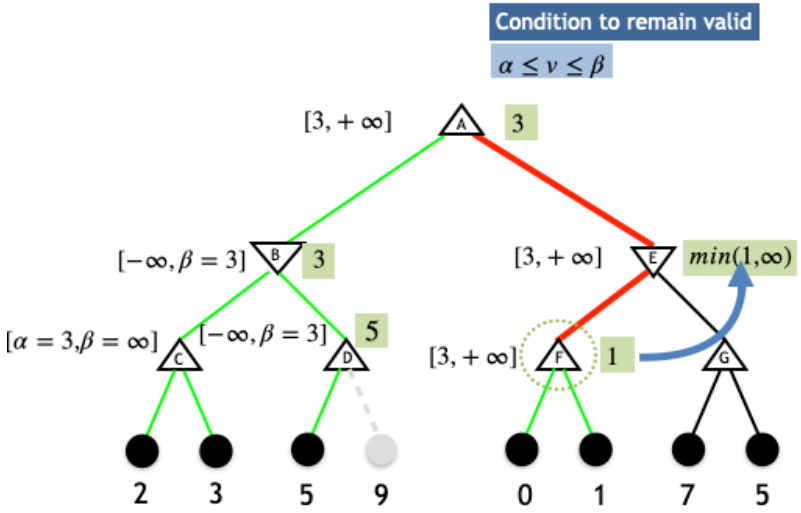
CS143: Artificial Intelligence

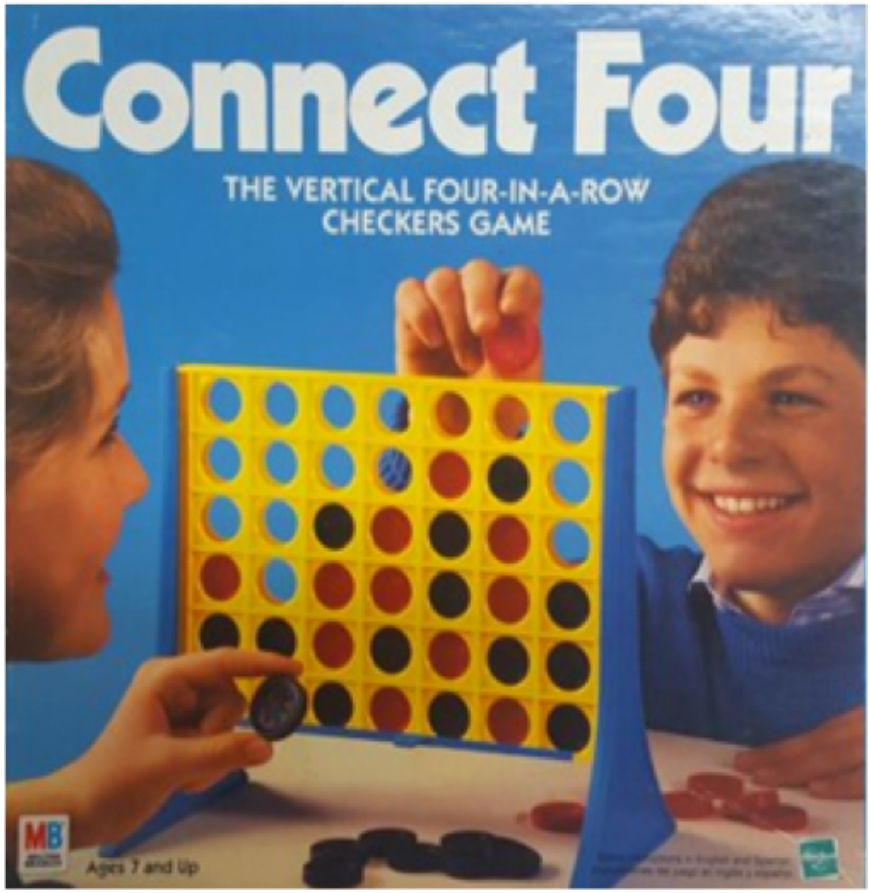


α-β Pruning Implementation

Games of Chance and Expectation

Drake UNIVERSITY

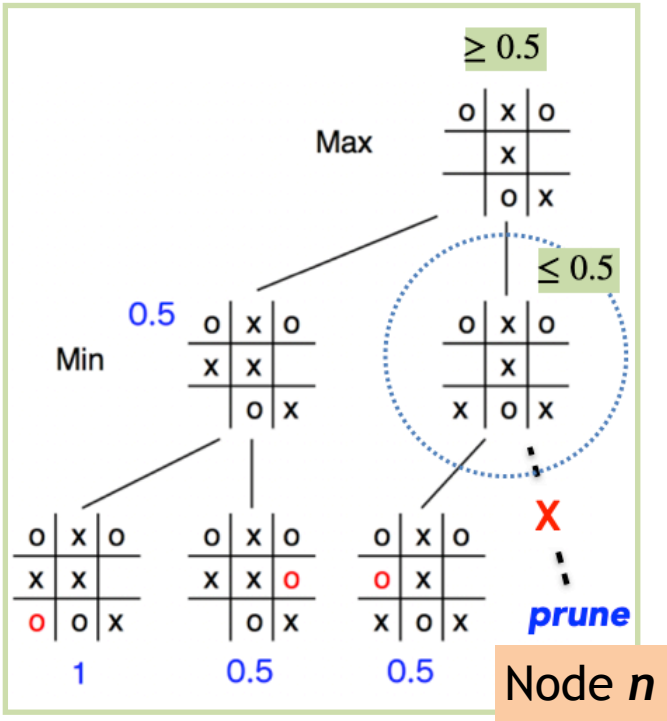
Adversarial Search to solve Game Connect Four



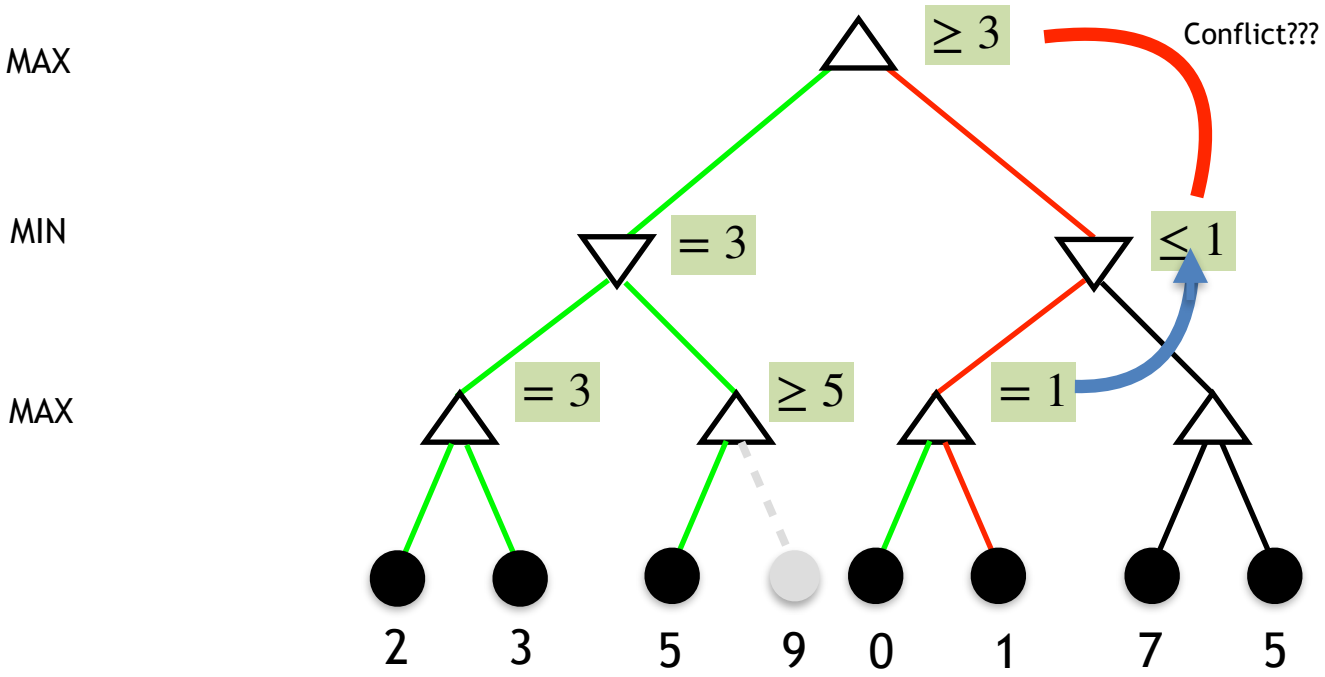
Recap: α-β Pruning: Main Idea

- Consider a node n that could be visited
- If a player has a better choice m at the parent of node n (or at any point farther up the tree), n will never be reached in actual play

- If so, prune it

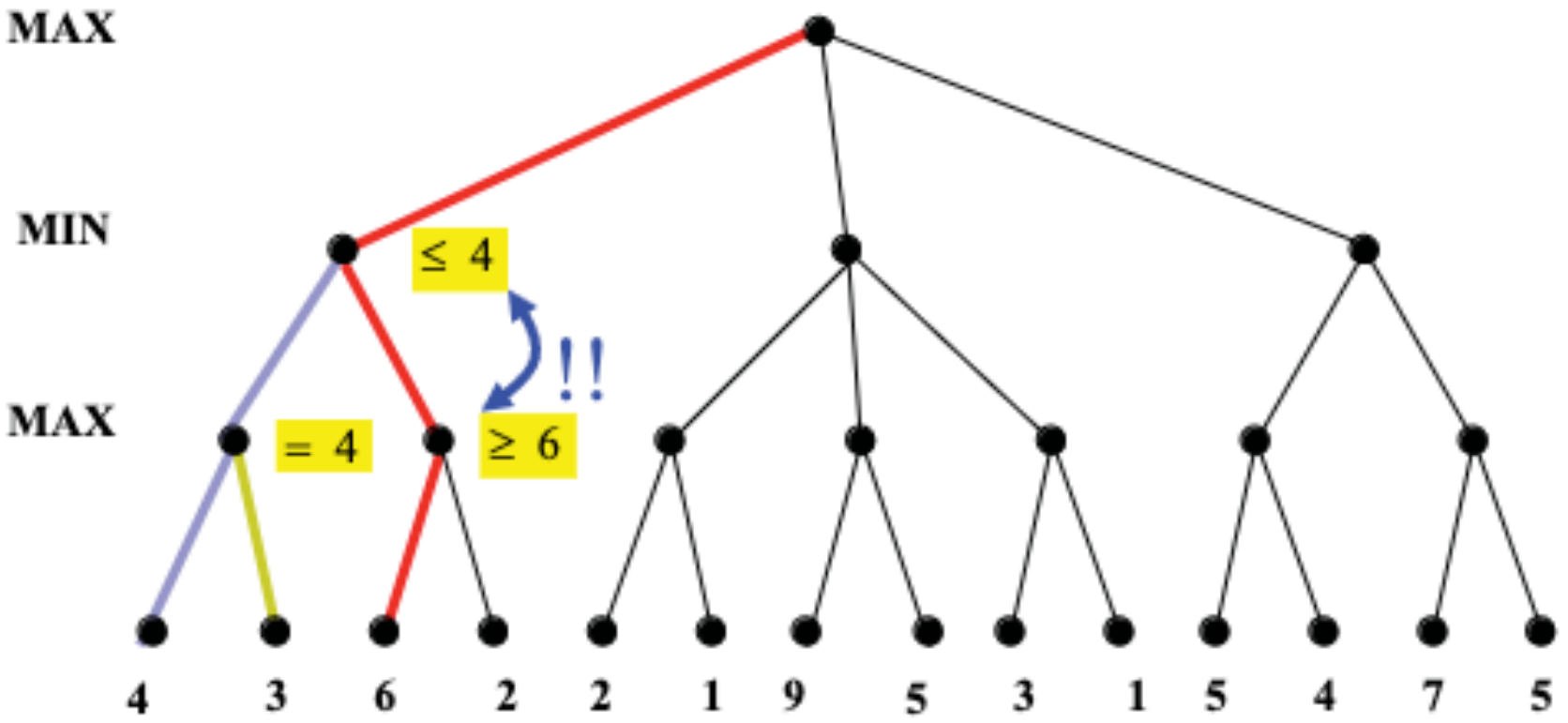


Recap: α-β Pruning Example



However, MIN's parent (which is a MAX) can never get less than 3
hence prune other branches from the MIN node

Recap: α-β Pruning Activity



α-β Pruning Pseudocode

```
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-β Pruning Pseudocode

```
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
```

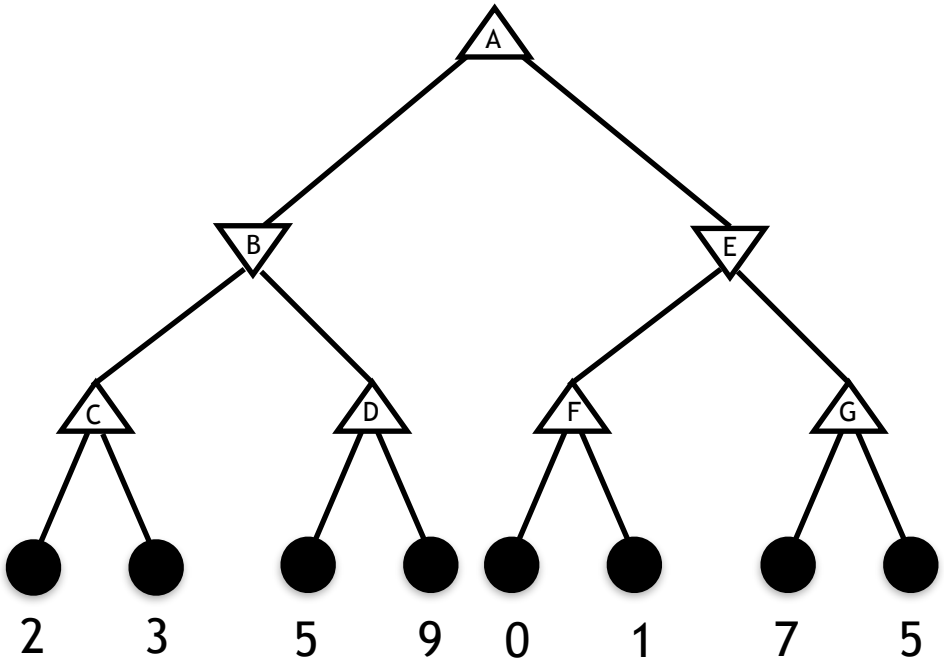
PRUNE OTHER BRANCHES

