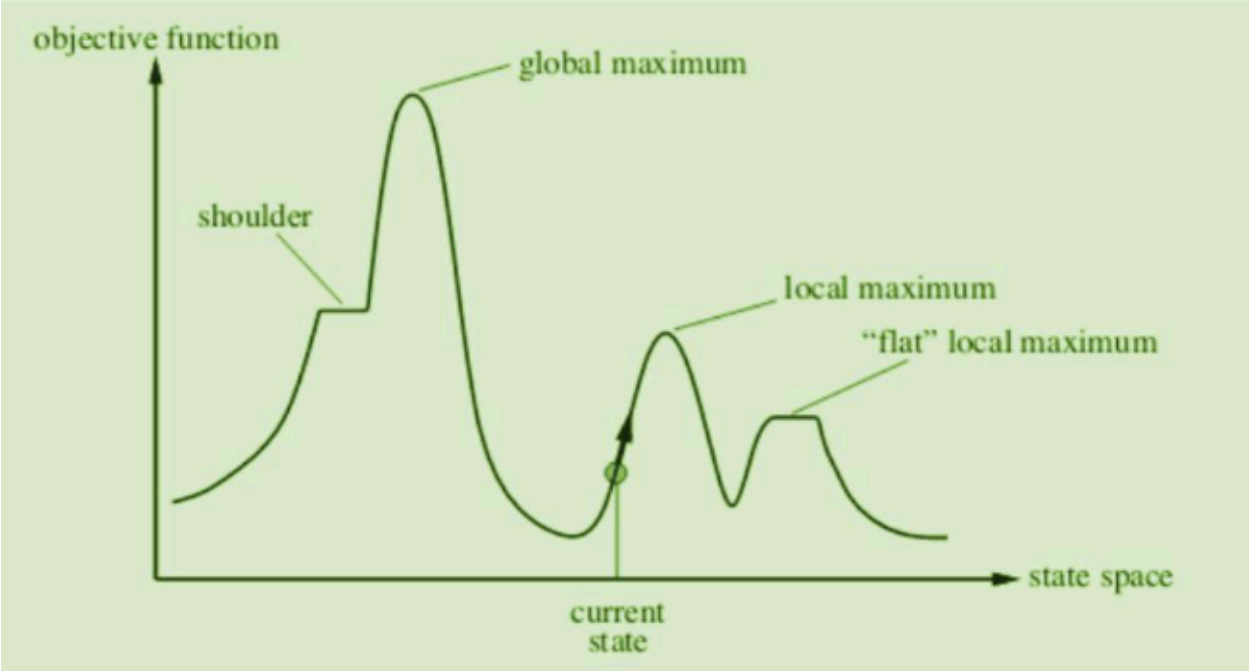
Hill Climbing Searching Algorithm



1-Overview

Hill climbing is an optimization algorithm used to improve a solution iteratively by making local changes. This project will focus on "Simple Hill Climbing". Which evaluates neighbors one at a time and moves to the first one that offers an improvement. We will also describe the other type briefly.

2-Brief Overview of Other Types

Steepest-Ascent Hill Climbing:

- **Evaluates all neighbors and moves to the one offering the steepest improvement.**
- **Thorough but computationally expensive.**

Stochastic Hill Climbing:

- **Randomly selects a neighbor and moves to it if it offers an improvement.**
- **Efficient for large solution spaces but less predictable.**

3-Simple Hill Climbing

Algorithm Steps:

1. Initialization: Start with an initial solution or state.

2. Evaluation: Compute the evaluation function for the current state.

3. Neighbor Selection:

- **Select a single neighbor of the current state.**
- **Evaluate the neighbor using the evaluation function.**

4. Move or Terminate:

- **Move to the neighbor if it offers an improvement over the current state.**
- **If no improvement is possible, terminate the algorithm, since there is no backtracking in the simple hill climbing algorithm.**

Advantages:

- **Easy to implement.**
- **Faster as it evaluates one neighbor at a time.**

Disadvantages:

- **May stop early at a local optimum.**
- **Limited exploration as it does not evaluate all neighbors.**

4-Applications of Simple Hill Climbing:

1. Pathfinding: hill climbing is used in AI systems that need to navigate or find the shortest path between points, such as in robotics or game development.

