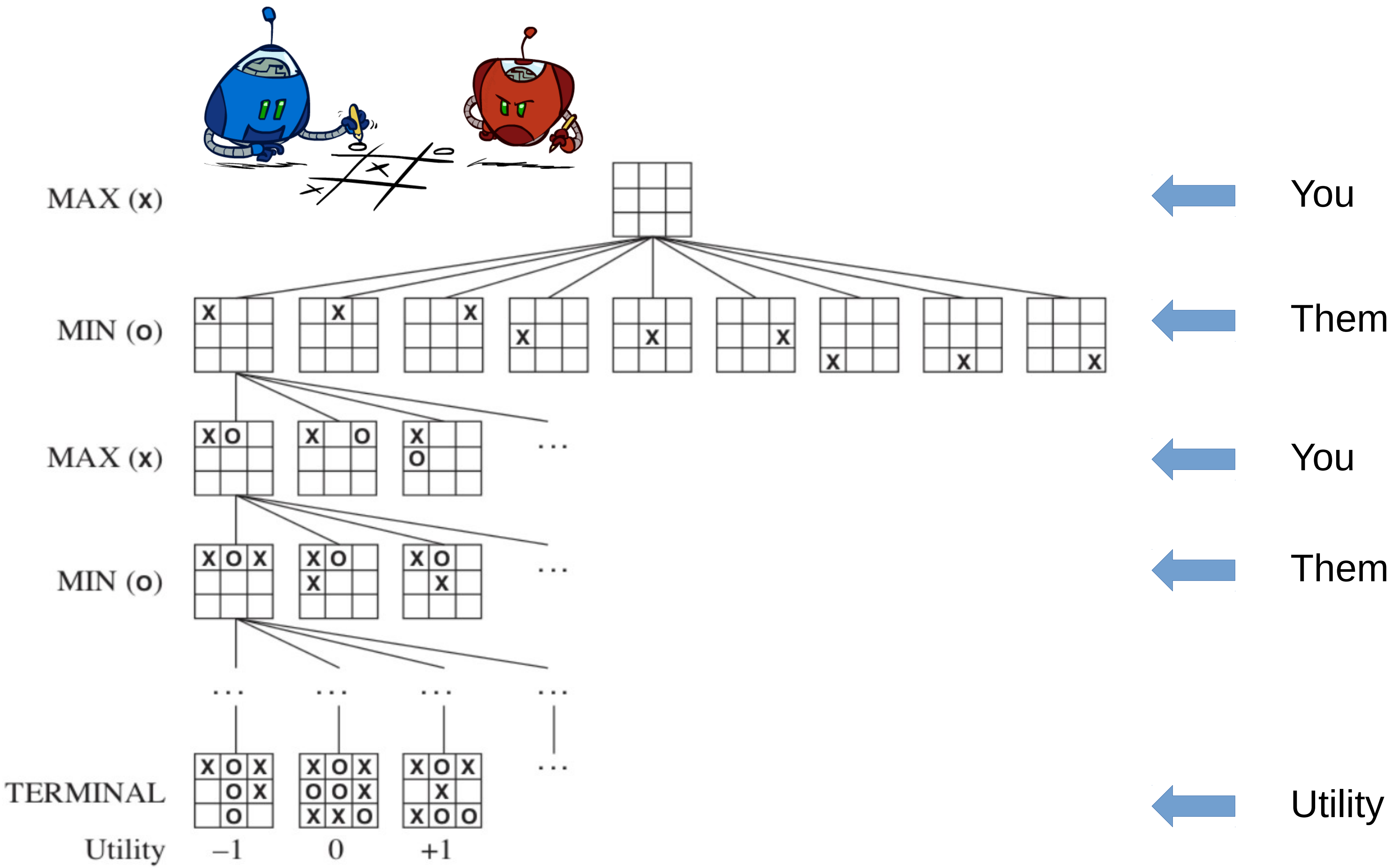

Images: AIMA, Berkeley CS188 course notes (downloaded Summer 2015)

# This is a game tree for tic-tac-toe




Images: AIMA, Berkeley CS188 course notes (downloaded Summer 2015)

# This is a game tree for tic-tac-toe




Images: AIMA, Berkeley CS188 course notes (downloaded Summer 2015)

# What is Minimax?

Consider a simple game:

1. you make a move
2. your opponent makes a move
3. game ends

# What is Minimax?

Consider a simple game:

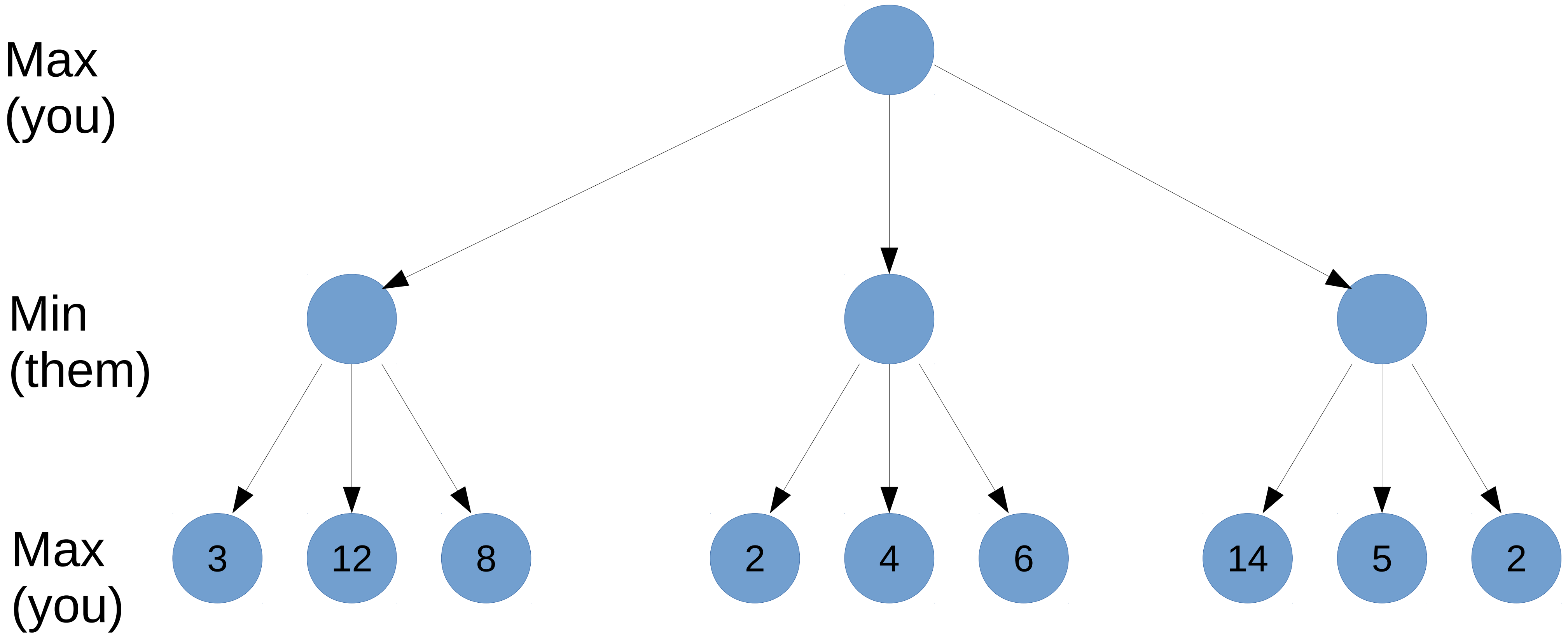
1. you make a move
2. your opponent makes a move
3. game ends

What does the minimax tree look like in this case?

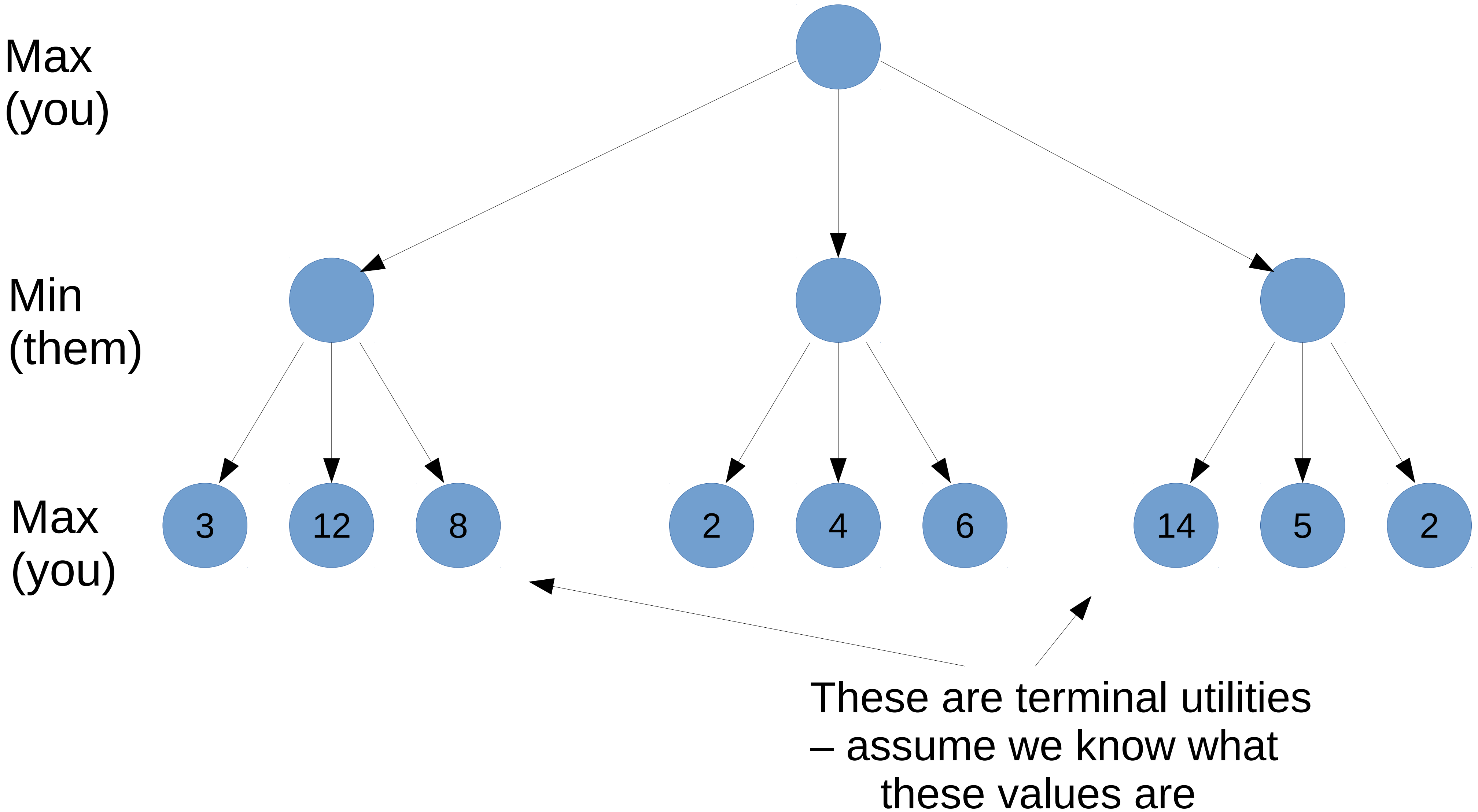
# What is Minimax?

Consider a simple game:
1. you make a move
2. your opponent makes a move
3. game ends

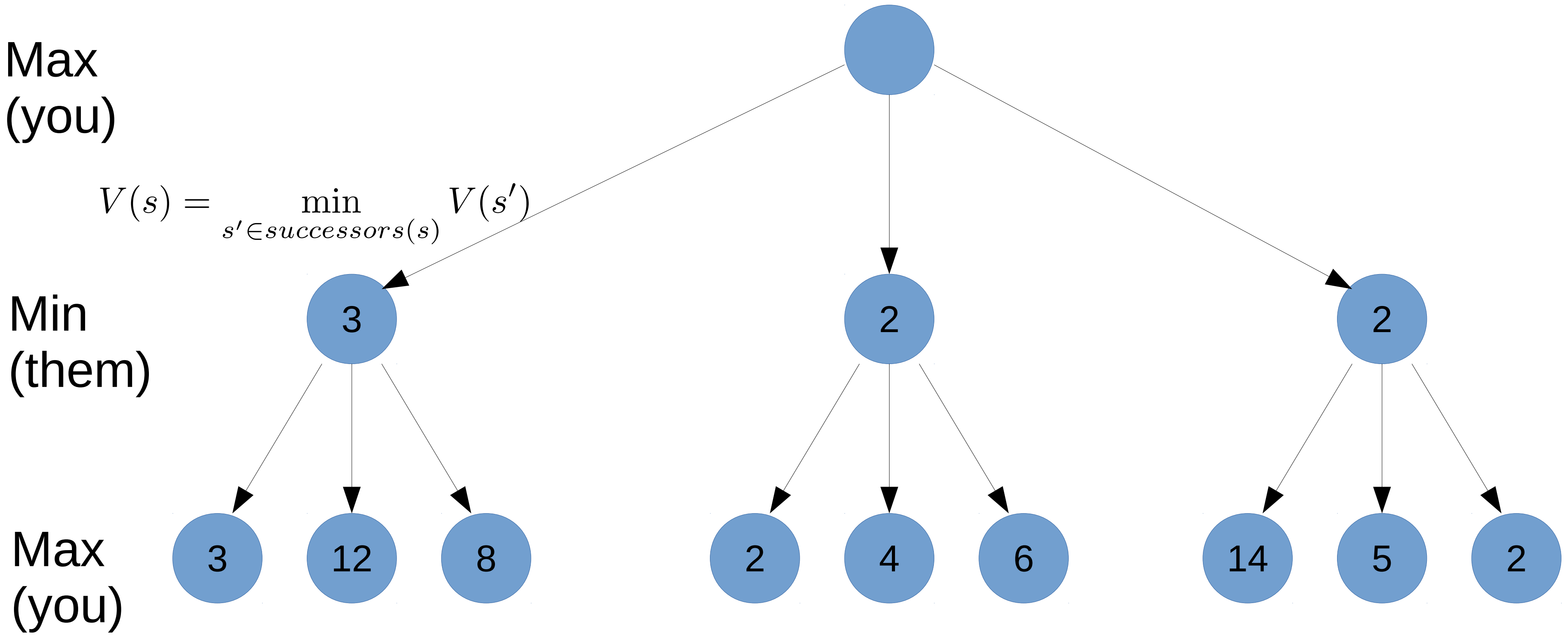
What does the minimax tree look like in this case?



# What is Minimax?

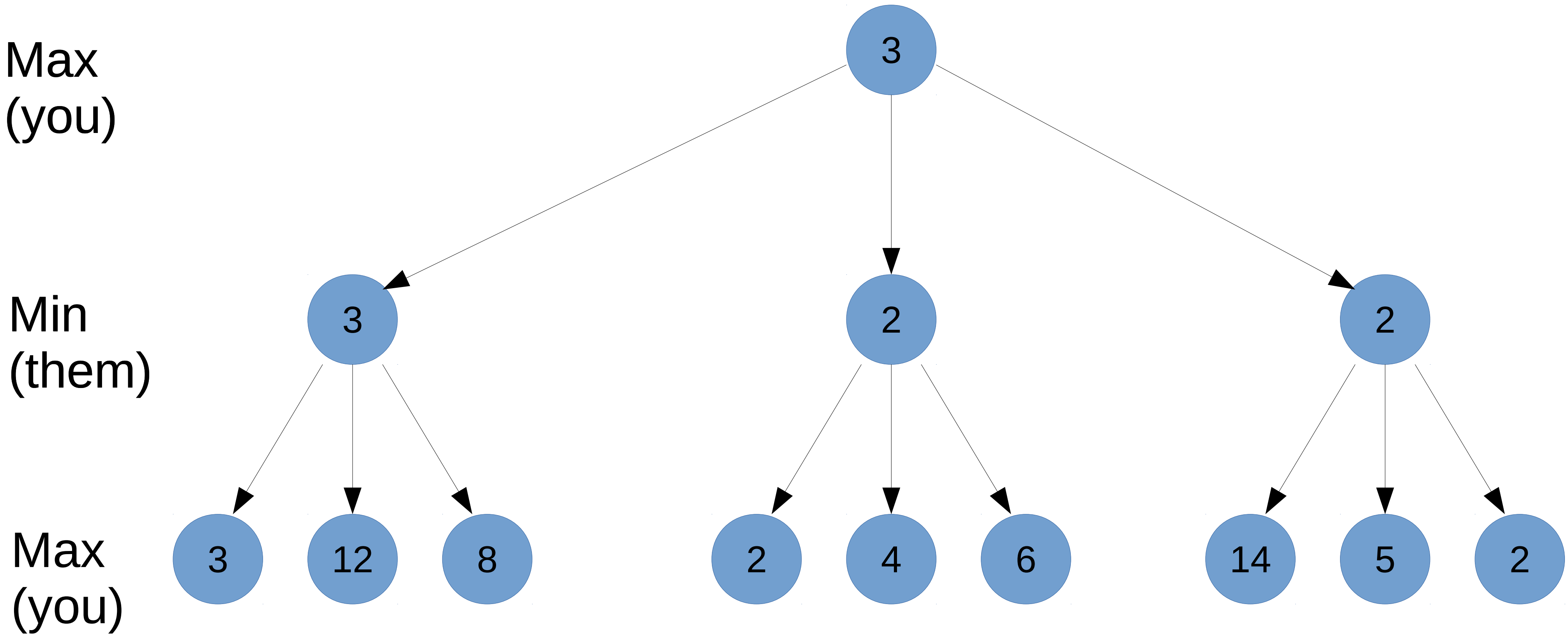


# What is Minimax?



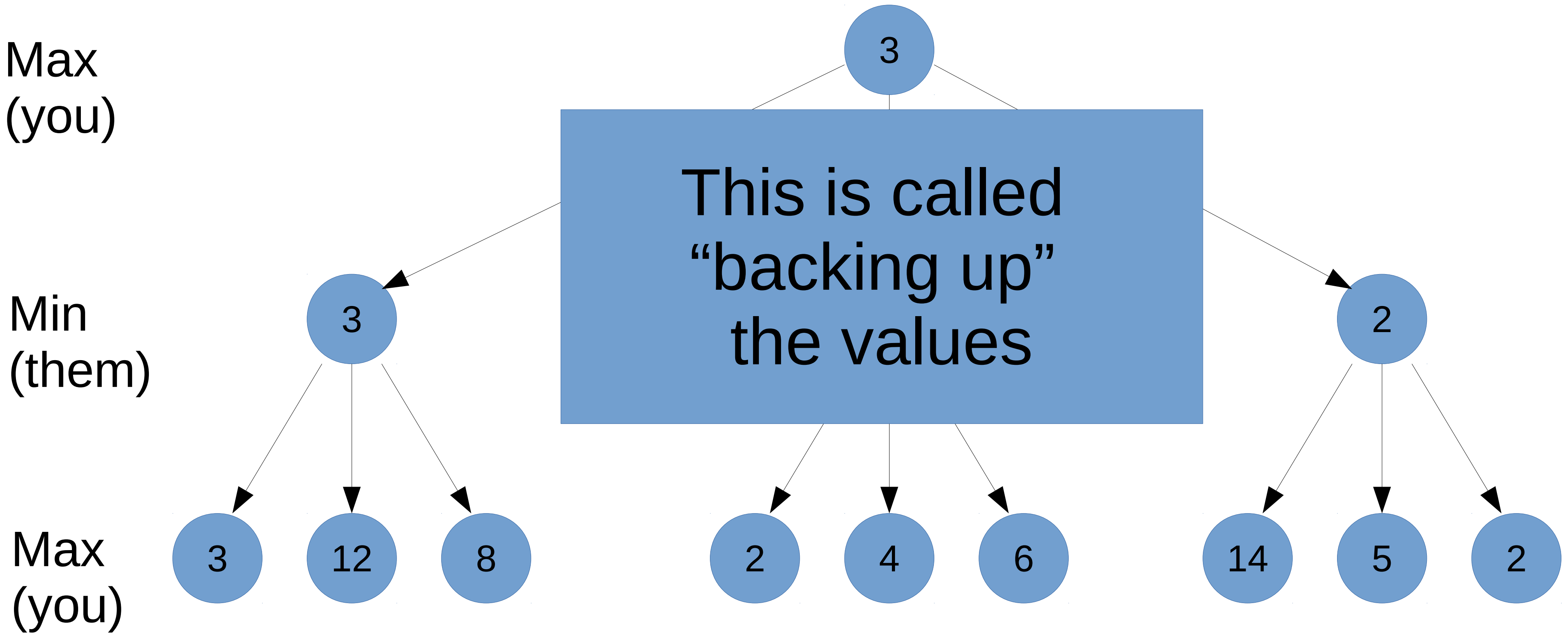
# What is Minimax?

