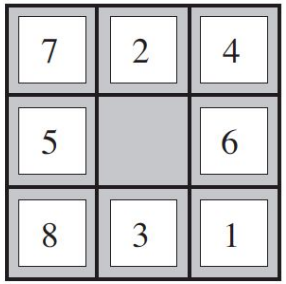
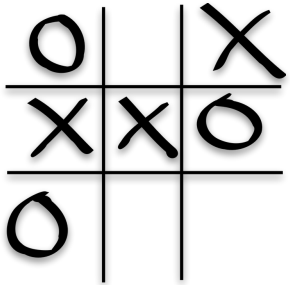
5. Adversarial Search

5.1 Adversarial search

- Search in competitive environments
  - where agents' goals are in conflict
- Games are good examples of adversarial search
  - States are easy to represent (unlike many other real world problems)
  - Agents are restricted to a small number of actions
  - Outcome of agent actions defined by precise rules
  - Usually too hard to solve



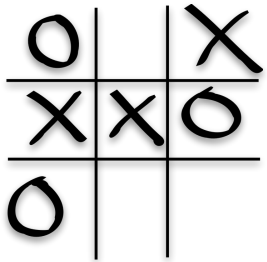


Which one/s are adversarial search?

THINK PAIR SHARE

5.1 How to formally define a game?

- Consider a game with two players MAX and MIN
  - MAX moves first (places X) followed by MIN
- A game can be formally defined as a search problem with the following elements:
  - S_0 - The initial state
  - Player (s) - Defines which player has the move in a state (s)
  - Actions(s) - Returns the set of legal moves in a state
  - Results (s, a) - The transition model which defines the result of a move
  - Terminal-Test (s) - True when game is over, false otherwise
  - Utility (s, p)
    - A utility function (also objective function or payoff function)
    - Defines the final numeric value for a game that ends in a terminal state s for player p
    - For example, in tic-tac-toe, the outcome is win, loss, or draw with values -1, 0, or 1



- The Initial State, 'Actions' function, and Result function define game tree for a game
  - Game tree - a tree where nodes are game states and edges are moves

5.1 Game tree for tic-tac-toe

- MAX has 9 possible moves from the initial state
- Play alternates between MAX's placing an X and MIN's placing an O
- The number in each leaf node indicates the utility value of the terminal state from the point of view of MAX
  - High values are assumed to be good for MAX and bad for MIN
- The game tree is relatively small
  - How many nodes are there in the game tree?

