CSCI 3202: Intro to Artificial Intelligence

Lecture 12: Games

Rachel Cox

Department of Computer Science

Relation of Games to Search

Search - no adversary

- Solution is a method for finding the goal.
- Heuristics and Constraint Satisfaction Problem techniques can find optimal solution
- Evaluation function: estimate of cost from start to goal through given node
- Examples: Path planning, scheduling activities

Games - adversary

- Solution is strategy - strategy specifies move for every possible opponent reply
- Time limits force approximate solutions
- Examples: chess, checkers, backgammon

Relation of Games to Search

- Our focus: deterministic, turn-taking, two-player, zero-sum games with perfect information (fully observable)

- Zero-sum Game: A participant's gain (or loss) is exactly balanced by the losses (or gains) of the other participant.

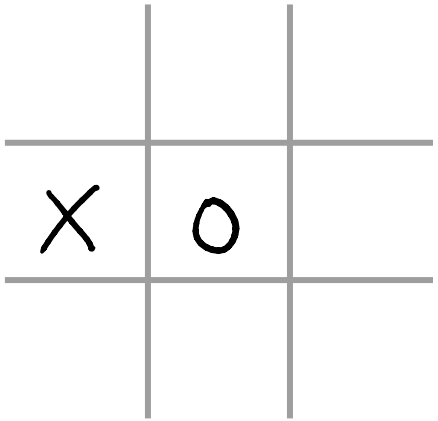
Elements of a Game

What do we need to build into our model?

- Two players: MAX and MIN
- MAX moves first and they take turns until the game is over.
- Games as Search

  - Initial state
  - *s*, current state
  - `to_play`(*s*) -- need to know whose turn it is in state *s*
  - `actions`(*s*) -- legal set of moves
  - `results`(*a*, *s*) -- result of action a in state s
  - Terminal test (`game_over`(*s*)) -- is the game over?
  - `utility`(*s*, *p*) -- the payoff to player *p* if the game ends in state *s*
    - Tic-tac-toe examples of utility payoff…

board = {(2,1): 'X'}



result(a=(2,2), s={ }) = {(2,2): 'O'}

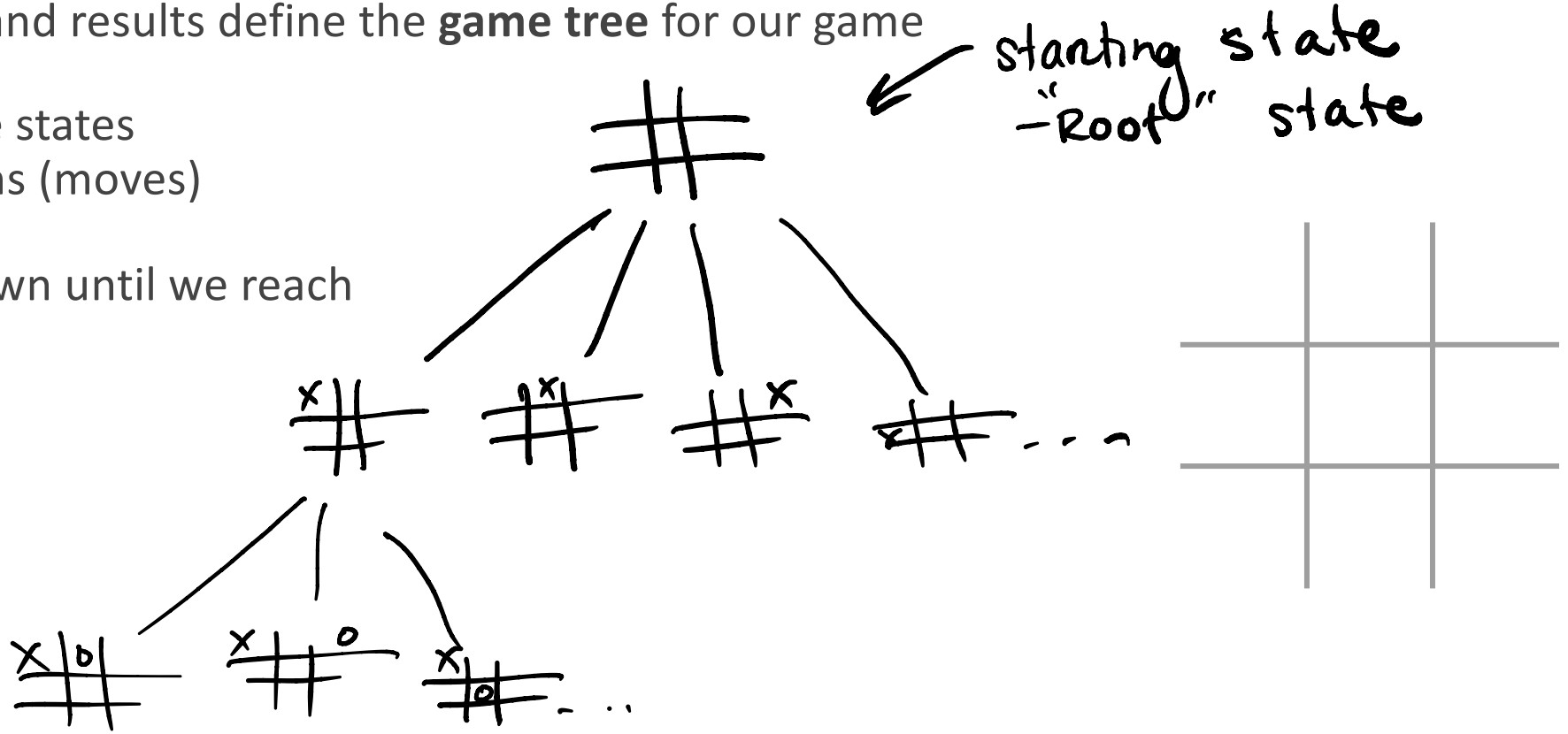
$$utility = \begin{cases} +1 & \text{if 'X' wins} \\ -1 & \text{if 'O' wins} \\ 0 & \text{if its a tie} \end{cases}$$

