راهنمای کامل حل‌کننده ژنتیک سیستم معادلات

Complete Guide to Genetic Algorithm Equation Solver

مستندات فنی الگوریتم ژنتیک

۱ خرداد ۱۴۰۴

فهرست مطالب

۱ مقدمه ۳
۱.۱ ویژگی‌های کلیدی ۳

۲ تابع شایستگی ۳
۱.۲ چرا از این تابع شایستگی استفاده شد؟ ۳
۲.۲ دلایل انتخاب این روش: ۳
۱.۲.۲ استفاده از خطای مربعی (result**2) ۳
۲.۲.۲ معکوس خطا (1.0 / total_error) ۴
۳.۲.۲ مدیریت استثناء ۴
۴.۲.۲ تشخیص جواب دقیق ۴

۳ روش‌های جایگزین تابع شایستگی ۴
۱.۳ روش خطای خطی (Linear Error) ۴
۲.۳ روش خطای لگاریتمی (Logarithmic Error) ۵
۳.۳ روش وزن‌دار (Weighted Error) ۵
۴.۳ روش نرمالایز شده (Normalized Error) ۶
۵.۳ روش تطبیقی (Adaptive Fitness) ۶
۶.۳ روش ترکیبی چندگانه (Multi-Objective) ۷

۴ جمع‌بندی تابع شایستگی ۸
۱.۴ چرا روش فعلی (خطای مربعی معکوس) بهترین انتخاب است؟ ۸
۲.۴ کی از روش‌های دیگر استفاده کنیم؟ ۸

۵ عوامل مؤثر در الگوریتم ژنتیک ۸
۱.۵ فهرست عوامل به ترتیب اولویت ۸
۲.۵ روش انتخاب (Selection Strategy) ۹
۱.۲.۵ تأثیر: **بالا** (اولویت ۱) ۹
۲.۲.۵ روش استفاده شده در کد: ۹
۳.۲.۵ روش‌های جایگزین: ۹

۶ مثال‌های کاربردی **۱۰**
۱.۶ حل سیستم معادلات خطی ۱۰
۲.۶ حل سیستم معادلات غیرخطی ۱۱

۷ بهینه‌سازی پارامترها **۱۱**
۱.۷ راهنمای تنظیم پارامترها ۱۱
۱.۱.۷ اندازه جمعیت (Population Size) ۱۱
۲.۱.۷ نرخ جهش (Mutation Rate) ۱۲
۳.۱.۷ اندازه تورنمنت (Tournament Size) ۱۲

۸ خطاهای رایج و راه‌حل‌ها **۱۲**
۱.۸ همگرایی زودرس ۱۲
۲.۸ همگرایی کند ۱۲

۹ مطالعات موردی **۱۳**
۱.۹ سیستم معادلات درجه دوم ۱۳
۲.۹ سیستم معادلات با پارامترهای کسری ۱۳

۱۰ نتیجه‌گیری **۱۴**
۱.۱۰ نقاط قوت سیستم ۱۴
۲.۱۰ محدودیت‌ها ۱۴
۳.۱۰ پیشنهادات برای توسعه آینده ۱۴

۱۱ منابع و مراجع **۱۴**

۱۲ ضمائم **۱۵**
۱.۱۲ ضمیمه الف: کد کامل کلاس اصلی ۱۵
۲.۱۲ ضمیمه ب: مثال‌های تست ۱۸
۳.۱۲ ضمیمه ج: جدول پارامترهای بهینه ۱۹

۱ مقدمه

این مستند راهنمای کاملی برای درک و استفاده از حل‌کننده ژنتیک سیستم معادلات ارائه می‌دهد. این سیستم قادر به حل انواع مختلف معادلات خطی و غیرخطی با استفاده از الگوریتم ژنتیک پیشرفته است.

