Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Ульяновский государственный технический университет»

Кафедра «Вычислительная техника»

Дисциплина «Системы искусственного интеллекта»

**Лабораторная работа 1**

**Генетические алгоритмы**

Выполнил

Студент группы ИВТАСбд-41

Панчин Н.С.

Проверил:

Старший преподаватель кафедры «ВТ»

Хайрулин И.Д.

Ульяновск

2025

# Задание

1. Необходимо разработать программу на языке python, реализующую генетический алгоритм по предложенному вариантом заданию.
2. Провести эксперименты по разным способам скрещивания (не менее 3-х), разным способам мутирования (не менее трех). Результат отобразить в виде графиков
3. Моделирование данных производить на основе максимально правдоподобных данных. Т.е. если рассматривается задача, в которой есть калорийность продуктов, то должны использоваться данные о реальных продуктах с реальной калорийностью.
4. Предоставить отчет о проделанной работе.

Вариант:

№20. На языке Python разработайте скрипт, который с помощью генетического алгоритма и полного перебора решает следующую задачу. Дано N полей для и k культур для посева. Для каждого поля известна характеристика урожайности каждой из k культур, а для каждой культуры – его закупочная стоимость. Необходимо получить самый лучший урожай за наименьшую стоимость.

# Ход работы

1. Постановка задачи и параметры

Код решает задачу оптимизации распределения сельскохозяйственных культур по полям для максимизации урожайности при минимизации затрат. Основные параметры:

* Культуры: 6 видов растений (Wheat, Corn, Barley, Soybean, Sunflower, Beet)
* Стоимость семян: фиксированные затраты на каждую культуру
* Поля: 10 участков с разной урожайностью для каждой культуры
* Матрица урожайности: 10×6 матрица FIELD\_YIELDS, где элемент [i,j] урожайность j-й культуры на i-м поле

1. Структура решения

Класс Individual представляет одно решение:

* Геном: массив из 10 элементов, где каждый элемент - индекс культуры для соответствующего поля
* Фитнес-функция: вычисляется как взвешенная сумма нормированных показателей:
* Нормированный урожай = total\_yield / MAX\_YIELD
* Нормированная стоимость = total\_cost / MAX\_COST
* Фитнес = ALPHA × нормированный\_урожай - BETA × нормированная\_стоимость
* где ALPHA=0.6, BETA=0.4 (приоритет отдается урожайности)

1. Генетический алгоритм

Инициализация популяции:

create\_random\_individual() создает особи со случайным распределением культур

Селекция:

select\_parents() использует турнирную селекцию: выбирает 5 случайных особей и берет двух лучших

Операторы кроссовера:

* Одноточечный (single\_point\_crossover) - разделение в одной точке
* Двухточечный (two\_point\_crossover) - обмен сегментом между двумя точками
* Равномерный (uniform\_crossover) - поэлементный обмен по маске

Операторы мутации:

* Случайная замена (random\_reset) - замена одной случайной культуры
* Обмен (swap\_mutation) - перестановка двух случайных культур
* Инверсия (inversion\_mutation) - обращение порядка культур на участке генома

1. Эволюционный процесс

* Параметры: POPULATION\_SIZE=20, GENERATIONS=50
* Вероятности: CROSSOVER\_PROBABILITY=0.7
* MUTATION\_PROBABILITY=0.5

Процесс: в каждой генерации создается новая популяция через селекцию, кроссовер и мутацию

1. Экспериментальная часть

Проводится сравнение 9 комбинаций операторов:

* 3 типа кроссовера × 3 типа мутации
* Для каждой комбинации отслеживается динамика фитнес-функции
* Выводится итоговое распределение культур, урожайность и затраты

1. Визуализация результатов

Строится график эволюции фитнес-функции для всех комбинаций операторов, позволяющий сравнить их эффективность и скорость сходимости.

