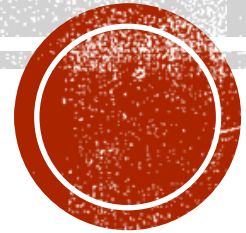
# ADVERSARIAL SEARCH



Prepared By:

Dipanita Saha

Assistant Professor

Institute of Information Technology(IIT)

Noakhali Science and Technology University

# GAMES

- **Competitive** environments, in which the agents' goals are in conflict, giving rise to **adversarial search** problems—often known as **games**.

- Mathematical **game theory**, views any multiagent environment as a game

- In AI, the most common games are of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, **zero-sum games** of **perfect information** (such as chess).

- this means deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite.

- For example, if one player wins a game of chess, the other player necessarily loses.

- a game is easy to represent, and agents are usually restricted to a small number of actions whose outcomes are defined by precise rules.

# Why would game playing be a good problem for AI research?

- game playing is non-trivial
  - players need “human-like” intelligence
  - games can be very complex (e.g. chess, go)
  - requires decision making within limited time

- games often are:
  - well-defined and repeatable
  - easy to represent
  - fully observable and limited environments

- can directly compare humans and computers

- A game can be formally defined as a kind of search problem with the following elements:

1) • *S0*: The **initial state**, which specifies how the game is set up at the start.
2) • *PLAYER(s)*: Defines which player has the move in a state.
3) • *ACTIONS(s)*: Returns the set of legal moves in a state.
4) • *RESULT(s, a):* The **transition model**, which defines the result of a move.
5) • *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**.
6) • *UTILITY(s, p):* A **utility function** , defines the final numeric value for a game that ends in terminal state s for a player p. In chess, the outcome is a win, loss, or draw, with values +1, 0, or 1/2 .

- A **zero-sum game** is defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has payoff of either 0 + 1, 1 + 0 or ½ + 1/2 .
- 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.

- Figure shows part of the game tree for tic-tac-toe (noughts and crosses).
- From the initial state, MAX has nine possible moves.

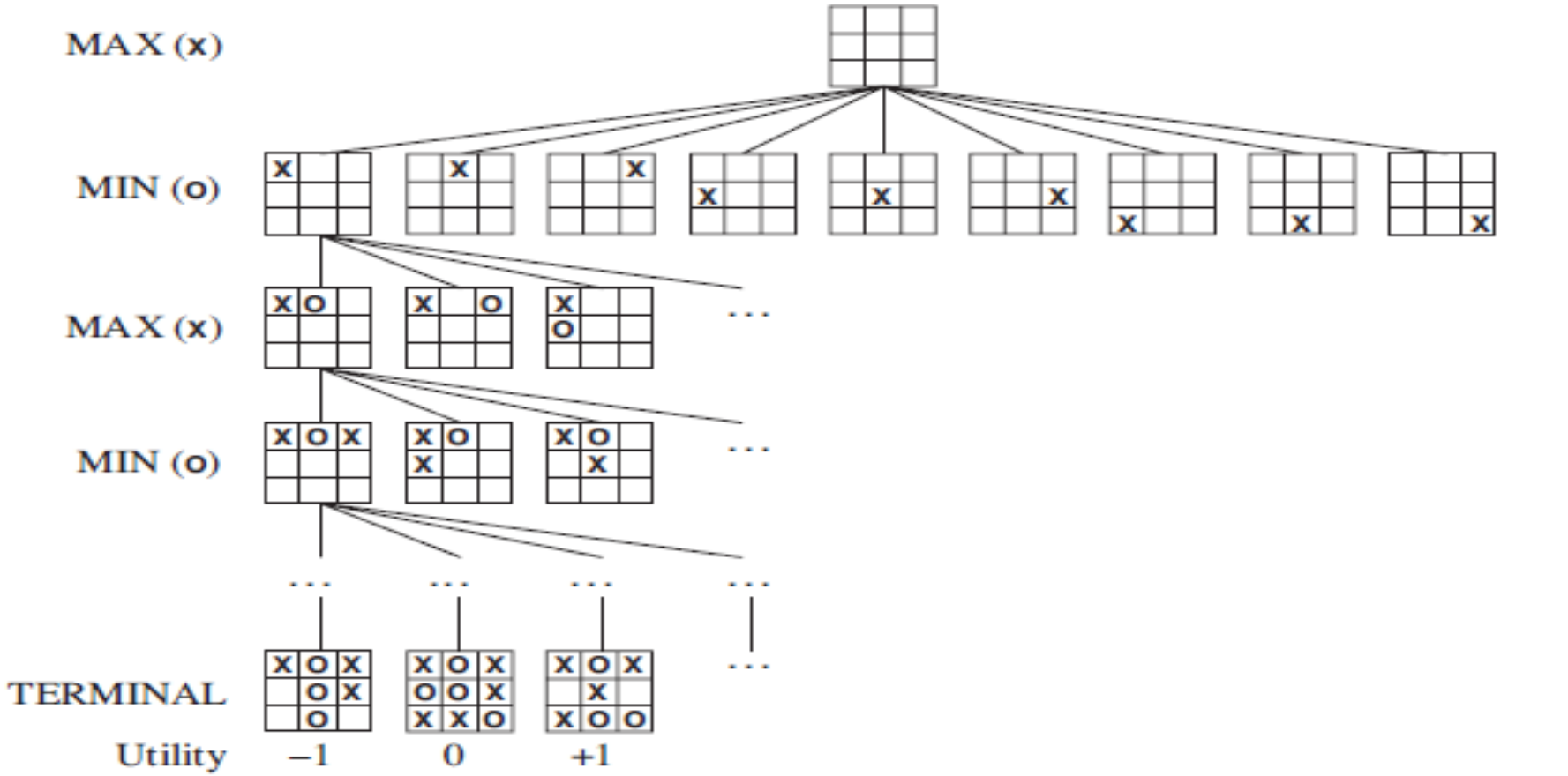



**Figure 5.1** A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities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

# OPTIMAL DECISIONS IN GAMES

- 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 has something to say about it. MAX therefore must find a contingent **strategy**, which specifies MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by MIN, then MAX's moves in the states resulting from every possible response by MIN to *those* moves, and so on.
- an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

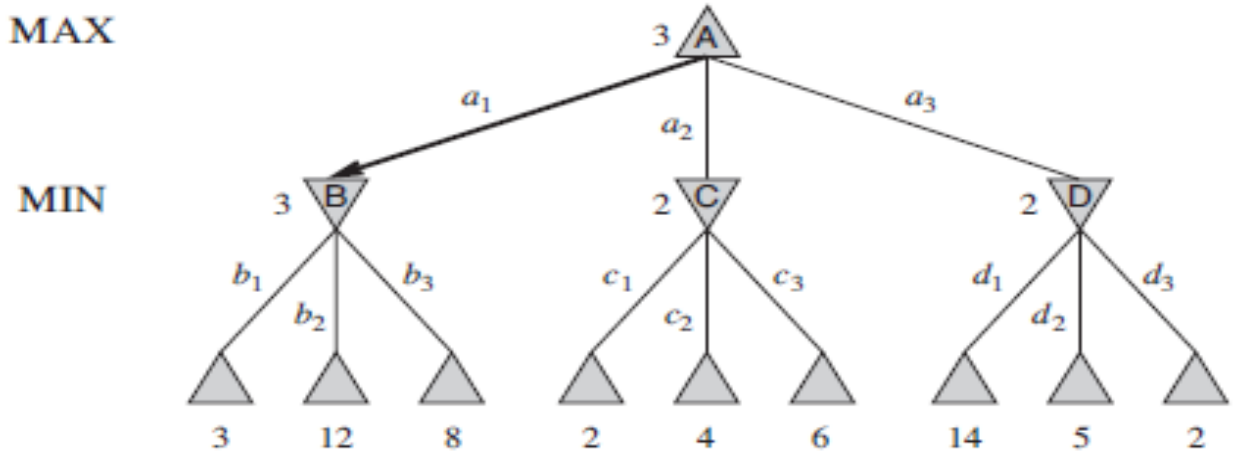



**Figure 5.2** A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is $a_1$, because it leads to the 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.

- The possible moves for MAX at the root node are labeled a1, a2, and a3. The possible replies to a1 for MIN are b1, b2, b3, and so on.
- This particular game ends after one move each by MAX and MIN.
- The utilities of PLY the terminal states in this game range from 2 to 14.
- Given a game tree, the optimal strategy can be determined from the **minimax value** of each node, which we write as MINIMAX(n)
- . The minimax value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally* from there to the end of the game.
- MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value.

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

- The first MIN node, labeled B, has three successor states with values 3, 12, and 8, so its minimax value is 3.
- Similarly, the other two MIN nodes have minimax value 2.
- The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3.
- the **minimax decision** at the root: action a1 is the optimal choice for MAX because it leads to the state with the highest minimax value.

# The minimax algorithm

- The **minimax algorithm** computes the minimax decision from the current state.
- It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations.
- 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.
- For example, in Figure , the algorithm first recurses down to the three bottom left nodes and uses the UTILITY function on them to discover that their values are 3, 12, and 8, respectively.
- Then it takes the minimum of these values, 3, and returns it as the backed up value of node B.
- Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.
- The minimax algorithm performs a complete depth-first exploration of the game tree.
- the time complexity of the minimax algorithm is O(bm).
- The space complexity is O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time.

## Minimax: Algorithm for Static Board Evaluator (SBE)

✱*A static board evaluation function estimates how good a board configuration is for the computer.*

- it reflects the computer's chances of winning from that state
- it must be easy to calculate from the board configuration

- For Example, Chess:

$$SBE = \alpha * materialBalance + \beta * centerControl + \gamma * ...$$

*material balance = Value of white pieces - Value of black pieces*
*(pawn = 1, rook = 5, queen = 9, etc).*

- The same as direct minimax, except
  - only goes to depth ***m***
  - estimates non-terminal states using SBE function
- How would this algorithm perform at chess?
  - if could look ahead ~4 pairs of moves (i.e. 8 ply) would be consistently beaten by average players
  - if could look ahead ~8 pairs as done in typical pc, is as good as human master

```
int minimax (Node s, int depth, int limit) {
  if (isTerminal(s) || depth == limit) //base case
    return(staticEvaluation(s));
  else {
    Vector v = new Vector();
    //do minimax on successors of s and save their values
    while (s.hasMoreSuccessors())
      v.addElement(minimax(s.getNextSuccessor(),
                                          depth+1, limit));
    if (isComputersTurn(s))
      return maxOf(v); //computer's move returns max of kids
    else
      return minOf(v); //opponent's move returns min of kids
  }
}
```

## Optimal decisions in multiplayer games

- Let us examine how to extend the minimax idea to multiplayer games.
- First, we need to replace the single value for each node with a *vector* of values.
- For example, in a three-player game with players A, B, and C, a vector vA, vB, vC is associated with each node.
- For terminal states, this vector gives the utility of the state from each player's viewpoint.
- The simplest way to implement this is to have the UTILITY function return a vector of utilities.
- Now we have to consider nonterminal states. Consider the node marked X in the game tree shown in Figure .

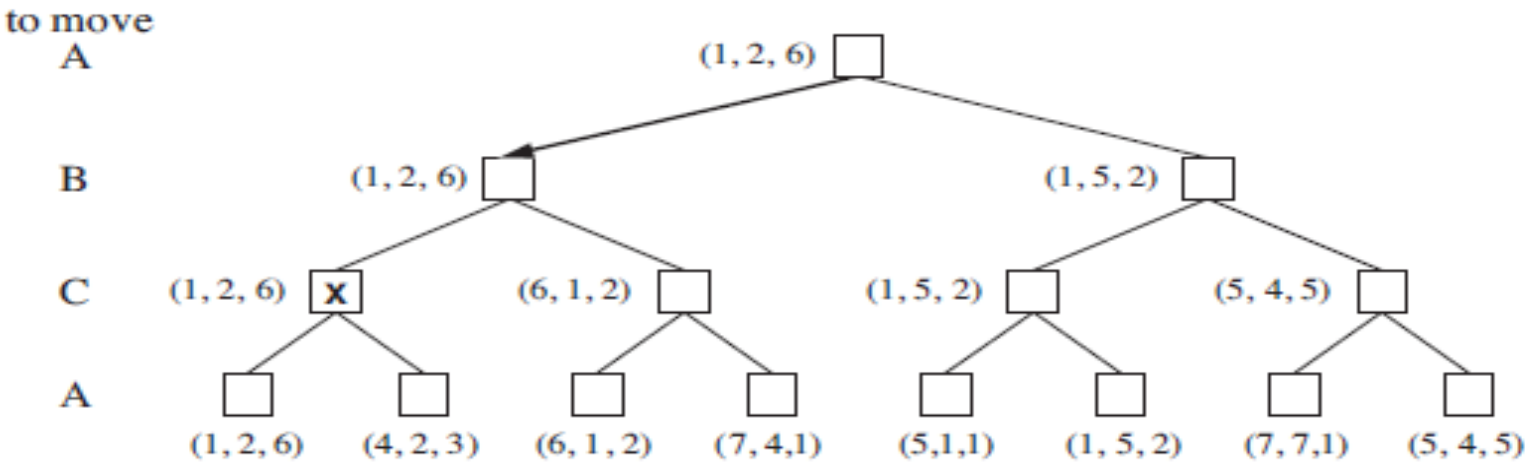



**Figure 5.4** The first three plies of a game tree with three players ($A$, $B$, $C$). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

- In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors vA =1, vB =2, vC =6 and vA =4, vB =2, vC =3.
- Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to a terminal state with utilities vA =1, vB =2, vC =6.
- The backed-up value of a node n is always the utility vector of the successor state with the highest value for the player choosing at n.
- Multiplayer games usually involve **alliances (**are made and broken as the game proceeds**)**, whether formal or informal, among the players.
- How are we to understand such behavior? Are alliances a natural consequence of optimal strategies for each player in a multiplayer game?
- For example suppose A and B are in weak positions and C is in a stronger position.
- Then it is often optimal for both A and B to attack C rather than each other, lest C destroy each of them individually.
- In this way, collaboration emerges from purely selfish behavior.
- as soon as C weakens under the joint onslaught, the alliance loses its value, and either A or B could violate the agreement.
- long-term disadvantage of being perceived as untrustworthy.

# ALPHA–BETA PRUNING

- The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.
- The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree.
- The particular technique we examine is called **alpha–beta pruning**.
- When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.
- While doing DFS of game tree, keep track of:
  - **alpha** at maximizing levels (computer's move)
    - highest SBE value seen so far (initialize to -infinity)
    - is *lower* bound on state's evaluation
  - **beta** at minimizing levels (opponent's move)
    - lowest SBE value seen so far (initialize to +infinity)
    - is *higher* bound on state's evaluation

- Beta cutoff pruning occurs when maximizing if *child's alpha ≥ parent's beta*
- Why stop expanding children?

  opponent won't allow computer to take this move

- Alpha cutoff pruning occurs when minimizing if *parent's alpha ≥ child's beta*
- Why stop expanding children?

