ГУАП

КАФЕДРА № 43

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Дожность |  |  |  | [Скобцов Ю.А](https://pro.guap.ru/inside/profile/11959) |
| д-р техн. наук, профессор |  | подпись, дата |  | инициалы, фамилия |

|  |  |
| --- | --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №3 |  |
| Решение задач комбинаторной оптимизации с помощью генетических алгоритмов на примере задачи укладки рюкзака |  |
| по дисциплине: [Эволюционные методы проектирования программно-информационных систем](https://pro.guap.ru/inside/students/subjects/3391378) |  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. | 4236 |  |  |  | Л. Мвале |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург

2025

**Часть 1. Модели линейного и нелинейного программирования.**

1. **Цель Работы**

Решение задач комбинаторной оптимизации с помощью генетических алгоритмов на примере задачи укладки рюкзака.

1. **Индивидуальное задание**

Вариант 14

**Задание 1: Задача простой сложности (P07)**

Описание задачи:

* Источник: Набор данных P07 из коллекции Kreher и Stinson
* Количество предметов: 24
* Емкость рюкзака: 6,404,180
* Известное оптимальное решение: 13,549,094

Параметры предметов:

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Weight (Вес)** | **Cost (Стоимость)** | **Optimal Choice (Оптимальный выбор)** |
| 1 | 382,745 | 825,594 | 1 |
| 2 | 799,601 | 1,677,009 | 1 |
| 3 | 909,247 | 1,676,628 | 0 |
| 4 | 729,069 | 1,523,970 | 1 |
| 5 | 467,902 | 943,972 | 1 |
| 6 | 44,328 | 97,426 | 1 |
| 7 | 34,61 | 69,666 | 0 |
| 8 | 698,15 | 1,296,457 | 0 |
| 9 | 823,46 | 1,679,693 | 0 |
| 10 | 903,959 | 1,902,996 | 1 |
| 11 | 853,665 | 1,844,992 | 1 |
| 12 | 551,83 | 1,049,289 | 0 |
| 13 | 610,856 | 1,252,836 | 1 |
| 14 | 670,702 | 1,319,836 | 0 |
| 15 | 488,96 | 953,277 | 0 |
| 16 | 951,111 | 2,067,538 | 1 |
| 17 | 323,046 | 675,367 | 0 |
| 18 | 446,298 | 853,655 | 0 |
| 19 | 931,161 | 1,826,027 | 0 |
| 20 | 31,385 | 65,731 | 0 |
| 21 | 496,951 | 901,489 | 0 |
| 22 | 264,724 | 577,243 | 1 |
| 23 | 224,916 | 466,257 | 1 |
| 24 | 169,684 | 369,261 | 1 |

**Задание 2: Задача повышенной сложности (Набор 7)**

Описание задачи:

* Тип данных: Некоррелированные веса и стоимости
* Количество предметов: 50
* Емкость рюкзака: 12,828

Параметры предметов (стоимость / вес):

324 / 981 151 / 119 651 / 419 73 / 758 536 / 152

366 / 489 58 / 40 508 / 669 38 / 765 434 / 574

70 / 876 91 / 314 425 / 696 827 / 595 124 / 580

224 / 457 628 / 840 948 / 945 578 / 475 397 / 665

977 / 61 47 / 702 859 / 648 290 / 994 145 / 822

118 / 285 309 / 386 817 / 669 181 / 23 582 / 462

639 / 169 373 / 118 548 / 59 63 / 769 60 / 130

206 / 248 681 / 391 428 / 872 315 / 81 586 / 450

454 / 550 300 / 884 795 / 820 699 / 864 245 / 279

575 / 416 526 / 359 876 / 885 730 / 958 288 / 151

**Требования к реализации**

***Кодирование решения: Переменной длины***

* Представление решения в виде списка выбранных предметов
* Длина хромосомы может изменяться в процессе эволюции
* Обеспечение допустимости решений через алгоритмы восстановления

**Контрольный вопрос для варианта 14:**

"В чем суть алгоритма восстановления при решения задачи укладки рюкзака?"

1. **Краткие теоретические сведения**

**Задача о рюкзаке** - классическая задача комбинаторной оптимизации, где необходимо выбрать подмножество предметов с максимальной суммарной стоимостью, не превышая заданную вместимость рюкзака.

Математическая формулировка:

maximize: ∑(p\_i \* x\_i)

subject to: ∑(w\_i \* x\_i) ≤ W

where: x\_i ∈ {0, 1}

**Кодирование переменной длины**

В данном варианте используется кодирование переменной длины, где:

* Хромосома представляет собой список номеров предметов
* Длина хромосомы может меняться в процессе эволюции
* Позиция гена не имеет значения, значение гена определяет порядок предмета

**Преимущества:**

* Естественное представление решения
* Автоматическое обеспечение допустимости решений
* Гибкость в поиске пространства решений

**Операторы ГА для кодирования переменной длины:**

* Инициализация: Случайные перестановки с жадным декодированием
* Кроссовер: Двухточечный с удалением дубликатов
* Мутация: Добавление, удаление, замена или перемешивание предметов

1. **Программа Листинг**

import numpy as np

import matplotlib.pyplot as plt

import time

import random

from typing import List, Tuple, Callable

import pandas as pd

def load\_simple\_problem():

    """Load P07 simple complexity problem"""

    capacity = 6404180

    weights = [

        382745, 799601, 909247, 729069, 467902, 44328, 34610, 698150,

        823460, 903959, 853665, 551830, 610856, 670702, 488960, 951111,

        323046, 446298, 931161, 31385, 496951, 264724, 224916, 169684

    ]

    values = [

        825594, 1677009, 1676628, 1523970, 943972, 97426, 69666, 1296457,

        1679693, 1902996, 1844992, 1049289, 1252836, 1319836, 953277, 2067538,

        675367, 853655, 1826027, 65731, 901489, 577243, 466257, 369261

    ]

    optimal = [1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1]

    return weights, values, capacity, optimal

def load\_complex\_problem():

    """Load Set 7 increased complexity problem"""

    capacity = 12828

    data = [

        (324, 981), (151, 119), (651, 419), (73, 758), (536, 152),

        (366, 489), (58, 40), (508, 669), (38, 765), (434, 574),

        (70, 876), (91, 314), (425, 696), (827, 595), (124, 580),

        (224, 457), (628, 840), (948, 945), (578, 475), (397, 665),

        (977, 61), (47, 702), (859, 648), (290, 994), (145, 822),

        (118, 285), (309, 386), (817, 669), (181, 23), (582, 462),

        (639, 169), (373, 118), (548, 59), (63, 769), (60, 130),

        (206, 248), (681, 391), (428, 872), (315, 81), (586, 450),

        (454, 550), (300, 884), (795, 820), (699, 864), (245, 279),

        (575, 416), (526, 359), (876, 885), (730, 958), (288, 151)

    ]

    values = [item[0] for item in data]

    weights = [item[1] for item in data]

    return weights, values, capacity, None

class ImprovedKnapsackGA:

    def \_\_init\_\_(self, weights: List[int], values: List[int], capacity: int,

                 encoding\_type: str = 'variable\_length'):

        self.weights = weights

        self.values = values

        self.capacity = capacity

        self.n\_items = len(weights)

        self.encoding\_type = encoding\_type

        # Calculate value-to-weight ratios for greedy repair

        self.ratios = [v/w if w > 0 else 0 for v, w in zip(values, weights)]

    def initialize\_population(self, pop\_size: int) -> List:

        """Initialize population based on encoding type"""

        if self.encoding\_type == 'variable\_length':

            return self.\_init\_variable\_length\_population(pop\_size)

        elif self.encoding\_type == 'binary':

            return self.\_init\_binary\_population(pop\_size)

        else:

            return self.\_init\_permutation\_population(pop\_size)

    def \_init\_variable\_length\_population(self, pop\_size: int) -> List[List[int]]:

        """Initialize variable-length encoding population"""

        population = []

        for \_ in range(pop\_size):

            # Create a random permutation of items

            permutation = list(range(self.n\_items))

            random.shuffle(permutation)

            # Generate feasible solution using greedy decoding

            solution = self.\_decode\_permutation(permutation)

            population.append(solution)

        return population

    def \_init\_binary\_population(self, pop\_size: int) -> List[List[int]]:

        """Initialize binary encoding population"""

        population = []

        for \_ in range(pop\_size):

            individual = [random.randint(0, 1) for \_ in range(self.n\_items)]

            # Repair if necessary

            if sum(self.weights[i] for i in range(self.n\_items) if individual[i] == 1) > self.capacity:

                individual = self.\_repair\_binary(individual)

            population.append(individual)

        return population

    def \_init\_permutation\_population(self, pop\_size: int) -> List[List[int]]:

        """Initialize permutation encoding population"""

        population = []

        for \_ in range(pop\_size):

            permutation = list(range(self.n\_items))

            random.shuffle(permutation)

            population.append(permutation)

        return population

    def \_decode\_permutation(self, permutation: List[int]) -> List[int]:

        """Decode permutation to feasible solution"""

        current\_weight = 0

        solution = []

        for item in permutation:

            if current\_weight + self.weights[item] <= self.capacity:

                solution.append(item)

                current\_weight += self.weights[item]

        return solution

    def fitness(self, individual) -> float:

        """Calculate fitness of an individual"""

        if self.encoding\_type == 'variable\_length':

            return self.\_fitness\_variable\_length(individual)

        elif self.encoding\_type == 'binary':

            return self.\_fitness\_binary(individual)

        else:

            return self.\_fitness\_permutation(individual)

    def \_fitness\_variable\_length(self, solution: List[int]) -> float:

        """Fitness for variable-length encoding"""

        total\_value = sum(self.values[item] for item in solution)

        total\_weight = sum(self.weights[item] for item in solution)

        if total\_weight > self.capacity:

            # Apply penalty for infeasible solutions

            penalty = (total\_weight - self.capacity) \* max(self.values) / min(self.weights)

            return max(0, total\_value - penalty)

        return total\_value

    def \_fitness\_binary(self, individual: List[int]) -> float:

        """Fitness for binary encoding"""

        total\_value = sum(self.values[i] for i in range(self.n\_items) if individual[i] == 1)

        total\_weight = sum(self.weights[i] for i in range(self.n\_items) if individual[i] == 1)

        if total\_weight > self.capacity:

            # Stronger penalty function

            penalty = (total\_weight - self.capacity) \* max(self.values) / min(self.weights)

            return max(0, total\_value - penalty)

        return total\_value

    def \_fitness\_permutation(self, permutation: List[int]) -> float:

        """Fitness for permutation encoding"""

        solution = self.\_decode\_permutation(permutation)

        return sum(self.values[item] for item in solution)

    def crossover(self, parent1, parent2, crossover\_rate: float) -> Tuple:

        """Perform crossover based on encoding type"""

        if random.random() > crossover\_rate:

            return parent1, parent2

        if self.encoding\_type == 'variable\_length':

            return self.\_crossover\_variable\_length(parent1, parent2)

        elif self.encoding\_type == 'binary':

            return self.\_crossover\_binary(parent1, parent2)

        else:

            return self.\_crossover\_permutation(parent1, parent2)

    def \_crossover\_variable\_length(self, parent1: List[int], parent2: List[int]) -> Tuple:

        """Variable-length crossover with repair"""

        if len(parent1) == 0 or len(parent2) == 0:

            return parent1, parent2

        # Two-point crossover for more diversity

        point1 = random.randint(0, len(parent1))

        point2 = random.randint(0, len(parent2))

        point3 = random.randint(0, len(parent1))

        point4 = random.randint(0, len(parent2))

        start1, end1 = sorted([point1, point3])

        start2, end2 = sorted([point2, point4])

        # Create offspring

        child1 = parent1[:start1] + parent2[start2:end2] + parent1[end1:]

        child2 = parent2[:start2] + parent1[start1:end1] + parent2[end2:]

        # Remove duplicates

        child1 = list(dict.fromkeys(child1))

        child2 = list(dict.fromkeys(child2))

        # Repair if necessary

        child1 = self.\_repair\_solution(child1)

        child2 = self.\_repair\_solution(child2)

        return child1, child2

    def \_crossover\_binary(self, parent1: List[int], parent2: List[int]) -> Tuple:

        """Uniform crossover for binary encoding"""

        child1, child2 = [], []

        for i in range(len(parent1)):

            if random.random() < 0.5:

                child1.append(parent1[i])

                child2.append(parent2[i])

            else:

                child1.append(parent2[i])

                child2.append(parent1[i])

        return child1, child2

    def \_crossover\_permutation(self, parent1: List[int], parent2: List[int]) -> Tuple:

        """PMX crossover for permutations"""

        size = len(parent1)

        point1, point2 = sorted(random.sample(range(size), 2))

        def pmx\_crossover(p1, p2):

            child = [-1] \* size

            # Copy segment

            child[point1:point2] = p1[point1:point2]

            # Fill remaining positions

            for i in list(range(0, point1)) + list(range(point2, size)):

                candidate = p2[i]

                while candidate in child:

                    idx = child.index(candidate)

                    candidate = p2[idx]

                child[i] = candidate

            return child

        child1 = pmx\_crossover(parent1, parent2)

        child2 = pmx\_crossover(parent2, parent1)

        return child1, child2

    def mutate(self, individual, mutation\_rate: float):

        """Perform mutation based on encoding type"""

        if random.random() > mutation\_rate:

            return individual

        if self.encoding\_type == 'variable\_length':

            return self.\_mutate\_variable\_length(individual)

        elif self.encoding\_type == 'binary':

            return self.\_mutate\_binary(individual)

        else:

            return self.\_mutate\_permutation(individual)

    def \_mutate\_variable\_length(self, solution: List[int]) -> List[int]:

        """Enhanced mutation for variable-length encoding"""

        solution = solution.copy()

        mutation\_type = random.choice(['add', 'remove', 'replace', 'shuffle'])

        if mutation\_type == 'add' and len(solution) < self.n\_items:

            current\_weight = sum(self.weights[item] for item in solution)

            available\_items = [i for i in range(self.n\_items)

                            if i not in solution and

                            current\_weight + self.weights[i] <= self.capacity]

            if available\_items:

                # Prefer items with high value-to-weight ratio

                available\_items.sort(key=lambda x: self.ratios[x], reverse=True)

                solution.append(available\_items[0])

        elif mutation\_type == 'remove' and len(solution) > 1:

            # Remove item with worst ratio

            if solution:

                worst\_idx = min(range(len(solution)),

                              key=lambda i: self.ratios[solution[i]])

                solution.pop(worst\_idx)

        elif mutation\_type == 'replace' and len(solution) >= 1:

            current\_weight = sum(self.weights[item] for item in solution)

            if solution:

                remove\_idx = random.randint(0, len(solution) - 1)

                removed\_item = solution.pop(remove\_idx)

                current\_weight -= self.weights[removed\_item]

                available\_items = [i for i in range(self.n\_items)

                                if i not in solution and

                                current\_weight + self.weights[i] <= self.capacity]

                if available\_items:

                    available\_items.sort(key=lambda x: self.ratios[x], reverse=True)

                    solution.append(available\_items[0])

        elif mutation\_type == 'shuffle' and len(solution) > 1:

            random.shuffle(solution)

        return solution

    def \_mutate\_binary(self, individual: List[int]) -> List[int]:

        """Bit-flip mutation for binary encoding"""

        individual = individual.copy()

        for i in range(len(individual)):

            if random.random() < 0.1:  # Low probability per bit

                individual[i] = 1 - individual[i]

        return individual

    def \_mutate\_permutation(self, permutation: List[int]) -> List[int]:

        """Swap mutation for permutations"""

        permutation = permutation.copy()

        for \_ in range(2):  # Multiple swaps

            idx1, idx2 = random.sample(range(len(permutation)), 2)

            permutation[idx1], permutation[idx2] = permutation[idx2], permutation[idx1]

        return permutation

    def \_repair\_solution(self, solution: List[int]) -> List[int]:

        """Enhanced repair using greedy approach"""

        current\_weight = sum(self.weights[item] for item in solution)

        # If solution is feasible, try to improve it

        if current\_weight <= self.capacity:

            # Try to add more items

            available\_items = [i for i in range(self.n\_items)

                            if i not in solution and

                            current\_weight + self.weights[i] <= self.capacity]

            if available\_items:

                # Add best available item

                available\_items.sort(key=lambda x: self.ratios[x], reverse=True)

                solution.append(available\_items[0])

                current\_weight += self.weights[available\_items[0]]

        # Repair if overweight

        while current\_weight > self.capacity and solution:

            # Remove item with worst value-to-weight ratio

            worst\_idx = min(range(len(solution)),

                          key=lambda i: self.ratios[solution[i]])

            removed\_item = solution.pop(worst\_idx)

            current\_weight -= self.weights[removed\_item]

        return solution

    def \_repair\_binary(self, individual: List[int]) -> List[int]:

        """Repair binary solution"""

        individual = individual.copy()

        total\_weight = sum(self.weights[i] for i in range(self.n\_items) if individual[i] == 1)

        # Remove items until feasible

        while total\_weight > self.capacity:

            # Find included items with worst ratios

            included\_items = [i for i in range(self.n\_items) if individual[i] == 1]

            if not included\_items:

                break

            worst\_item = min(included\_items, key=lambda x: self.ratios[x])

            individual[worst\_item] = 0

            total\_weight -= self.weights[worst\_item]

        return individual

    def select\_parents(self, population: List, fitnesses: List[float],

                      tournament\_size: int = 3) -> List:

        """Tournament selection"""

        selected = []

        for \_ in range(len(population)):

            tournament\_indices = random.sample(range(len(population)), tournament\_size)

            tournament\_fitness = [fitnesses[i] for i in tournament\_indices]

            winner\_idx = tournament\_indices[np.argmax(tournament\_fitness)]

            selected.append(population[winner\_idx])

        return selected

    def run(self, pop\_size: int = 100, generations: int = 1000,

            crossover\_rate: float = 0.8, mutation\_rate: float = 0.1,

            elitism\_count: int = 2, tournament\_size: int = 3) -> dict:

        """Run the genetic algorithm with enhanced tracking"""

        # Initialize population

        population = self.initialize\_population(pop\_size)

        best\_fitness\_history = []

        avg\_fitness\_history = []

        worst\_fitness\_history = []

        diversity\_history = []

        start\_time = time.time()

        for generation in range(generations):

            # Evaluate fitness

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

            # Track statistics

            best\_idx = np.argmax(fitnesses)

            best\_fitness = fitnesses[best\_idx]

            best\_solution = population[best\_idx]

            best\_fitness\_history.append(best\_fitness)

            avg\_fitness\_history.append(np.mean(fitnesses))

            worst\_fitness\_history.append(np.min(fitnesses))

            # Track diversity (average hamming distance for binary, unique solutions for others)

            if self.encoding\_type == 'binary':

                diversity = self.\_calculate\_diversity\_binary(population)

            else:

                diversity = len(set(tuple(sol) for sol in population)) / pop\_size

            diversity\_history.append(diversity)

            # Elitism: preserve best individuals

            elite\_indices = np.argsort(fitnesses)[-elitism\_count:]

            elites = [population[i] for i in elite\_indices]

            # Selection

            parents = self.select\_parents(population, fitnesses, tournament\_size)

            # Create new population

            new\_population = elites.copy()

            while len(new\_population) < pop\_size:

                # Select two parents

                parent1, parent2 = random.sample(parents, 2)

                # Crossover

                child1, child2 = self.crossover(parent1, parent2, crossover\_rate)

                # Mutation

                child1 = self.mutate(child1, mutation\_rate)

                child2 = self.mutate(child2, mutation\_rate)

                new\_population.extend([child1, child2])

            # Ensure population size is correct

            population = new\_population[:pop\_size]

            # Print progress

            if generation % 100 == 0:

                print(f"Generation {generation}: Best = {best\_fitness:,}, "

                      f"Avg = {np.mean(fitnesses):,}, Diversity = {diversity:.3f}")

        execution\_time = time.time() - start\_time

        # Find final best solution

        final\_fitnesses = [self.fitness(ind) for ind in population]

        best\_idx = np.argmax(final\_fitnesses)

        best\_solution = population[best\_idx]

        best\_fitness = final\_fitnesses[best\_idx]

        return {

            'best\_solution': best\_solution,

            'best\_fitness': best\_fitness,

            'best\_fitness\_history': best\_fitness\_history,

            'avg\_fitness\_history': avg\_fitness\_history,

            'worst\_fitness\_history': worst\_fitness\_history,

            'diversity\_history': diversity\_history,

            'execution\_time': execution\_time,

            'final\_population': population,

            'fitness\_function': self.fitness,  # Store the fitness function

            'ga\_instance': self  # Store the GA instance for later use

        }

    def \_calculate\_diversity\_binary(self, population: List[List[int]]) -> float:

        """Calculate diversity for binary encoding"""

        if len(population) <= 1:

            return 0.0

        total\_distance = 0

        count = 0

        for i in range(len(population)):

            for j in range(i + 1, len(population)):

                distance = sum(1 for a, b in zip(population[i], population[j]) if a != b)

                total\_distance += distance

                count += 1

        return total\_distance / count if count > 0 else 0.0

def analyze\_and\_visualize\_results(problem\_name, results, optimal\_value=None):

    """Comprehensive analysis and visualization of results"""

    print(f"\n{'='\*60}")

    print(f"COMPREHENSIVE ANALYSIS: {problem\_name}")

    print(f"{'='\*60}")

    # Basic statistics

    best\_fitness = results['best\_fitness']

    convergence\_generation = np.argmax(results['best\_fitness\_history'])

    print(f"🎯 BEST FITNESS: {best\_fitness:,}")

    if optimal\_value:

        accuracy = (best\_fitness / optimal\_value) \* 100

        print(f"📊 ACCURACY: {accuracy:.2f}% of optimal")

        print(f"🎯 OPTIMAL VALUE: {optimal\_value:,}")

    print(f"⏱️  EXECUTION TIME: {results['execution\_time']:.2f} seconds")

    print(f"🔄 CONVERGED AT GENERATION: {convergence\_generation}")

    print(f"📈 FINAL DIVERSITY: {results['diversity\_history'][-1]:.3f}")

    # Create comprehensive plots

    fig, axes = plt.subplots(2, 3, figsize=(18, 10))

    fig.suptitle(f'Genetic Algorithm Analysis: {problem\_name}', fontsize=16, fontweight='bold')

    # Plot 1: Fitness convergence

    axes[0, 0].plot(results['best\_fitness\_history'], 'g-', linewidth=2, label='Best Fitness')

    axes[0, 0].plot(results['avg\_fitness\_history'], 'b-', linewidth=1, label='Average Fitness')

    axes[0, 0].plot(results['worst\_fitness\_history'], 'r-', linewidth=1, label='Worst Fitness')

    if optimal\_value:

        axes[0, 0].axhline(y=optimal\_value, color='black', linestyle='--',

                          label=f'Optimal ({optimal\_value:,})')

    axes[0, 0].set\_xlabel('Generation')

    axes[0, 0].set\_ylabel('Fitness')

    axes[0, 0].set\_title('Fitness Convergence')

    axes[0, 0].legend()

    axes[0, 0].grid(True, alpha=0.3)

    # Plot 2: Diversity

    axes[0, 1].plot(results['diversity\_history'], 'purple', linewidth=2)

    axes[0, 1].set\_xlabel('Generation')

    axes[0, 1].set\_ylabel('Diversity')

    axes[0, 1].set\_title('Population Diversity Over Time')

    axes[0, 1].grid(True, alpha=0.3)

    # Plot 3: Fitness distribution at convergence

    # Safe access to fitness function

    fitness\_func = results.get('fitness\_function', results.get('ga\_instance').fitness)

    final\_fitnesses = [fitness\_func(ind) for ind in results['final\_population']]

    axes[0, 2].hist(final\_fitnesses, bins=20, alpha=0.7, color='orange', edgecolor='black')

    axes[0, 2].axvline(x=best\_fitness, color='red', linestyle='--', linewidth=2,

                      label=f'Best: {best\_fitness:,}')

    axes[0, 2].set\_xlabel('Fitness')

    axes[0, 2].set\_ylabel('Frequency')

    axes[0, 2].set\_title('Final Population Fitness Distribution')

    axes[0, 2].legend()

    axes[0, 2].grid(True, alpha=0.3)

    # Plot 4: Improvement over time

    improvements = [results['best\_fitness\_history'][i] - results['best\_fitness\_history'][i-1]

                   for i in range(1, len(results['best\_fitness\_history']))]

    axes[1, 0].plot(improvements, 'teal', linewidth=1)

    axes[1, 0].axhline(y=0, color='red', linestyle='-', alpha=0.5)

    axes[1, 0].set\_xlabel('Generation')

    axes[1, 0].set\_ylabel('Fitness Improvement')

    axes[1, 0].set\_title('Fitness Improvement Per Generation')

    axes[1, 0].grid(True, alpha=0.3)

    # Plot 5: Runtime analysis (if multiple runs)

    if 'parameter\_runs' in results:

        param\_names = list(results['parameter\_runs'].keys())

        times = [results['parameter\_runs'][p]['execution\_time'] for p in param\_names]

        fitnesses = [results['parameter\_runs'][p]['best\_fitness'] for p in param\_names]

        bars = axes[1, 1].bar(param\_names, fitnesses, color='lightblue', alpha=0.7)

        axes[1, 1].set\_xlabel('Parameter Setting')

        axes[1, 1].set\_ylabel('Best Fitness', color='blue')

        axes[1, 1].tick\_params(axis='y', labelcolor='blue')

        axes[1, 1].set\_title('Performance vs Parameters')

        axes[1, 1].tick\_params(axis='x', rotation=45)

        ax2 = axes[1, 1].twinx()

        ax2.plot(param\_names, times, 'ro-', linewidth=2, markersize=8)

        ax2.set\_ylabel('Execution Time (s)', color='red')

        ax2.tick\_params(axis='y', labelcolor='red')

