ГУАП

КАФЕДРА № 43

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

| Дожность | | Скобцов Ю.А |
|---|---|---|
| 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**

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.

### 2. 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?"

## 3. 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

## 4. 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 = []
```

```python
        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:
```

```python
            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,
```

```python
            '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 = []
```

```python
    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
```

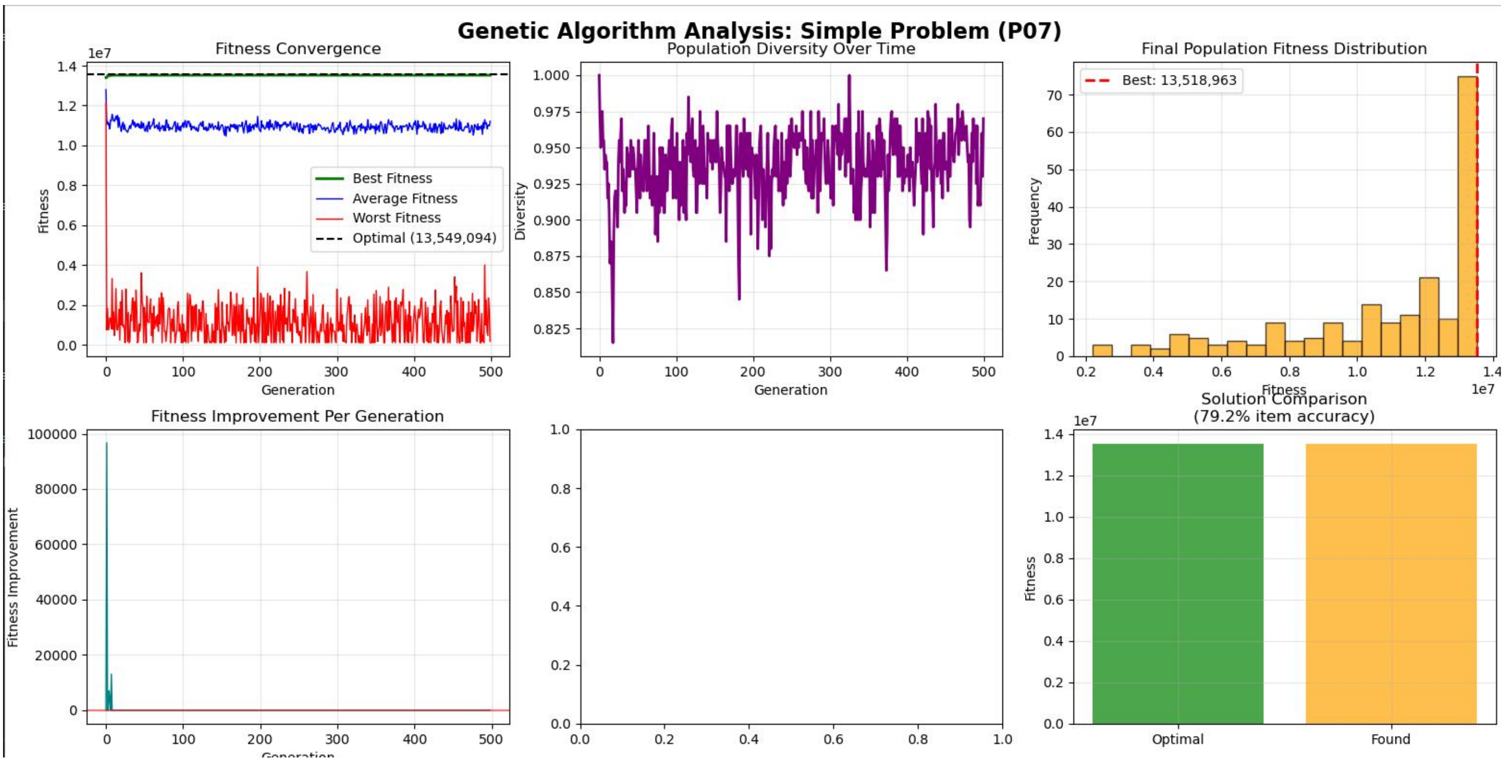
```
import io
from typing import Any