Game Trees

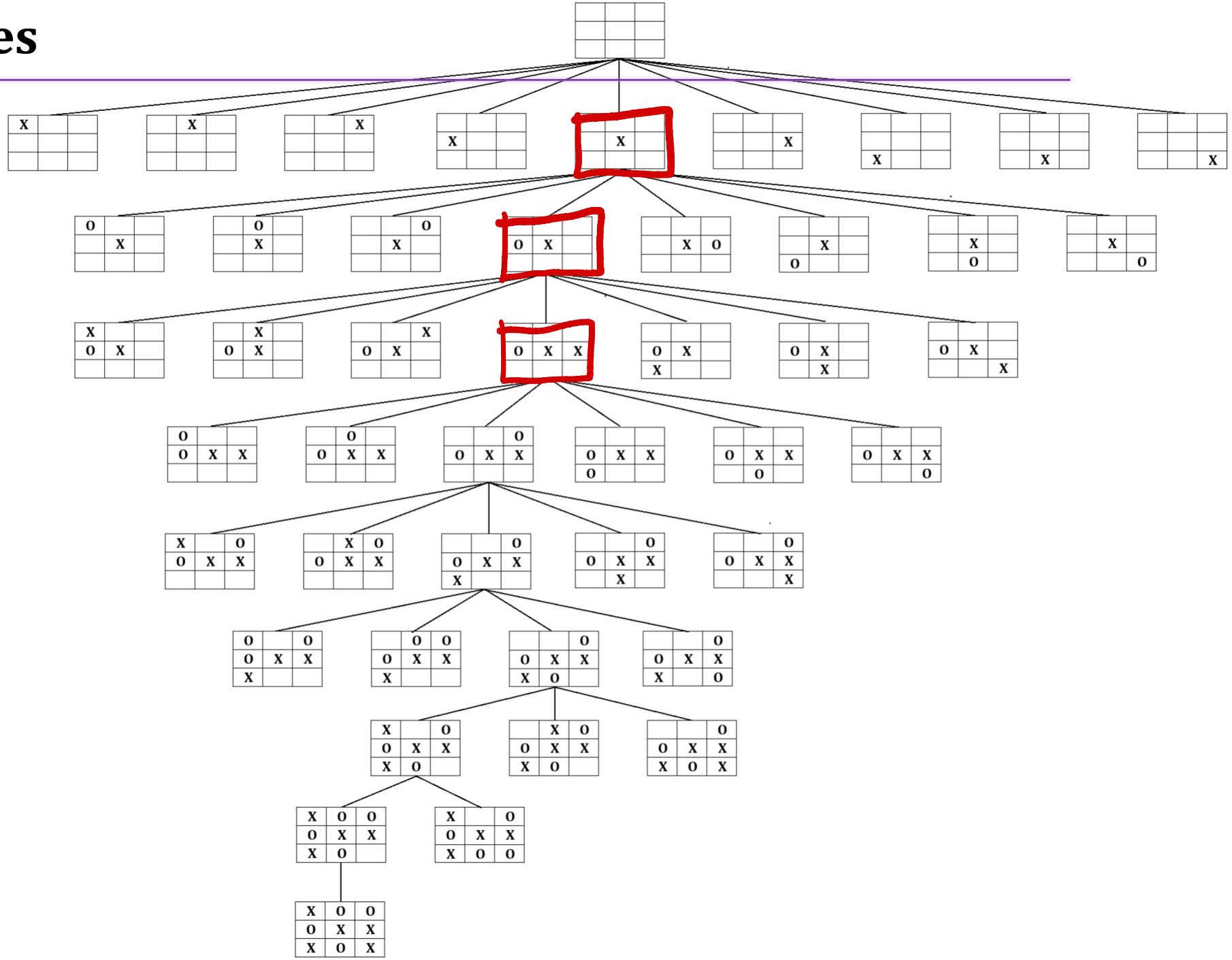
Initial state, actions, and results define the **game tree** for our game

- Nodes are game states
- Edges are actions (moves)

Can work the tree down until we reach a **terminal state**



Game Trees



Game Trees

Initial state, actions, and results define the **game tree** for our game

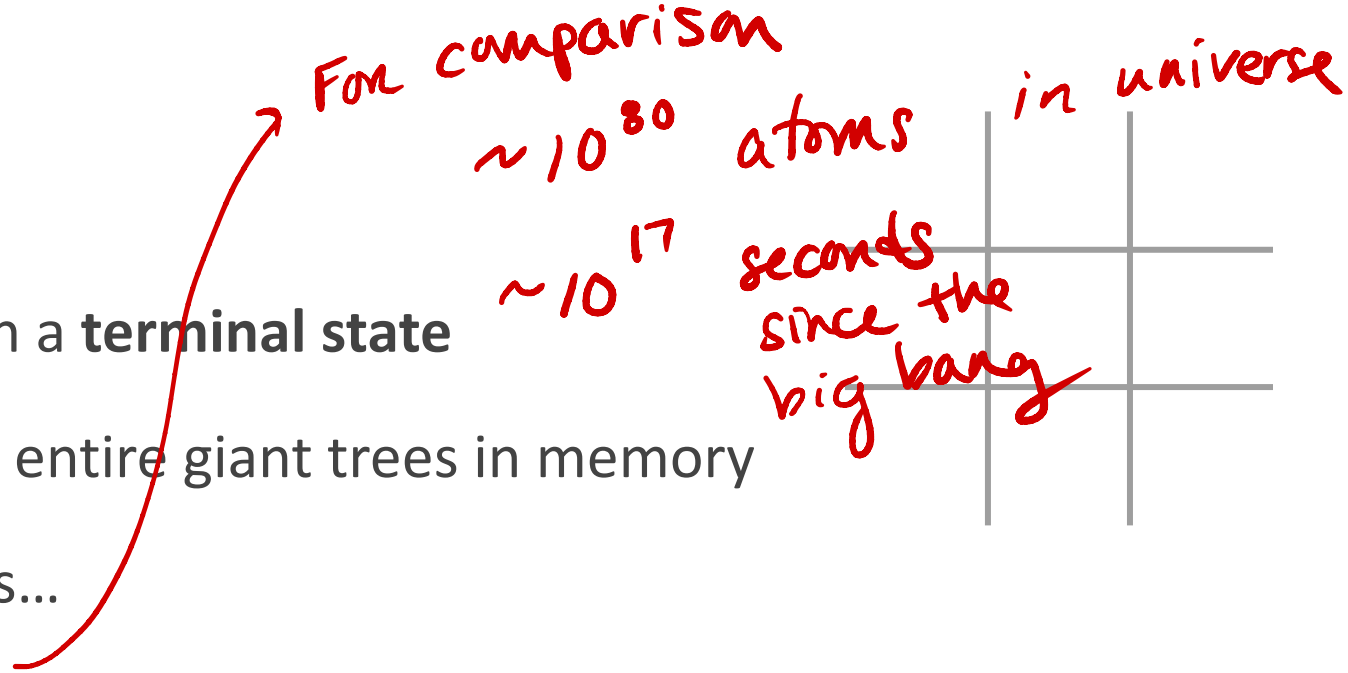
- Nodes are game states
- Edges are actions (moves)