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



# What is Minimax?

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

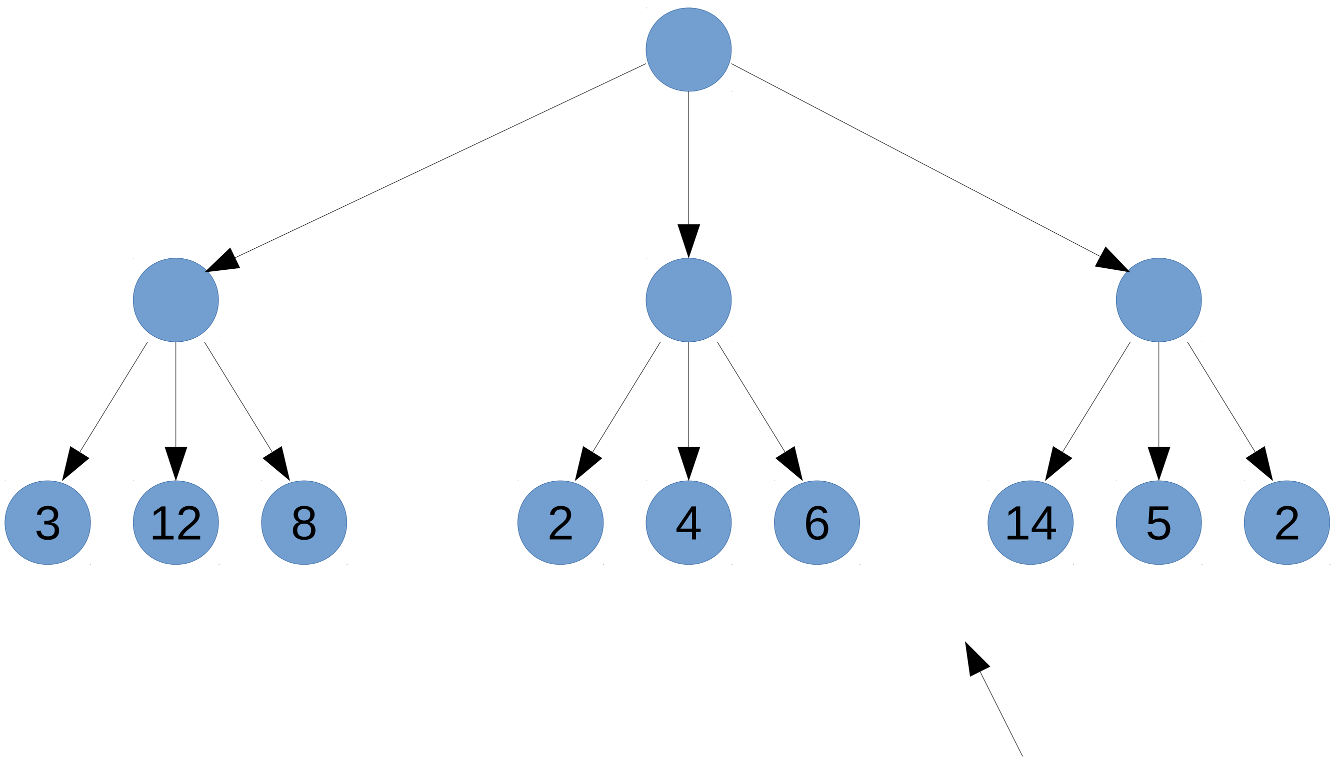



This is called
“backing up”
the values

# What is Minimax?

Okay – so we know how to back up values ...

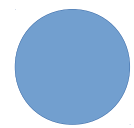
… but, how do we construct the tree?




This tree is already built...

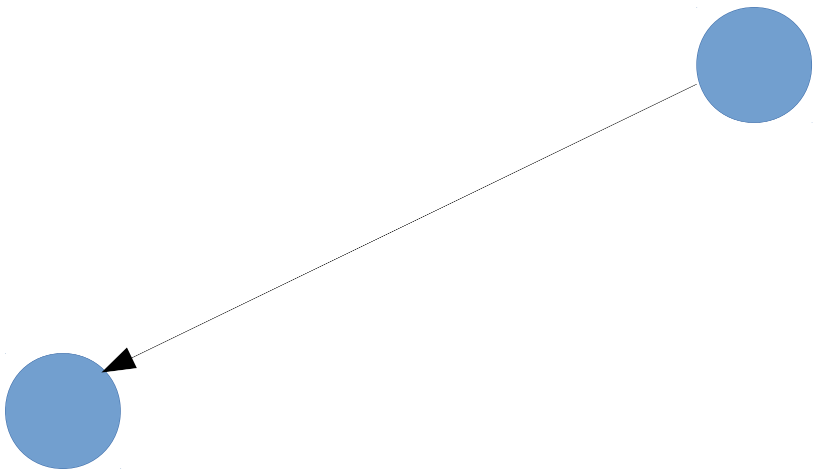
# What is Minimax?

Notice that we only get utilities at the *bottom* of the tree …
– therefore, DFS makes sense.



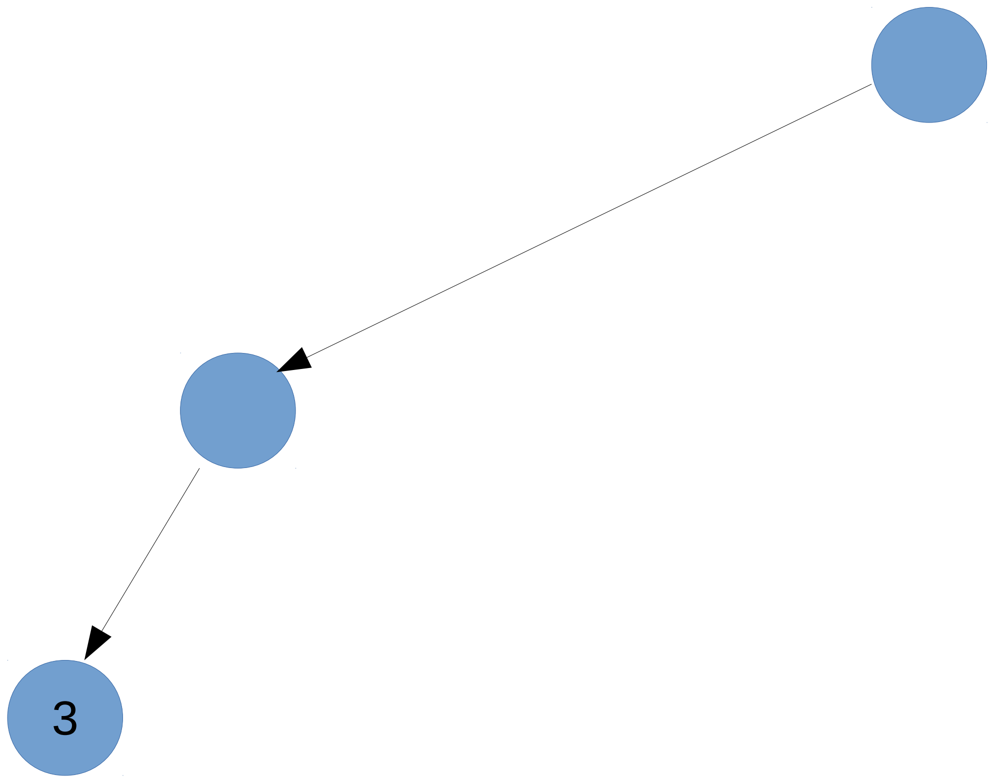
# What is Minimax?

Notice that we only get utilities at the *bottom* of the tree …
– therefore, DFS makes sense.



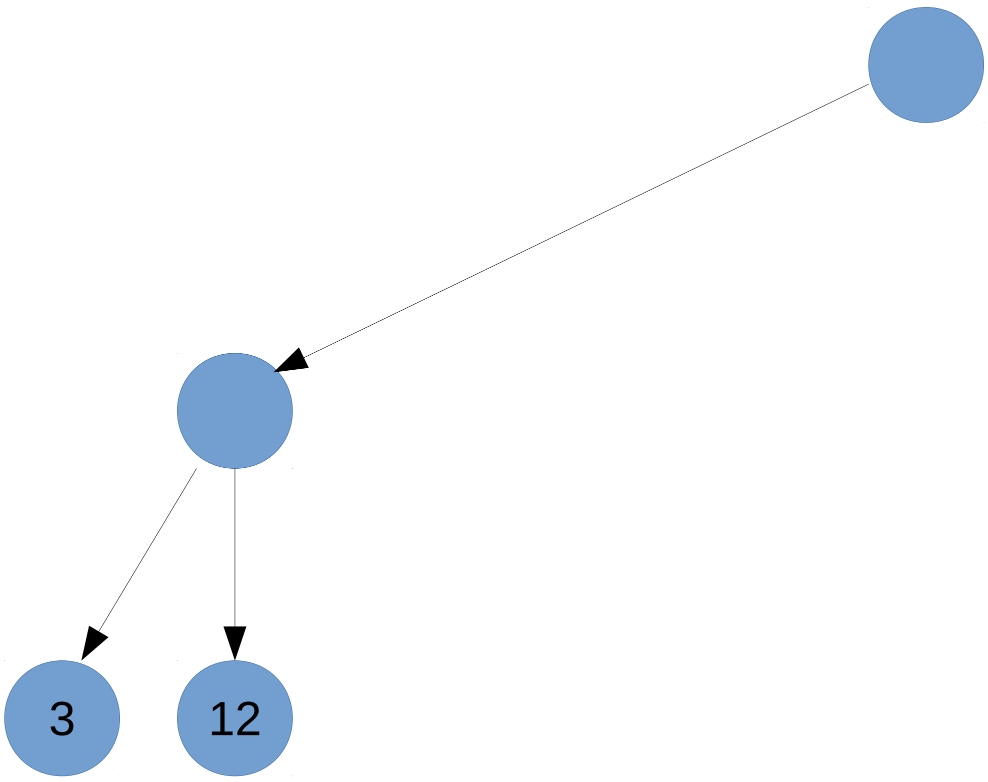
# What is Minimax?

Notice that we only get utilities at the *bottom* of the tree …
– therefore, DFS makes sense.



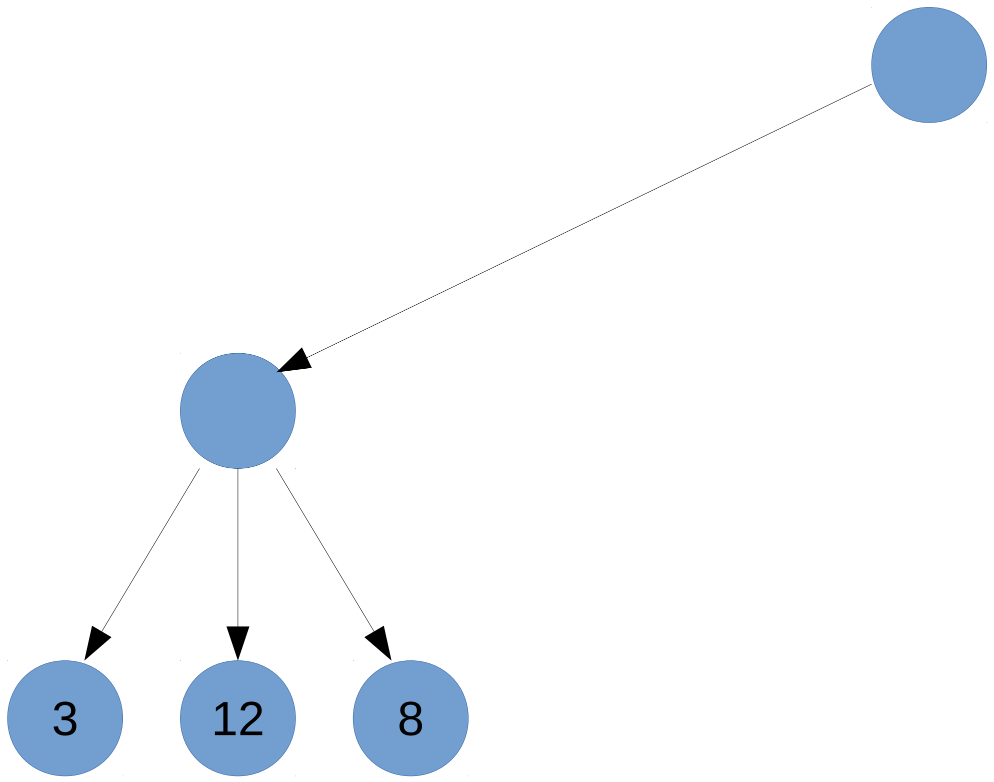
# What is Minimax?

Notice that we only get utilities at the *bottom* of the tree …
– therefore, DFS makes sense.



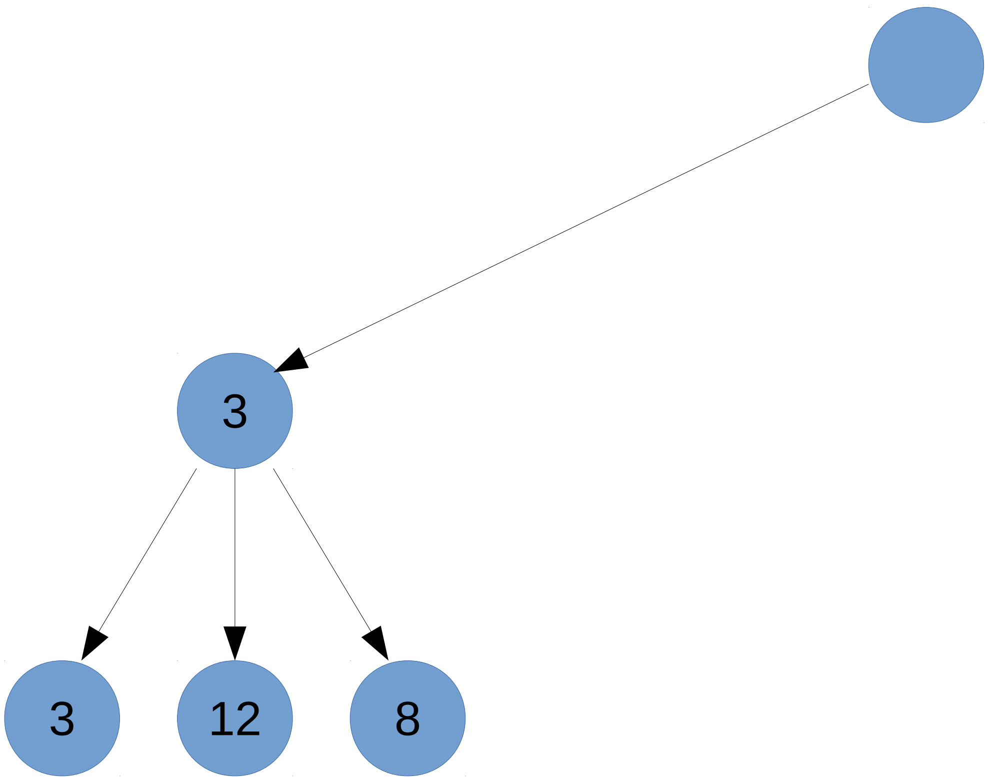
# What is Minimax?

Notice that we only get utilities at the *bottom* of the tree …
– therefore, DFS makes sense.



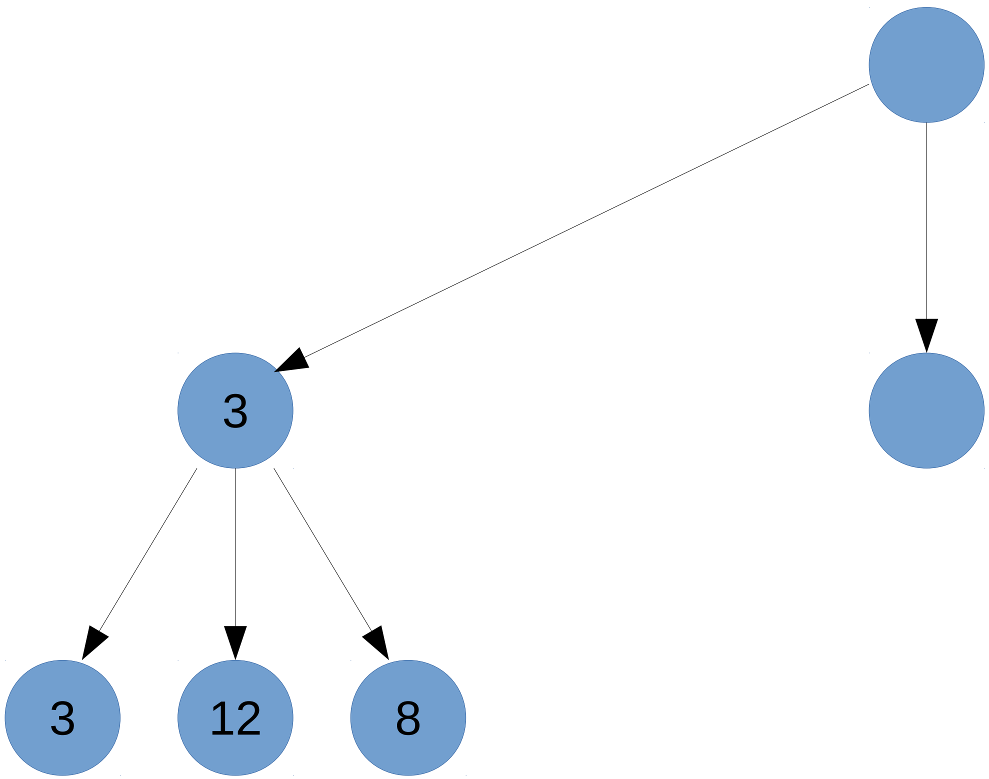
# What is Minimax?