# --- 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",
```

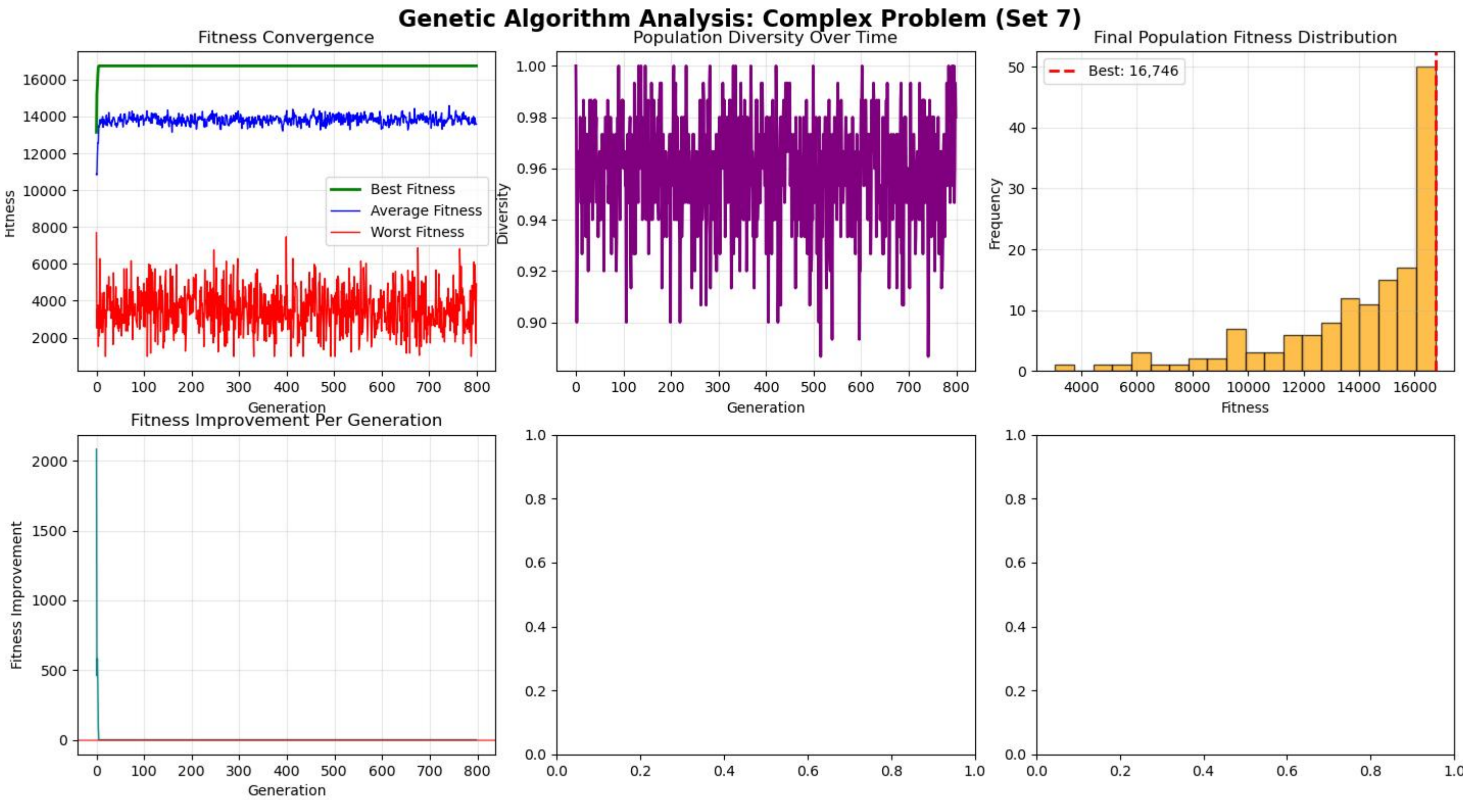
```
    "Solution Comparison": "Сравнение решений",
    "SOLUTION DETAILS:": "ДЕТАЛИ РЕШЕНИЯ:",
    "Items selected:": "Выбрано предметов:",
    "Total weight:": "Общий вес:",
    "Solution vector:": "Вектор решения:",
    "Best Fitness Found:": "Лучший найденный fitness:",
    "Optimal Fitness:": "Оптимальный 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.
```

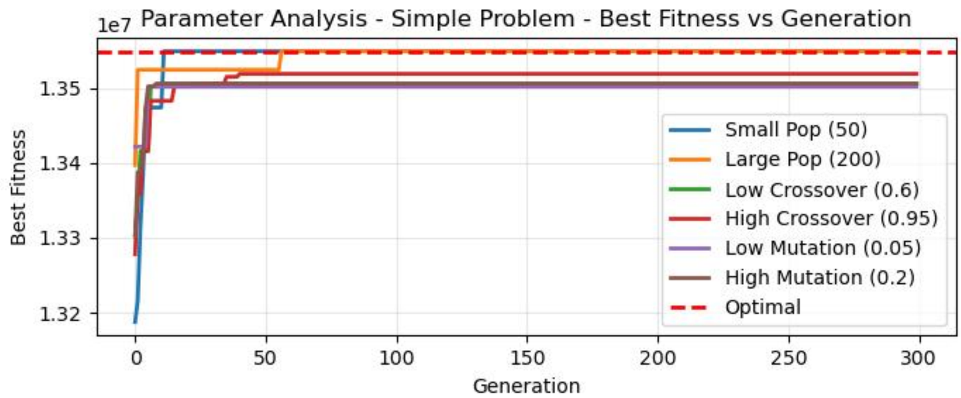
```
    for k in sorted(TRANSLATIONS.keys(), key=len, reverse=True):
        if k in s:
            s = s.replace(k, TRANSLATIONS[k])
    return s