Tic-tac-toe example…

Can work the tree down until we reach a **terminal state**

But typically don't want to build these entire giant trees in memory

- Tic-tac-toe: about 300,000 leaves…
- Chess: about 10^{120} leaves…

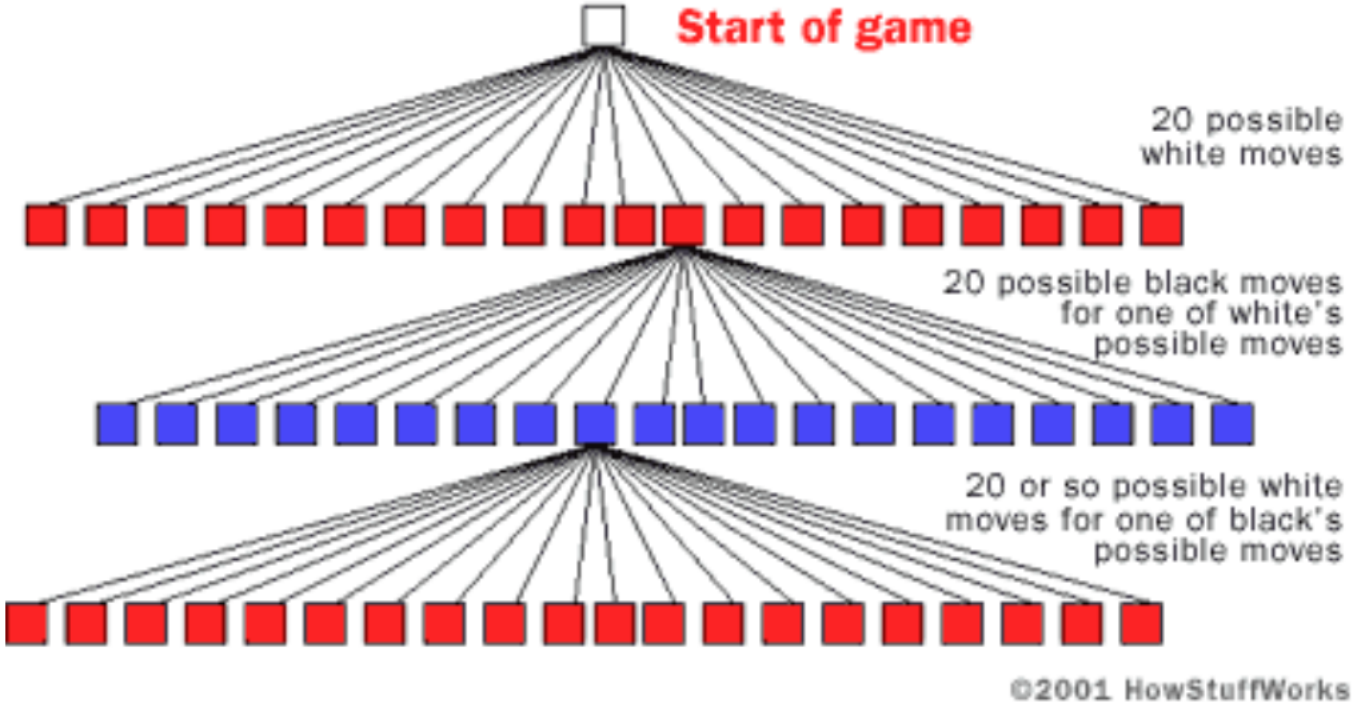


Instead, we think about the **search tree** -- finding a good move despite an intractable complete tree

Minimax

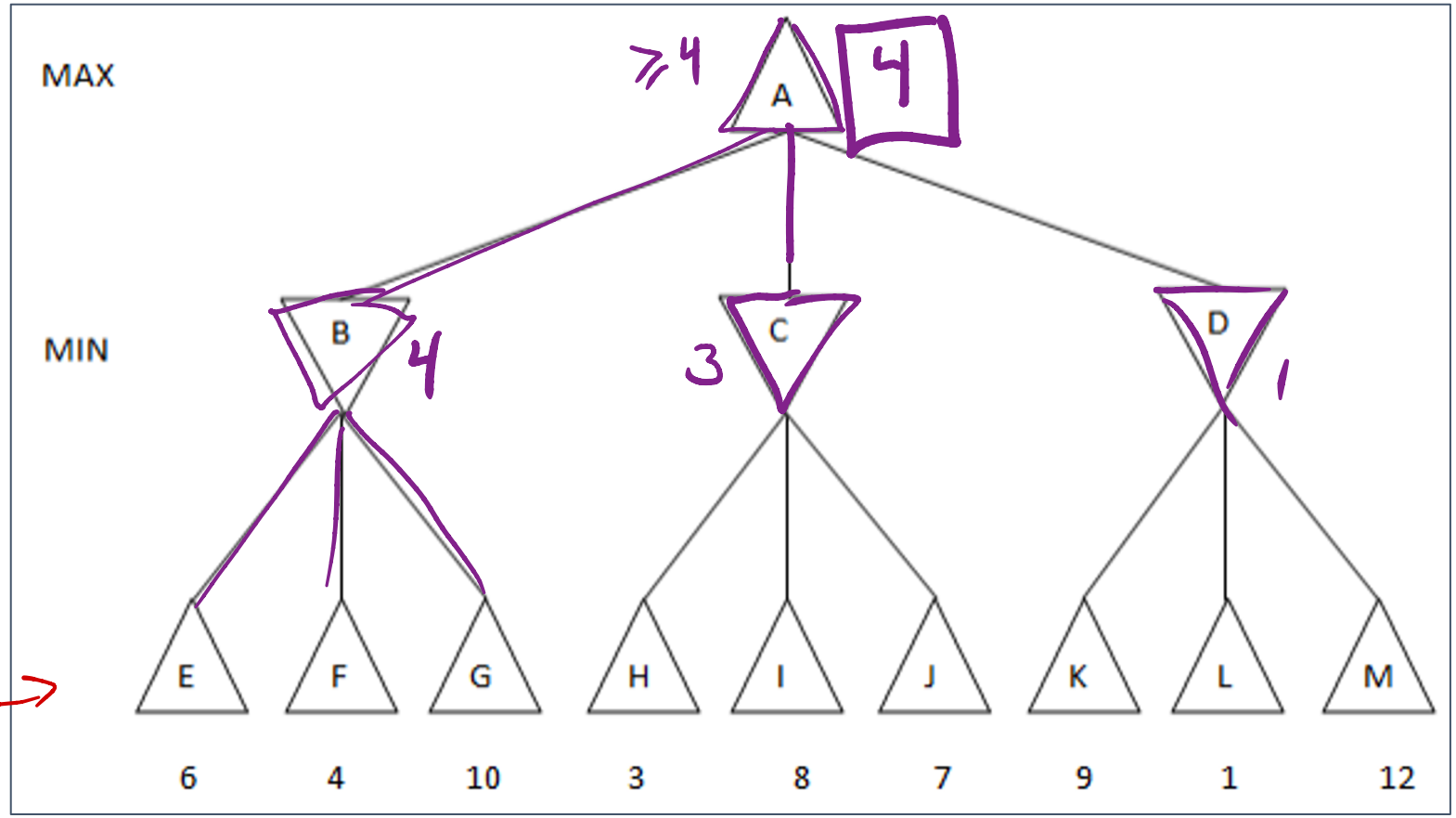
Backtracking algorithm that is used in decision making and game theory

Turn-based games: Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.



