N Queens using Genetic Algorithm

Step 1: Representation of Chromosomes (Encoding)

- **Chromosome**: A chromosome is a potential solution to the N-Queens problem. Each **chromosome** represents the **positions of queens** on an N×N chessboard.
- **Encoding**:
  - **Permutation encoding** is used: Each gene in the chromosome represents a **row**, and the value at each gene represents the **column** where the queen is placed.
  - Example for N=8:

    Chromosome=[3,7,1,4,6,8,2,5]

    - This means the queens are placed at:
    - Column 1, Row 3
    - Column 2, Row 7
    - Column 3, Row 1
    - Column 4, Row 4
    - Column 5, Row 6
    - Column 6, Row 8
    - Column 7, Row 2
    - Column 8, Row 5.

Step 2: Initial Population

- **Create an initial population** of chromosomes (potential solutions). The population size is typically a parameter to the algorithm (e.g., 50 or 100 chromosomes).
- **Random initialization**: Generate random valid positions of queens (i.e., ensuring that each queen is in a different column and different row) for each chromosome.

Step 3: Fitness Function

- The **fitness function** evaluates how good each chromosome (solution) is. For the N-Queens problem, the goal is to **minimize conflicts** between queens.
- **Fitness Calculation**:
    - The fitness of a chromosome is the **number of non-attacking pairs** of queens. The maximum number of non-attacking pairs for N-queens is:

$$\text{Max Non-Attacking Pairs} = \frac{N \times (N-1)}{2}$$

- The fitness is calculated by subtracting the number of attacking pairs from this maximum value.

- **Attack Calculation**:
  - Queens are attacking each other if they are:
    - On the same **row**: This is prevented in the encoding.
    - On the same **diagonal**: Calculate the diagonals to see if any queens share the same diagonal.
- **Fitness Example**:
  - If N=8, the max non-attacking pairs = 28.
  - If a chromosome has 4 attacking pairs, the fitness is: **Fitness=28−4=24**

Step 4: Selection

- **Select parents** for reproduction (crossover). Use a **selection mechanism** to pick chromosomes with a higher fitness to produce the next generation.
- Common selection methods include:
  - **Roulette Wheel Selection**: The probability of selecting a chromosome is proportional to its fitness.
  - **Tournament Selection**: Randomly pick a few chromosomes, and select the best one out of the group.
- **Example**:
  - Assume we have 5 chromosomes with fitness values 20, 22, 24, 26, and 28. Chromosome with fitness 28 will have the highest probability of being selected for reproduction.

Step 5: Crossover (Recombination)

- Perform **crossover** to generate new chromosomes (offspring) by combining genes from two parent chromosomes.
- **Crossover Method**:
  - **One-point crossover** is common for the N-Queens problem:
    - Randomly select a crossover point in the chromosome, and swap the sections between two parents.
- **Example**:
  - Parent 1: [3,7,1,4,6,8,2,5]
  - Parent 2: [4,1,5,2,7,3,8,6]
  - Random crossover point: 4 (meaning after 4th gene)
  - Offspring:
    - Offspring 1: [3,7,1,4,7,3,8,6]
    - Offspring 2: [4,1,5,2,6,8,2,5]
  - **Note**: This might result in invalid chromosomes (duplicate positions), so after crossover, you may apply a **repair function** to ensure no duplicates.