# --- 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:
```

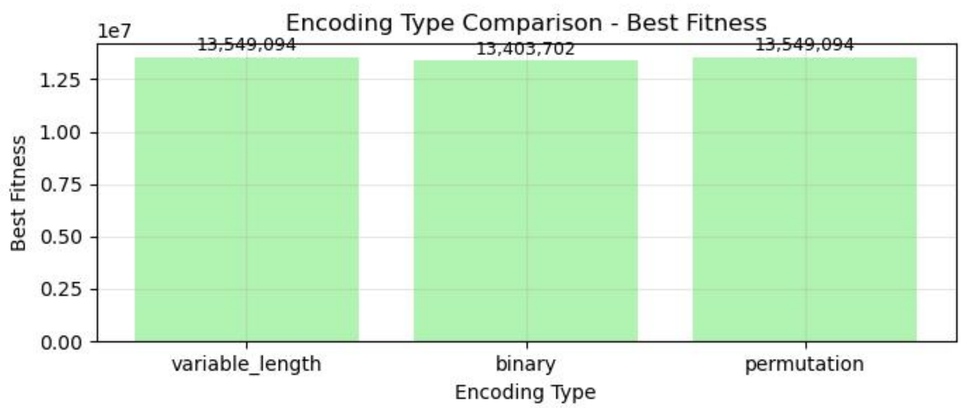
```
        runpy.run_path(target, run_name="__main__")
    except SystemExit as e:
        # Пропускаем SystemExit, чтобы обёртка не падала
        pass
    except Exception as e:
        # В случае исключения — печатаем трассировку (она тоже будет
переведена по словарю, где возможно)
        import traceback
        _original_print("Error while running target script:", file=sys.stderr)
        traceback.print_exc()