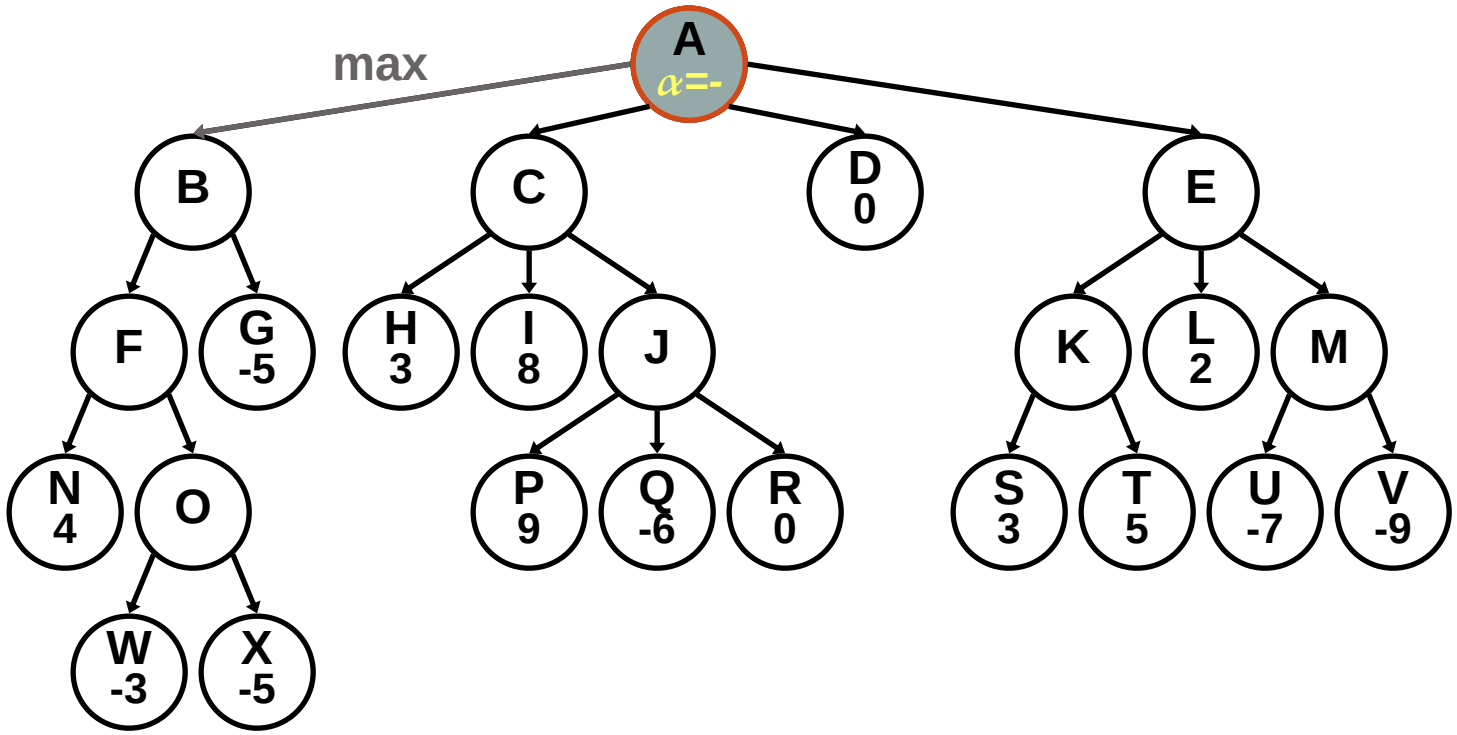
  computer has a better move than this.

**Alpha-Beta Search Example**

# ALPHA-BETA SEARCH EXAMPLE

`minimax(A,0,4)` alpha initialized to -infinity

**Expand A?** Yes since there are successors, no cutoff test for root



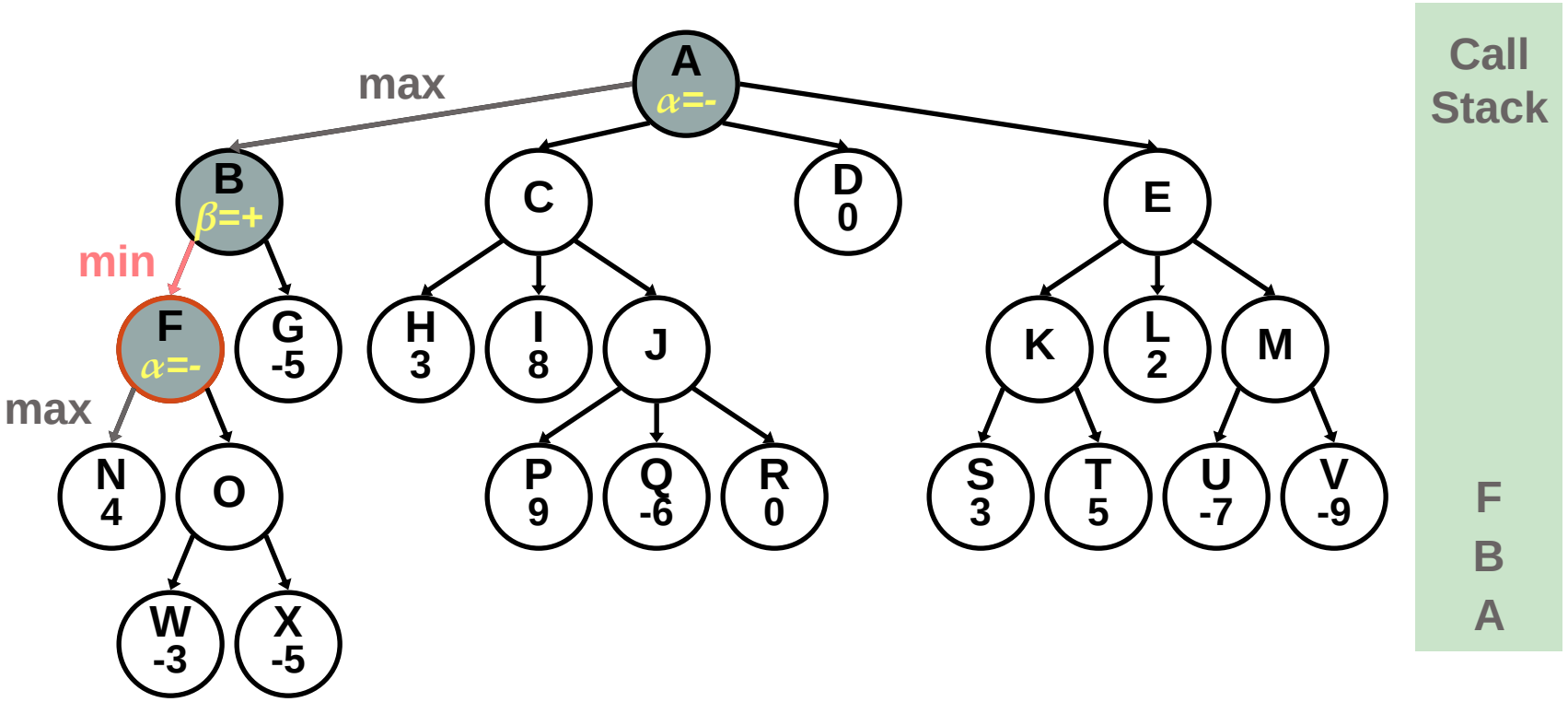

Call Stack

A

# ALPHA-BETA SEARCH EXAMPLE

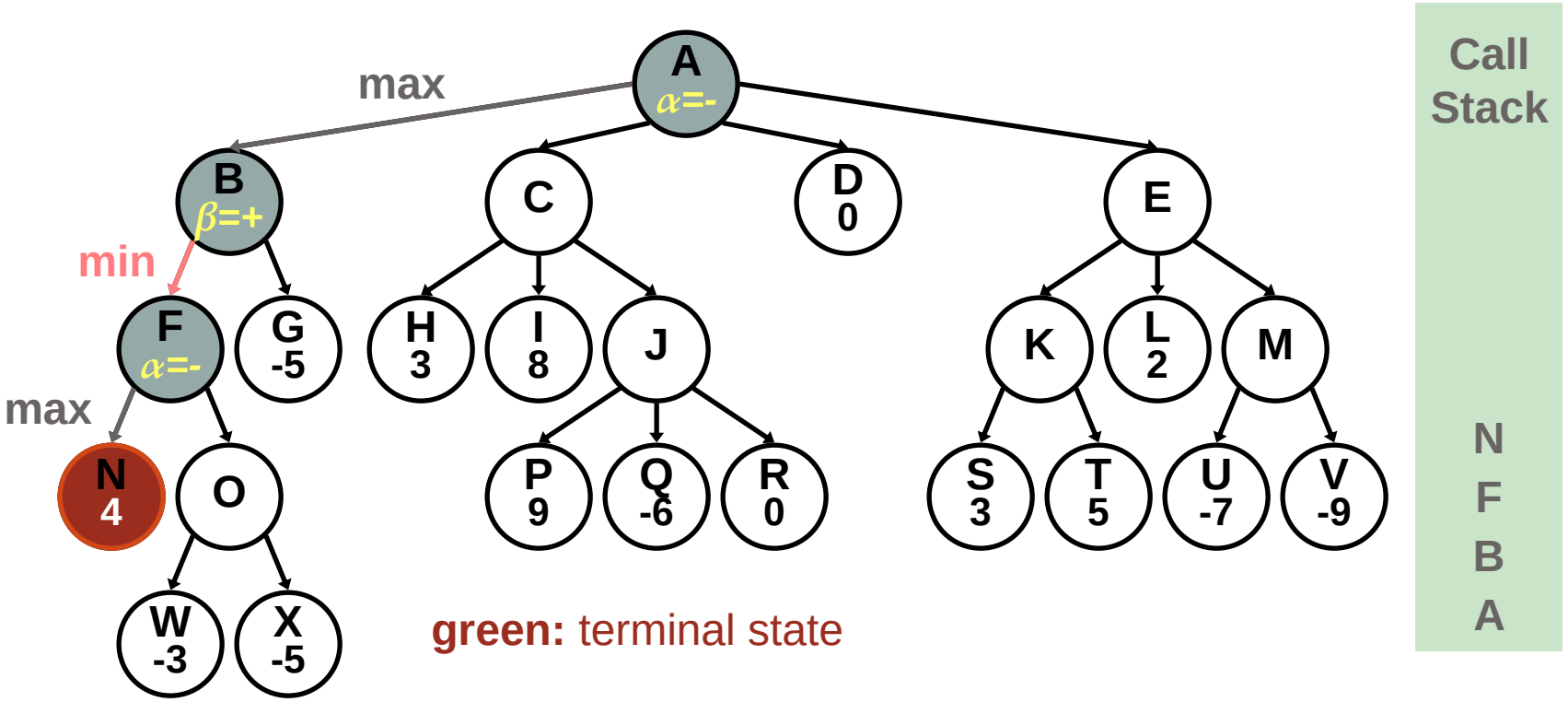
`minimax(F,2,4)` alpha initialized to -infinity

**Expand F?** Yes since F's alpha >= B's beta is false, no beta cutoff



# ALPHA-BETA SEARCH EXAMPLE

`minimax(N,3,4)` evaluate and return SBE value

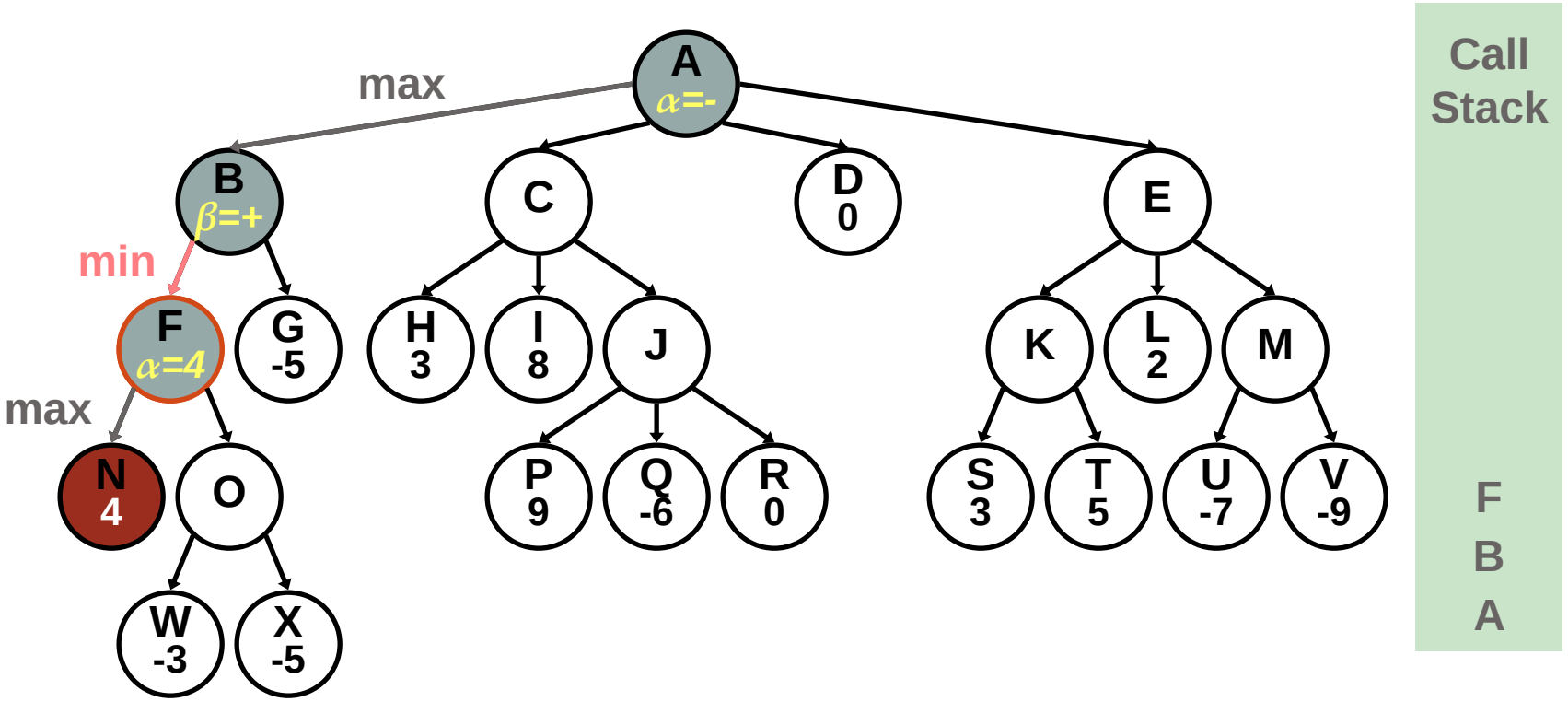


# ALPHA-BETA SEARCH EXAMPLE

**back to** `minimax(F,2,4)`

alpha = 4, since 4 >= -infinity (maximizing)

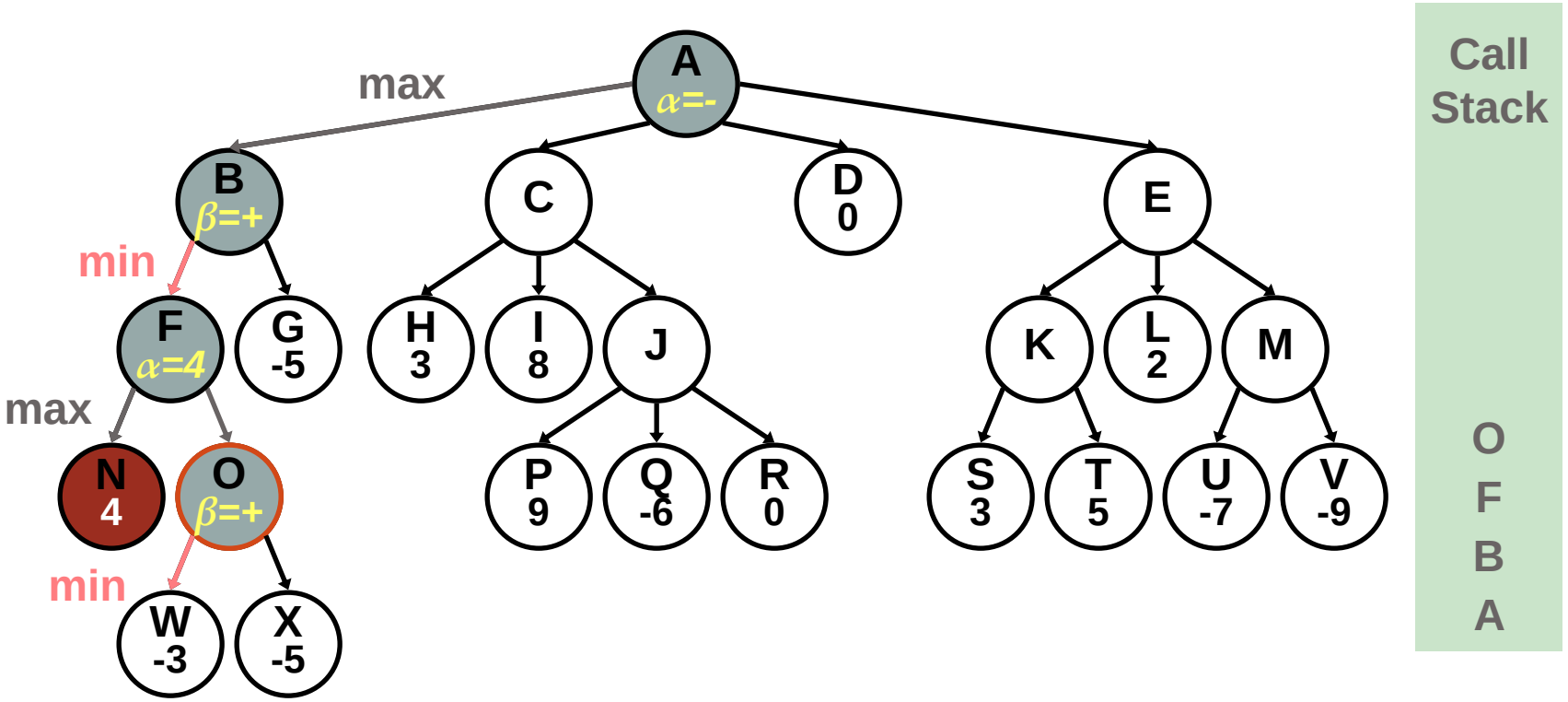
**Keep expanding F?** Yes since F's alpha >= B's beta is false, no beta cutoff