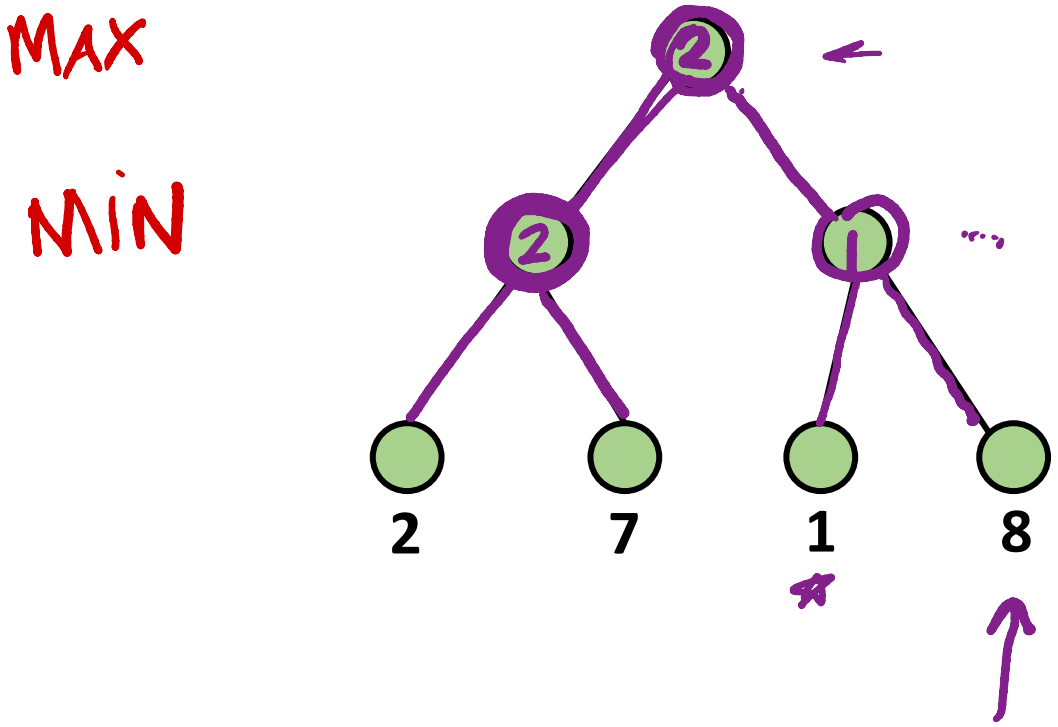
Minimax

- Two players: **Max** and **Min**
- Alternate making moves
- Optimal strategy from state *s* determined by **minimax value** of that state.



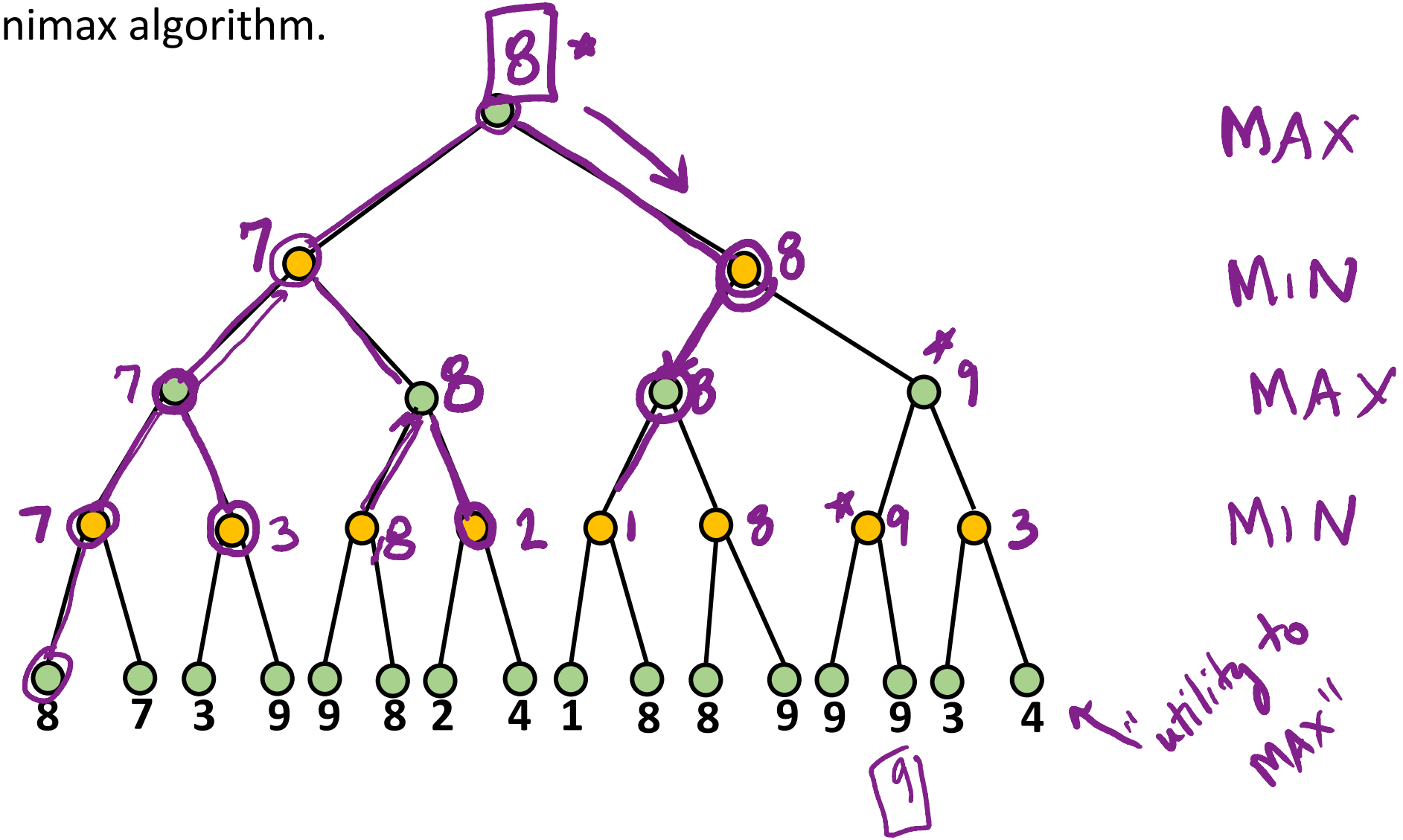
Minimax

Example: Consider the Game Tree to the right. Find the resulting value at the root node by following the minimax algorithm.