2. Optimization: hill climbing can be used for solving optimization problems where the goal is to maximize or minimize a particular objective function, such as scheduling or resource allocation problems.

3. Game AI: puzzle solving (ex: 8-puzzle, Sudoku, etc.).

4. Machine Learning: Simple Hill Climbing can be used for hyperparameter tuning by iteratively evaluating and improving a machine learning model's performance. The algorithm modifies one hyperparameter at a time and moves to the new configuration only if it improves the model's performance.

5-Code Explanation

General Idea:

- Hill Climbing is an iterative algorithm designed for solving optimization problems by searching incrementally for a better solution.

How It Works:

1. Starting Point:

- **Begins with an initial guess or solution for the problem.**

2. Neighbor Evaluation:

- **Evaluates neighboring solutions and moves to the one that appears most promising.**

3. Local Optimum:

- **Stops when no neighbors offer a better solution, potentially causing the algorithm to get stuck at local maxima/minima.**

Key Code Snippet:

```
from setup import*

def hill_climb(root:Node, target):

  path = [root.value]
  current = root
  next_node = root

  while next_node is not None and path[-1] != target:

    next_node = None

    for child in current.children:
      print(current.cost)
      if child.cost <= current.cost:
        next_node = child
        break

    if next_node is not None:
      current = next_node

    path.append(current.value)
  return path


print(test_algorithm((0,0), (9,9), 10, 10, 30, hill_climb, (9,9)))
```

```
def hill_climb(root:Node, target):
```

Parameters:

- **root: Node: The starting point of the search, representing the current state or position within a graph.**
- **target: The goal or endpoint the algorithm aims to reach.**

```
  path = [root.value]
  current = root
  next_node = root
```

Initialization:

- **path: This list keeps track of the nodes visited during the search. It starts with the value of the root node.**
- **current: A variable to keep track of the current node being evaluated.**
- **next_node: Initially set to the root, it will point to the next node to move to during the search.**

```
  while next_node is not None and path[-1] != target:
    next_node = None
```

- **Loop Condition: The loop runs as long as there is a next_node and the last node in the path isn't the target. This ensures the algorithm continues to search until the target is found or no viable next move exists.**
- **Reset next_node: Before evaluating children, next_node is reset to None to check if a better step is found in this iteration.**

```
    for child in current.children:
      if child.cost <= current.cost:
        next_node = child
        break
```

Searching Neighbors:

- **Iterates over current.children to evaluate neighboring nodes (children).**
- **Move Choice: Finds the first child whose cost is less than or equal to current.cost and selects it as next_node. This reflects a greedy choice, aiming for an immediate improvement or stability.**