# ALPHA-BETA SEARCH EXAMPLE

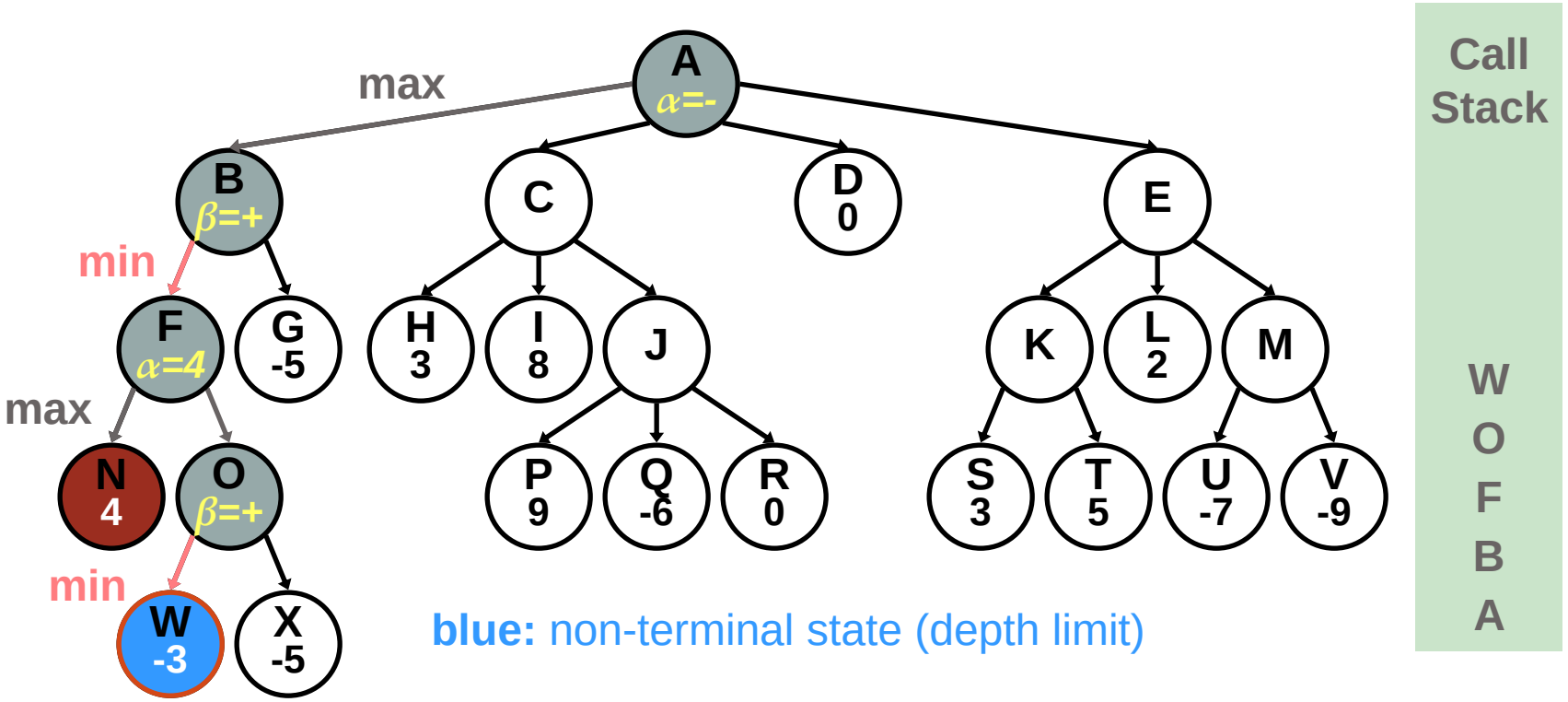
`minimax(O,3,4)` beta initialized to +infinity

**Expand O?** Yes since F's alpha >= O's beta is false, no alpha cutoff



# ALPHA-BETA SEARCH EXAMPLE

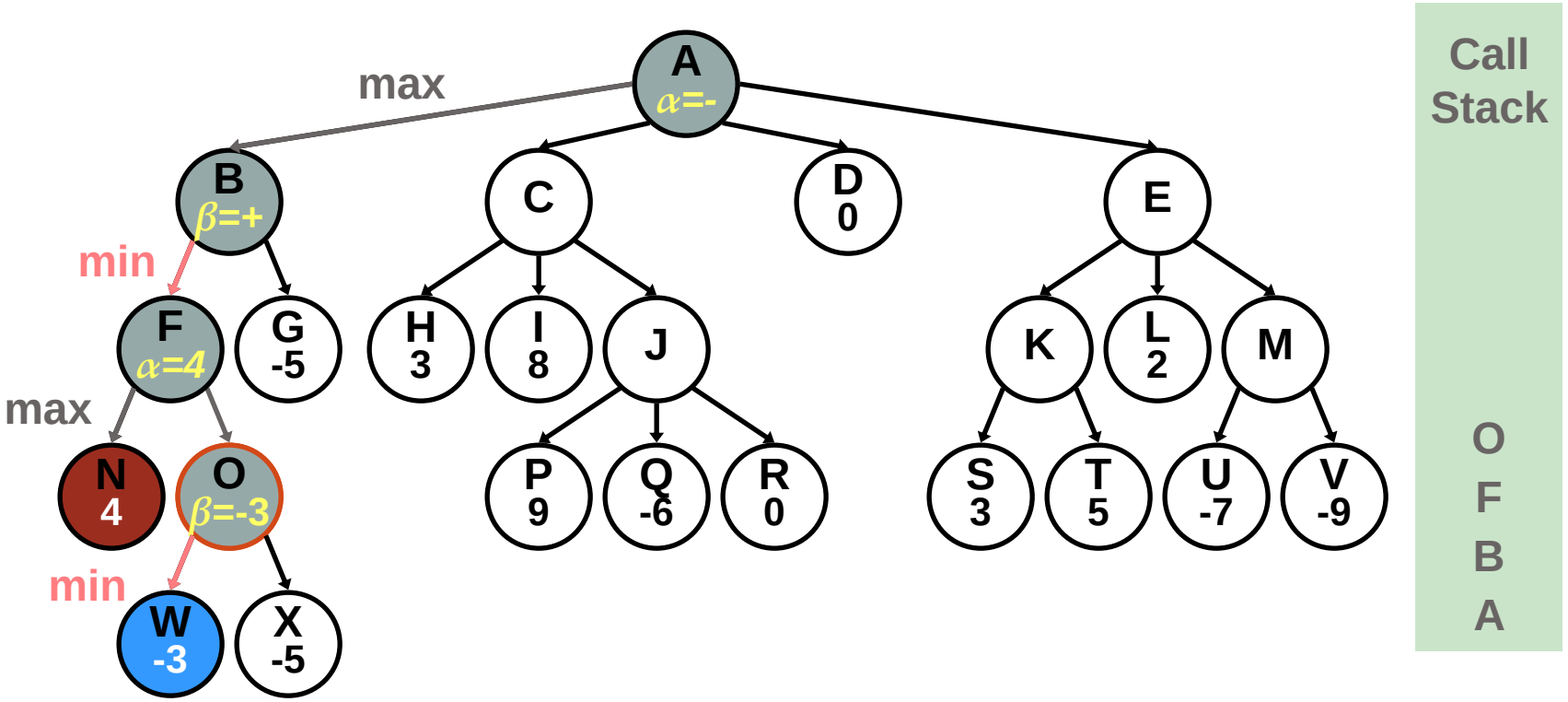
`minimax(W,4,4)` evaluate and return SBE value



**back to** `minimax(O,3,4)`

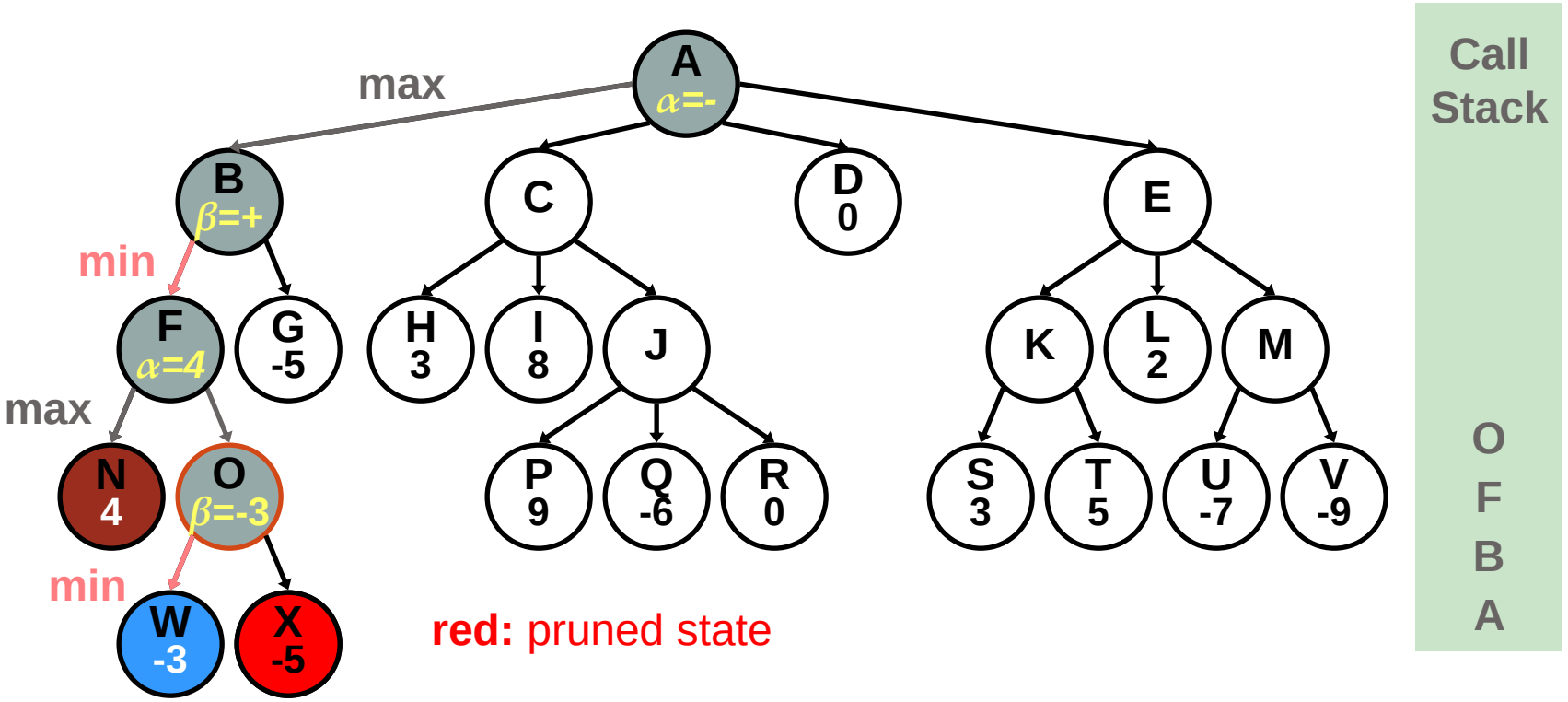
beta = -3, since -3 <= +infinity (minimizing)

**Keep expanding O?** No since F's alpha >= O's beta is true: **alpha cutoff**



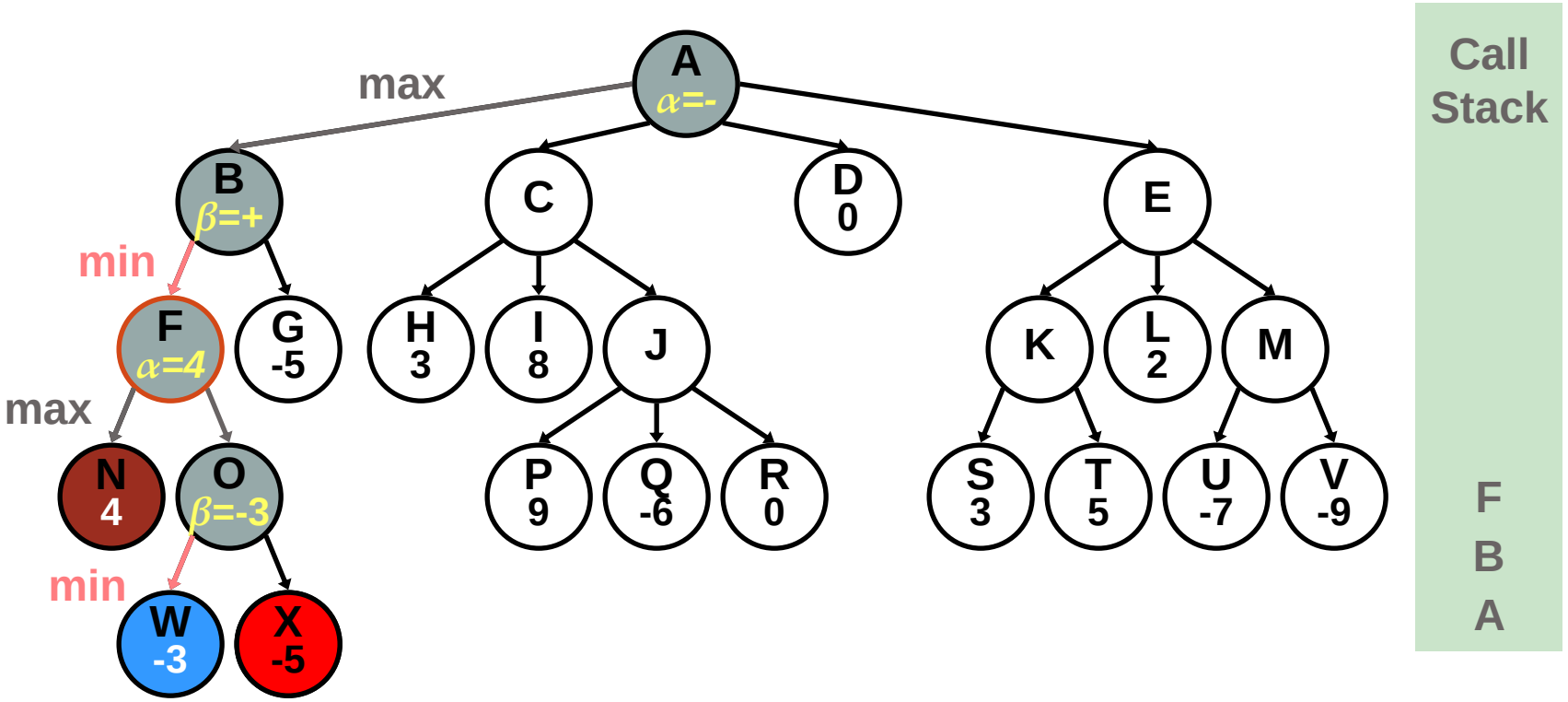
☞ Why?

Smart opponent will choose W or worse, thus O's upper bound is –3.
Computer already has better move at N.