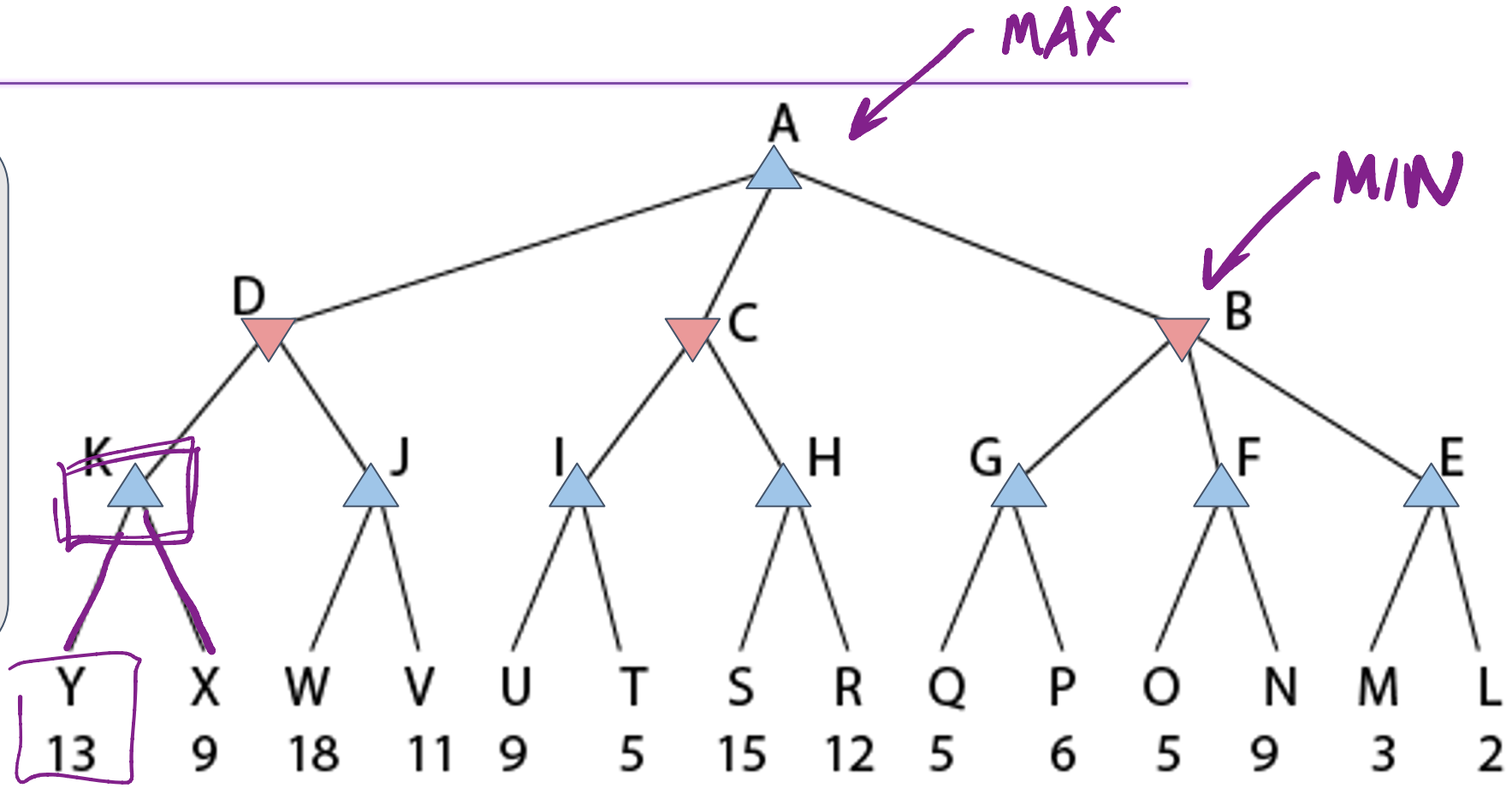
Minimax

Example: Consider the Game Tree to below. Find the resulting value at the root node by following the minimax algorithm.



Minimax

The **minimax decision** in any given state *s* is the action that is the *optimal choice* for Max/Min, because it leads to the highest/lowest minimax value.



Utility to Max

$$\texttt{minimax}(s) = \begin{cases} \texttt{utility}(s) & \text{if } \texttt{terminal}(s) \\ \max_{a \in \texttt{actions}(s)} \texttt{minimax}(\texttt{result}(s,a)) & \text{if } \texttt{player}(s) = \texttt{Max} \\ \min_{a \in \texttt{actions}(s)} \texttt{minimax}(\texttt{result}(s,a)) & \text{if } \texttt{player}(s) = \texttt{Min} \end{cases}$$

Minimax

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← 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 ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

test = float('inf')

test > 100000000

True

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

```
def minimax_decision(state):
    all_actions = what are the available actions?
    best_action = action that maximizes min_value(result(action, state))
    return best_action

def min_value(state):
    if terminal_state(state):
        return utility(state)
    value = infinity
    for action in all available actions:
        value = min(value, max_value(result(action, state)))
    return value

def max_value(state):
    if terminal_state(state):
        return utility(state)
    value = -infinity
    for action in all available actions:
        value = max(value, min_value(result(action, state)))
    return value
```

Minimax – issues

- Number of game states is exponential in the number of moves.
  - Solution: Do not examine every node.
  - **Alpha-Beta pruning**
    - Remove branches that do not influence final decision
    - General idea: you can bracket the highest/lowest value at a node, even before all its successors have been evaluated.