۱.۱ ویژگی‌های کلیدی

- حل سیستم‌های معادلات خطی و غیرخطی
- استفاده از استراتژی‌های متنوع جهش و ترکیب
- پشتیبانی از تابع شایستگی تطبیقی
- قابلیت راه‌اندازی مجدد خودکار
- تبدیل اعداد اعشاری به کسرهای دقیق

۲ تابع شایستگی

۱.۲ چرا از این تابع شایستگی استفاده شد؟

```
    def fitness_function(self, chromosome, equations):
    total_error = 0
    for eq in equations:
    try:
    result = eq(*chromosome)
    total_error += result**2  #
    except:
    total_error += 1e6

    if total_error < 1e-15:
    return float('inf')

    return 1.0 / total_error

```

Listing 1: اصلی شایستگی تابع

۲.۲ دلایل انتخاب این روش:

۱.۲.۲ استفاده از خطای مربعی (`result**2`)

- **دلیل**: خطاهای بزرگ بیشتر تنبیه می‌شوند
- **مزیت**: همگرایی سریع‌تر به جواب دقیق
- **مثال**: اگر دو کروموزوم خطاهای ۱.۰ و ۰.۱ داشته باشند:

- روش خطی: ۱.۰ vs ۰.۱۰ (تفاوت ۱۰ برابری)
- روش مربعی: ۰.۱۰ vs ۰.۰۰۰۱ (تفاوت ۱۰۰ برابری)

۲.۲.۲ معکوس خطا (`1.0 / total_error`)

- **دلیل**: تبدیل مسئله minimization به maximization
- **مزیت**: الگوریتم ژنتیک معمولاً برای بیشینه‌سازی طراحی شده
- **نتیجه**: خطای کمتر = شایستگی بیشتر

۳.۲.۲ مدیریت استثناء

- **جریمه سنگین** (`1e6`) برای کروموزوم‌های نامعتبر
- **جلوگیری** از تقسیم بر صفر، سرریز عددی، و خطاهای محاسباتی

۴.۲.۲ تشخیص جواب دقیق

- اگر خطا کمتر از `1e-15` باشد، شایستگی `inf` می‌شود
- **نتیجه**: توقف زودهنگام در صورت یافتن جواب دقیق

۳ روش‌های جایگزین تابع شایستگی

۱.۳ روش خطای خطی (Linear Error)

```
    def linear_fitness(self, chromosome, equations):
    total_error = 0
    for eq in equations:
    total_error += abs(eq(*chromosome))
    return 1.0 / (1.0 + total_error)

```

Listing 2: خطی شایستگی تابع

مزایا:

- ساده‌تر در پیاده‌سازی
- حساسیت کمتر به نویز
- پایداری عددی بهتر

معایب:

- همگرایی کندتر
- تمایز کمتر بین جواب‌های خوب و عالی
- ممکن است در جواب‌های تقریبی گیر کند

۲.۳ روش خطای لگاریتمی (Logarithmic Error)

```
    def log_fitness(self, chromosome, equations):
    total_error = 0
    for eq in equations:
    error = abs(eq(*chromosome))
    if error > 0:
    total_error += math.log(1 + error)
    return 1.0 / (1.0 + total_error)

```

Listing 3: لگاریتمی شایستگی تابع

مزایا:

- کاهش اثر خطاهای بزرگ
- تعادل بهتر بین دقت و پایداری
- مناسب برای مسائل با مقیاس‌های مختلف

معایب:

- پیچیدگی محاسباتی بیشتر
- ممکن است دقت نهایی کمتری داشته باشد
- نیاز به تنظیم پارامترهای اضافی

۳.۳ روش وزن‌دار (Weighted Error)

```
    def weighted_fitness(self, chromosome, equations, weights=None):
    if weights is None:
    weights = [1.0] * len(equations)

    total_error = 0
    for eq, weight in zip(equations, weights):
    error = eq(*chromosome)
    total_error += weight * (error ** 2)

    return 1.0 / (1.0 + total_error)

```

Listing 4: وزن‌دار شایستگی تابع

مزایا:

- امکان اولویت‌بندی معادلات مختلف
- انعطاف‌پذیری بالا
- کنترل بهتر بر فرآیند همگرایی

معایب:

- نیاز به تعیین وزن‌های مناسب
- پیچیدگی بیشتر در تنظیم
- ممکن است به برخی معادلات بی‌توجهی کند