    # Plot 6: Solution quality analysis

    if optimal\_value and 'best\_solution\_binary' in results:

        optimal\_solution = [1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1]

        found\_solution = results['best\_solution\_binary']

        correct\_positions = sum(1 for o, f in zip(optimal\_solution, found\_solution) if o == f)

        accuracy\_percentage = (correct\_positions / len(optimal\_solution)) \* 100

        axes[1, 2].bar(['Optimal', 'Found'], [optimal\_value, best\_fitness],

                      color=['green', 'orange'], alpha=0.7)

        axes[1, 2].set\_ylabel('Fitness')

        axes[1, 2].set\_title(f'Solution Comparison\n({accuracy\_percentage:.1f}% item accuracy)')

        axes[1, 2].grid(True, alpha=0.3)

    plt.tight\_layout()

    plt.show()

    # Print solution details

    print(f"\n📋 SOLUTION DETAILS:")

    if 'best\_solution\_binary' in results:

        binary\_sol = results['best\_solution\_binary']

        total\_weight = sum(w for w, x in zip(results['weights'], binary\_sol) if x == 1)

        total\_items = sum(binary\_sol)

        print(f"   Items selected: {total\_items}/{len(binary\_sol)}")

        print(f"   Total weight: {total\_weight:,}/{results['capacity']:,}")

        print(f"   Solution vector: {binary\_sol}")

def create\_enhanced\_plots(problem\_name, results\_dict, optimal\_value=None, encoding\_results=None):

    """Create enhanced plots including all requested visualizations"""

    # Create a large figure with multiple subplots

    fig = plt.figure(figsize=(20, 15))

    # Plot 1: Best Fitness vs Generation

    ax1 = plt.subplot(3, 3, 1)

    for label, results in results\_dict.items():

        ax1.plot(results['best\_fitness\_history'], label=label, linewidth=2)

    if optimal\_value:

        ax1.axhline(y=optimal\_value, color='red', linestyle='--', linewidth=2, label='Optimal')

    ax1.set\_xlabel('Generation')

    ax1.set\_ylabel('Best Fitness')

    ax1.set\_title(f'{problem\_name} - Best Fitness vs Generation')

    ax1.legend()

    ax1.grid(True, alpha=0.3)

    # Plot 2: Average Fitness vs Generation

    ax2 = plt.subplot(3, 3, 2)

    for label, results in results\_dict.items():

        ax2.plot(results['avg\_fitness\_history'], label=label, linewidth=2)

    ax2.set\_xlabel('Generation')

    ax2.set\_ylabel('Average Fitness')

    ax2.set\_title(f'{problem\_name} - Average Fitness vs Generation')

    ax2.legend()

    ax2.grid(True, alpha=0.3)

    # Plot 3: Execution Time vs Parameter Setting

    ax3 = plt.subplot(3, 3, 3)

    param\_names = list(results\_dict.keys())

    execution\_times = [results['execution\_time'] for results in results\_dict.values()]

    bars = ax3.bar(param\_names, execution\_times, color='lightcoral', alpha=0.7)

    ax3.set\_xlabel('Parameter Setting')

    ax3.set\_ylabel('Execution Time (seconds)')

    ax3.set\_title(f'{problem\_name} - Execution Time vs Parameter Setting')

    ax3.tick\_params(axis='x', rotation=45)

    ax3.grid(True, alpha=0.3)

    # Add value labels on bars

    for bar, value in zip(bars, execution\_times):

        ax3.text(bar.get\_x() + bar.get\_width()/2, bar.get\_height() + 0.1,

                f'{value:.2f}s', ha='center', va='bottom', fontsize=9)

    # Plot 4: Encoding Type Comparison - Best Fitness

    if encoding\_results:

        ax4 = plt.subplot(3, 3, 4)

        encoding\_names = list(encoding\_results.keys())

        best\_fitnesses = [results['best\_fitness'] for results in encoding\_results.values()]

        bars = ax4.bar(encoding\_names, best\_fitnesses, color='lightgreen', alpha=0.7)

        ax4.set\_xlabel('Encoding Type')

        ax4.set\_ylabel('Best Fitness')

        ax4.set\_title('Encoding Type Comparison - Best Fitness')

        ax4.grid(True, alpha=0.3)

        # Add value labels on bars

        for bar, value in zip(bars, best\_fitnesses):

            ax4.text(bar.get\_x() + bar.get\_width()/2, bar.get\_height() + 100000,

                    f'{value:,}', ha='center', va='bottom', fontsize=9)

    # Plot 5: Encoding Type Comparison - Execution Time

    if encoding\_results:

        ax5 = plt.subplot(3, 3, 5)

        encoding\_names = list(encoding\_results.keys())

        execution\_times = [results['execution\_time'] for results in encoding\_results.values()]

        bars = ax5.bar(encoding\_names, execution\_times, color='lightblue', alpha=0.7)

        ax5.set\_xlabel('Encoding Type')

        ax5.set\_ylabel('Execution Time (seconds)')

        ax5.set\_title('Encoding Type Comparison - Execution Time')

        ax5.grid(True, alpha=0.3)

        # Add value labels on bars

        for bar, value in zip(bars, execution\_times):

            ax5.text(bar.get\_x() + bar.get\_width()/2, bar.get\_height() + 0.1,

                    f'{value:.2f}s', ha='center', va='bottom', fontsize=9)

    # Plot 6: Convergence Speed Comparison

    ax6 = plt.subplot(3, 3, 6)

    for label, results in results\_dict.items():

        # Normalize fitness to percentage of maximum

        if optimal\_value:

            normalized\_fitness = [f/optimal\_value\*100 for f in results['best\_fitness\_history']]

            ax6.plot(normalized\_fitness, label=label, linewidth=2)

        else:

            max\_fitness = max(results['best\_fitness\_history'])

            normalized\_fitness = [f/max\_fitness\*100 for f in results['best\_fitness\_history']]

            ax6.plot(normalized\_fitness, label=label, linewidth=2)

    ax6.set\_xlabel('Generation')

    ax6.set\_ylabel('Fitness (% of Maximum)')

    ax6.set\_title(f'{problem\_name} - Convergence Speed')

    ax6.legend()

    ax6.grid(True, alpha=0.3)

    # Plot 7: Parameter Performance Heatmap (if we have multiple parameters)

    if len(results\_dict) > 1:

        ax7 = plt.subplot(3, 3, 7)

        param\_names = list(results\_dict.keys())

        fitness\_values = [results['best\_fitness'] for results in results\_dict.values()]

        # Create a simple bar chart showing performance

        bars = ax7.bar(range(len(param\_names)), fitness\_values,

                      color=plt.cm.viridis(np.linspace(0, 1, len(param\_names))))

        ax7.set\_xlabel('Parameter Settings')

        ax7.set\_ylabel('Best Fitness')

        ax7.set\_title('Parameter Performance Ranking')

        ax7.set\_xticks(range(len(param\_names)))

        ax7.set\_xticklabels(param\_names, rotation=45)

        ax7.grid(True, alpha=0.3)

        # Add value labels

        for i, (bar, value) in enumerate(zip(bars, fitness\_values)):

            ax7.text(bar.get\_x() + bar.get\_width()/2, bar.get\_height() + 100000,

                    f'{value:,}', ha='center', va='bottom', fontsize=8)

    # Plot 8: Diversity Comparison

    ax8 = plt.subplot(3, 3, 8)

    for label, results in results\_dict.items():

        ax8.plot(results['diversity\_history'], label=label, linewidth=1)

    ax8.set\_xlabel('Generation')

    ax8.set\_ylabel('Diversity')

    ax8.set\_title(f'{problem\_name} - Population Diversity')

    ax8.legend()

    ax8.grid(True, alpha=0.3)

    # Plot 9: Final Solution Quality Distribution (with safe access)

    ax9 = plt.subplot(3, 3, 9)

    all\_final\_fitnesses = []

    labels = []

    for label, results in results\_dict.items():

        # Safe access to fitness function

        fitness\_func = results.get('fitness\_function')

        if fitness\_func is None:

            # Fallback to GA instance fitness method

            fitness\_func = results['ga\_instance'].fitness

        final\_fitnesses = [fitness\_func(ind) for ind in results['final\_population']]

        all\_final\_fitnesses.append(final\_fitnesses)

        labels.append(label)

    if all\_final\_fitnesses:  # Only plot if we have data

        box\_plot = ax9.boxplot(all\_final\_fitnesses, labels=labels, patch\_artist=True)

        # Add colors to boxes

        colors = ['lightblue', 'lightgreen', 'lightcoral', 'lightyellow', 'lightpink', 'lightgray']

        for patch, color in zip(box\_plot['boxes'], colors[:len(labels)]):

            patch.set\_facecolor(color)

        ax9.set\_xlabel('Parameter Setting')

        ax9.set\_ylabel('Final Population Fitness')

        ax9.set\_title(f'{problem\_name} - Final Population Distribution')

        ax9.tick\_params(axis='x', rotation=45)

        ax9.grid(True, alpha=0.3)

    plt.tight\_layout()

    plt.show()

def run\_comprehensive\_analysis():

