# Adversarial Search and Game Theory

## Lecture 10: Artificial Intelligence

Copyright ©2025. All Rights Reserved.

Dr. Bonaventure Chidube Molokwu

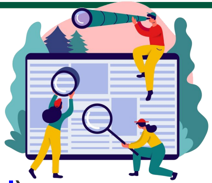
bonaventure.molokwu[at]csus.edu

College of Engineering and Computer Science
California State University
Sacramento
California - United States of America

1. **Lesson/Week Outline:**
   1.1 Adversarial Search
   1.2 Game Theory

2. **Learning Outcomes:**
   2.1 Finding optimal decisions in games via `Minimax` Decision, $\alpha - \beta$ Pruning, Monte Carlo Tree Search (MCTS) algorithms.
   2.2 Resource limits and approximate evaluations.
   2.3 Games of chance
   2.4 Games of imperfect information.

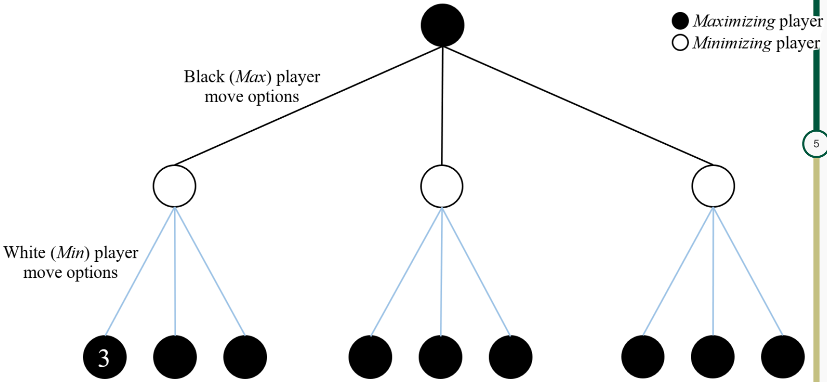
# Adversarial Search and Game Theory
## Prelude



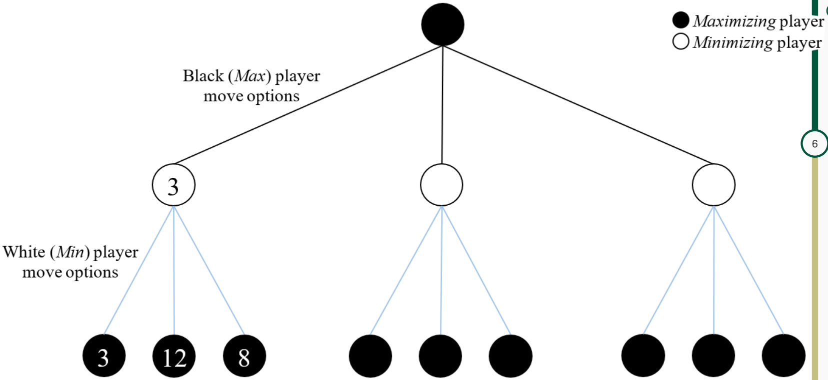
- **search(Adversarial):**
  - Refer to competitive and/or search environments, in which 2/more agent(Problem-Solving) have conflicting goals.
- **Related Terminologies wrt. theory(Game):**
  - term(Move) is a synonym for term(Action) taken by a player.
  - term(Position) is a synonym for term(State) reached by a player.
  - function(Fitness/Utility/Pay-off/Objective): determines the value(Final) for a player, $p$, when a game ends in terminal state, $s$. In *chess*, it can be win(+1), loss(-1), or draw(0). Some games have a wider range of possible outcomes.
  - game(Zero-Sum): a player's `gain` is EXACTLY balanced by the opponent's (other player) `loss`. In other words, what is `good` for a player is `bad` for the opponent (other player). Thus, there is no "win-win" outcome.