```
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
```

PRUNE OTHER BRANCHES

Reference: Russell and Norvig Textbook

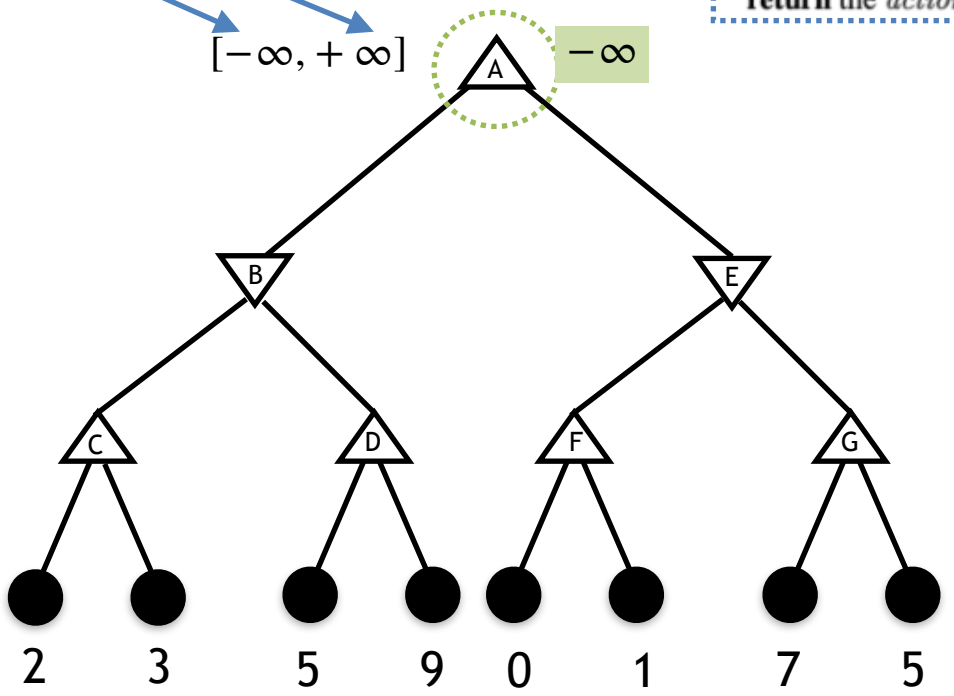
α-β Pruning on a Game Tree



α-β Pruning Pseudocode: Step-by-Step Execution

α-β Pruning Code Execution

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

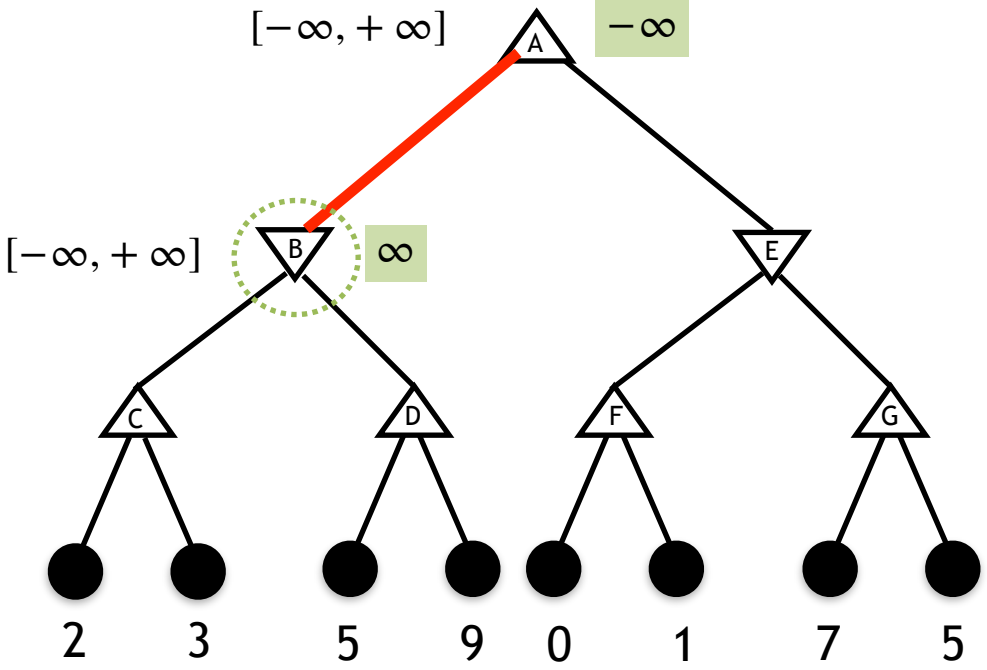


- *function* alphabeta() gets called with α = *-infinity* and β =*+infinity* at the root node **A**
- Node **A** is not a terminal node hence it will recursively call *function* alphabeta() on its left child **B** with α = *-infinity* and β =*+infinity*

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$

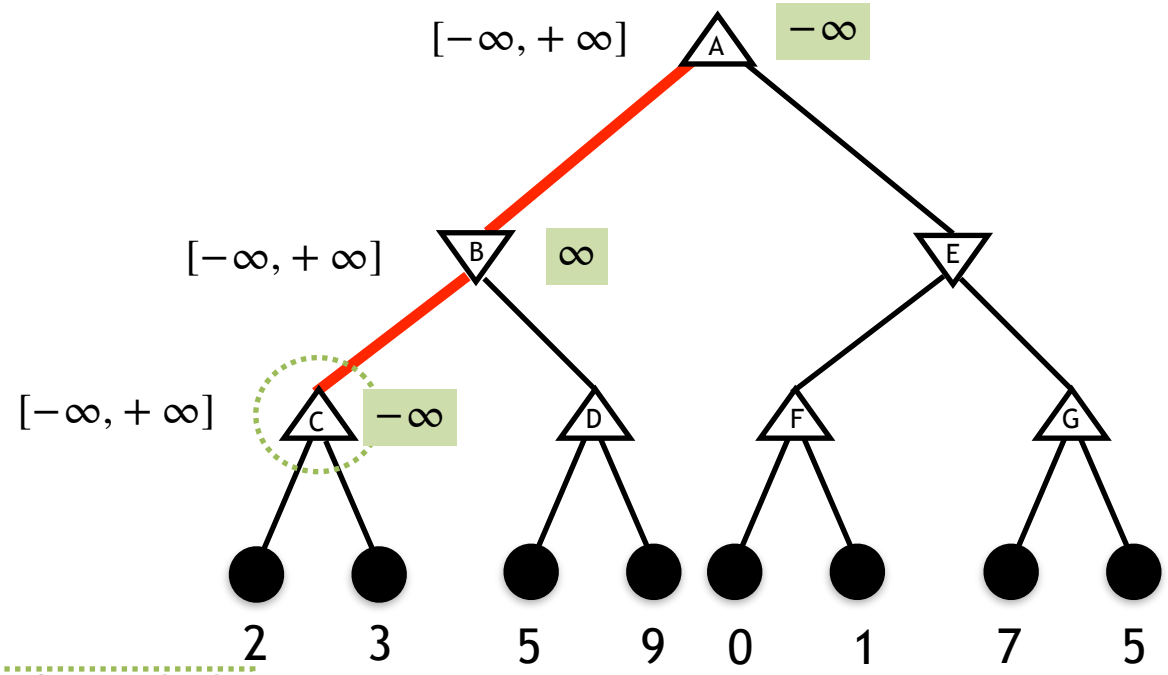


- Node **B** will initialize its α = *-infinity* and β =*+infinity*
- Node **B** is not a terminal node hence it will recursively call *function* alphabeta() on its left child **C**

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$



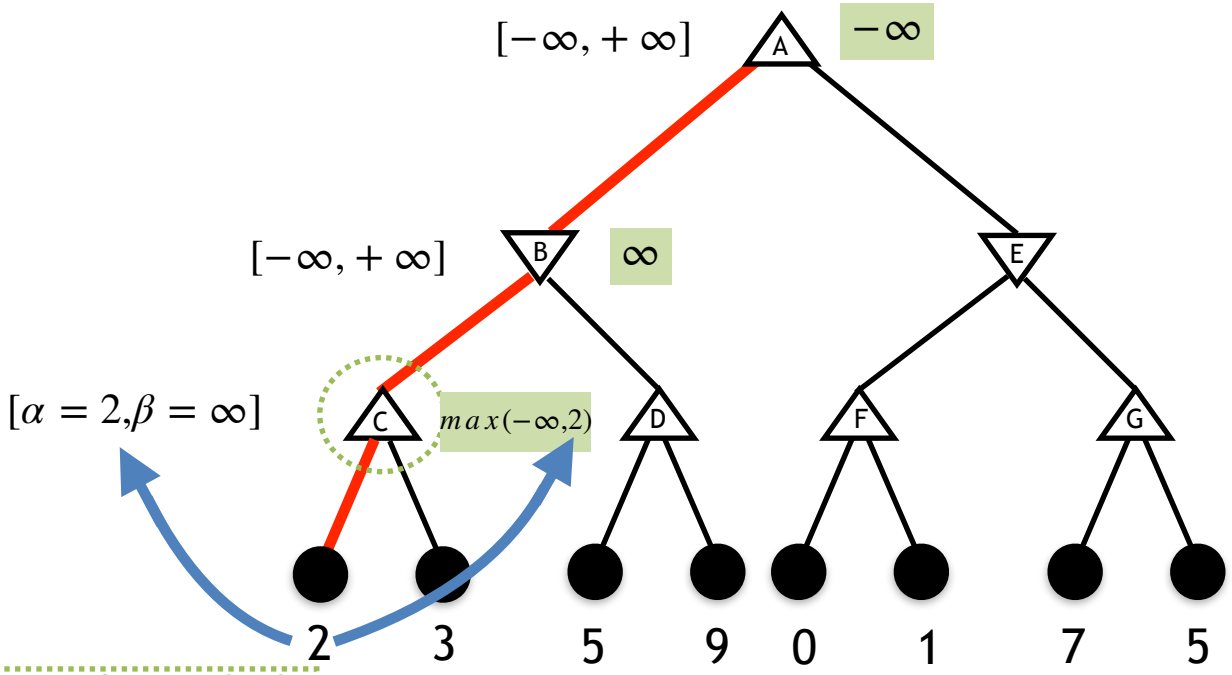

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

- Node **C** will initialize its α = *-infinity* and β =*+infinity*
- Node **C** is not a terminal node hence it will recursively call f*unction* alphabeta() on its left child

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$



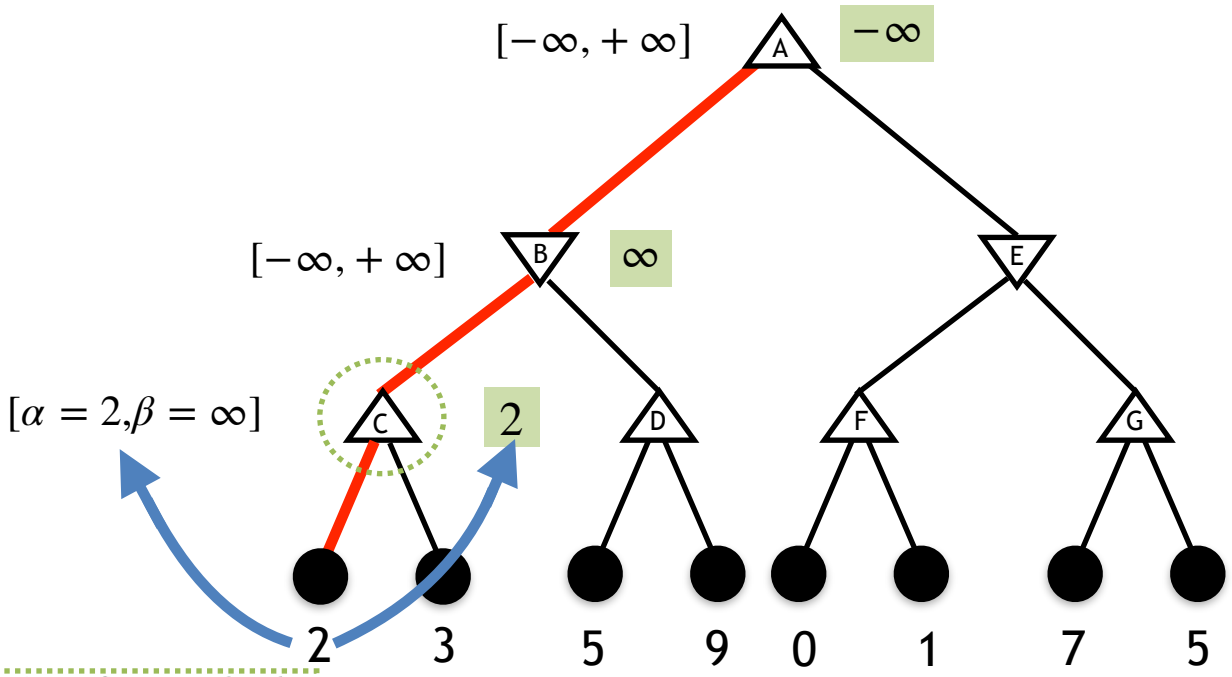

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

- Node **C** receives value **2** from its left terminal node
- Node **C** will update its $\alpha = 2$
- Since **C** is a MAX node, it will not update its β value. It remains at $\beta = +infinity$

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$



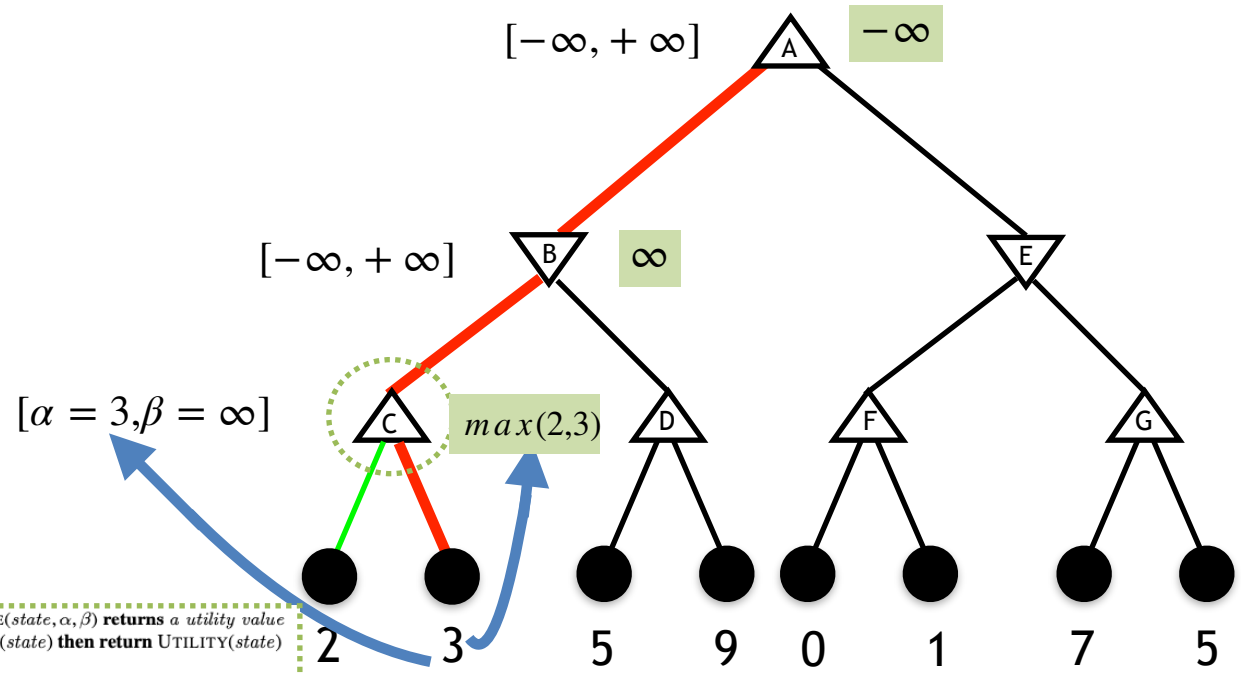

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

- Node **C** receives value **2** from its left terminal node
- Node **C** will update its $\alpha = 2$
- Since **C** is a MAX node, it will not update its β value. It remains at $\beta = +infinity$

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$



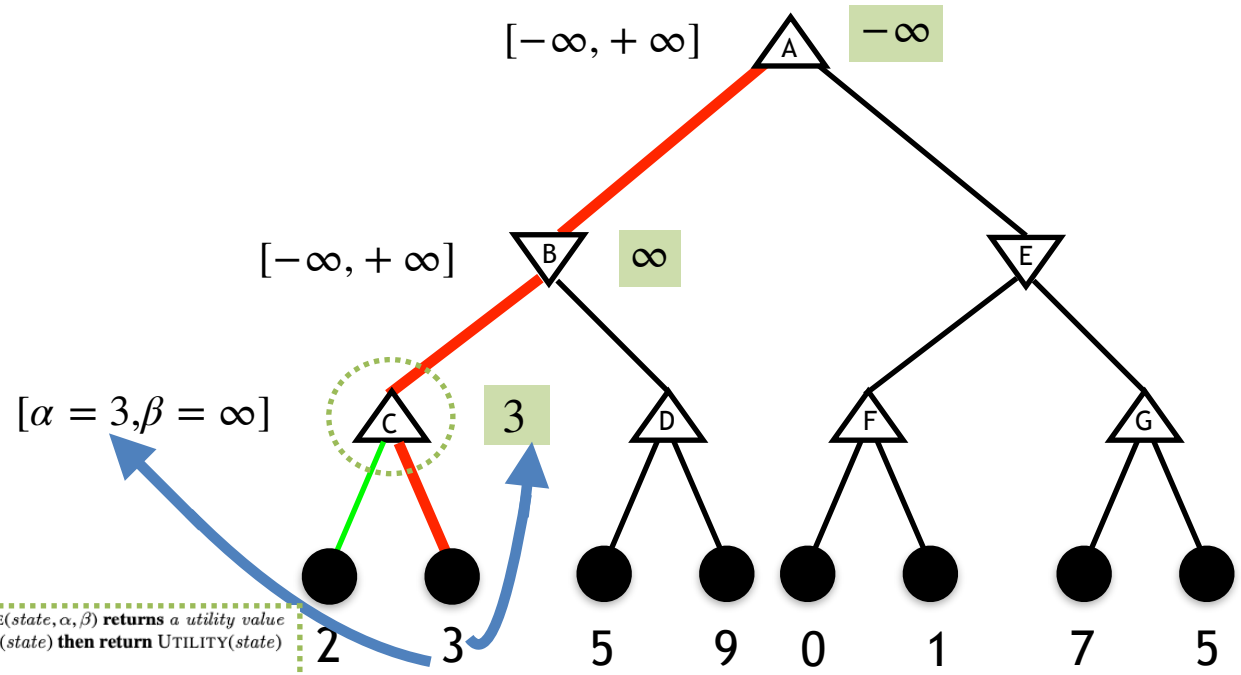

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

- Node **C** receives value **3** from its right terminal node
- Node **C** will update its $\alpha = 3$
- Since **C** is a MAX node, it will not update its β value. It remains at $\beta = +infinity$

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$