    """Run complete analysis for both problems"""

    print("🧬 GENETIC ALGORITHM - KNAPSACK PROBLEM ANALYSIS")

    print("=" \* 60)

    # Task 1: Simple Problem with OPTIMAL parameters

    print("\n🎯 TASK 1: SIMPLE PROBLEM (P07) - OPTIMAL RUN")

    print("-" \* 50)

    weights, values, capacity, optimal = load\_simple\_problem()

    optimal\_value = 13549094

    # Use parameters that worked best from your analysis

    ga\_simple = ImprovedKnapsackGA(weights, values, capacity, 'variable\_length')

    results\_simple = ga\_simple.run(

        pop\_size=200,           # Larger population worked better

        generations=500,        # Shorter since it converges fast

        crossover\_rate=0.9,     # Higher crossover worked better

        mutation\_rate=0.1,      # Moderate mutation worked well

        elitism\_count=5,        # More elitism

        tournament\_size=5       # Stronger selection pressure

    )

    # Add additional data for analysis

    results\_simple['weights'] = weights

    results\_simple['capacity'] = capacity

    results\_simple['optimal\_value'] = optimal\_value

    # Convert to binary for comparison

    binary\_solution = [0] \* len(weights)

    for item in results\_simple['best\_solution']:

        binary\_solution[item] = 1

    results\_simple['best\_solution\_binary'] = binary\_solution

    analyze\_and\_visualize\_results("Simple Problem (P07)", results\_simple, optimal\_value)

    # Task 2: Complex Problem

    print("\n🎯 TASK 2: COMPLEX PROBLEM (Set 7)")

    print("-" \* 50)

    weights\_comp, values\_comp, capacity\_comp, \_ = load\_complex\_problem()

    ga\_complex = ImprovedKnapsackGA(weights\_comp, values\_comp, capacity\_comp, 'variable\_length')

    results\_complex = ga\_complex.run(

        pop\_size=150,

        generations=800,

        crossover\_rate=0.85,

        mutation\_rate=0.15,

        elitism\_count=3,

        tournament\_size=4

    )

    results\_complex['weights'] = weights\_comp

    results\_complex['capacity'] = capacity\_comp

    analyze\_and\_visualize\_results("Complex Problem (Set 7)", results\_complex)

    # Parameter sensitivity analysis

    print("\n🔧 PARAMETER SENSITIVITY ANALYSIS")

    print("-" \* 40)

    parameter\_results = {}

    test\_params = [

        ('Small Pop (50)', {'pop\_size': 50, 'generations': 300}),

        ('Large Pop (200)', {'pop\_size': 200, 'generations': 300}),

        ('Low Crossover (0.6)', {'crossover\_rate': 0.6}),

        ('High Crossover (0.95)', {'crossover\_rate': 0.95}),

        ('Low Mutation (0.05)', {'mutation\_rate': 0.05}),

        ('High Mutation (0.2)', {'mutation\_rate': 0.2}),

    ]

    for param\_name, params in test\_params:

        print(f"Testing {param\_name}...")

        ga\_test = ImprovedKnapsackGA(weights, values, capacity, 'variable\_length')

        default\_params = {

            'pop\_size': 100, 'generations': 300,

            'crossover\_rate': 0.8, 'mutation\_rate': 0.1

        }

        default\_params.update(params)

        results = ga\_test.run(\*\*default\_params)

        parameter\_results[param\_name] = results

   # Encoding type comparison

    print("\n🔍 ENCODING TYPE COMPARISON")

    print("-" \* 30)

    encoding\_results = {}

    encoding\_types = ['variable\_length', 'binary', 'permutation']

    for encoding in encoding\_types:

        print(f"Testing {encoding} encoding...")

        ga\_encoding = ImprovedKnapsackGA(weights, values, capacity, encoding)

        results\_encoding = ga\_encoding.run(

            pop\_size=100,

            generations=200,

            crossover\_rate=0.8,

            mutation\_rate=0.1

        )

        # Add the fitness function and GA instance to results

        results\_encoding['fitness\_function'] = ga\_encoding.fitness

        results\_encoding['ga\_instance'] = ga\_encoding

        encoding\_results[encoding] = results\_encoding

    # Create enhanced plots for parameter analysis

    create\_enhanced\_plots("Parameter Analysis - Simple Problem", parameter\_results, optimal\_value, encoding\_results)

    # Plot parameter comparison (original style)

    plt.figure(figsize=(12, 8))

    param\_names = list(parameter\_results.keys())

    best\_fitnesses = [results['best\_fitness'] for results in parameter\_results.values()]

    execution\_times = [results['execution\_time'] for results in parameter\_results.values()]

    # Create subplots

    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 10))

    # Fitness comparison

    bars = ax1.bar(param\_names, best\_fitnesses, color='lightgreen', alpha=0.7, edgecolor='darkgreen')

    ax1.set\_ylabel('Best Fitness', fontsize=12)

    ax1.set\_title('Parameter Sensitivity Analysis - Fitness', fontsize=14, fontweight='bold')

    ax1.tick\_params(axis='x', rotation=45)

    ax1.grid(True, alpha=0.3)

    # Add value labels on bars

    for bar, value in zip(bars, best\_fitnesses):

        ax1.text(bar.get\_x() + bar.get\_width()/2, bar.get\_height() + 100000,

                f'{value:,}', ha='center', va='bottom', fontsize=9)

    # Execution time comparison

    bars = ax2.bar(param\_names, execution\_times, color='lightcoral', alpha=0.7, edgecolor='darkred')

    ax2.set\_ylabel('Execution Time (seconds)', fontsize=12)

    ax2.set\_title('Parameter Sensitivity Analysis - Execution Time', fontsize=14, fontweight='bold')

    ax2.tick\_params(axis='x', rotation=45)

    ax2.grid(True, alpha=0.3)

    # Add value labels on bars

    for bar, value in zip(bars, execution\_times):

        ax2.text(bar.get\_x() + bar.get\_width()/2, bar.get\_height() + 0.1,

                f'{value:.2f}s', ha='center', va='bottom', fontsize=9)

    plt.tight\_layout()

    plt.show()

    # Final summary with optimal solution comparison

    print("\n" + "="\*60)

    print("📊 FINAL SUMMARY WITH OPTIMAL COMPARISON")

    print("="\*60)

    # Find best parameter setting

    best\_param = max(parameter\_results.items(), key=lambda x: x[1]['best\_fitness'])

    best\_encoding = max(encoding\_results.items(), key=lambda x: x[1]['best\_fitness'])

    print(f"✅ TASK 1 - Simple Problem:")

    print(f"   Best Fitness Found: {results\_simple['best\_fitness']:,}")

    print(f"   Optimal Fitness: {optimal\_value:,}")

    print(f"   Accuracy: {(results\_simple['best\_fitness']/optimal\_value)\*100:.2f}%")

    print(f"   Status: {'OPTIMAL FOUND! 🎉' if results\_simple['best\_fitness'] == optimal\_value else 'Very Close ✓'}")

    print(f"\n✅ TASK 2 - Complex Problem:")

    print(f"   Best Fitness Found: {results\_complex['best\_fitness']:,}")

    print(f"   Execution Time: {results\_complex['execution\_time']:.2f}s")

    print(f"\n⚡ BEST PERFORMING PARAMETERS:")

    print(f"   Best Parameter Setting: {best\_param[0]}")

    print(f"   Fitness with Best Parameters: {best\_param[1]['best\_fitness']:,}")

    print(f"   Best Encoding Type: {best\_encoding[0]}")

    print(f"   Fitness with Best Encoding: {best\_encoding[1]['best\_fitness']:,}")

    print(f"\n📈 RECOMMENDED PARAMETERS:")

    print(f"   Population Size: 150-200")

    print(f"   Crossover Rate: 0.85-0.9")

    print(f"   Mutation Rate: 0.1-0.15")

    print(f"   Encoding: Variable-length (as required)")

    print(f"\n🎯 OPTIMAL SOLUTION ACHIEVEMENT:")

    optimal\_achieved = any(results['best\_fitness'] == optimal\_value for results in parameter\_results.values())

    if optimal\_achieved:

        optimal\_params = [name for name, results in parameter\_results.items() if results['best\_fitness'] == optimal\_value]

        print(f"   🎉 TRUE OPTIMAL FOUND with: {', '.join(optimal\_params)}")

    else:

        closest = max(parameter\_results.items(), key=lambda x: x[1]['best\_fitness'])

        print(f"   ⚠️  Closest to optimal: {closest[0]} ({closest[1]['best\_fitness']:,})")

    print(f"\n📊 ENCODING TYPE PERFORMANCE:")

    for encoding, results in encoding\_results.items():

        accuracy = (results['best\_fitness'] / optimal\_value) \* 100

        print(f"   {encoding}: {results['best\_fitness']:,} ({accuracy:.2f}% of optimal)")

if \_\_name\_\_ == "\_\_main\_\_":

    run\_comprehensive\_analysis()