**back to** `minimax(F,2,4)` alpha doesn't change, since -3 < 4 (maximizing)

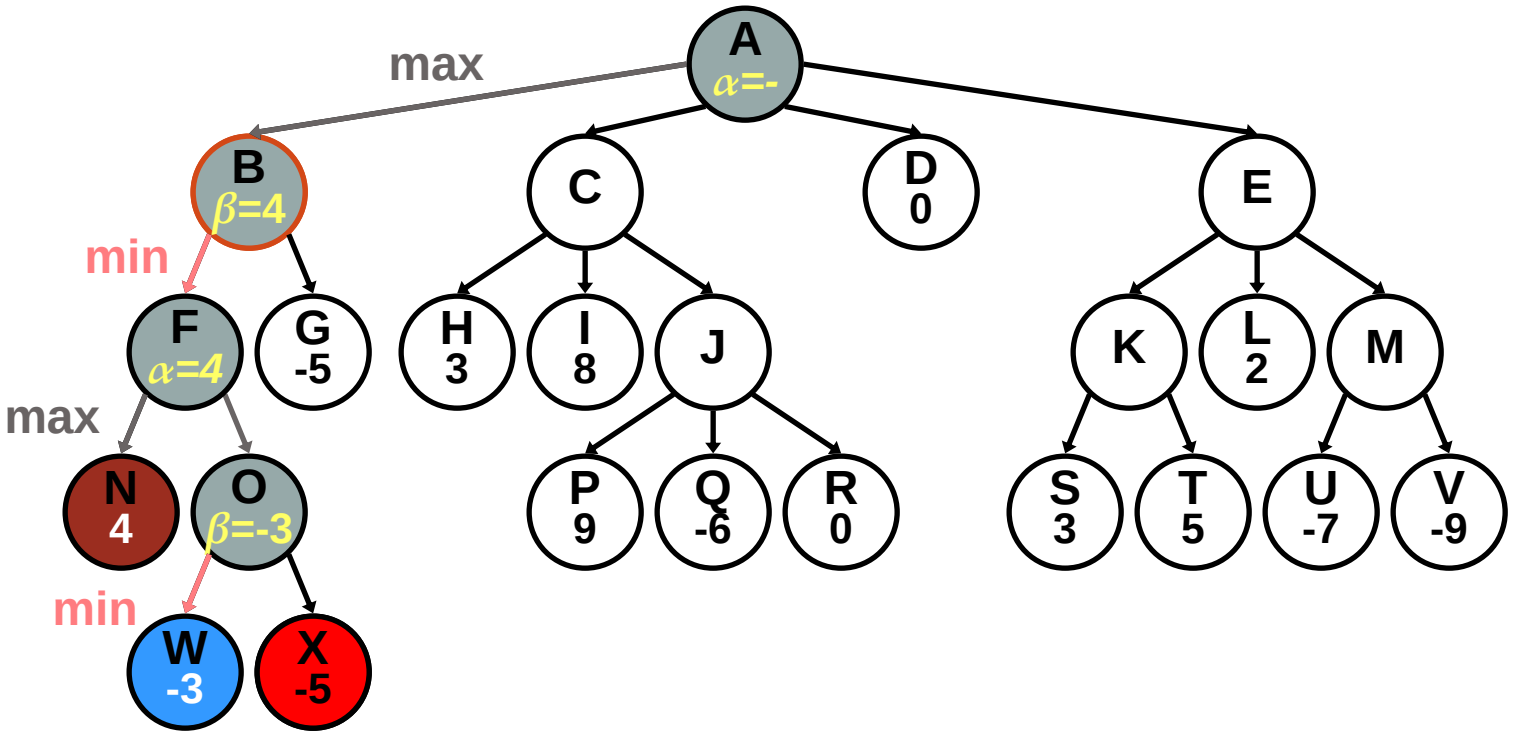
**Keep expanding F?** No since no more successors for F



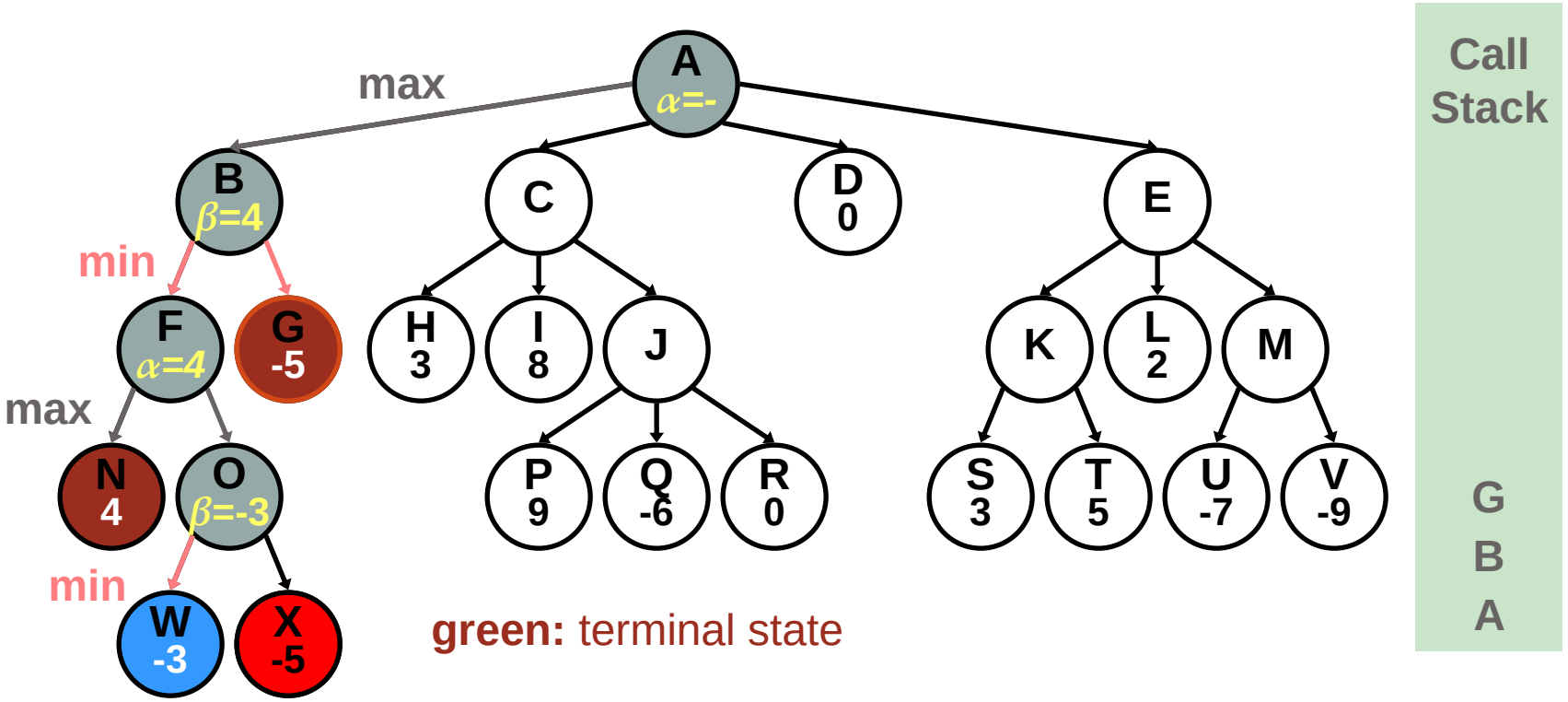
**back to** `minimax(B,1,4)`

beta = 4, since 4 <= +infinity (minimizing)

**Keep expanding B?** Yes since A's alpha >= B's beta is false, no alpha cutoff



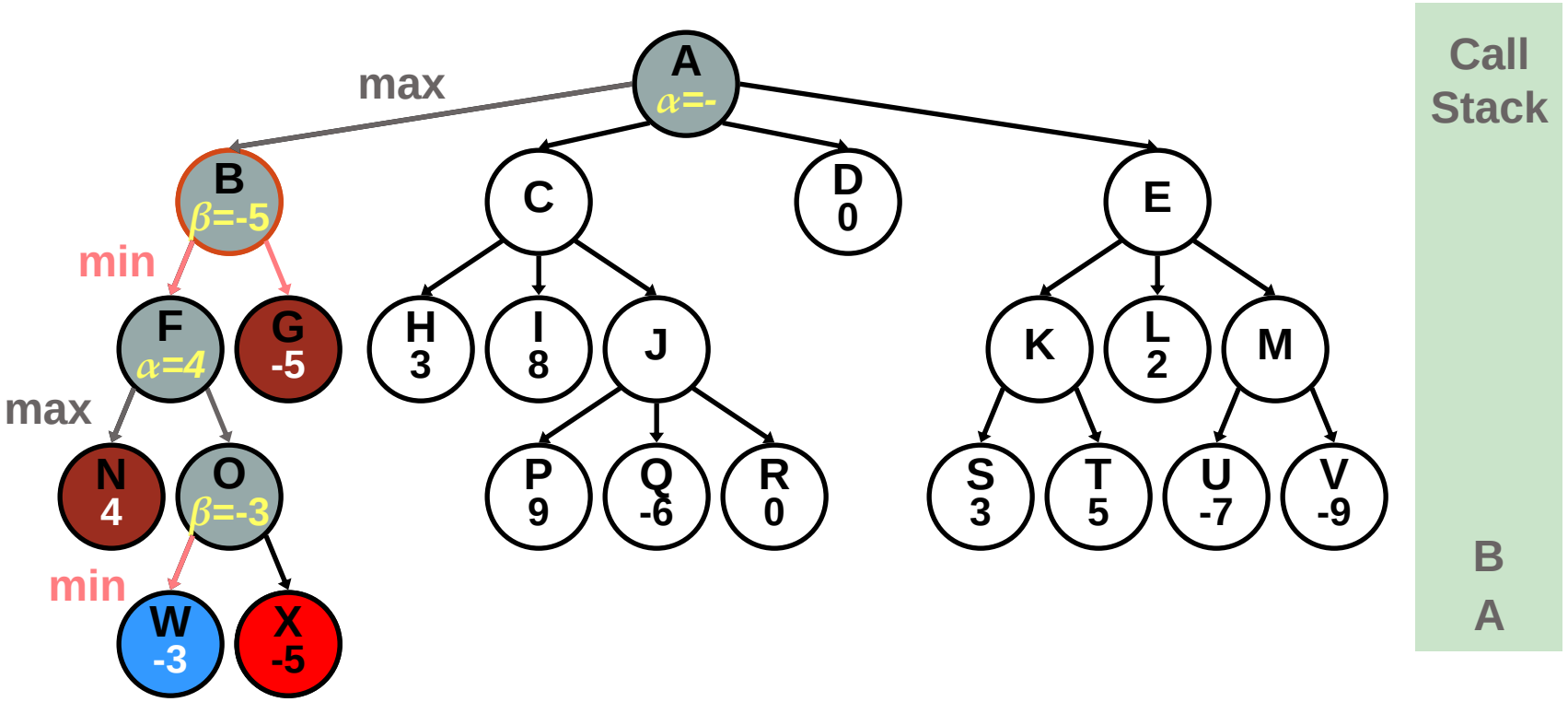
`minimax(G,2,4)` evaluate and return SBE value



**back to** `minimax(B,1,4)`

beta = -5, since -5 <= 4 (minimizing)

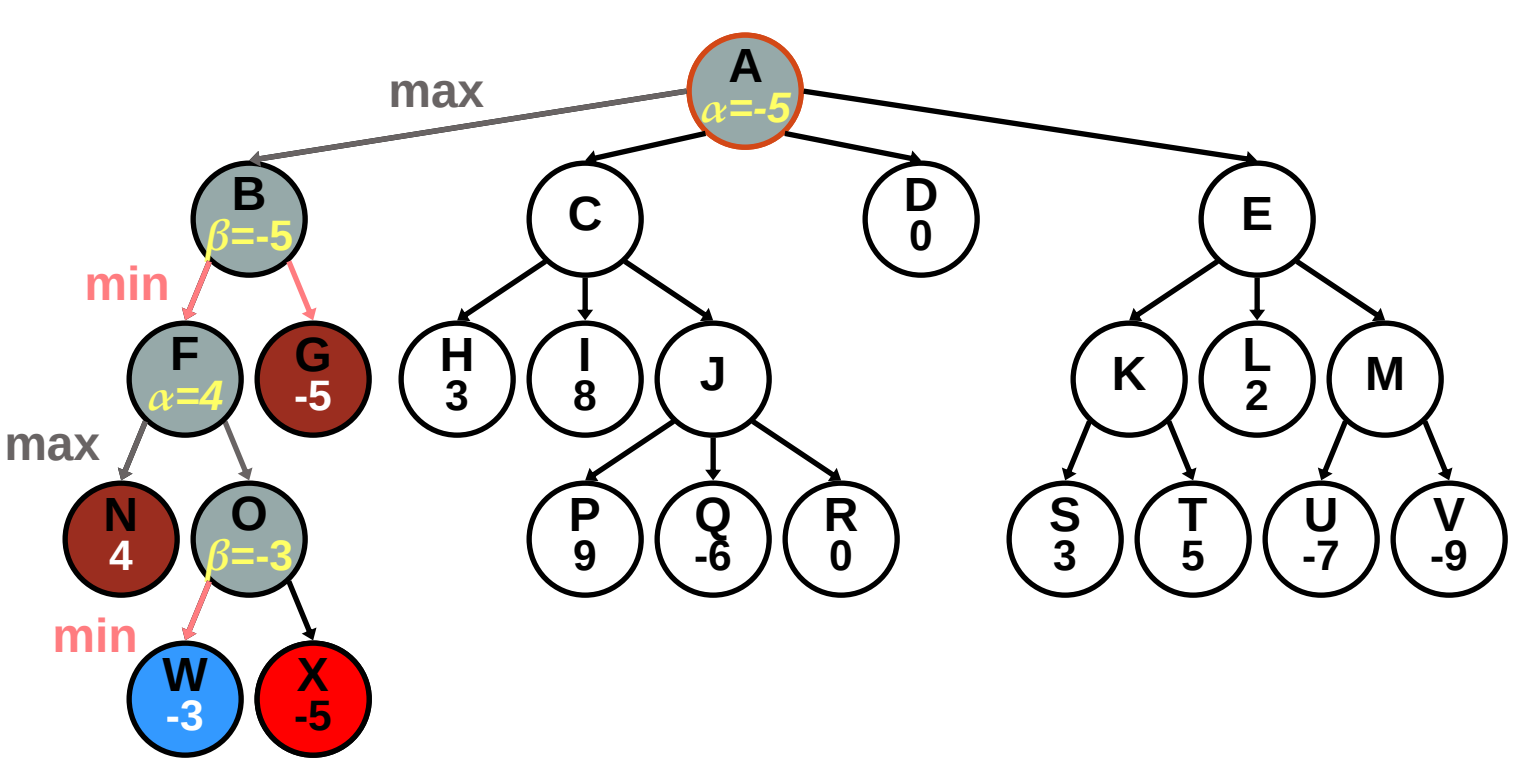
**Keep expanding B?** No since no more successors for B