```
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
```

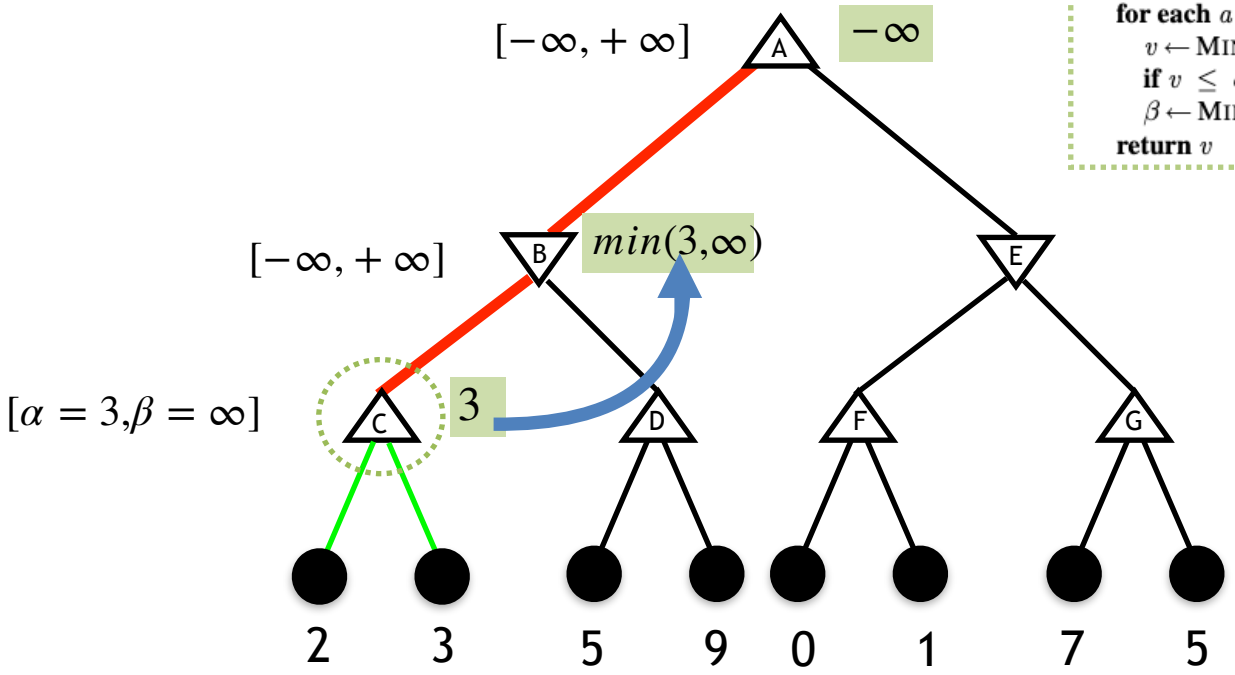
- Node **C** receives value **3** from its right terminal node
- Node **C** will update its $\alpha = 3$
- Since **C** is a MAX node, it will not update its β value. It remains at $\beta = +infinity$

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$

```
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
```



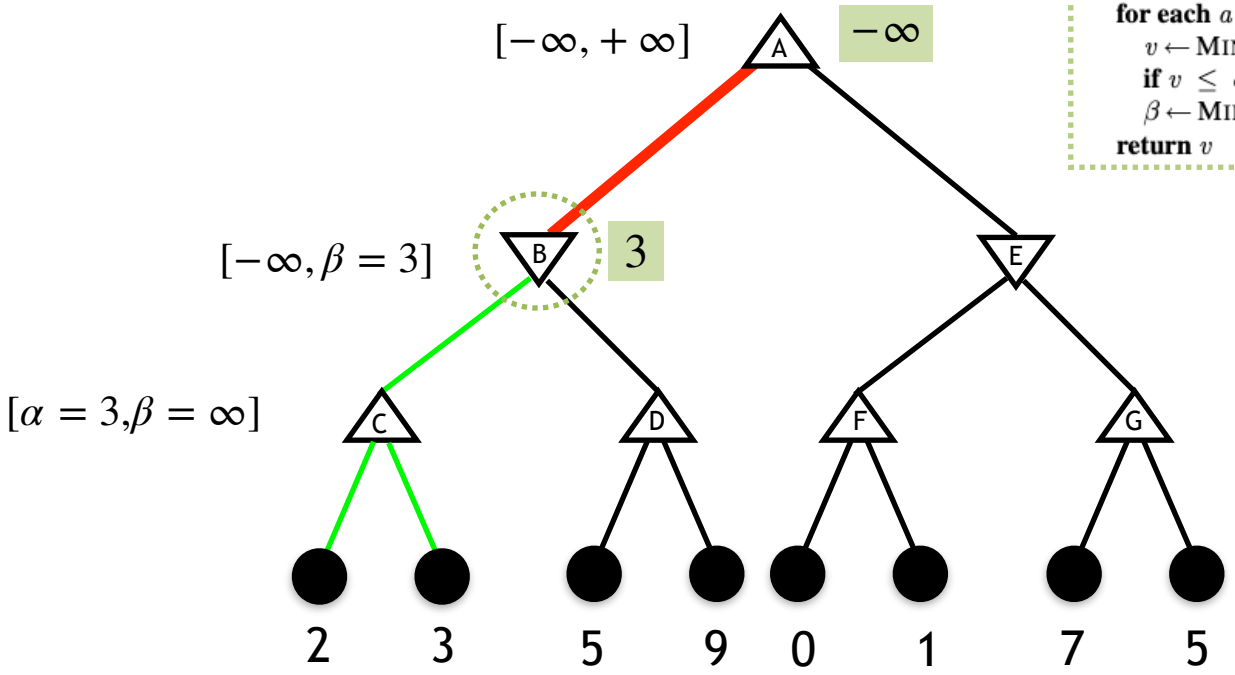
- Finally, Node **C** has no more children, it will break the loop and return value **3** to its parent Node **B**

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$

```
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
```

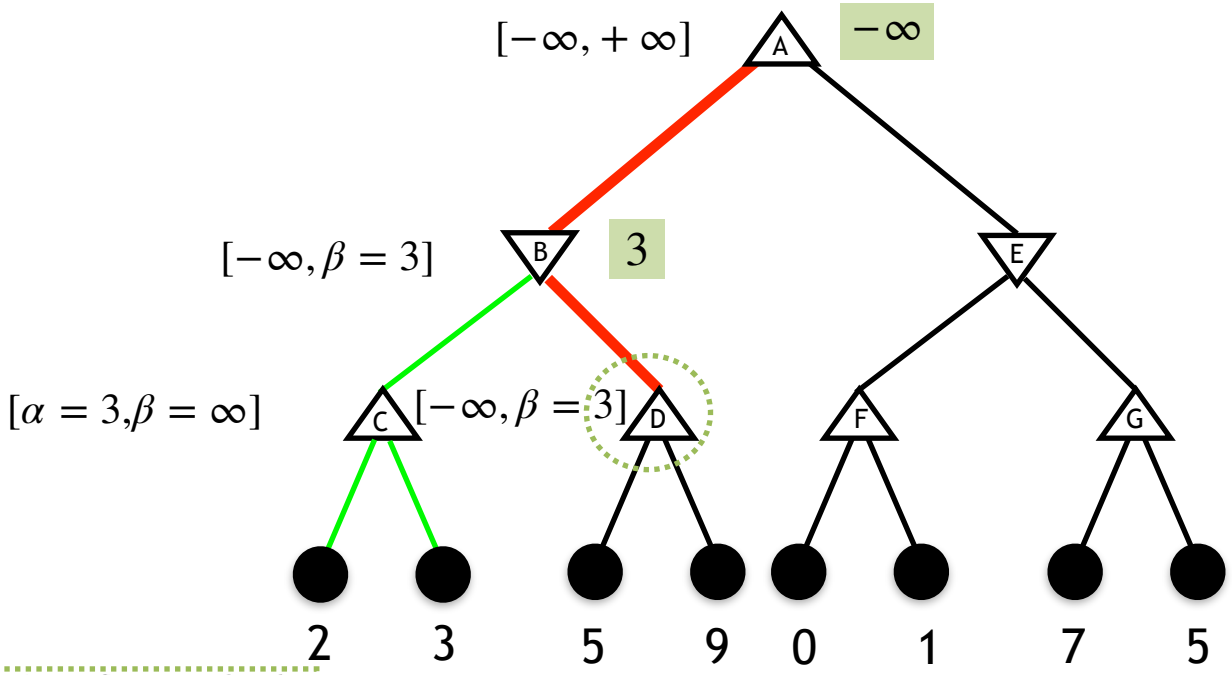


- Node **B** receives value **3** from its left child **C**
- Node **B** will update its $\beta = 3$
- Since **B** is a MIN node, it will not update its α value. It remains at $\alpha = -infinity$
- Node **B** will recursively call *function* alphabeta() on its right child Node **D** with $\alpha = -infinity$ and $\beta = 3$

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$



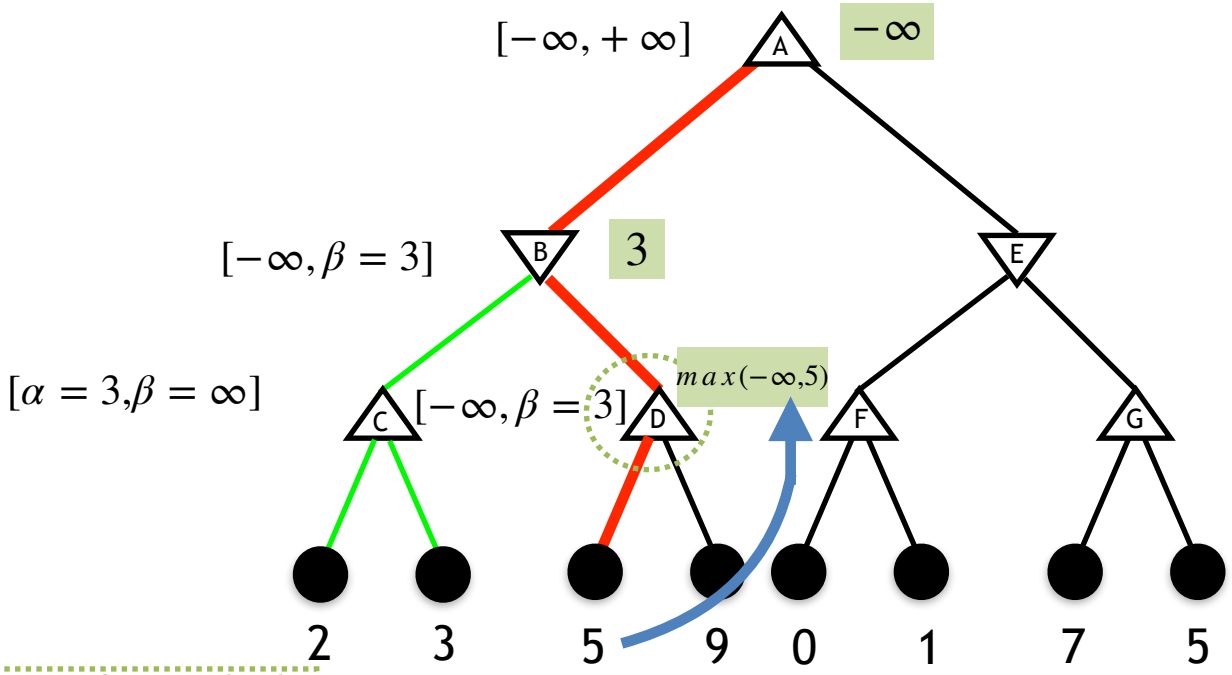

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

- Node **D** will initialize its α = *-infinity* and β =*3*
- Node **D** is not a terminal node hence it will recursively call *function* alphabeta() on its left child

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$



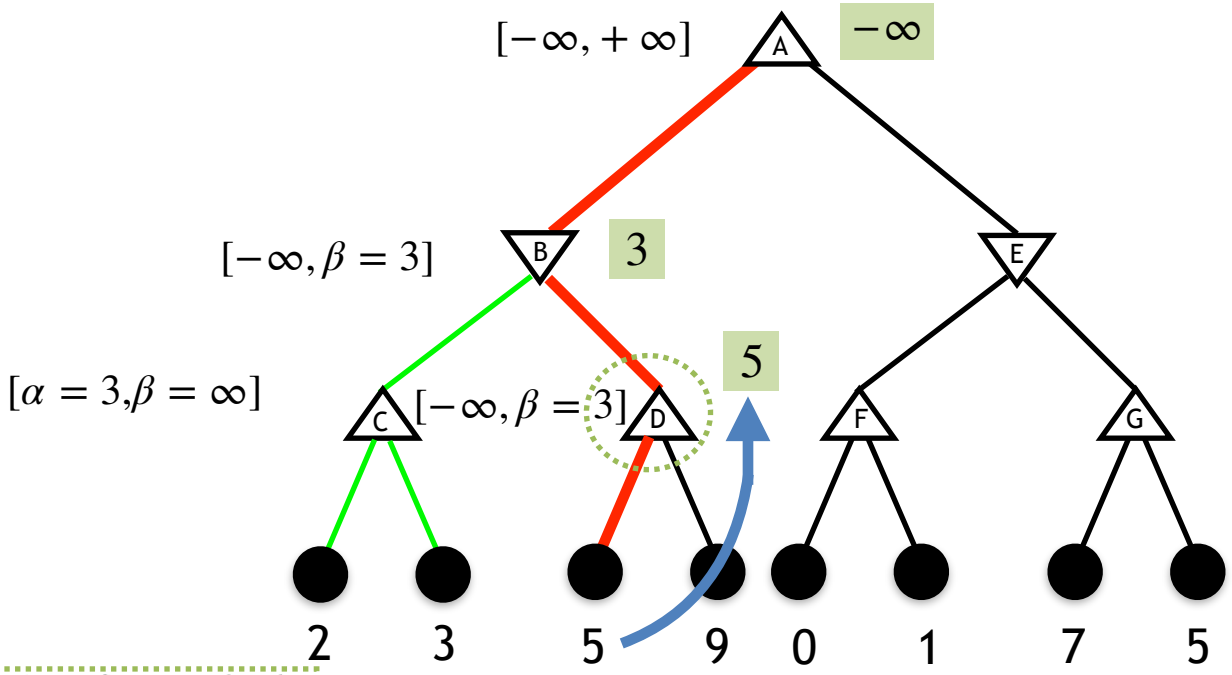

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

- Node **D** receives value **5** from its left terminal node
- Node **D** will update its $\alpha = 5$
- Since **D** is a MAX node, it will not update its β value. It remains at $\beta = 3$

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$