Step 6: Mutation

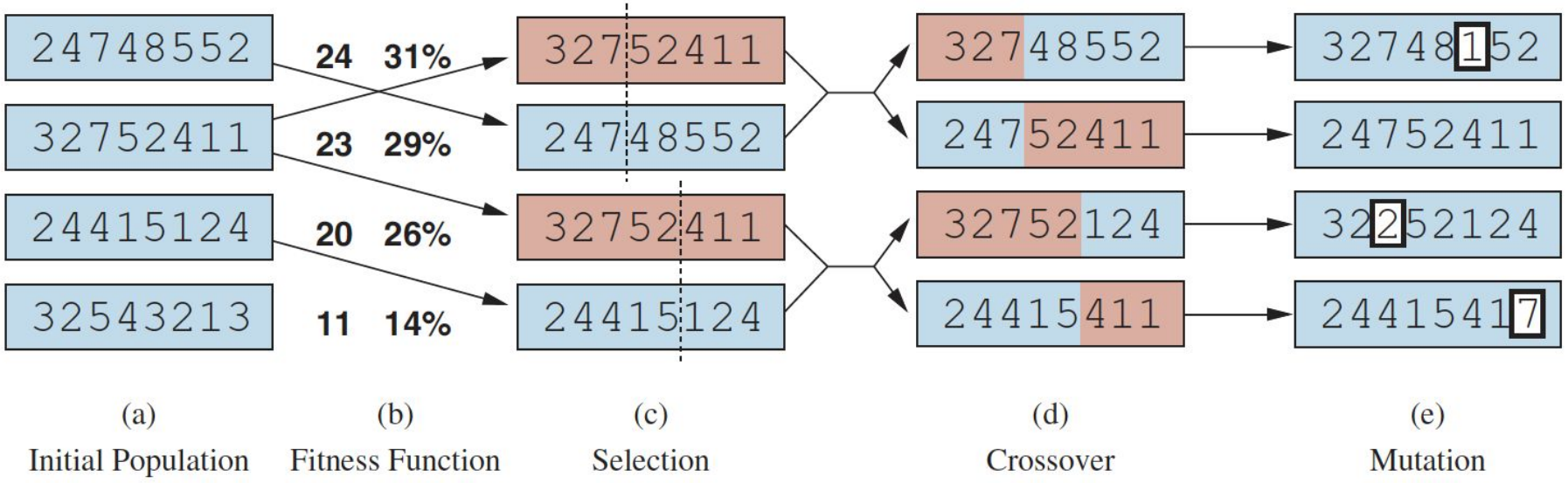
- **Mutation** introduces randomness and helps maintain genetic diversity in the population, preventing the algorithm from getting stuck in local optima.
- **Mutation Method**:
  - For N-Queens, a typical mutation is to **swap two random genes** (swap the positions of two queens).
- **Example**:
  - Chromosome before mutation: [3,7,1,4,6,8,2,5]
  - After mutation: [3,7,1,6,4,8,2,5] (Here, queens in positions 5 and 4 are swapped).
- **Mutation Rate**:
  - Mutation usually happens with a low probability (e.g., 1% or 5%).

Step 7: Replacement

- After generating new offspring through crossover and mutation, **replace** the old population with the new population.
- **Replacement Strategy**:
    - The new population may consist entirely of the offspring, or you can use **elitism**, where the best-performing individuals from the current generation are carried forward to the next generation to preserve their good traits.

Step 8: Termination Condition

- The algorithm continues for several generations, applying selection, crossover, and mutation in each generation.
- **Termination Criteria**:
    - The algorithm stops when one of the following occurs:
        1. A chromosome with a **fitness of 28** (for N=8) is found, meaning a valid solution has been identified (no conflicts).
        2. A **maximum number of generations** is reached (e.g., 1000 generations).
        3. The **population has converged** (no significant improvement over several generations).



A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Adversarial Search

- **Adversarial search** is a type of search used in **competitive environments** where two agents are in direct opposition to one another. One agent, called the **maximizer**, tries to maximize its score, while the other agent, the **minimizer**, tries to either minimize the maximizer's score or maximize its own score, depending on the context.
- **Key Idea**:
  - In adversarial search, both agents are working under the assumption that their opponent is playing optimally. The maximizer makes moves to maximize its score, knowing that the minimizer will try to block or counter those moves.
  - Adversarial search applies to games where players take alternating turns (e.g., chess, checkers, tic-tac-toe).

Applications in Turn-Based Games

Games: Adversarial search is commonly used in **turn-based games** where two players compete against each other by making sequential moves. Some classic examples include:

- **Chess**: The maximizer and minimizer alternate moves, with the goal of checkmating the opponent.
- **Checkers**: Players alternate moves to either capture the opponent's pieces or block their moves.
- **Tic-Tac-Toe**: Each player takes turns placing marks (X or O) on a 3x3 grid, aiming to get three in a row while preventing their opponent from doing the same.