Notice that we only get utilities at the *bottom* of the tree …
– therefore, DFS makes sense.



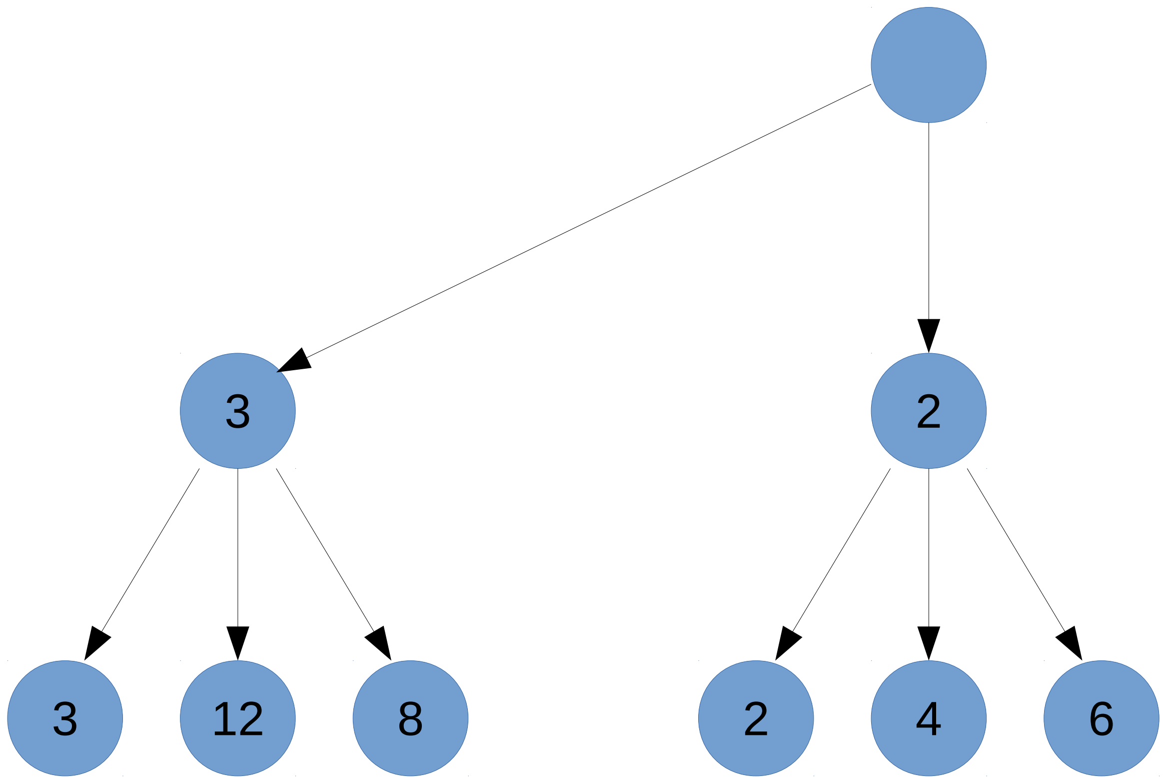
# What is Minimax?

Notice that we only get utilities at the *bottom* of the tree …
– therefore, DFS makes sense.



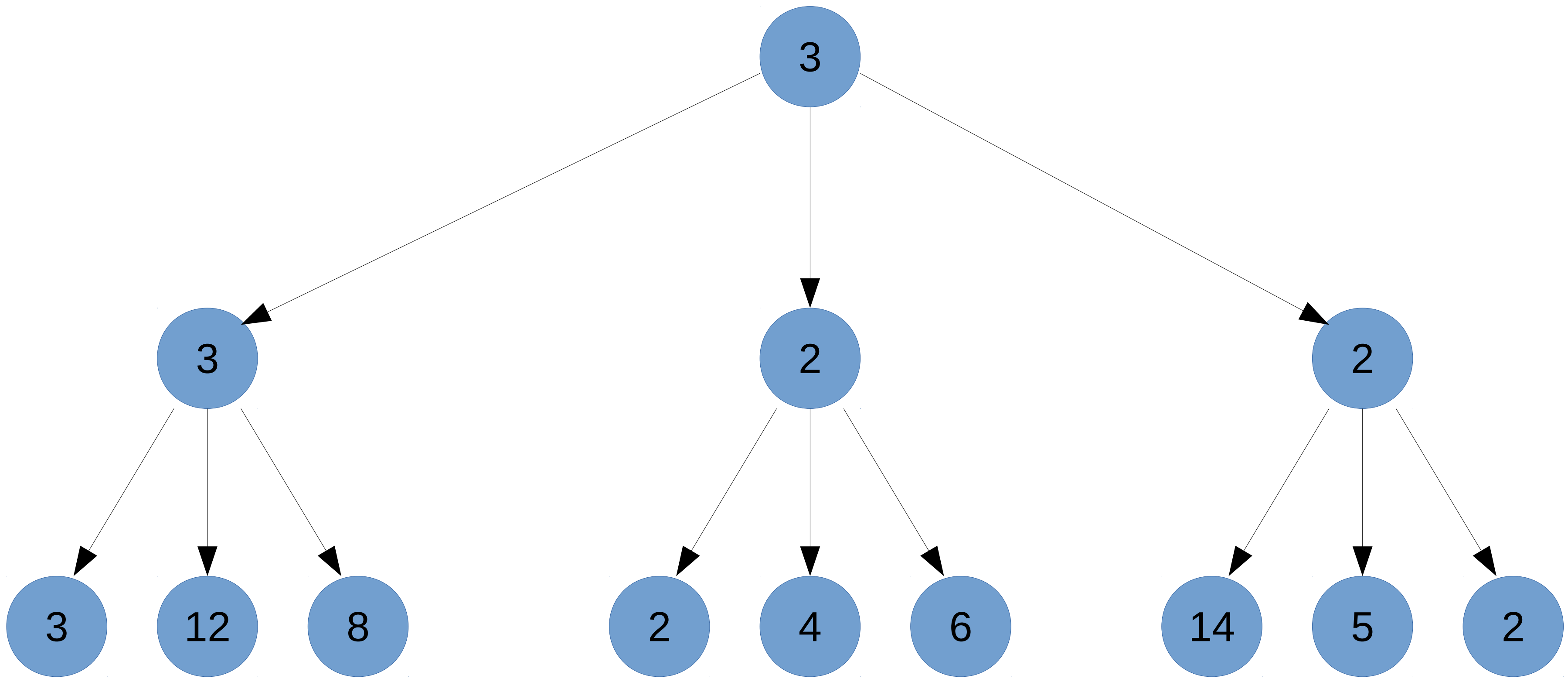
# What is Minimax?

Notice that we only get utilities at the *bottom* of the tree … – therefore, DFS makes sense.



# What is Minimax?

Notice that we only get utilities at the *bottom* of the tree …
– therefore, DFS makes sense.



# What is Minimax?

Notice that we only get utilities at the *bottom* of the tree …
– therefore, DFS makes sense.
– since most games have forward progress, the distinction between tree search and graph search is less important

# What is Minimax?

**function** MINIMAX-DECISION(*state*) **returns** *an action*
  **return** $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

---

**function** MAX-VALUE(*state*) **returns** *a utility value*
  **if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  **for each** *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT($s$, $a$)))
  **return** $v$

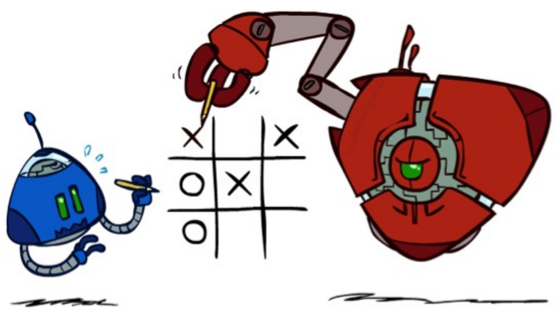
---

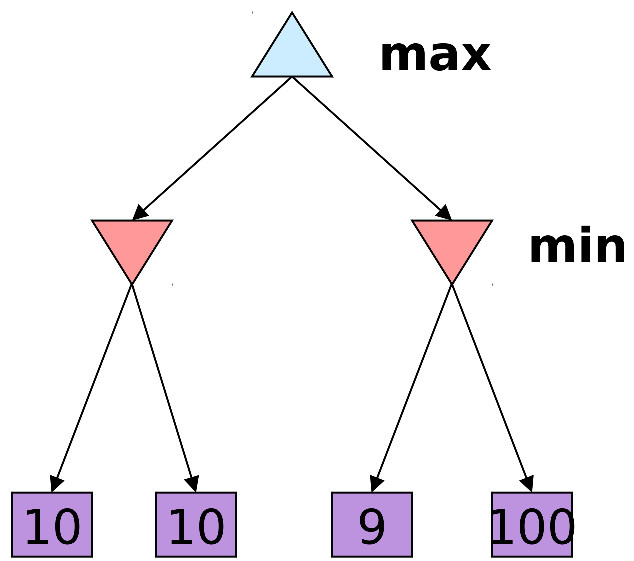
**function** MIN-VALUE(*state*) **returns** *a utility value*
  **if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  **for each** *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT($s$, $a$)))
  **return** $v$

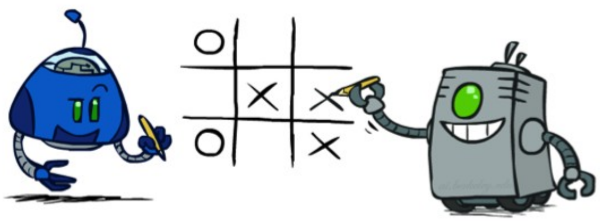
**Figure 5.3** An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in S} f(a)$ computes the element $a$ of set $S$ that has the maximum value of $f(a)$.

# Minimax properties

Is it always correct to assume your opponent plays optimally?

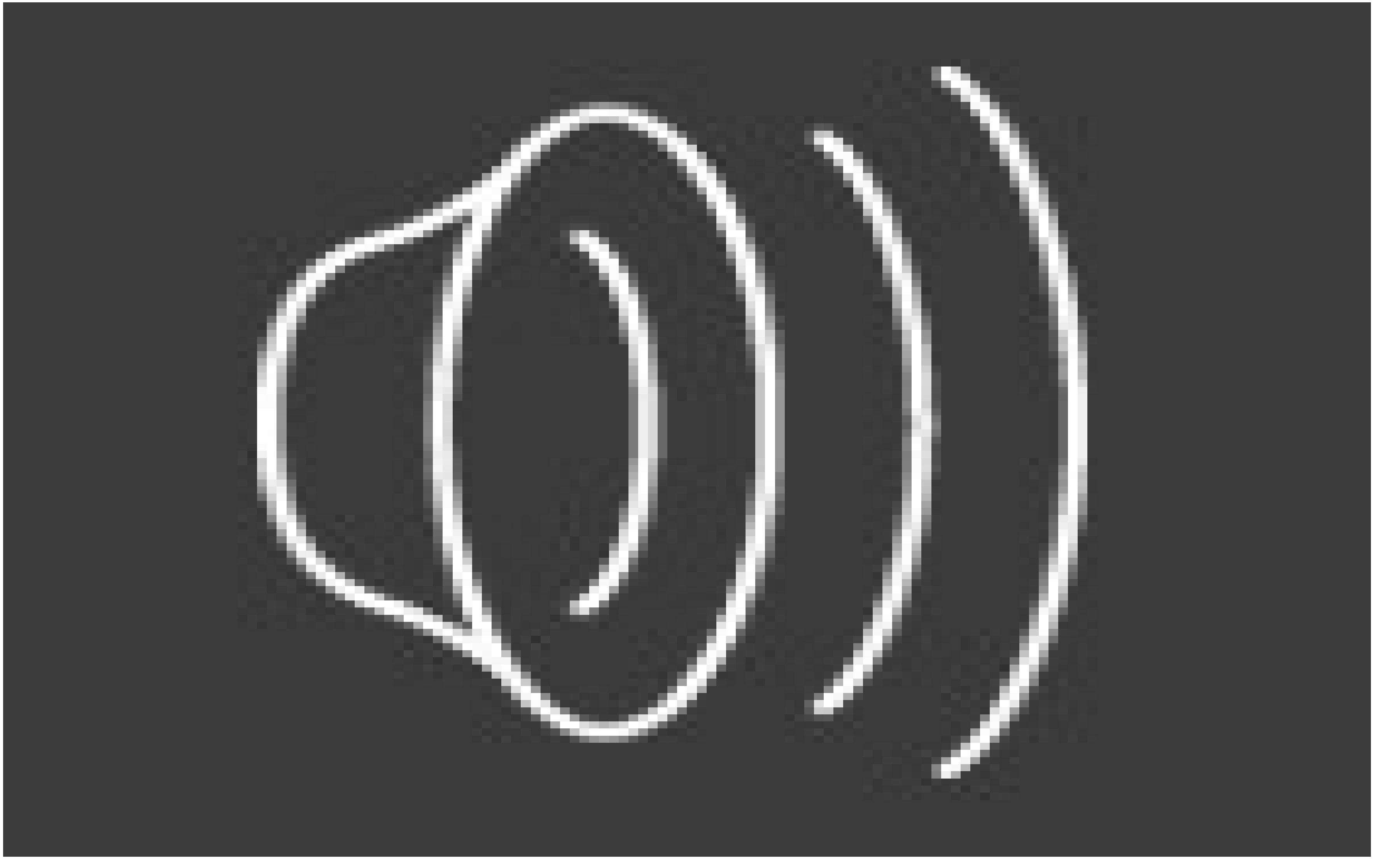




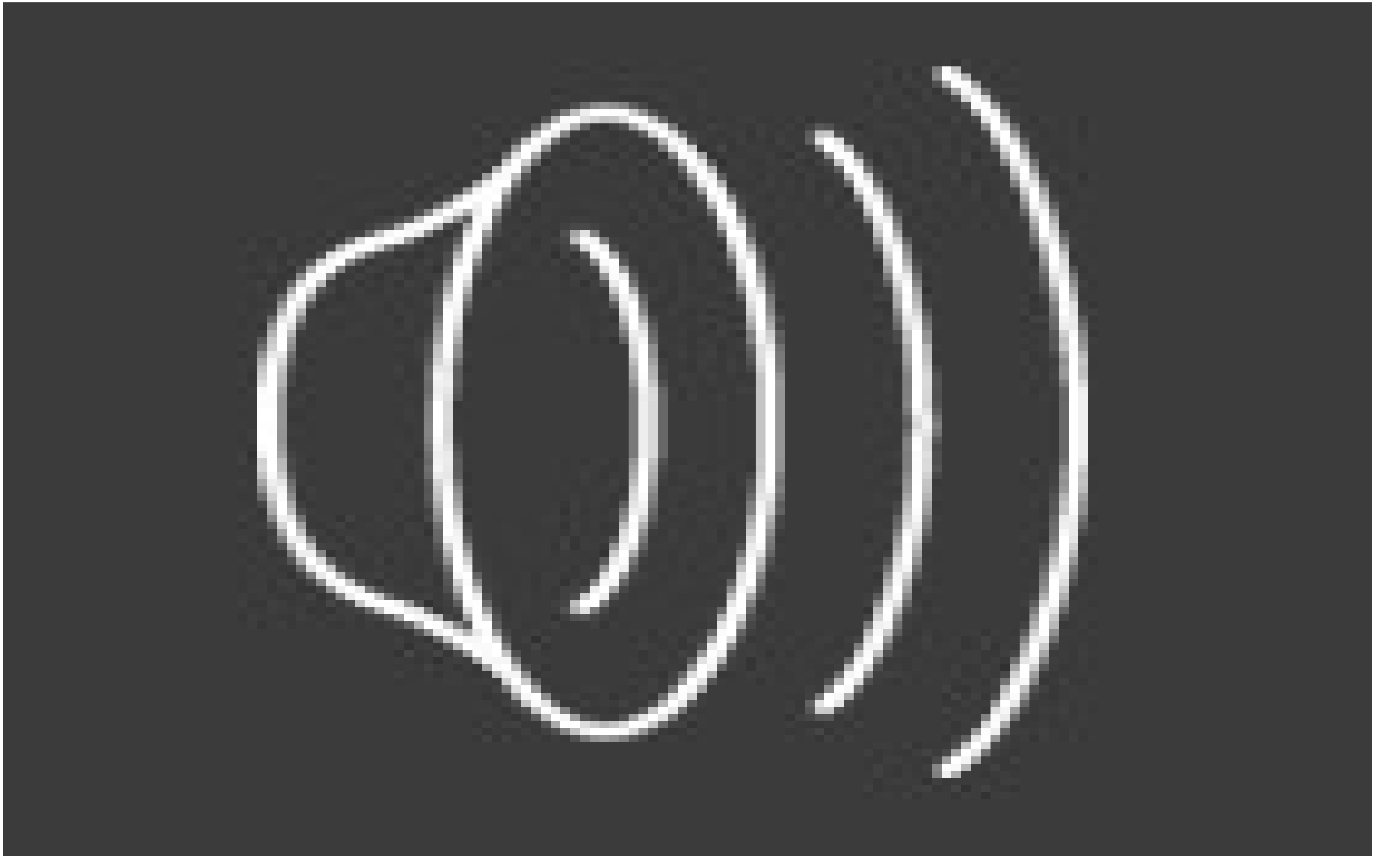
Slide: Berkeley CS188 course notes (downloaded Summer 2015)

# Minimax vs “expectimax”



Slide: Berkeley CS188 course notes (downloaded Summer 2015)

# Minimax vs “expectimax”



Slide: Berkeley CS188 course notes (downloaded Summer 2015)

# Minimax properties

Is minimax optimal? Is it complete?

# Minimax properties

Is minimax optimal? Is it complete?

Time complexity = ?

Space complexity = ?

# Minimax properties

Is minimax optimal? Is it complete?

Time complexity = $O(b^d)$

Space complexity = $O(bd)$

# Minimax properties

Is minimax optimal? Is it complete?

Time complexity = $O(b^d)$

Space complexity = $O(bd)$

Is it practical? In chess, b=35, d=100

# Minimax properties

Is minimax optimal? Is it complete?

Time complexity = $O(b^d)$

Space complexity = $O(bd)$

Is it practical? In chess, b=35, d=100

$O(35^{100})$ is a big number...

# Minimax properties

Is minimax optimal? Is it complete?

Time complexity = $O(b^d)$

Space complexity = $O(bd)$

Is it practical? In chess, b=35, d=100

$O(35^{100})$ is a big number...

So what can we do?

# Evaluation functions

Key idea: cut off search at a certain depth and give the corresponding nodes an estimated value.