Formally, game(Zero-Sum) = total(Gain) + total(Loss) = 0

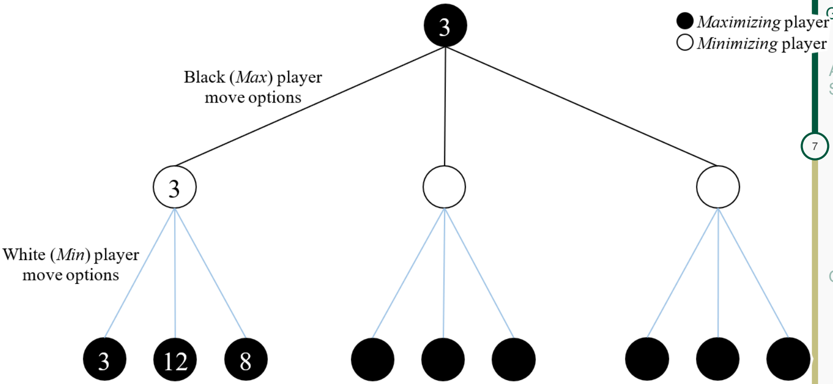
- **search(Minimax):**
  - A algorithm(Decision-Making) employed in 2-player, turn-by-turn games like chess, checkers, etc.
  - Usually applied in games that require outcomes(Multiple).
  - player(Maximizing): aims at moving to a state of *maximum value*, wrt. function(Fitness/Utility/Pay-off/Objective), during its turn.
  - player(Minimizing): aims at moving to a state of *minimum value*, wrt. function(Fitness/Utility/Pay-off/Objective), during its turn.
  - principle(Operation): Leaves or nodes(Terminal) are evaluated, by means of a function(Fitness/Utility/Pay-off/Objective), for `Utility values`. Thereafter, it recursively backs up these `Utility values` through the tree. Finally, the node(Root)'s `best value` determines the move(Optimal) wrt. the player at the node(Root).



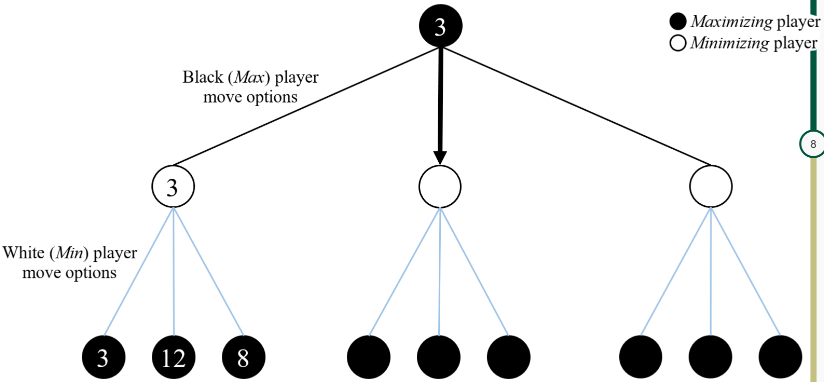

- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.
  - The game ends after 1 `move` from player(Maximizing) AND 1 `move` from player(Minimizing).



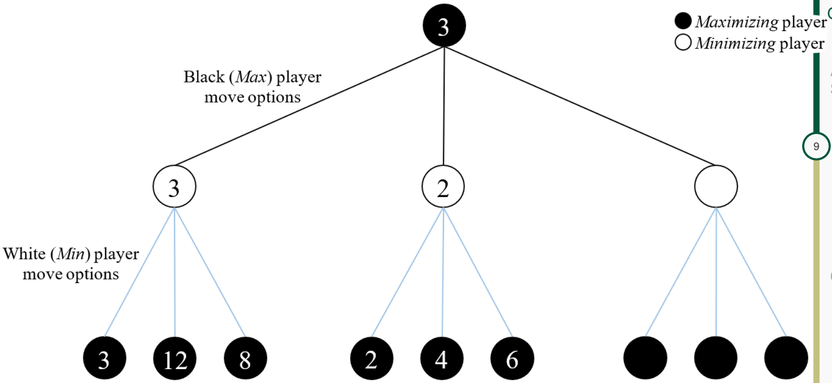

- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.
  - The game ends after 1 `move` from player(Maximizing) AND 1 `move` from player(Minimizing).