Maximizer and Minimizer:

- The two agents involved in adversarial search have different goals:
  - **Maximizer**: The player who tries to **maximize the score** or gain a strategic advantage. This player wants to make moves that put them in a winning position or increase their chances of winning.
  - **Minimizer**: The player who tries to **minimize the maximizer's score** or improve their own position while limiting the maximizer's gains. In many cases, the minimizer plays defensively, trying to block the maximizer from winning.

Example: In a game like chess, the **maximizer** (e.g., White) aims to checkmate the opponent as quickly as possible, while the **minimizer** (e.g., Black) tries to delay or prevent the checkmate and possibly win themselves. Each move by the maximizer (White) is countered by the minimizer (Black), and vice versa.

Minimax Algorithm

- **Minimax** is a decision-making algorithm used in **two-player zero-sum games**, where one player's gain is another player's loss. The algorithm assumes that both players are playing optimally, meaning each player tries to maximize their own score while minimizing the opponent's score. It is commonly applied to adversarial search problems, such as chess, checkers, or tic-tac-toe.
- **Zero-Sum Game**:
  - A **zero-sum game** is a scenario in which one player's gain is exactly balanced by the other player's loss. In these games, the goal for each player is to either maximize their own payoff (if they are the maximizer) or minimize their opponent's payoff (if they are the minimizer).
- The **purpose** of the minimax algorithm is to **find the optimal move** for the current player by simulating all possible future moves and counter-moves of the opponent. It systematically evaluates all possible game outcomes, assuming that both players are playing to their full potential.
- **Maximizer's Goal**: Maximize the score (or utility).
- **Minimizer's Goal**: Minimize the maximizer's score, thereby reducing their own potential loss.

Step 1: Generating the Game Tree

- **Game Tree**: The first step is to generate the **entire game tree** from the current state to the terminal states. Each node in the tree represents a game state, and each edge represents a move by one of the players.
    - The root node is the current state of the game.
    - The child nodes represent the possible future states after each player's move.
- **Depth of the Tree**: The game tree is constructed until it reaches the **terminal nodes**, where the game has ended (e.g., one player wins, or the game results in a draw).

Example:

- In tic-tac-toe, the root node might represent the current board state (e.g., Player X's turn), and each branch represents a possible move Player X could make (e.g., placing an X in an empty square). The game tree expands until every possible sequence of moves has been considered.

Step 2: Assign Values to Terminal States

- **Leaf Nodes**: Once the game tree has been fully generated, the algorithm assigns values to the **terminal nodes** (leaf nodes) based on the outcome of the game. These values represent the utility for the maximizer:
  - **+1** for a win for the maximizer.
  - **-1** for a win for the minimizer (or a loss for the maximizer).
  - **0** for a draw.
- The terminal values are crucial because they allow the algorithm to evaluate whether a particular sequence of moves leads to a favorable or unfavorable outcome for the maximizer.

Example:

- In tic-tac-toe, if a leaf node represents Player X winning the game, that node will be assigned a value of **+1**. If Player O wins, the node will be assigned a value of **-1**. A draw would result in a value of **0**.

