Here's the updated explanation incorporating **Tree-Style Genetic Programming (TGP)** and the **John Koza breeding technique**:

**Tree-Style Genetic Programming (TGP)**

Genetic Programming (GP) is an evolutionary computation technique that **evolves computer programs** to solve problems. One of the most common forms of GP is **Tree-Style Genetic Programming (TGP)**, where programs are represented as **syntax trees** rather than fixed-length chromosomes (as in traditional Genetic Algorithms).

**Key Aspects of Tree-Style Genetic Programming**

1. **Representation**:
   * Each individual in the population is a **tree**, where:
     + **Nodes** represent operators/functions (e.g., +, -, if-else).
     + **Leaves** represent variables/constants.
   * Example:
   * +
   * / \
   * x 3

This tree represents the function **x + 3**.

1. **Selection Process**:
   * **Tournament Selection** is commonly used, where t = 7 individuals are randomly selected, and the best one is chosen based on fitness.
   * This ensures a **highly selective** process, favoring strong individuals.
2. **Crossover with John Koza’s Breeding Technique**:
   * **No explicit mutation is performed**.
   * Instead, a **biased coin flip** determines the reproduction method:
     + **90% probability**:
       - Select **two parents**.
       - Perform **subtree crossover** (swap random subtrees).
     + **10% probability**:
       - Select **one parent** and directly copy it into the next generation.
   * This **direct copying** makes the algorithm **exploitative**, favoring well-performing individuals.
3. **Crossover as Implicit Mutation**:
   * The **subtree crossover** used in TGP is highly mutative.
   * Instead of swapping single genes, entire **sub-expressions** of programs are exchanged, making **traditional mutation unnecessary**.
4. **Population Control**:
   * The crossover process may generate **one extra offspring**—simply discard it to maintain a constant population size.

**Python Implementation**

Below is a **Python implementation of Tree-Style GP** using Koza’s breeding technique.

import random

import copy

# Define a simple tree structure for Genetic Programming

class Node:

def \_\_init\_\_(self, value, left=None, right=None):

self.value = value

self.left = left

self.right = right

def evaluate(self, x):

"""Evaluate the tree using input x."""

if self.value == '+':

return self.left.evaluate(x) + self.right.evaluate(x)

elif self.value == '-':

return self.left.evaluate(x) - self.right.evaluate(x)

elif self.value == '\*':

return self.left.evaluate(x) \* self.right.evaluate(x)

elif self.value == '/':

return self.left.evaluate(x) / self.right.evaluate(x) if self.right.evaluate(x) != 0 else 1

else:

return x if self.value == 'x' else float(self.value)

def copy(self):

"""Return a deep copy of the tree."""

return copy.deepcopy(self)

# Generate a random tree (Problem-Specific)

def generate\_random\_tree(depth=3):

if depth == 0:

return Node(random.choice(['x', str(random.randint(1, 10))])) # Terminal (variable or constant)

op = random.choice(['+', '-', '\*', '/']) # Internal node

return Node(op, generate\_random\_tree(depth - 1), generate\_random\_tree(depth - 1))

# Perform subtree crossover

def crossover(parent1, parent2):

"""Swap random subtrees between two parents."""

child1, child2 = parent1.copy(), parent2.copy()

def get\_random\_subtree(node):

"""Recursively select a random subtree."""

if random.random() < 0.3 or (node.left is None and node.right is None):

return node

return get\_random\_subtree(node.left if random.random() < 0.5 else node.right)

# Swap random subtrees

subtree1 = get\_random\_subtree(child1)

subtree2 = get\_random\_subtree(child2)

subtree1.value, subtree1.left, subtree1.right = subtree2.value, subtree2.left, subtree2.right

return child1, child2

# Tournament selection

def tournament\_selection(population, fitnesses, t=7):

"""Select the best individual from a random subset of the population."""

tournament = random.sample(list(zip(population, fitnesses)), t)

return max(tournament, key=lambda x: x[1])[0] # Return best individual

# Fitness function (Example: Minimize function error)

def fitness(individual):

"""Evaluate fitness based on the accuracy of the generated function."""

test\_cases = [(x, x + 3) for x in range(-10, 10)] # Example: Learn f(x) = x + 3

error = sum(abs(individual.evaluate(x) - y) for x, y in test\_cases)

return -error # Lower error = higher fitness

# Genetic Programming Algorithm

def genetic\_programming(popsize=10, max\_generations=50):

population = [generate\_random\_tree() for \_ in range(popsize)]

best\_individual = None

for generation in range(max\_generations):

fitnesses = [fitness(ind) for ind in population]

best\_individual = max(population, key=fitness)

# Selection & Reproduction

new\_population = []

while len(new\_population) < popsize:

if random.random() < 0.9: # 90% Crossover

parent1 = tournament\_selection(population, fitnesses)

parent2 = tournament\_selection(population, fitnesses)

child1, child2 = crossover(parent1, parent2)

new\_population.append(child1)

if len(new\_population) < popsize:

new\_population.append(child2)

else: # 10% Direct Copy

parent = tournament\_selection(population, fitnesses)

new\_population.append(parent.copy())

population = new\_population # Update population

return best\_individual

# Run the Genetic Programming Algorithm

best\_tree = genetic\_programming()

print("Best Tree Function Found:", best\_tree.evaluate(5)) # Example evaluation

**Key Features of the Code**

✔ **Tree Representation**: Uses a Node class to represent **expression trees**.  
✔ **Tournament Selection (t=7)**: Ensures a **highly selective** process.  
✔ **John Koza’s Breeding Strategy**:

* **90% probability**: Select **two parents**, perform **subtree crossover**.
* **10% probability**: Select **one parent**, **directly copy** into the next generation.  
  ✔ **No explicit mutation**, but subtree crossover is inherently **mutative**.  
  ✔ **Extra offspring handling**: If an extra child is produced, it is discarded.

**Final Thoughts**

Tree-Style Genetic Programming is a powerful approach for evolving programs. The **Koza-style breeding technique** ensures strong **exploration (via subtree crossover)** and **exploitation (via direct copying)**, making the algorithm effective for optimization tasks.

Would you like any enhancements (e.g., adding mutation, visualization of trees)? 🚀