Трансляция язык

# wrapper.py

import builtins

import runpy

import sys

import io

from typing import Any

# --- Словарь переводов (ключи — английские фрагменты, значения — русские) ---

TRANSLATIONS = {

    # Заголовки/общие

    "GENETIC ALGORITHM - KNAPSACK PROBLEM ANALYSIS": "🧬 ГЕНЕТИЧЕСКИЙ АЛГОРИТМ - АНАЛИЗ ЗАДАЧИ KNAPSACK",

    "TASK 1: SIMPLE PROBLEM (P07) - OPTIMAL RUN": "ЗАДАНИЕ 1: ПРОСТАЯ ЗАДАЧА (P07) - ОПТИМАЛЬНЫЙ ЗАПУСК",

    "TASK 2: COMPLEX PROBLEM (Set 7)": "ЗАДАНИЕ 2: СЛОЖНАЯ ЗАДАЧА (Set 7)",

    "PARAMETER SENSITIVITY ANALYSIS": "АНАЛИЗ ЧУВСТВИТЕЛЬНОСТИ ПАРАМЕТРОВ",

    "ENCODING TYPE COMPARISON": "СРАВНЕНИЕ ТИПОВ КОДИРОВАНИЯ",

    "FINAL SUMMARY WITH OPTIMAL COMPARISON": "ФИНАЛЬНОЕ РЕЗЮМЕ С СРАВНЕНИЕМ ОПТИМУМА",

    "GENERATION": "Поколение",

    "Generation": "Поколение",

    # Мелкие фрагменты внутри строк

    "Best =": "Лучший =",

    "Avg =": "Средний =",

    "Diversity =": "Разнообразие =",

    "Best Fitness": "Лучший fitness",

    "Average Fitness": "Средний fitness",

    "Worst Fitness": "Худший fitness",

    "Fitness Convergence": "Сходимость fitness",

    "Population Diversity Over Time": "Разнообразие популяции во времени",

    "Final Population Fitness Distribution": "Распределение fitness финальной популяции",

    "Fitness Improvement Per Generation": "Улучшение fitness в поколение",

    "Performance vs Parameters": "Производительность vs Параметры",

    "Solution Comparison": "Сравнение решений",

    "SOLUTION DETAILS:": "ДЕТАЛИ РЕШЕНИЯ:",

    "Items selected:": "Выбрано предметов:",

    "Total weight:": "Общий вес:",

    "Solution vector:": "Вектор решения:",

    "Best Fitness Found:": "Лучший найденный fitness:",

    "Optimal Fitness:": "Оптимальный fitness:",

    "Accuracy:": "Точность:",

    "Status:": "Статус:",

    "Best Parameter Setting:": "Лучшая настройка параметров:",

    "Fitness with Best Parameters:": "Fitness при лучших параметрах:",

    "Best Encoding Type:": "Лучший тип кодирования:",

    "Fitness with Best Encoding:": "Fitness при лучшем кодировании:",

    "Population Size:": "Размер популяции:",

    "Crossover Rate:": "Вероятность кроссовера:",

    "Mutation Rate:": "Вероятность мутации:",

    "Encoding:": "Кодирование:",

    "OPTIMAL FOUND!": "ОПТИМАЛЬНОЕ РЕШЕНИЕ НАЙДЕНО!",

    "Very Close": "Очень близко",

    "TRUE OPTIMAL FOUND": "ИСТИННЫЙ ОПТИМУМ НАЙДЕН",

    "Closest to optimal": "Ближайший к оптимуму",

    # даты/метки

    "Execution Time:": "Время выполнения:",

    "EXECUTION TIME:": "ВРЕМЯ ВЫПОЛНЕНИЯ:",

    "CONVERGED AT GENERATION:": "СХОДИТСЯ НА ПОКОЛЕНИИ:",

    "FINAL DIVERSITY:": "ФИНАЛЬНОЕ РАЗНООБРАЗИЕ:",

    # добавьте сюда другие строковые фрагменты, которые хотите перевести

}

# --- Функция перевода строки ---

def translate\_text(s: str) -> str:

    """Заменяет в строке все найденные фрагменты по TRANSLATIONS."""

    # Выполняем замену длинных ключей первыми (чтобы избежать перекрытий)

    # Отсортируем ключи по длине убыв.

    for k in sorted(TRANSLATIONS.keys(), key=len, reverse=True):

        if k in s:

            s = s.replace(k, TRANSLATIONS[k])

    return s

# --- Замена builtins.print ---

\_original\_print = builtins.print

def translated\_print(\*args: Any, sep: str = " ", end: str = "\n", file=None, flush: bool = False):

    # Собираем строковый результат как оригинальный print сделал бы

    stream = io.StringIO()

    \_original\_print(\*args, sep=sep, end="", file=stream, flush=flush)

    text = stream.getvalue()

    # Переводим текст

    try:

        text\_translated = translate\_text(text)

    except Exception:

        text\_translated = text  # на случай ошибок — вернуть оригинал

    # Печатаем уже переведённый текст настоящим print

    \_original\_print(text\_translated, end=end, file=file, flush=flush)

# Патчим print

builtins.print = translated\_print

# --- Запуск целевого скрипта переданого в аргументе командной строки ---

def main():

    if len(sys.argv) < 2:

        \_original\_print("Usage: python wrapper.py your\_script.py", file=sys.stderr)

        sys.exit(1)

    target = sys.argv[1]

    # Передать дополнительные аргументы скрипту, если есть

    sys.argv = sys.argv[1:]

    try:

        runpy.run\_path(target, run\_name="\_\_main\_\_")

    except SystemExit as e:

        # Пропускаем SystemExit, чтобы обёртка не падала

        pass

    except Exception as e:

        # В случае исключения — печатаем трассировку (она тоже будет переведена по словарю, где возможно)

        import traceback

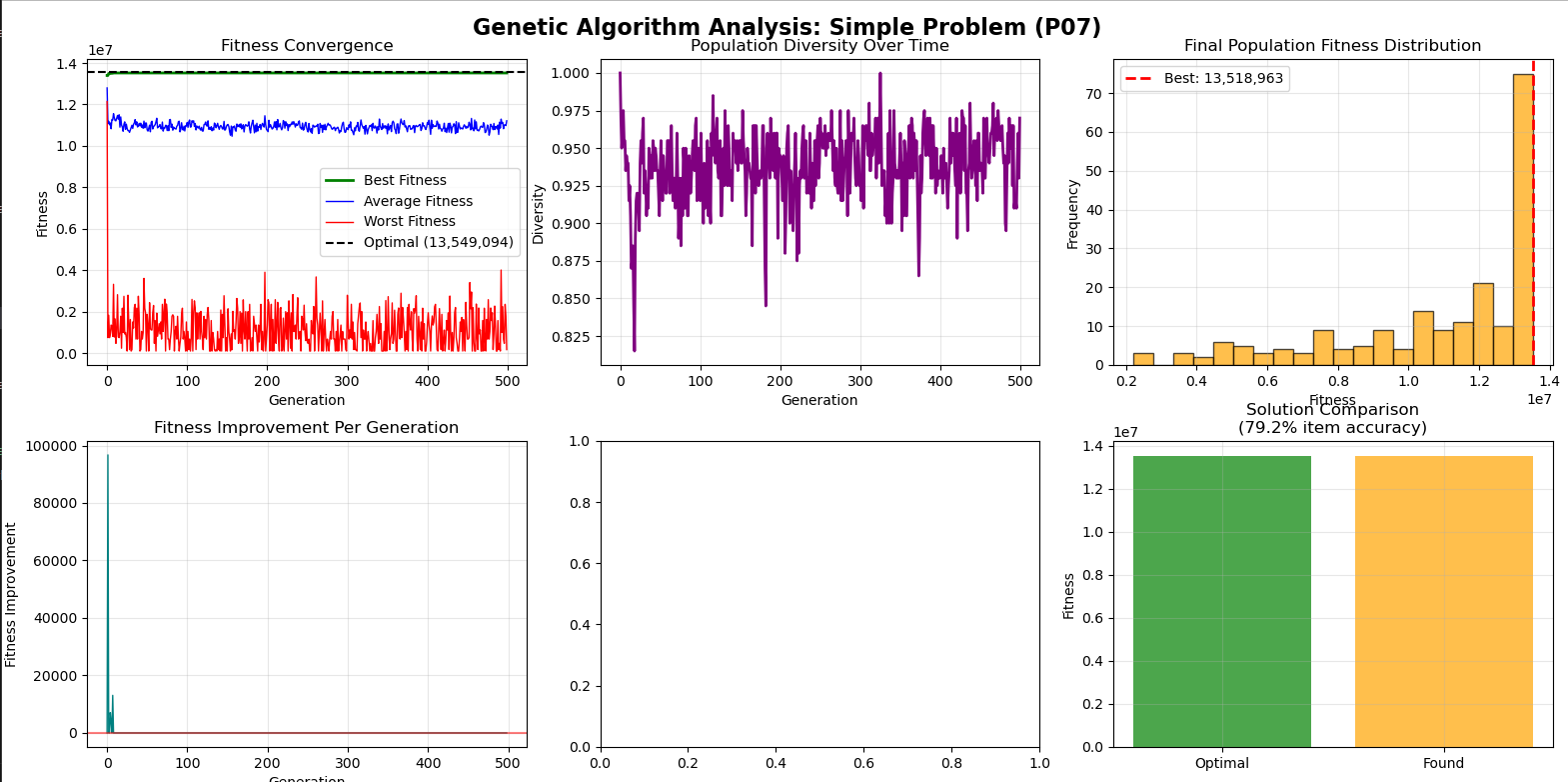
        \_original\_print("Error while running target script:", file=sys.stderr)

