Adversarial Search I

The Core Idea

- Let’s say you are playing a game
  - Used to be playing a 1 player game
    - (Even if >1 player, agent’s “search” didn’t consider other player(s) actions)
- What if we know the other player(s) move(s)?
  - Can we model their goals?
  - Can we predict what actions they will take?
  - If so, can we make better choices?

Updating Some Terms

- Utility function $u(s) \rightarrow u(s, p)$ Only works on terminal states
- Transition function $t(s, a) \rightarrow t(s, p, a)$

- Previously talked about goal function $g(s)$
  - More general term: terminal test function $terminal - test(s)$

- Game is a 6 tuple now:
  - Initial state s_0
  - Set of players
  - Actions available to each player in each state (assume same actions for now)
  - $terminal - test(s)$
  - $u(s, p)$
  - $t(s, p, a)$

How can We Make Better Choices?

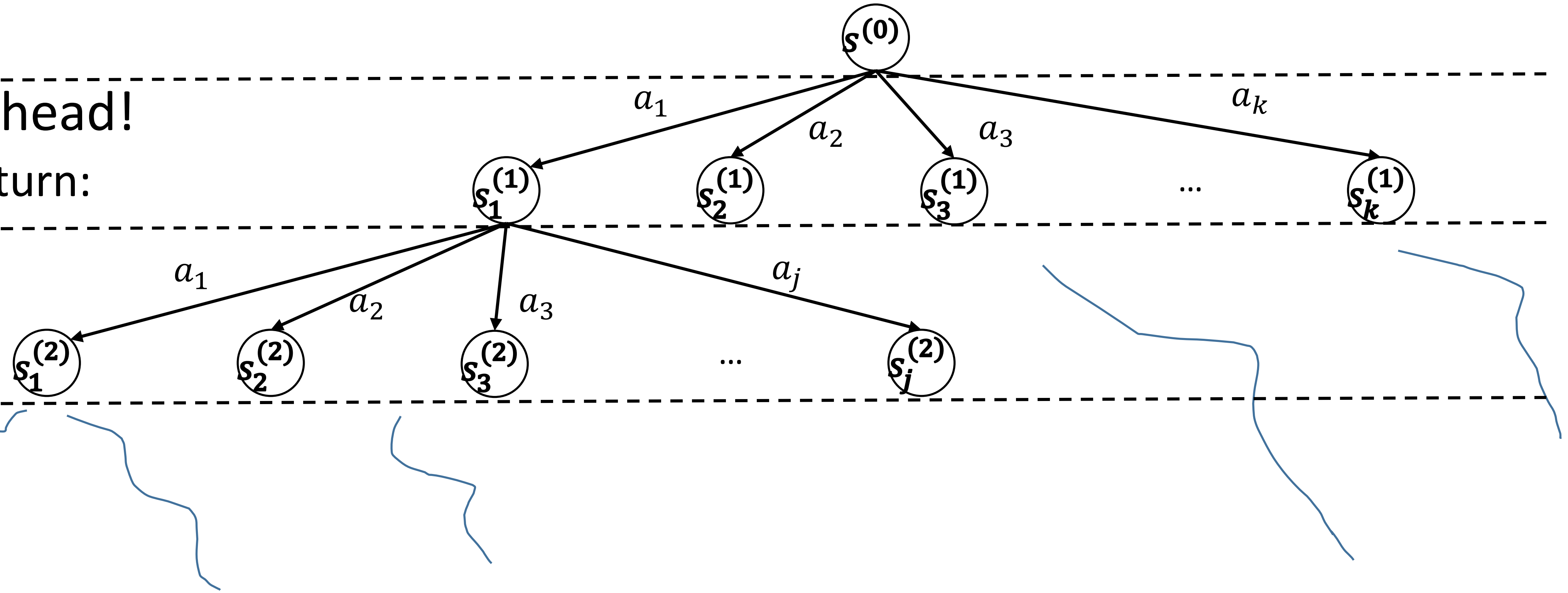
Turn 1   Turn 2   …

$p_1 \to p_2 \to p_3 \to \cdots \to p_n \to p_1 \to p_2 \to \cdots \to p_n \to \cdots$