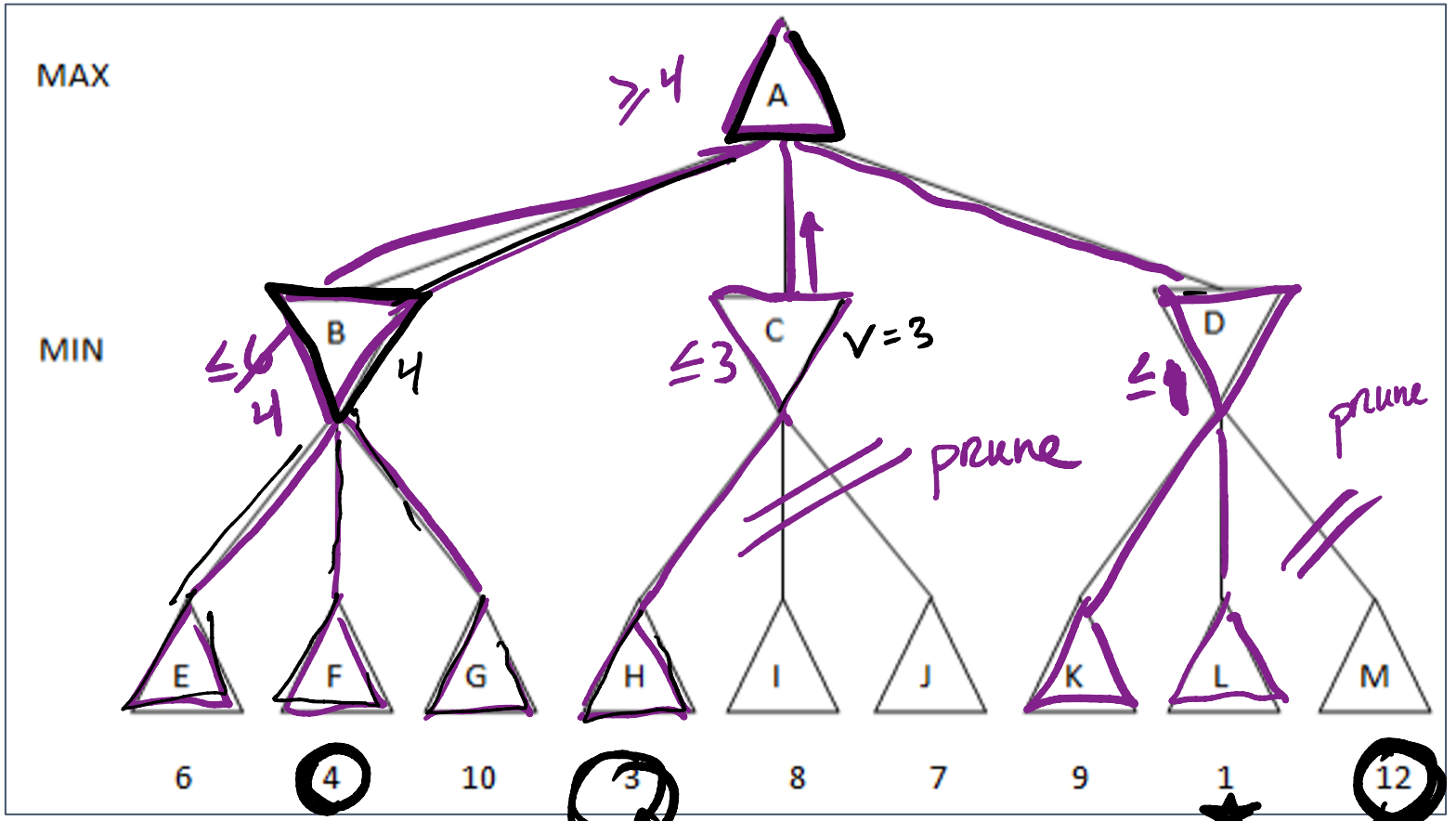
Alpha–Beta Pruning

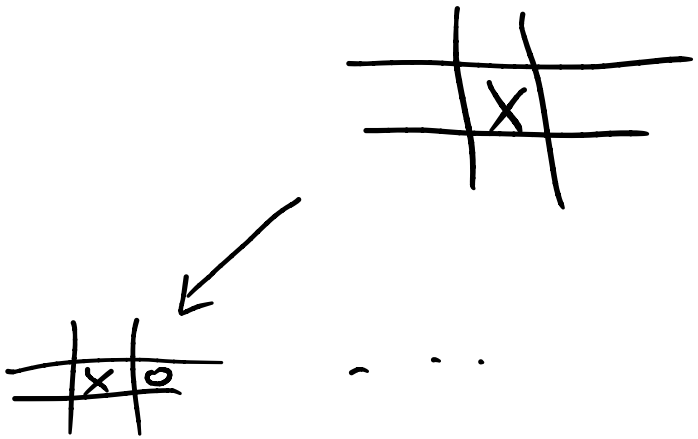
Doing a full search on the game tree is generally intractable.

DFS time complexity: $O(b^m)$
- m = maximum depth
- b = branching factor

BUT if we assume optimal play, then there are some branches of the game tree that we don't need to explore

→ ***pruning*** those branches



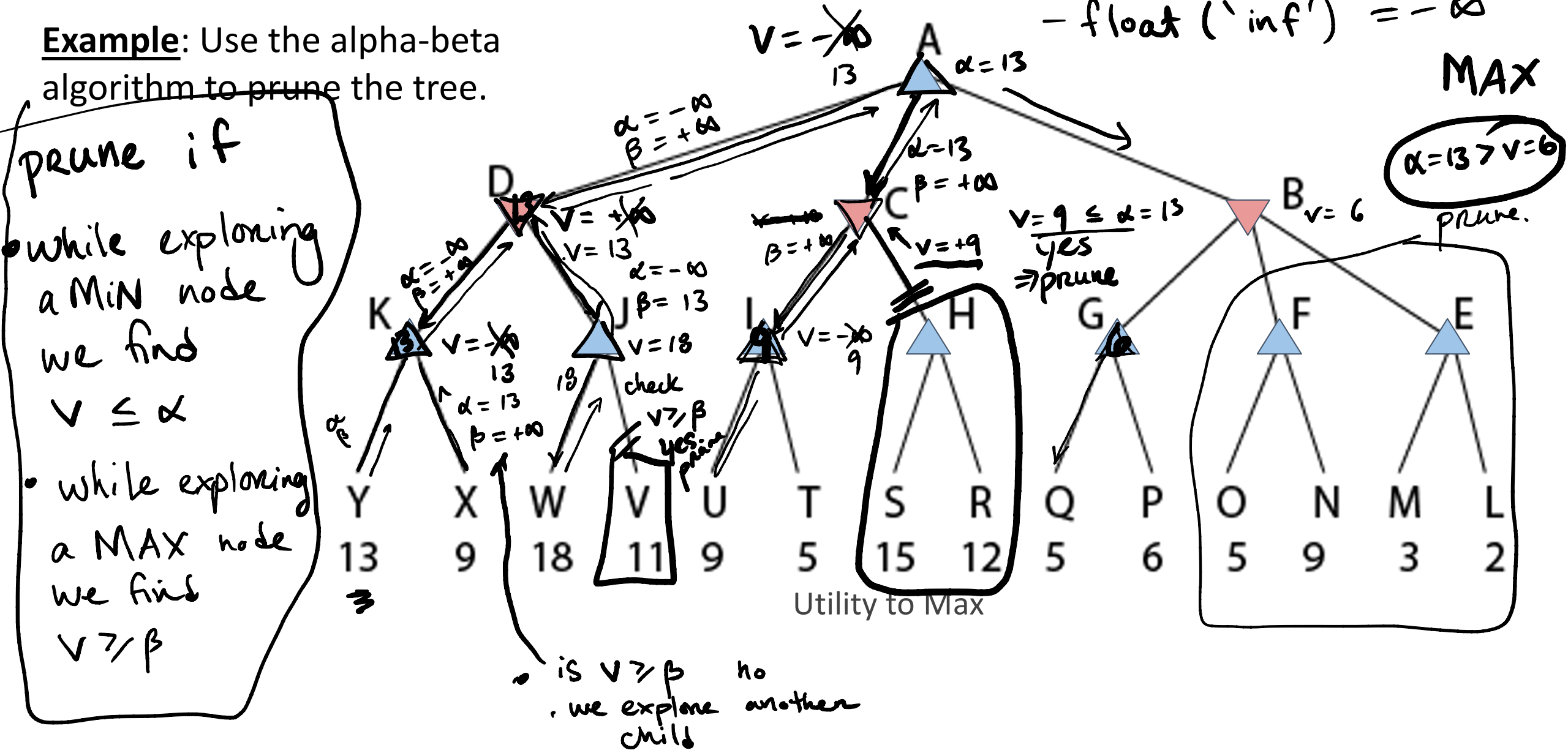


Alpha–Beta Pruning

float('inf') = ∞

−float('inf') = −∞

Example: Use the alpha-beta algorithm to prune the tree.

