GAMES

CMSC 170 – Introduction to Artificial Intelligence

2nd Semester AY 2013-2014

CNM Peralta

Every game has different properties. For simplicity, we will deal with games with 1-2 players that are deterministic.

How do we describe games?

1.

We define S as the set of states; there exists s_0, the start state and $s_0 \in S$.

2.

The set P is the set of players.
For now, we can have:
$P = \{P_1, P_2\}$ (2-player) or
$P = \{P_1\}$ (1-player).

3.

$Result(s, a) \rightarrow s'$

Gives s' as the result of doing action a at state s.

4.

Actions(s, p)

Gives the set of actions possible at state s by player p.

5.

Terminal(s)

Returns true if s is terminal, false otherwise.

6.

Utility(s, p)

Returns the utility of a state s to a player p.

Utility

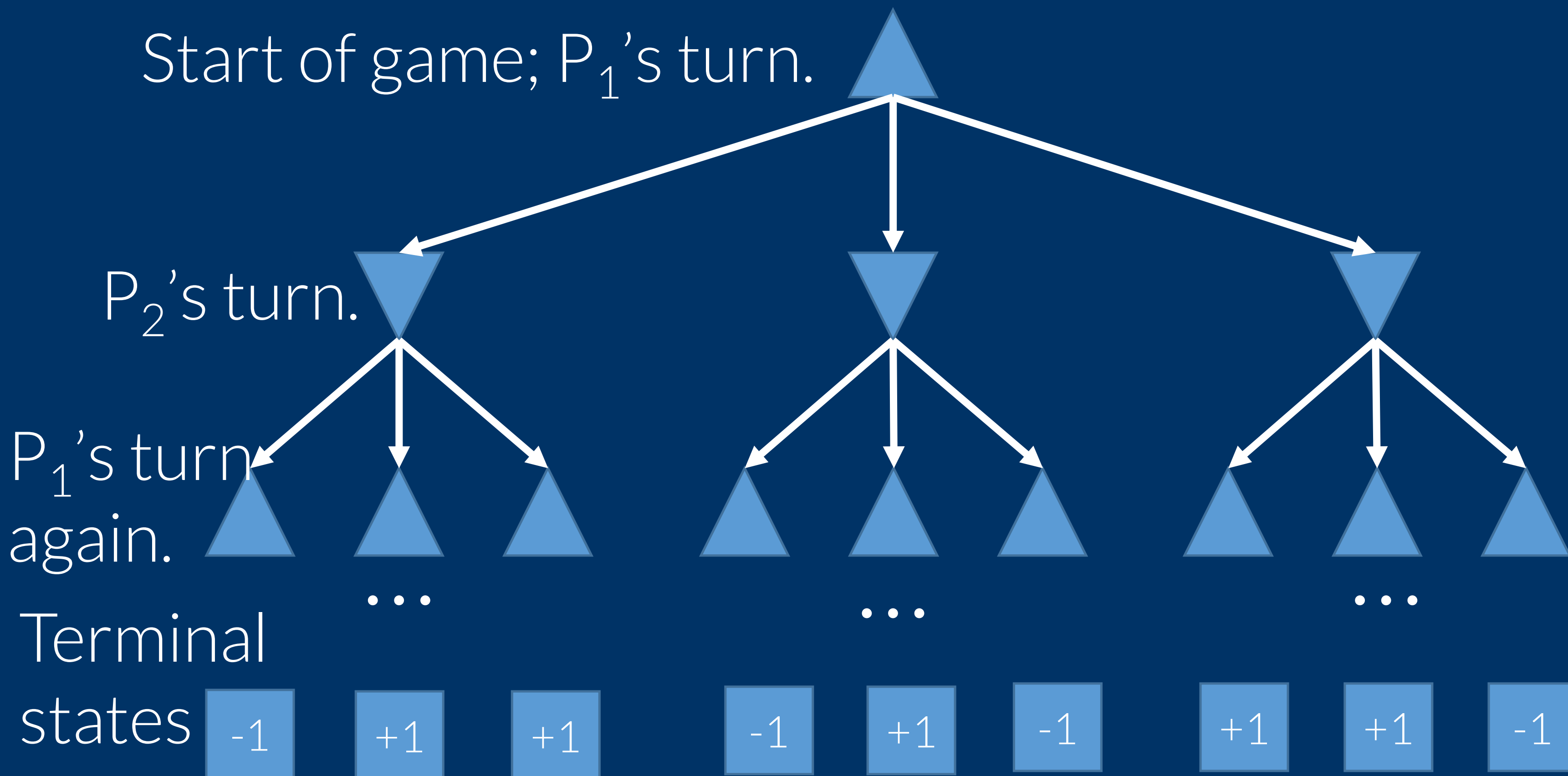
expresses the value of current game state to the player; usually expressed as +/- numbers or 0's and 1's.

For the subsequent discussion, we will consider 2-player, deterministic, zero-sum games.

Zero-Sum Games

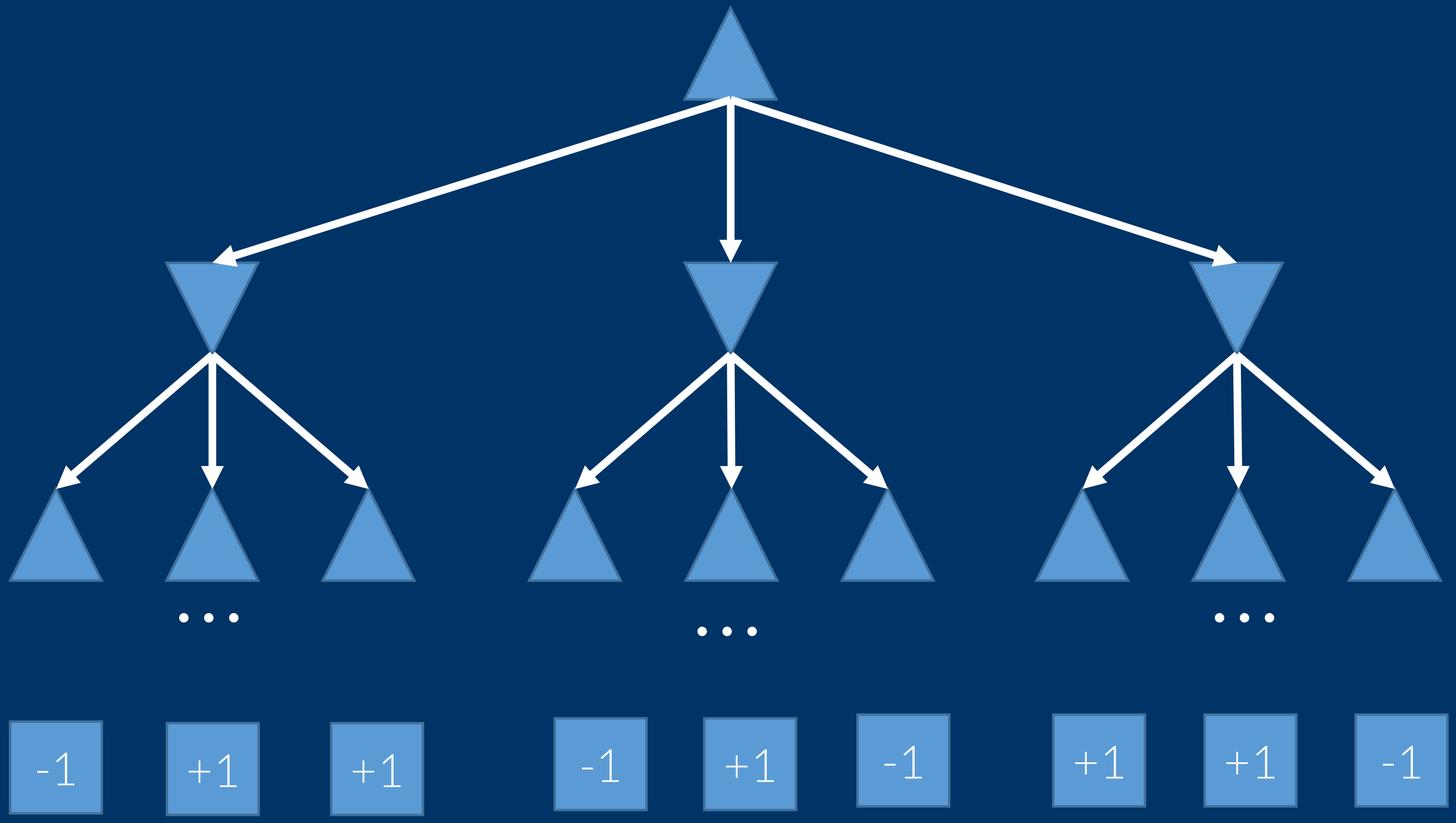
are games where the overall sum of the utilities for both players is 0; for example, if the utility for a win is 1 and a loss is -1.

Such games can be expressed as game trees as follows:



Each node has a utility value. P_1 attempts to maximize the value; P_2 attempts to minimize the value, resulting in an adversarial environment.