- Lets assume game has known order of turns
  - Each player goes one at a time
  - Static world (world pauses while each agent thinks)

p_1's perspective

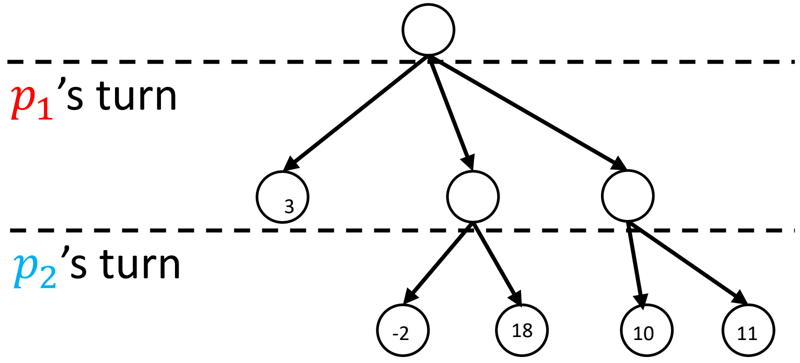
- Think ahead!
  - p_1's turn:



What to Do?

- Can expand this tree all the way down to terminal states
  - Pick the path that ends with a terminal state that's good for us?

- Why doesn't this work?
  - Practical problems:
    - Tree is massive!
    - Takes too long to expand the whole thing
  - Theoretical problems:
    - We only know utility value of terminal states:
      - How do we determine utility values of non-terminal states?
      - Combinatorial ways of combinations (b/c of lots of players)



- Need:
  - Faster tree algorithms
  - Polynomial time ways of combining other player's move(s)

Simplify (for now)

- Consider a 2-player game
  - agent1 vs agent 2 (competitive/adversarial game)
- Zero sum game (a major simplifying assumption)
  - My loss is your gain (and vice versa)
  - “true utilities” always sum to same constant (may not be zero!)

Adversary wants to force me to go to bad states (for me)

I want to force adversary in bad states (for them)

- With these two assumptions:
  - Tree is much smaller
  - Don’t need to worry about combining multiple adversaries together
    - What’s bad for me is good for you
    - What’s good for me is bad for you
  - Nonterminal states utility value = function of children utility values

The best state for me in the future!

How good my (nonterminal) state is

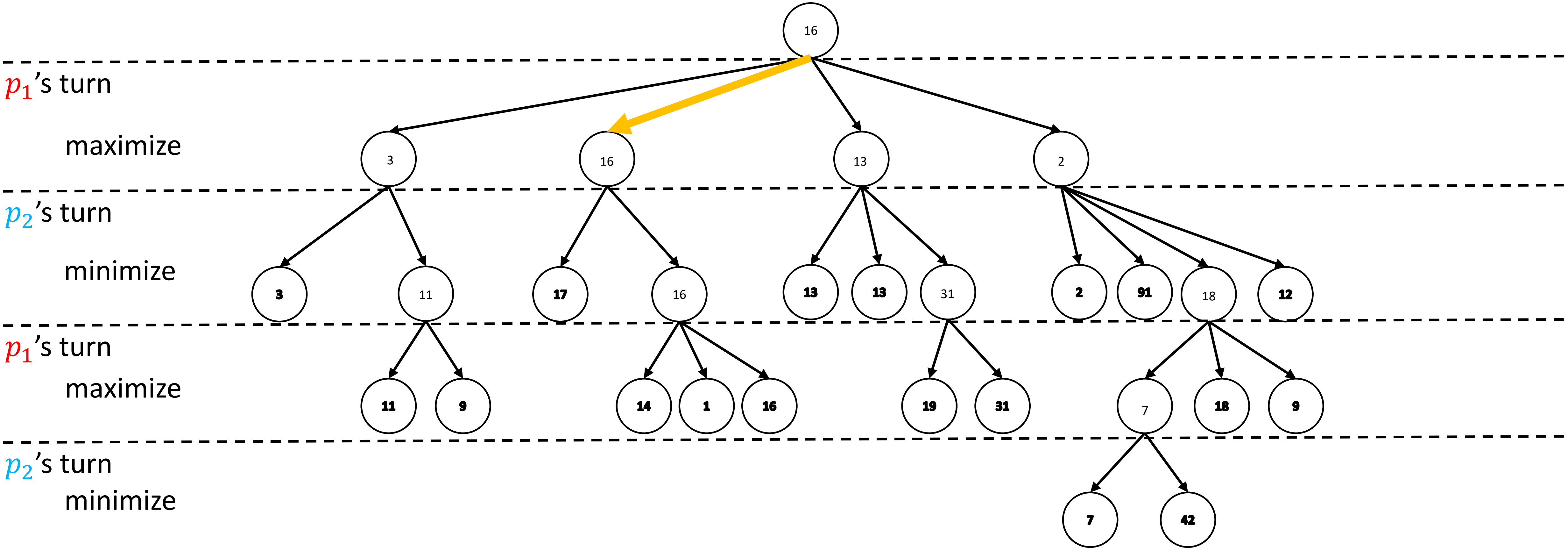
$$u(s,p) = \begin{cases} terminal - test(s) \;\cap\; p == me & u(s) \\ terminal - test(s) \;\cap\; p! = me & u(s) \\ !\,terminal - test(s) \;\cap\; p == me & \max_a u(t(s,p,a), me) \\ !\,terminal - test(s) \;\cap\; p! = me & \min_a u(t(s,p,a), me) \end{cases}$$