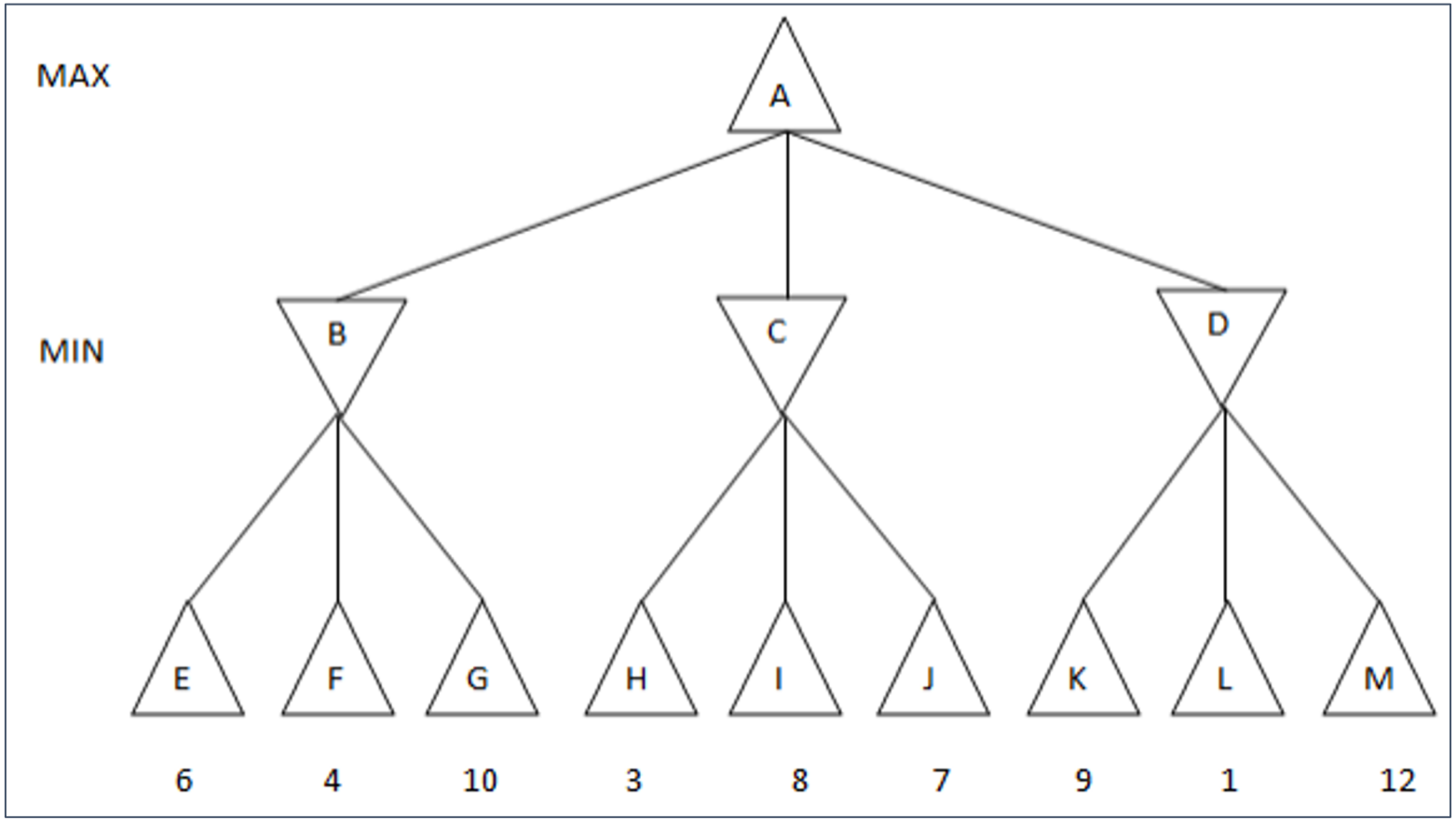


Alpha–Beta Pruning

We can keep track of the best choices for each player using two parameters:

α = value of best (highest) choice we have found so far along path from the root for Max

β = value of best (lowest) choice we have found so far along path from the root for Min



Alpha–Beta Pruning

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a), α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

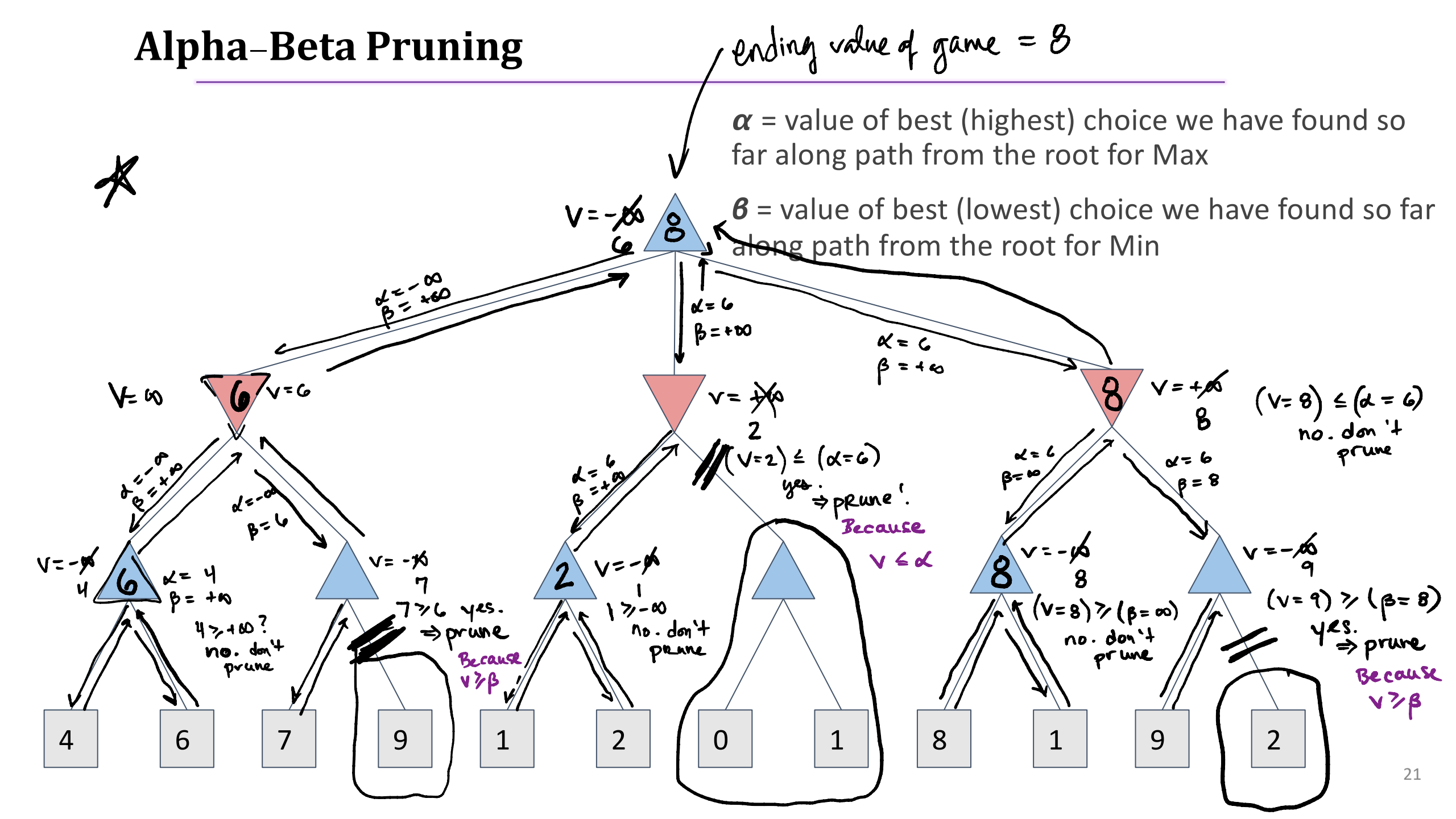
α = best available option for MAX along path to Root