How do we find the utility of each inner (non-terminal; non-leaf) node?



We start using the utilities of terminal states and work our way up by following the min-max behavior of P_1 and P_2.

HOW?

```
value(s)
    if s is □: Utility(s)
    if s is △: maxValue(s)
    if s is ▽: minValue(s)
```

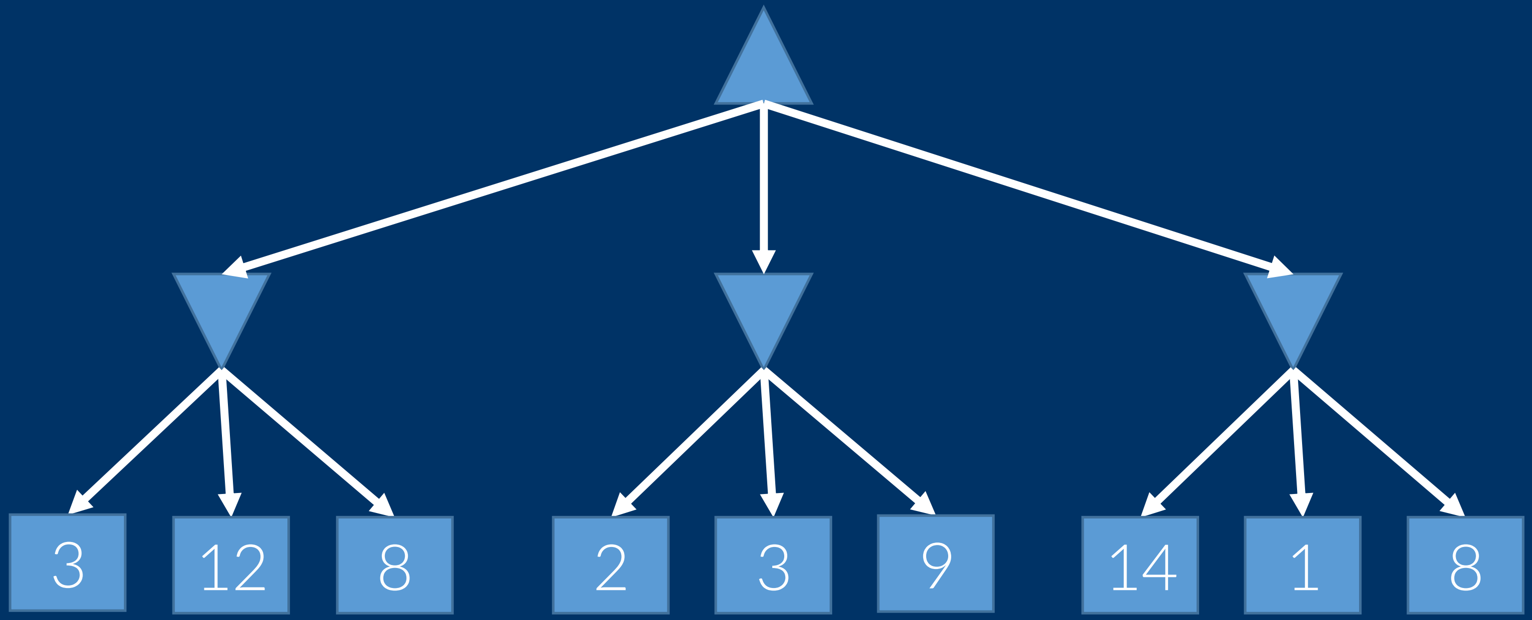
HOW?

```
maxValue(s)
    m = -∞
    for a, s' in successors(s)
        v = value(s')
        m = max(m, v)
    return m
```

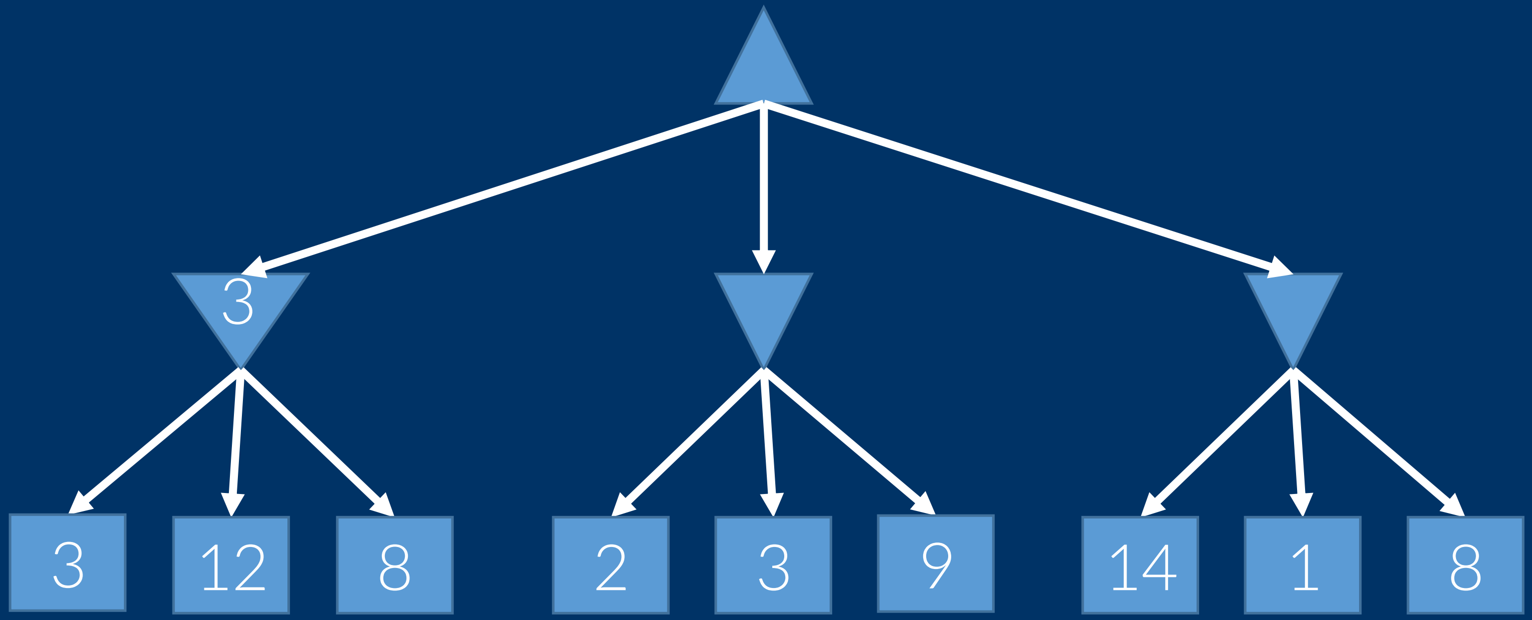
HOW?

```
minValue(s)
    m = +∞
    for a, s’ in successors(s)
        v = value(s’)
        m = min(m, v)
    return m
```