How good adversary’s (nonterminal) state is

The worst state for me in the future

The Minimax Algorithm

- Expand the tree (dfs is common)
- Once you get to terminal states (leaf nodes):
    - Go back to parent nodes and assign utilities
- At root:
    - Pick action which leads to the best child state

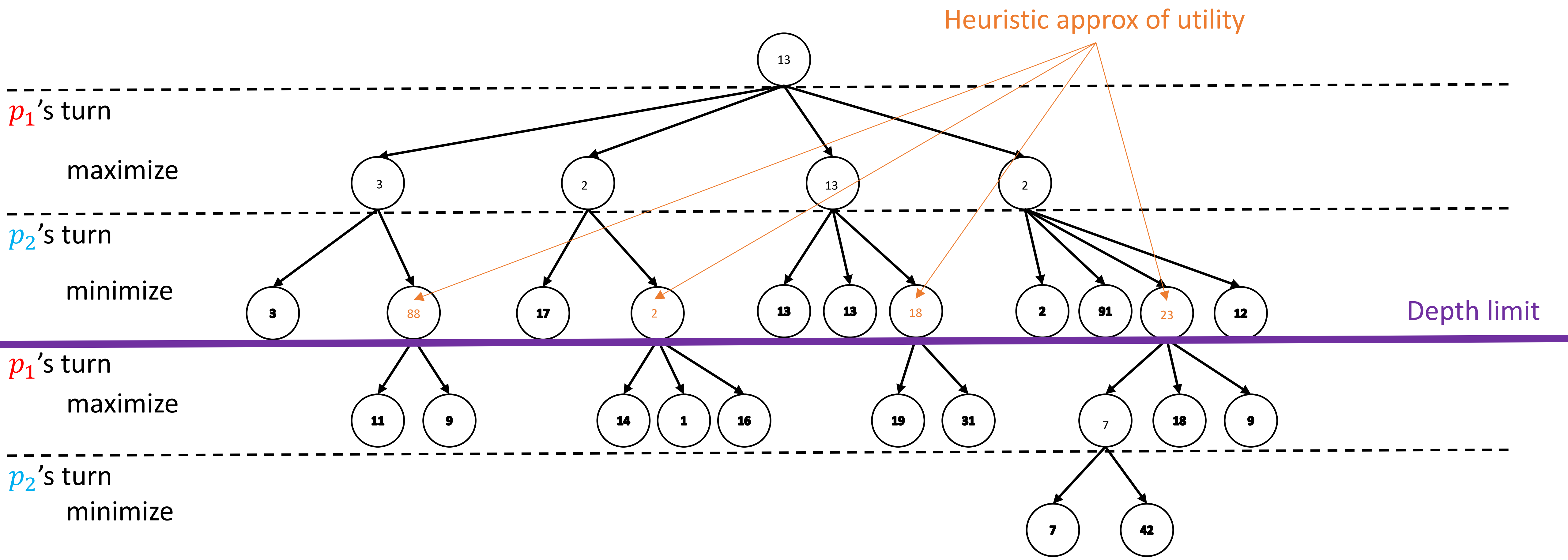


Is this a Good Model?

- Assumes adversary always makes the “best” choice
  - Humans are notorious for not doing this
  - How you beat chess bots!