۴.۳ روش نرمالایز شده (Normalized Error)

```
    def normalized_fitness(self, chromosome, equations):
    errors = []
    for eq in equations:
    errors.append(abs(eq(*chromosome)))

    #
    mean_error = sum(errors) / len(errors)
    if mean_error == 0:
    return float('inf')

    normalized_error = sum(e / mean_error for e in errors)
    return 1.0 / (1.0 + normalized_error)

```

Listing 5: تابع شایستگی نرمالایز شده

مزایا:

- تعادل بهتر بین معادلات مختلف
- کاهش اثر معادلات با مقیاس بزرگ
- عملکرد یکنواخت‌تر

معایب:

- محاسبات اضافی برای نرمالایز
- ممکن است اطلاعاتی از دست برود
- پیچیدگی در پیاده‌سازی

۵.۳ روش تطبیقی (Adaptive Fitness)

```
    def adaptive_fitness(self, chromosome, equations, generation):
    total_error = 0

    #
    adaptive_power = 2.0 + (generation / 1000.0)

    for eq in equations:
    error = abs(eq(*chromosome))
```

```
        total_error += error ** adaptive_power

        return 1.0 / (1.0 + total_error)

```

Listing 6: تطبیق‌پذیری شایستگی تابع

مزایا:

- تطبیق خودکار با پیشرفت الگوریتم
- شروع با جستجوی گسترده، ادامه با جستجوی دقیق
- بهینه‌سازی مراحل مختلف حل

معایب:

- پیچیدگی زیاد
- نیاز به تنظیم پارامترهای بیشتر
- ممکن است پیش‌بینی‌ناپذیر باشد

۶.۳ روش ترکیبی چندگانه (Multi-Objective)

```
        def multi_objective_fitness(self, chromosome, equations):
        #    :1
        total_error = sum(eq(*chromosome)**2 for eq in equations)

        #    :2           )       (
        complexity = sum(abs(x) for x in chromosome)

        #    :3
        fraction_score = sum(1 for x in chromosome
        if abs(x - round(x*4)/4) < 1e-3)

        #
        fitness = (1.0 / (1 + total_error)) * 0.7 + \
        (1.0 / (1 + complexity)) * 0.2 + \
        fraction_score * 0.1

        return fitness

```

Listing 7: چندهدفه شایستگی تابع

مزایا:

- در نظر گیری چندین معیار همزمان
- تولید جواب‌های عملی‌تر
- انعطاف‌پذیری بسیار بالا

معایب:

- پیچیدگی زیاد در طراحی و تنظیم
- ممکن است همگرایی آهسته‌تر داشته باشد
- نیاز به دانش تخصصی برای تنظیم وزن‌ها

۴ جمع‌بندی تابع شایستگی

۱.۴ چرا روش فعلی (خطای مربعی معکوس) بهترین انتخاب است؟

۱. **سادگی و کارایی**: پیاده‌سازی ساده با عملکرد عالی
۲. **همگرایی سریع**: خطاهای بزرگ شدیداً تنبیه می‌شوند
۳. **دقت بالا**: قادر به یافتن جواب‌های بسیار دقیق
۴. **پایداری**: مدیریت مناسب حالات استثنایی

۲.۴ کی از روش‌های دیگر استفاده کنیم؟

- **خطای خطی**: برای مسائل حساس به نویز
- **وزن‌دار**: وقتی برخی معادلات اهمیت بیشتری دارند
- **نرمالایز**: برای معادلات با مقیاس‌های مختلف
- **تطبیقی**: برای مسائل خیلی پیچیده
- **چندهدفه**: وقتی کیفیت جواب اهمیت دارد

این انتخاب بستگی به نوع مسئله، دقت مورد نیاز، و منابع محاسباتی در دسترس دارد.