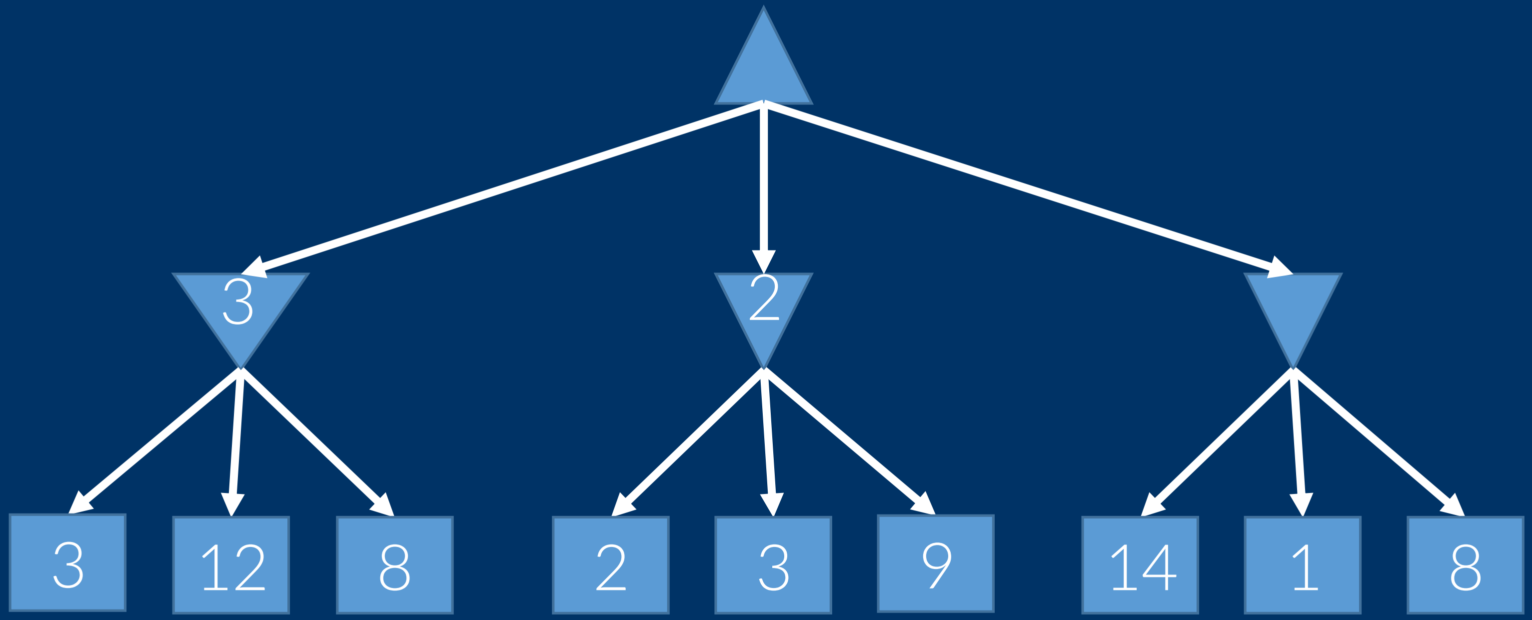
EXAMPLE



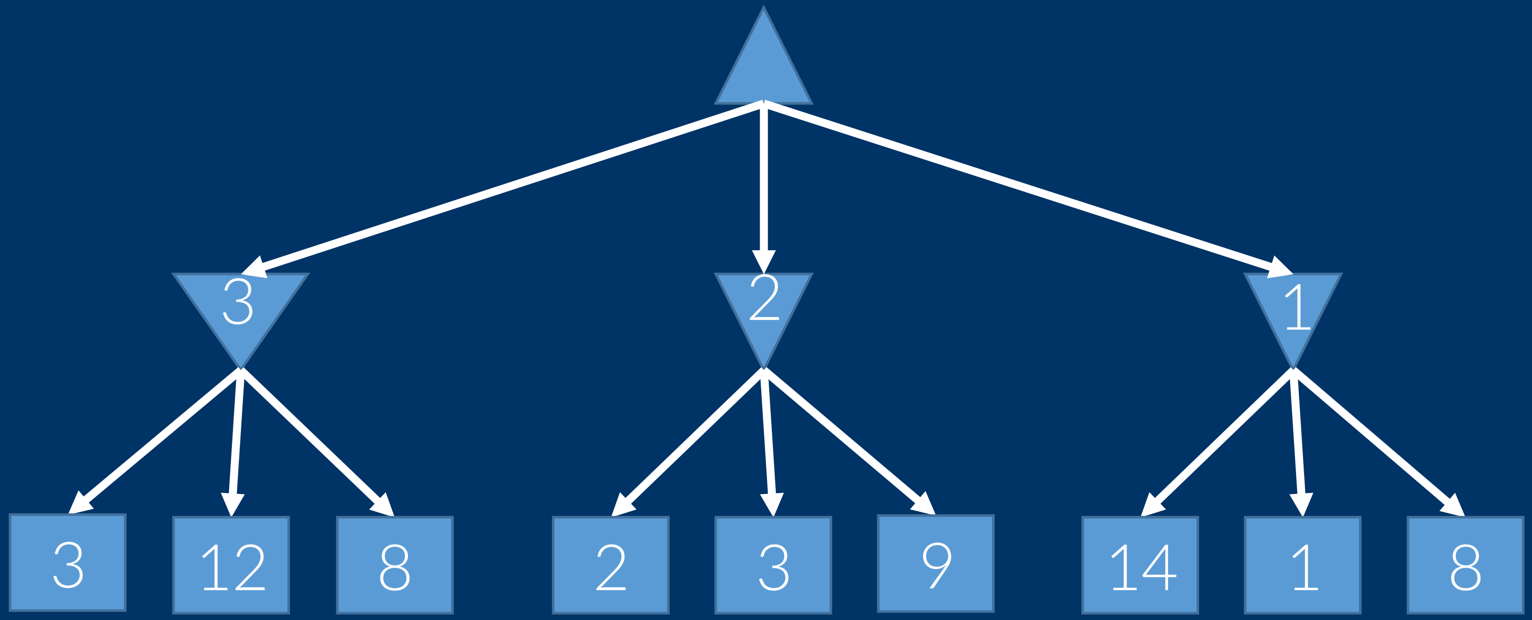
EXAMPLE



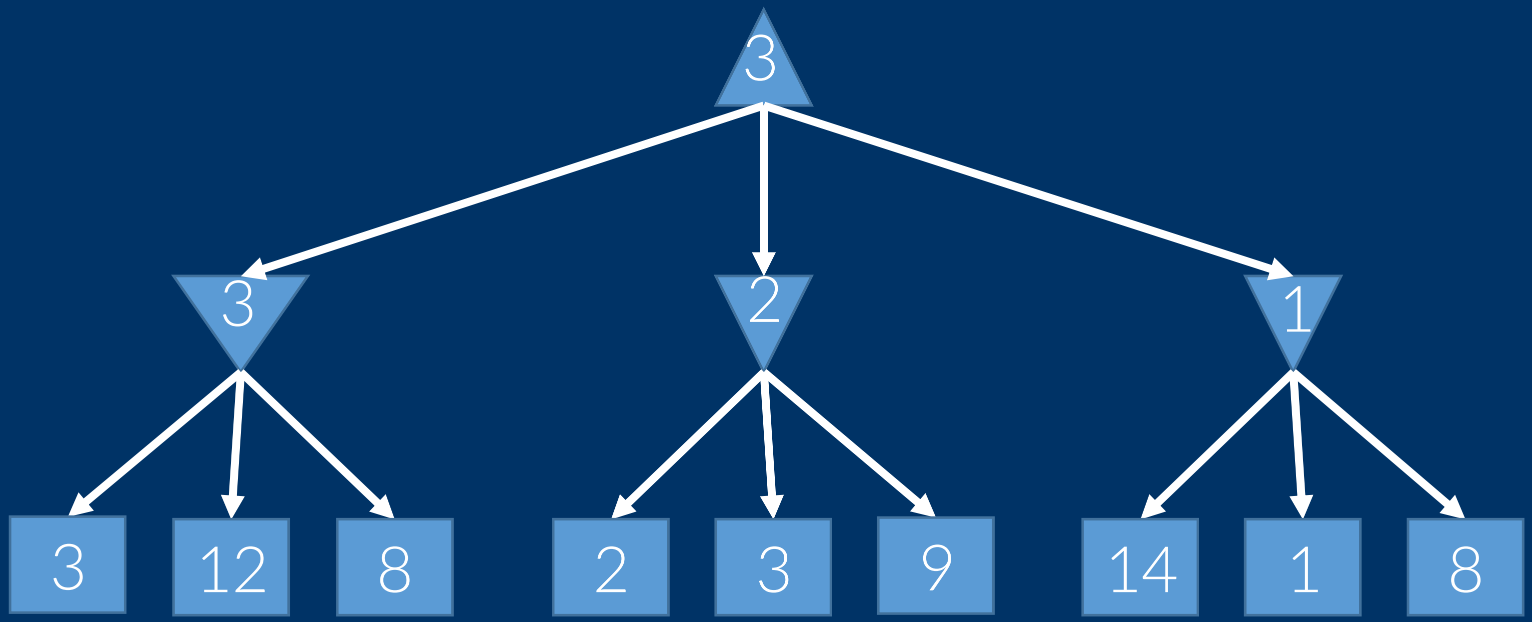
EXAMPLE



EXAMPLE



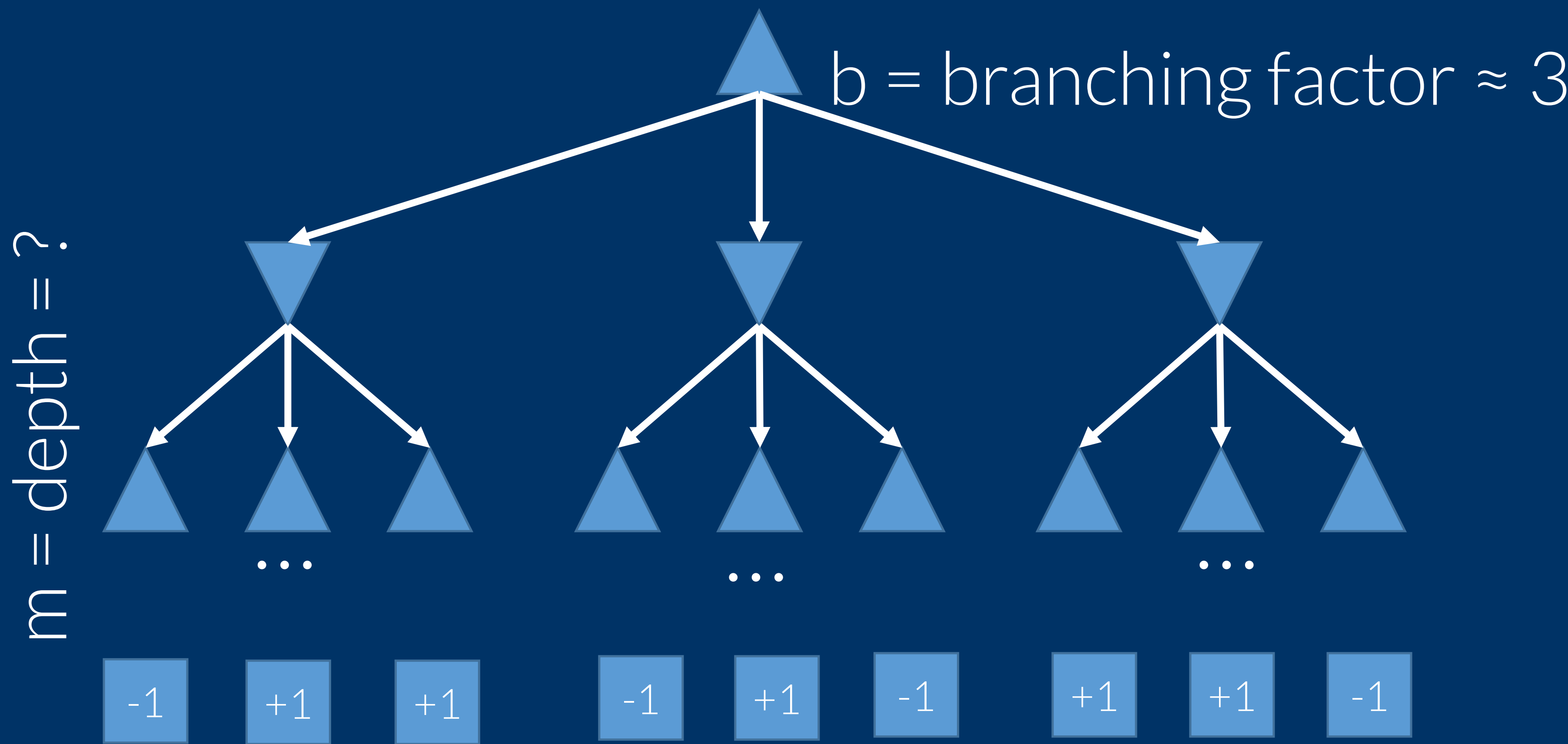
EXAMPLE



BUT!!!

The method has a **time complexity** of

$$O(b^m).$$



b = branching factor ≈ 3
m = depth = ?
…
…
…
-1
+1
+1
-1
+1
-1
+1
+1
-1

Branching Factor

approximate number of choices (successors) a player has at each node.

Depth

total number of turns for P_1 and P_2 to reach a terminal state.