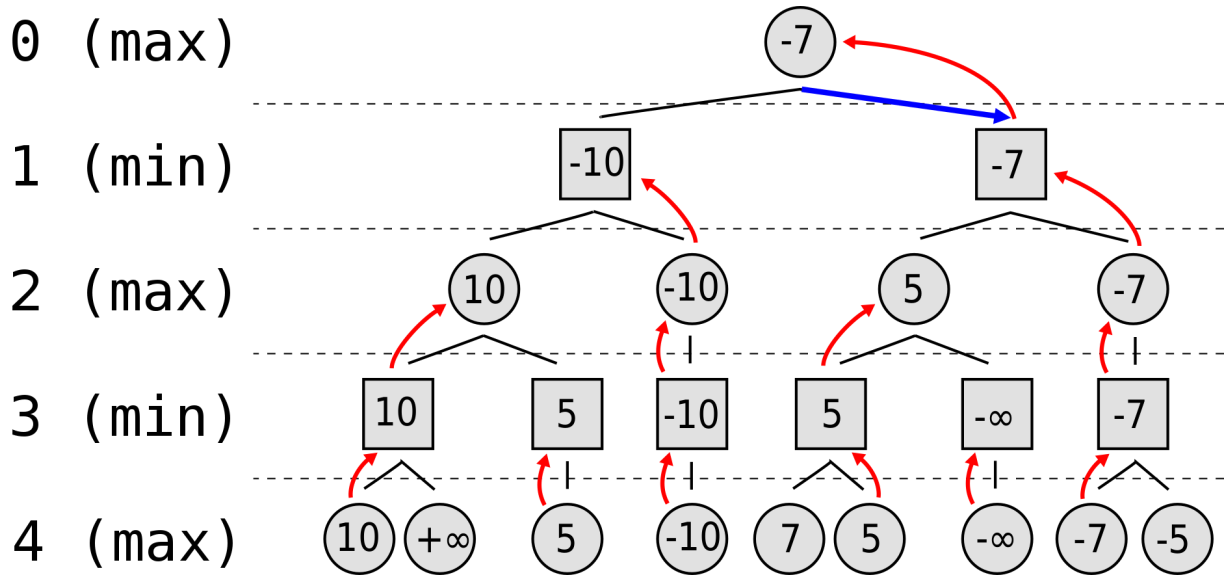
```
    if next_node is not None:
      current = next_node
    path.append(current.value)
  return path
```

- **Update Current Node: If a valid next node is found (next_node is not None), update current to this node, moving one step closer towards the goal.**
- **Path Update: Append the current node's value to path, maintaining a record of the path taken by the algorithm.**
- **Return Statement: The function returns the path, detailing the sequence of nodes visited from the root to the stopping point (either the target or where no further improvement is possible).**

6-Resources

1. **https://www.javatpoint.com/hill-climbing-algorithm-in-ai**
2. **https://iq.opengenus.org/hill-climbing-algorithm/**

Minimax Algorithm



1-Overview

The “Minimax Algorithm” is a decision-making algorithm used in game theory and artificial intelligence for two-player, zero-sum games like chess, tic-tac-toe, and checkers. It works by minimizing the possible loss for a worst-case scenario (minimizing the opponent's maximum payoff). The algorithm simulates all possible moves, evaluates game states using a utility function, and chooses the move that maximizes the player's advantage while minimizing the opponent's advantage.

2-Types of Minimax Algorithms

2.1 Basic Minimax

The traditional implementation assumes both players play optimally and alternates between minimizing the opponent's advantage and maximizing the current player's score.

Algorithm Steps:

1. **Tree Generation:** **Generate a game tree of all possible moves and resulting states.**
2. **Evaluation:** **Use a utility function to assign scores to terminal states (e.g., +1 for a win, -1 for a loss, and 0 for a draw).**
3. **Backtracking:**

    - **At the leaf nodes, return their scores.**
    - **For "min" levels (opponent's turn), choose the minimum score from the child nodes.**
    - **For "max" levels (current player's turn), choose the maximum score from the child nodes.**
4. **Optimal Move: Choose the best outcome based on the Minimax calculation.**

Advantages:

- **Ensures optimal play for both players.**
- **Guarantees a solution if the game tree is finite.**

Disadvantages:

- **Computationally expensive for games with large state spaces.**

2.2 Minimax with Alpha-Beta Pruning

This enhancement skips evaluating branches of the game tree that cannot influence the final decision, reducing the number of nodes explored.

Key Points:

- **Alpha (max player's best value): Tracks the best score the maximizing player can guarantee.**
- **Beta (min player's best value): Tracks the best score the minimizing player can guarantee.**

Advantages:

- **More efficient than basic Minimax.**
- **Handles larger game trees by reducing redundant evaluations.**

2.3 Iterative Deepening Minimax

This version of the Minimax algorithm explores the game tree in multiple passes, starting with a shallow depth and gradually increasing the depth.

How does it work?