```
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
```

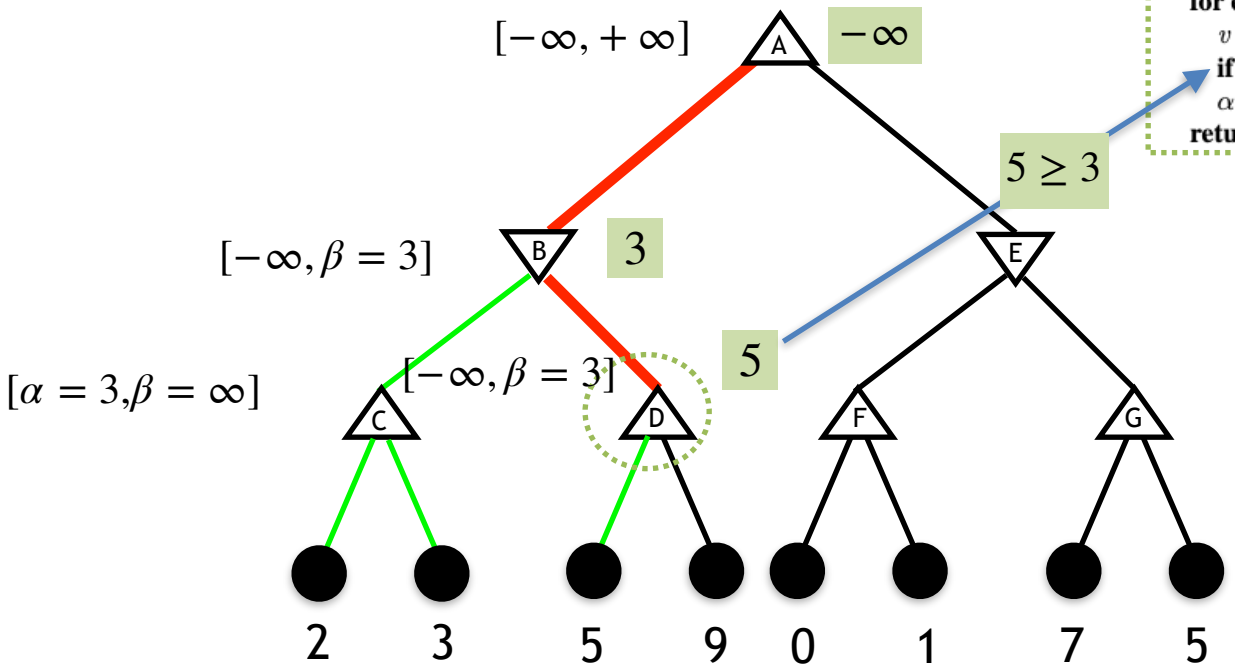
- Node **D** receives value **5** from its left terminal node
- Node **D** will update its $\alpha = 5$
- Since **D** is a MAX node, it will not update its β value. It remains at $\beta = 3$

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$

```
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
```



- **D** is a MAX node, it will test the **PRUNE CONDITION**: $v \geq \beta$

  $5 \geq 3$

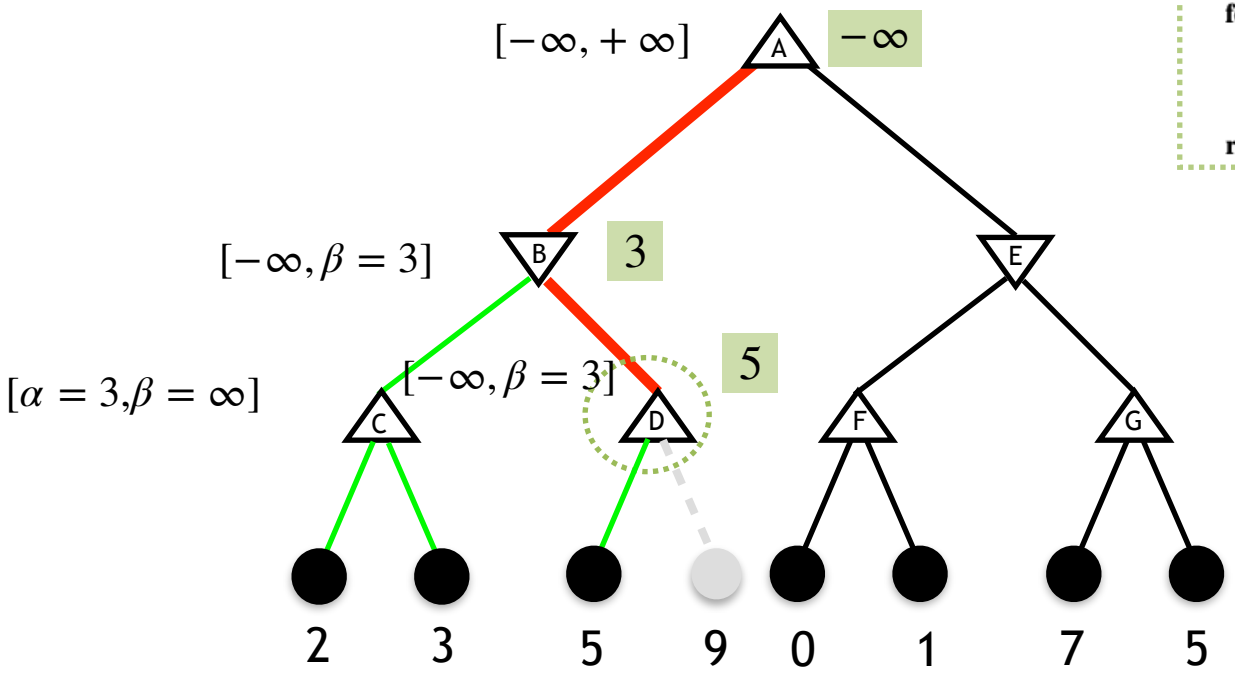
- Since **PRUNE CONDITION** is satisfied, Node **D** will prune its remaining branches

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$

```
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
```



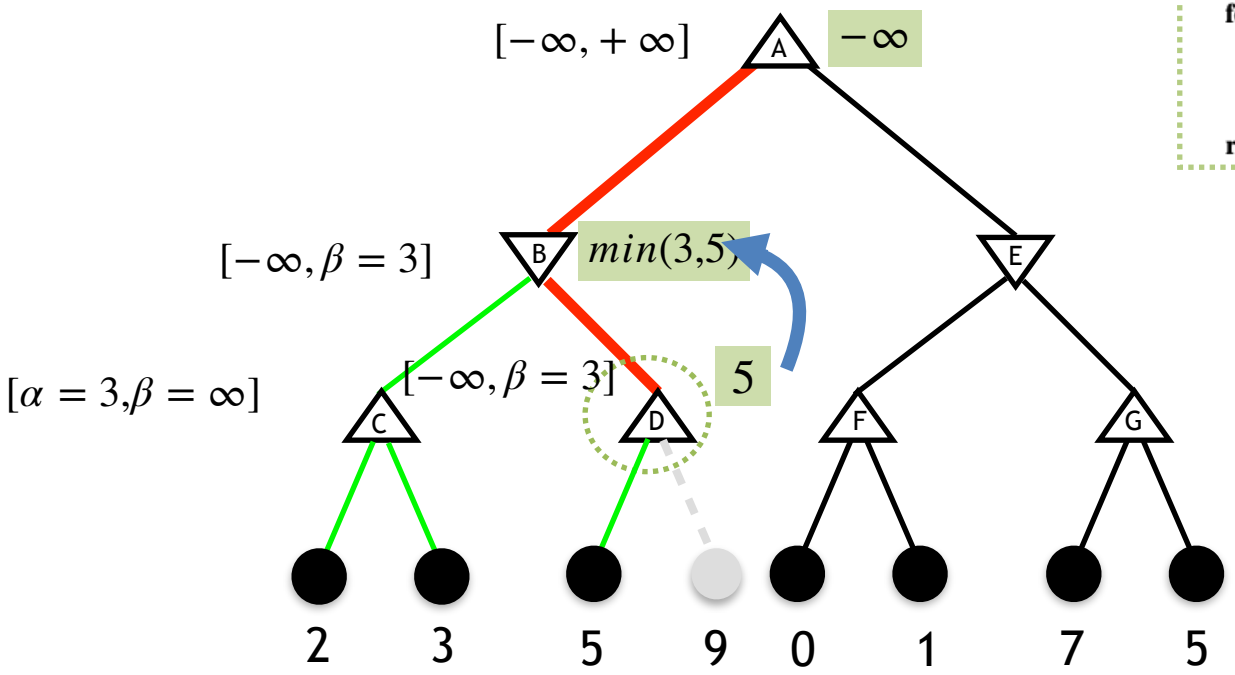
- Node **D** prunes its other branches

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$

```
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
```



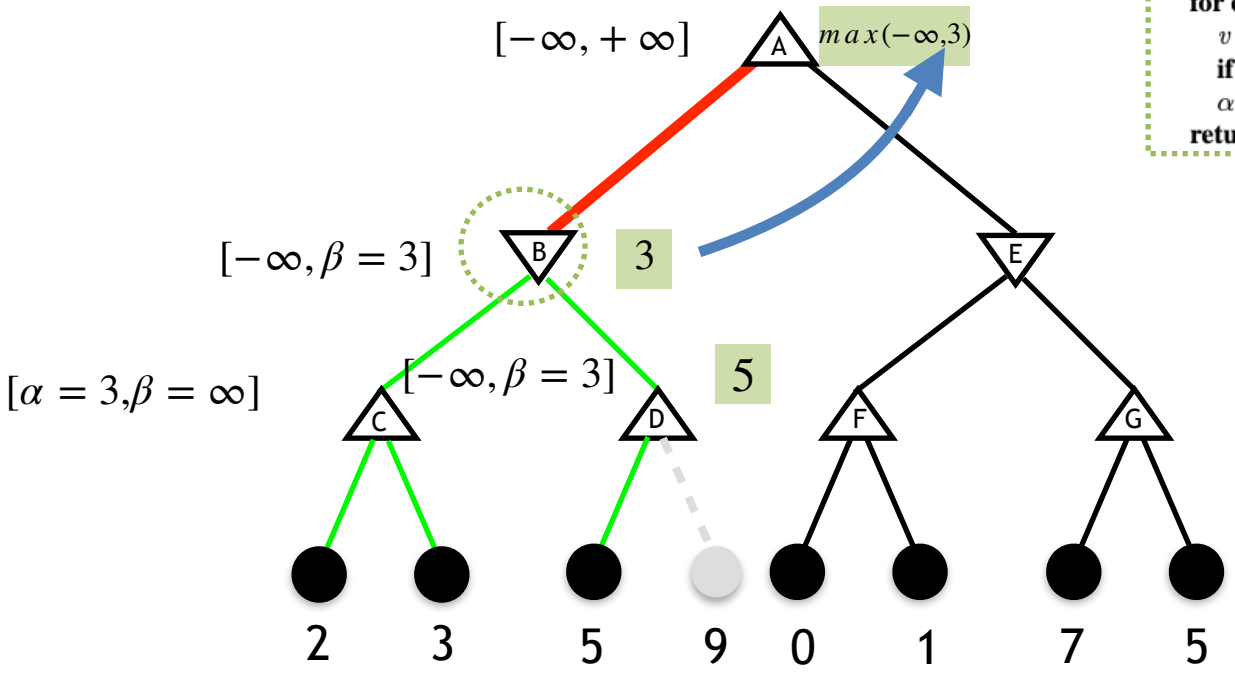
- Node **D** returns the value **5** to its parent MIN node **B**

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$

```
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
```



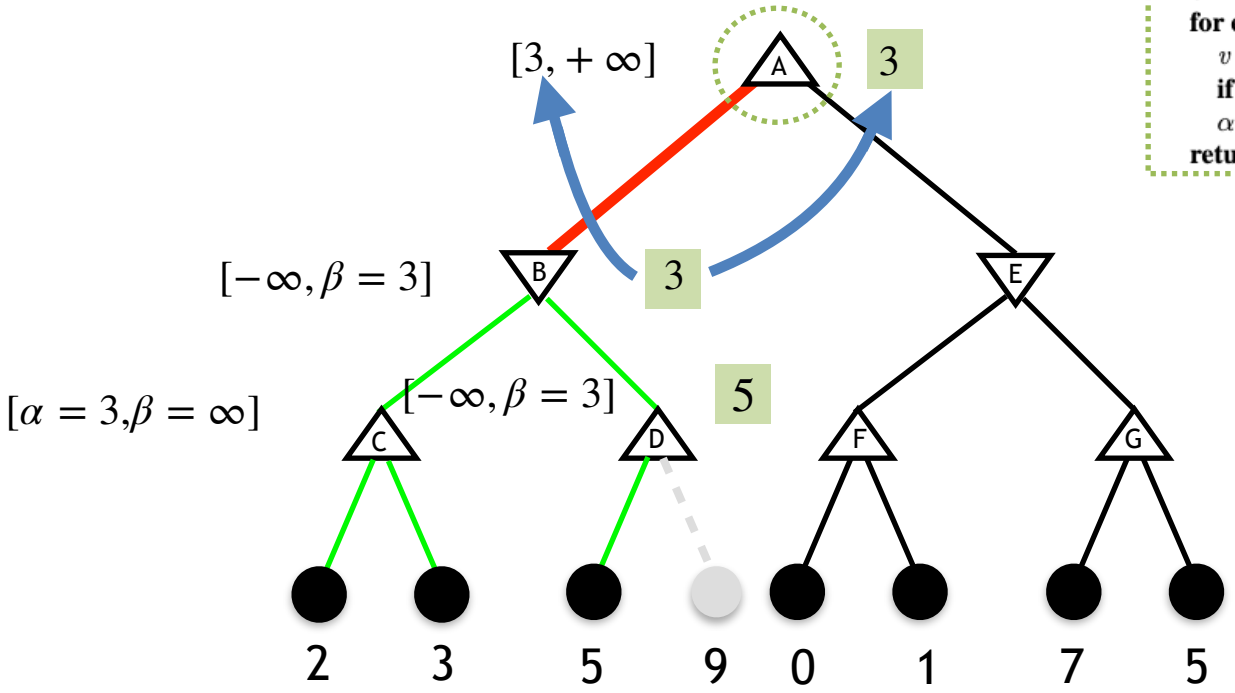
- Node **B** retains its value to **3** as min(3, 5) = 3
- Node **B** returns the value **3** to its parent MAX node **A**

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$

```
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
```



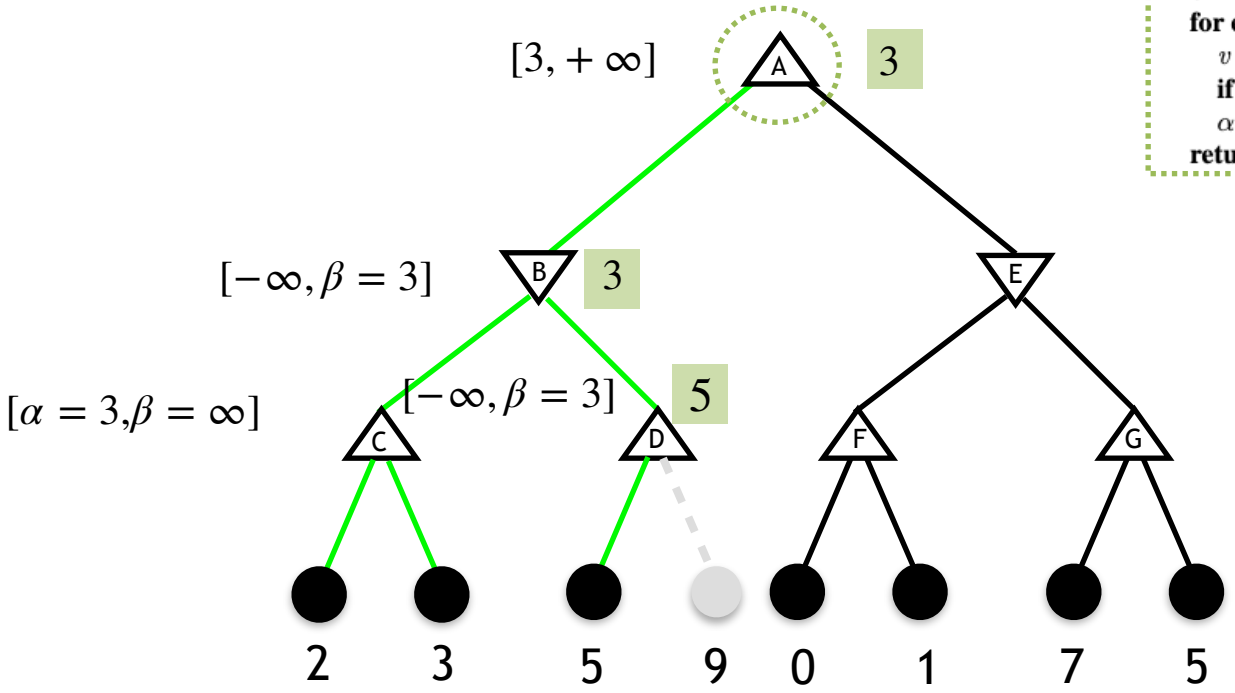
- Node **A** receives value **3** from its left MIN node and updates it value to **3** as $max(-\infty,3)$
- Node **A** will update its $\alpha = 3$
- Since **A** is a MAX node, it will not update its β value. It remains at $\beta = +\infty$

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$