THINK PAIR SHARE

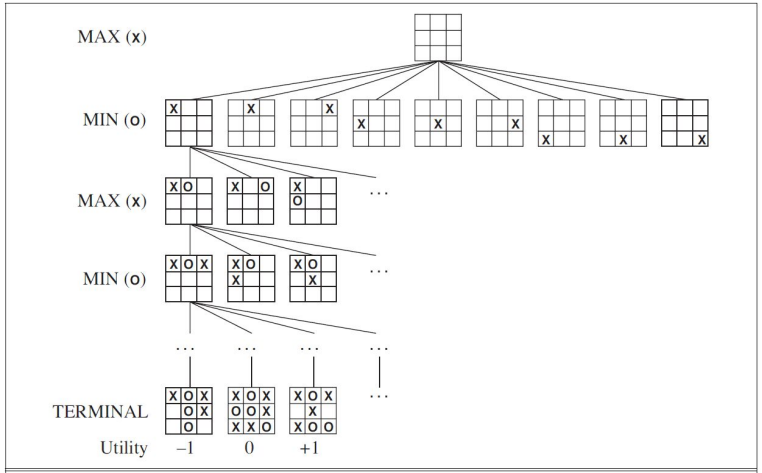


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

5.2 Optimal decisions in games

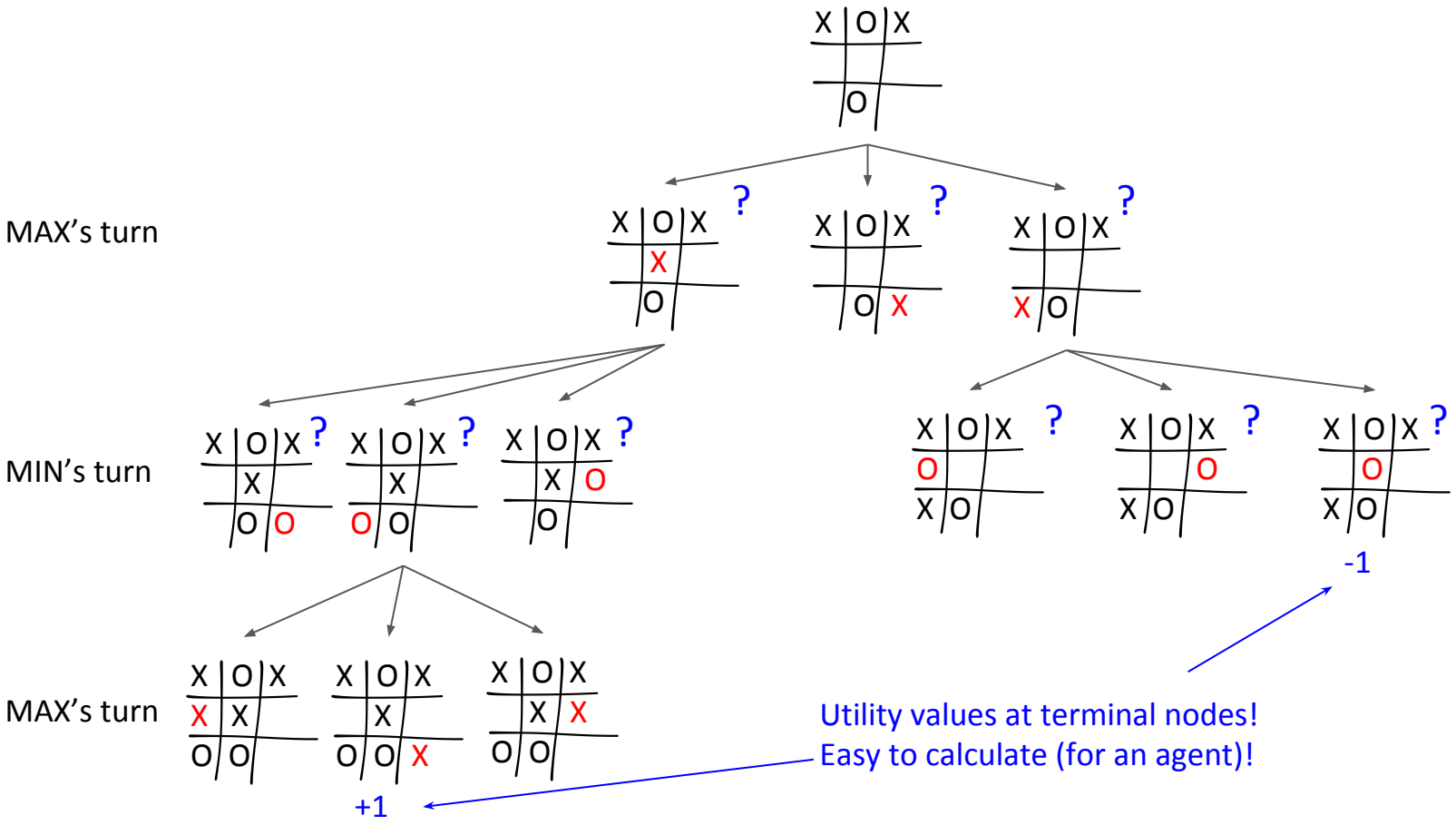
A normal search problem:

- The optimal solution is a sequence of actions leading to a goal state

An adversarial search problem:

- 'MIN' interferes the sequence of actions
- Strategy for 'MAX':
  - Specify moves in the initial state
  - Observe every possible response by MIN
  - Specify moves in response
  - and so on..
- This optimal strategy can be determined from the minimax value of each node
  - This number tells you if a state is 'gold' or 'brass'

How to calculate utility of non-terminal nodes?