1. **Start Small: The algorithm first calculates the best move by looking only one step ahead.**
2. **Go Deeper: It then looks two steps ahead, then three steps, and so on, recalculating the best move each time.**
3. **Time Management: This process continues until a time limit is reached or the entire game tree is explored.**

Advantages:

- **Offers a solution within a fixed time limit.**
- **Allows finding better moves as time permits.**

3-Applications of Minimax Algorithm

1. AI in Board Games:

- **Chess, tic-tac-toe, and checkers.**
- **Decision-making in turn-based games.**

2. Game Theory:

- **Analyzing strategic interactions in competitive scenarios**

3. Artificial Intelligence:

- **Developing intelligent agents for adversarial environments.**

4. Optimization:

- **Can be adapted for scenarios where players have opposing objectives.**

4-Code Explanation

General Idea:

- The Minimax algorithm is a recursive strategy used for decision-making in two-player games. It helps determine the optimal move by assuming that your opponent also plays optimally.

- The goal is to maximize the player's minimum payoff (from which the name derives: "minimize the maximum loss").

How It Works:

1. Game Tree Exploration:

- The algorithm creates a decision tree model where each node represents a game state and edges represent possible moves.

- Two types of nodes are considered: maximizing nodes (the player's turn) and minimizing nodes (opponent's turn).

2. Recursive Evaluation:

- Each recursive call considers all possible moves for the current player.

- At terminal nodes (game end or cutoff depth), a heuristic evaluation function determines the node's score.

- The maximizer chooses the move with the highest value, while the minimizer chooses the lowest.

Key Code Snippet:

```
from setup import*
def minmax(state, depth, maximizing):
    if state.is_leaf():
        return state.value

    if maximizing:
        return max(minmax(child, depth + 1, False) for child in state.children)
    else:
        return min(minmax(child, depth + 1, True) for child in state.children)
```

```
from setup import*
```

Import Statement: This line imports necessary module

```
def minmax(state, depth, maximizing):
```

This function takes three parameters which are:

- **state: represents the current state of the game or decision tree node.**
- **depth: Indicates the current depth in the game tree.**
- **maximizing: A boolean flag indicating whether the current player is the maximizer (True) or the minimizer (False).**

```
    if state.is_leaf():
        return state.value
```

Leaf Node Check: This checks if the current state is a terminal node (i.e., a game state where the game ends). If it is, the function returns the state.value, which represents the utility or score of that state. This is the base case for the recursion.

```
    if maximizing:
        return max(minmax(child, depth + 1, False) for child in
state.children)
```

Maximizing Player: If the current player is the maximizer, the function evaluates all possible moves (children of the current state) and recursively calls minmax for each child with maximizing set to False. It returns the maximum value obtained from these recursive calls, representing the best possible outcome for the maximizer.

```
    else:
        return min(minmax(child, depth + 1, True) for child in
state.children)
```

Minimizing Player: if the current player is the minimizer, the function evaluates all possible moves and recursively calls minmax for each child with maximizing set to True. It returns the minimum value obtained, representing the best possible outcome for the minimizer.