Step 3: Bottom-Up Evaluation of the Tree

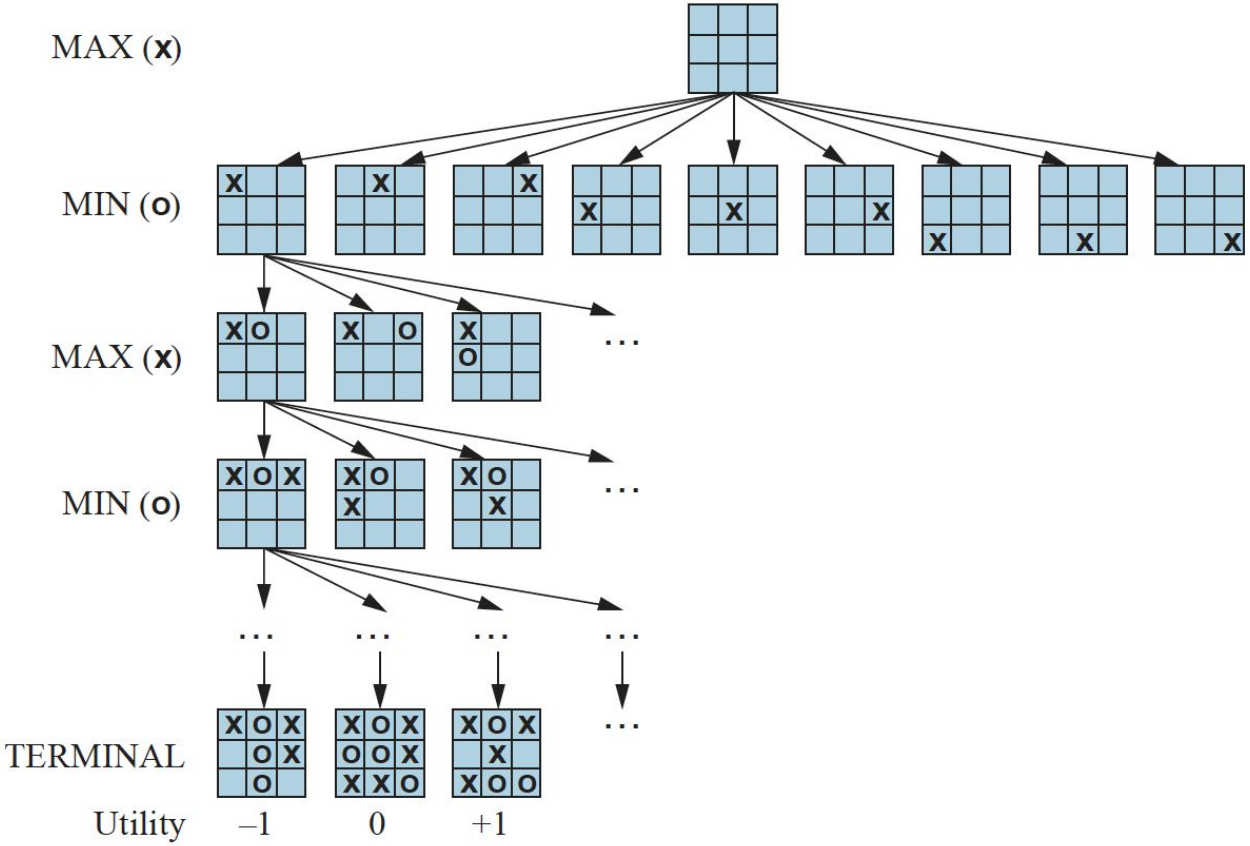
- Once the terminal nodes have been evaluated, the minimax algorithm performs a **bottom-up evaluation** of the tree. This means that it starts at the terminal nodes and works its way back up to the root node, deciding the best possible move for each player.
- **Minimizer's Turn**:
  - If it is the **minimizer's turn** (Player O in tic-tac-toe), the algorithm chooses the **minimum** value of the child nodes. The minimizer is trying to minimize the maximizer's score, so it will pick the move that results in the least favorable outcome for the maximizer.
- **Example**:
  - If Player O can either lose (-1) or draw (0) in a set of moves, the minimizer will choose the draw (0), because it minimizes Player X's gain.
- **Maximizer's Turn**:
  - If it is the **maximizer's turn** (Player X in tic-tac-toe), the algorithm selects the **maximum** value of the child nodes. The maximizer wants to maximize their own score, so they will pick the move that results in the most favorable outcome.
- **Example**:
  - If Player X has a choice between a win (+1) and a draw (0), the maximizer will choose the win (+1), as it provides the best outcome.

Step 4: Choosing the Best Move

- After performing the bottom-up evaluation of the game tree, the minimax algorithm selects the move that leads to the **best possible outcome** for the current player. This is based on the values calculated in Step 3.
- For the **maximizer**, this means selecting the move with the **highest value**. For the **minimizer**, it means selecting the move with the **lowest value**.

Key Points:

- The algorithm considers all possible future moves by both players.
- Each player is assumed to play optimally, so the maximizer chooses moves that give the highest value, while the minimizer chooses moves that give the lowest value.



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.

Alpha-Beta Pruning

Alpha-beta pruning is a technique used to improve the efficiency of the minimax algorithm by cutting off (or "pruning") branches in the game tree that **cannot affect** the final decision.

- It allows the minimax algorithm to ignore parts of the tree that are irrelevant to finding the optimal move, thereby reducing the number of nodes evaluated.