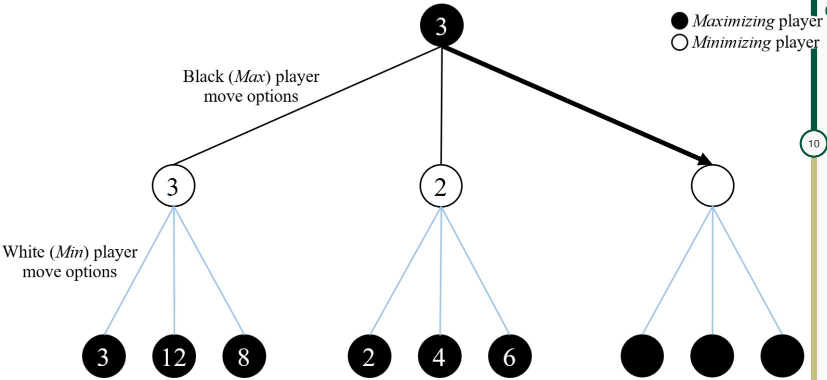

- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.
  - The game ends after 1 `move` from player(Maximizing) AND 1 `move` from player(Minimizing).



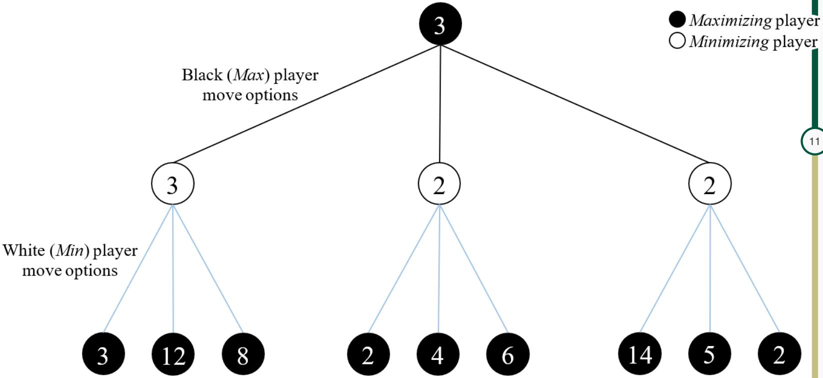

- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.
  - The game ends after 1 `move` from player(Maximizing) AND 1 `move` from player(Minimizing).




- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.
  - The game ends after 1 `move` from player(Maximizing) AND 1 `move` from player(Minimizing).




- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.
  - The game ends after 1 `move` from player(Maximizing) AND 1 `move` from player(Minimizing).




- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.
  - The game ends after 1 `move` from player(Maximizing) AND 1 `move` from player(Minimizing).

**Algorithm 1** Minimax-Search algorithm

```
function MINIMAX(node, depth, isMaxPlayer)
    if isLeaveNode(node) || depth == 0 then
        return eval(node)
    end if

    if isMaxPlayer == true then                          ▷ player(Maximizing)
        best = −∞
        for each child in node.children do
            evalRes = MINIMAX(child, depth − 1, false)
            best = max(best, evalRes)
            return best
        end for
    else                                                 ▷ player(Minimizing)
        worst = +∞
        for each child in node.children do
            evalRes = MINIMAX(child, depth − 1, true)
            worst = min(worst, evalRes)
            return worst
        end for
    end if
end function
```

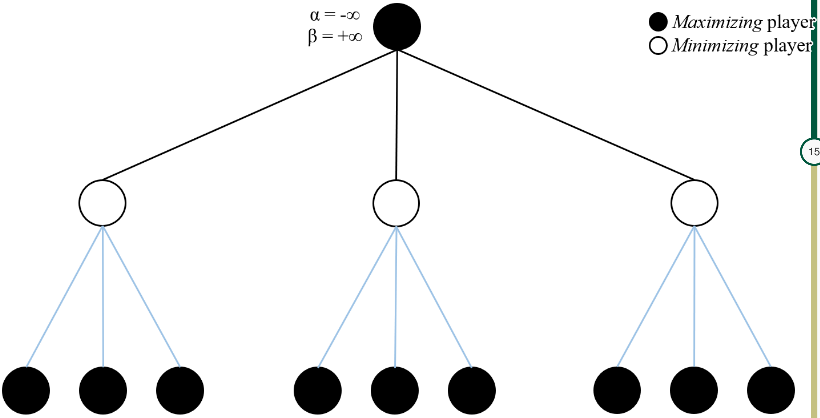
- **property(Optimality):**
  - Does yield the solution(Optimal) provided that both players play perfectly.

