Adversarial Search

Game Theory

- Choosing from a **set of rational choices** in a **multiagent situation**
- Dealing with deciding on a given set of options.
- Multi-agent situation- choices affect the choices of the opponent in the game, and their decision affects our choices

Adversarial Search

- *In which we examine the problems that arise when we try to plan ahead in a world where other agents are planning against us*

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?

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:
  - S0: The **initial state**, which specifies how the game is set up at the start
  - PLAYER(s): Defines which player has the move in a state.
  - ACTIONS(s): Returns the set of legal moves in a state.
  - RESULT(s, a): The **transition model**, which defines the result of a move.
  - TERMINAL-TEST(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
  - UTILITY(s, p):
    - A **utility function** (also called an objective function or payoff function),
    - defines the final numeric value for a game that ends in terminal state s for a player p.
    - In tic-tac-toe the outcome is win, loss or draw with value 1, -1, 0

Game tree

- The initial state, ACTIONS function, and RESULT function define the game tree for the game
  - a tree where the nodes are game states and the edges are moves.
- 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

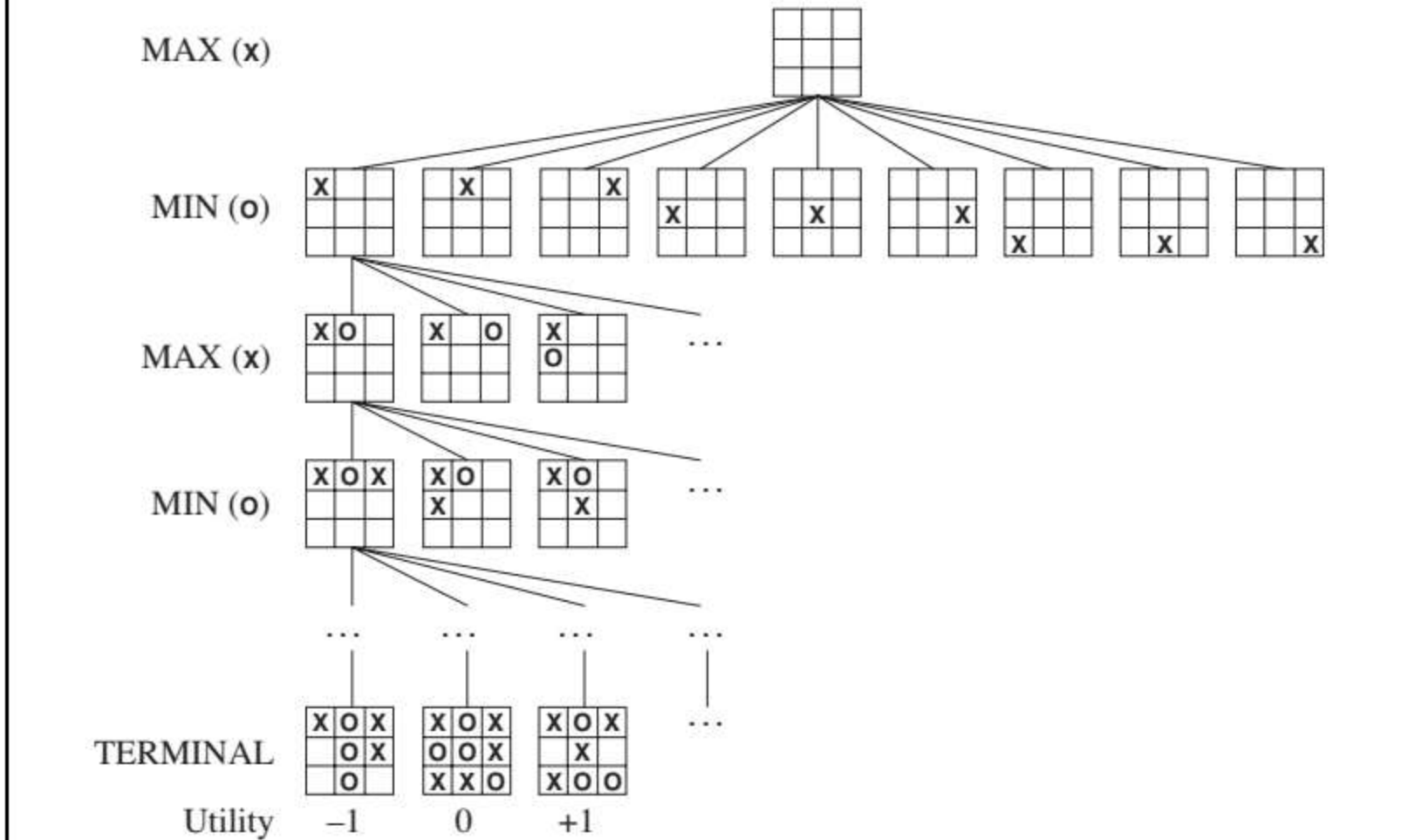


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities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

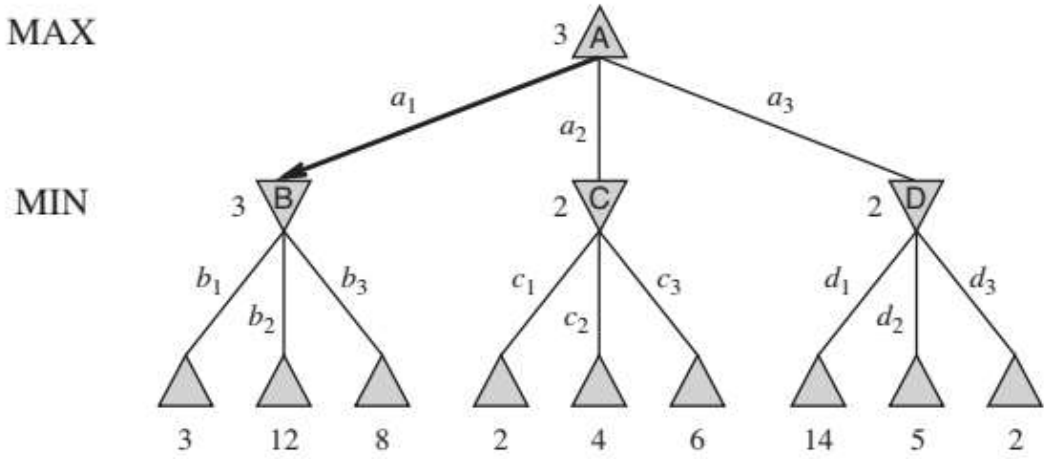
Optimal decisions in games

- In a normal search problem:
  - the optimal solution would be a sequence of actions leading to a goal state—a terminal state that is a win.
- In adversarial search,
  - MIN Interferes the sequence of actions
  - Strategy for MAX
    - specifies MAX's move in the initial state,
    - Observes every possible response by MIN
    - Speciy moves in response
    - And so on..
  - The 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 minimax value

- Consider a reduced tic-ta-toe game
  - 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

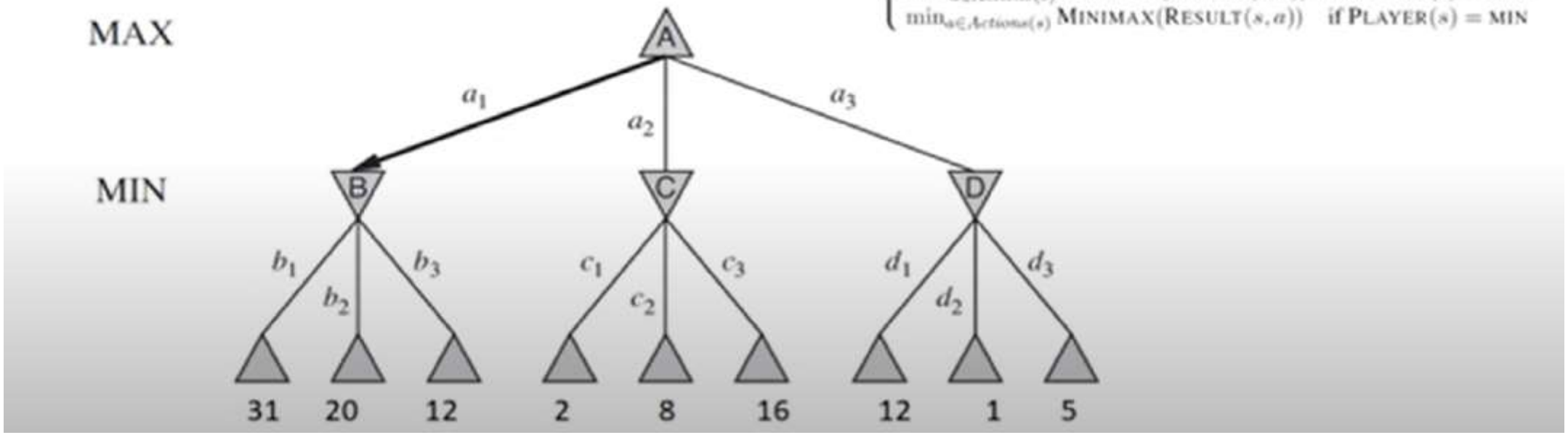
$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$