The central idea behind alpha-beta pruning is that, during the search process, the algorithm keeps track of two values:

1. **Alpha (α)**:
   - Alpha represents the **best value** that the **maximizer** can guarantee so far.
   - The maximizer will update alpha as it finds better values. Alpha acts as a threshold that the minimizer must respect.
2. **Beta (β)**:
   - Beta represents the **best value** that the **minimizer** can guarantee so far.
   - The minimizer will update beta as it finds better values. Beta acts as a threshold that the maximizer must respect.

Pruning Condition:

- **Key Principle**: If, during the search, a move (branch) is found to **guarantee a worse outcome** than one already explored, it is **pruned** (ignored).
  - Specifically, if at any node, **alpha ≥ beta**, further exploration of that node is unnecessary because the minimizer will not allow the maximizer to reach that state, or vice versa.

How Alpha-Beta Pruning Works

Initialization:

- The algorithm starts at the root node (the current game state) and initiates the search.
- Initially, alpha is set to **negative infinity** (`−∞`), and beta is set to **positive infinity** (`+∞`), indicating that there are no constraints on the maximizer or minimizer.

Maximizer's Turn (Alpha Update):

- The maximizer explores each child node and tries to find the move that provides the highest score.
- As the maximizer finds better values, it updates **alpha** with the highest value seen so far.
- If a value greater than or equal to beta is found (`alpha ≥ beta`), the remaining child nodes are **pruned**, since the minimizer will not allow the game to proceed along this branch.

Minimizer's Turn (Beta Update):

- The minimizer explores each child node, seeking the move that provides the lowest score for the maximizer.
- As the minimizer finds better values, it updates **beta** with the lowest value seen so far.
- If a value less than or equal to alpha is found (`beta ≤ alpha`), the remaining child nodes are **pruned**, as the maximizer would not choose to continue down this path.

Example of Alpha-Beta Pruning in Tic-Tac-Toe

Let's consider a simplified example of alpha-beta pruning in the game of **tic-tac-toe**, where Player X is the maximizer, and Player O is the minimizer.

- **Step 1**:
  - Player X (maximizer) makes the first move and examines each of its possible moves.
  - Alpha is initialized to $-\infty$ (no known maximum yet), and beta is initialized to $+\infty$ (no known minimum yet).
- **Step 2**:
  - Player O (minimizer) responds by examining possible counter-moves after Player X's chosen move.
  - As Player O explores, it updates beta with the lowest value seen so far, trying to minimize Player X's score.
- **Step 3**:
  - As Player O explores a move, it finds that no better score than beta can be achieved for Player X (e.g., it leads to a loss for Player X).
  - If a branch results in a score worse than an already-known outcome (i.e., `alpha ≥ beta`), further exploration of that branch is stopped, and the algorithm **prunes** the rest of the branches.
- **Step 4**:
  - Alpha and beta are continuously updated, and branches are pruned when the conditions `alpha ≥ beta` are met.

Result:

- The algorithm prunes away parts of the game tree that do not influence the final decision, significantly speeding up the evaluation process by avoiding unnecessary calculations.

Advantages of Alpha-Beta Pruning

Efficiency:

- By pruning branches of the game tree, alpha-beta pruning reduces the number of nodes that need to be explored, making the search **more efficient**. In the best-case scenario, it can reduce the time complexity of the minimax algorithm from $O(b^d)$ to $O(b^{d/2})$, where b is the branching factor, and d is the depth of the tree.

No Loss of Optimality:

- Alpha-beta pruning **does not affect** the final result of the minimax algorithm. It finds the same optimal move as minimax, but faster, by ignoring irrelevant branches.

Scalability:

- Alpha-beta pruning allows adversarial search to scale to more complex games (like chess), where evaluating the entire game tree is computationally prohibitive.

Minimax Algorithm and Alpha-Beta Pruning: Example in Chess

Let's explore how the **minimax algorithm** and **alpha-beta pruning** work in a more familiar **game-related scenario—chess**. Chess is a two-player, zero-sum game where each player's goal is to **maximize** their advantage (or minimize their disadvantage) while the opponent tries to do the same. We'll focus on a simplified scenario where only a few moves are considered for each player.