- **property(Run-time Complexity):**
  - $O(b^d)$: where $b$ = Branching factor (average number of children per node) AND $d$ = Depth of the optimal solution.

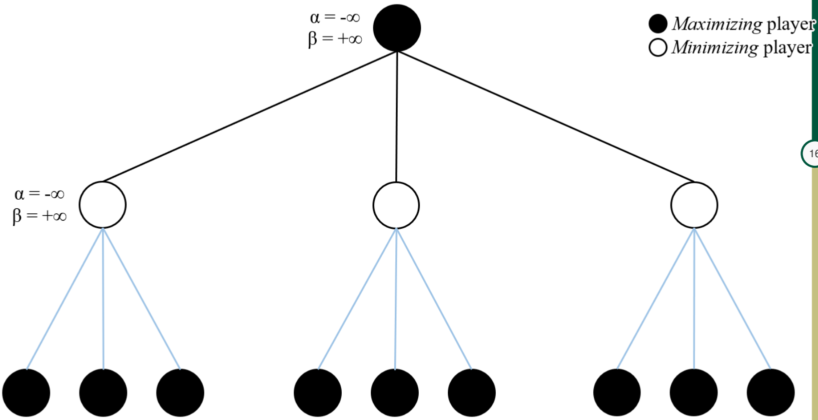
- **property(Space Complexity):**
  - $O(b \cdot d)$: where $b$ = Branching factor (average number of children per node) AND $d$ = Depth of the optimal solution.

- **property(Other):**
  - Works well in finite, deterministic games or if moves(Game) can be expressed as a tree(Finite).

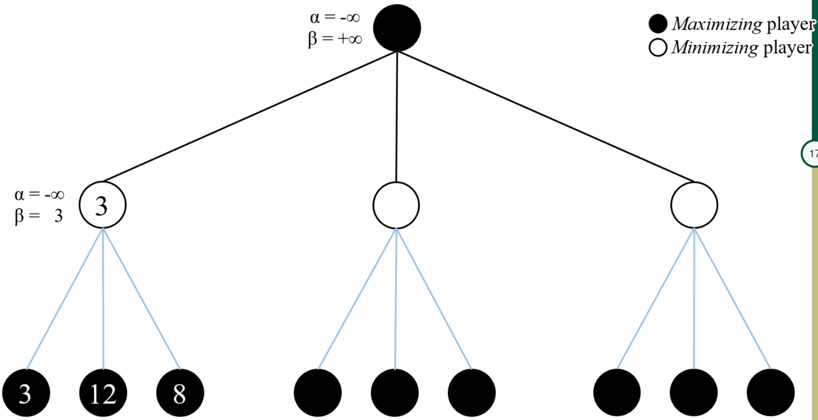
- **search(Alpha–Beta, $\alpha - \beta$):**
  - A version(Optimized) of the search(Minimax) algorithm, and it is employed as a algorithm(Decision-Making) wrt. 2-player, turn-by-turn games like chess, checkers, etc.
  - Its optimization is based on the fact it cuts off branches in a tree(Search) that do NOT affect the decision(Final) of the game.
  - This optimization improves its speed(Processing) without losing *accuracy*.
  - Alpha, $\alpha$: denotes the BEST-explored value wrt. player(Maximizing).
  - Beta, $\beta$: denotes the BEST-explored value wrt. player(Minimizing).
  - At any *move/node*, where Beta ($\beta$) $\leq$ Alpha ($\alpha$), it denotes that the current branch of the tree(Search) is WORSE than a previously explored branch - this triggers a `truncation/pruning`.
  - Aim/Goal: In evaluating a *move/node*, IF a better *move/node* has already been found or explored before, THEN stop evaluating current *move/node*.