EXAMPLE

A chess game with ≈30 possible moves at each turn and takes 40 total turns to finish will require a search tree to have 30^{40} nodes to explore.

That's

$$1.2157665 \times 10^{59}.$$

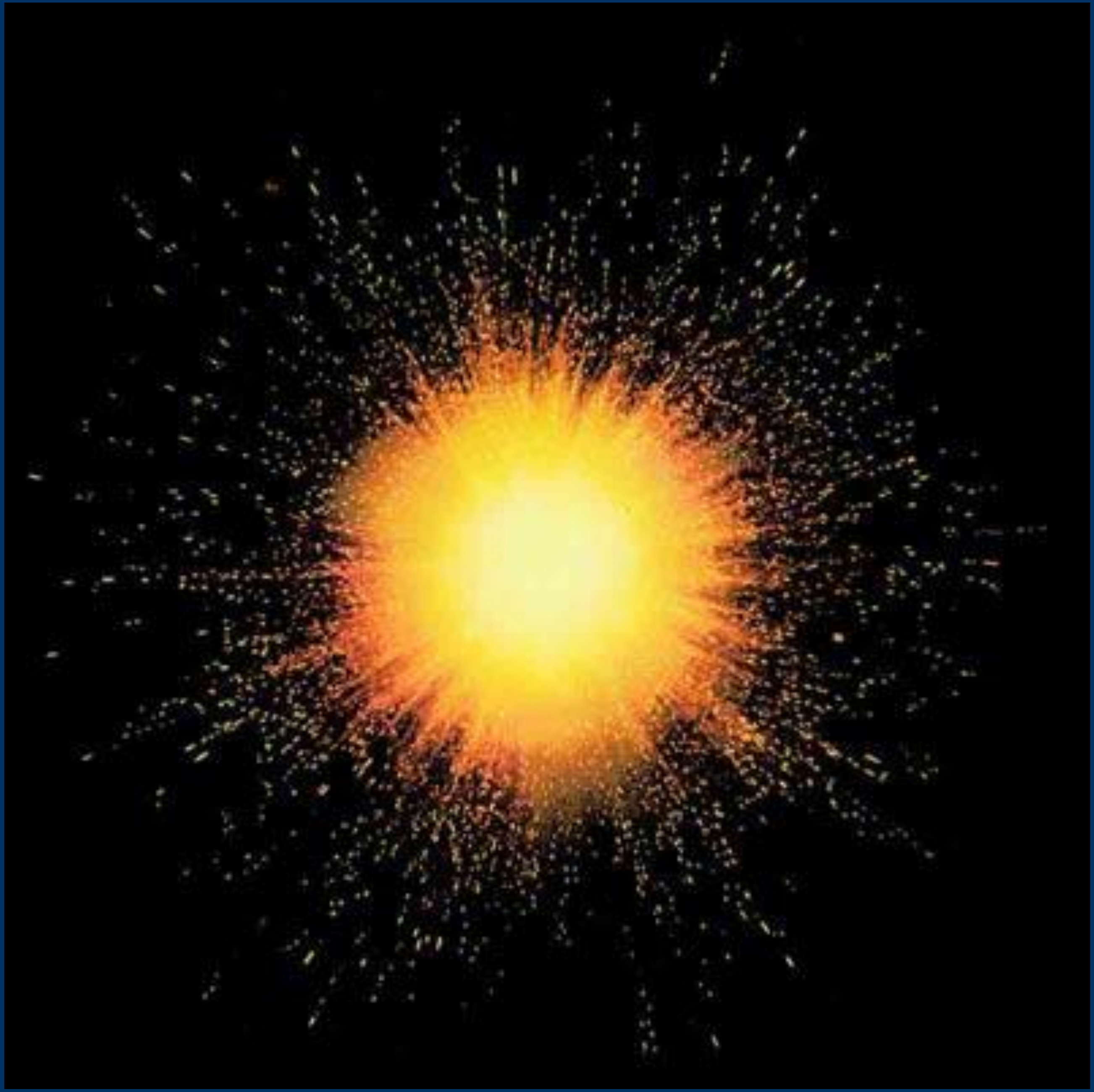
Even if we were able to execute 1 Billion operations per second **(which we can't)**, we would need

1.2157665×10^{50} seconds.

That's

$3.85261384458042593 \times 10^{42}$

YEARS.

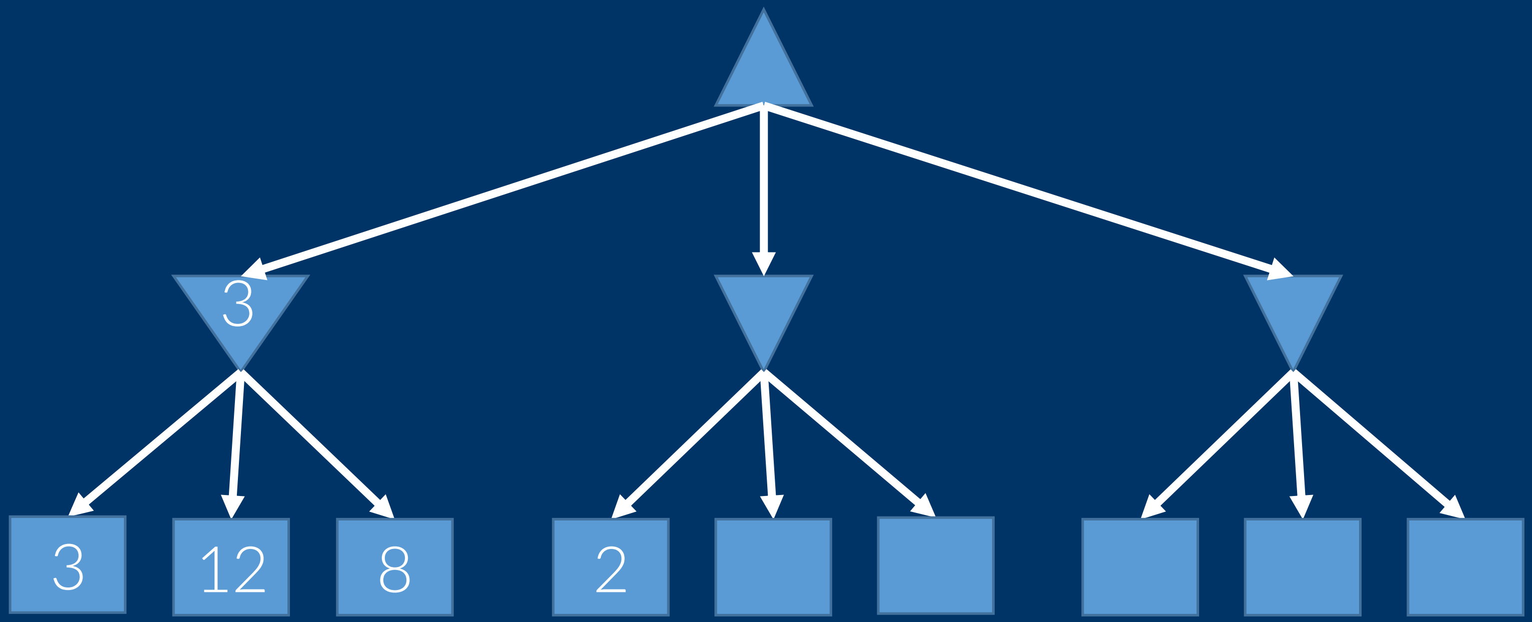


We need to reduce the complexity of the problem.