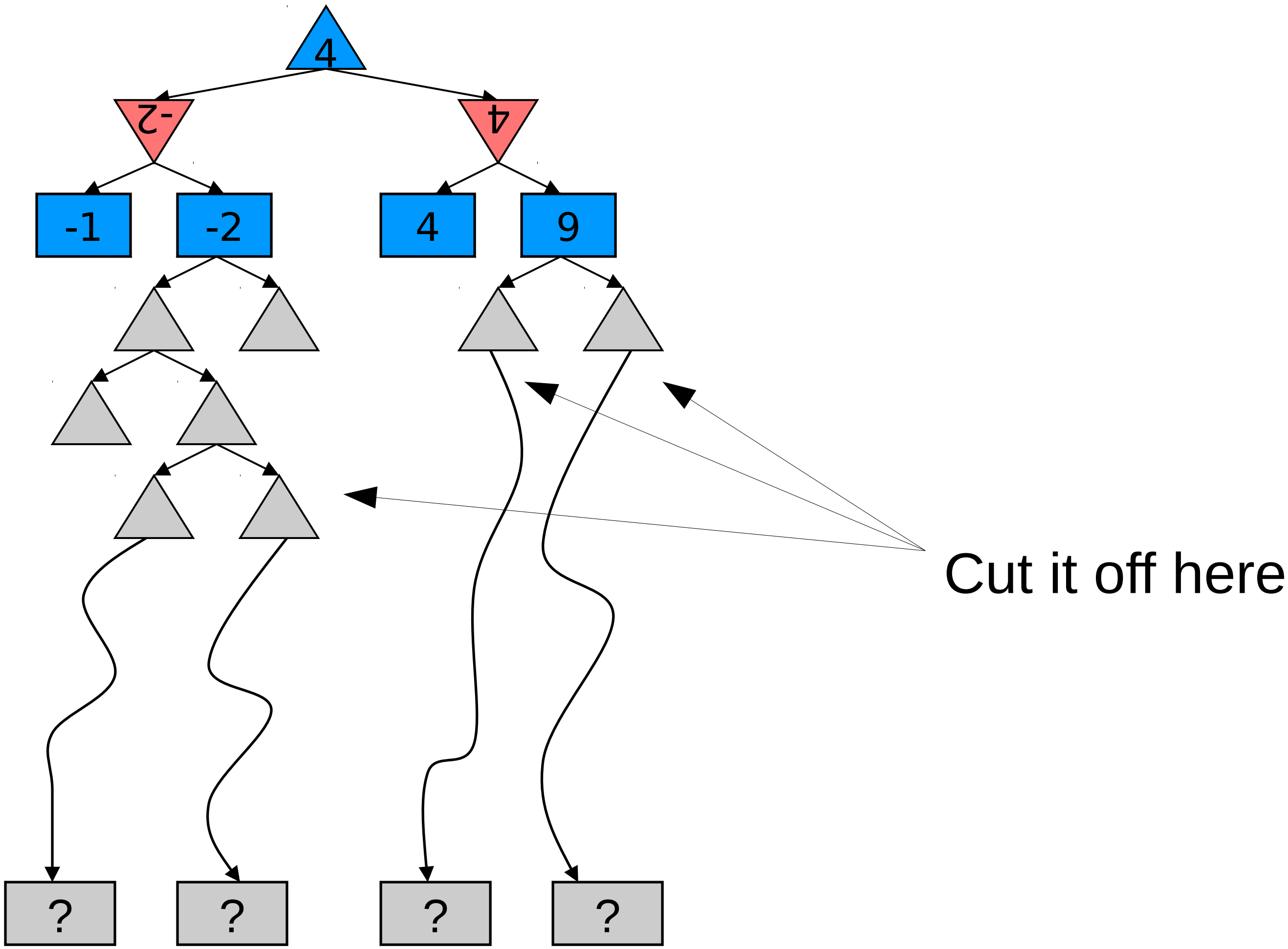



Image: Berkeley CS188 course notes (downloaded Summer 2015)

# Evaluation functions

Key idea: cut off search at a certain depth and give the corresponding nodes an estimated value.

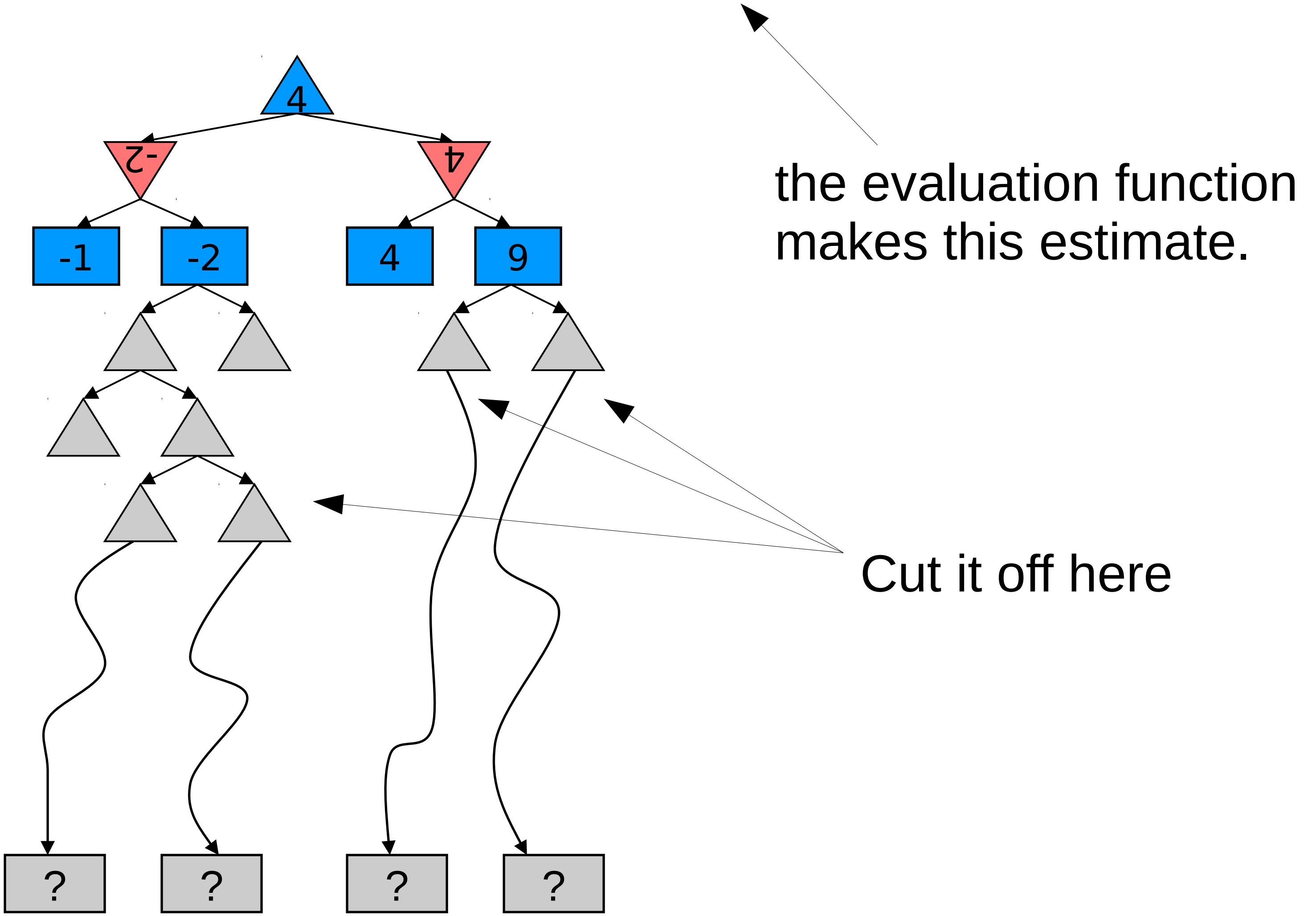



Image: Berkeley CS188 course notes (downloaded Summer 2015)

# Evaluation functions

How does the evaluation function make the estimate?
– depends upon domain

For example, in chess, the value of a state might equal the sum of piece values.

– a pawn counts for 1
– a rook counts for 5
– a knight counts for 3
...

# A weighted linear evaluation function

$$eval(s) = w_1 f_1(s) + \cdots + w_n f_n(s)$$

$f_1(s) \equiv$ number of pawns on the board

$f_2(s) \equiv$ number of knights on the board

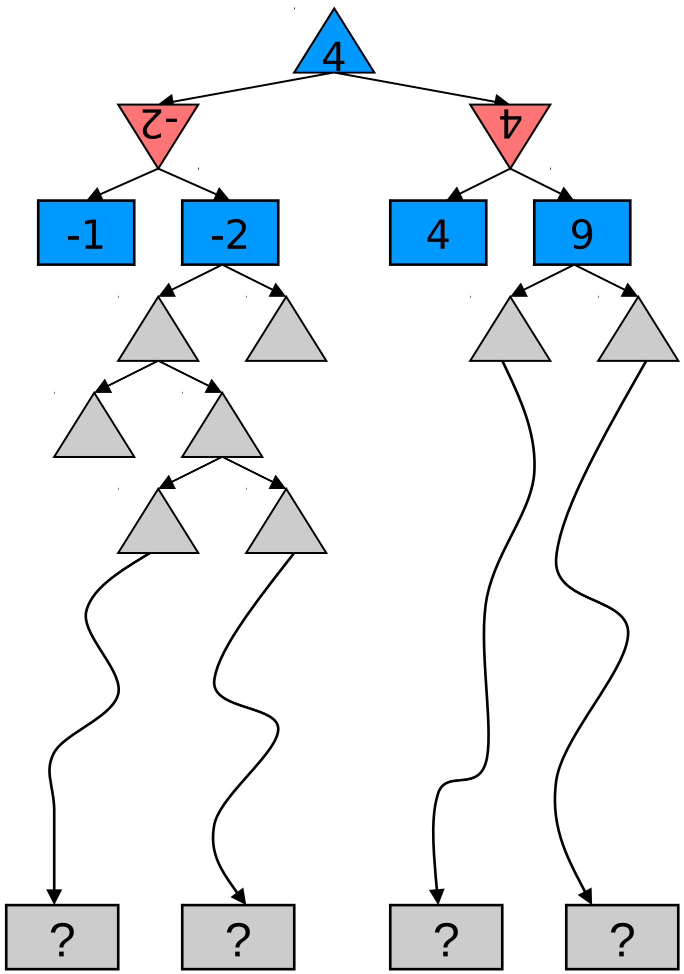
$\vdots$

$w_1 = 1$ ← A pawn counts for 1

$w_2 = 3$ ← A knight counts for 3

$\vdots$

# At what depth do you run the evaluation function?



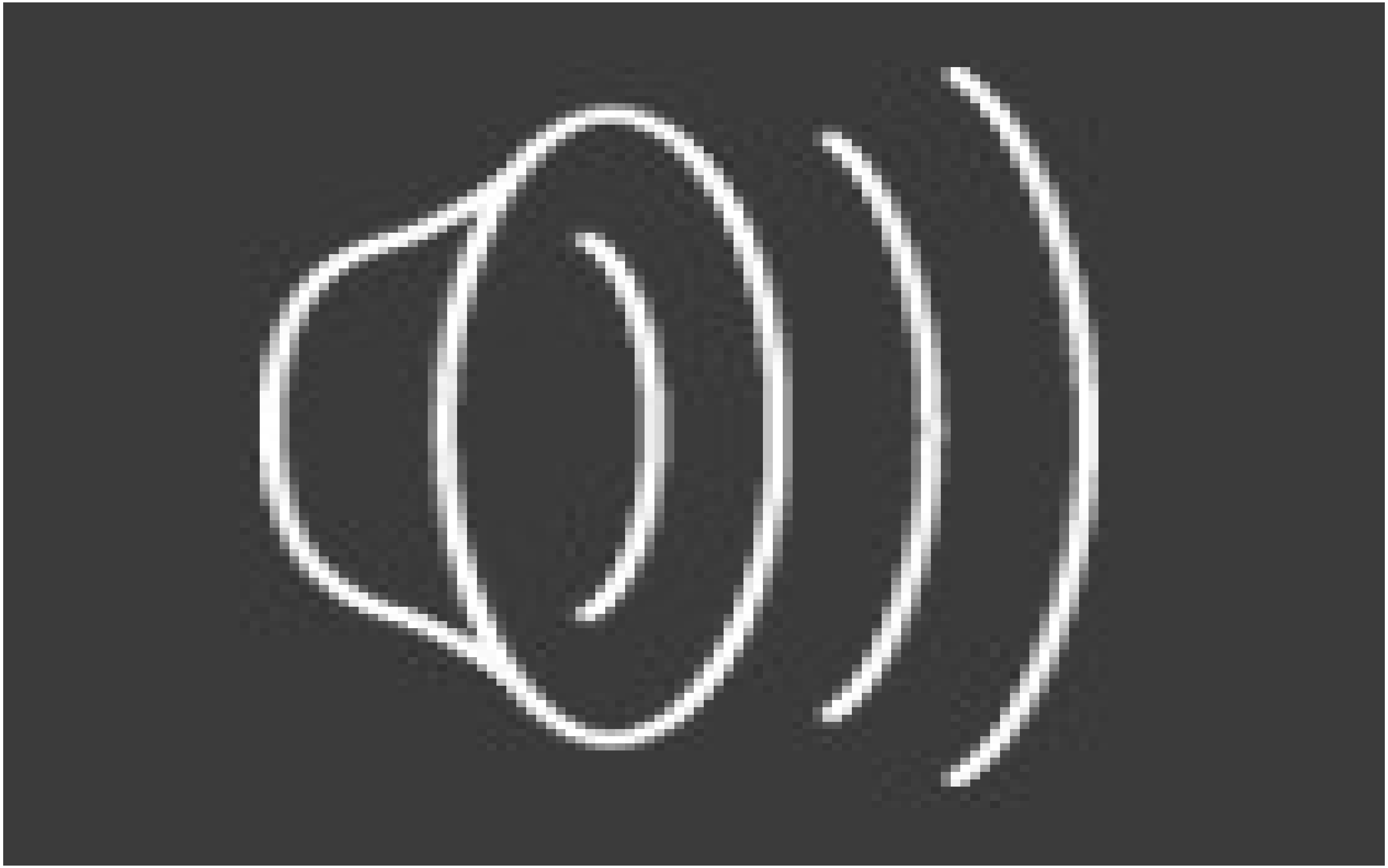

Option 1: cut off search at a fixed depth

Option 2: cut off search at quiescient states deeper than a certain threshold

Option 3: ?

The deeper your threshold, the less the quality of the evaluation function matters...

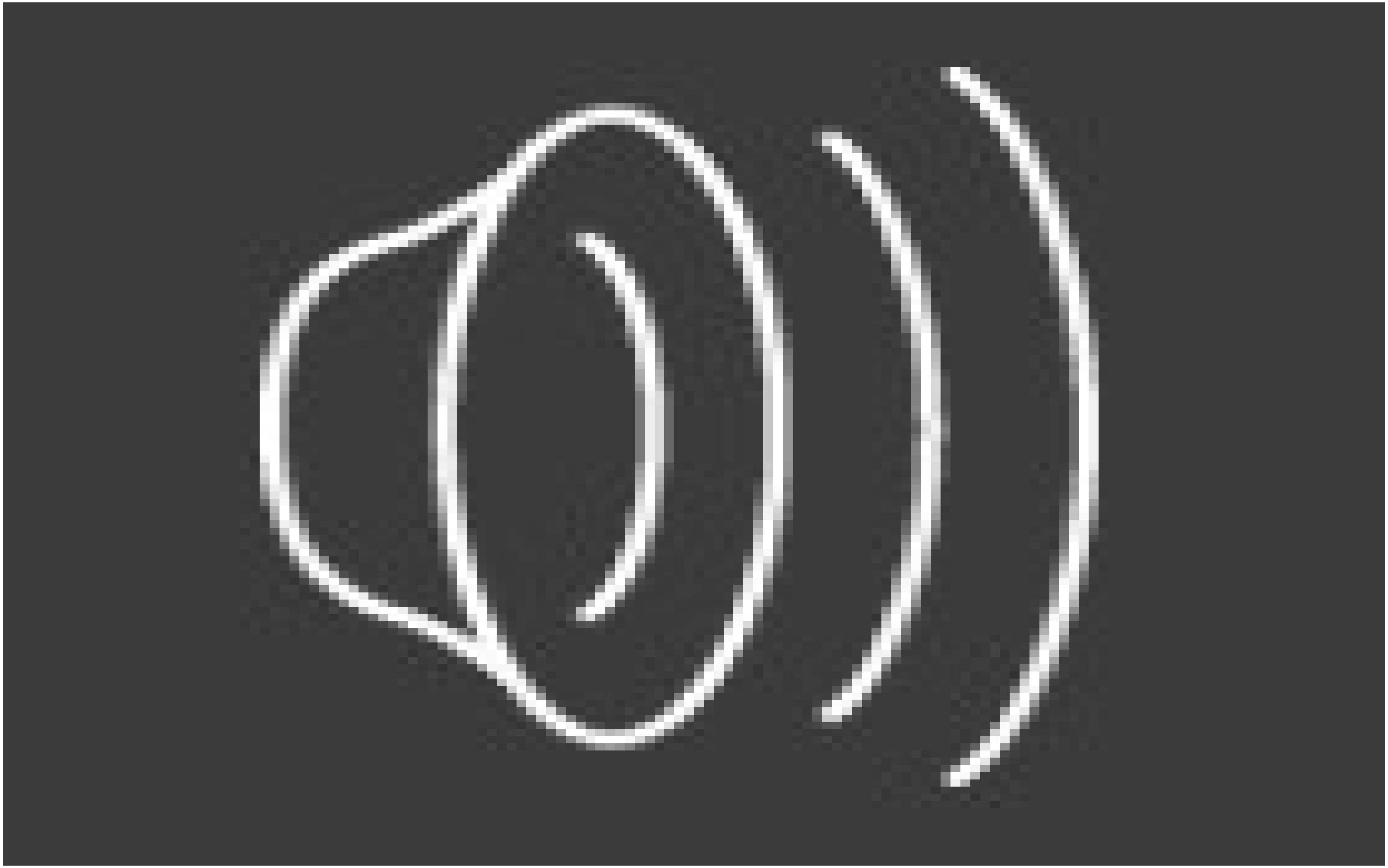
# At what depth do you run the evaluation function?