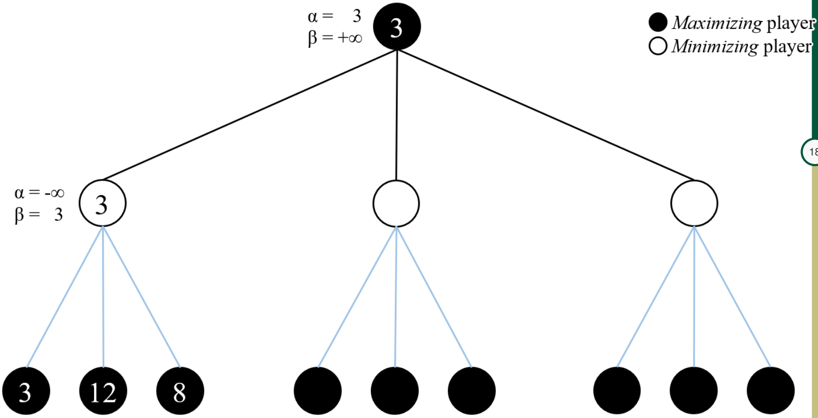

- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.




- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.



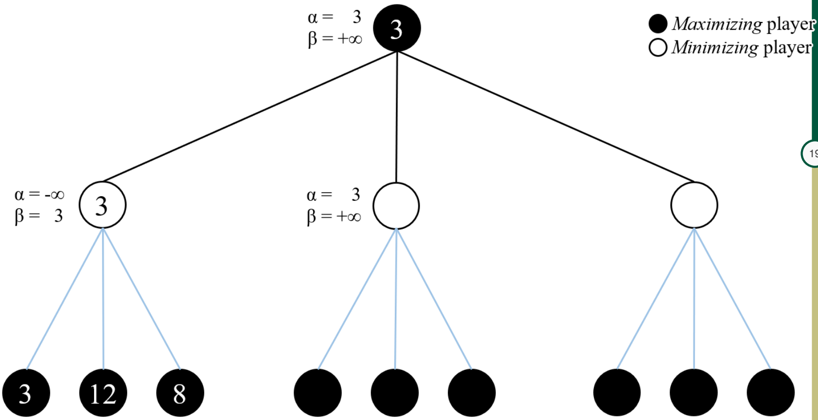

- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.



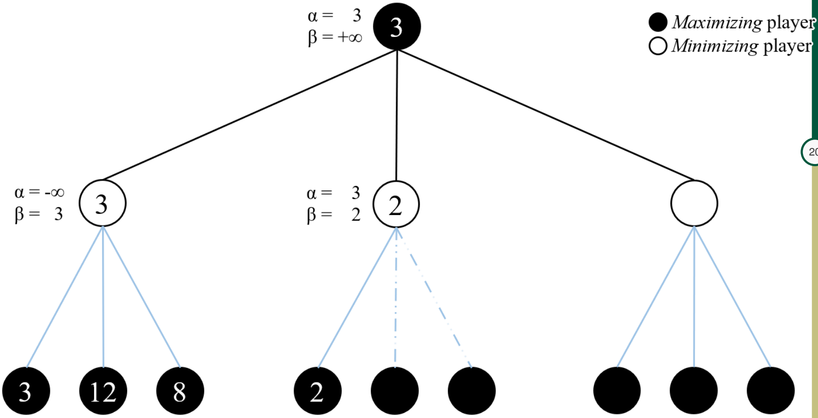

- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.

# Adversarial Search and Game Theory

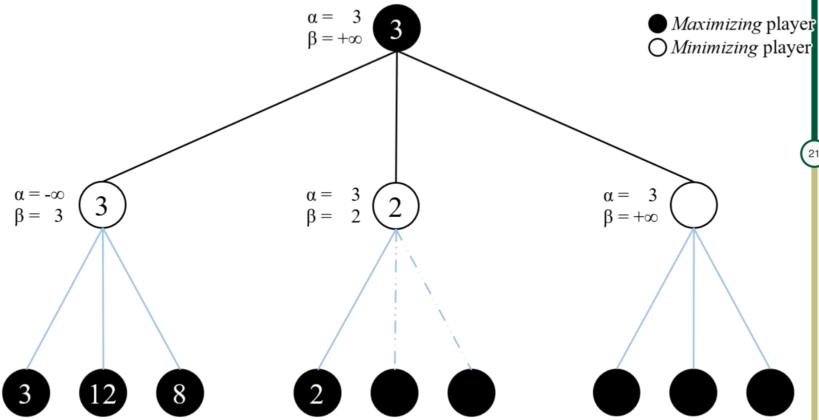
## Alpha–Beta ($\alpha - \beta$) Search Algorithm