if __name__ == "__main__":
    main()
```

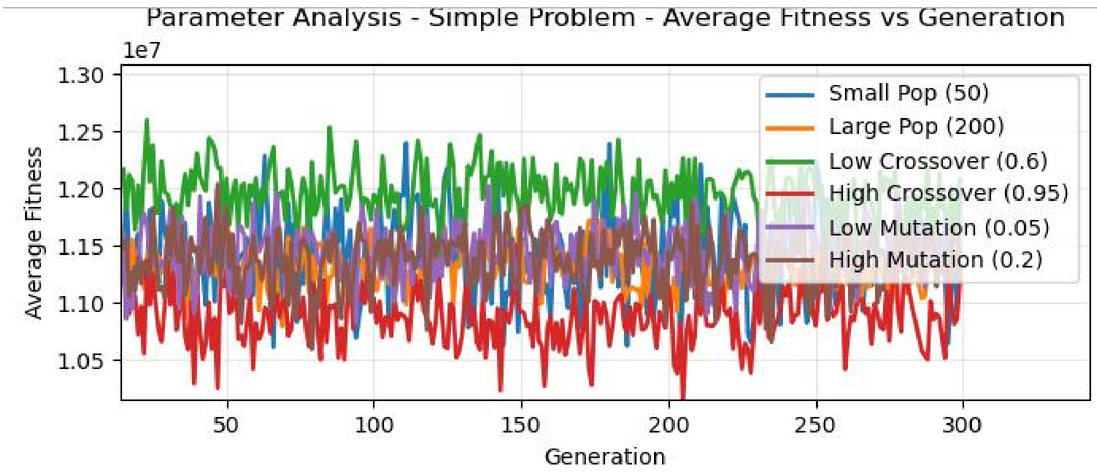
## 5. результаты выполнения

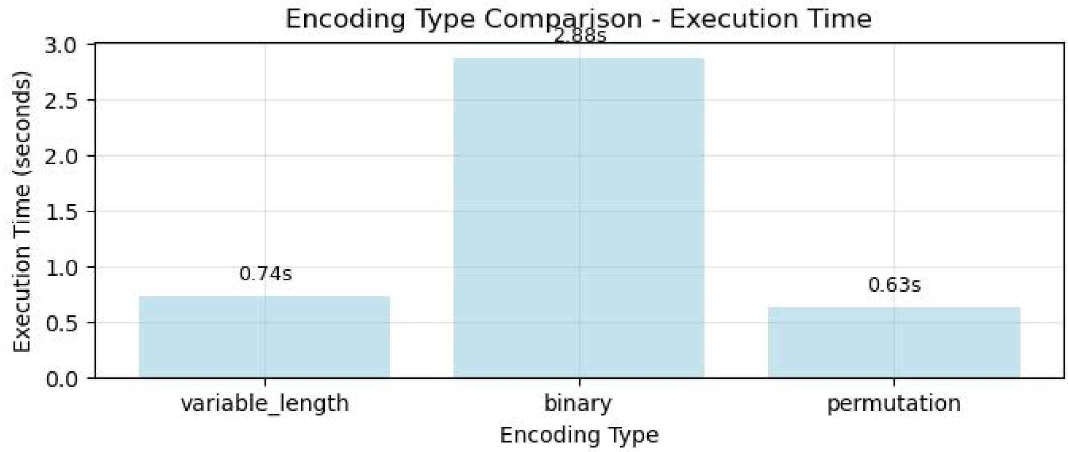


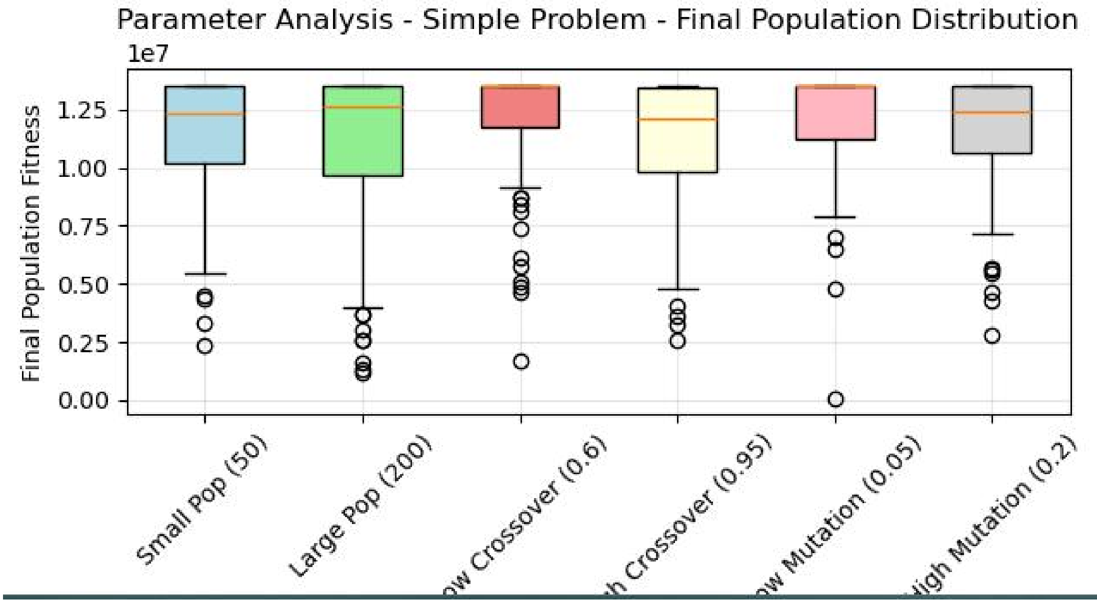

Genetic Algorithm Analysis: Complex Problem (Set 7)
Fitness Convergence
Best Fitness
Average Fitness
Worst Fitness
Fitness
Generation
Population Diversity Over Time
Diversity
Generation
Final Population Fitness Distribution
Best: 16,746
Frequency
Fitness
Fitness Improvement Per Generation
Fitness Improvement
Generation

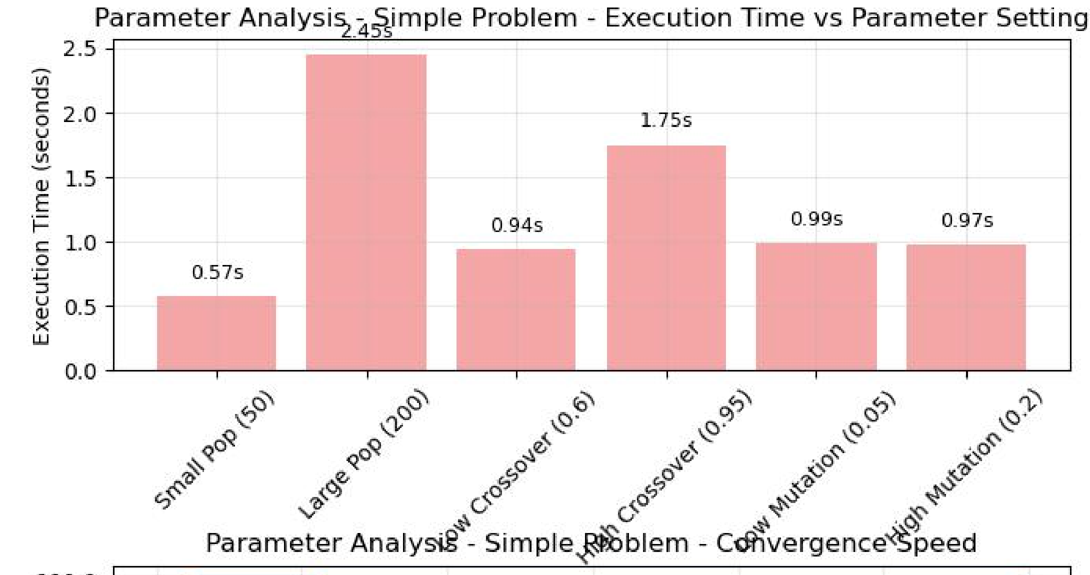

Parameter Analysis - Simple Problem - Best Fitness vs Generation
1e7
Small Pop (50)
Large Pop (200)
Low Crossover (0.6)
High Crossover (0.95)
Low Mutation (0.05)
High Mutation (0.2)
Optimal
Best Fitness
Generation