**back to** `minimax(A,0,4)`

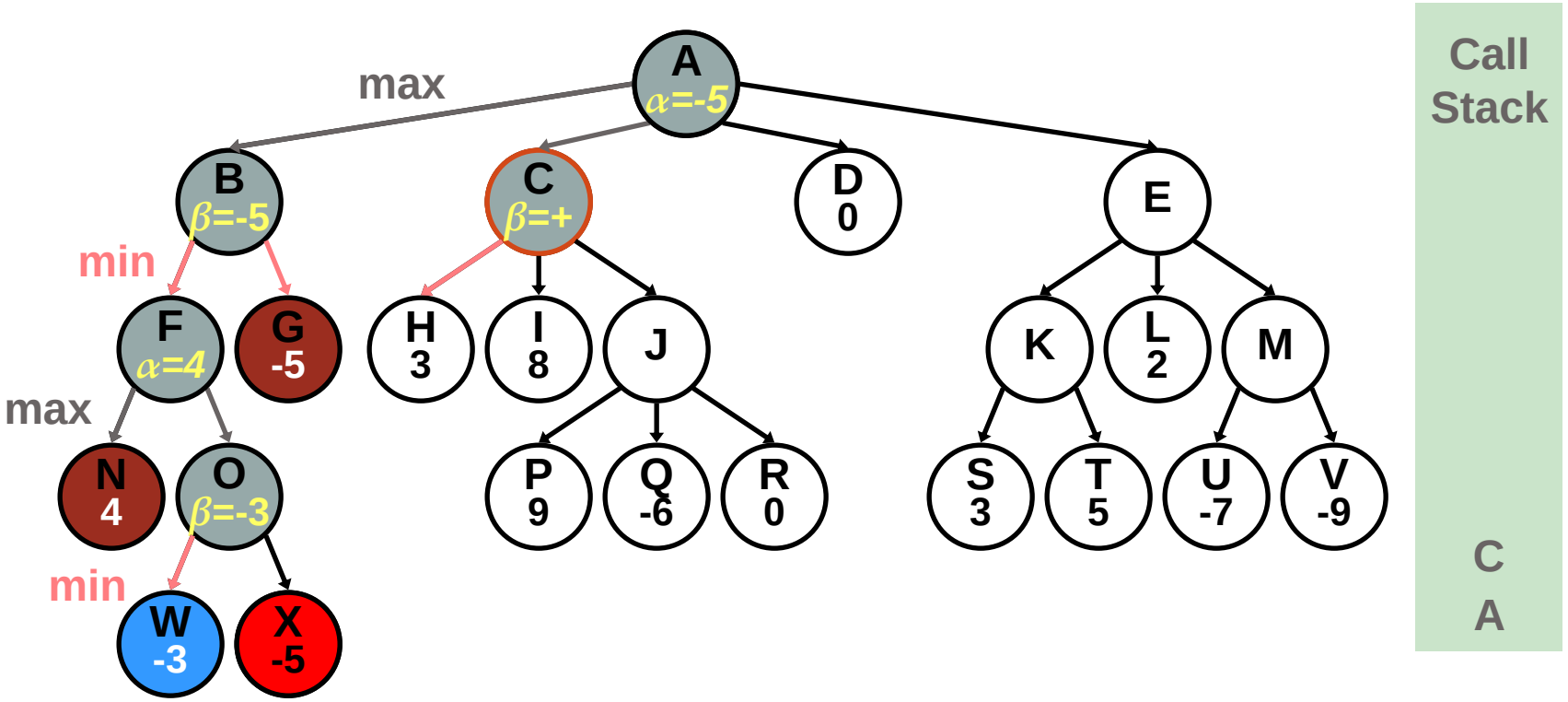
alpha = -5, since -5 >= -infinity (maximizing)

**Keep expanding A?** Yes since there are more successors, no cutoff test

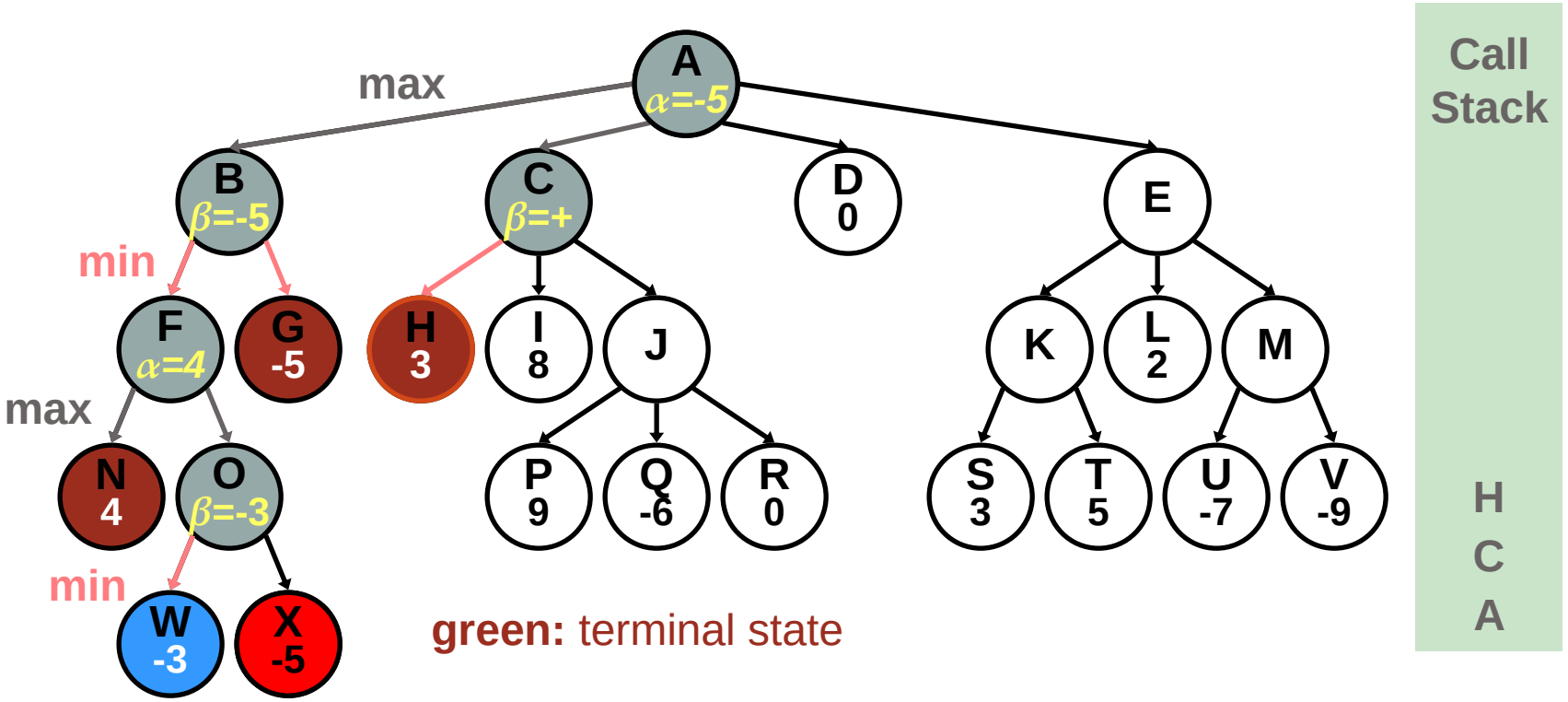


`minimax(C,1,4)` beta initialized to +infinity

**Expand C?** Yes since A's alpha >= C's beta is false, no alpha cutoff



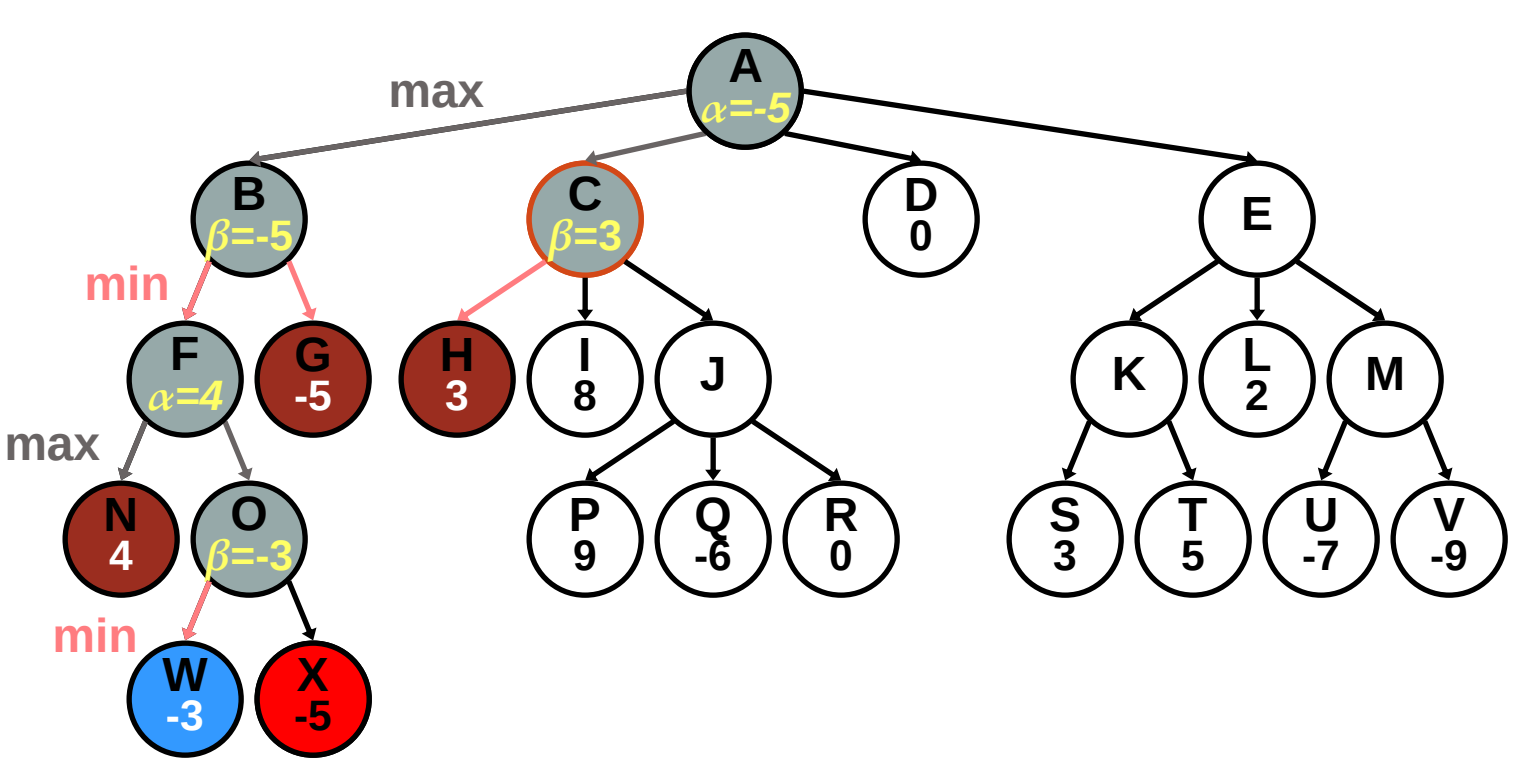
`minimax(H,2,4)` evaluate and return SBE value



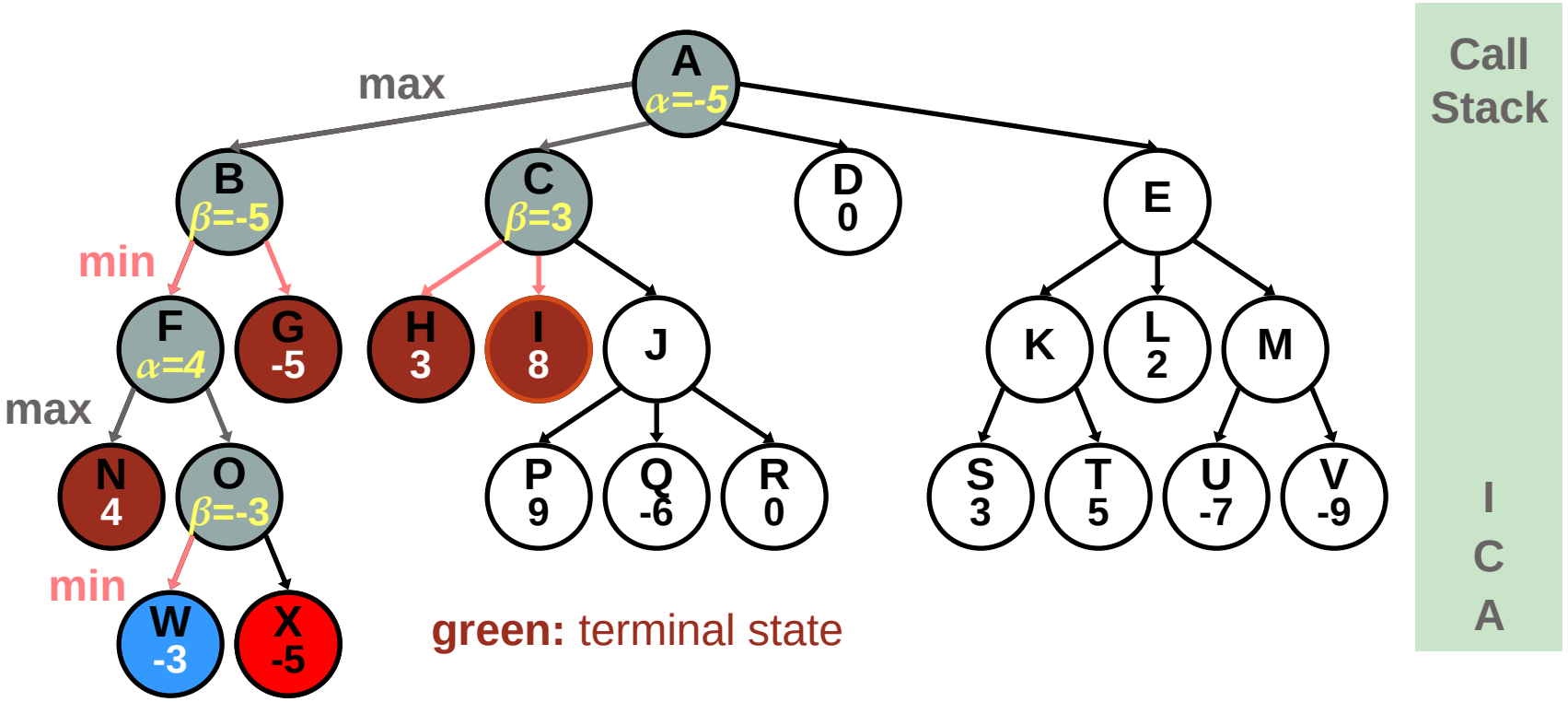
**back to** `minimax(C,1,4)`

beta = 3, since 3 <= +infinity (minimizing)

**Keep expanding C?** Yes since A's alpha >= C's beta is false, no alpha cutoff



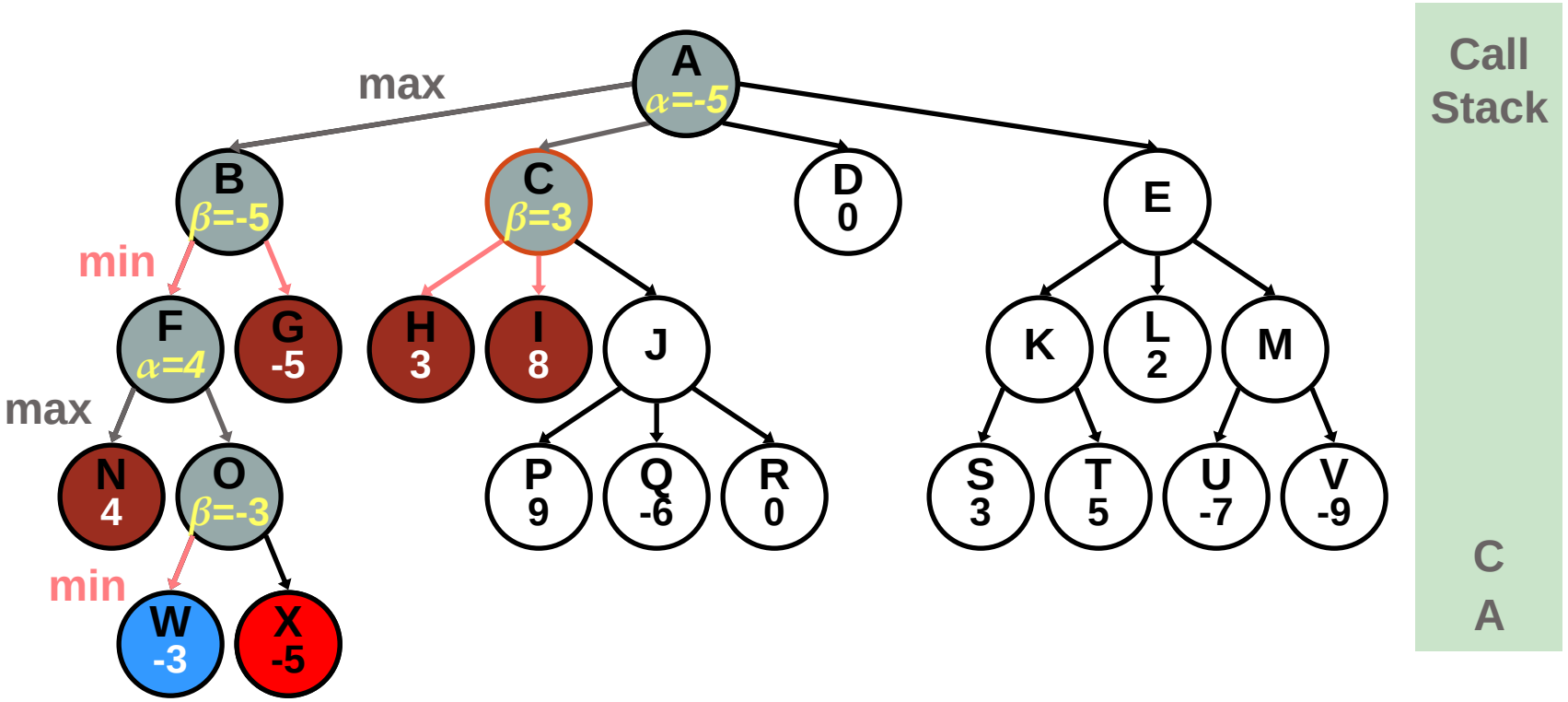
`minimax(I,2,4)` evaluate and return SBE value