Key Points

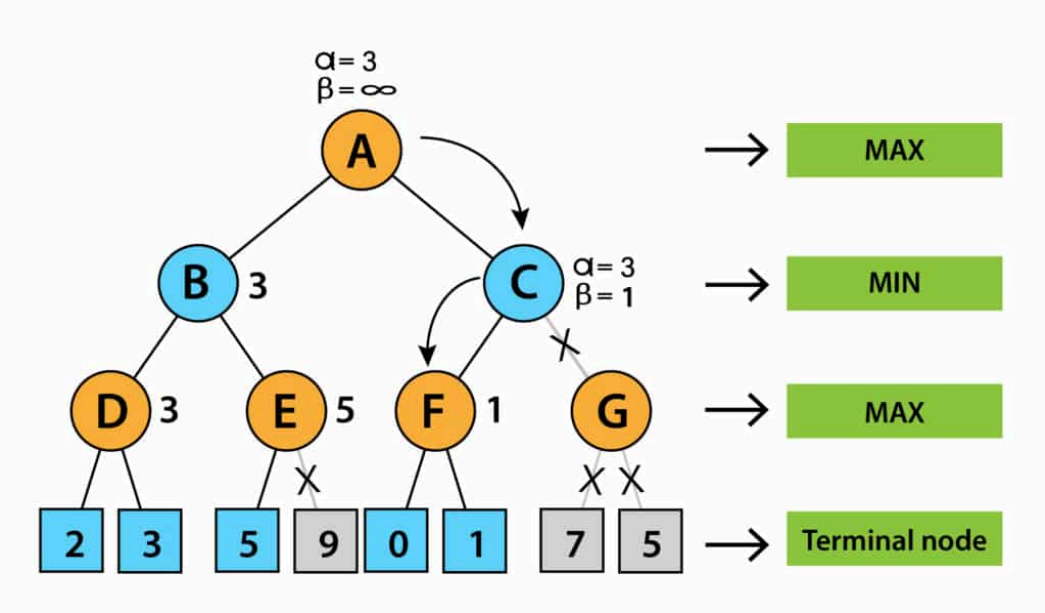
- **Recursive Nature: The function uses recursion to explore the entire game tree, alternating between maximizing and minimizing at each level.**
- **Base Case: The recursion terminates when a leaf node is reached, returning the node's value.**
- **Optimal Strategy: By evaluating all possible moves and their outcomes, the algorithm determines the optimal strategy for both players, assuming perfect play.**

5-Resources

1. https://www.javatpoint.com/mini-max-algorithm-in-ai
2. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

3. https://iq.opengenus.org/minimax-algorithm/

Alpha-Beta Pruning



1-Overview

Alpha-Beta Pruning is an optimization technique for the Minimax algorithm used in decision-making and game-playing scenarios, such as chess or tic-tac-toe. By systematically pruning branches of the search tree that cannot influence the final decision, it significantly reduces the number of nodes evaluated, improving computational efficiency. It achieves the same outcome as the standard Minimax algorithm while exploring fewer states.

2-Key Concepts

1. Alpha and Beta:

- **Alpha (α): The best value achievable by the maximizing player so far.**
- **Beta (β): The best value achievable by the minimizing player so far.**
- **The pruning occurs when α≥β, as further exploration cannot yield better outcomes.**

2. Tree Traversal:

- **The algorithm alternates between maximizing and minimizing layers of the game tree.**
- **Pruning saves computational resources by skipping nodes that cannot affect the final decision.**

3. Efficiency:

- **Alpha-Beta Pruning reduces the effective branching factor of a game tree. For example, in a chess-like scenario with 36 branches per node and a 4-ply depth, pruning can reduce the number of terminal nodes evaluated from over 1 million to just about 2,000.**

3-Advantages

- **Reduced Complexity: Cuts the search space significantly, often by half or more compared to Minimax.**
- **Scalability: Enables deeper searches in game trees with fixed computational resources**
- **Optimal Results: Produces the same results as Minimax without unnecessary calculations**

4-Disadvantages

- **Dependent on Move Ordering.**
- **Exponential Growth in Large Search Trees.**

5-Applications

- **Chess Engines: Alpha-Beta Pruning is a core part of modern chess programs like Stockfish, enabling evaluation of millions of possible moves within seconds.**
- **Strategic Games: Applied in decision-making processes for games requiring strategic planning, including board games and resource management scenarios.**
- **AI Systems: Used in real-time decision-making for adversarial AI agents**

6-Code Explanation

General Idea:

- An optimization for Minimax, Alpha-Beta Pruning reduces the number of nodes evaluated, hence speeding up the decision-making process.

How It Works:

1. Alpha and Beta Values:

- Alpha (`α`) represents the best (maximal) choice along the path to the root for the maximizer.

- Beta (`β`) represents the best (minimal) choice for the minimizer.