Scenario Setup

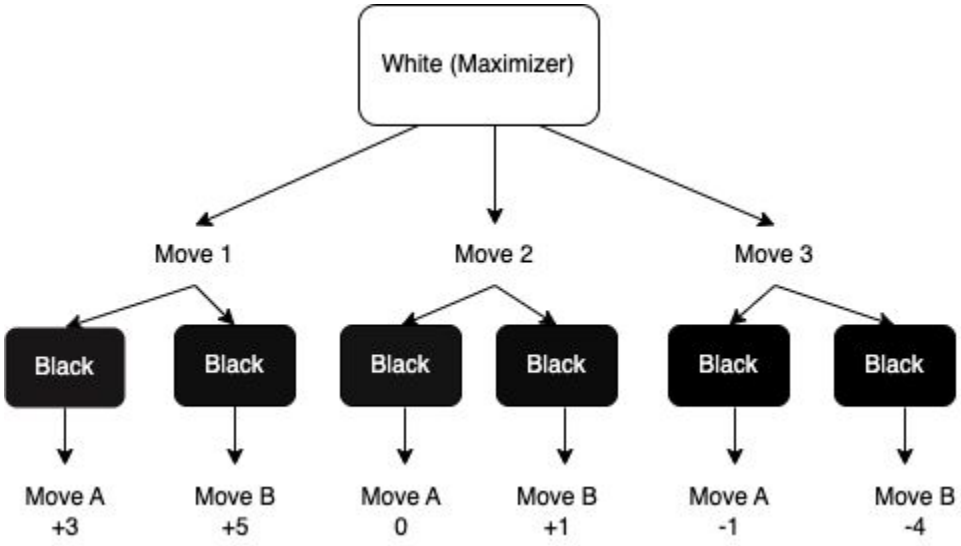
- **White** is the **maximizer** (trying to checkmate Black).
- **Black** is the **minimizer** (trying to avoid getting checkmated and win if possible).
- We'll explore three possible moves for White, followed by three possible responses for Black, and we will use the **minimax algorithm** to determine White's best move. The goal is to evaluate these possible moves and identify the optimal choice.

Game Tree Setup

Imagine White has three potential moves to choose from:

1. **Move 1** (leading to potential outcomes evaluated by Black).
2. **Move 2** (leading to potential outcomes evaluated by Black).
3. **Move 3** (leading to potential outcomes evaluated by Black).

For each move White makes, Black has two possible counter-moves that either reduce White's advantage or increase Black's advantage.



Step 1: Generate the Game Tree:

- The game tree shows White's possible moves (Move 1, Move 2, and Move 3), and for each of White's moves, Black has two possible responses (Move A and Move B).
- Each terminal state is given an evaluation score representing White's advantage. For example, if Black responds to White's **Move 1** with **Move A**, the result might give White an advantage of **+3**. If Black responds with **Move B**, White gets a better advantage of **+5**.

Step 2: Assign Values to Terminal States (Leaf Nodes):

- The terminal nodes (or leaf nodes) are evaluated based on the outcome for White:
  - **Move 1** → **Move A** results in a score of **+3** for White.
  - **Move 1** → **Move B** results in a score of **+5** for White.
  - **Move 2** → **Move A** results in a score of **0** (neutral).
  - **Move 2** → **Move B** results in a score of **+1** for White.
  - **Move 3** → **Move A** results in a score of **-1** (bad for White).
  - **Move 3** → **Move B** results in a score of **-4** (even worse for White).

Step 3: Bottom-Up Evaluation (Minimax):

- **Black's Turn (Minimizer)**: Black, being the minimizer, will choose the **minimum value** among the possible outcomes, aiming to minimize White's advantage.
  - For **Move 1**, Black compares **+3** and **+5** and will choose **Move A** (giving White **+3**), because it is worse for White.
  - For **Move 2**, Black compares **0** and **+1**, and will choose **Move A** (resulting in **0**), which neutralizes White's advantage.
  - For **Move 3**, Black compares **-1** and **-4** and will choose **Move A** (giving White **-1**), as it limits White's advantage more than Move B.
- **White's Turn (Maximizer)**: White, as the maximizer, will select the **maximum value** from the outcomes Black has left:
  - From **Move 1**, White can get **+3**.
  - From **Move 2**, White can get **0**.
  - From **Move 3**, White can get **-1**.
- Therefore, White will choose **Move 1** since it provides the best outcome of **+3**.