۵ عوامل مؤثر در الگوریتم ژنتیک

۱.۵ فهرست عوامل به ترتیب اولویت

۱. روش انتخاب (Selection Strategy)
۲. عملگر ترکیب (Crossover Operator)
۳. استراتژی جهش (Mutation Strategy)
۴. مدیریت جمعیت (Population Management)
۵. پارامترهای کنترلی (Control Parameters)
۶. نحوه ایجاد جمعیت اولیه (Initialization Strategy)
۷. معیارهای توقف (Termination Criteria)

۲.۵ روش انتخاب (Selection Strategy)

۱.۲.۵ تأثیر: بالا (اولویت ۱)

دلیل: تعیین‌کننده نحوه انتشار ژن‌های خوب در جمعیت

۲.۲.۵ روش استفاده شده در کد:

```
    def selection(self, ranked_population):
    tournament = random.sample(ranked_population, self.tournament_size)
    if random.random() < 0.9:
    return max(tournament, key=lambda x: x[1])[0]
    else:
    sorted_tournament = sorted(tournament, key=lambda x: x[1], reverse=True)
    return sorted_tournament[1][0] if len(sorted_tournament) > 1 else
  sorted_tournament[0][0]

```

Listing 8: Tournament Selection

نوع: Selection Tournament با اندازه ۵

۳.۲.۵ روش‌های جایگزین:

۱.۱. Selection Wheel Roulette

```
    def roulette_wheel_selection(self, ranked_population):
    total_fitness = sum(fitness for _, fitness in ranked_population)
    pick = random.uniform(0, total_fitness)
    current = 0
    for chromosome, fitness in ranked_population:
    current += fitness
    if current > pick:
    return chromosome

```

Listing 9: Roulette Wheel Selection

مزایا:

- احتمال انتخاب متناسب با شایستگی
- حفظ تنوع بهتر
- پیاده‌سازی ساده

معایب:

- حساس به scaling مقادیر شایستگی
- ممکن است افراد ضعیف زیاد انتخاب شوند
- مشکل با شایستگی‌های منفی

Selection Rank-Based .۲.۱

```
    def rank_selection(self, ranked_population):
    n = len(ranked_population)
    #     rank                    fitness
    ranks = list(range(n, 0, -1))  #     = n     = 1
    total_rank = sum(ranks)
    pick = random.uniform(0, total_rank)
    current = 0
    for i, (chromosome, _) in enumerate(ranked_population):
    current += ranks[i]
    if current > pick:
    return chromosome

```

Listing 10: Rank-Based Selection

مزایا:

- مستقل از مقیاس شایستگی
- کنترل بهتر فشار انتخاب
- جلوگیری از dominance زودرس

معایب:

- از دست دادن اطلاعات دقیق شایستگی
- محاسبات بیشتر برای رتبه‌بندی
- ممکن است همگرایی کندتر باشد

۶ مثال‌های کاربردی

۱.۶ حل سیستم معادلات خطی

```
    #         :
    # 2x + 3y = 7
    # x - y = 1

    equations_text = [
    "2*x + 3*y - 7",
    "x - y - 1"
    ]

    \begin{ltrcode}
      \begin{lstlisting}[language=Python, caption  =           [
        #         :
        # 2x + 3y = 7
        # x - y = 1

        equations_text = [
        "2*x + 3*y - 7",
```

```
        "x - y - 1"
        ]

        solver = EnhancedGeneticEquationSolver(
        pop_size=300,
        generations=1000,
        mutation_rate=0.15
        )

        result = solve_equations(equations_text, solver)
        print(f"  : x = {result[0]}, y = {result[1]}")

```

Listing 11: مثال سیستم دو خطی سیستم غیرخطی

۲.۶ حل سیستم معادلات غیرخطی

```
        #         :
        # x² + y² = 25
        # x + y = 7

        equations_text = [
        "x**2 + y**2 - 25",
        "x + y - 7"
        ]

        solver = EnhancedGeneticEquationSolver(
        pop_size=500,
        generations=3000,
        mutation_rate=0.2,
        adaptive_mutation=True
        )

        result = solve_equations(equations_text, solver)
        print(f"  : x = {result[0]}, y = {result[1]}")

```

Listing 12: مثال سیستم غیرخطی

۷ بهینه‌سازی پارامترها

۱.۷ راهنمای تنظیم پارامترها

۱.۱.۷ اندازه جمعیت (Population Size)