**back to** `minimax(C,1,4)`

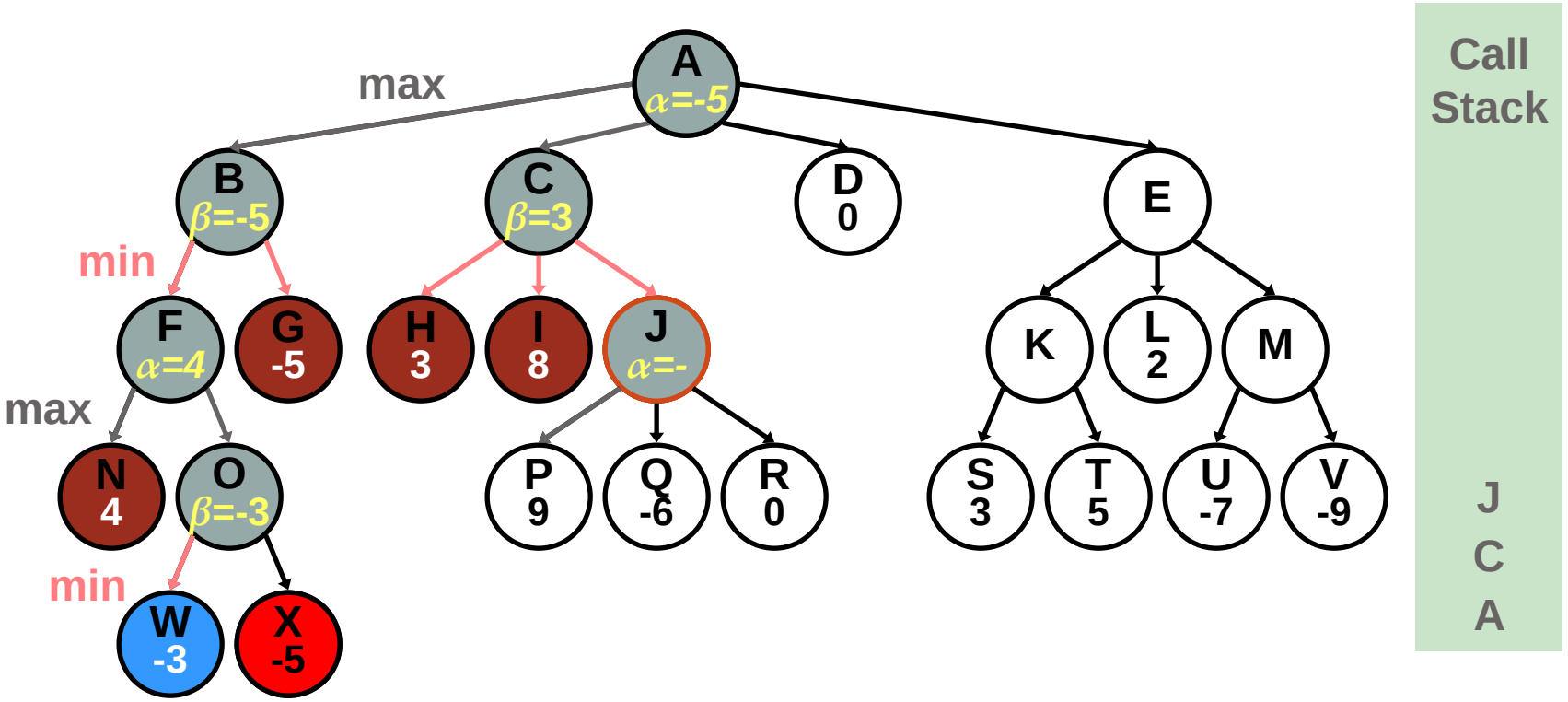
beta doesn't change, since 8 > 3 (minimizing)

**Keep expanding C?** Yes since A's alpha >= C's beta is false, no alpha cutoff

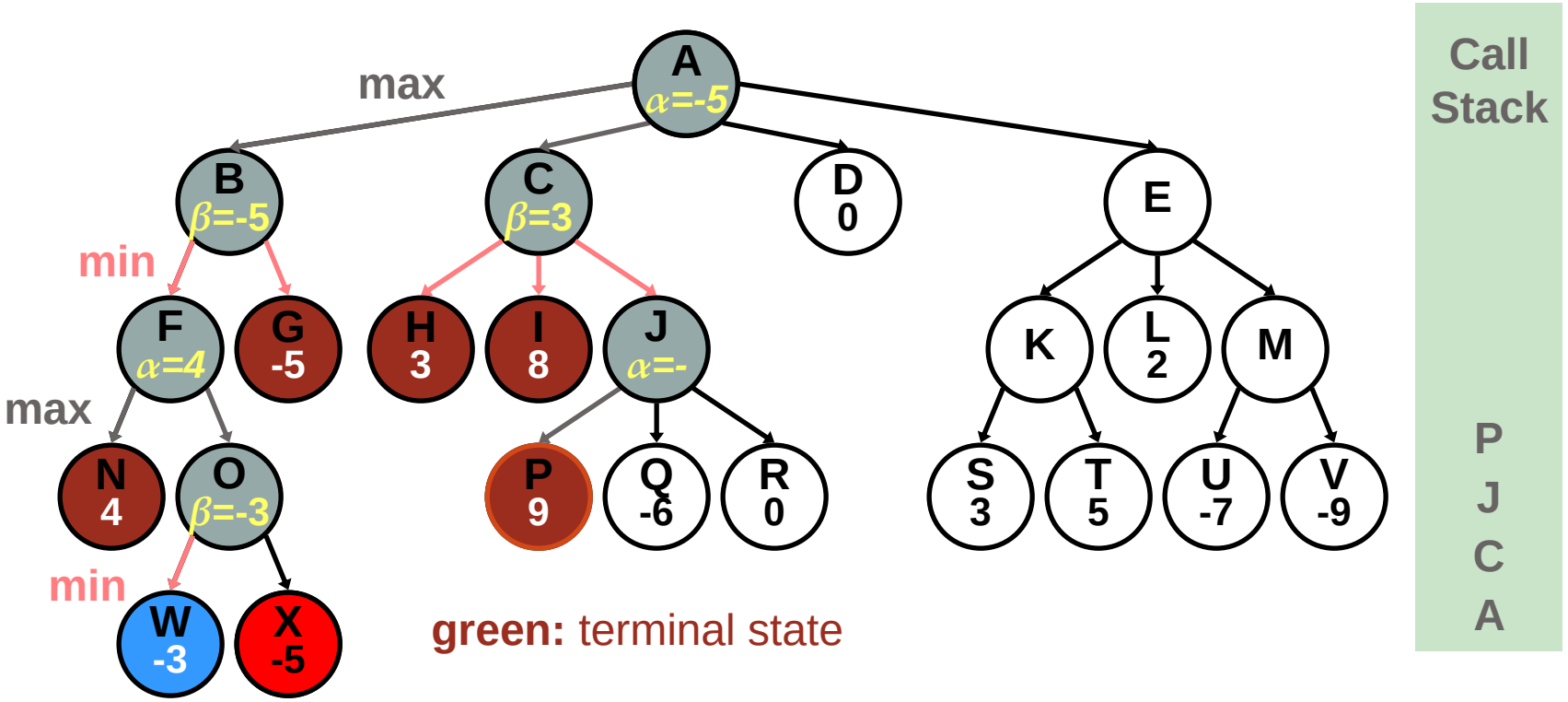


`minimax(J,2,4)` alpha initialized to -infinity

**Expand J?** Yes since J's alpha >= C's beta is false, no beta cutoff



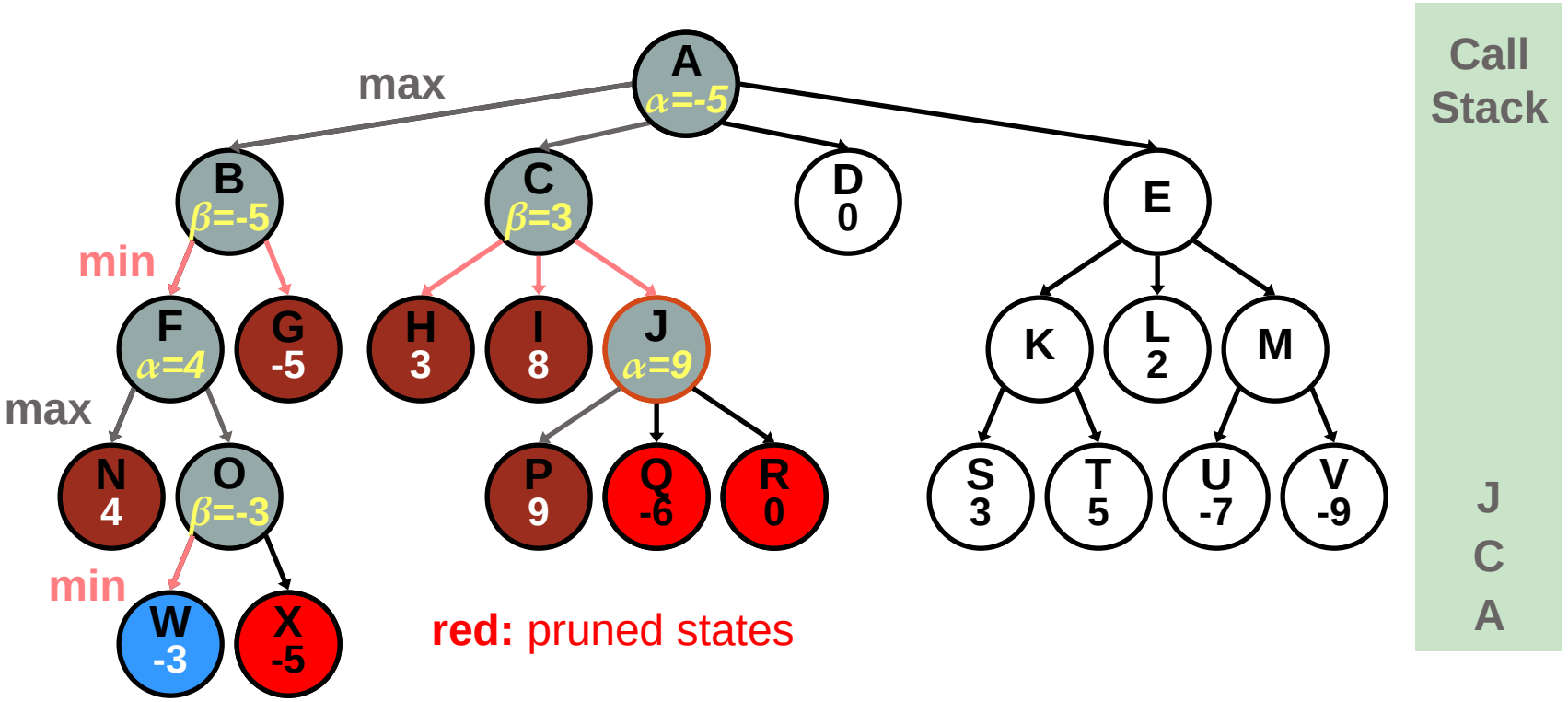
`minimax(P,3,4)` evaluate and return SBE value



**back to** `minimax(J,2,4)`

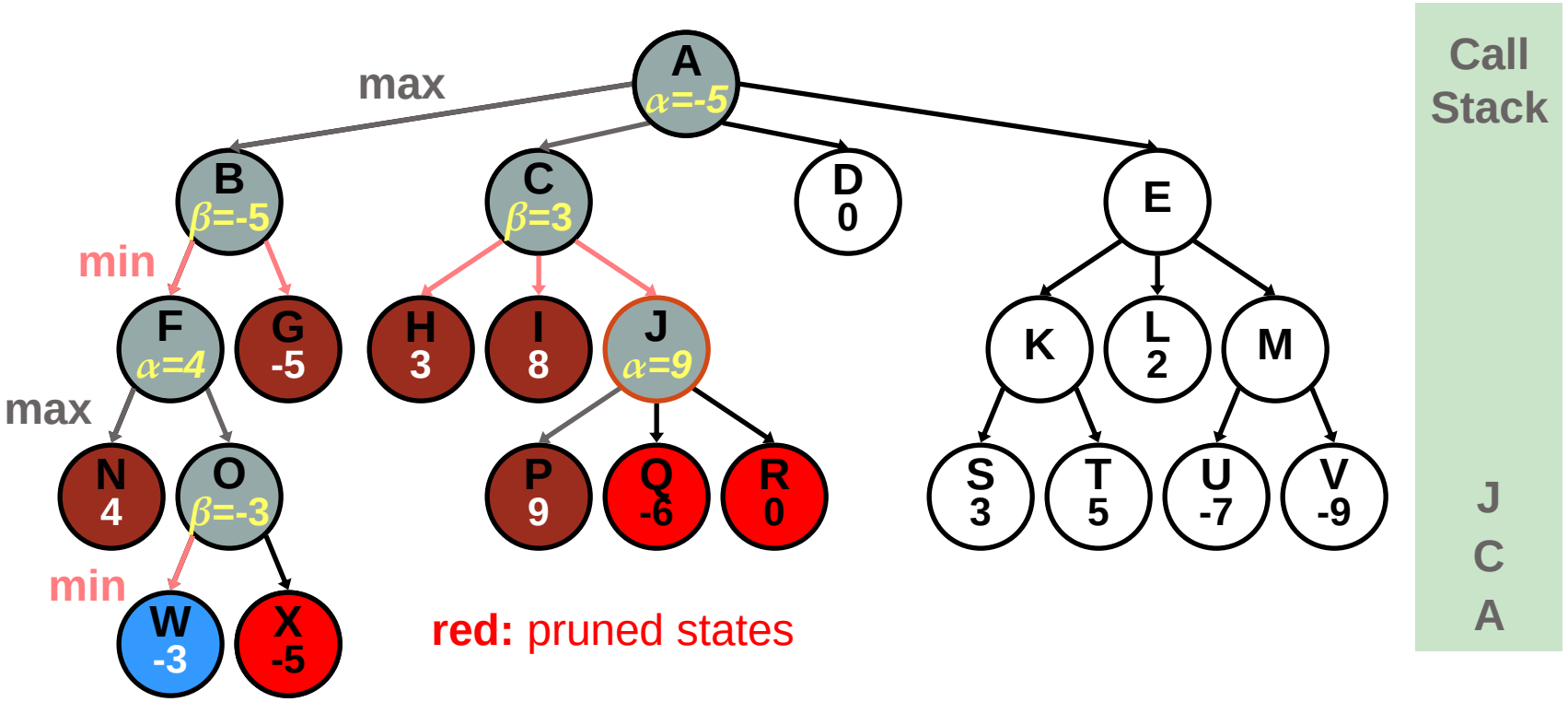
alpha = 9, since 9 >= -infinity (maximizing)