Now let's optimize the decision-making process using **alpha-beta pruning**.

- **Alpha (α)** is initialized to **-∞** (the worst value for White, the maximizer).
- **Beta (β)** is initialized to **+∞** (the worst value for Black, the minimizer).

1. **Maximizer's Turn (White)**:
    - White first considers **Move 1**.
2. **Minimizer's Turn (Black after Move 1)**:
    - Black looks at **Move A** and sees a score of **+3** for White. Since **+3** is better than alpha (**α = -∞**), alpha is updated to **+3**.
    - Black then evaluates **Move B**, which gives **+5** for White. Since **+5** is better for White than **+3**, alpha is updated to **+5**.
    - At this point, White knows the outcome of **Move 1** is **+5** (before Black's response would be **Move A**). So, alpha is set at **+5** for now.
3. **Maximizer's Turn (White evaluates Move 2)**:
    - White now evaluates **Move 2**, with an alpha of **+5** carried forward.
    - Black evaluates **Move A** for Move 2, which results in **0**. Since **0** is less than **alpha = +5**, beta is updated to **0**.
    - Black also evaluates **Move B**, which results in **+1** for White. However, since **0 ≤ alpha = +5**, there's no need to continue evaluating this branch. **Move 2 is pruned**.
4. **Maximizer's Turn (White evaluates Move 3)**:
    - Now White evaluates **Move 3**. The current alpha value is **+5**, and beta is **+∞**.
    - Black looks at **Move A**, which gives **-1** for White. Since **-1** is worse than alpha (**α = +5**), beta is updated to **-1**.
    - Black evaluates **Move B**, which results in **-4**, but since **-1 ≤ alpha = +5**, this branch is **pruned** as well.

Final Decision:

- After applying **alpha-beta pruning**, White will choose **Move 1** because it provides the best possible outcome, **+3**, without needing to evaluate every branch in the tree.
- The unnecessary branches (those where the values couldn’t improve the decision) were **pruned**, making the algorithm more efficient.

Real-Life Example of Minimax Algorithm and Alpha-Beta Pruning: Investment Strategy

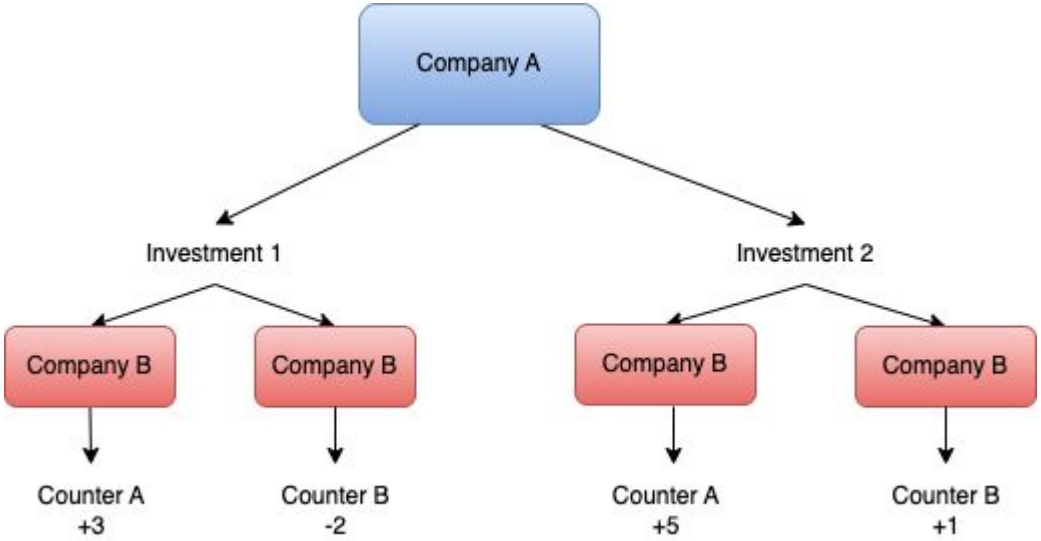
Let's consider a real-life scenario where two companies (**Company A** and **Company B**) are competing in the market by making investment decisions. The goal for each company is to maximize its profit while minimizing the profit of the competitor. This can be modeled as a zero-sum game, where the gain of one company results in a loss for the other. We'll use the **minimax algorithm** and **alpha-beta pruning** to help **Company A** (the maximizer) make the best investment decision, assuming **Company B** (the minimizer) is playing optimally.