Search depth=2

Slide: Berkeley CS188 course notes (downloaded Summer 2015)

# At what depth do you run the evaluation function?



Search depth=10

Slide: Berkeley CS188 course notes (downloaded Summer 2015)

# Alpha/Beta pruning

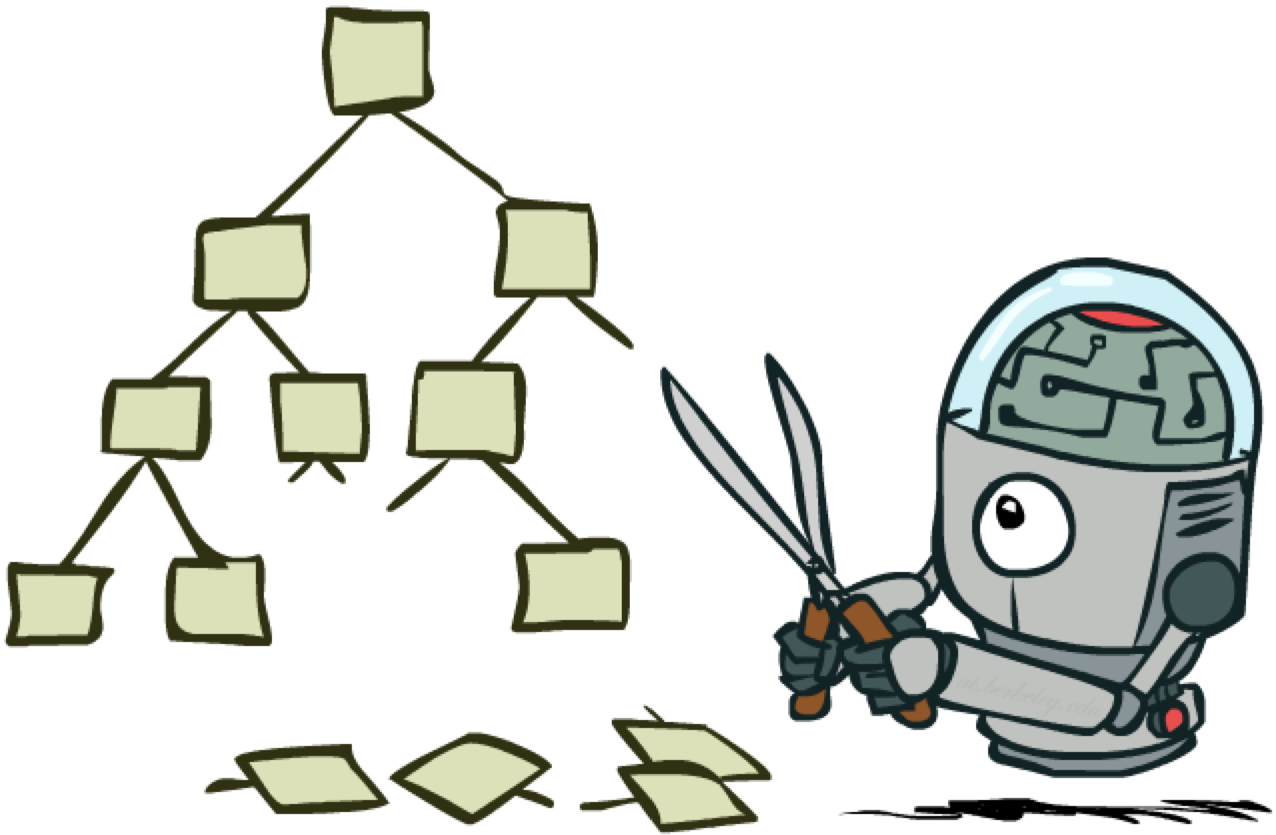
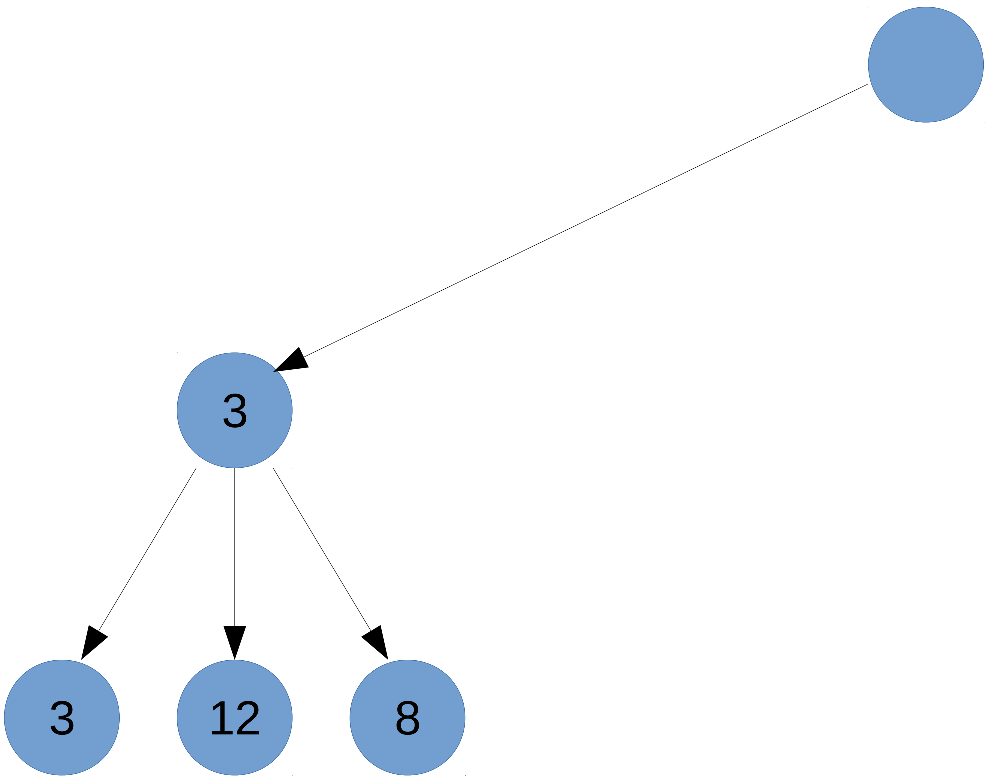
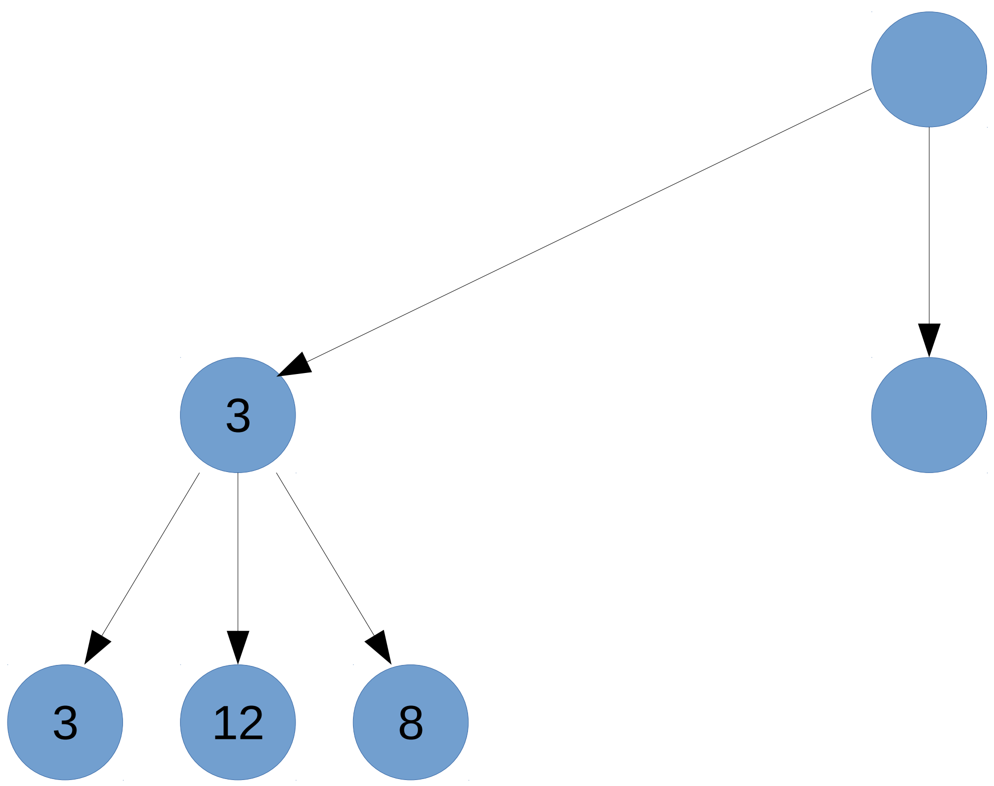


Image: Berkeley CS188 course notes (downloaded Summer 2015)