1.

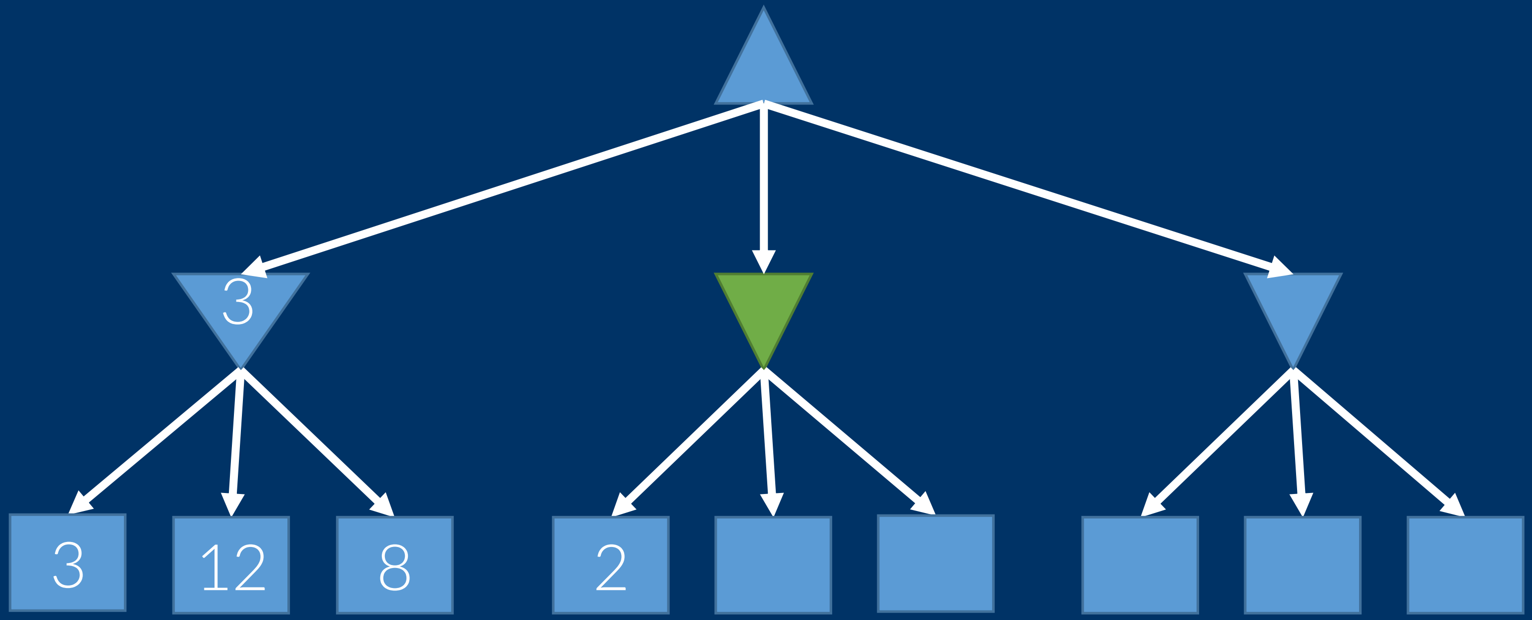
We can reduce the branching factor, b, by pruning branches that do not need to be explored anymore.

EXAMPLE



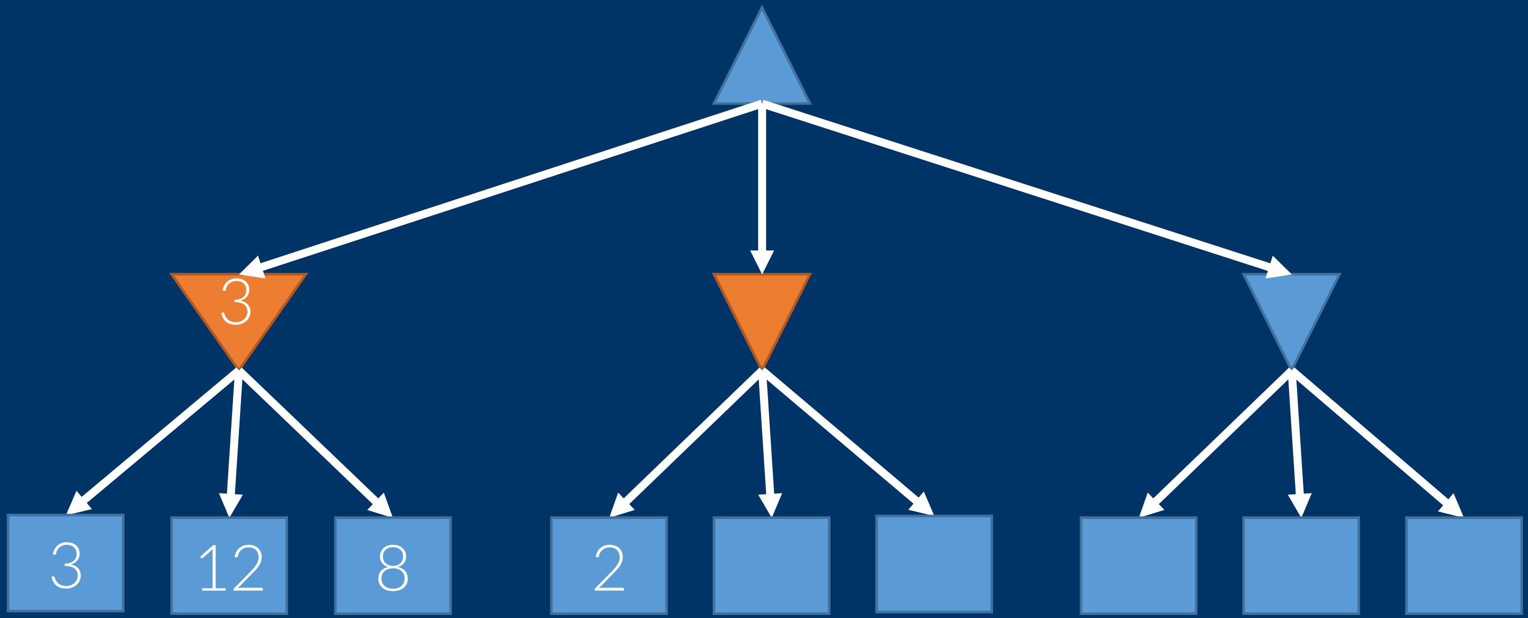
What if we have only explored 2?

EXAMPLE



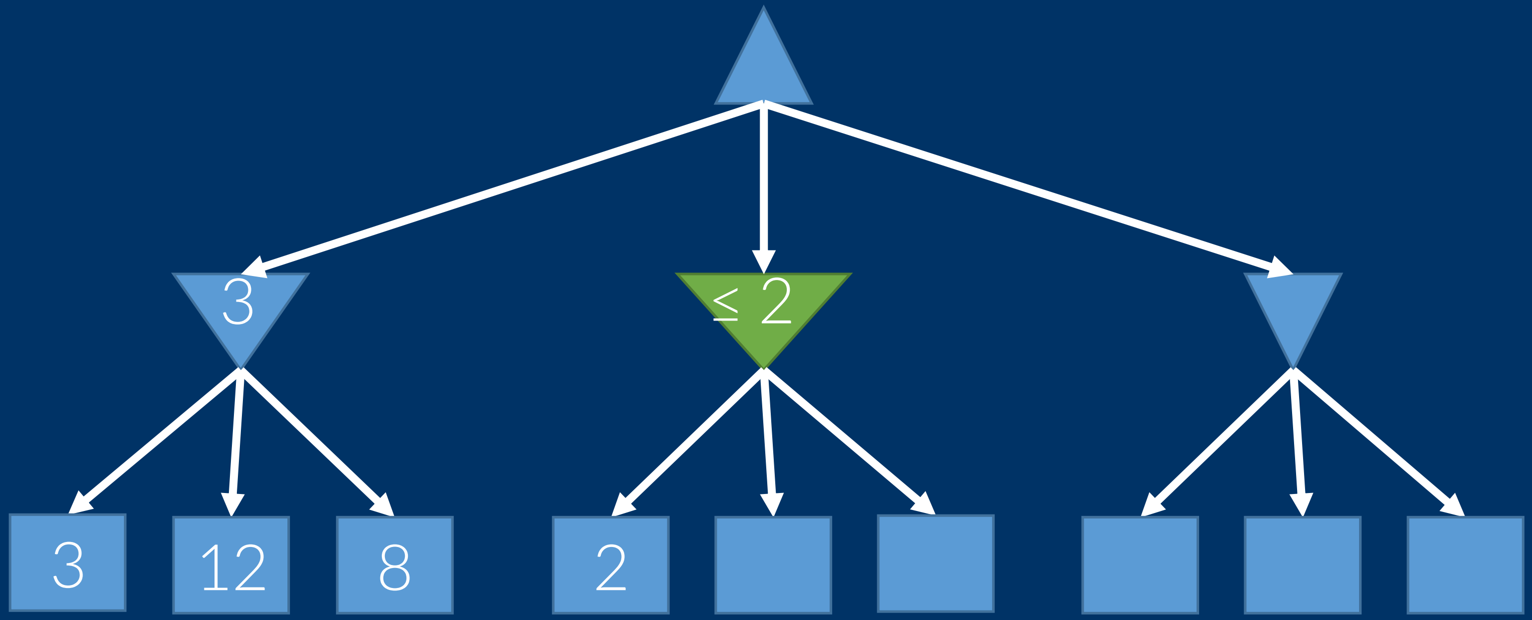
If the two other siblings are > 2, then 2 will be chosen since the node choosing is a min node.