5.2 How to calculate **minimax** value?

Consider a ‘reduced’ tic-tac-toe game (because game tree for full tic-tac-toe is too big)

- The possible moves for MAX at the root are a1, a2, and a3
- Possible replies to a1 for MIN are b1, b2, b3, and so on

Optimal strategy can be determined using minimax value of each node MINIMAX(n)

- MINIMAX(n) for the user MAX is the utility of being in the corresponding state
- So, MAX will always prefer to move to a state of maximum value

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

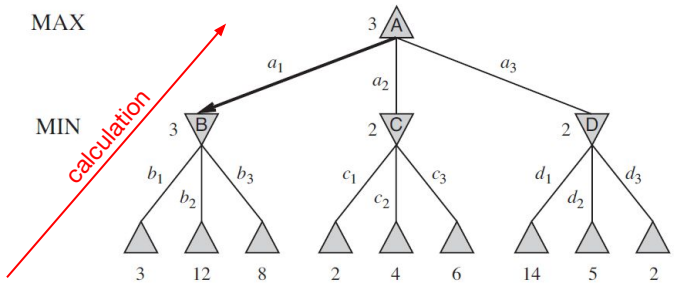


Figure 5.2 A two-ply game tree. The △ nodes are “MAX nodes,” in which it is MAX’s turn to move, and the ▽ nodes are “MIN nodes.” The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX’s best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN’s best reply is b_1, because it leads to the state with the lowest minimax value.

Making the **‘minimax decision’**

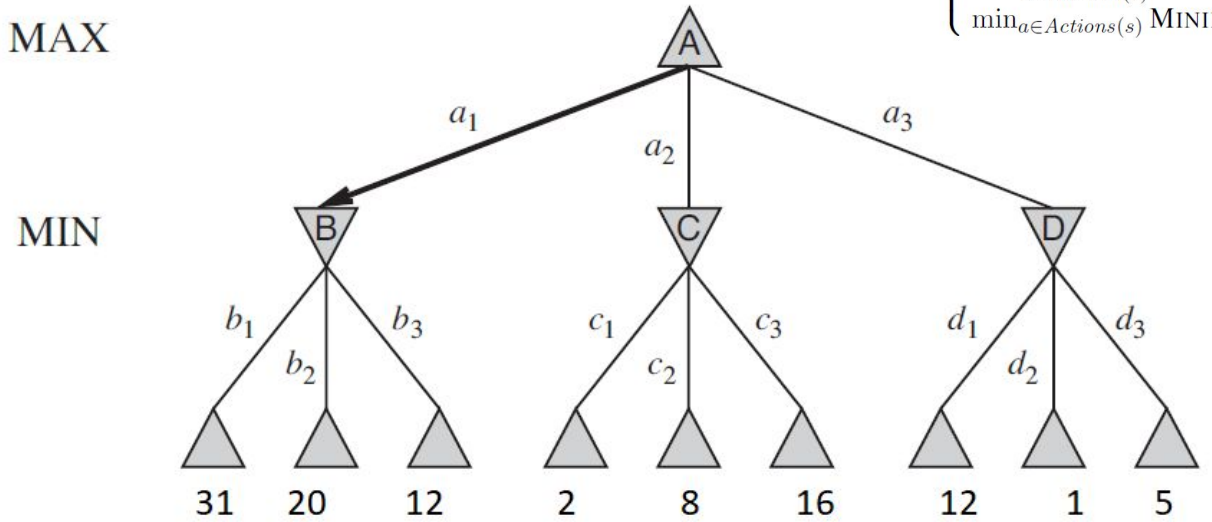
1. The Utility (s, p) function gives the utility values of the terminal nodes (3, 12, 8, 2, 4, etc.)
2. For terminal nodes, Utility (s, p) = minimax value
3. For node B (when player is MIN) the utility/minimax value will be min(3,12,8) = 3.
4. For node C, minimax value = min(2,4,6)
5. For the root node A (when player is MAX), the successor nodes have minimax values of 3, 2, and 2. So, max(3,2,2) = 3

So, the minimax of node A for the player MAX is 3 and action a1 is the optimal choice for MAX.