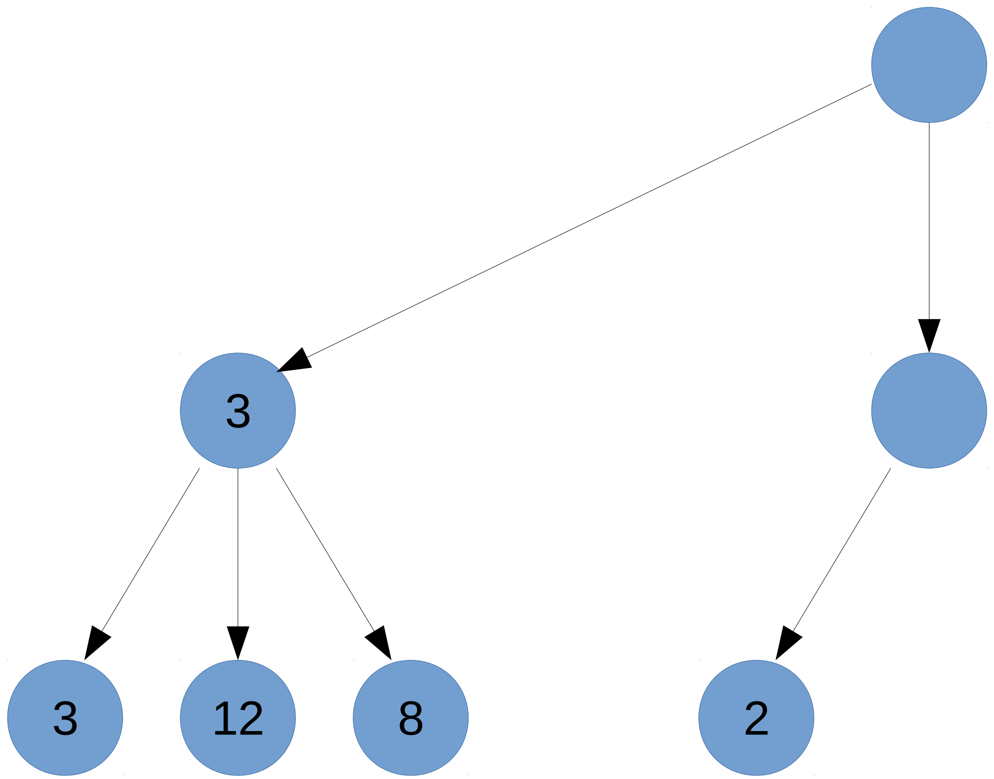
# Alpha/Beta pruning



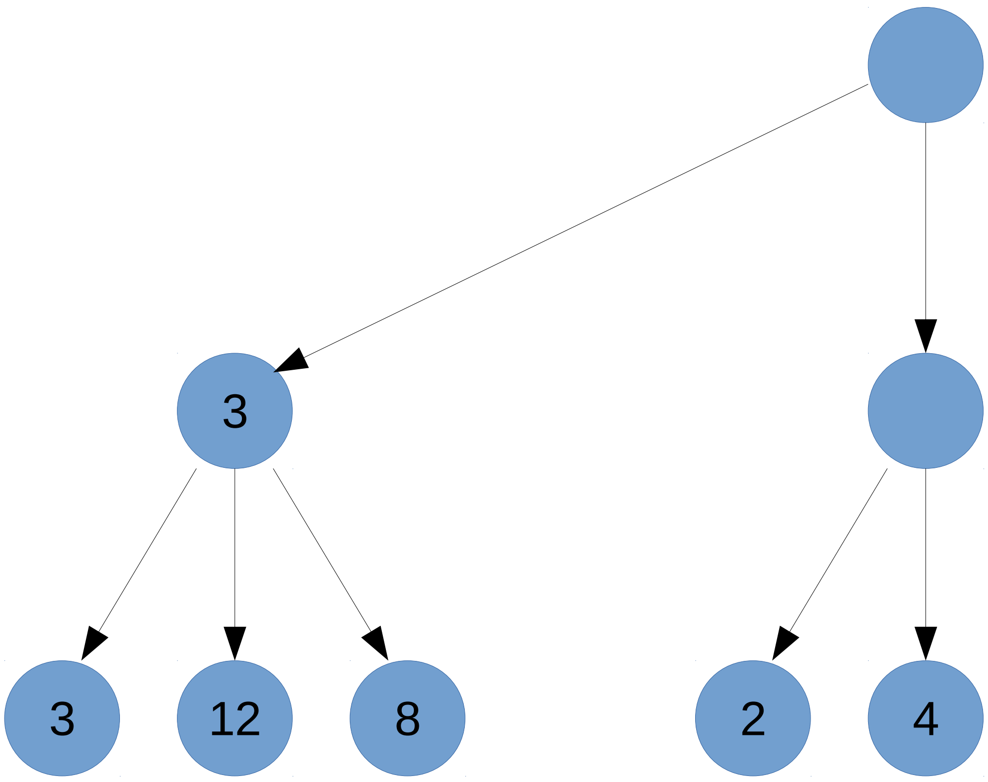
# Alpha/Beta pruning



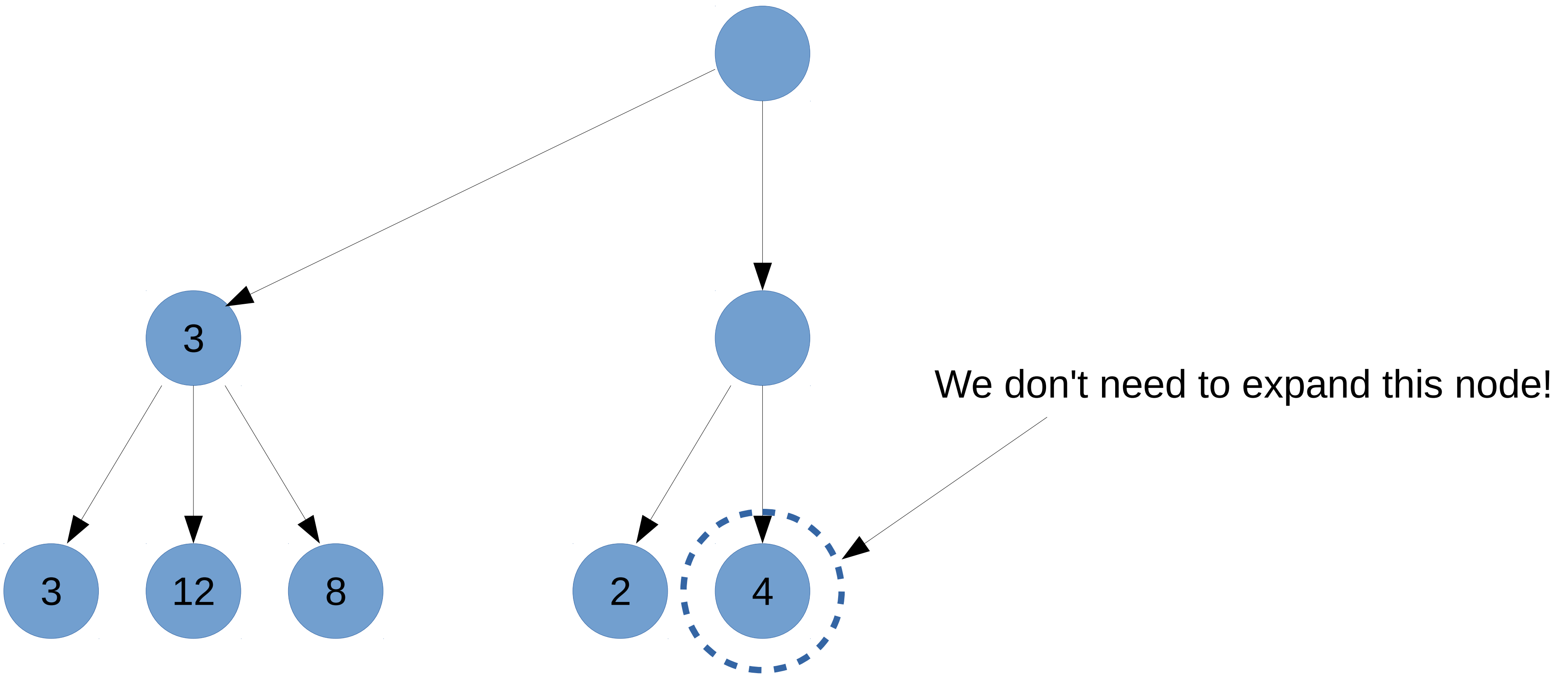
# Alpha/Beta pruning



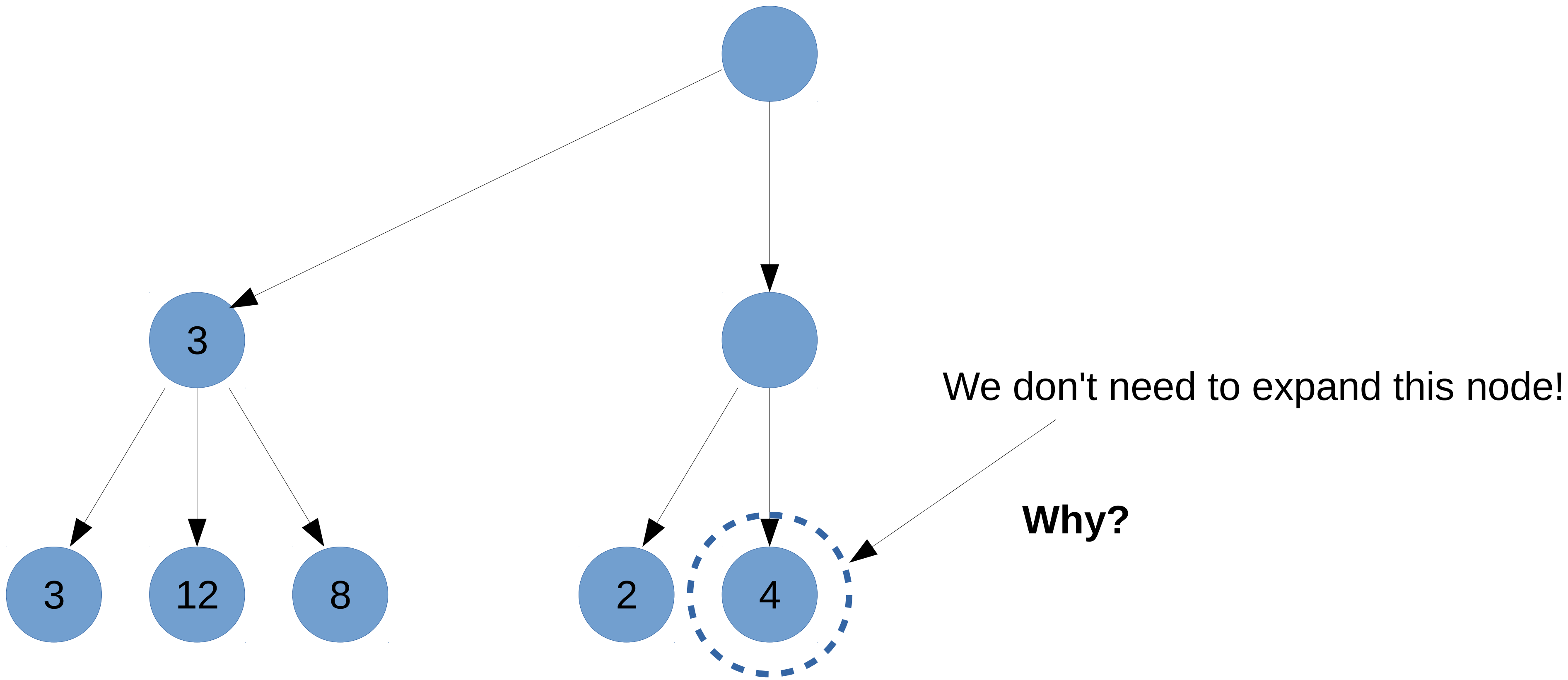
# Alpha/Beta pruning



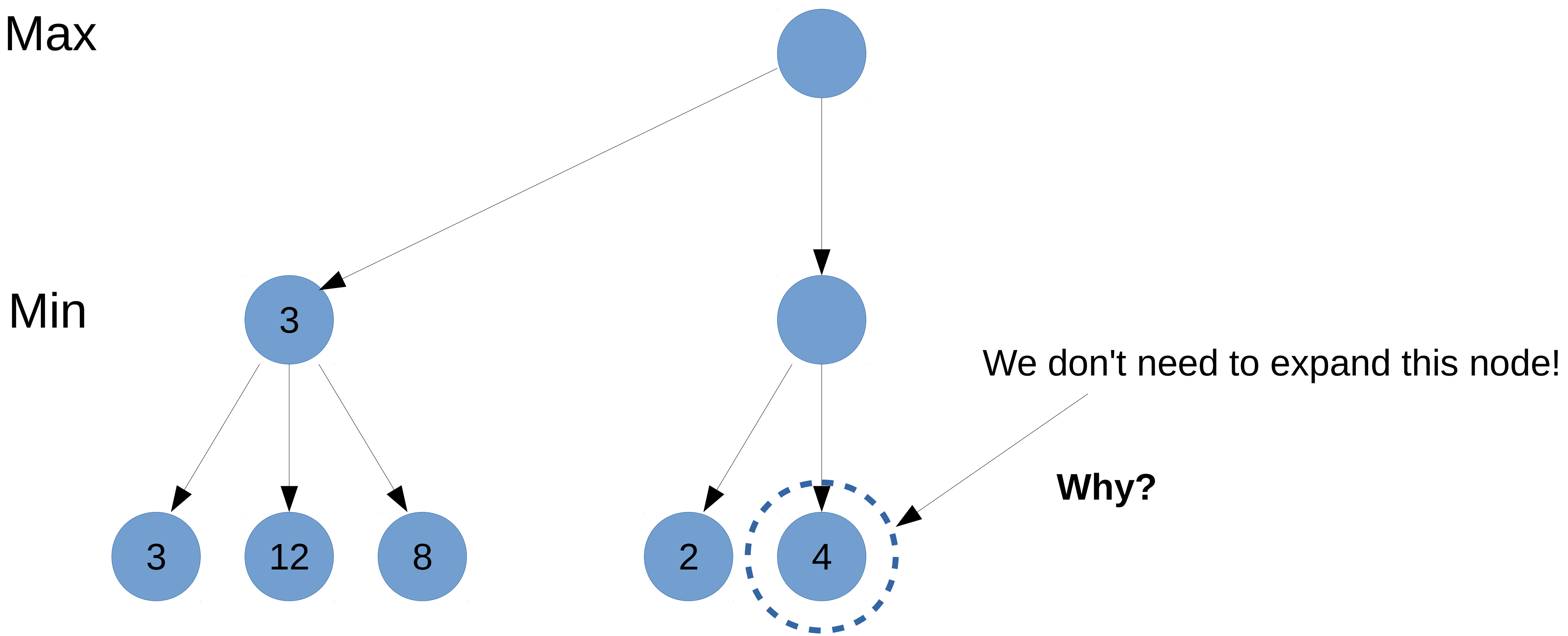
# Alpha/Beta pruning



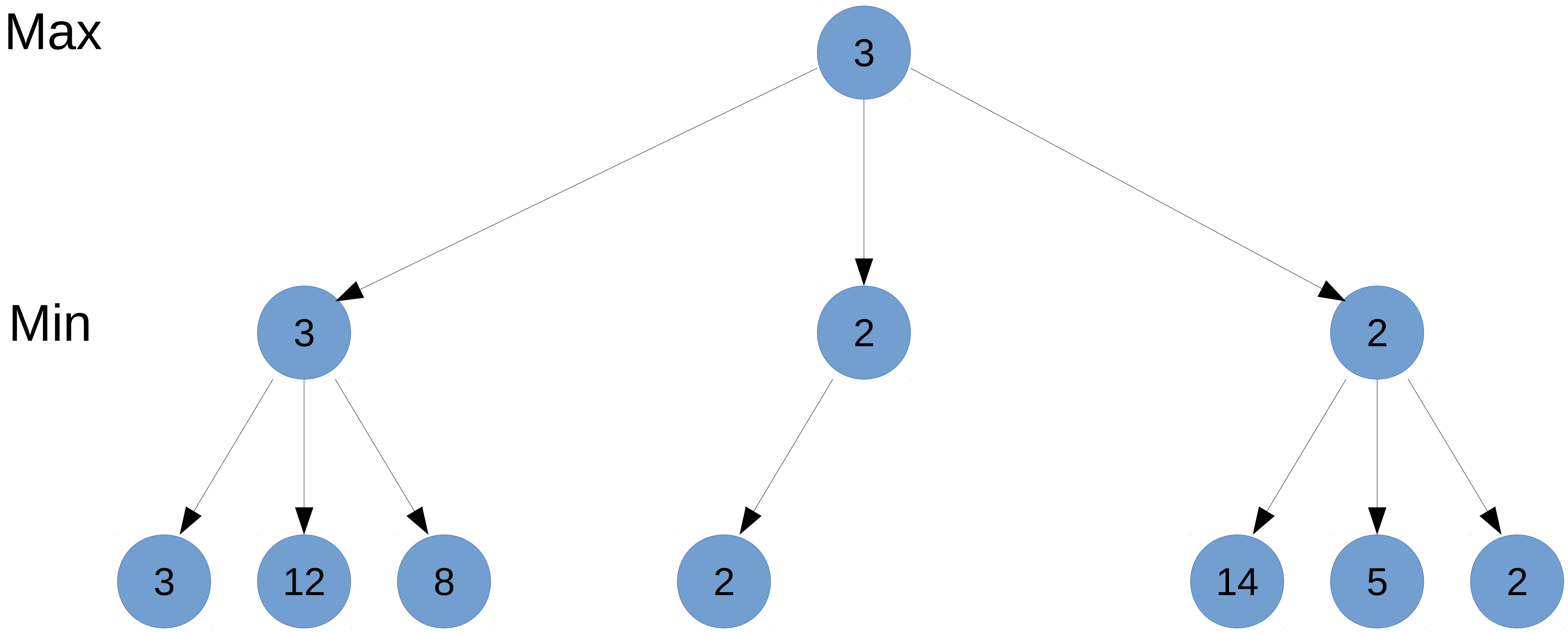
# Alpha/Beta pruning



# Alpha/Beta pruning

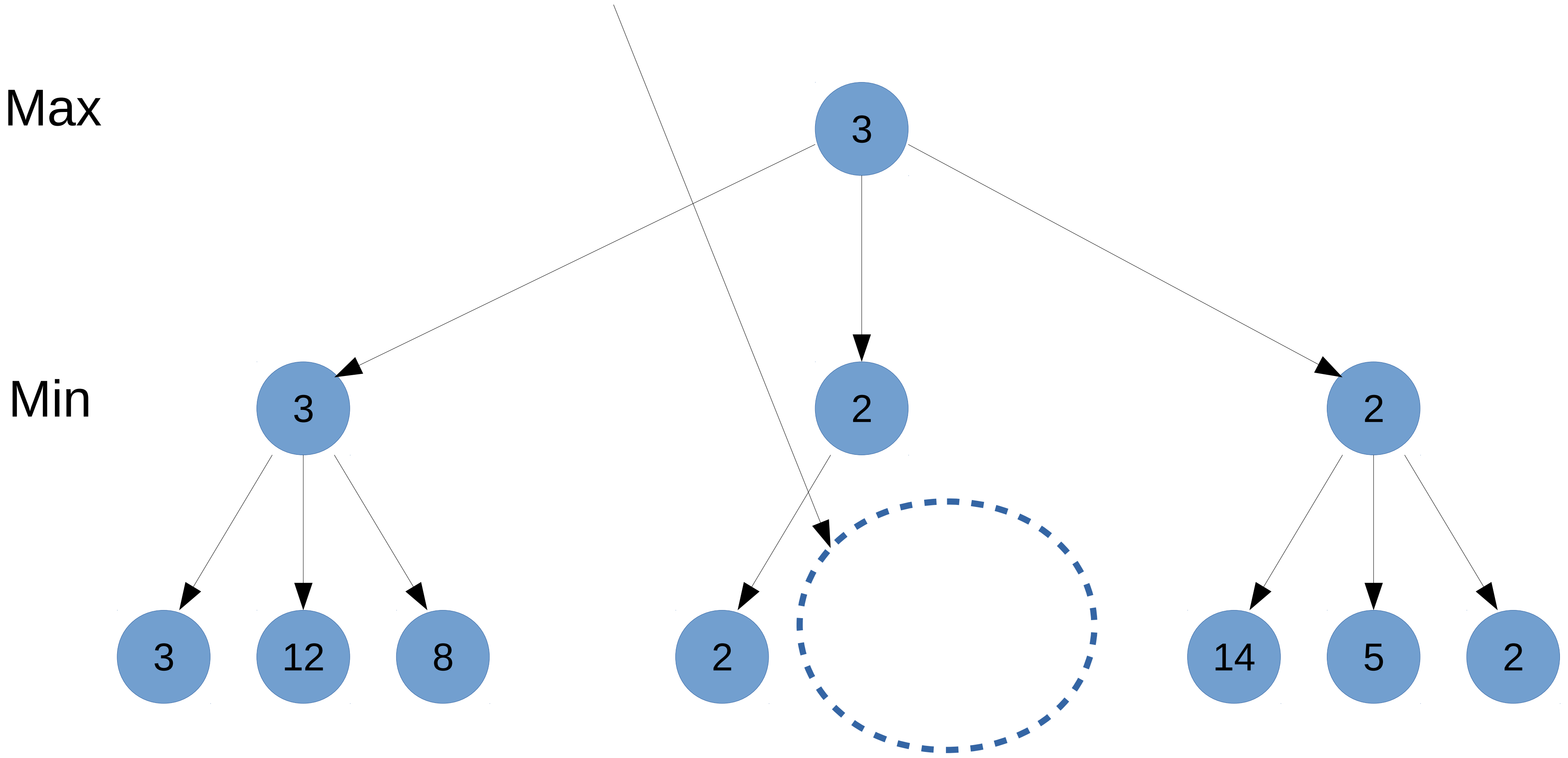


# Alpha/Beta pruning



# Alpha/Beta pruning

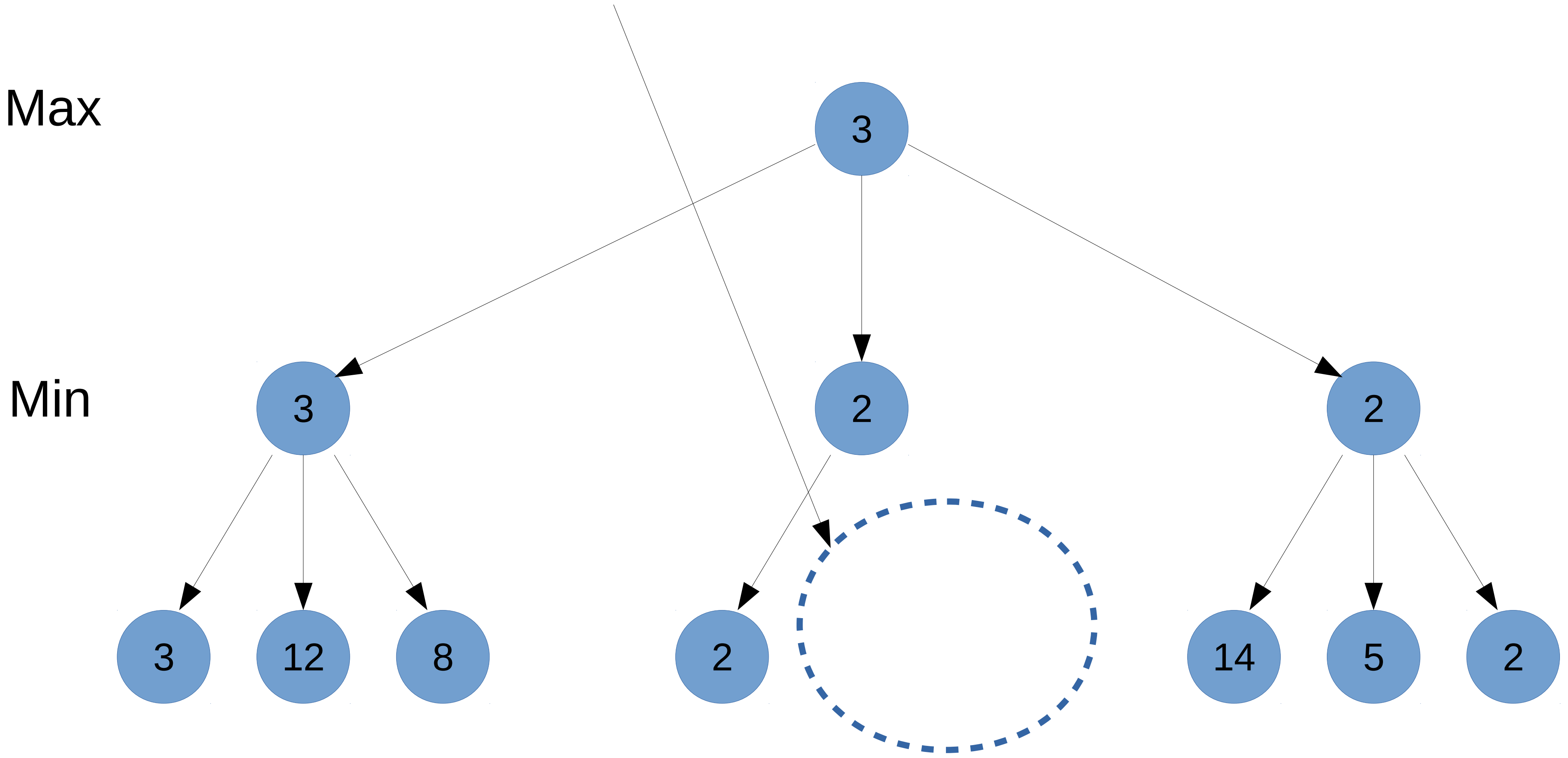
So, we don't need to expand these nodes
in order to back up correct values!



# Alpha/Beta pruning

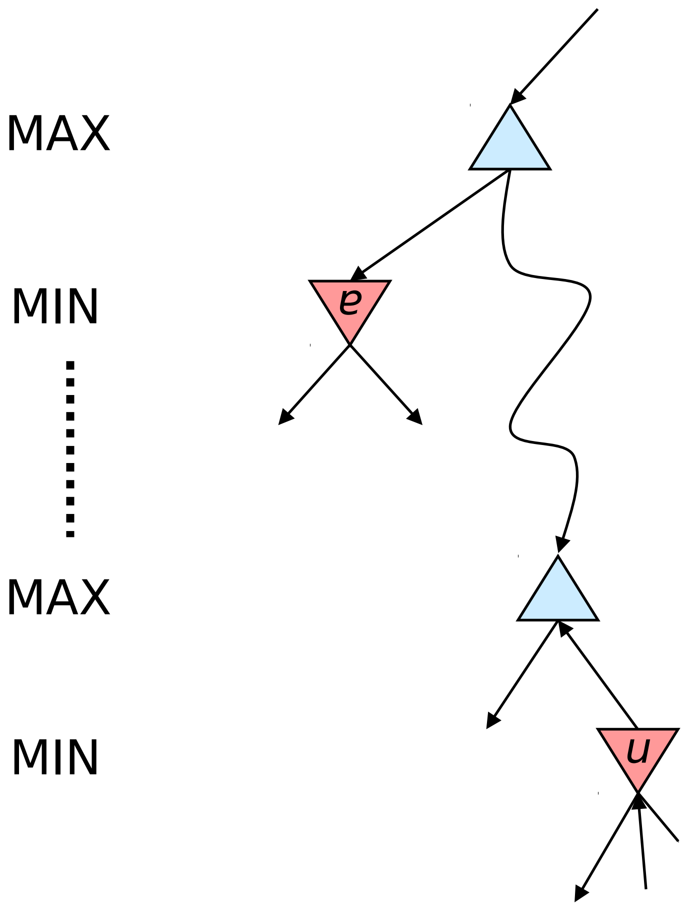
So, we don't need to expand these nodes
in order to back up correct values!

That's alpha-beta
pruning.