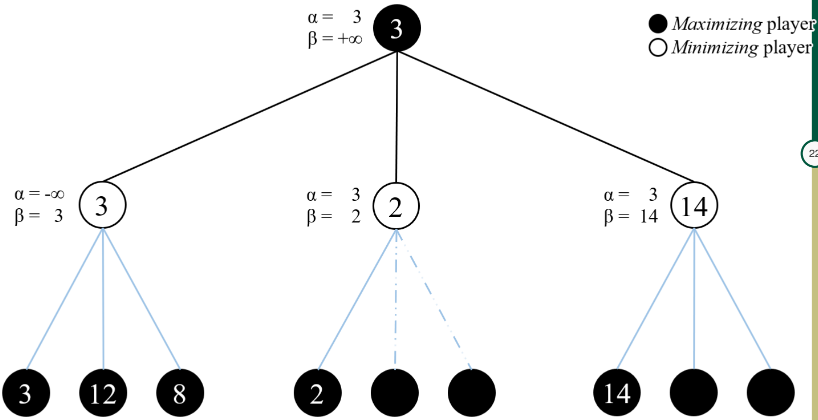

- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.



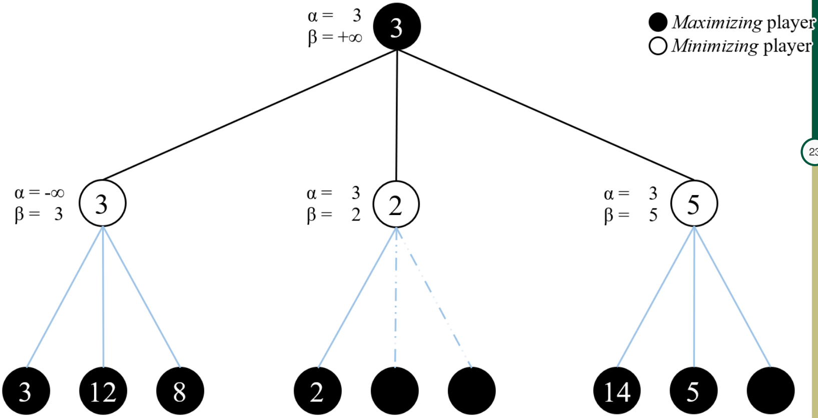

- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.



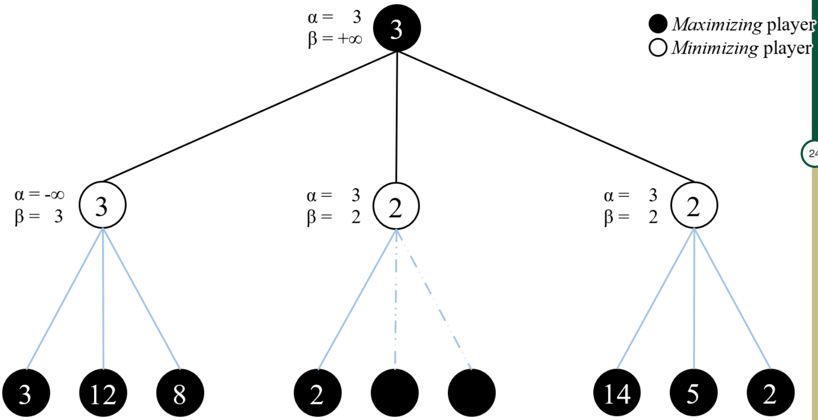

- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.




- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.




- **Example:**
  - Possible moves, for player(Maximizing) at the node(Root), is 3; and the possible responses, from player(Minimizing) wrt. each move, is also 3 for this game.




- **Example:**
  - Possible `moves`, for player(Maximizing) at the node(Root), is 3; and the possible `responses`, from player(Minimizing) wrt. each `move`, is also 3 for this game.