Encoding Type Comparison - Best Fitness
1e7
13,549,094
13,403,702
13,549,094
variable_length
binary
permutation
Best Fitness
Encoding Type

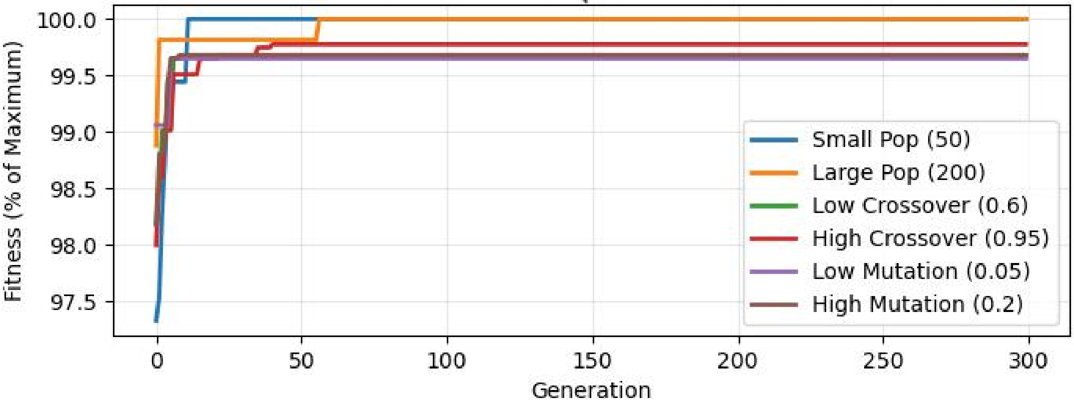

Parameter Analysis - Simple Problem - Average Fitness vs Generation
1e7
Average Fitness
Generation
Small Pop (50)
Large Pop (200)
Low Crossover (0.6)
High Crossover (0.95)
Low Mutation (0.05)
High Mutation (0.2)


Encoding Type Comparison - Execution Time
Execution Time (seconds)
0.74s
2.88s
0.63s
variable_length
binary
permutation
Encoding Type

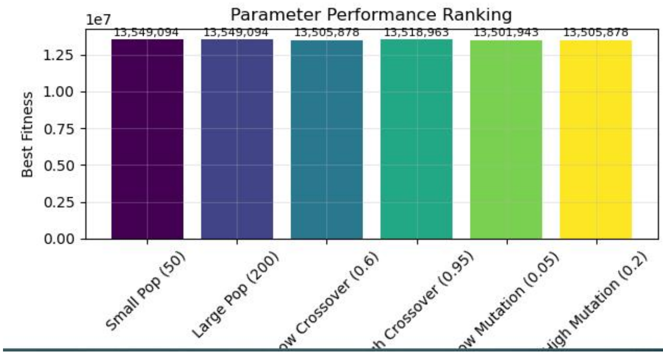

Parameter Analysis - Simple Problem - Final Population Distribution
1e7
Final Population Fitness
Small Pop (50)
Large Pop (200)



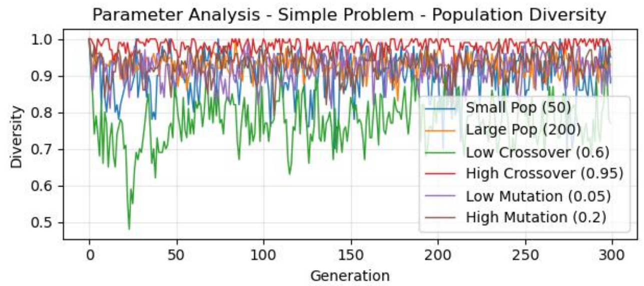
Parameter Analysis - Simple Problem - Execution Time vs Parameter Setting
Execution Time (seconds)
0.57s
2.45s
0.94s
1.75s
0.99s
0.97s
Small Pop (50)
Large Pop (200)
Low Crossover (0.6)
High Crossover (0.95)
Low Mutation (0.05)
High Mutation (0.2)

Parameter Analysis - Simple Problem - Convergence Speed
Fitness (% of Maximum)
Generation
Small Pop (50)
Large Pop (200)
Low Crossover (0.6)
High Crossover (0.95)
Low Mutation (0.05)
High Mutation (0.2)

Parameter Performance Ranking
1e7
Best Fitness
13,549,094
13,549,094
13,505,878
13,518,963
13,501,943
13,505,878
Small Pop (50)
Large Pop (200)

Parameter Analysis - Simple Problem - Population Diversity
Diversity
Generation
Small Pop (50)
Large Pop (200)
Low Crossover (0.6)
High Crossover (0.95)
Low Mutation (0.05)
High Mutation (0.2)