5.2 How to calculate **minimax** value?

Practice Problem: Calculate the minimax values at nodes A, B, C, and D for the player MAX given the game tree below. The numbers at the leaf nodes represent the values of `Utility (leafnode, MAX)`.

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$



Utility function vs Minimax function

Utility (s, p)

- defines the final numeric value for a game that ends in a terminal state s for player p

Minimax (s, p)

- defines the numeric values at all other nodes

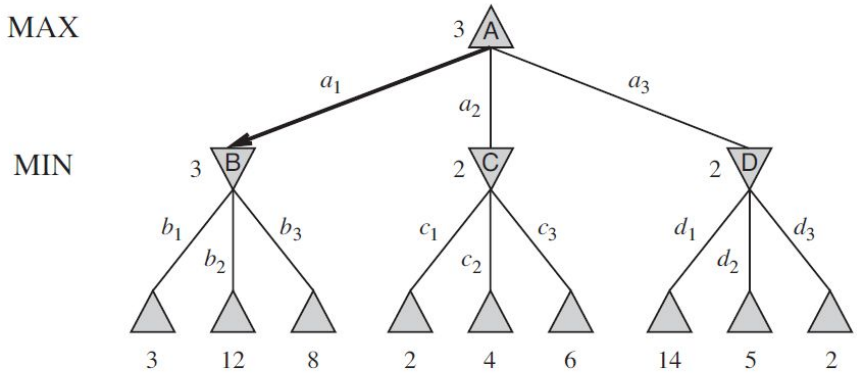
$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

5.2.1 The minimax algorithm

The recursion proceeds all the way down to the leaves of the tree, and then the **minimax values are backed up through the tree** as the recursion unwinds.

The minimax algorithm performs a **complete depth-first exploration of the game tree.**

If the maximum depth of the tree is 'm' and there are 'b' legal moves at each point, **time Complexity = O(b^m).** This time complexity is high?



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

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 $f(a)$.

5.3 Do we need to compute all ‘MINIMAX’ values?

Let the two unevaluated successors of C have values x and y.

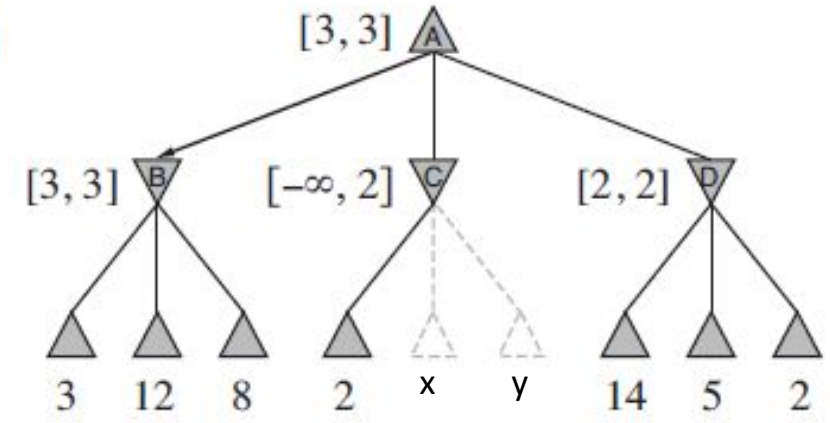
Calculate the MINIMAX(root) or node A (in terms of x and y):

```
= max(min(3,12,8), min(2,x,y), min(14,5,2)

=

=

=
```



Is the minimax decision dependent on the values of x and y?

5.3 Which nodes to prune?

Principle for pruning:

- Consider a node ‘n’ somewhere in the tree, such that the “player” has a choice of moving to that node
- If the “player” has a better choice ‘m’ either at the parent of node ‘n’ or at any choice point further up, then ‘n’ will never be reached in actual play

Once we know enough about ‘n’,
to reach such a conclusion,
we can prune it.