# Adversarial Search and Game Theory

## Alpha–Beta ($\alpha - \beta$) Search Algorithm

```
function ALPHABETA(node, depth, alpha, beta, isMaxPlayer)
    if isLeaveNode(node) || depth == 0 then
        return eval(node)
    end if

    if isMaxPlayer == true then                          ▷ player(Maximizing)
        best = −∞
        for each child in node.children do
            evalRes = ALPHABETA(child, depth − 1, alpha, beta, false)
            best = max(best, evalRes)
            alpha = max(alpha, evalRes)
            if beta ≤ alpha then break-off       ▷ Prune @ parent node(Max.)
            return best
        end for
    else                                                 ▷ player(Minimizing)
        worst = +∞
        for each child in node.children do
            evalRes = ALPHABETA(child, depth − 1, alpha, beta, true)
            worst = min(worst, evalRes)
            beta = min(beta, evalRes)
            if beta ≤ alpha then break-off       ▷ Prune @ parent node(Min.)
            return worst
        end for
    end if
end function
```

- **property(Optimality):**
  - Does yield the solution(Optimal) provided that both players play perfectly; and good *move* ordering improves effectiveness of pruning.

- **property(Run-time Complexity):**
  - $\underline{O(b^{\frac{m}{2}})}$: Only if perfect "Move-Ordering" is attained; where $b$ = Branching factor (average number of children per node), $d$ = Depth of the deepest node (maximum depth of the search tree).

- **property(Space Complexity):**
  - $\underline{O(b \cdot d)}$: where $b$ = Branching factor (average number of children per node) AND $d$ = Depth of the optimal solution.

- **property(Other):**
  - Early `pruning/truncation` does NOT affect result(Final). It is *much faster* by avoiding unnecessary/useless evaluations.

- **search(Monte-Carlo Tree):**
  - It is a algorithm(Decision-Making) & search(Heuristic) algorithm employed in chess, gaming, planning, reinforcement learning, etc.
  - Based on the concept of playing a game severally and with many random starts (i.e. random sampling wrt. game); thereafter, it uses results of these random sampled games to decide how to make a *move* wrt. game.
  - Employs sampling(Random) wrt. focusing on the most promising parts of the search tree.
  - Modern game(GO) have abandoned search(Alpha–Beta), and instead use search(Monte-Carlo Tree).
  - Does NOT require domain-specific heuristics wrt. selection on a given *node/move*, because it executes selection on a *node/move* via a policy(Selection).
  - Its policy/strategy(Selection) naturally balances `exploration` and `exploitation`.

- **Steps wrt. search(Monte-Carlo Tree):**

  - Selection: start from the root and select child nodes based on some strategy/policy (e.g. UCB - Upper Confidence Bound).

    $UCB1 = \text{term}(\text{Exploitation}) + \text{term}(\text{Exploration})$

    ⋆ term(Exploitation): aims at favoring good *moves/nodes*.

    ⋆ term(Exploration): aims at trying less-explored *moves/nodes*.

    $$UCB1 = \frac{w_i}{n_i} + c \cdot \sqrt{\frac{\ln N}{n_i}}$$

    ⋆ $w_i$ = total wins wrt. this *move/node*.

    ⋆ $n_i$ = number of times this *move/node* was visited or tried.

    ⋆ $N$ = overall total number of times node(Root) was visited.

    ⋆ $c = \sqrt{2}$ = constant that balances Exploitation and Exploration.

  - Expansion: expand/grow the tree by adding one or more child nodes (which denote possible *moves/nodes*).

  - Simulation (Playout or Rollout): from the newly added *node/move*, simulate a random game to the end (or for a no. of steps) and record the outcome.

  - Backpropagation: use the result (i.e. wins, losses) of the Simulation (Playout/Rollout) to update the tree(Search) along the path that leads up to the node(Root).

**Algorithm 2** Monte-Carlo Tree-Search algorithm

```
function MCTS(rootNode, playoutCnt)
    for each playout in simulationCnt do
        node = rootNode

        while isLeaveNode(node) == false do            ▷ strategy(Selection)
            node = bestChild(node)
        end while

        if isLeaveNode(node) == true then              ▷ strategy(Expansion)
            node = expand(node)
        end if

        result = simulate(node)                        ▷ strategy(Simulation/Playout)

        backPropagate(result, node)                    ▷ strategy(Backpropagation)
    end for

    return bestMove(rootNode)
end function
```