EXAMPLE



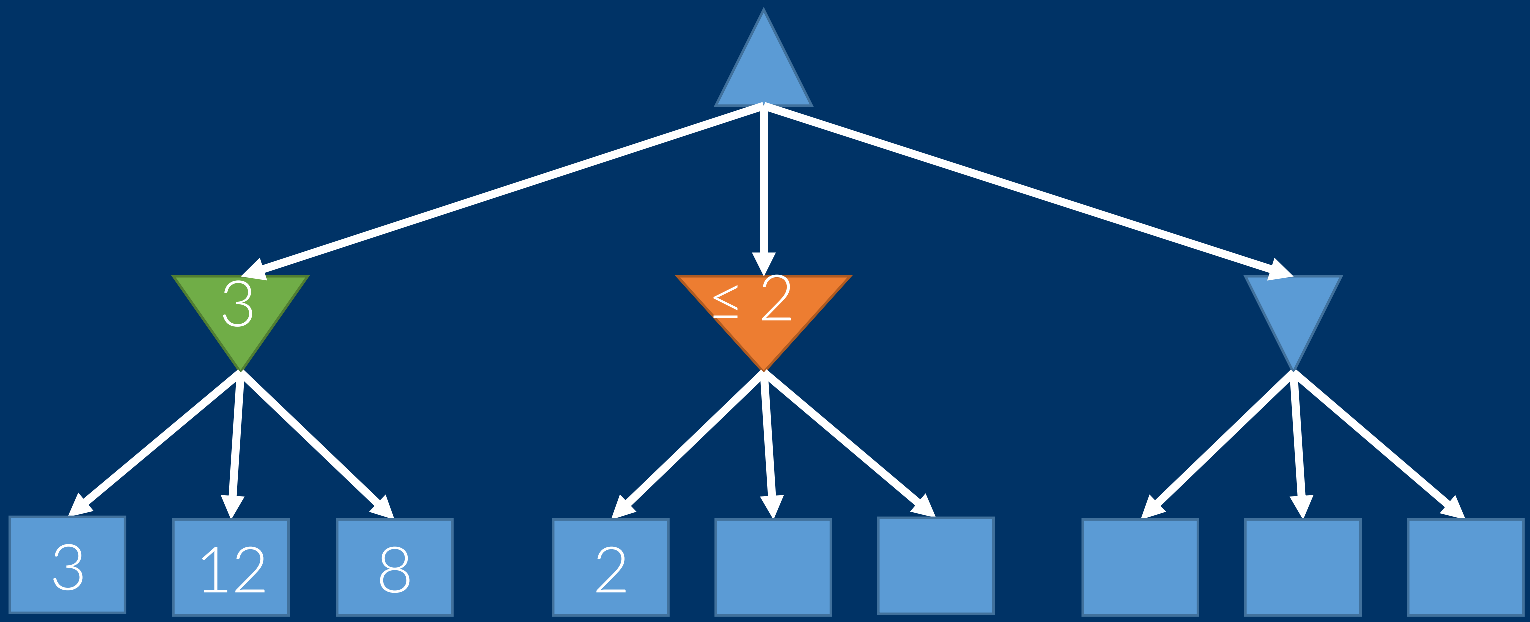
On the other hand, 2 is already smaller than 3 (on the left branch).

EXAMPLE



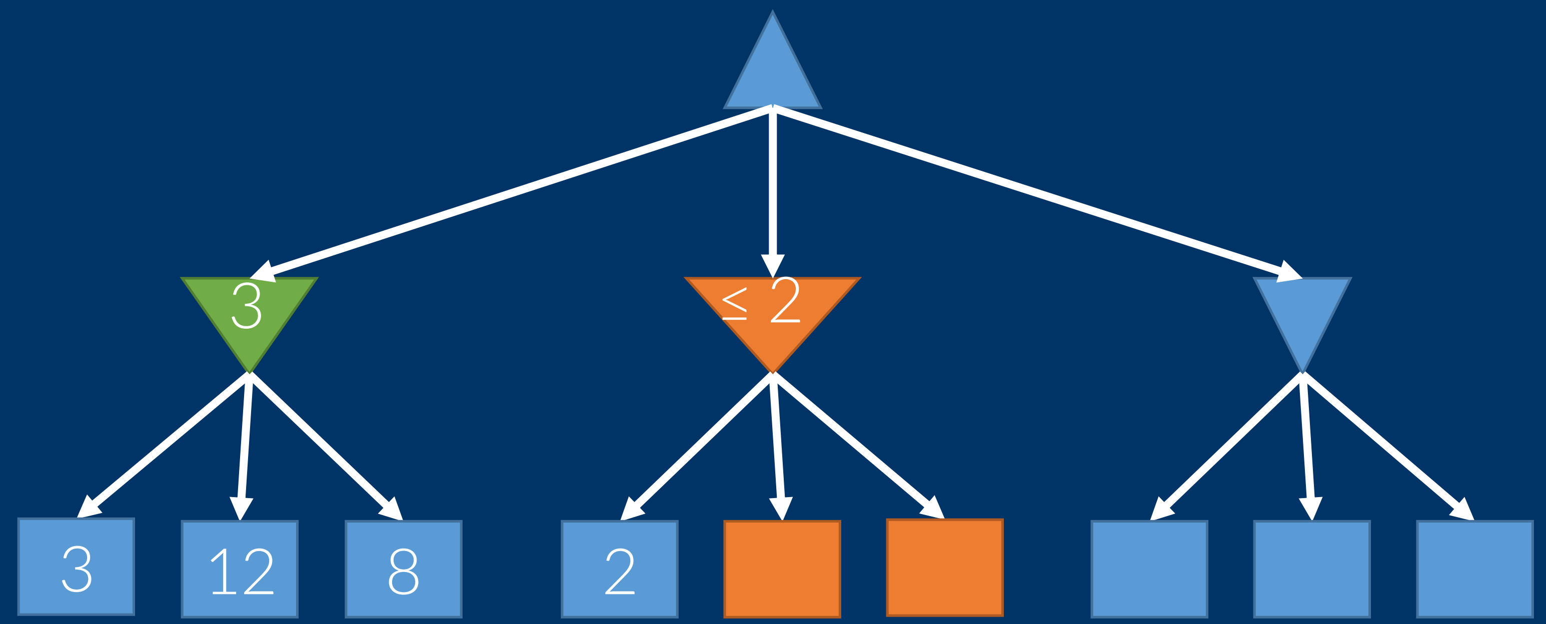
We can thus generalize that the min node will have a value ≤ 2.