    - Hint: don’t play chess with them (you will lose)
    - Instead:
      - Take advantage of time limits: make them think
        - If they run out of time they might play sub-optimally

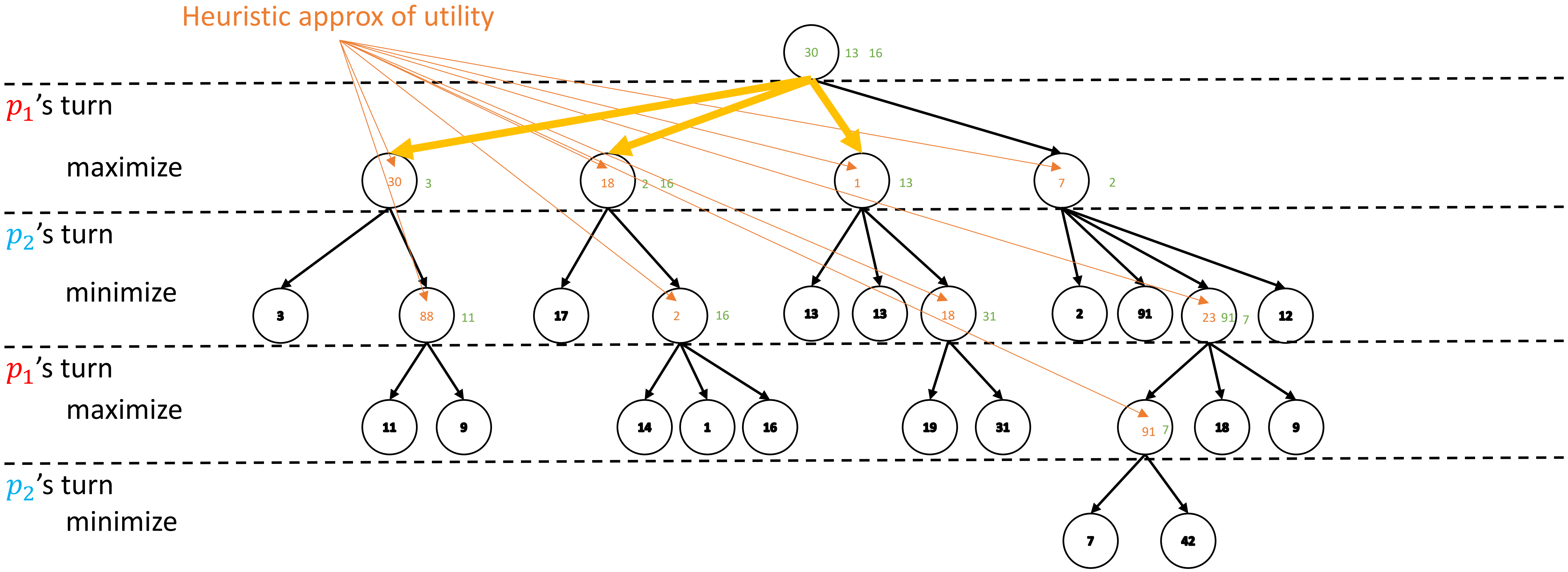
- Major problem:
  - Even these trees can be too big
  - Idea: don’t go all the way to terminal states!
    - Ex. only plan “3 moves ahead” or something
    - Need a stand in for utility values of leaf nodes in the tree
      - heuristics