# Alpha/Beta pruning: algorithm idea

- 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

Slide: Berkeley CS188 course notes (downloaded Summer 2015)

# Alpha/Beta pruning: algorithm

α: best value so far for MAX along path to root

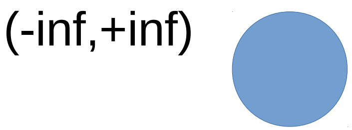
β: best value so far for MIN along path to root

```
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
```

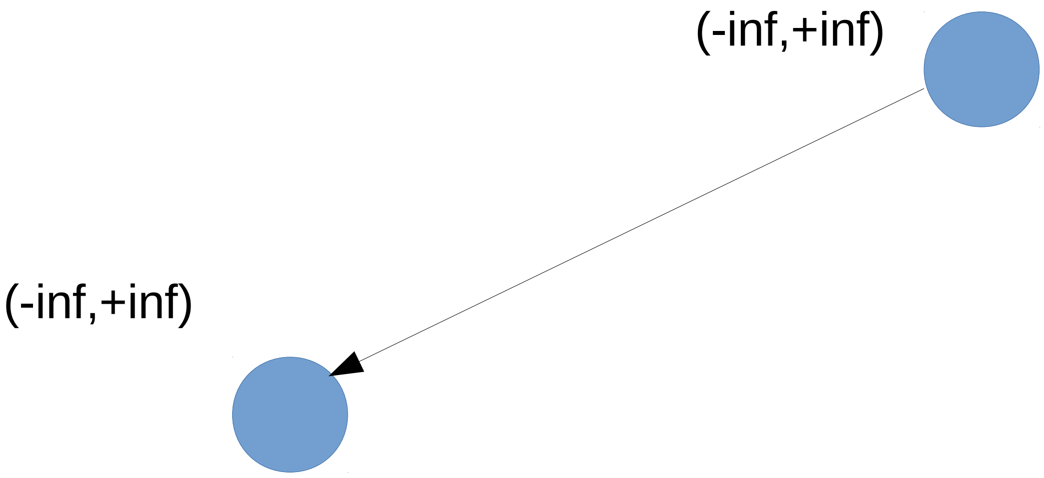
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v,
          value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Slide: adapted from Berkeley CS188 course notes (downloaded Summer 2015)