EXAMPLE



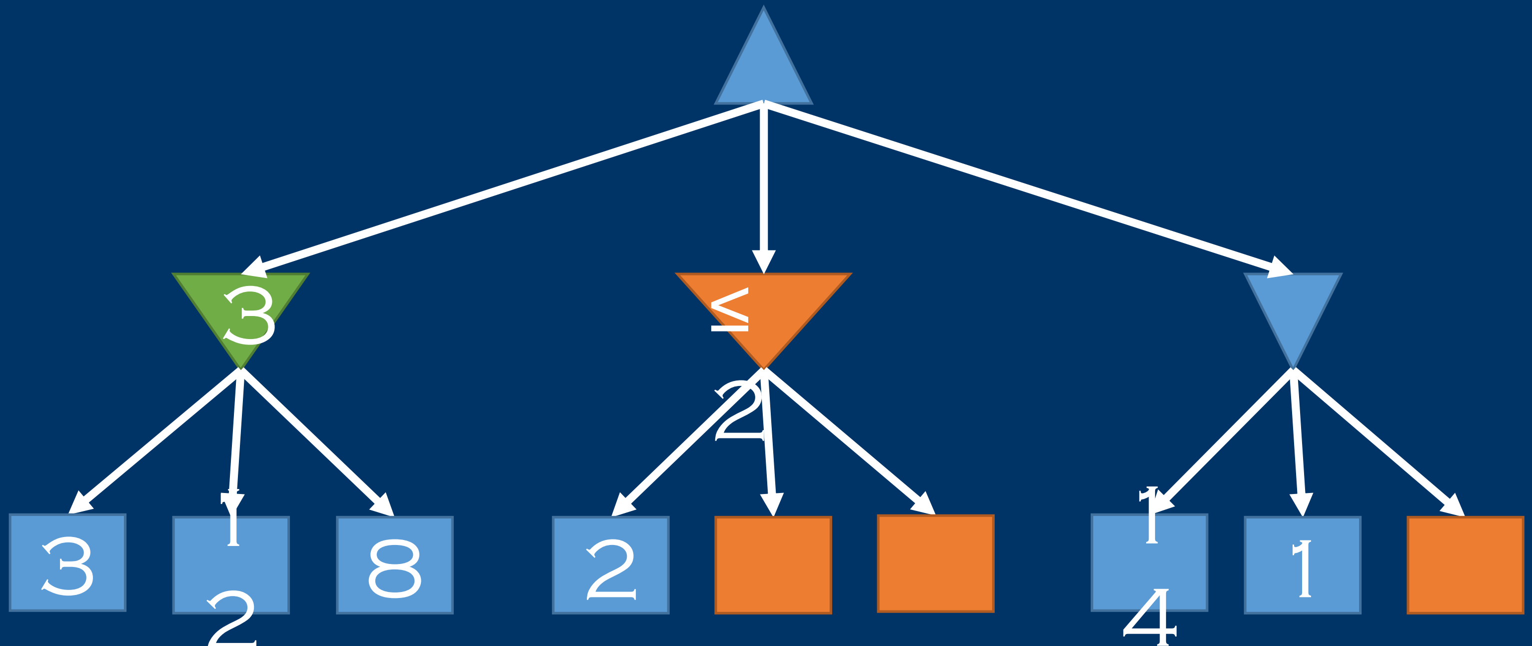
Thus, if the shallower max node were to choose, it would never consider 2, much less values smaller than 2.

EXAMPLE



We no longer need to expand 2’s siblings.

Example



In the case of the third branch, the 14 and the 1 will be explored, but the last child need not be explored.

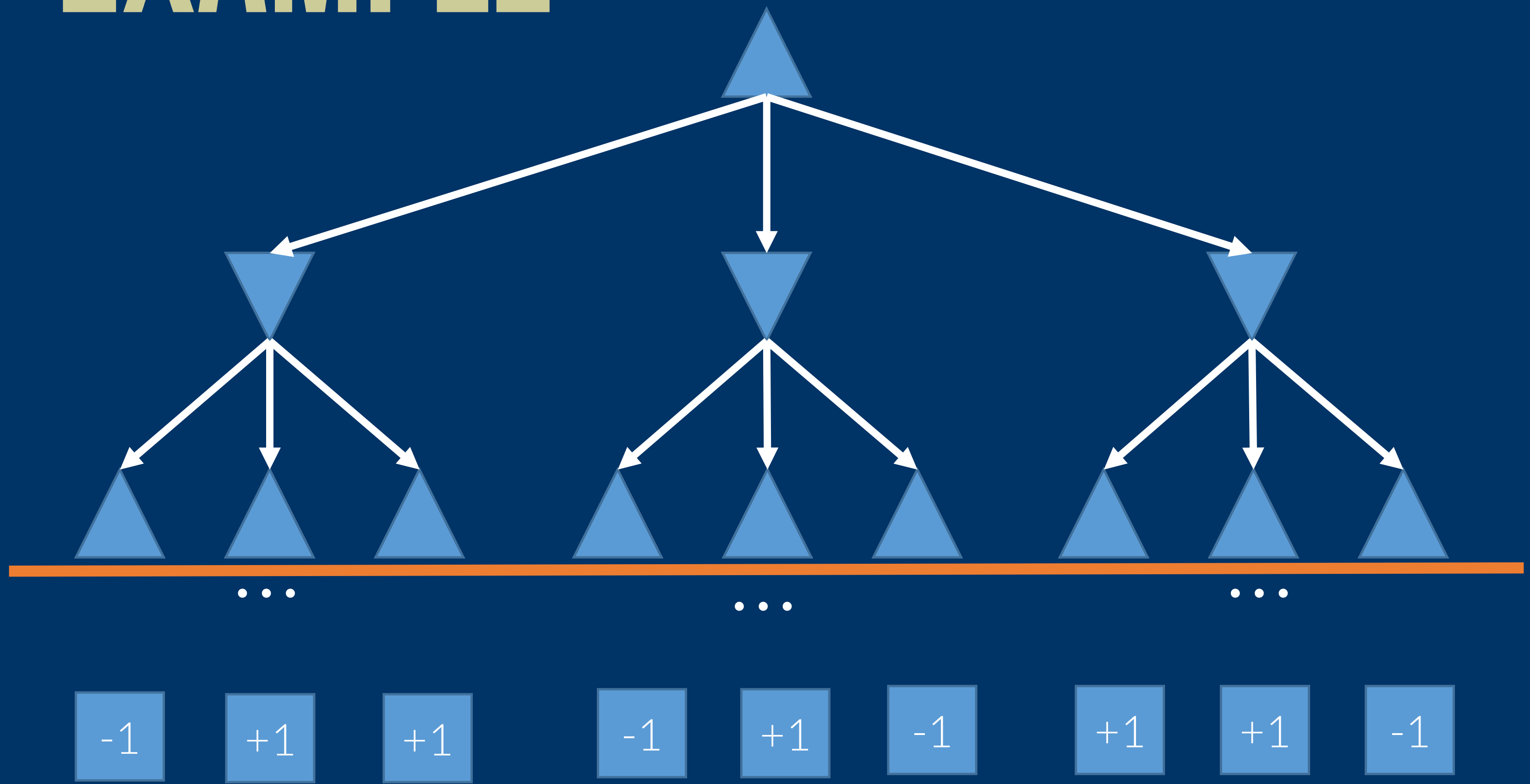
EXAMPLE

Although this might not seem effective since tree is only of depth 3, its effect will be profound if the branches eliminated actually lead to big subtrees.

2.

We can reduce the depth, m, by cutting off the search at a certain depth and using an estimation function to estimate the utility of the nodes at that depth.

EXAMPLE



Cut off at depth 3; estimate utility of nodes at depth 3 and treat them as if they were terminal nodes.

We thus modify the value, maxValue and minValue functions we defined earlier.

HOW?

```
value(s, currDepth, α, β)
    if CUTOFF(s, depth): Eval(s)
    if s is □: Utility(s)
    if s is △: maxValue(s,
                 currDepth, α, β)
    if s is ▽: minValue(s currDepth,
                 α, β)
```

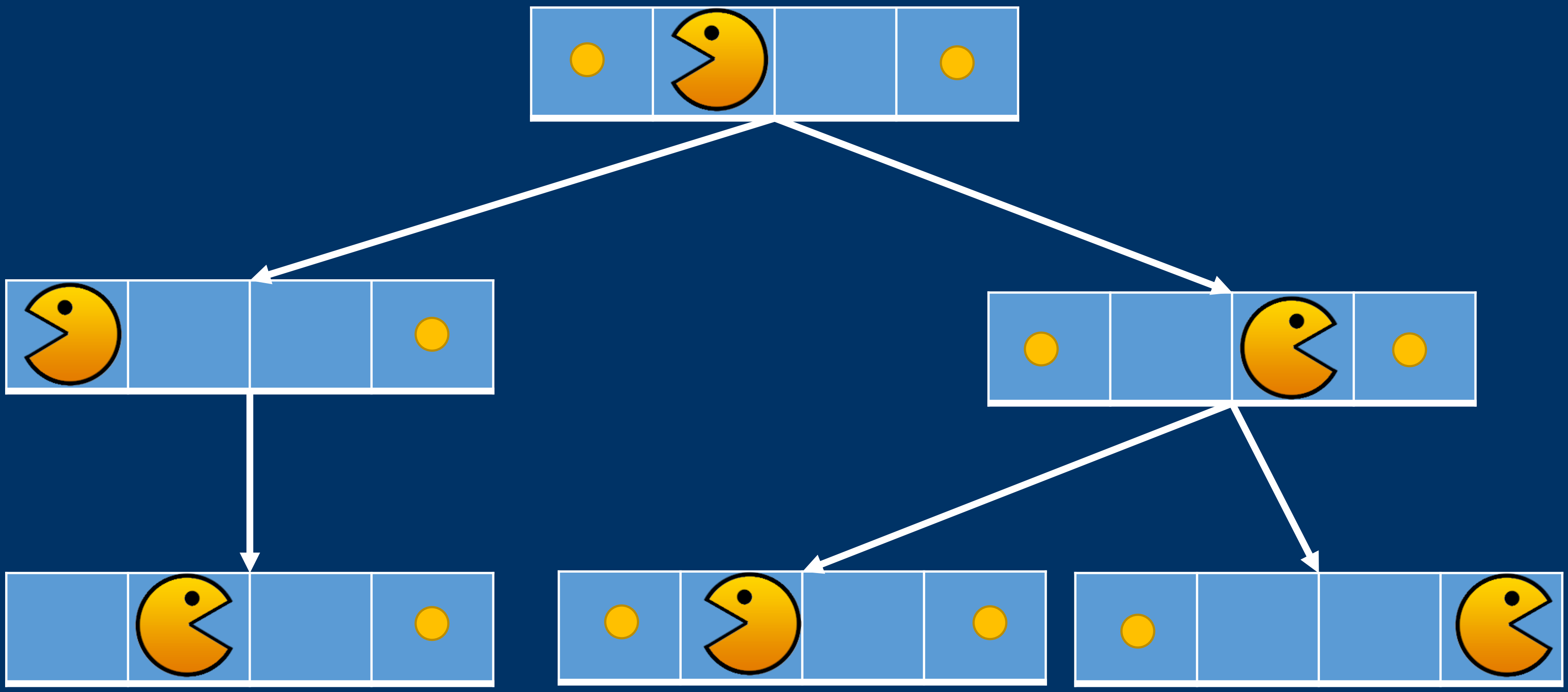
HOW?

```
maxValue(s, currDepth, α, β)
    v = -∞
    for a, s' in successors(s)
        v = max(v, value(s',
            depth+1, α, β)
        if(v ≥ β): return v
        α = max(α, v)
    return v
```