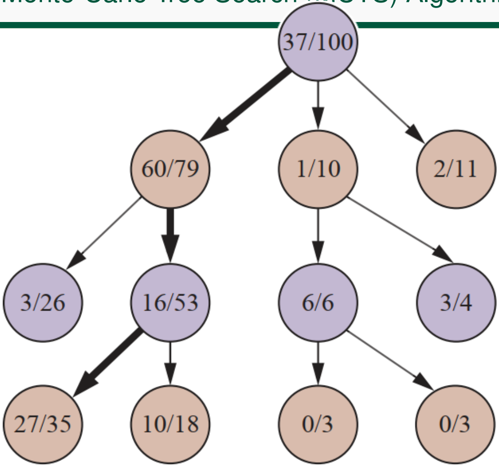



Figure: **Selection** wrt. search(Monte-Carlo Tree)

- **Sample Game wrt. search(Monte-Carlo Tree):**
  - **NB:** a high value(Exploitation), $\frac{w_i}{n_i}$, usually denotes high $UCB1$ value, and which decides our selection(Node/Move).

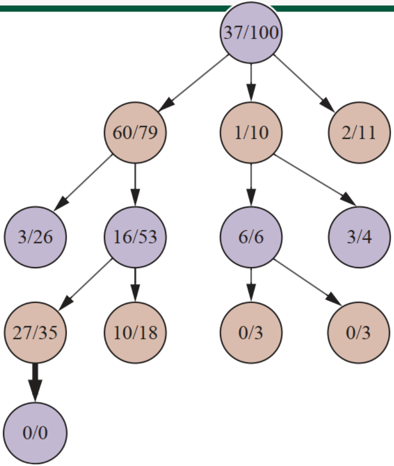



Figure: **Expansion** wrt. search(Monte-Carlo Tree)

- **Sample Game wrt. search(Monte-Carlo Tree):**
  - Generate possible *moves/nodes* by adding 1/more nodes(Child).

# Adversarial Search and Game Theory

## **Simulation/Playout:** Monte-Carlo Tree Search (MCTS) Algorithm

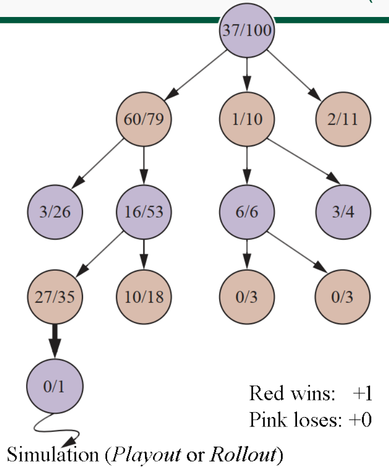



Figure: **Simulation/Rollout** wrt. search(Monte-Carlo Tree)

- **Sample Game wrt. search(Monte-Carlo Tree):**
  - From newly added *node/move*, play a “random game” from *start*-to-*finish* OR play from *start*-to-*given-phase/stage*.

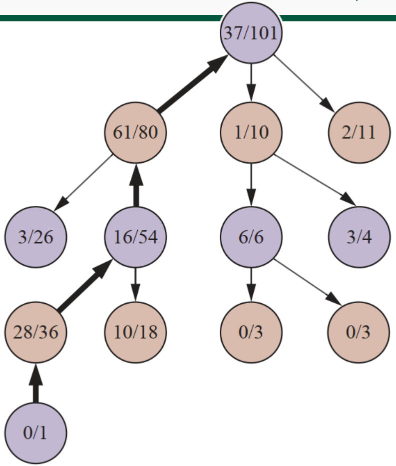



Figure: **Backpropagation** wrt. search(Monte-Carlo Tree)

- **Sample Game wrt. search(Monte-Carlo Tree):**
  - Send backward the result of the played "random game" along the path which leads up to the node(Root).

- **property(Other):**
  - Efficient and Effective wrt. large space(Search).
  - Its policy/strategy(Selection) naturally balances `Exploration` and `Exploitation` wrt. space(Search).
  - May become *slower* if too many simulations/playout/rollouts are required.
  - Some random simulations/playout/rollouts may be inaccurate.
  - Incures more memory(Space) as the tree(Search) grows *larger* as well as *deeper*.

**In-class Activity**

1. **Explain the concept of search(MiniMax)?**

2. **Define in details the concept of search(Alpha-Beta)?**

3. **Explain the concept of search(Monte-Carlo Tree)?**

Questions?
&
Answers!