Scenario Setup:

- **Company A (Maximizer)** is choosing between two potential investments: **Investment 1** and **Investment 2**.
- After **Company A** makes a choice, **Company B (Minimizer)** responds by choosing between **Counter Investment A** and **Counter Investment B**.
- Each combination of investments has different potential outcomes for Company A, depending on how Company B responds.

Game Tree for the Investment Decision:

The decision-making process can be represented as a game tree where each node represents a decision point, and the leaves represent the outcomes (profits/losses). We will assign values to these outcomes to represent the profit/loss for **Company A**.



1. **Step 1: Generate the Game Tree**:
    - The root node represents **Company A's decision** between **Investment 1** and **Investment 2**.
    - Each branch represents **Company B's counter decision** between **Counter Investment A** and **Counter Investment B**.
2. **Step 2: Assign Values to Terminal States (Leaf Nodes)**:
    - Each leaf node represents the outcome for **Company A** based on the combined investment decisions of both companies:
        - **Investment 1 → Counter Investment A =** +3 (good for Company A).
        - **Investment 1 → Counter Investment B =** -2 (bad for Company A).
        - **Investment 2 → Counter Investment A =** +5 (very good for Company A).
        - **Investment 2 → Counter Investment B =** +1 (neutral outcome for Company A).
3. **Step 3: Bottom-Up Evaluation**:
    - **Company B's turn (Minimizer)**: Company B will try to minimize **Company A's** profit.
        - For **Investment 1**: Company B chooses the minimum between +3 and -2, so **Company B** will select **Counter Investment B**, leading to an outcome of **-2** for **Company A**.
        - For **Investment 2**: Company B chooses the minimum between +5 and +1, so **Company B** will select **Counter Investment B**, leading to an outcome of **+1** for **Company A**.
4. **Step 4: Choose the Best Move for the Maximizer (Company A)**:
    - Now it's **Company A's turn** (Maximizer). Company A will compare the values from each branch and choose the option that **maximizes its profit**.
        - **Investment 1** leads to a profit of **-2**.
        - **Investment 2** leads to a profit of **+1**.
    - **Company A** will choose **Investment 2**, as it provides the maximum profit of **+1**.

Now, let's apply **alpha-beta pruning** to the same scenario to **improve efficiency** by eliminating branches that do not affect the final decision.

1. **Initialization**:
   - **Alpha (α)** is initialized to **-∞** (the worst possible outcome for the maximizer).
   - **Beta (β)** is initialized to **+∞** (the worst possible outcome for the minimizer).
2. **Maximizer's Turn (Company A)**:
   - **Company A** evaluates **Investment 1** first.
3. **Minimizer's Turn (Company B)** (after **Investment 1**):
   - **Company B** explores **Counter Investment A** first, with an outcome of **+3** for **Company A**. Since this is greater than alpha (**α = +3**), alpha is updated to **+3**.
   - **Company B** then explores **Counter Investment B** with an outcome of **-2**. Since this is lower than alpha, beta is updated to **-2**. The value for **Investment 1** is now **-2**.
4. **Maximizer's Turn (Company A)**:
   - **Company A** now evaluates **Investment 2**. The current alpha value is **-2**.
5. **Minimizer's Turn (Company B)** (after **Investment 2**):
   - **Company B** explores **Counter Investment A** with an outcome of **+5**. This is greater than alpha, so alpha is updated to **+5**.
   - **Company B** now explores **Counter Investment B** with an outcome of **+1**. Since **alpha > beta** (because **α = +5** and **β = -2** from the previous node), we **prune** this branch. There's no need to continue evaluating this node, as the minimizer would not allow this outcome.
6. **Conclusion**:
   - The final value for **Investment 2** is **+1**. Since **+1** is greater than **-2**, **Company A** will choose **Investment 2**.