- **مسائل ساده:** ۱۰۰-۳۰۰
- **مسائل متوسط:** ۳۰۰-۵۰۰
- **مسائل پیچیده:** ۵۰۰-۱۰۰۰

۲.۱.۷ نرخ جهش (Mutation Rate)

- **شروع**: ۲.۰-۳.۰ (اکتشاف بالا)
- **میانه**: ۱.۰-۲.۰ (تعادل)
- **پایان**: ۰۵.۰-۱.۰ (بهره‌برداری)

۳.۱.۷ اندازه تورنمنت (Tournament Size)

- **کوچک** (۲-۳): تنوع بیشتر، همگرایی کندتر
- **متوسط** (۴-۶): تعادل مناسب
- **بزرگ** (۷-۱۰): همگرایی سریع، خطر optimum محلی

۸ خطاهای رایج و راه‌حل‌ها

۱.۸ همگرایی زودرس

علائم:

- توقف بهبود در نسل‌های اولیه
- کاهش شدید تنوع جمعیت

راه‌حل‌ها:

- افزایش اندازه جمعیت
- کاهش اندازه تورنمنت
- افزایش نرخ جهش
- استفاده از mechanism restart

۲.۸ همگرایی کند

علائم:

- بهبود بسیار آهسته fitness
- عدم رسیدن به جواب در تعداد نسل‌های تعیین شده

راه‌حل‌ها:

- افزایش اندازه تورنمنت
- کاهش نرخ جهش
- بهبود تابع شایستگی
- استفاده از elitism بیشتر

۹ مطالعات موردی

۱.۹ سیستم معادلات درجه دوم

مسئله:

$$x^2 + y^2 = 13 \quad (۱)$$
$$x + y = 5 \quad (۲)$$

تحلیل:

- دو جواب دارد: (۲،۳) و (۳،۲)
- نیاز به تنوع بالا برای یافتن هر دو جواب
- تابع شایستگی مربعی مناسب است

پارامترهای بهینه:

```
        solver = EnhancedGeneticEquationSolver(
        pop_size=400,
        generations=2000,
        mutation_rate=0.25,
        tournament_size=4,
        elite_size=40
        )

```

۲.۹ سیستم معادلات با پارامترهای کسری

مسئله:

$$\frac{1}{2}x + \frac{1}{3}y = 1 \quad (۳)$$
$$\frac{1}{4}x - \frac{1}{6}y = 0 \quad (۴)$$

تحلیل:

- جواب کسری دارد
- نیاز به precision بالا
- استفاده از fraction_precision=True

پارامترهای بهینه:

```
        solver = EnhancedGeneticEquationSolver(
        pop_size=300,
        generations=1500,
        mutation_rate=0.15,
        fraction_precision=True,
        adaptive_mutation=True
        )

```

۱۰ نتیجه‌گیری

۱.۱۰ نقاط قوت سیستم

۱. **انعطاف‌پذیری**: قابلیت حل انواع مختلف معادلات
۲. **مقاومت**: عدم گیر کردن در optimum محلی
۳. **دقت**: یافتن جواب‌های بسیار دقیق
۴. **سازگاری**: تطبیق خودکار با انواع مسائل
۵. **کارایی**: استفاده از تکنیک‌های بهینه‌سازی

۲.۱۰ محدودیت‌ها

۱. **زمان محاسبه**: نیاز به زمان بیشتر نسبت به روش‌های مستقیم
۲. **تنظیم پارامتر**: نیاز به دانش برای تنظیم بهینه
۳. **تضمین همگرایی**: عدم تضمین یافتن جواب در زمان محدود
۴. **پیچیدگی**: درک و debug کردن سخت‌تر از روش‌های کلاسیک

۳.۱۰ پیشنهادات برای توسعه آینده

۱. **Processing Parallel**: موازی‌سازی محاسبات
۲. **Methods Hybrid**: ترکیب با روش‌های عددی کلاسیک
۳. **Learning Machine**: یادگیری خودکار پارامترها
۴. **Multi-Objective**: حل همزمان چندین هدف
۵. **Handling Constraint**: مدیریت بهتر محدودیت‌ها