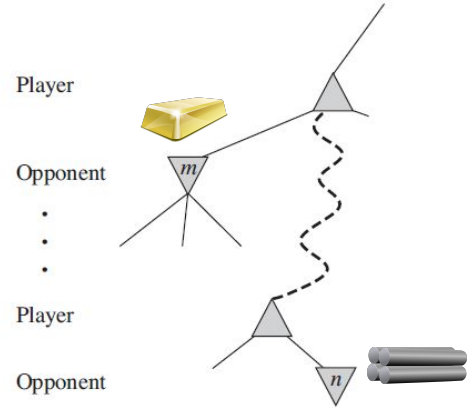


Figure 5.6 The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

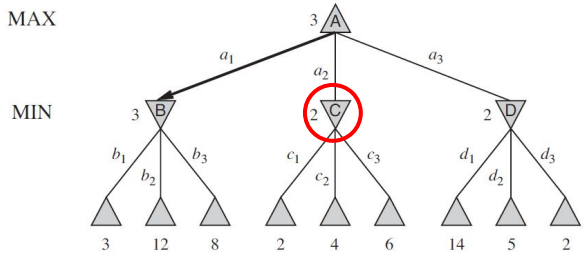
5.3 The Alpha-beta pruning algorithm

Minimax search is depth-first, so we consider the nodes along a single path in the search tree.

α = the value of the best node (highest value) we have found so far at any choice point along the path for MAX

β = the value of the best node (lowest value) we have found so far at any choice point along the path for MIN

Principle: Alpha-beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e. terminates the recursive call) as soon as the value of the current node is known to be worse than the current value of α and β for MAX and MIN respectively.



5.3 The Alpha-beta pruning algorithm

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

---

function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in ACTIONS($state$) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **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), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

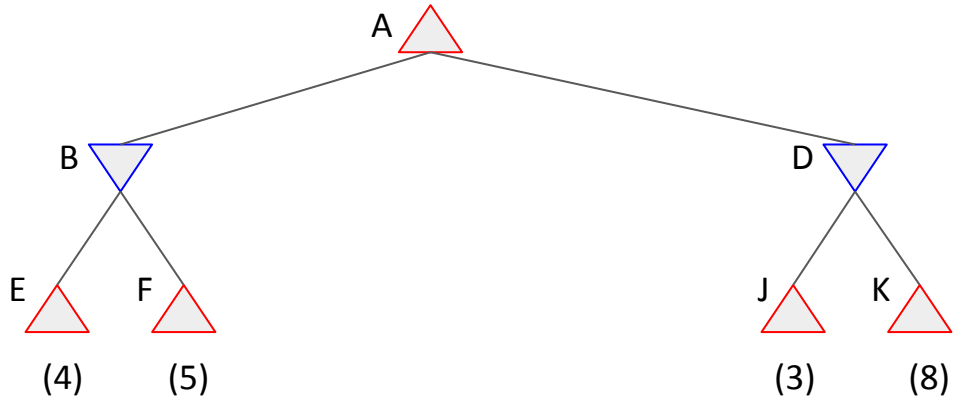
---

function MIN-VALUE($state, \alpha, \beta$) **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) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

5.3 The Alpha-beta pruning algorithm

```
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
```



5.3 The Alpha-beta pruning algorithm

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

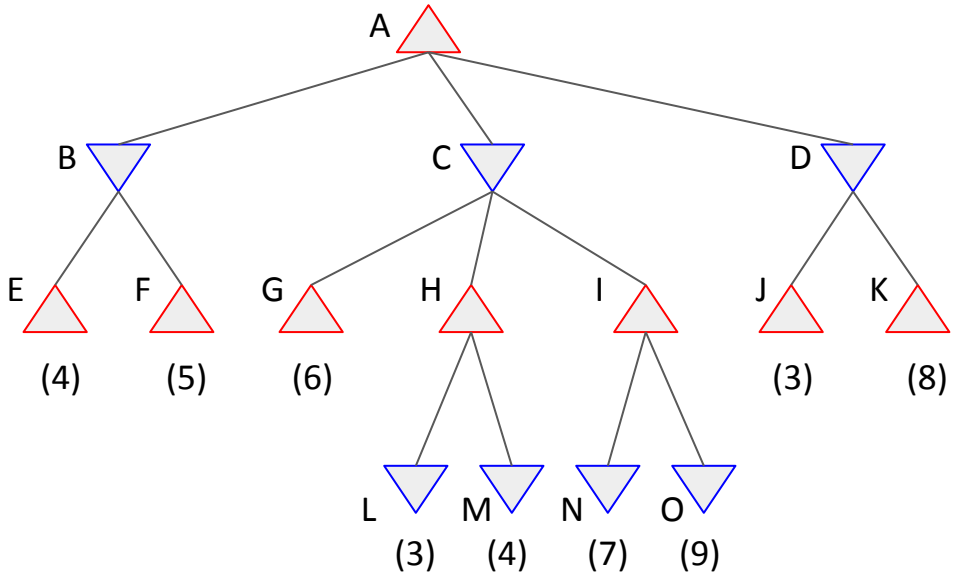
```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
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
```