Depth-Thresholded Minimax



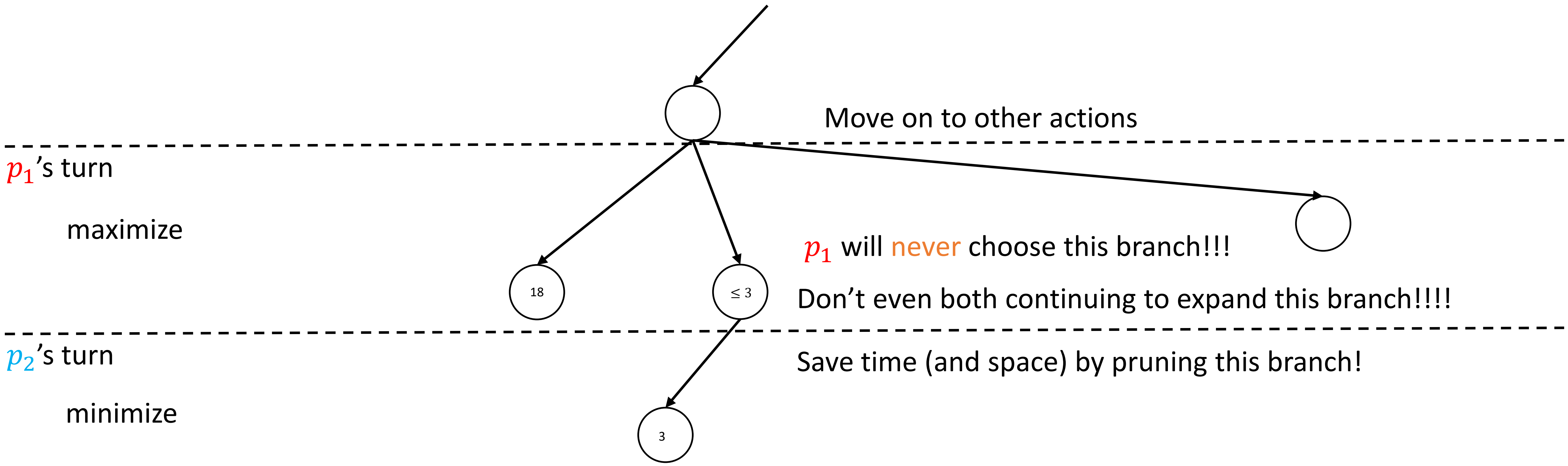
Iterative Deepening

- Expand the tree using BFS
  - Use heuristics to determine best move at a level
  - If you have more time/resources: do next level
- Whenever resources run out: return best choice so far



Problems

- Minimax is inefficient
  - Inefficiency is subtle
    - Iterative deepening / depth-thresholding don’t solve it

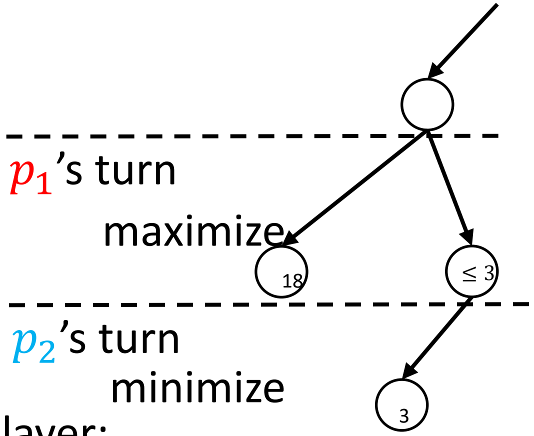


Alpha-Beta Pruning

- How do we detect scenarios like this?
  - Need to keep track of "best choice" for each player