۱۱ منابع و مراجع

۱. E. D. Goldberg، (۱۹۸۹). Genetic Algorithms in Search، Optimization and Machine Learning

۲. H. J. Holland، (۱۹۹۲). Adaptation in Natural and Artificial Systems

۳. Z. Michalewicz، (۱۹۹۶). Genetic Algorithms + Data Structures = Evolution Programs

۴. C. A. C. Coello، (۲۰۰۲). Theoretical and numerical constraint-handling techniques

۵. K. Deb، (۲۰۰۱). Multi-Objective Optimization using Evolutionary Algorithms

۱۲ ضمائم

۱.۱۲ ضمیمه الف: کد کامل کلاس اصلی

```
import random
import math
from fractions import Fraction

class EnhancedGeneticEquationSolver:
def __init__(self, pop_size=500, generations=5000, mutation_rate=0.2,
elite_size=50, var_range=(-100, 100), tournament_size=5,
stagnation_limit=200, adaptive_mutation=True,
fraction_precision=True):

self.pop_size = pop_size
self.generations = generations
self.base_mutation_rate = mutation_rate
self.current_mutation_rate = mutation_rate
self.elite_size = elite_size
self.var_range = var_range
self.tournament_size = tournament_size
self.stagnation_limit = stagnation_limit
self.adaptive_mutation = adaptive_mutation
self.fraction_precision = fraction_precision

#
self.generation_stats = []
self.best_fitness_history = []
self.restart_count = 0

def fitness_function(self, chromosome, equations):
total_error = 0
for eq in equations:
try:
result = eq(*chromosome)
total_error += result**2
except:
total_error += 1e6

if total_error < 1e-15:
return float('inf')

return 1.0 / total_error

def initialize_population(self, var_count):
population = []
for _ in range(self.pop_size):
chromosome = []
for _ in range(var_count):
strategy = random.choice(['integer', 'small', 'fraction', 'decimal'])

if strategy == 'integer':
value = random.randint(-10, 10)
elif strategy == 'small':
value = random.uniform(-2, 2)
elif strategy == 'fraction':
```

```
            value = random.choice([0.5, 0.25, 0.75, 1.5, 2.5, -0.5, -0.25])
            else:  # decimal
            value = random.uniform(self.var_range[0], self.var_range[1])

            chromosome.append(value)
            population.append(chromosome)
            return population

            def evaluate_population(self, population, equations):
            evaluated = []
            for chromosome in population:
            fitness = self.fitness_function(chromosome, equations)
            evaluated.append((chromosome, fitness))
            return sorted(evaluated, key=lambda x: x[1], reverse=True)

            def selection(self, ranked_population):
            tournament = random.sample(ranked_population, self.tournament_size)
            if random.random() < 0.9:
            return max(tournament, key=lambda x: x[1])[0]
            else:
            sorted_tournament = sorted(tournament, key=lambda x: x[1], reverse=True)
            return sorted_tournament[1][0] if len(sorted_tournament) > 1 else
    sorted_tournament[0][0]

            def crossover(self, parent1, parent2):
            strategy = random.choice(['uniform', 'arithmetic', 'single_point'])

            if strategy == 'uniform':
            child = [p1 if random.random() < 0.5 else p2 for p1, p2 in zip(parent1,
    parent2)]
            elif strategy == 'arithmetic':
            alpha = random.random()
            child = [alpha * p1 + (1 - alpha) * p2 for p1, p2 in zip(parent1, parent2)]
            else:  # single_point
            point = random.randint(1, len(parent1) - 1)
            child = parent1[:point] + parent2[point:]

            return child

            def mutation(self, chromosome, generation, max_generations):
            if random.random() > self.current_mutation_rate:
            return chromosome

            mutated = chromosome.copy()
            gene_index = random.randint(0, len(mutated) - 1)

            strategy = random.choices(
            ['random', 'small_change', 'fraction', 'zero', 'sign_flip'],
            weights=[0.3, 0.4, 0.2, 0.05, 0.05],
            k=1
            )[0]

            if strategy == 'random':
            mutated[gene_index] = random.uniform(self.var_range[0], self.var_range[1])
            elif strategy == 'small_change':
            change = random.gauss(0, 0.1)
            mutated[gene_index] += change
            elif strategy == 'fraction':
```

```
            mutated[gene_index] = random.choice([0.5, 0.25, 0.75, 1.0, 2.0, -0.5, -1.0])
            elif strategy == 'zero':
            mutated[gene_index] = random.uniform(-0.1, 0.1)
            elif strategy == 'sign_flip':
            mutated[gene_index] = -mutated[gene_index]

            # Update mutation rate
            if self.adaptive_mutation:
            progress = min(1.0, generation / (max_generations * 0.7))
            self.current_mutation_rate = self.base_mutation_rate * (1.0 - 0.6 * progress
    )

            return mutated

            def solve(self, equations, max_time=300):
            var_count = len(equations)
            population = self.initialize_population(var_count)

            best_solution = None
            best_fitness = 0
            stagnation_counter = 0

            for generation in range(self.generations):
            # Evaluate population
            ranked_population = self.evaluate_population(population, equations)
            current_best_fitness = ranked_population[0][1]

            # Check for improvement
            if current_best_fitness > best_fitness:
            best_fitness = current_best_fitness
            best_solution = ranked_population[0][0].copy()
            stagnation_counter = 0
            else:
            stagnation_counter += 1

            # Check termination conditions
            if best_fitness > 1e10:  # Found exact solution
            break

            if stagnation_counter >= self.stagnation_limit:
            # Restart with new population
            population = self.initialize_population(var_count)
            self.restart_count += 1
            stagnation_counter = 0
            continue

            # Create next generation
            new_population = []

            # Elitism
            for i in range(self.elite_size):
            new_population.append(ranked_population[i][0])

            # Generate offspring
            while len(new_population) < self.pop_size:
            parent1 = self.selection(ranked_population)
            parent2 = self.selection(ranked_population)
            child = self.crossover(parent1, parent2)
```

```
            child = self.mutation(child, generation, self.generations)
            new_population.append(child)

            population = new_population

            return best_solution, best_fitness