Practice problem

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)`.

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



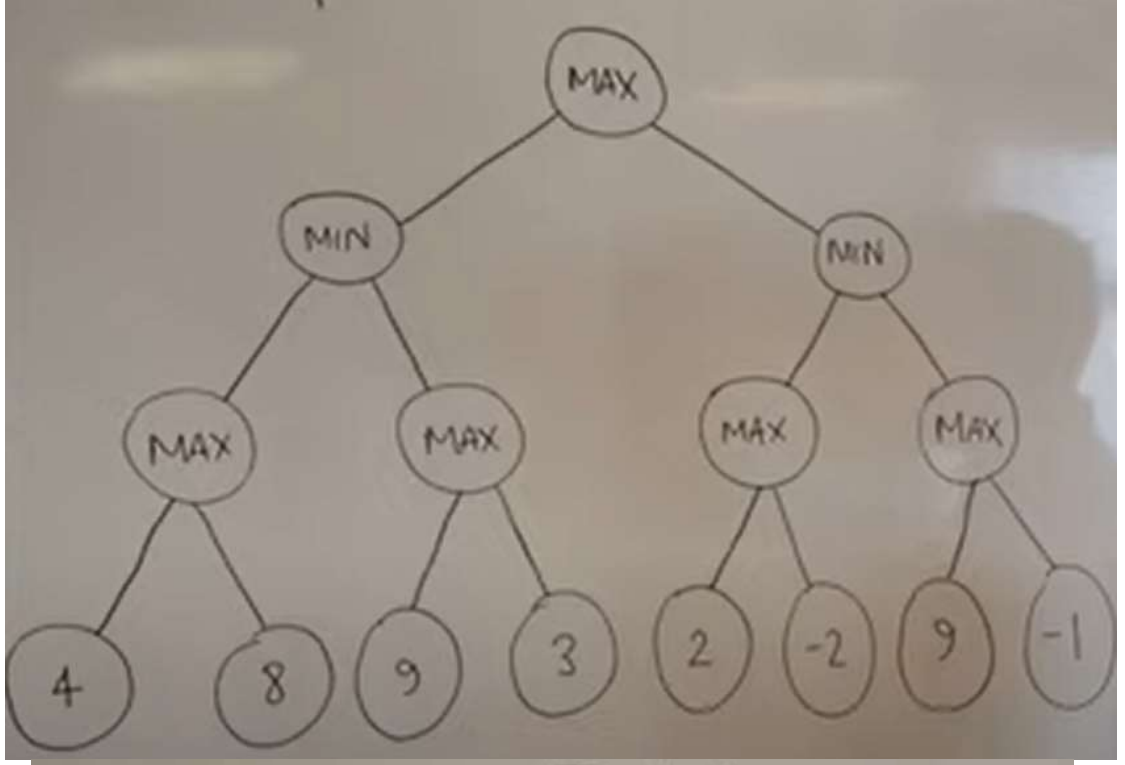
The minimax algorithm

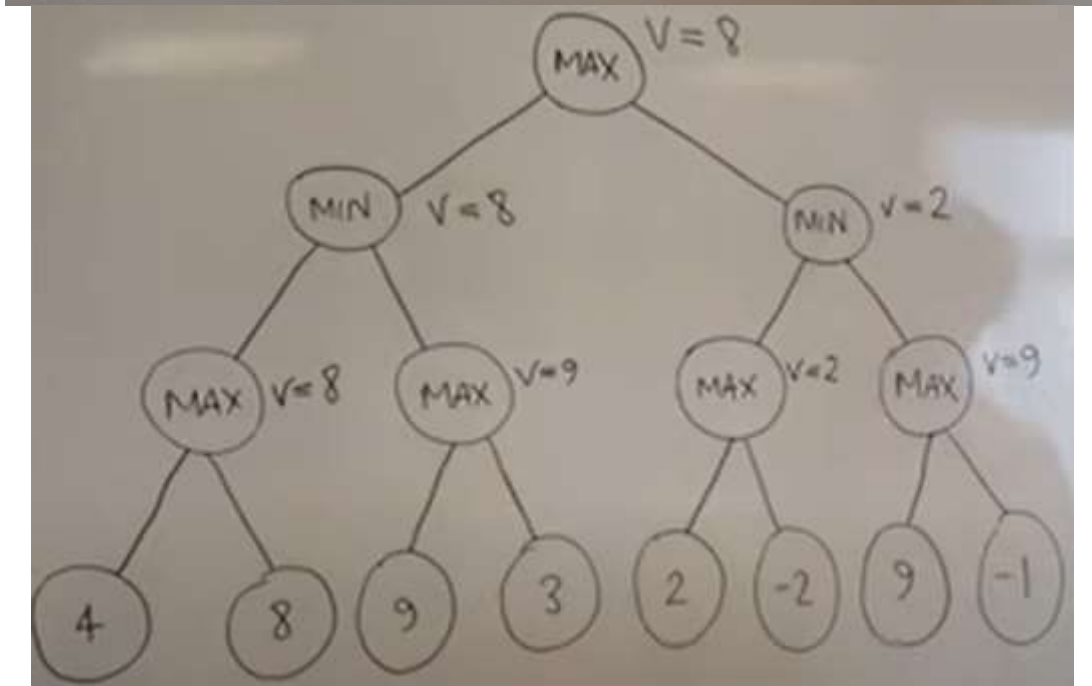
```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

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
```

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 $f(a)$.

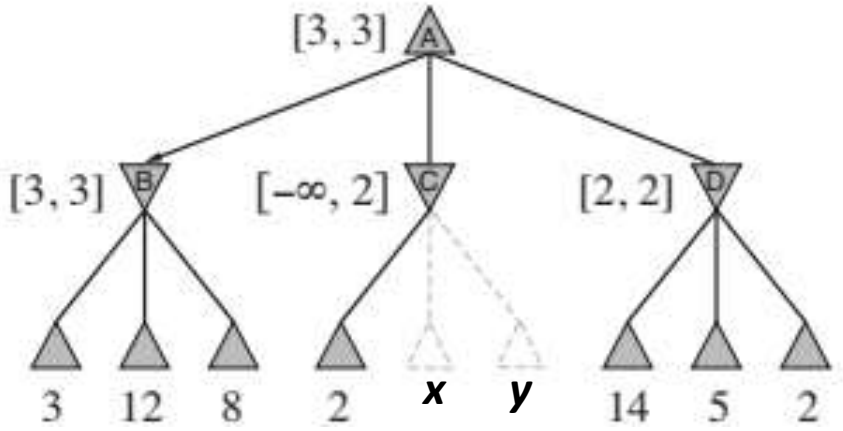




Do we need to computer all minimax values?

- 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):

$$
\begin{aligned}
\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3,z,2) \qquad \text{where } z = \min(2,x,y) \le 2 \\
&= 3.
\end{aligned}
$$



- Is the minimax value dependent on x,y

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.

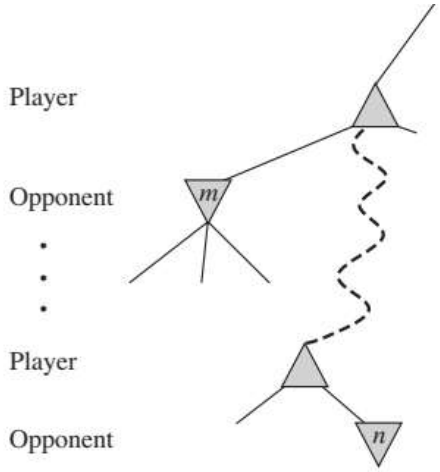


Figure 5.6 The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

Alpha-beta search

```
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).