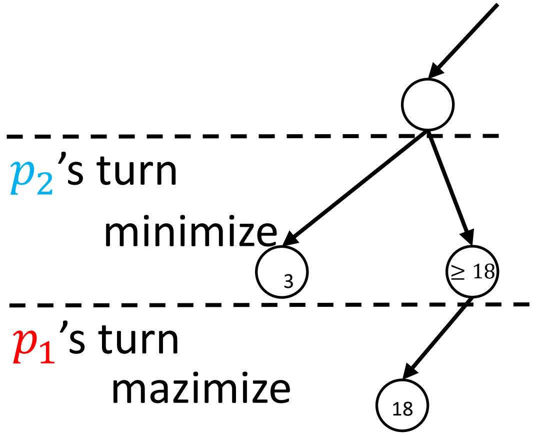
    Max player is guaranteed at least this score
  - α = min score of the maximizing player (me) so far

    Min player is guaranteed at most this score
  - β = max score of the minimizing player (adversary) so far
  - Every node gets its own pair of values (α, β)
  - Child nodes inherit values of parents
    - Child can change it's own values of (α, β) values
  - Alpha-Beta pruning is Minimax w/ "early stopping"

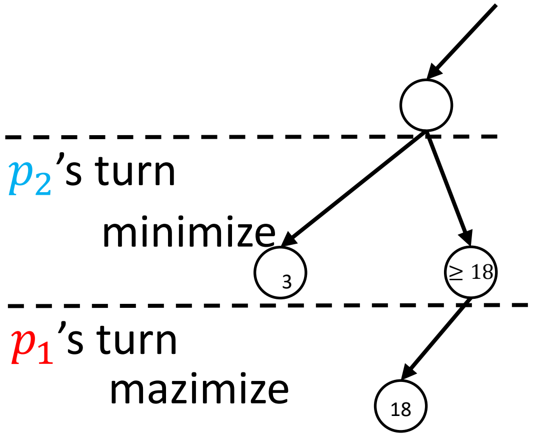


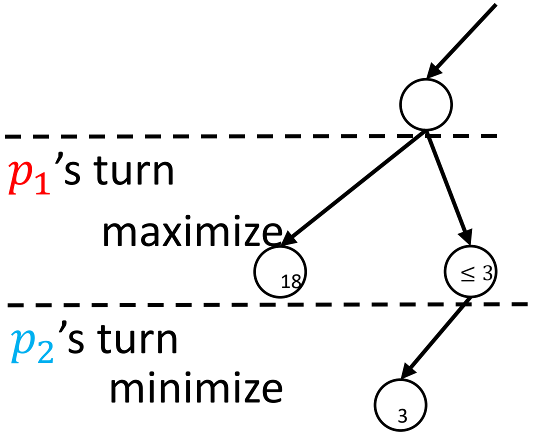
When min player:
If I encounter child state $< \alpha$:
max player will never get here
stop expanding this branch!

When max player:
If I encounter child state $> \beta$:
min player will never get here
stop expanding this branch!



Alpha-Beta Pruning





function alphabeta(s, α, β, p):
  if $terminal - test(s)$ then:
    return $u(s)$

  if $is - maximizing - player(p)$:
    $v = -\infty$
    for each child state s' of s do:
      $v = \max(v,$ alphabeta$(s', \alpha, \beta, other - player(p)))$
      if $v > \beta$ then:
        return v
      $\alpha = \max(\alpha, v)$
    return v
  else: // minimizing player
    $v = +\infty$
    for each child state s' of s do:
      $v = \min(v,$ alphabeta$(s', \alpha, \beta, other - player(p)))$
      if $v < \alpha$ then:
        return v
      $\beta = \min(\beta, v)$
    return v