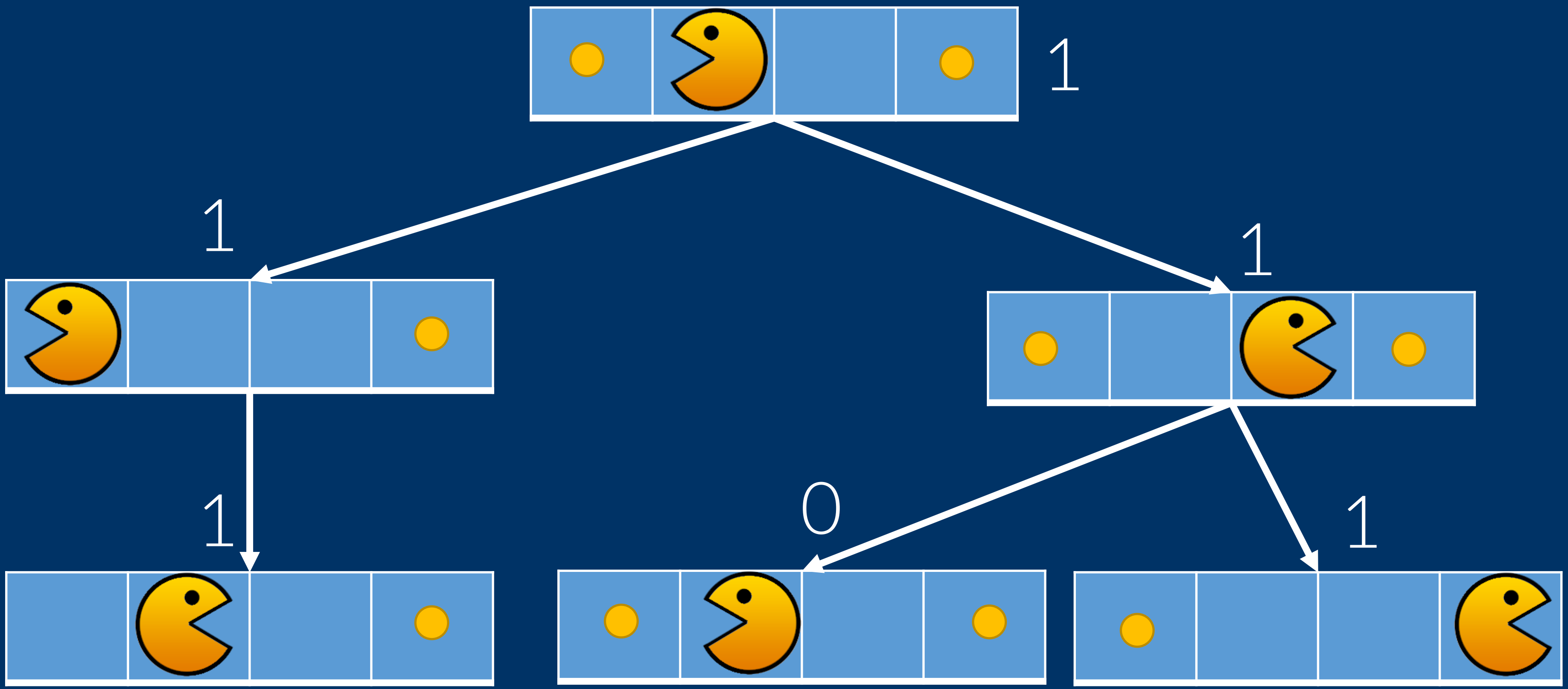
HOW?

```
minValue(s, currDepth, α, β)
    v = +∞
    for a, s' in successors(s)
        v = min(v, value(s',
            depth+1, α, β)
        if(v ≤ α): return v
        β = min(β, v)
    return v
```

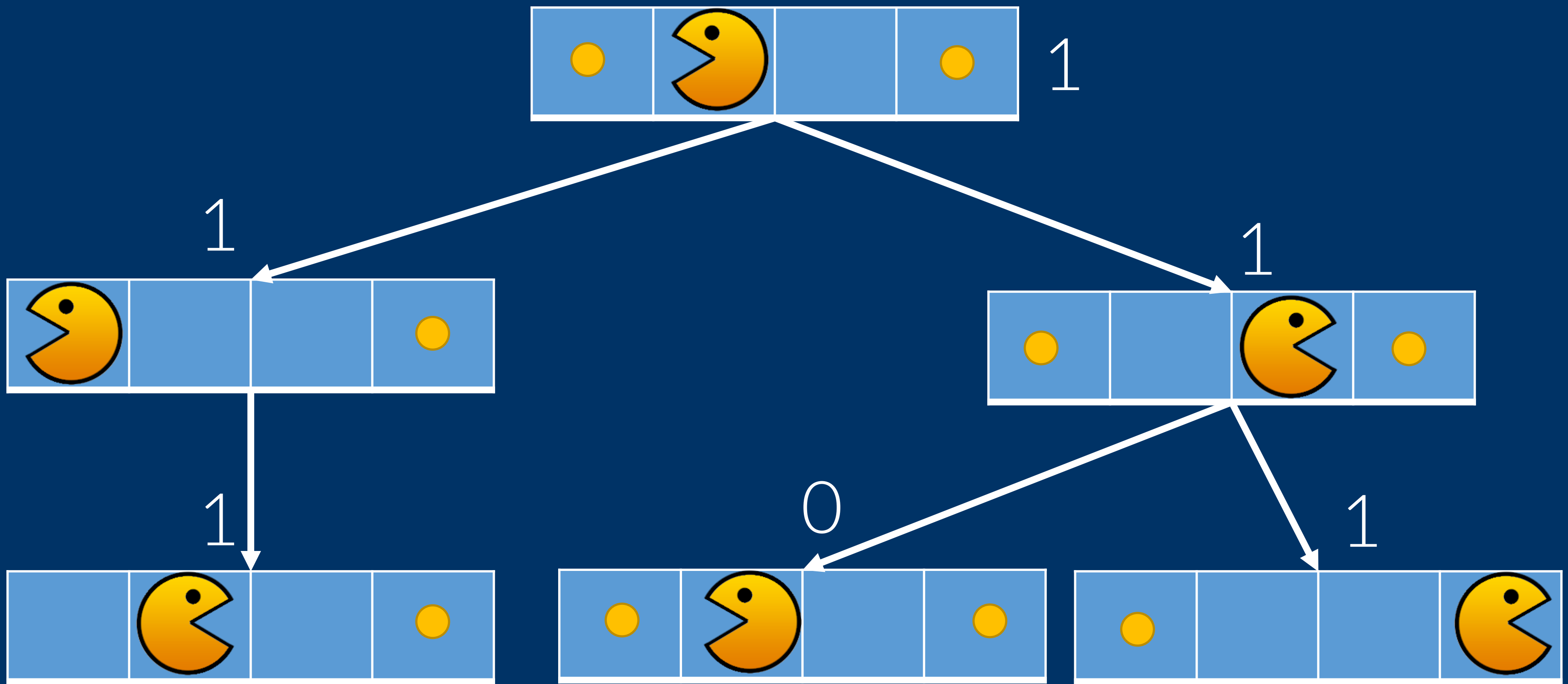
Reducing the depth is **not a perfect way** of resolving complexity since we are just **estimating**.



Say our estimation function is the amout of food Pacman gets to eat.



Say our estimation function is the amout of food Pacman gets to eat.



The agent will think that the two subtrees are equally good; they are not.