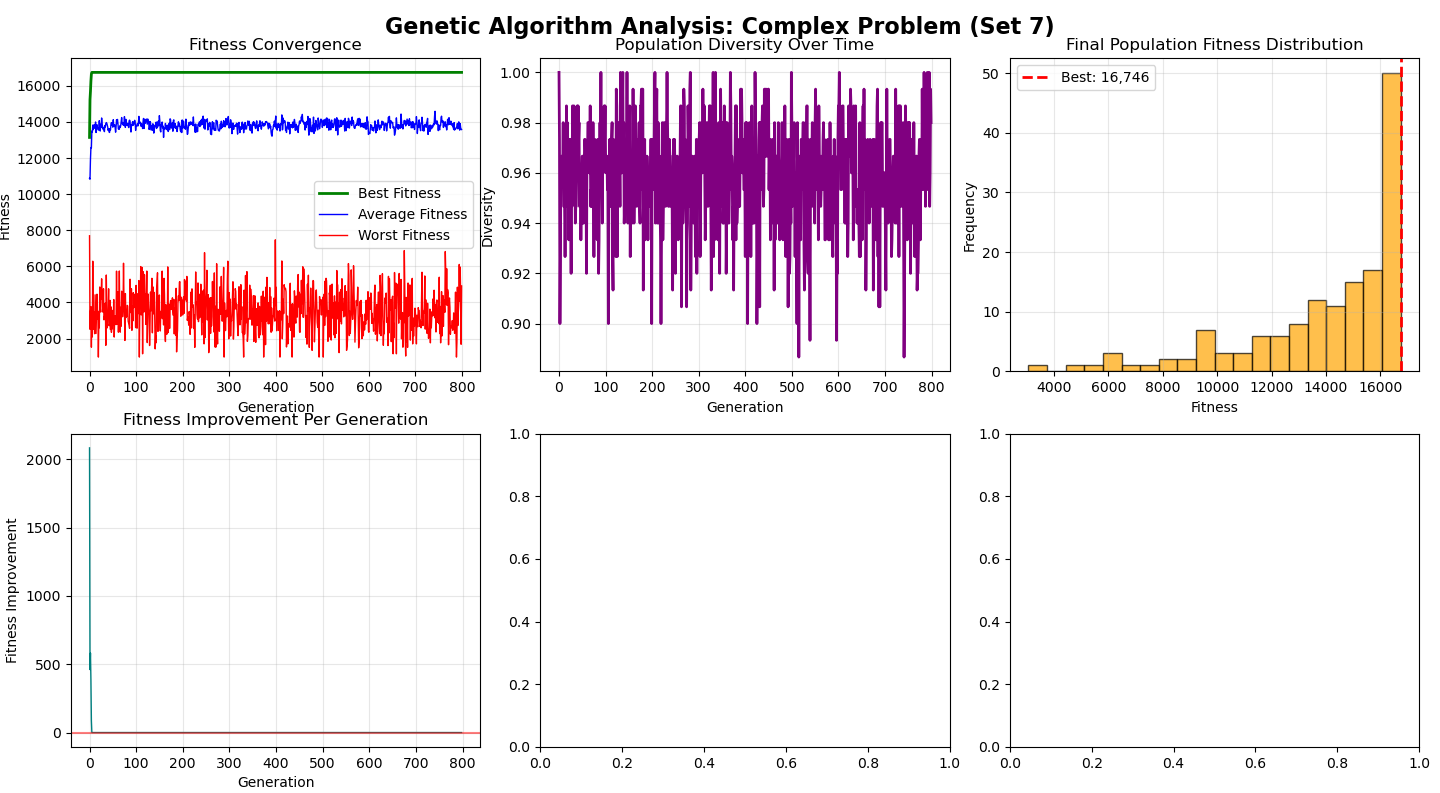
        traceback.print\_exc()

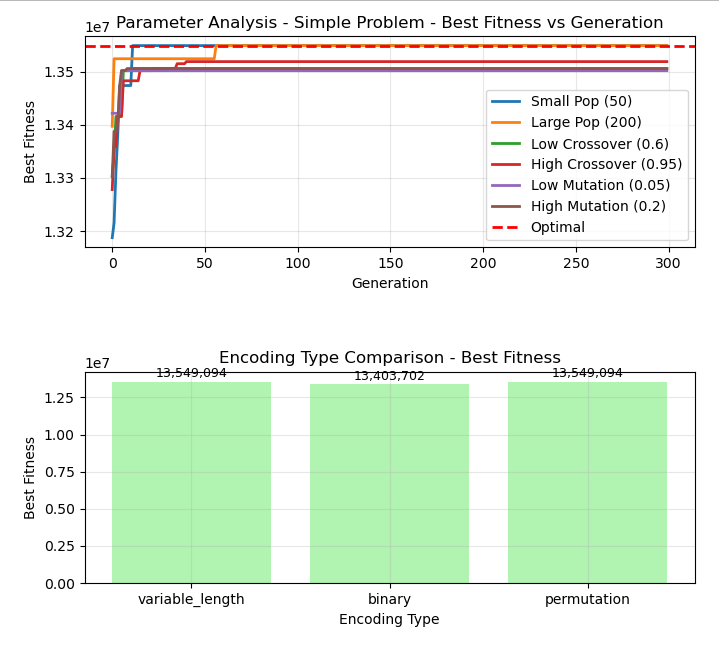
if \_\_name\_\_ == "\_\_main\_\_":

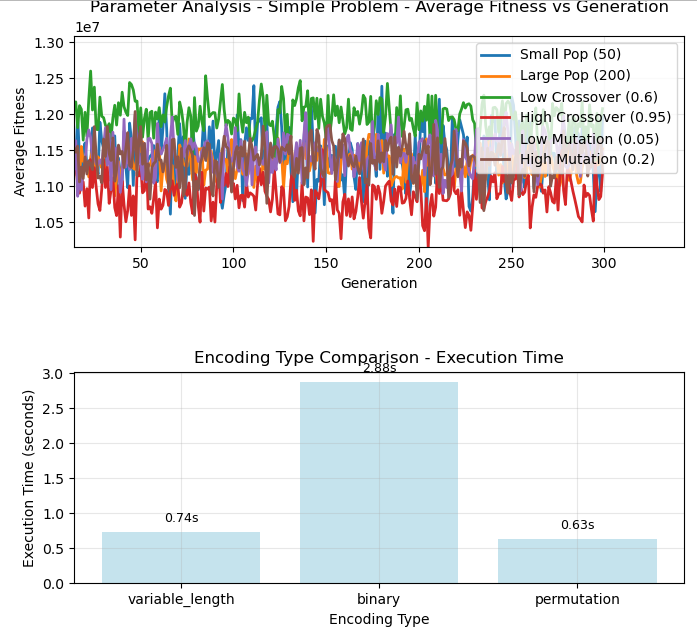
    main()