```
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
```



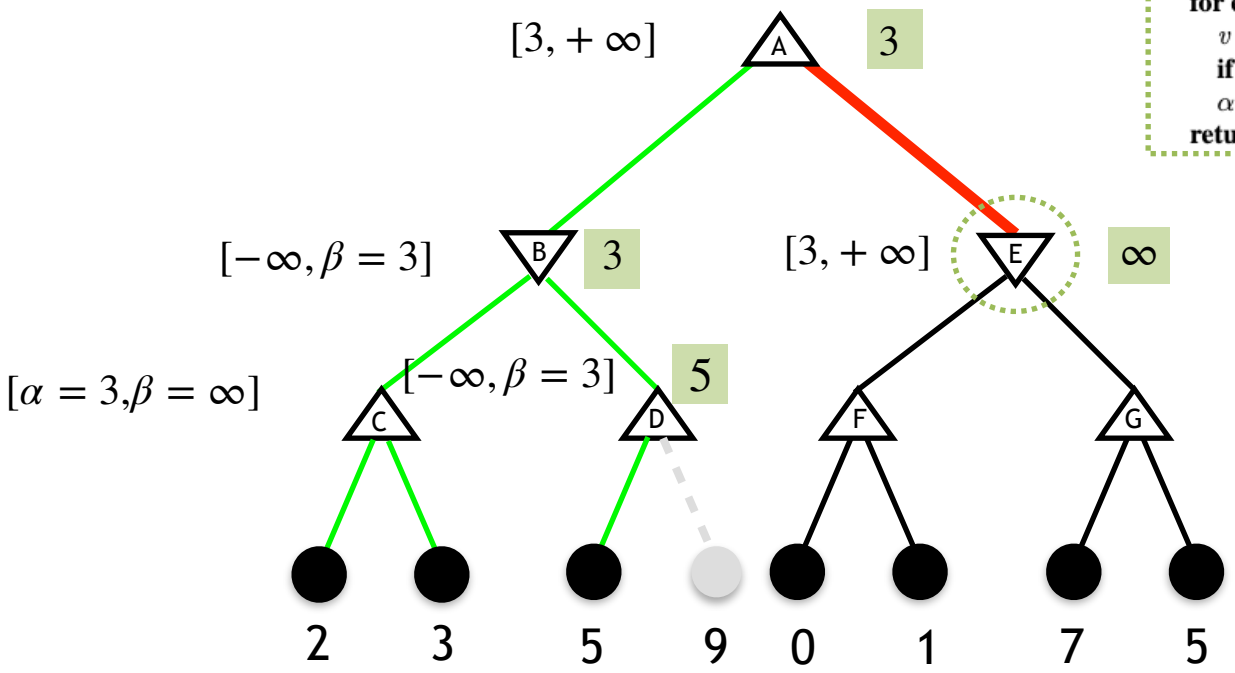
- Node **A** receives value **3** from its left MIN node and updates it value to **3** as $max(-\infty,3)$
- Node **A** will update its $\alpha = 3$
- Since **A** is a MAX node, it will not update its β value. It remains at $\beta = +\infty$
- Finally, Node **A** recursively call *function* alphabeta() on its right child **E** with $\alpha = 3$ and $\beta = +\infty$

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$

```
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
```



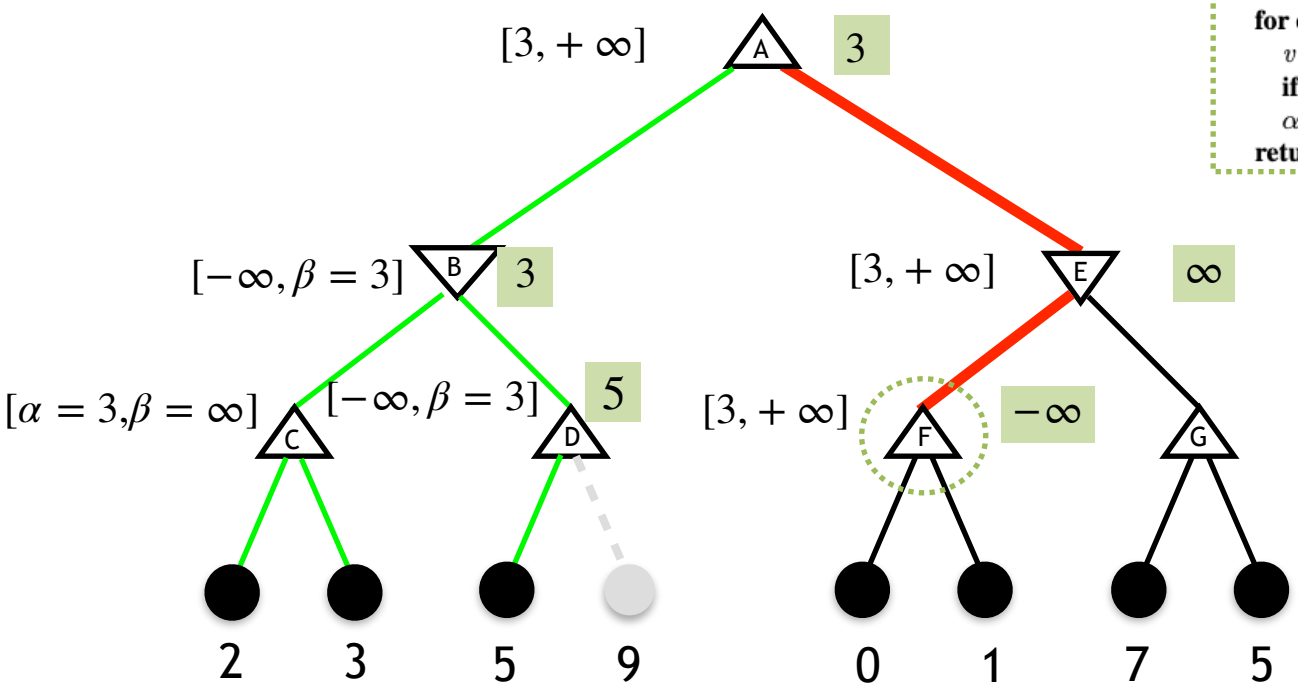
- Node **E** will initialize its $\alpha = 3$ and $\beta = +\infty$
- Node **E** is not a terminal node hence it will recursively call *function* alphabeta() on its left child **F**

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$

```
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
```



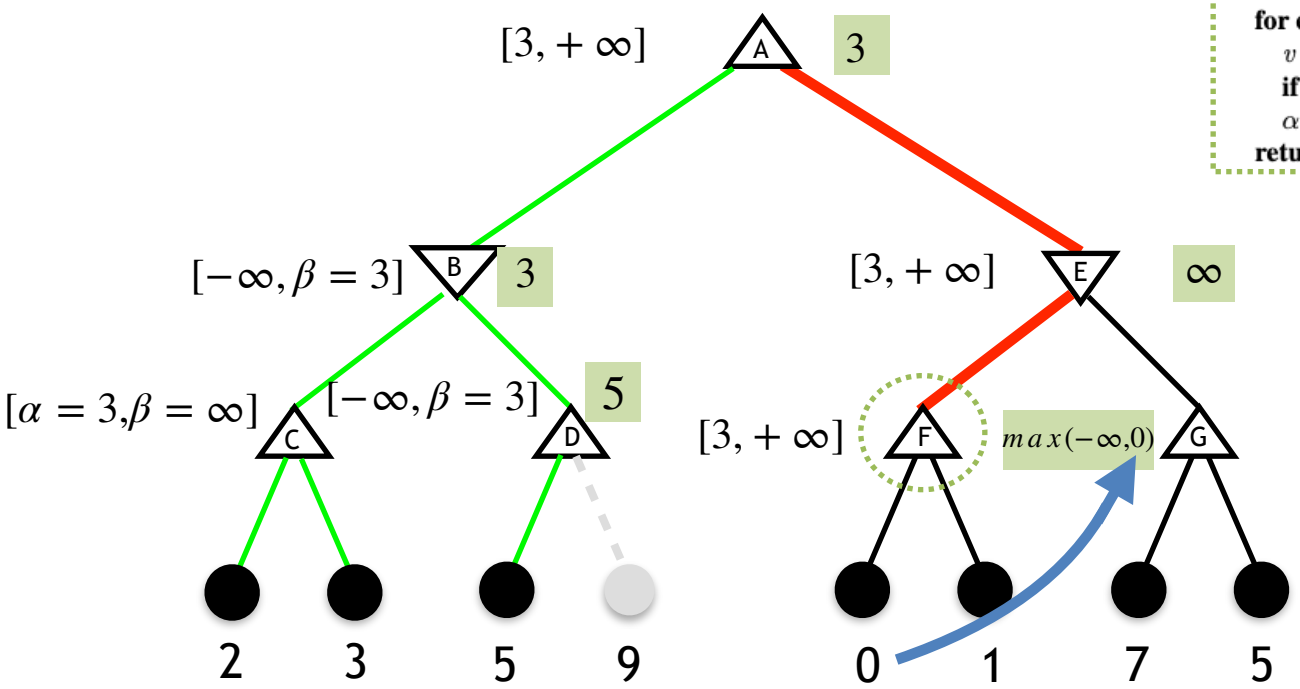
- Node **F** will initialize its $\alpha = 3$ and $\beta = +\infty$
- Node **F** is not a terminal node hence it will recursively call f*unction* alphabeta() on its left child

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$

```
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
```



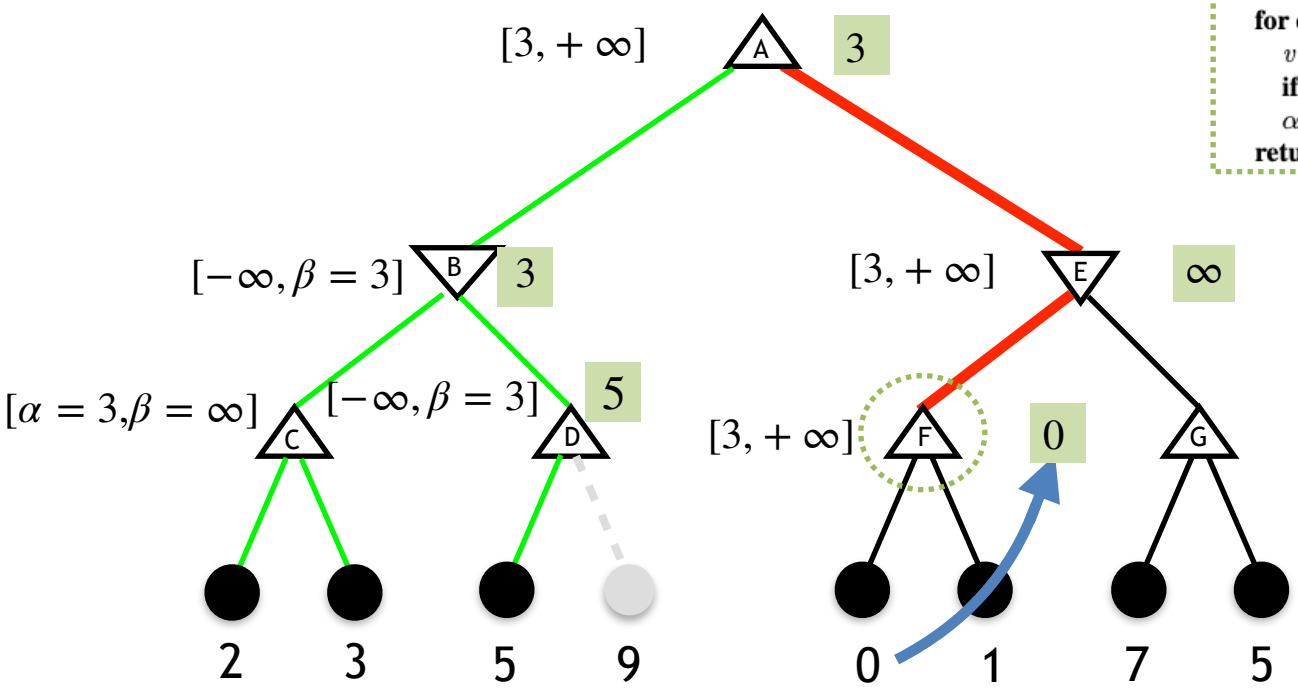
- Node **F** receives value **0** from its left terminal node; then it will set the v = $max(-\infty,0) = 0$
- Node **F** will try to update α but retains its old value $\alpha = 3$ (as $max(3,0) = 3$)
- Since **F** is a MAX node, it will not update its β value. It remains at $\beta = +\infty$

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$

```
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
```



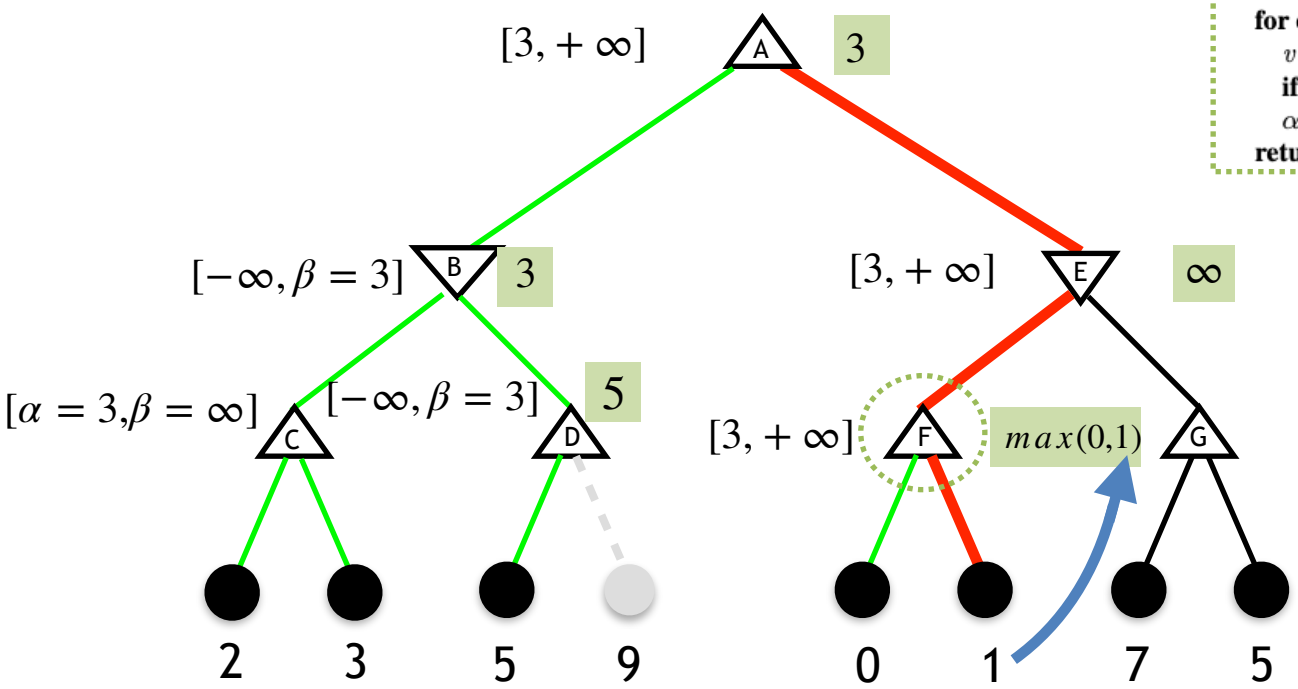
- Node **F** receives value **0** from its left terminal node; then it will set the v = $max(-\infty, 0) = 0$
- Node **F** will try to update α but retains its old value $\alpha = 3$ (as $\mathbf{max(3,0) = 3}$)
- Since **F** is a MAX node, it will not update its β value. It remains at $\beta = +\infty$

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$