```
PS E:\Genetic-Programing> & C:/Users/1/anaconda3/python.exe e:/Genetic-Programing/lab3/wrapper.py
🧬 🧬 ГЕНЕТИЧЕСКИЙ АЛГОРИТМ - АНАЛИЗ ЗАДАЧИ KNAPSACK
============================================================

🎯 ЗАДАНИЕ 1: ПРОСТАЯ ЗАДАЧА (P07) - ОПТИМАЛЬНЫЙ ЗАПУСК
--------------------------------------------------
Поколение 0: Лучший = 13,338,640, Средний = 12,806,151.405, Разнообразие = 1.000
Поколение 100: Лучший = 13,549,094, Средний = 11,088,845.775, Разнообразие = 0.970
Поколение 200: Лучший = 13,549,094, Средний = 11,247,255.57, Разнообразие = 0.985
Поколение 300: Лучший = 13,549,094, Средний = 11,157,694.275, Разнообразие = 0.970
Поколение 400: Лучший = 13,549,094, Средний = 11,301,356.71, Разнообразие = 0.975

============================================================
COMPREHENSIVE ANALYSIS: Simple Problem (P07)
============================================================
🎯 BEST FITNESS: 13,549,094
📊 ACCURACY: 100.00% of optimal
🎯 OPTIMAL VALUE: 13,549,094
⏰ ВРЕМЯ ВЫПОЛНЕНИЯ: 3.37 seconds
🔄 СХОДИТСЯ НА ПОКОЛЕНИИ: 11
📈 ФИНАЛЬНОЕ РАЗНООБРАЗИЕ: 0.945

📋 ДЕТАЛИ РЕШЕНИЯ:
   Выбрано предметов: 12/24
   Общий вес: 6,402,560/6,404,180
   Вектор решения: [1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1]
```

```
🎯 ЗАДАНИЕ 2: СЛОЖНАЯ ЗАДАЧА (Set 7)
--------------------------------------------------
Поколение 0: Лучший = 13,219, Средний = 10,964.22, Разнообразие = 1.000
Поколение 100: Лучший = 16,746, Средний = 14,087.68, Разнообразие = 0.927
Поколение 200: Лучший = 16,746, Средний = 13,816.273333333333, Разнообразие = 0.960
Поколение 300: Лучший = 16,746, Средний = 13,836.24, Разнообразие = 0.947
Поколение 400: Лучший = 16,746, Средний = 14,029.306666666667, Разнообразие = 0.907
Поколение 500: Лучший = 16,746, Средний = 13,876.293333333333, Разнообразие = 0.940
Поколение 600: Лучший = 16,746, Средний = 13,879.92, Разнообразие = 0.953
Поколение 700: Лучший = 16,746, Средний = 13,675.0, Разнообразие = 0.967