**Keep expanding J?** No since J's alpha >= C's beta is true: **beta cutoff**



☞ **Why?**

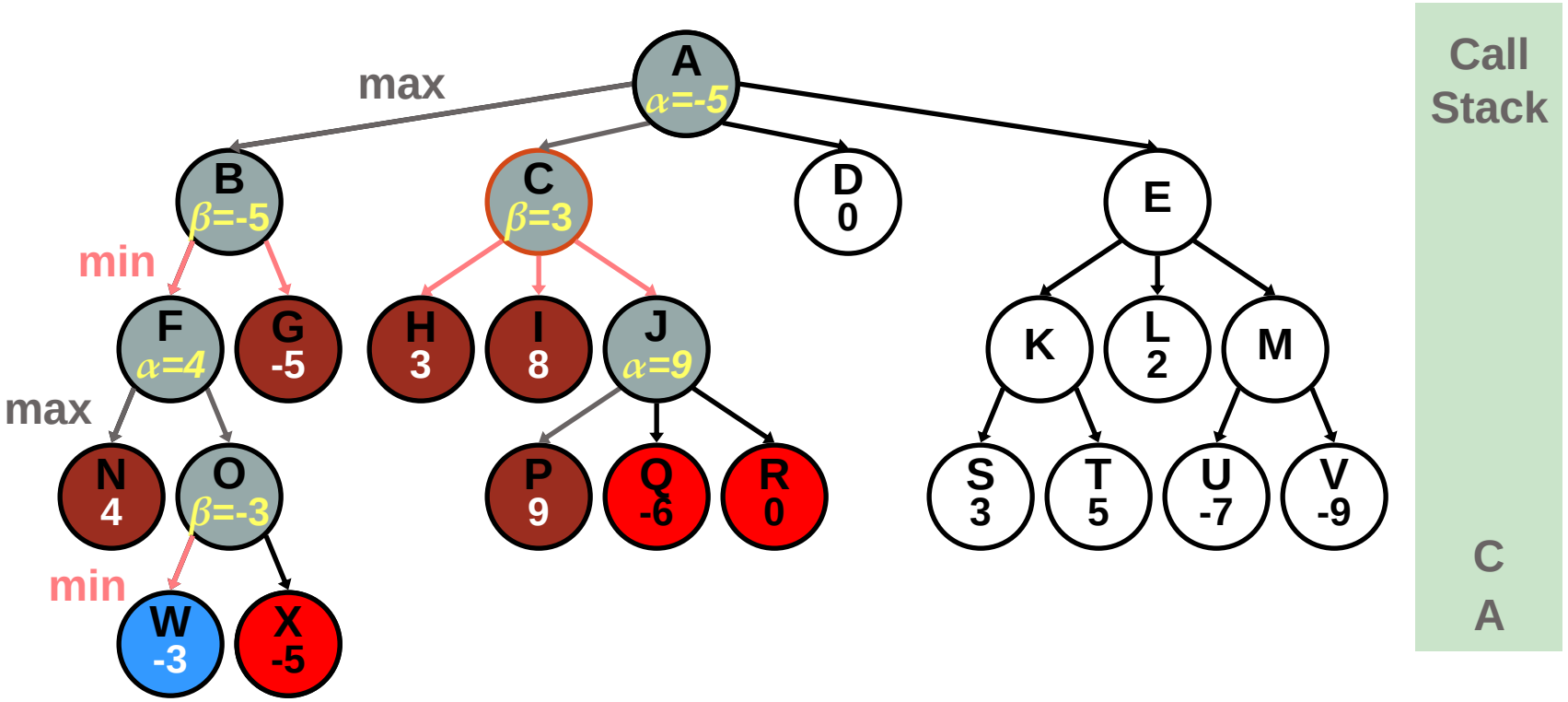
Computer will choose P or better, thus J's lower bound is 9.
Smart opponent won't let computer take move to J
(since opponent already has better move at H).



**back to** `minimax(C,1,4)`

beta doesn't change, since 9 > 3 (minimizing)

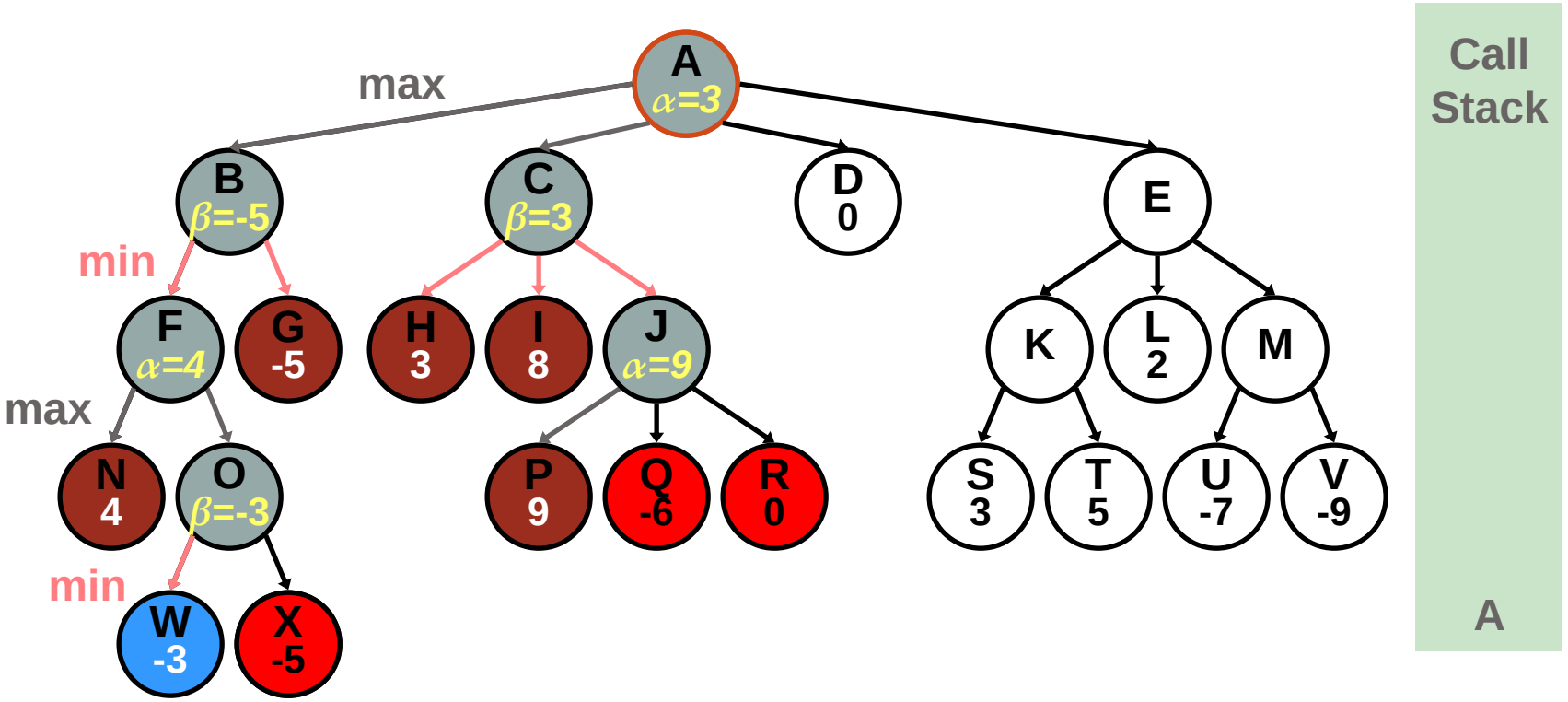
**Keep expanding C?** No since no more successors for C



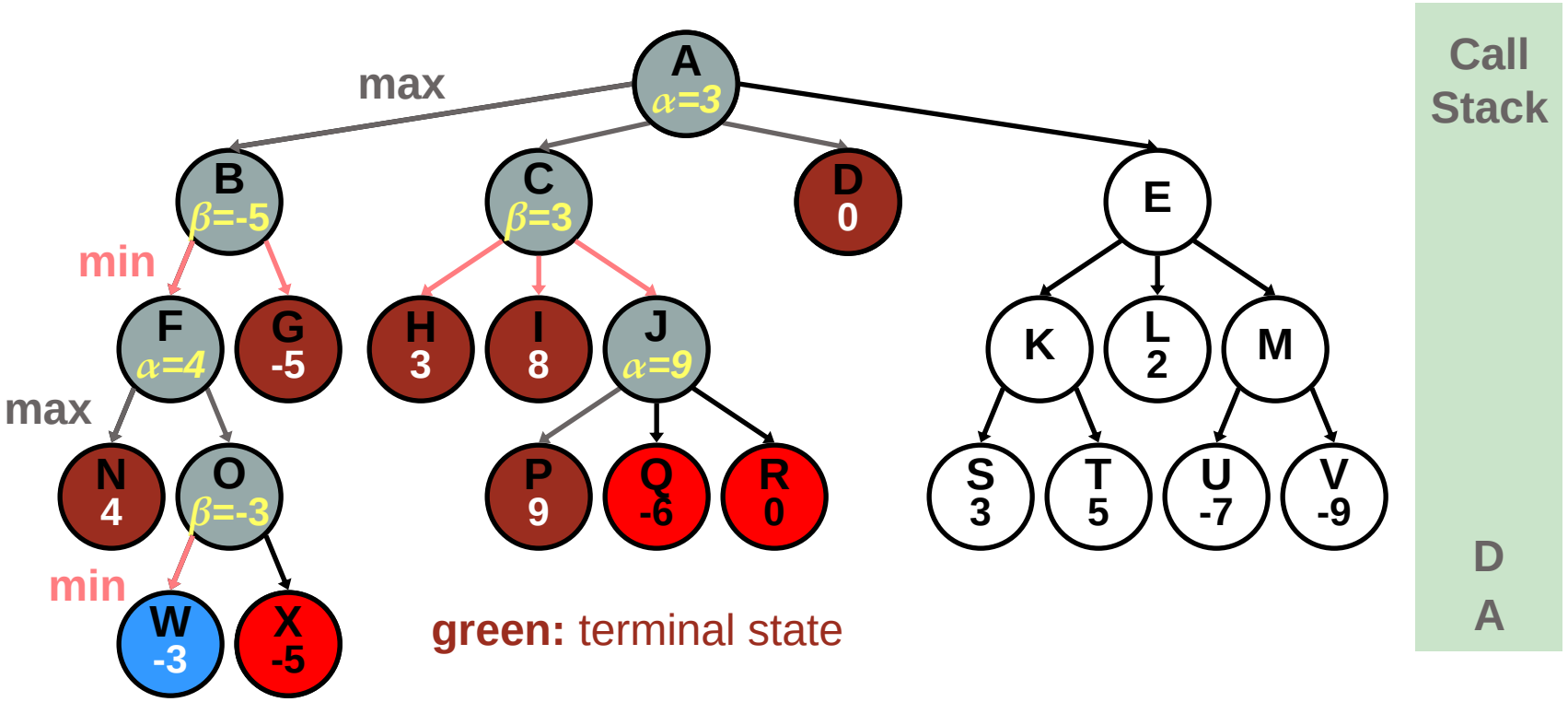
**back to** `minimax(A,0,4)`

alpha = 3, since 3 >= -5 (maximizing)