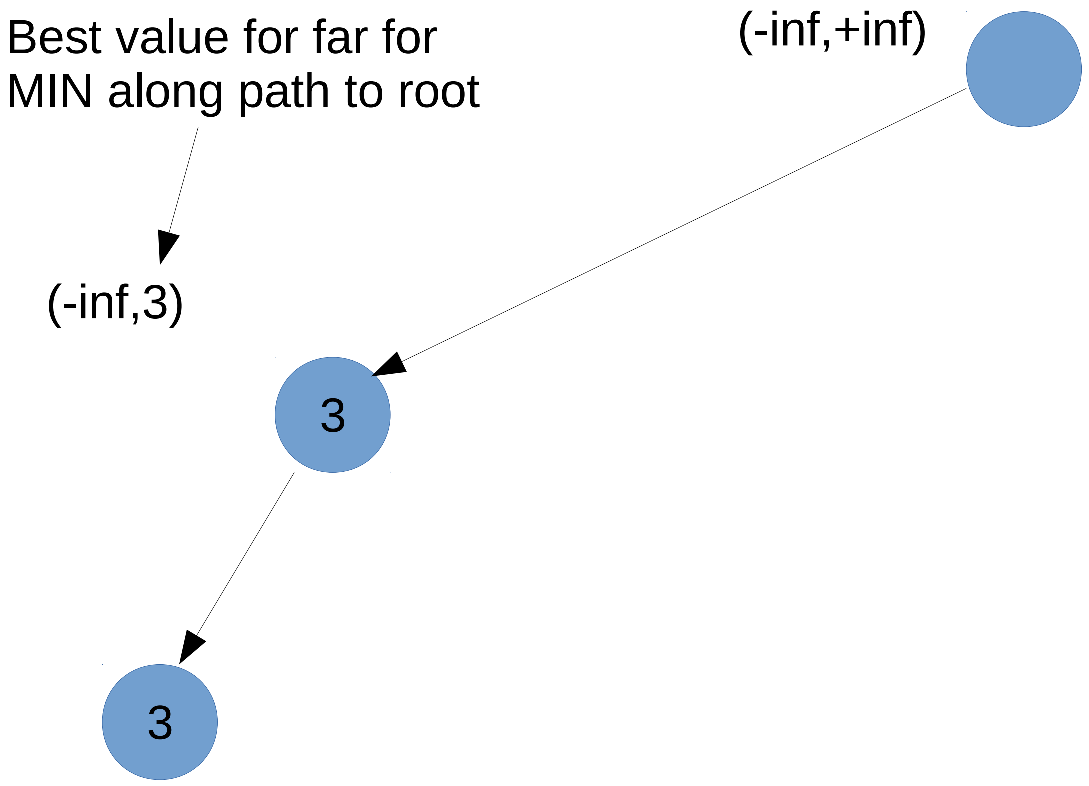
# Alpha/Beta pruning



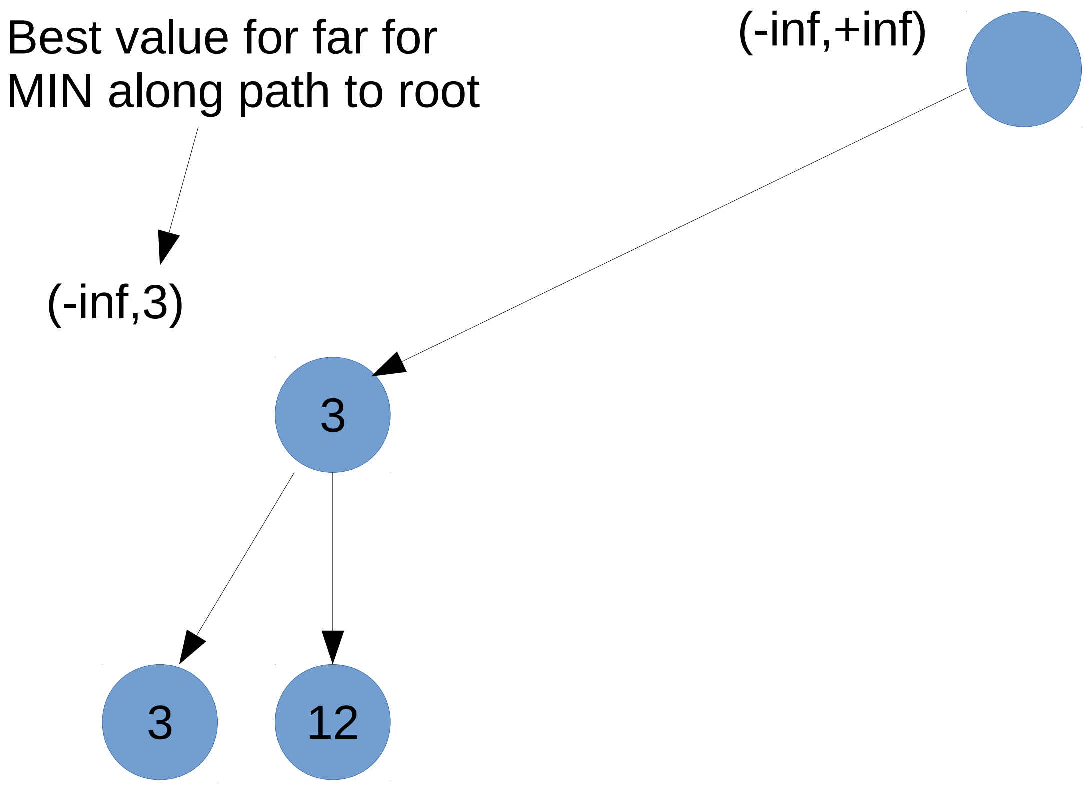
# Alpha/Beta pruning



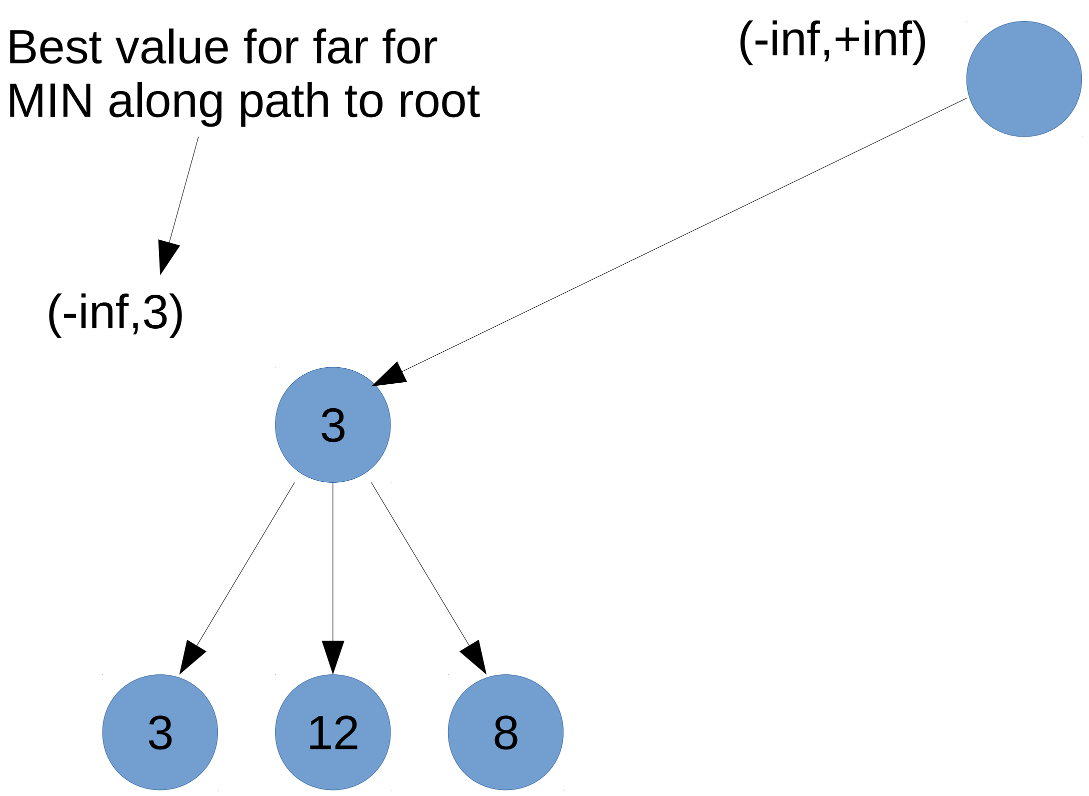
# Alpha/Beta pruning



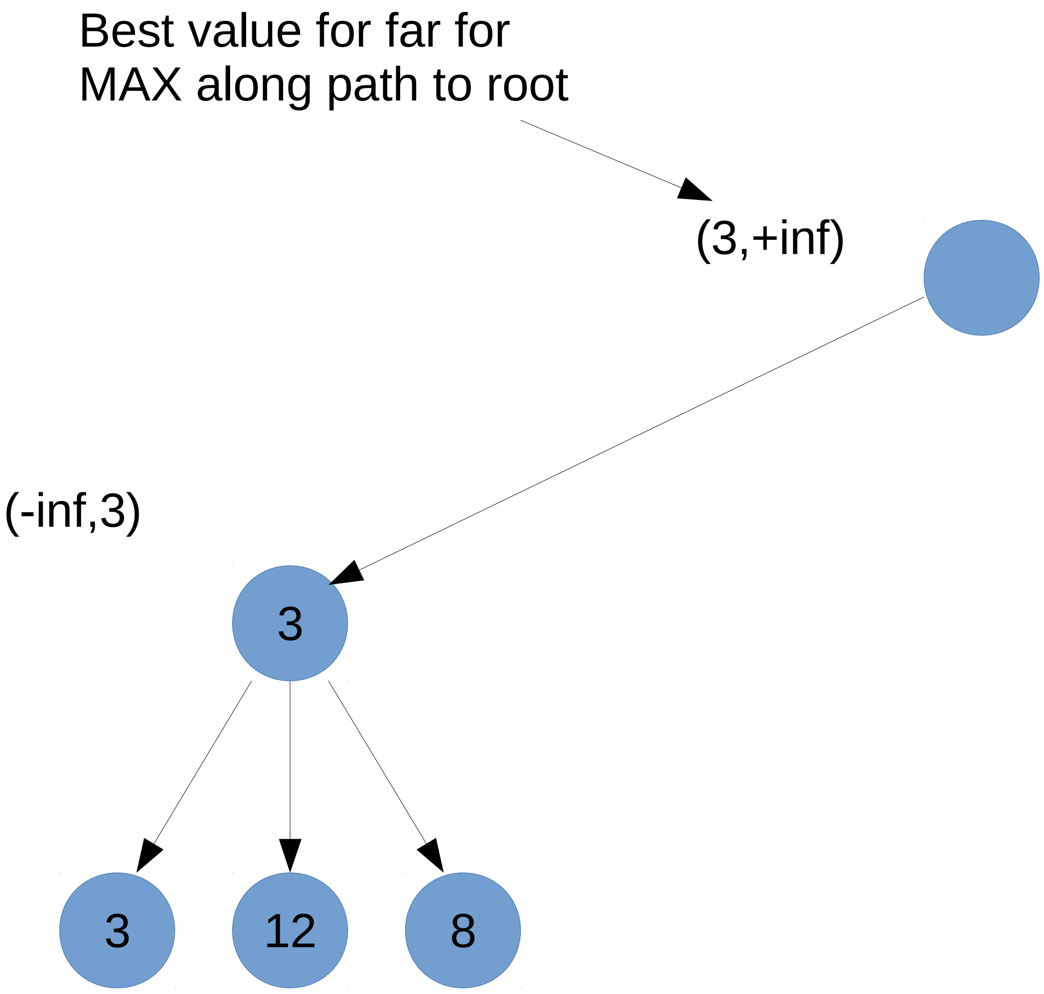
# Alpha/Beta pruning



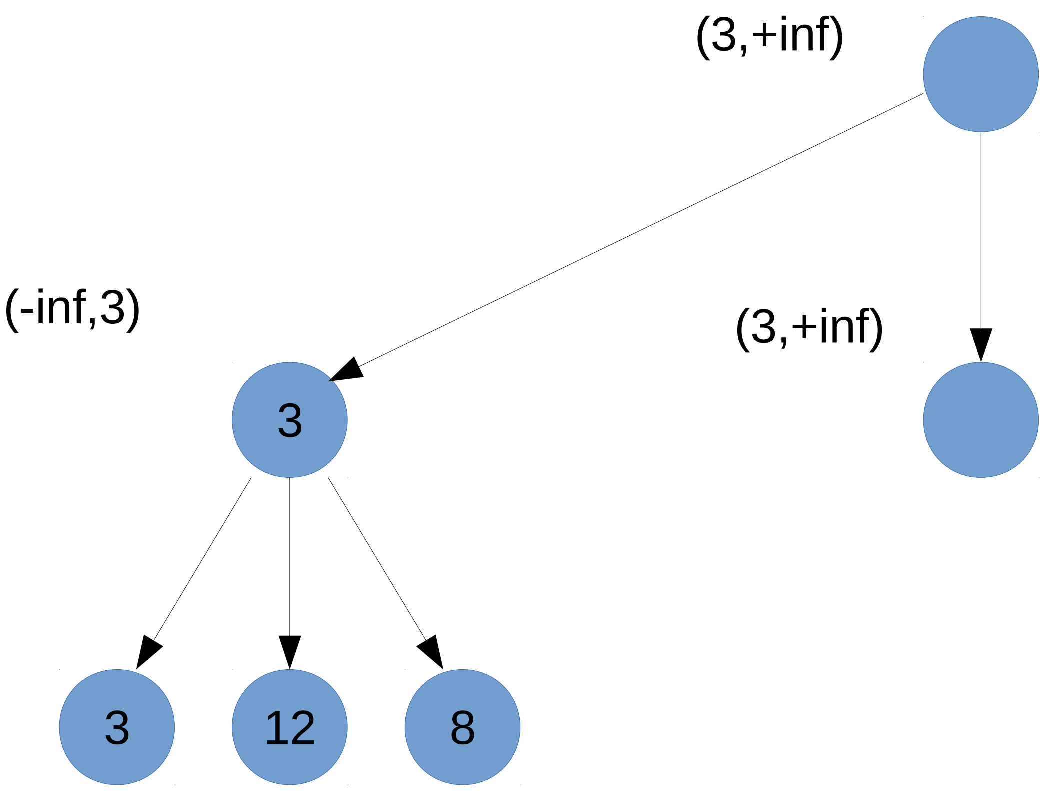
# Alpha/Beta pruning



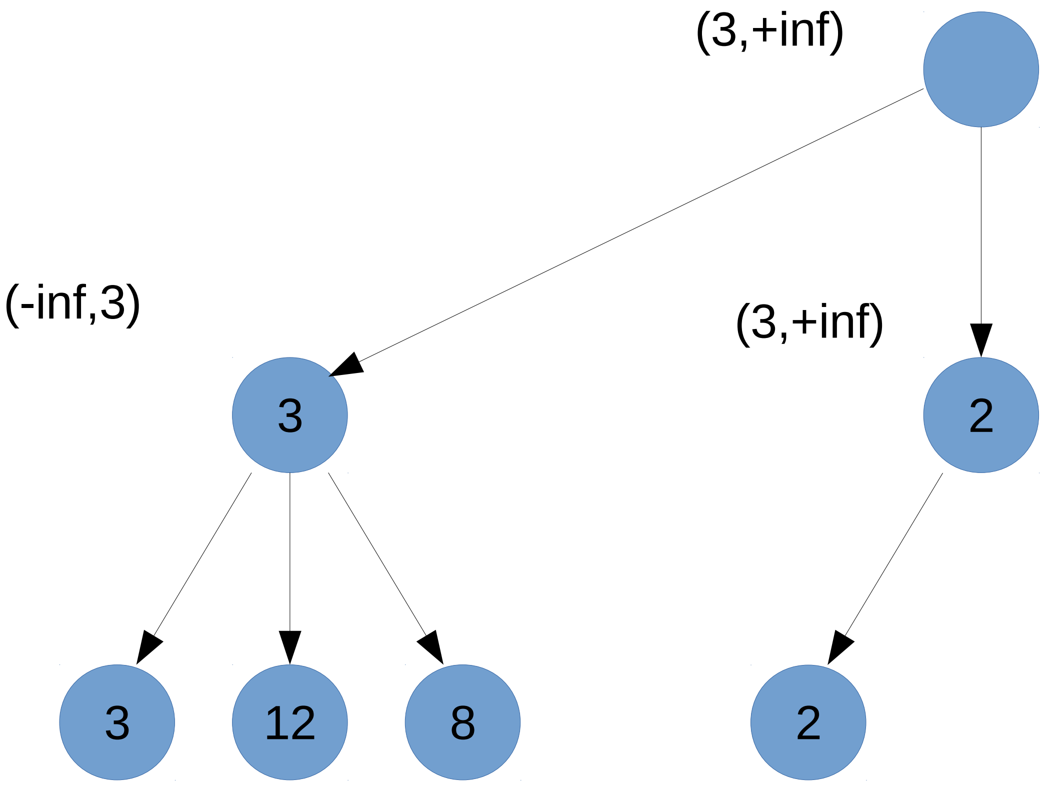
# Alpha/Beta pruning



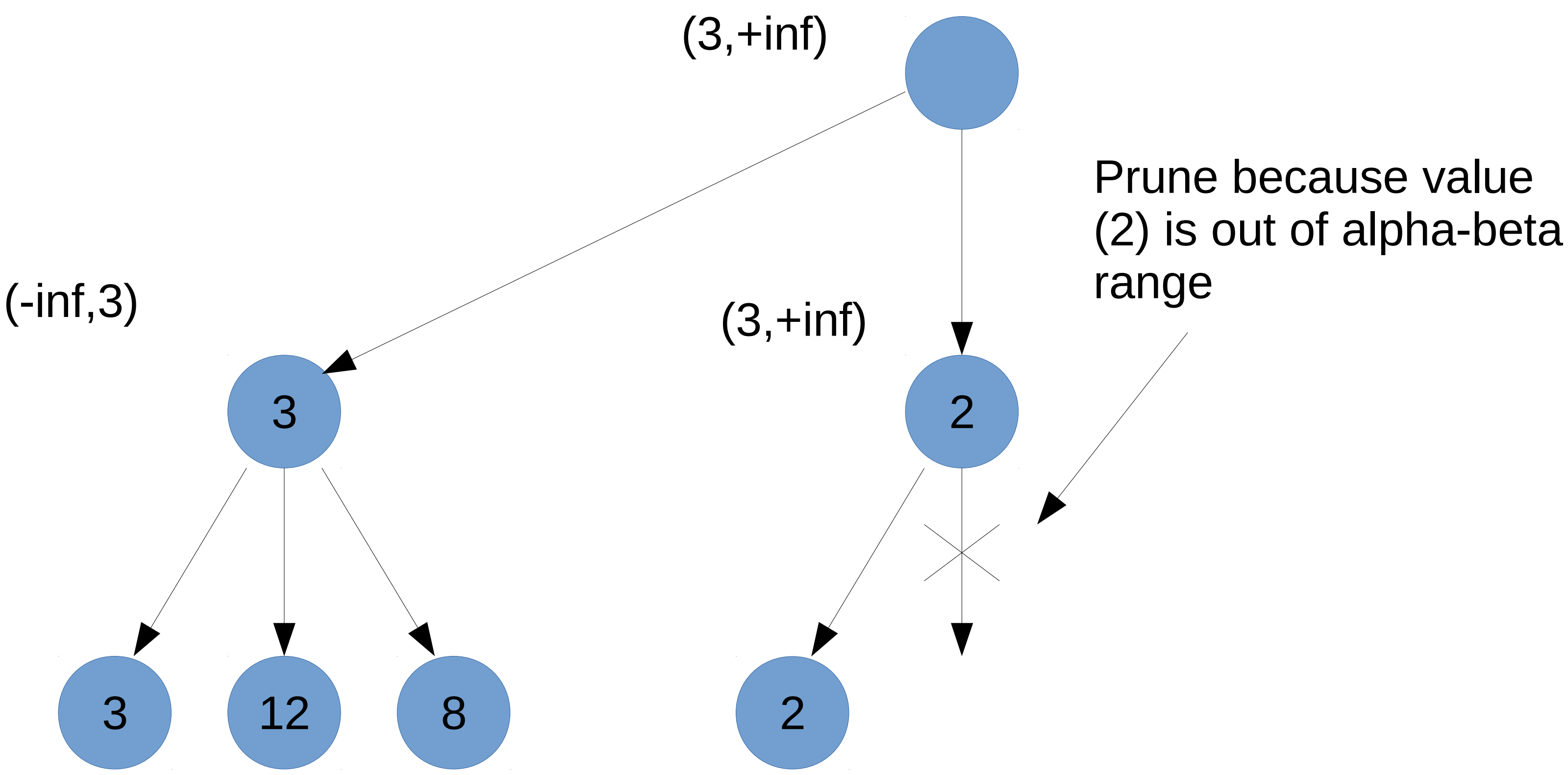
# Alpha/Beta pruning



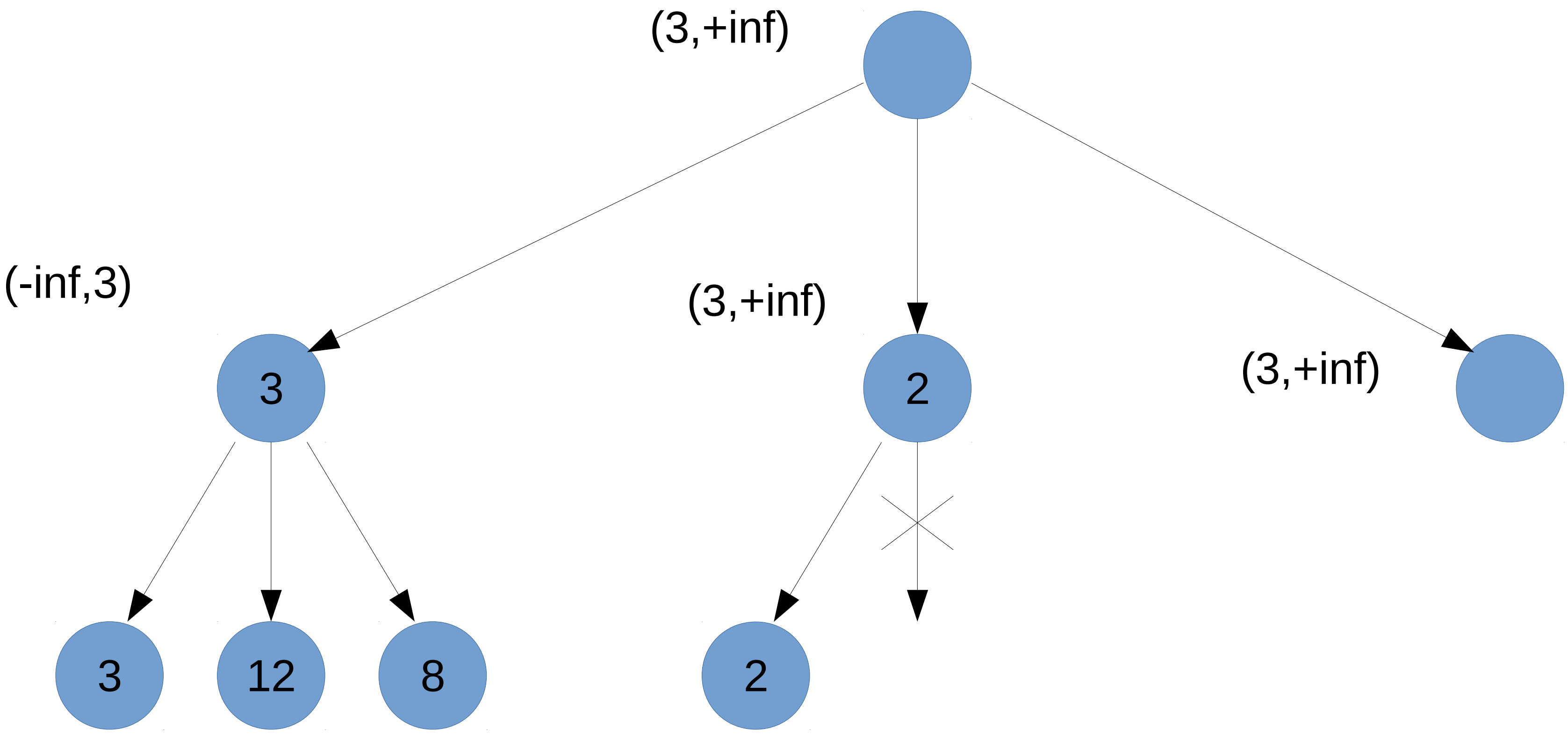
# Alpha/Beta pruning



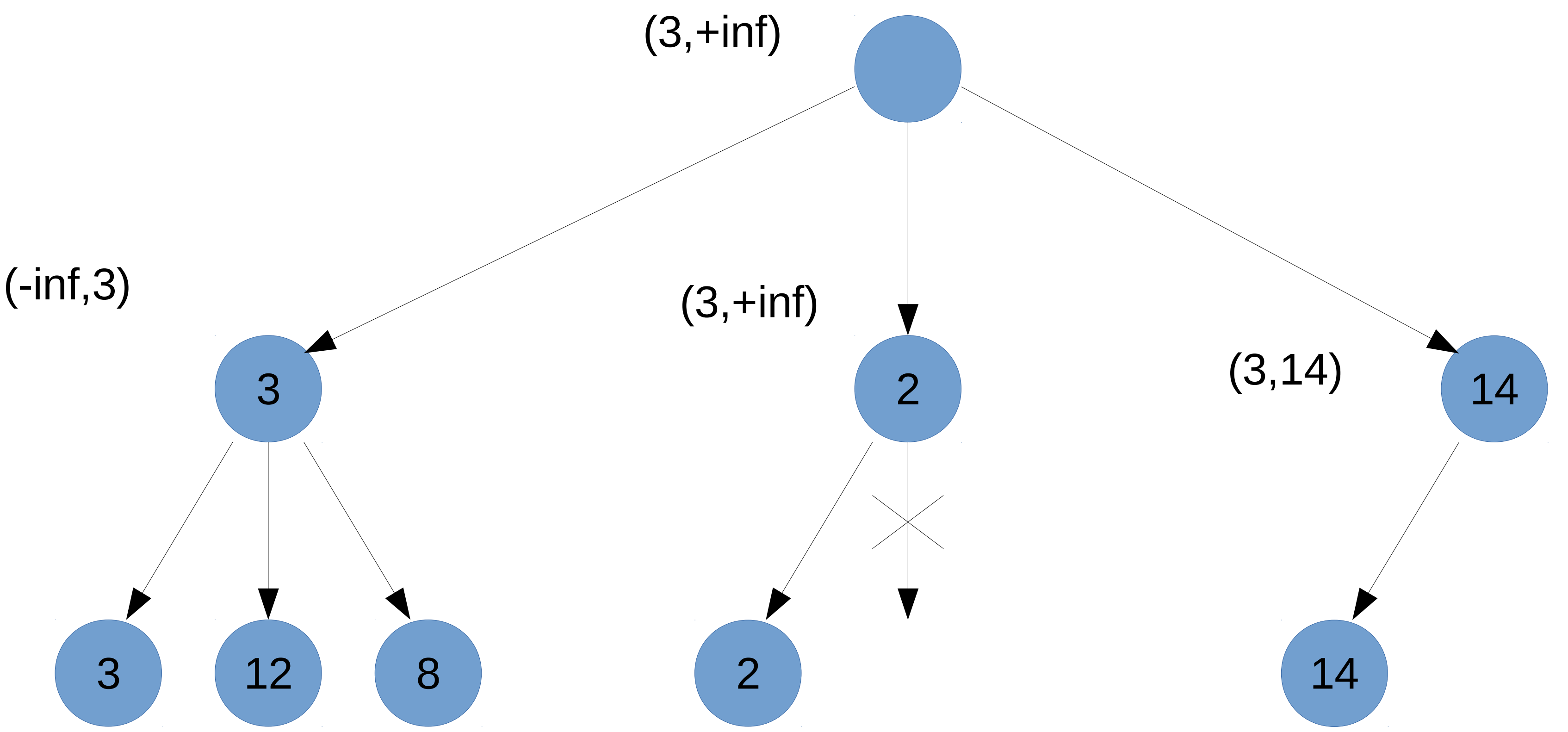
# Alpha/Beta pruning



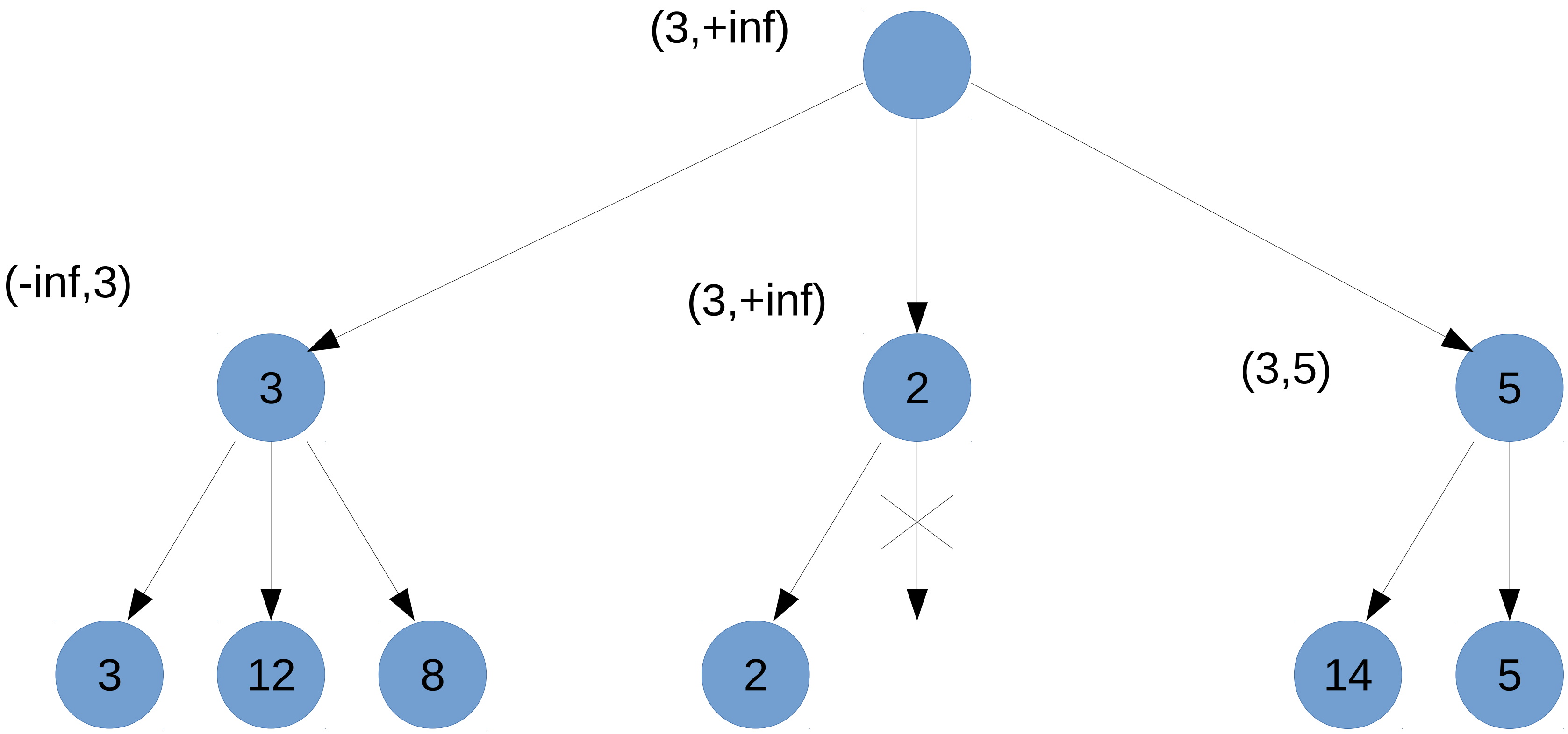
# Alpha/Beta pruning



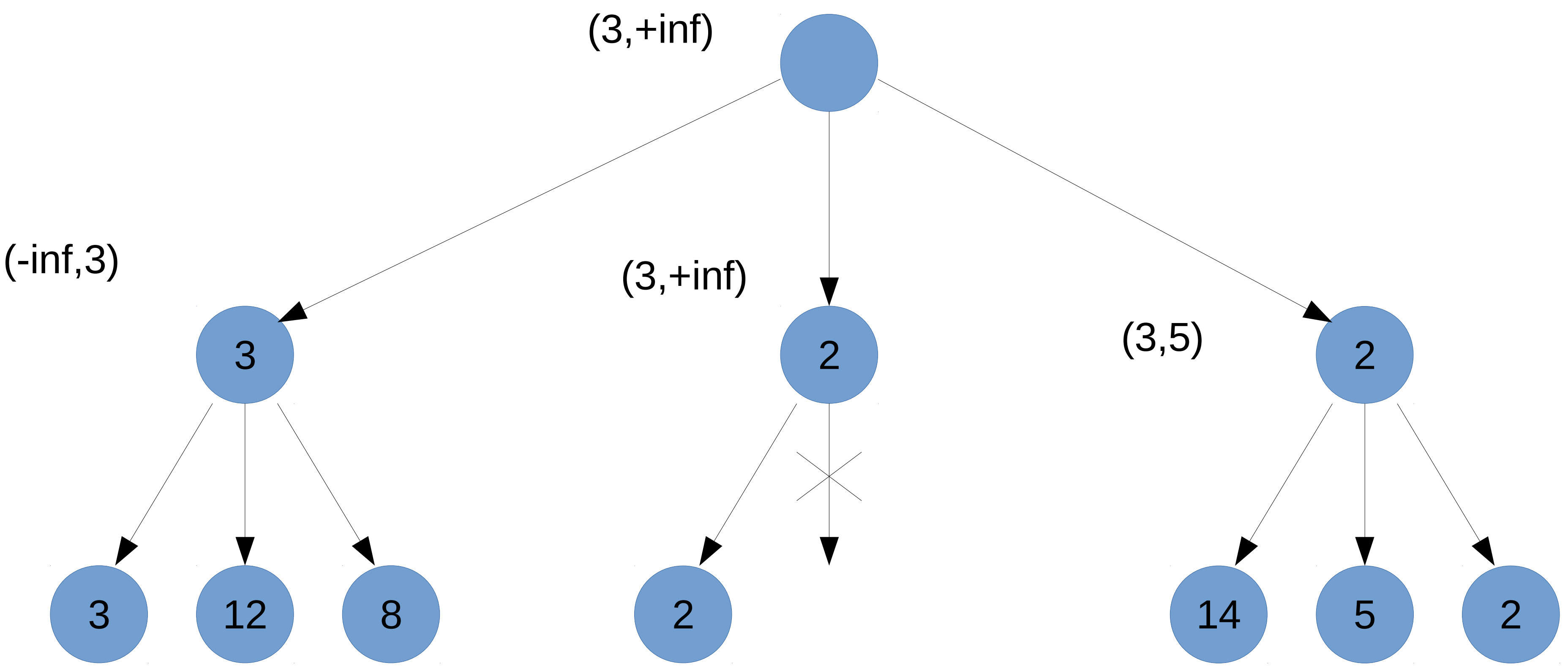
# Alpha/Beta pruning



# Alpha/Beta pruning



# Alpha/Beta pruning



# Alpha/Beta properties

Is it complete?

# Alpha/Beta properties

Is it complete?

How much does alpha/beta help relative to minimax?

Minimax time complexity = $O(b^m)$

Alpha/beta time complexity >= $O(b^{\frac{m}{2}})$

– the improvement w/ alpha/beta depends upon move ordering...

# Alpha/Beta properties

Is it complete?

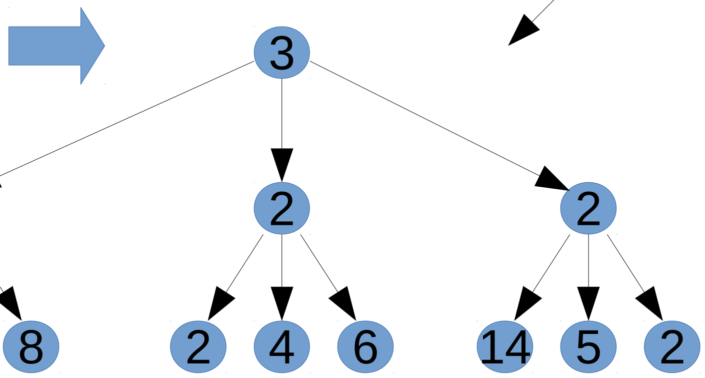
How much does alpha/beta help relative to minimax?

Minimax time complexity = $O(b^m)$

Alpha/beta time complexity >= $O(b^{\frac{m}{2}})$

– the improvement w/ alpha/beta depends upon move ordering...

The order in which we expand a node.



# Alpha/Beta properties

Is it complete?

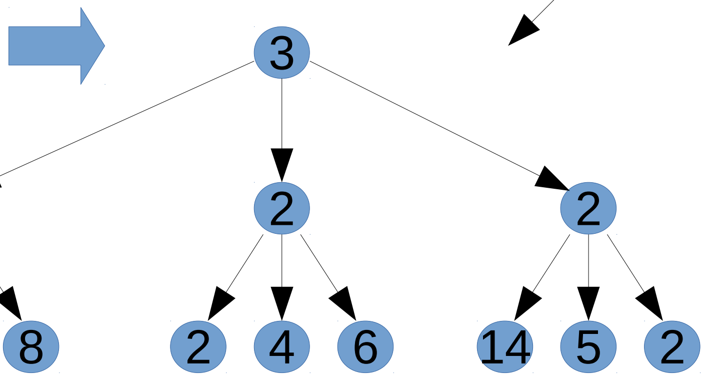
How much does alpha/beta help relative to minimax?

Minimax time complexity = $O(b^m)$

Alpha/beta time complexity >= $O(b^{\frac{m}{2}})$

– the improvement w/ alpha/beta depends upon move ordering...

The order in which we expand a node.




How to choose move ordering? Use IDS.
– on each iteration of IDS, use prior run to inform ordering of next node expansions.