2. Pruning Strategy:

- As the tree is traversed, if any part of a branch is found where the minimax value is worse than a previously examined branch, further exploration of that branch is halted (pruned).

- This occurs when `beta <= alpha`, indicating no better outcome can be achieved along this branch.

Key Code Snippet:

```
MAX, MIN = 1000, -1000
from setup import*

# Returns optimal value for current player
#(Initially called for root and maximizer)
def minimax(node, depth, maximizingPlayer, alpha, beta):


    if node.is_leaf():
        return node.value

    if maximizingPlayer:

        best = MIN

        for child in node.children:

            val = minimax(depth + 1, child, False, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)

            if beta <= alpha:
                break

        return best

    else:
        best = MAX
        for child in node.children:

            val = minimax(depth + 1, child, False, alpha, beta)
            best = min(best, val)
            alpha = min(alpha, best)

            if beta <= alpha:
                break
        return best
```

```
MAX, MIN = 1000, -1000
from setup import*

# Returns optimal value for current player
#(Initially called for root and maximizer)
def minimax(node, depth, maximizingPlayer, alpha, beta):
    if node.is_leaf():
        return node.value
```

- **Constants MAX and MIN: These are used to initialize the best scores for the maximizer and minimizer. They represent the worst possible scores for each player.**

- **Leaf Node Check: The function checks if the current node is a terminal node (i.e., a game state where the game ends). If it is, the function returns the node's value, which represents the utility or score of that state.**

```
    if maximizingPlayer:
        best = MIN
         for child in node.children:
            val = minimax(depth + 1, child, False, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                break
        return best
```

Maximizing Player:

- **Initialization: best is initialized to MIN, representing the worst possible score for the maximizer.**
- **Recursive Call: For each child of the current node, the minimax function is called recursively with maximizingPlayer set to False, indicating the next move is for the minimizer.**
- **Best Value Update: The best value is updated to the maximum of its current value and the value returned from the recursive call.**
- **Alpha Update: alpha is updated to the maximum of its current value and best. This represents the best score the maximizer can guarantee at this level.**
- **Pruning Condition: If beta <= alpha, the loop breaks, pruning the remaining branches. This means the minimizer has a better option elsewhere, so further exploration is unnecessary.**

```
    else:
        best = MAX
        for child in node.children:
            val = minimax(depth + 1, child, False, alpha, beta)
            best = min(best, val)
            alpha = min(alpha, best)
            if beta <= alpha:
                break
        return best
```

Minimizing Player:

- **Initialization: best is initialized to MAX, representing the worst possible score for the minimizer.**
- **Recursive Call: For each child, the minimax function is called with maximizingPlayer set to True, indicating the next move is for the maximizer.**
- **Best Value Update: The best value is updated to the minimum of its current value and the value returned from the recursive call.**

- **Beta Update: beta is updated to the minimum of its current value and best. This represents the best score the minimizer can guarantee at this level.**
- **Pruning Condition: If beta <= alpha, the loop breaks, pruning the remaining branches.**

Key Points

- **Alpha and Beta Values: These are used to keep track of the best scores that the maximizer and minimizer can guarantee. They help in pruning branches that cannot improve the outcome.**
- **Pruning: The condition beta <= alpha is crucial for pruning. It stops further exploration of branches that cannot affect the final decision, thus optimizing the search process.**
- **Recursive Nature: The function uses recursion to explore the game tree, alternating between maximizing and minimizing at each level.**