α = the value of the best node (highest value) we have found so far at any choice point along the path for MAX
β = the value of the best node (lowest value) we have found so far at any choice point along the path for MIN

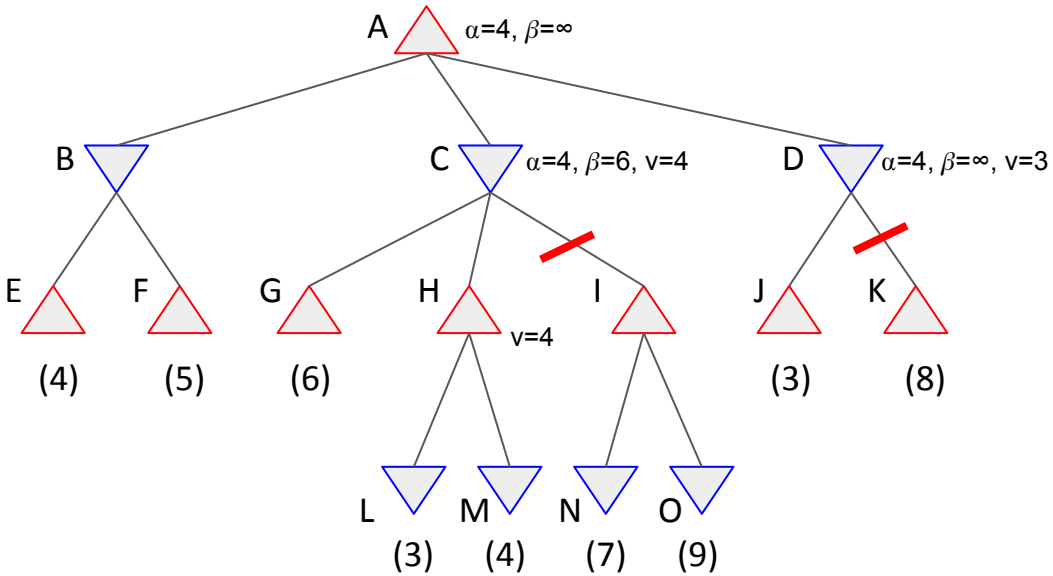
Tips:
(1) Maintain alpha, beta, and v at each non-leaf node
(2) Update v step-by-step

Problem: Calculate the minimax values for the non-leaf nodes in the search tree below. Also, identify the nodes (or sub-tree) that will be pruned. Assume: 1) that MAX plays first at node A followed by MIN, and 2) actions are taken in alphabetical order.