**Keep expanding A?** Yes since there are more successors, no cutoff test

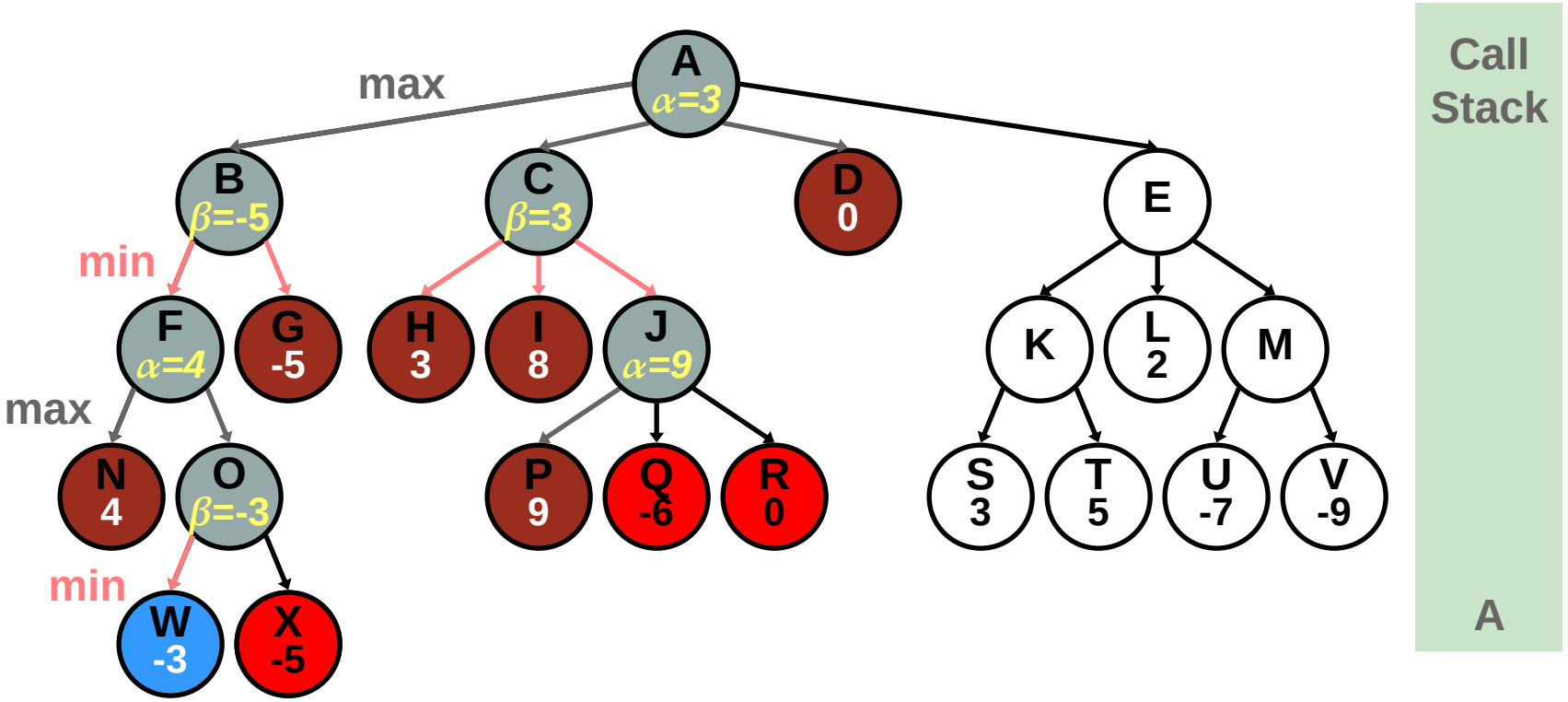


`minimax(D,1,4)` evaluate and return SBE value

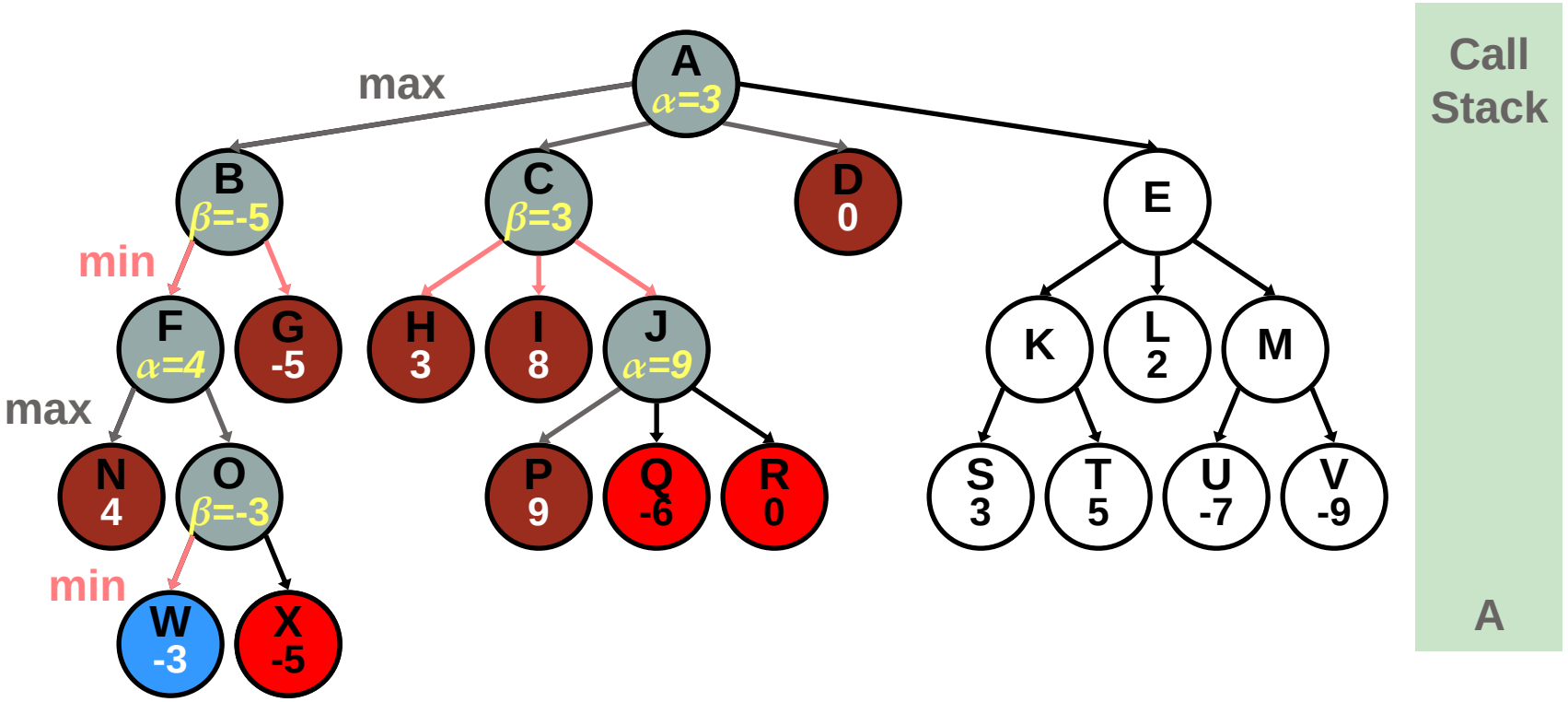


**back to** `minimax(A,0,4)` alpha doesn't change, since 0 < 3 (maximizing)

**Keep expanding A?** Yes since there are more successors, no cutoff test



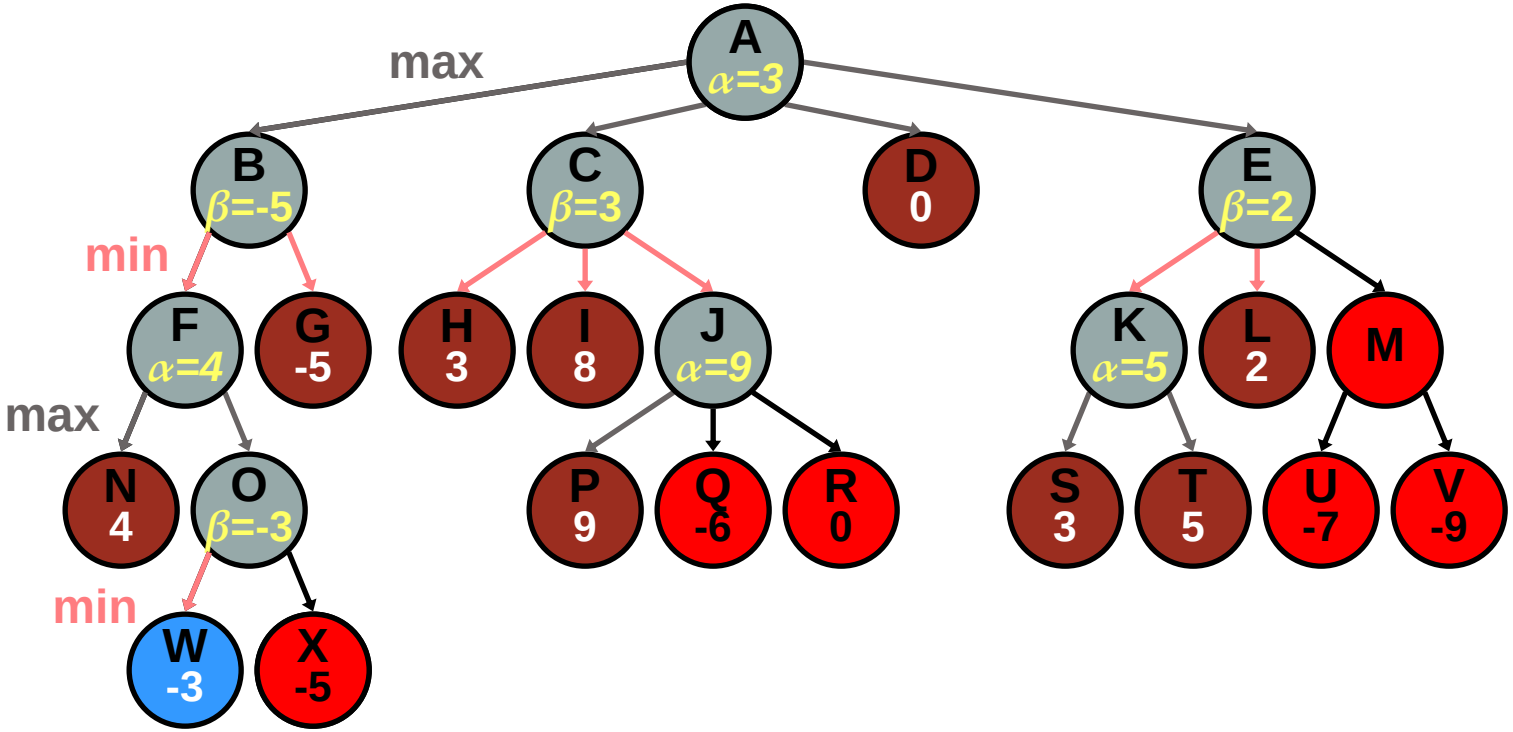
☞ How does the algorithm finish searching the tree?



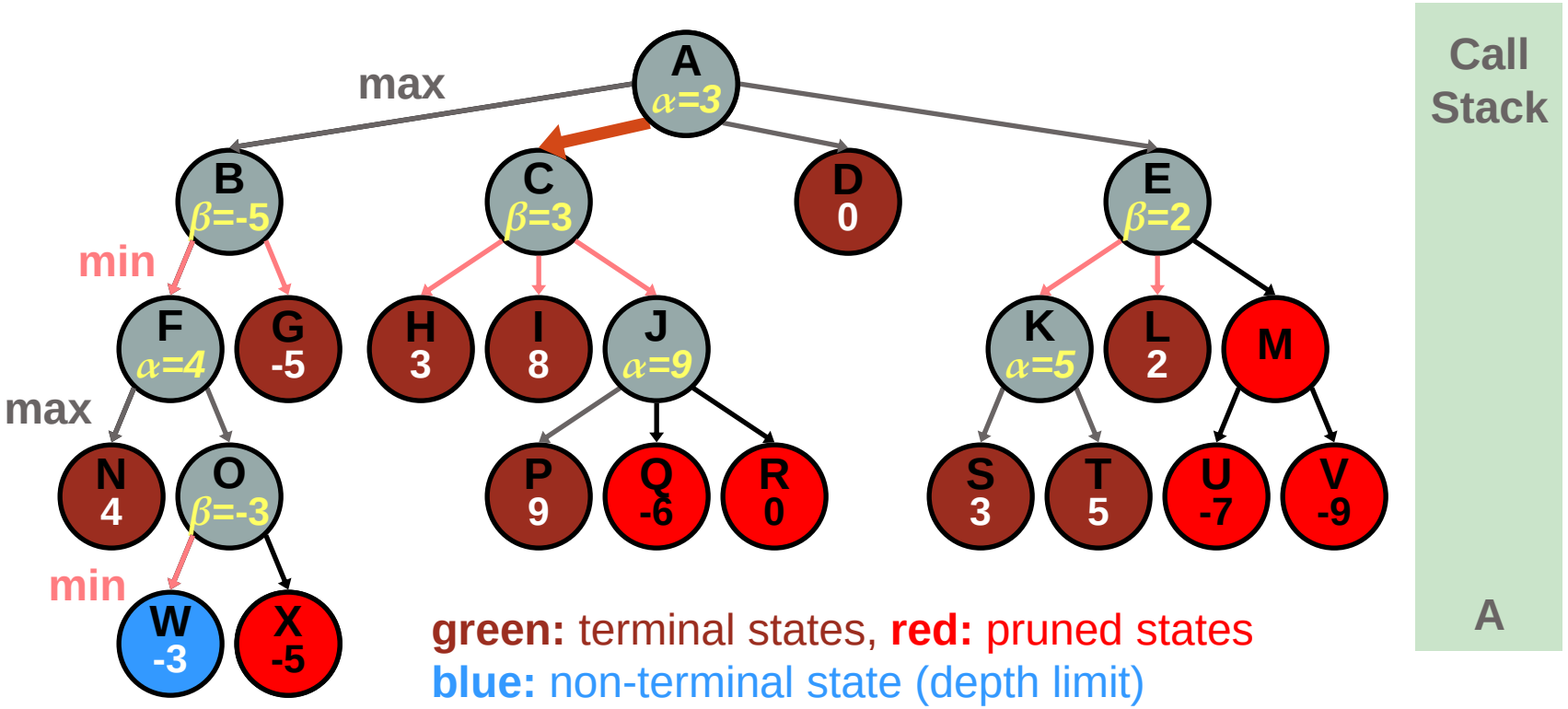
Stop Expanding E since A's alpha >= E's beta is true: alpha cutoff

☞ **Why?**

Smart opponent will choose L or worse, thus E's upper bound is 2.
Computer already has better move at C.



**Result:** Computer chooses move to C.



# ▪ STOCHASTIC GAMES

- Many unpredictable external events can put us into unforeseen situations.
- Many games mirror this unpredictability by including a random element, such as the throwing of dice. We call these **stochastic games**.
- Backgammon is a typical game that combines luck and skill. Dice are rolled at the beginning of a player's turn to determine the legal moves.
- In the backgammon position of Figure, for example, White has rolled a 6–5 and has four possible moves.

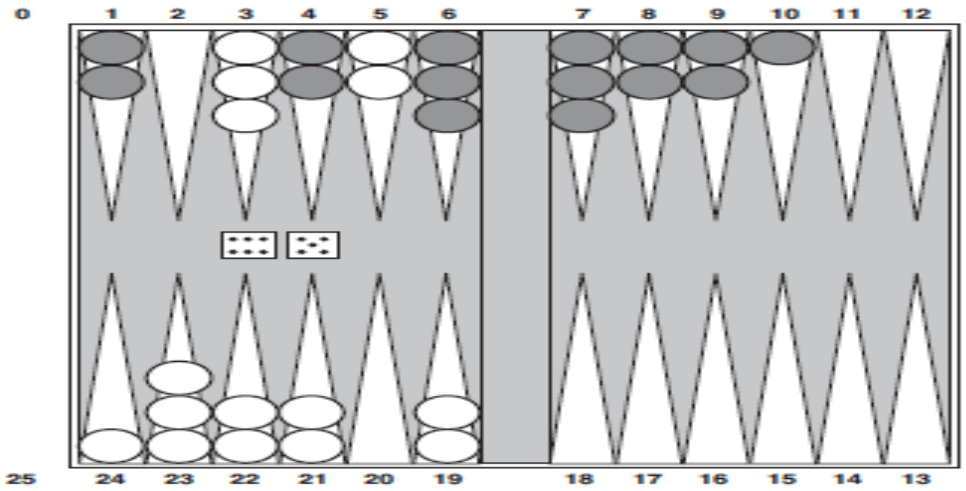
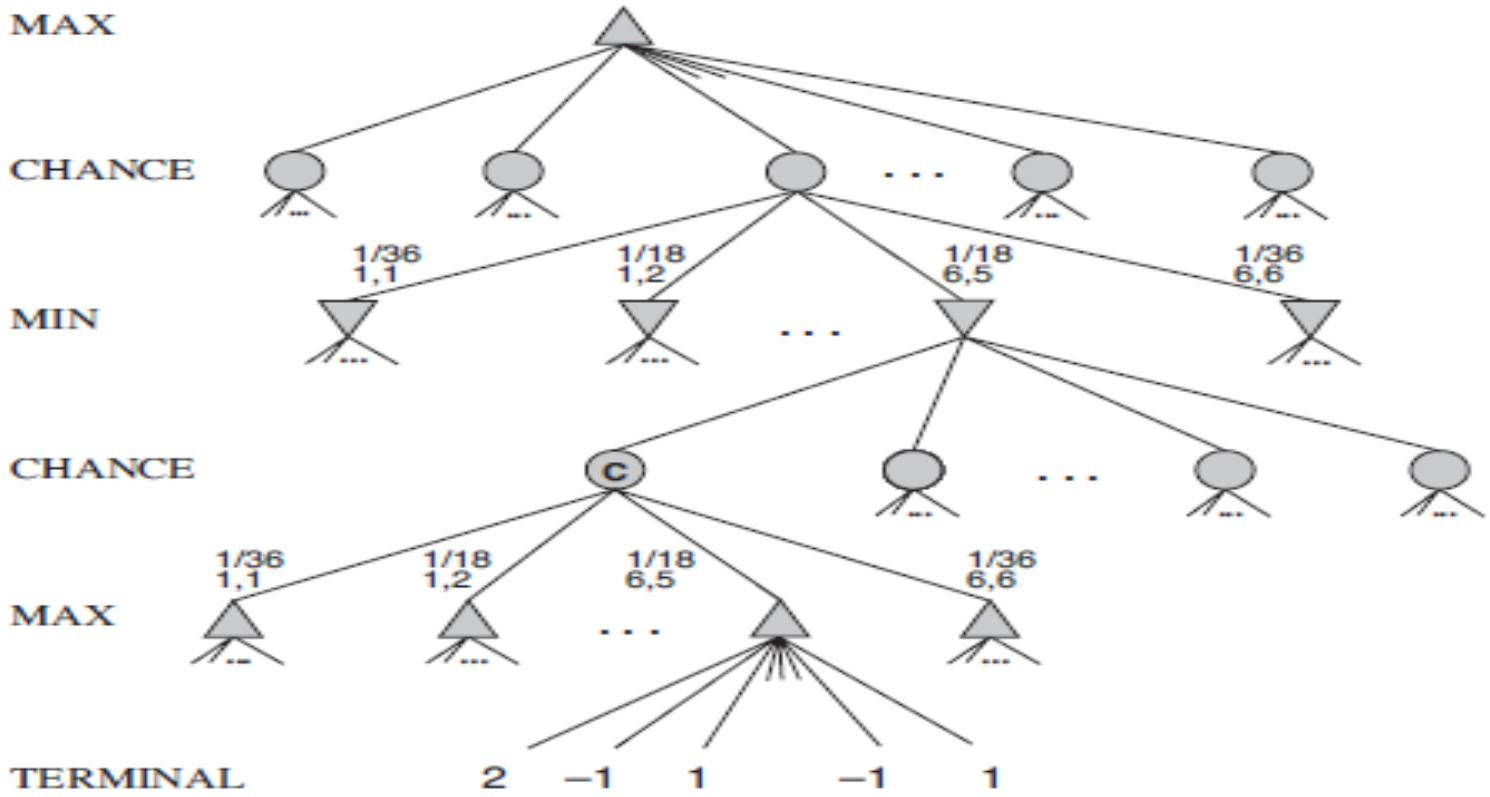



**Figure 5.10** A typical backgammon position. The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25, and Black moves counterclockwise toward 0. A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over. In the position shown, White has rolled 6–5 and must choose among four legal moves: (5–10,5–11), (5–11,19–24), (5–10,10–16), and (5–11,11–16), where the notation (5–11,11–16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.

- Although White knows what his or her own legal moves are, White does not know what Black is going to roll and thus does not know what Black's legal moves will be.
- A game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes.
- Chance nodes are shown as circles in Figure 5.11. The branches leading from each chance node denote the possible dice rolls; each branch is labeled with the roll and its probability.
- There are 36 ways to roll two dice, each equally likely; but because a 6–5 is the same as a 5–6, there are only 21 distinct rolls.
- The six doubles (1–1 through 6–6) each have a probability of 1/36, so we say P(1–1) = 1/36. The other 15 distinct rolls each have a 1/18 probability.
- we can only calculate the **expected value** of a position: the average over all possible outcomes of the chance nodes.
- This leads us to generalize the **minimax value** for deterministic games to an **expecti- minimax value** for games with chance nodes.
- For chance nodes we compute the expected value, which is the sum of the value over all outcomes, weighted by the probability of each chance action:

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \\ \sum_r P(r)\text{EXPECTIMINIMAX}(\text{RESULT}(s,r)) & \text{if PLAYER}(s) = \text{CHANCE} \end{cases}$$




**5.11** Schematic game tree for a backgammon position.

where r represents a possible dice roll (or other chance event) and RESULT(s, r) is the same
state as s, with the additional fact that the result of the dice roll is r.