β = best available option for MIN along path to root

Alpha–Beta Pruning

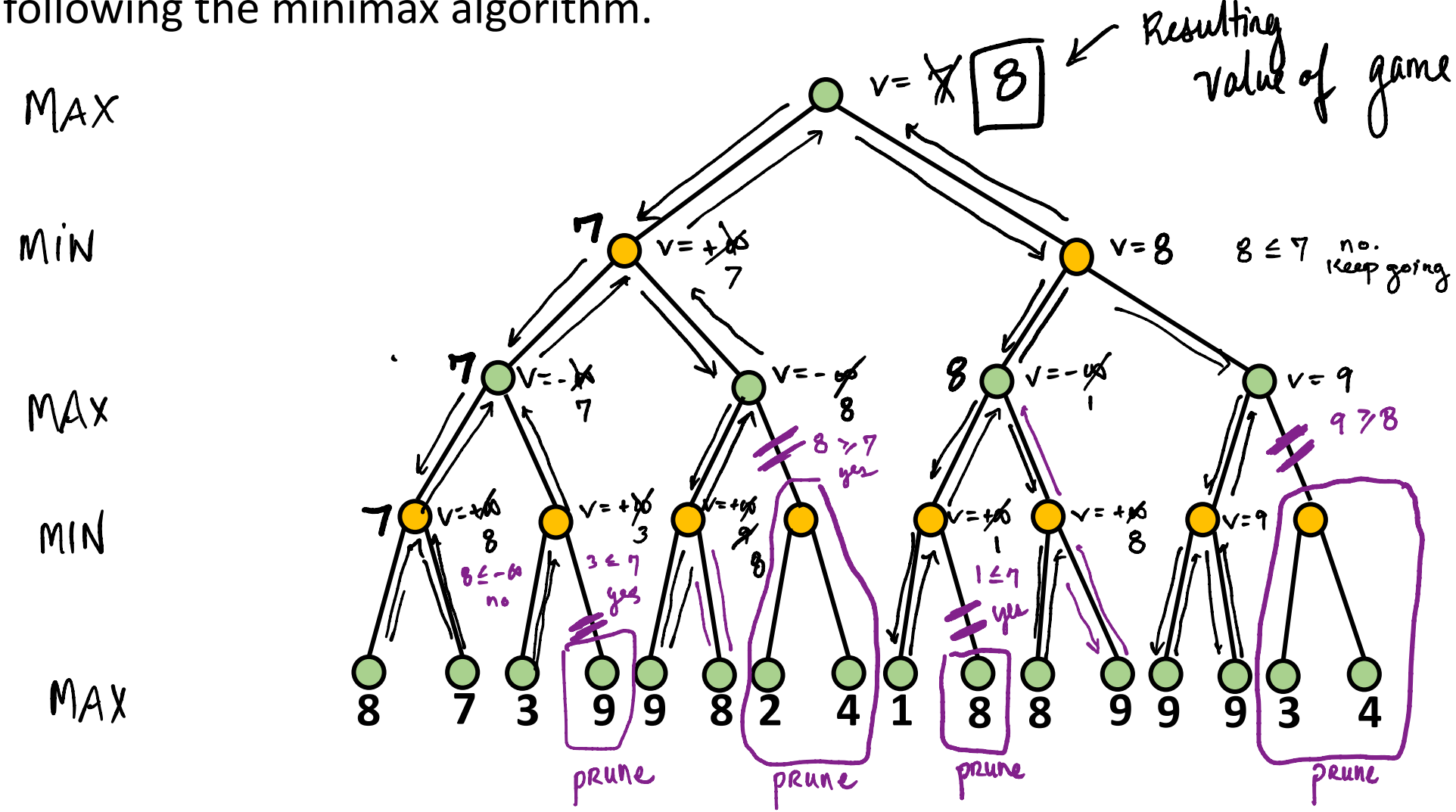
```
def alphabeta_search(state):
    alpha = -infinity
    beta  = +infinity
    value = max_value(state, alpha, beta)
    best_action = action that has utility=value to Max
    return

def max_value(state, alpha, beta):
    if terminal_state(state):
        return utility(state)
    value = -infinity
    for action in all available actions:
        value = max(value, min_value(result(action, state), alpha, beta))
        if value >= beta: return value
        alpha = max(value, alpha)
    return value

def min_value(state, alpha, beta):
    if terminal_state(state):
        return utility(state)
    value = +infinity
    for action in all available actions:
        value = min(value, max_value(result(action, state), alpha, beta))
        if value <= alpha: return value
        beta = min(value, beta)
    return value
```

Alpha–Beta Pruning



Alpha–Beta Pruning

Example: Consider the Game Tree to below. Find the resulting value at the root node by following the minimax algorithm.



Alpha–Beta Pruning

- Pruning does not effect final results
- Entire subtrees can be pruned, not just leaves.
- Alpha-beta pruning can look twice as deep as minimax in the same amount of time.
- Minimax and alpha-beta pruning still have exponential complexity.

Next Time

- ***Games Notebook Day!***