============================================================
COMPREHENSIVE ANALYSIS: Complex Problem (Set 7)
============================================================
🎯 BEST FITNESS: 16,746
⏰ ВРЕМЯ ВЫПОЛНЕНИЯ: 7.92 seconds
🔄 СХОДИТСЯ НА ПОКОЛЕНИИ: 10
📈 ФИНАЛЬНОЕ РАЗНООБРАЗИЕ: 0.987

📋 ДЕТАЛИ РЕШЕНИЯ:

🔧 АНАЛИЗ ЧУВСТВИТЕЛЬНОСТИ ПАРАМЕТРОВ
----------------------------------------
Testing Small Pop (50)...
Поколение 0: Лучший = 13,230,077, Средний = 12,769,595.98, Разнообразие = 1.000
Поколение 100: Лучший = 13,494,420, Средний = 11,930,355.5, Разнообразие = 0.960
Поколение 200: Лучший = 13,494,420, Средний = 11,589,369.0, Разнообразие = 0.820
Testing Large Pop (200)...
Поколение 0: Лучший = 13,353,672, Средний = 12,840,347.255, Разнообразие = 1.000
Поколение 100: Лучший = 13,518,963, Средний = 11,385,612.99, Разнообразие = 0.910
Поколение 200: Лучший = 13,518,963, Средний = 10,918,804.465, Разнообразие = 0.925
Testing Low Crossover (0.6)...
Поколение 0: Лучший = 13,115,652, Средний = 12,764,799.48, Разнообразие = 1.000
Поколение 100: Лучший = 13,505,878, Средний = 12,200,086.25, Разнообразие = 0.730
Поколение 200: Лучший = 13,505,878, Средний = 11,831,476.8, Разнообразие = 0.750
Testing High Crossover (0.95)...
Поколение 0: Лучший = 13,373,067, Средний = 12,834,036.83, Разнообразие = 1.000
Поколение 100: Лучший = 13,549,094, Средний = 10,961,842.63, Разнообразие = 0.950
Поколение 200: Лучший = 13,549,094, Средний = 10,711,101.56, Разнообразие = 0.990
Testing Low Mutation (0.05)...
Поколение 0: Лучший = 13,340,723, Средний = 12,827,254.31, Разнообразие = 1.000
Поколение 100: Лучший = 13,549,094, Средний = 11,406,044.05, Разнообразие = 0.930
Поколение 200: Лучший = 13,549,094, Средний = 11,137,874.66, Разнообразие = 0.950
Testing High Mutation (0.2)...
Поколение 0: Лучший = 13,185,157, Средний = 12,770,123.72, Разнообразие = 1.000
Поколение 100: Лучший = 13,518,963, Средний = 10,818,939.53, Разнообразие = 0.930
Поколение 200: Лучший = 13,518,963, Средний = 11,341,985.71, Разнообразие = 0.910
```

```
================================================================
📊 ФИНАЛЬНОЕ РЕЗЮМЕ С СРАВНЕНИЕМ ОПТИМУМА
================================================================
✅ TASK 1 - Simple Problem:
   Лучший найденный fitness: 13,549,094
   Оптимальный fitness: 13,549,094
   Точность: 100.00%
   Статус: ОПТИМАЛЬНОЕ РЕШЕНИЕ НАЙДЕНО! 🎉

✅ TASK 2 - Complex Problem:
   Лучший найденный fitness: 16,746
   Время выполнения: 7.92s

⚡ BEST PERFORMING PARAMETERS:
   Лучшая настройка параметров: High Crossover (0.95)
   Fitness при лучших параметрах: 13,549,094
   Лучший тип кодирования: variable_length
   Fitness при лучшем кодировании: 13,549,094
```

```
🎯 OPTIMAL SOLUTION ACHIEVEMENT:
   🎉 ИСТИННЫЙ ОПТИМУМ НАЙДЕН with: High Crossover (0.95), Low Mutation (0.05)

📊 ENCODING TYPE PERFORMANCE:
   variable_length: 13,549,094 (100.00% of optimal)
   binary: 13,518,963 (99.78% of optimal)
   permutation: 13,549,094 (100.00% of optimal)
```

## 6. 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.