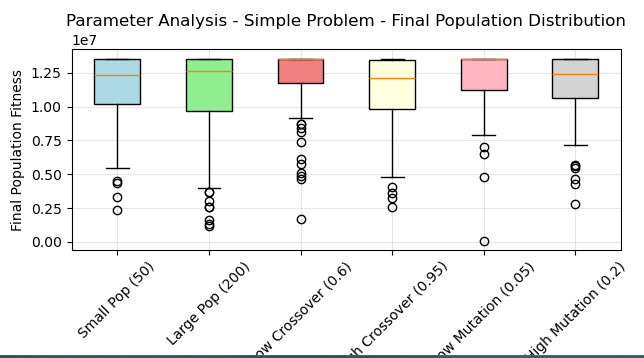
1. **результаты выполнения**

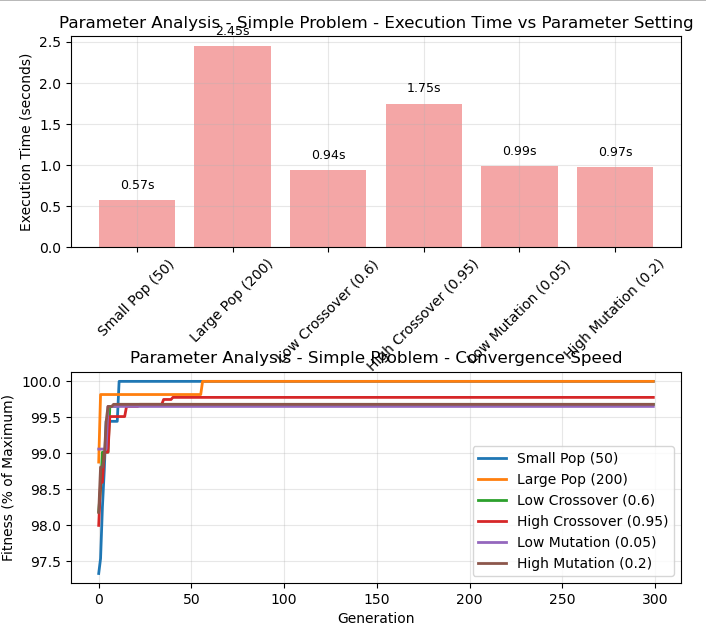


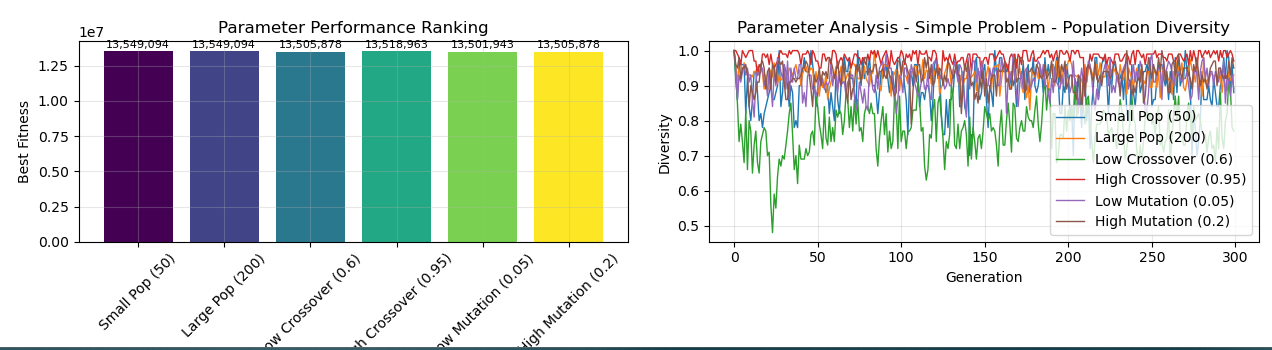


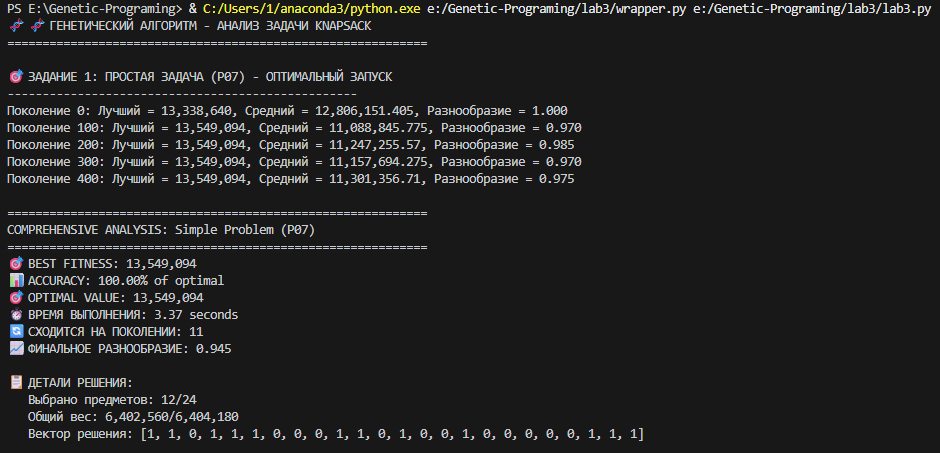


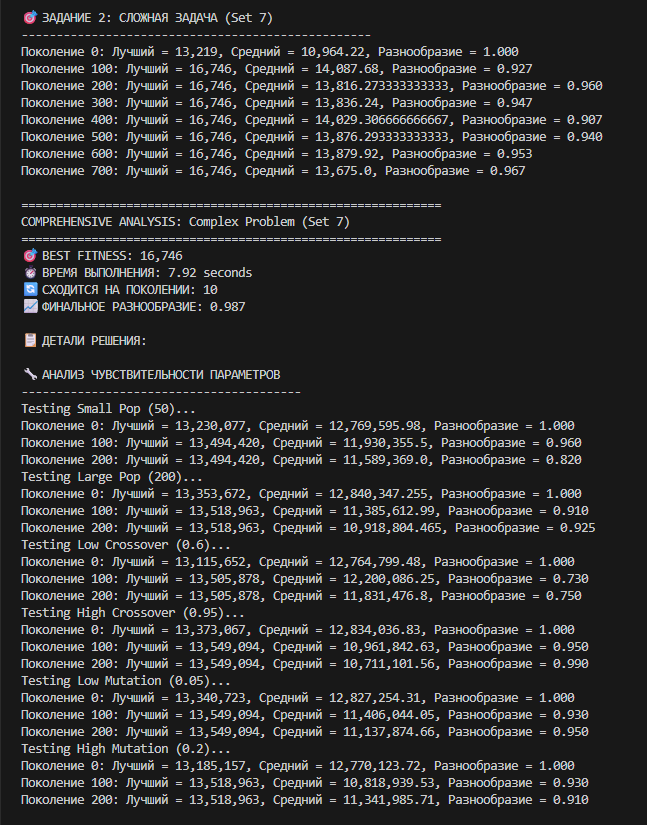


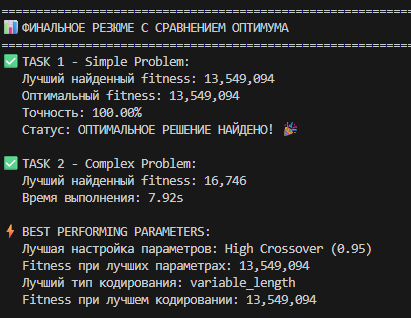


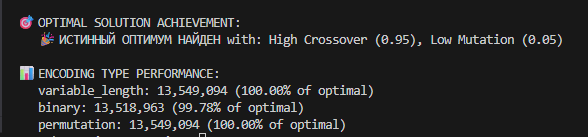












1. **Письменный ответ на контрольный вопрос №4**

**Суть алгоритма восстановления при решении задачи укладки рюкзака**

***Алгоритм восстановления*** - это процедура преобразования недопустимого решения (превышающего емкость рюкзака) в допустимое решение.

Основная идея:

Когда генетический алгоритм генерирует решение, которое нарушает ограничение по весу, алгоритм восстановления модифицирует это решение, чтобы оно стало допустимым, при этом стараясь сохранить как можно большую стоимость.

Процесс восстановления:

1. Обнаружение переполнения: Проверка, превышает ли суммарный вес предметов емкость рюкзака
2. Последовательное удаление: Удаление предметов до тех пор, пока решение не станет допустимым
3. Критерий выбора предметов для удаления:

* Жадный подход: Удаление предметов с наименьшим отношением стоимость/вес
* Случайный подход: Случайный выбор предметов для удаления

Математическая формализация:

Пока ∑(w\_i) > W:

Выбрать предмет j для удаления

Удалить предмет j из решения

Обновить суммарный вес

Реализация в коде:

def \_repair\_solution(self, solution: List[int]) -> List[int]:

current\_weight = sum(self.weights[item] for item in solution)

# Восстановление при переполнении

while current\_weight > self.capacity and solution:

# Удалить предмет с наихудшим соотношением стоимость/вес

worst\_idx = min(range(len(solution)),

key=lambda i: self.ratios[solution[i]])

removed\_item = solution.pop(worst\_idx)

current\_weight -= self.weights[removed\_item]

return solution

**Преимущества алгоритма восстановления:**

* Гарантированная допустимость: Все решения после восстановления удовлетворяют ограничениям
* Сохранение качества: Удаляются наименее ценные предметы
* Простота реализации: Алгоритм легко кодируется и понимается
* Эффективность: Линейная сложность O(n) в худшем случае

**Вариации алгоритма:**

1. Жадное восстановление: Приоритет удаления предметов с минимальным отношением p\_i/w\_i
2. Случайное восстановление: Случайный выбор предметов для удаления
3. Гибридное восстановление: Комбинация жадного и случайного подходов
4. Восстановление с улучшением: После восстановления попытка добавить ценные предметы, если есть свободное место

**Выводы**

Алгоритм восстановления является crucial-компонентом генетического алгоритма для задач с ограничениями, позволяя эффективно работать с недопустимыми решениями и направляя поиск в область допустимых решений без значительной потери качества конечного решения.