```
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
```



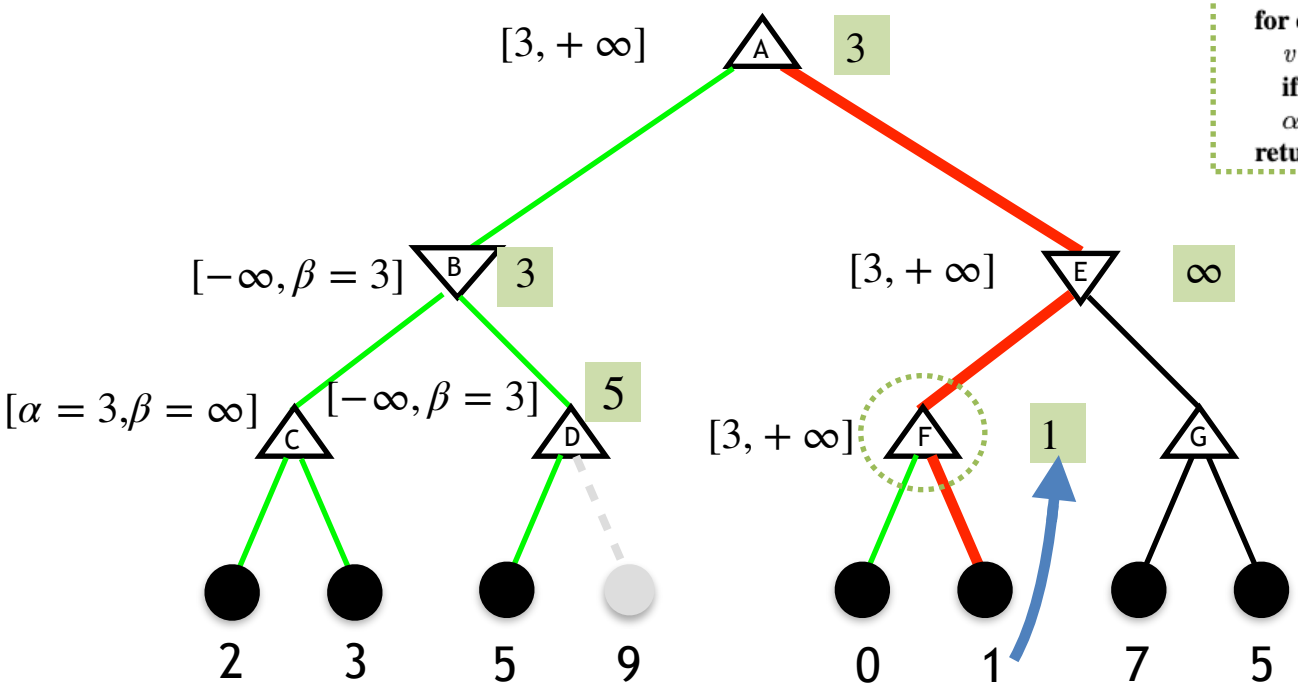
- Node **F** receives value **1** from its right terminal node; then it will set the v = $max(0,1) = 1$
- Node **F** will try to update α but retains its old value $\alpha = 1$ (as $\mathbf{max(3,1) = 3}$)
- Since **F** is a MAX node, it will not update its β value. It remains at $\beta = +\infty$

α-β Pruning Code Execution

Condition to remain valid

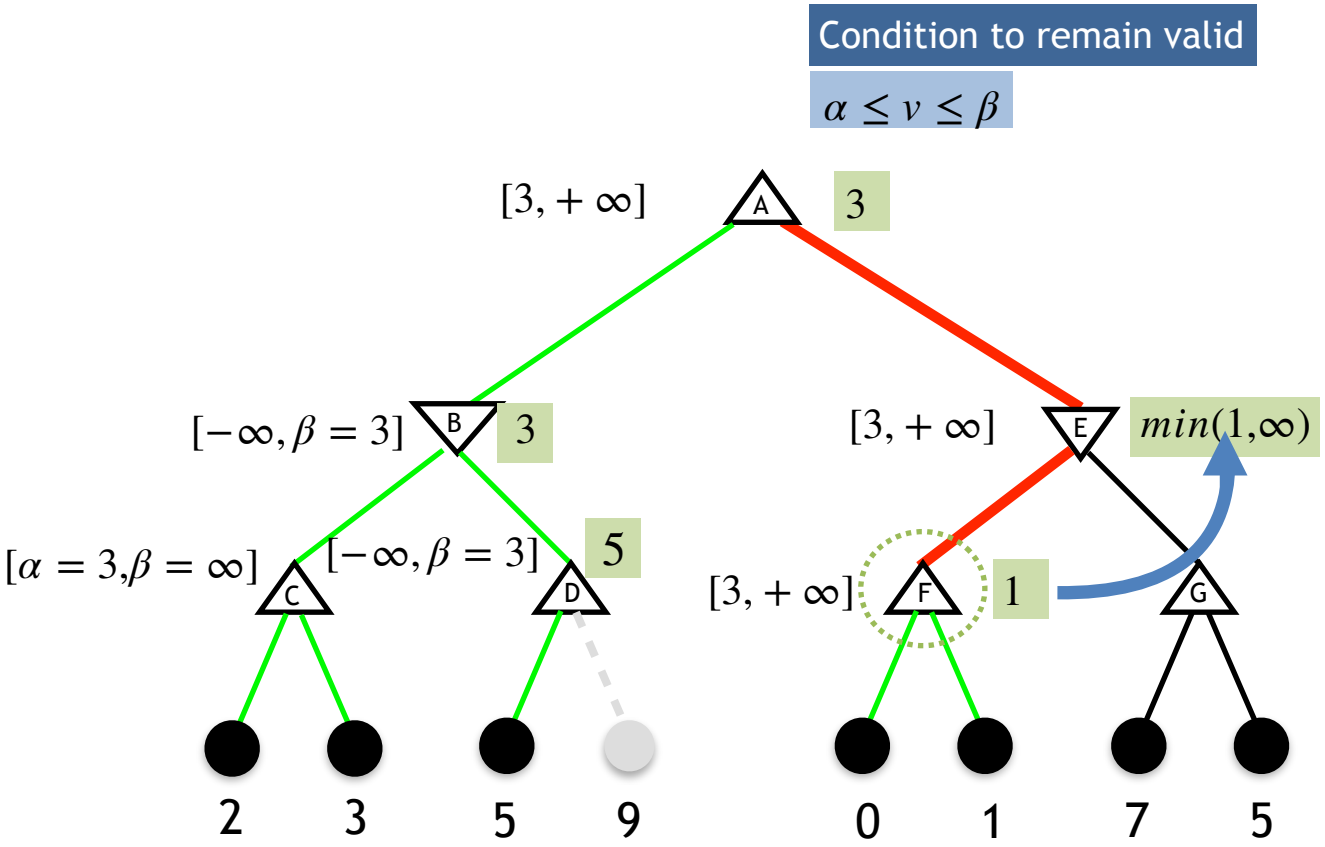
$\alpha \leq v \leq \beta$

```
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
```



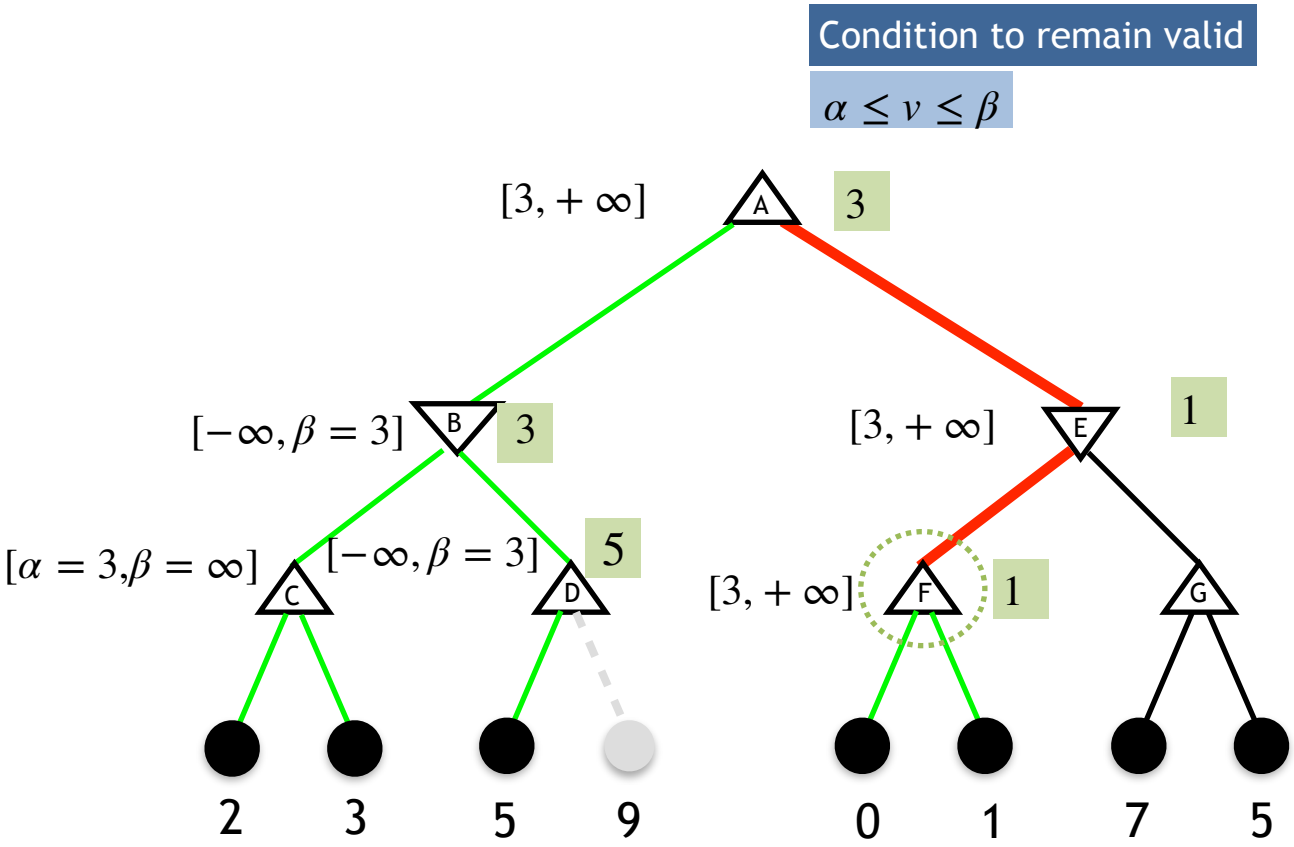
- Node **F** receives value **1** from its right terminal node; then it will set the v = $max(0,1) = 1$
- Node **F** will try to update α but retains its old value $\alpha = 1$ (as $\max(3,1) = 3$)
- Since **F** is a MAX node, it will not update its β value. It remains at $\beta = +\infty$

α-β Pruning Code Execution



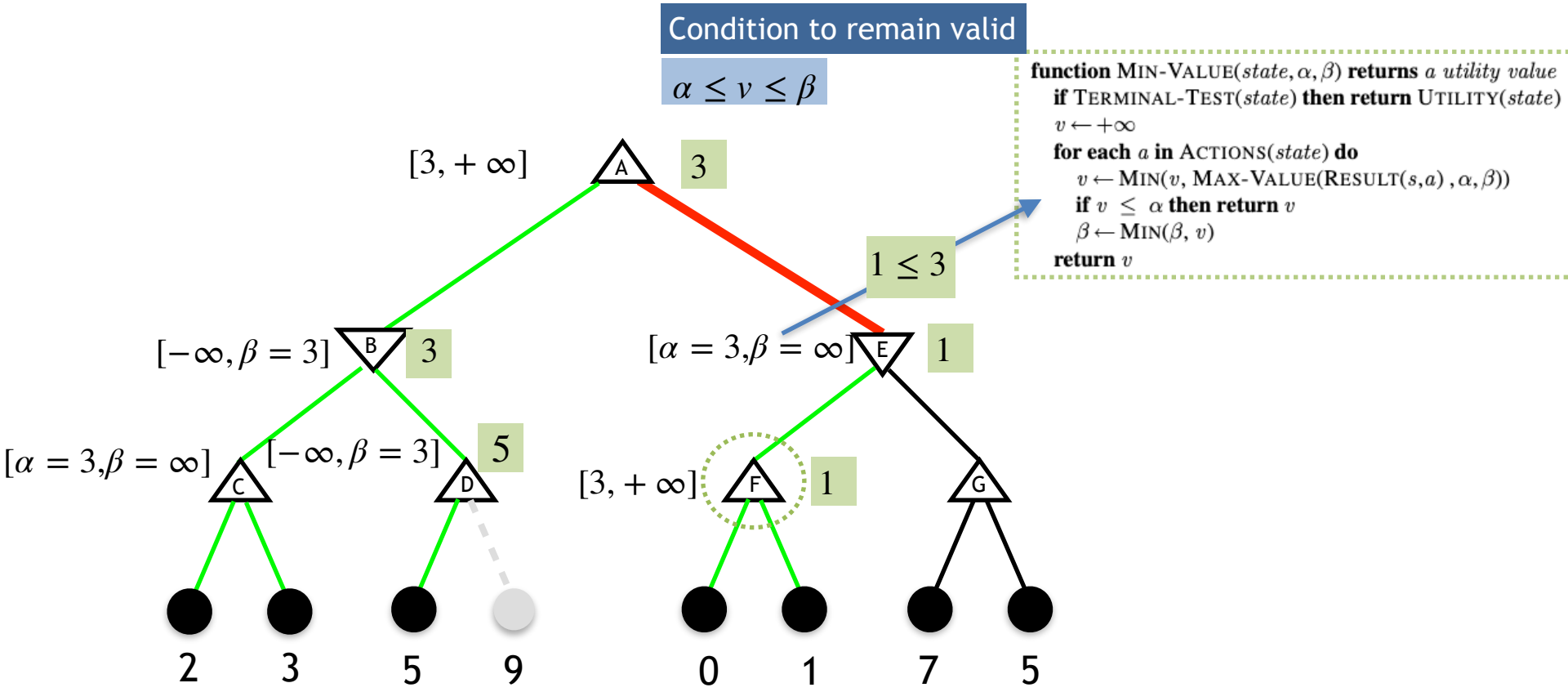
- Node **F** has no more children, it will break the loop and return value **1** to its parent MIN Node **E**

α-β Pruning Code Execution



- Node **E** receives value **1** from its left child **F**
- Node **E** will update its $\beta = 1$
- Since **E** is a MIN node, it will not update its α value. It remains at $\alpha = 3$
- Node **E** will recursively call *function* alphabeta() on its right child Node **G** with $\alpha = 3$ and $\beta = 1$

α-β Pruning Code Execution



- **E** is a MIN node, it will test the **PRUNE CONDITION**: $v \leq \alpha$

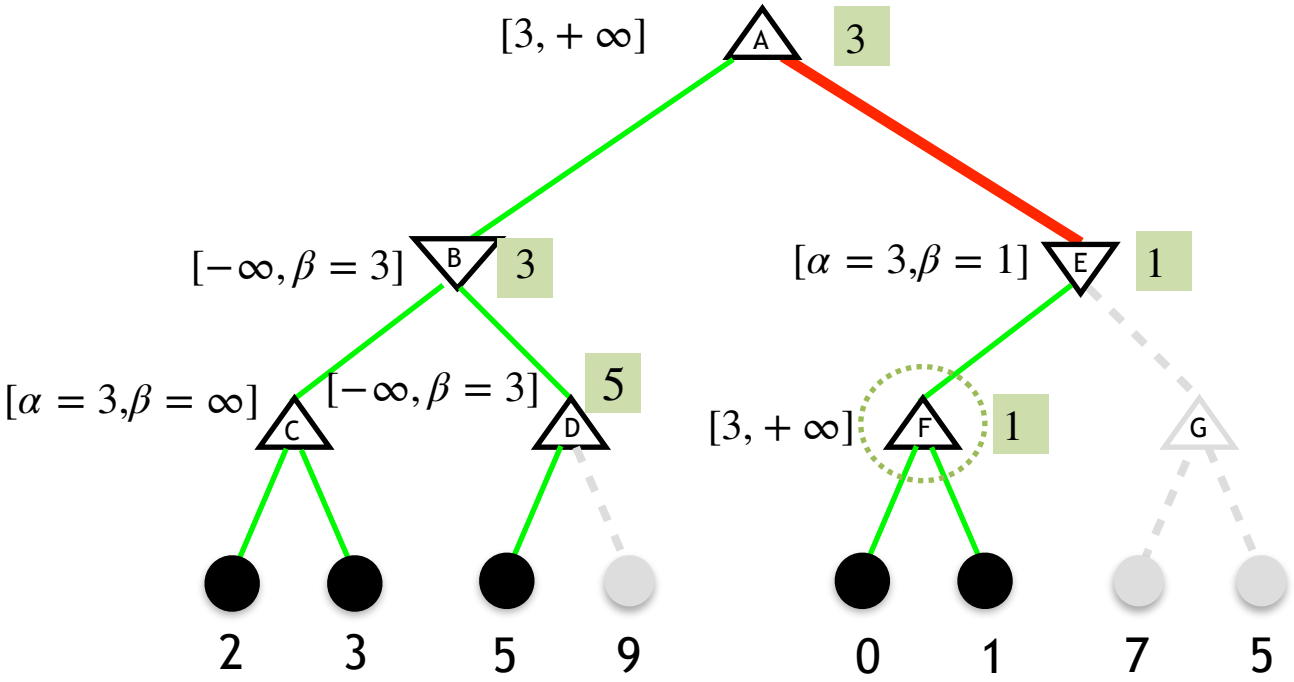
  $1 \leq 3$

- Since **PRUNE CONDITION** is satisfied, Node **E** will prune its remaining branches

α-β Pruning Code Execution

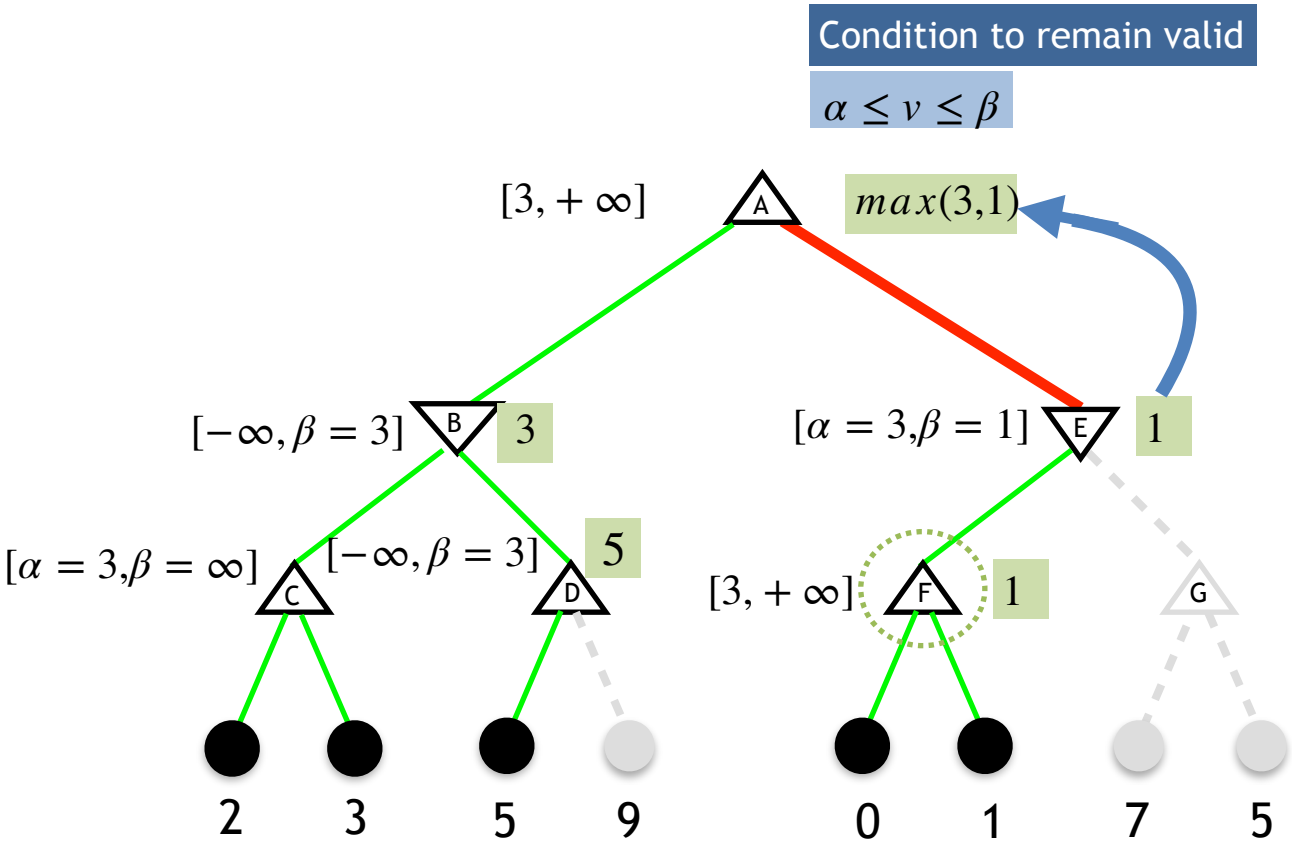
Condition to remain valid

$\alpha \leq v \leq \beta$



- Node **E** prunes its other branches

α-β Pruning Code Execution

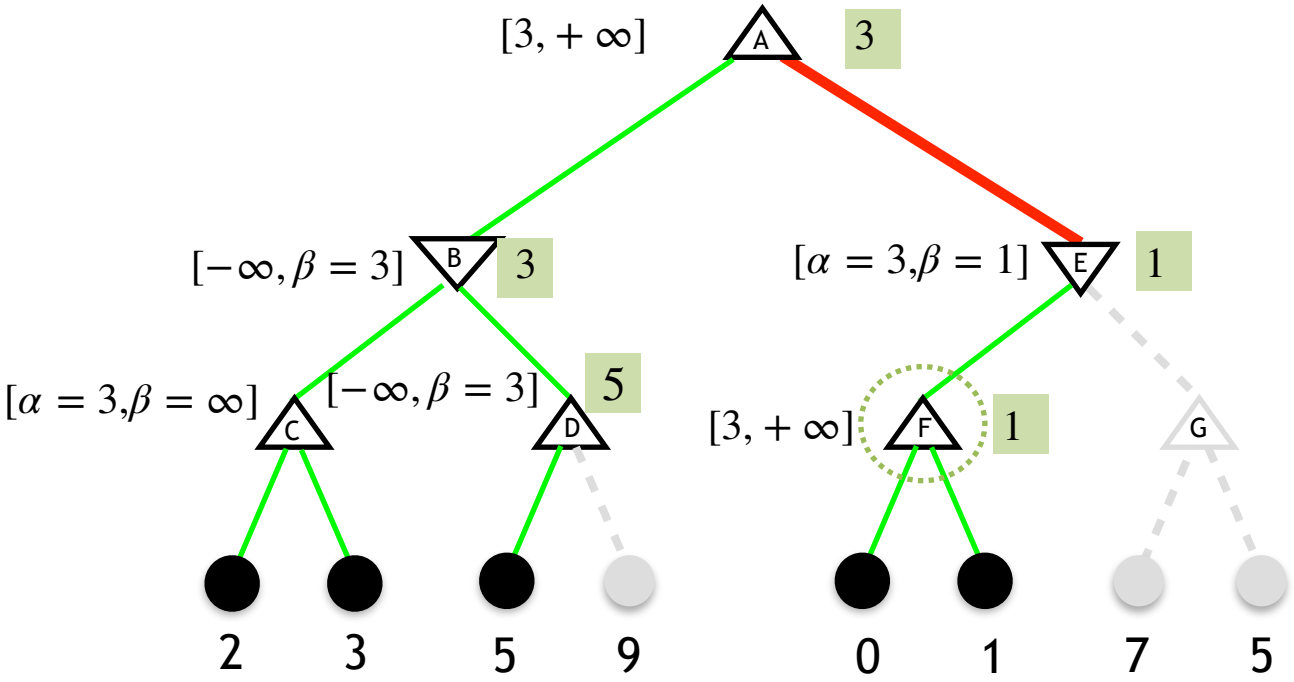


- Node **E** returns the value **1** to its parent MAX node **A**

α-β Pruning Code Execution

Condition to remain valid

$\alpha \leq v \leq \beta$



- Node **E** returns the value **1** to its parent MAX node **A**

α-β Pruning Pseudocode: Alternate implementation

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            if value > β then
                break (* β cutoff *)
            α := max(α, value)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            if value < α then
                break (* α cutoff *)
            β := min(β, value)
        return value
```