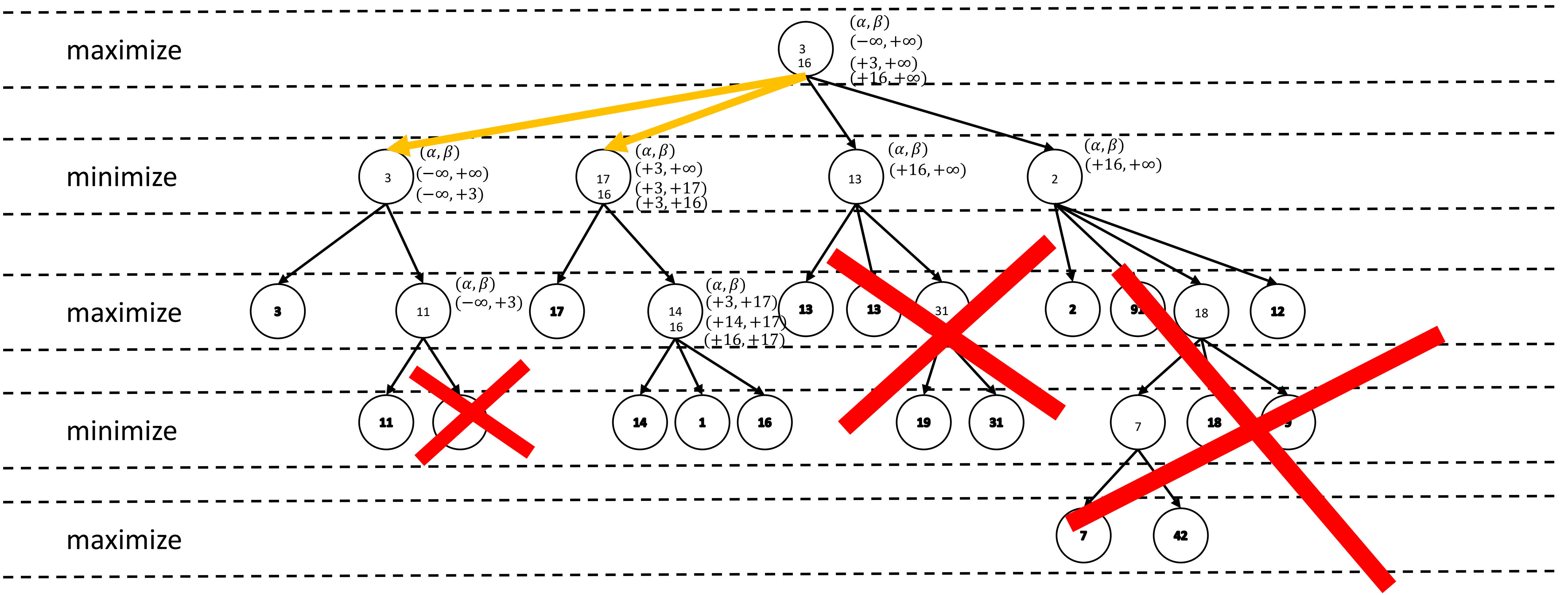
Alpha-Beta Pruning

MAX player (you)

$v = -\infty$
for each child state s' of s do:
  $v = \max(v, \text{alphabeta}(s', \alpha, \beta, other - player(p)))$
  if $v > \beta$ then:
    return v
  $\alpha = \max(\alpha, v)$
return v

MIN player (adversary)

$v = +\infty$
for each child state s' of s do:
  $v = \min(v, \text{alphabeta}(s', \alpha, \beta, other - player(p)))$
  if $v < \alpha$ then:
    return v
  $\beta = \min(\beta, v)$
return v



Variants of Alpha-Beta Pruning

- Even though we prune: Alpha-Beta still examines all the way to terminal states

- Solved this with Minimax:
  - Depth threshold ← Most common!
  - Iterative deepening

- These solutions work with Alpha-Beta too!

Problems with Alpha-Beta Pruning

- Even with Depth-Thresholding/Iterative Deepening:
  - Order in which child states are enumerated matters!
    - Pruning only occurs when we know a better option already exists!
    - What if we see better options last?
      - Never prune!
      - Without pruning, Alpha-Beta is just Minimax!

- How can we speed this up?
  - Transposition table:
    - Cache utility value for states we've seen
    - When we encounter the same state in the future, no need to expand!
    - How many states should we cache?
  - How can we encourage Alpha-Beta to prune?
    - More heuristics!
      - Impose an order on child enumeration
      - Children we think are better choices should come first in the order!

There are other ideas
Can implement multiple!

Super popular