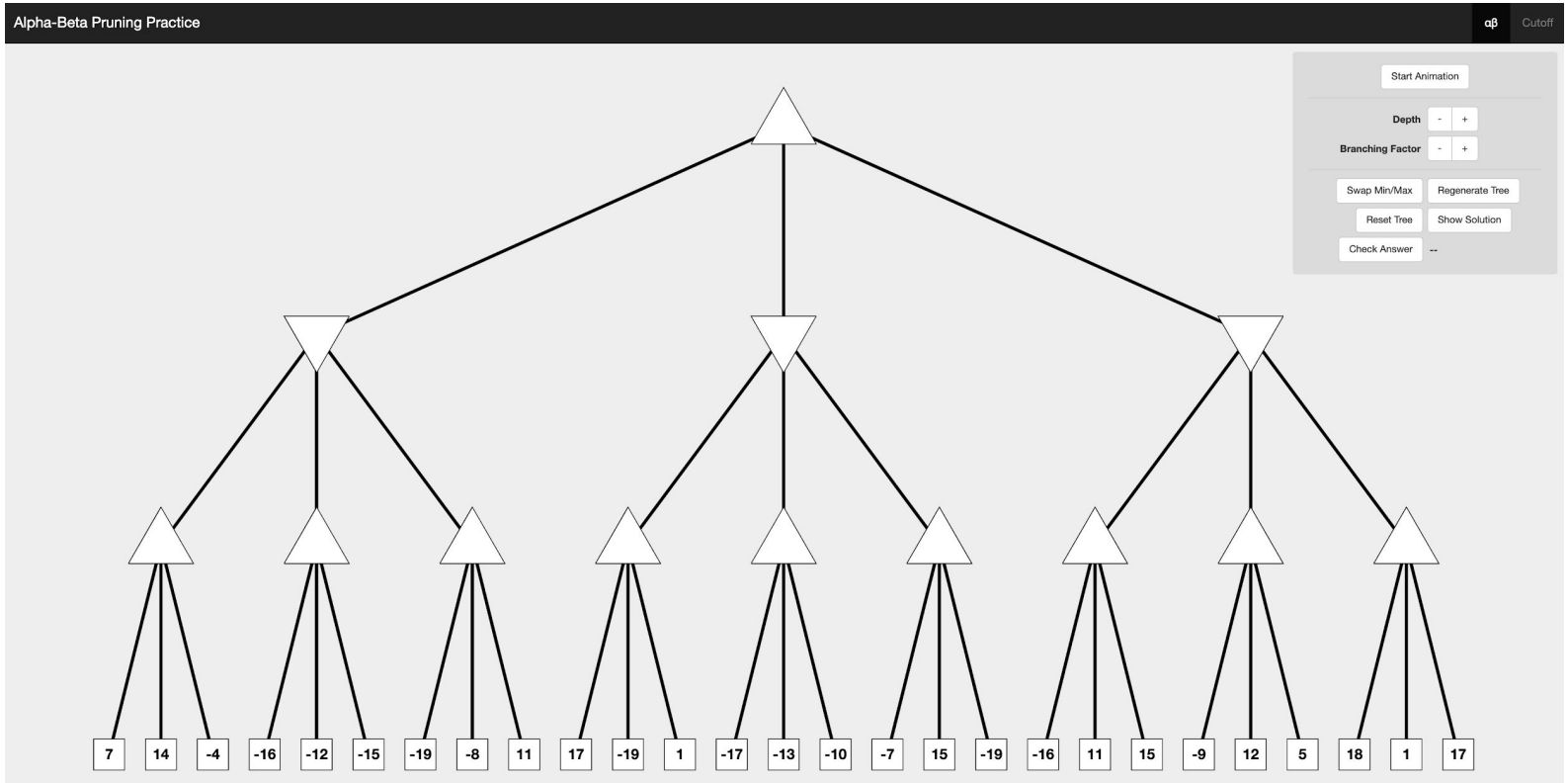
Algorithm will be provided in tests!

5.3 The Alpha-beta pruning algorithm



Practice

http://inst.eecs.berkeley.edu/~cs61b/fa14/ta-materials/apps/ab_tree_practice/



Summary

- A game can be defined by the **initial state** (how the board is set up), the legal **actions in each state**, the **result of each action**, a **terminal test** (which says when the game is over), and a **utility function** that applies to terminal states.
- In two-player zero-sum games with perfect information, the minimax algorithm can select optimal moves by a depth-first enumeration of the game tree.
- The alpha–beta search algorithm **computes the same optimal move as minimax**, but achieves much greater efficiency by eliminating subtrees that are **provably irrelevant**.

Supplementary Slide