Reference: wikipedia

α-β pruning Activity

Properties of α-β Pruning

- Pruning does *not* affect the final result of the game

- Good *move ordering* improves effectiveness of pruning
  - The order in which you examine leaf nodes can make a difference
  - We can use a heuristic to determine which node to examine first

Properties of α-β Pruning

- Pruning does *not* affect the final result of the game
- Suppose the maximum depth of the search tree is m and the number of choices at each step is at most b
- With "perfect ordering", time complexity $O(b^{\frac{m}{2}})$
  - Thus we have cut the exponent in half or equivalently replaced the branching b factor to $\sqrt{b}$
  - Unfortunately, going from 10^{40} to 10^{20} still results in an infeasible situation

Imperfect Real-time Decision

- Using α-β pruning, we still have to search all the way to terminal states
  - not practical for large search spaces

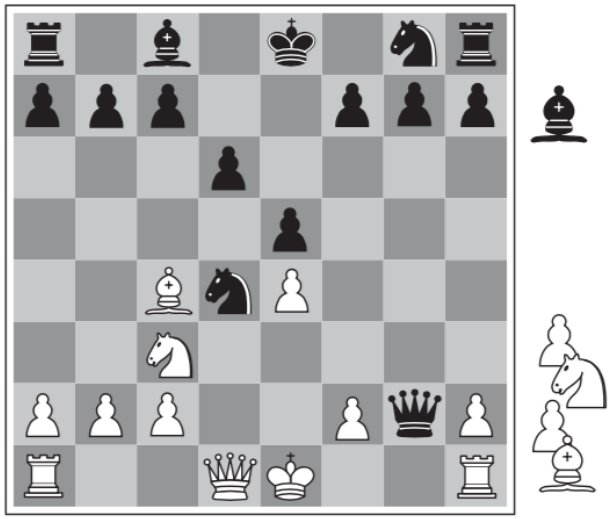
- Idea: Replace the utility function with something like a heuristic function
  - Doing so treats nonterminal nodes as terminal leaves
  - In addition, we replace our terminal test to a cutoff test to determine when to apply the heuristic (for instance, at a certain depth in the tree)

Imperfect Real-time Decision

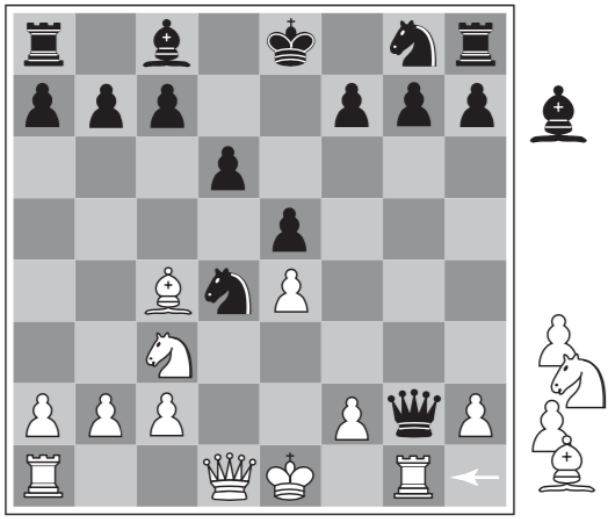
- An **evaluation function** is an estimate (heuristic) of the distance to the goal
- You can think of the function as providing an estimate of the chances of winning
- Most evaluation functions are calculated in terms of various features of the current state

Imperfect Real-time Decision

- Most **evaluation functions** are calculated in terms of various features of the current state



(a) White to move



(b) White to move

For chess, typically linear weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (number of black queens), etc.

Games of Chance and Expectation

Games of Chance

- Games that incorporate **nondeterminism (randomness)**, decisions not controlled by any player, e.g. through:
  - Dice: backgammon, Yahtzee, craps, …
  - Shuffled cards: poker, blackjack, …
  - Spinners or wheels: roulette, Life, …

- How is this different from the games we saw before?
  - Chess is also non-deterministic *for a given player,* so we assumed that the other player would make *their best move*
  - How to handle "real" nondeterminism?

Expected Values

- The ***expected value*** of a random event is the average value produced by that event with respect to the probability of each possible outcome

- In terms of equation, the expected value of a (discrete) random variable X

$$E[X] = \sum_{x \in Val(X)} xP(x)$$

Expected Values

- **Example 1:** The expected number of dots showing on a six-sided die with random variable X is denoted by E[X]:

$$\left(1 \cdot \frac{1}{6}\right) + \left(2 \cdot \frac{1}{6}\right) + \left(3 \cdot \frac{1}{6}\right) + \left(4 \cdot \frac{1}{6}\right) + \left(5 \cdot \frac{1}{6}\right) + \left(6 \cdot \frac{1}{6}\right) = 3.5$$

- **Example 2:** Let X be the amount of money you win in the $133 million Powerball Jackpot (March 2026). The probability of winning is about 1 in 300 million. What is E[X]?

$$0 + \frac{133}{300} = 0.44 Million$$

Expectiminimax

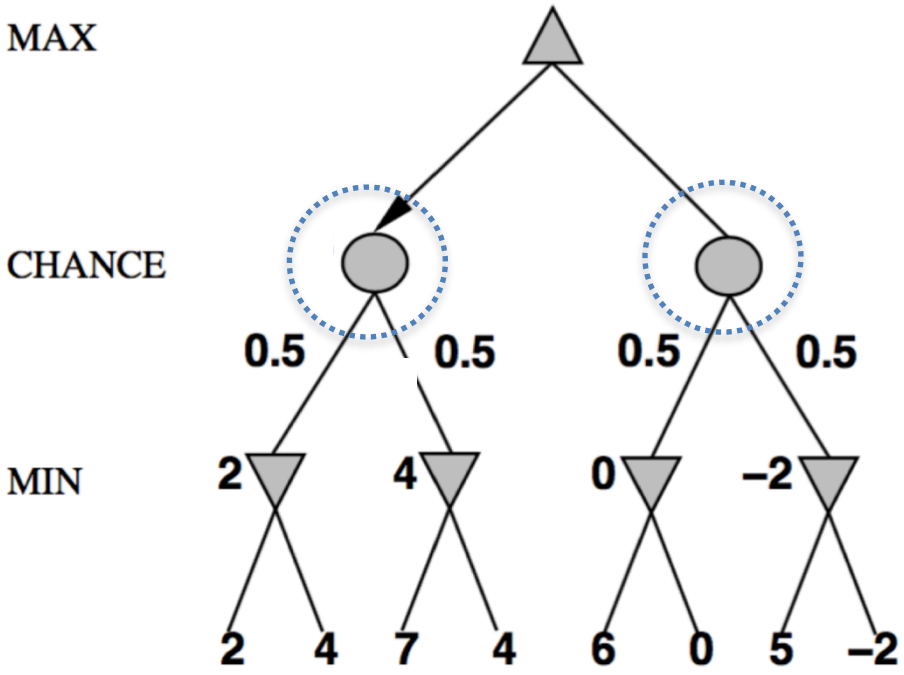
- How to make *rational* decisions in game of chance?