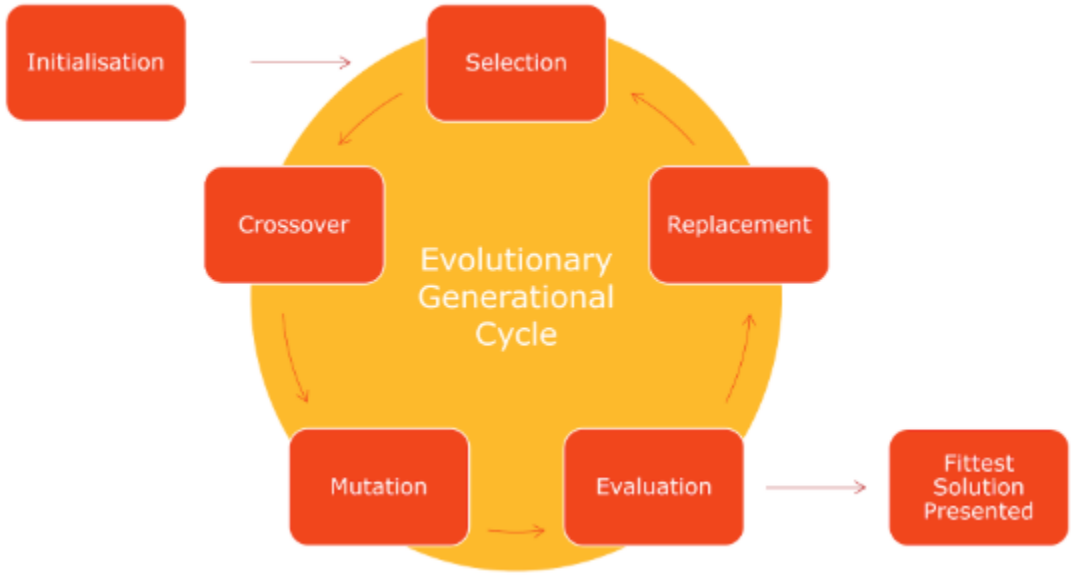
Resources:

1. https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

Genetic Algorithm (GA)

1-Overview

A “Genetic Algorithm (GA)” is a search and optimization algorithm inspired by the process of natural selection. It mimics biological evolution to solve complex problems by iteratively improving a population of candidate solutions using techniques like selection, crossover, and mutation.



2-Types of Genetic Algorithms

2.1 Basic Genetic Algorithm (Simple GA)

This is the standard version of the algorithm, focusing on the fundamental operations: selection, crossover, and mutation.

Algorithm Steps:

1. **Initialization: Create an initial population of candidate solutions randomly.**
2. **Evaluation: Evaluate each individual using a fitness function that measures solution quality.**
3. **Selection: Choose individuals based on their fitness to create a mating pool.**
4. **Crossover (Recombination): Combine pairs of individuals (parents) to produce new individuals (offspring).**
5. **Mutation: Apply random changes to some offspring to maintain diversity.**

6. **Replacement: Form a new population by selecting the best candidates from parents and offspring.**
7. **Termination: Repeat until a stopping criterion is met (e.g., maximum generations or satisfactory fitness).**

Advantages:

- **Effective for complex, multimodal problems.**
- **Can explore a wide solution space due to randomness.**

Disadvantages:

- **Requires careful tuning of parameters (e.g., mutation rate, population size).**
- **Can converge prematurely to suboptimal solutions.**

2.2 Steady-State Genetic Algorithm

In this approach, only a small portion of the population is replaced in each iteration rather than the entire population.

Key Features:

- **Maintains population stability.**
- **Focuses on gradual improvement.**

2.3 Parallel Genetic Algorithm

This variant divides the population into subpopulations (islands) that evolve independently. Occasionally, individuals migrate between subpopulations.

Key Features:

- **Increases efficiency by leveraging parallel computing.**
- **Enhances diversity across the population.**

---

3. Applications of Genetic Algorithms

1. **Optimization Problems:**
    - **Traveling Salesman Problem (TSP).**
    - **Resource allocation and scheduling.**
2. **Machine Learning:**
    - **Feature selection.**
    - **Hyperparameter tuning.**
3. **Robotics:**
    - **Designing efficient paths or behaviors.**
4. **Game AI:**
    - **Creating intelligent agents for adversarial games.**
5. **Engineering Design:**
    - **Optimizing structures and systems.**

---

4. Resources

1. https://www.javatpoint.com/genetic-algorithm-in-machine-learning
2. https://www.geeksforgeeks.org/genetic-algorithms/