```

Listing 13: کامل کلاس EnhancedGeneticEquationSolver

۲.۱۲ ضمیمه ب: مثال‌های تست

```
            def test_linear_2var():
            equations = [
            "2*x + 3*y - 7",
            "x - y - 1"
            ]
            return equations

            def test_nonlinear_2var():
            equations = [
            "x**2 + y**2 - 25",
            "x + y - 7"
            ]
            return equations

            def test_complex_3var():
            equations = [
            "x + y + z - 6",
            "2*x - y + z - 1",
            "x + 2*y - z - 2"
            ]
            return equations

            def solve_equations(equation_strings, solver):
            # Parse equations
            equations = []
            for eq_str in equation_strings:
            # Convert to lambda function
            variables = ['x', 'y', 'z', 'w'][:len(equation_strings)]
            eq_lambda = eval(f"lambda {', '.join(variables)}: {eq_str}")
            equations.append(eq_lambda)

            # Solve
            solution, fitness = solver.solve(equations)

            return solution

```

Listing 14: مختلف توابع تست

۳.۱۲ ضمیمه ج: جدول پارامترهای بهینه

نوع مسئله	اندازه جمعیت	نرخ جهش	تعداد نسل	اندازه تورنمنت
خطی ساده	۲۰۰-۳۰۰	۱۵.۰-۲.۰	۵۰۰-۱۰۰۰	۳-۴
خطی پیچیده	۳۰۰-۵۰۰	۲.۰-۲۵.۰	۱۰۰۰-۲۰۰۰	۴-۵
غیرخطی ساده	۴۰۰-۶۰۰	۲.۰-۳.۰	۱۵۰۰-۳۰۰۰	۴-۶
غیرخطی پیچیده	۵۰۰-۸۰۰	۲۵.۰-۳۵.۰	۳۰۰۰-۵۰۰۰	۵-۷
کسری	۳۰۰-۵۰۰	۱۵.۰-۲.۰	۱۰۰۰-۲۰۰۰	۳-۵