- In this non-deterministic setup, ***chance*** is introduced by
  - coin-flipping
  - rolling a dice
  - shuffling cards, etc

- So, add a ***chance node*** within a minimax tree
  - *chance node* is the *expected value* of all possibilities

Expectiminimax

- We can apply a version of minimax by computing expected values at chance nodes



Expectiminimax Algorithm

- The utility of a MAX/MIN node in the game tree is the **max/min** of the utility values of its successor

- The expected utility of a CHANCE node is the **expected value** of the utility values of its successors

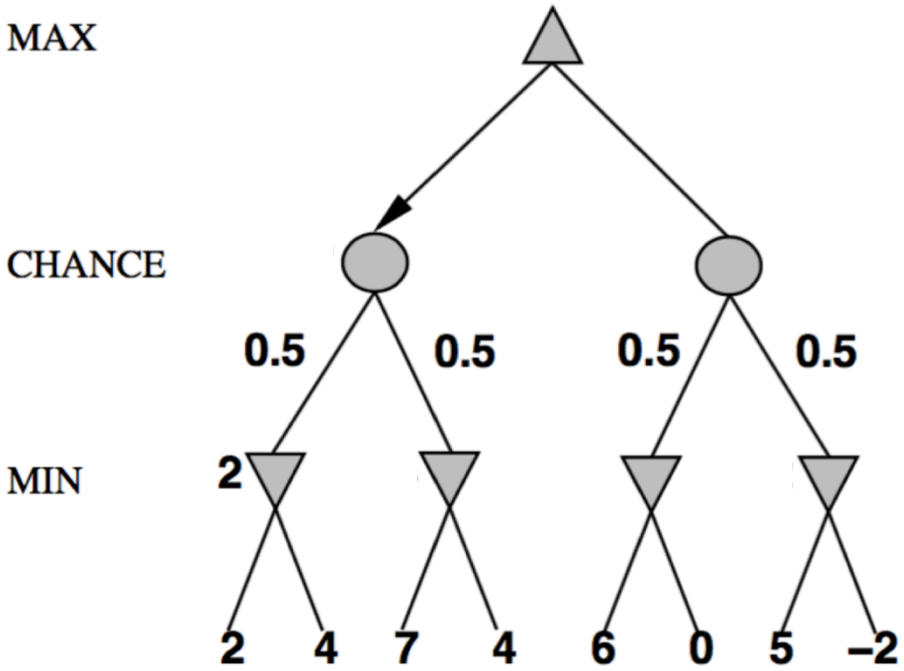
$$\text{ExpectedValue}(s) = \sum_{s' \in \text{SUCC}(s)} \text{ExpectedValue}(s')\, P(s') \qquad \textbf{CHANCE nodes}$$

$$\text{MinimaxValue}(s) = \max_{s' \in \text{SUCC}(s)} \text{MinimaxValue}(s') \qquad \textbf{MAX nodes}$$

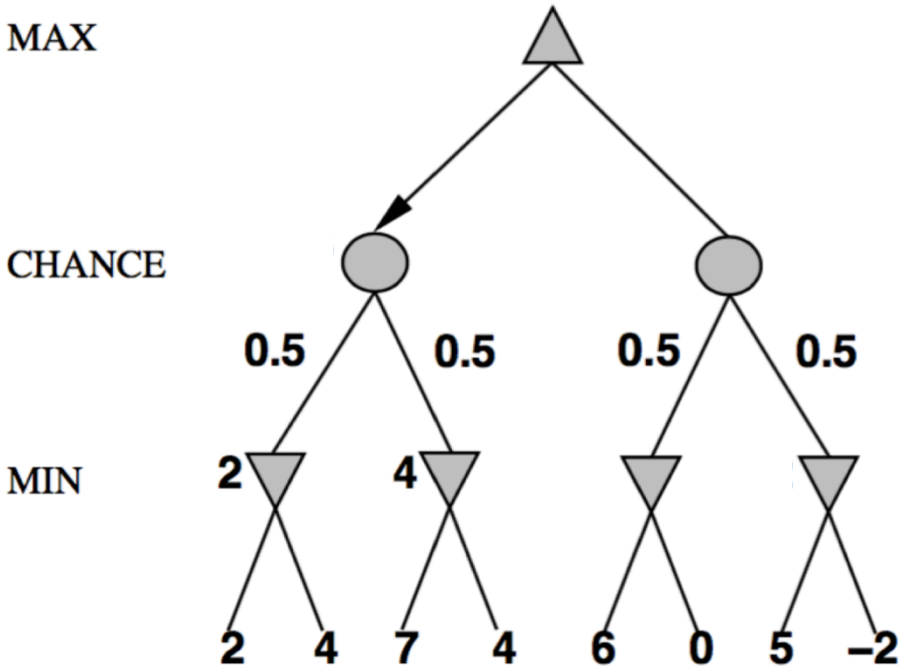
$$\text{MinimaxValue}(s) = \min_{s' \in \text{SUCC}(s)} \text{MinimaxValue}(s') \qquad \textbf{MIN nodes}$$

Expectiminimax Example

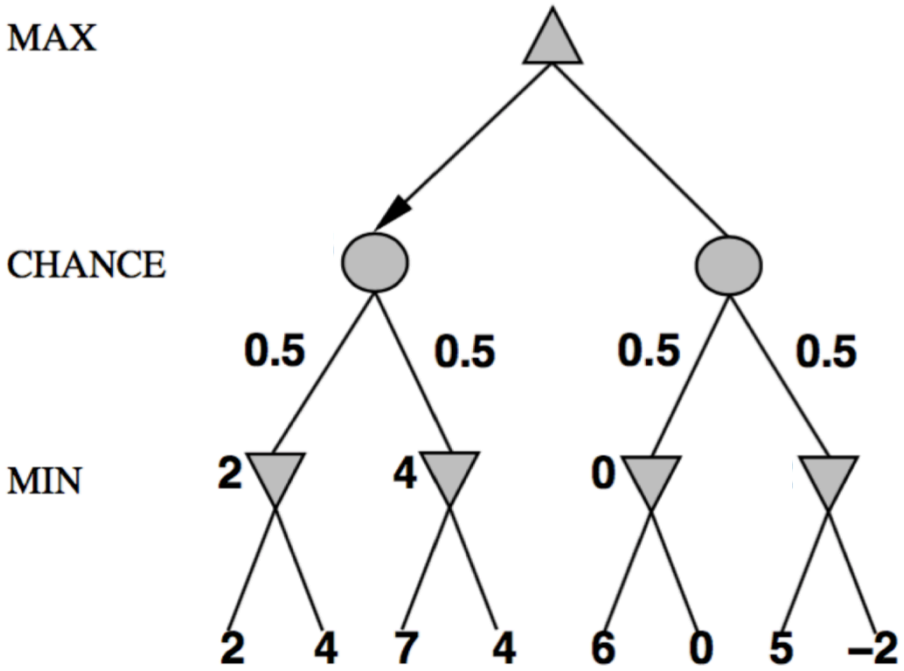
Chance nodes are introduced by coin-flipping



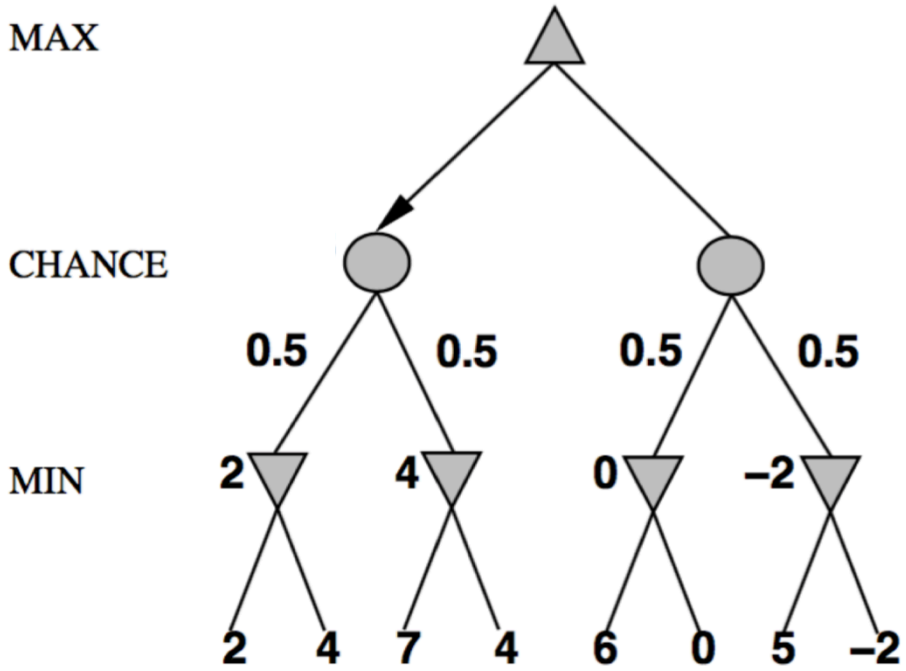
Expectiminimax Example



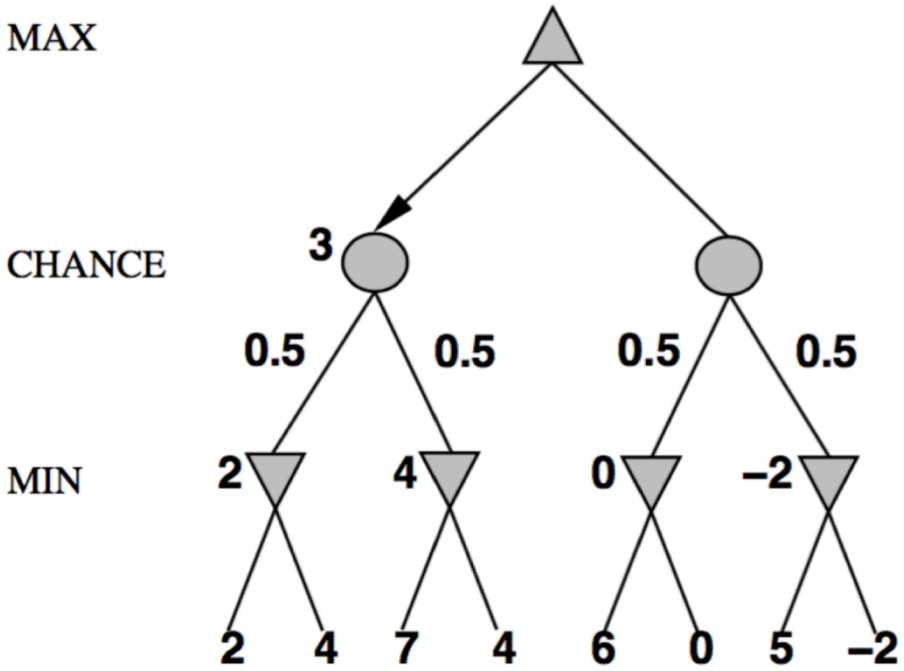
Expectiminimax Example



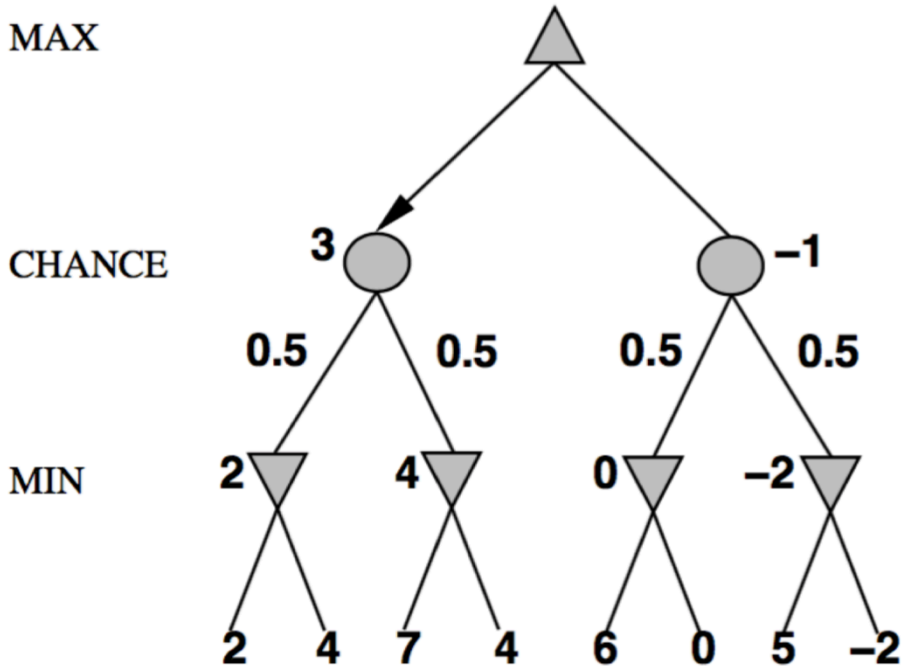
Expectiminimax Example



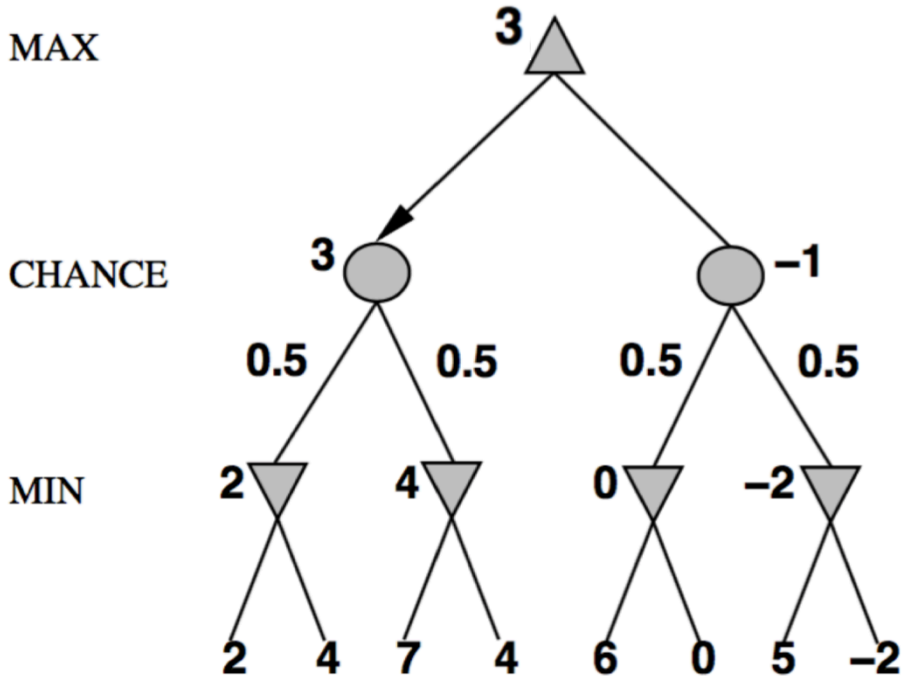
Expectiminimax Example



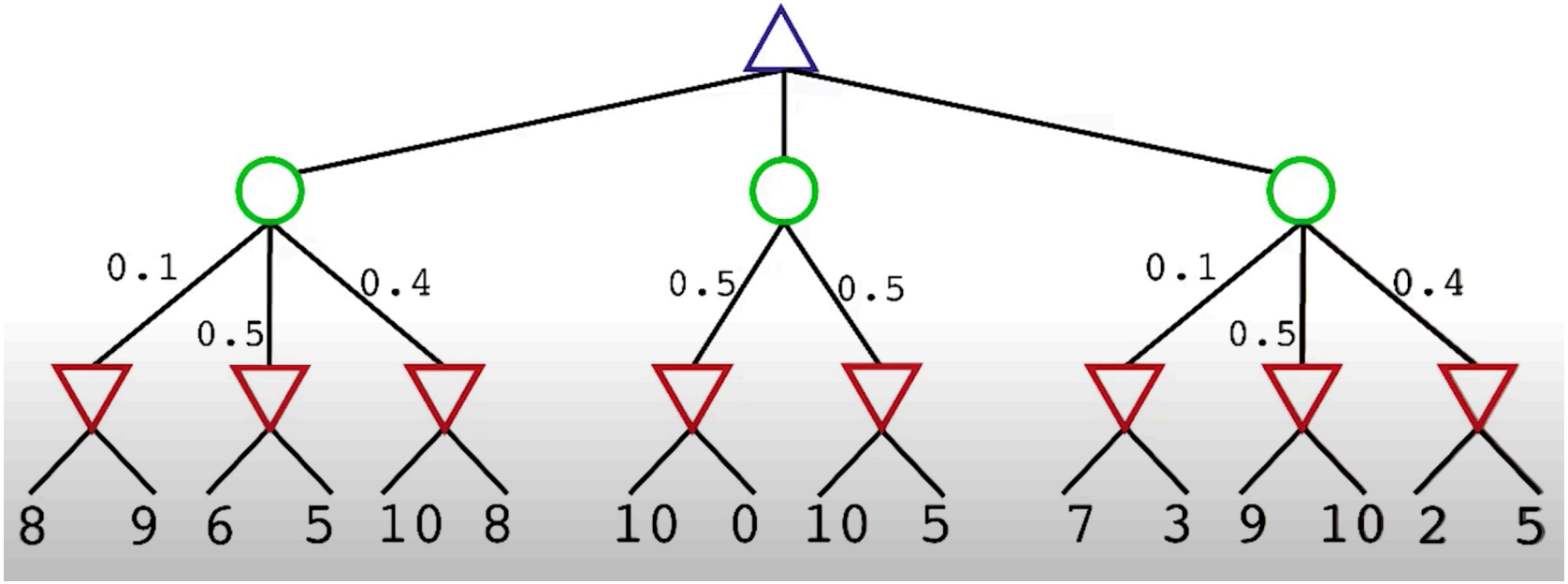
Expectiminimax Example



Expectiminimax Example



Expectiminimax Activity



Adversarial Search

- Discussed three different of Adversarial Searches
  - Minimax
  - Alpha-beta pruning
  - Expectiminimax
- Next Assignment will be based on Adversarial Search to find optimal move in a game
  - Minimax
  - Alpha-beta pruning
- Read chapter 6 (Russel and Norvig Textbook)

Project Assignment#4

Adversarial Search to solve Connect Four