# Тестирование

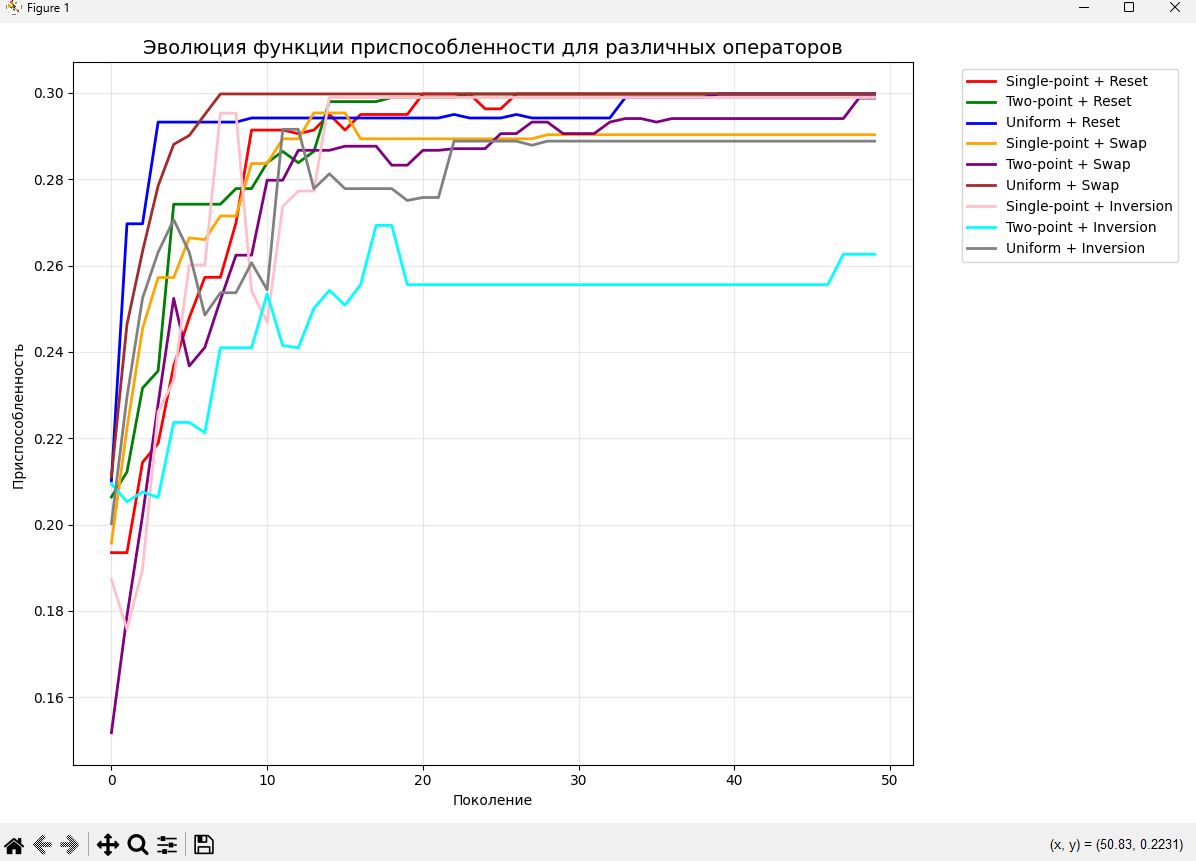


Рис. 1. График сравнения всех комбинаций всех операторов

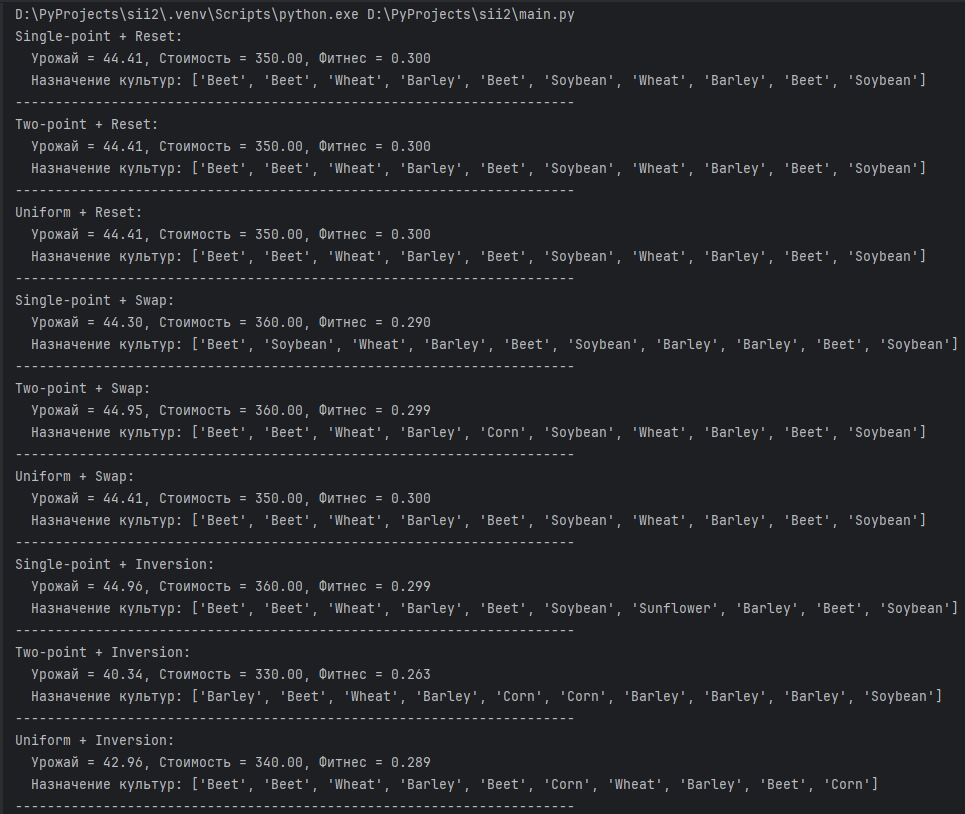


Рис. 2. Вывод в консоль вычисленных значений всех комбинаций

# Вывод

В результате выполнения лабораторной работы был успешно реализован и протестирован генетический алгоритм для оптимизации сельскохозяйственного планирования, который продемонстрировал эффективность в поиске компромиссных решений между максимизацией урожая и минимизацией затрат. Экспериментальное сравнение различных операторов скрещивания и мутации показало существенное влияние выбора генетических операторов на качество и скорость сходимости алгоритма, при этом наилучшие результаты были достигнуты при комбинациях, обеспечивающих баланс между исследованием пространства решений и использованием найденных хороших решений. Полученные результаты подтверждают практическую применимость разработанного подхода для решения задач оптимизации в сельском хозяйстве.

**Листинг**

import numpy as np  
import matplotlib.pyplot as plt  
import random  
from typing import List, Tuple, Callable  
  
# Константы  
CROPS = ["Wheat", "Corn", "Barley", "Soybean", "Sunflower", "Beet"]  
CROP\_COSTS = np.array([30, 45, 25, 50, 40, 35])  
FIELD\_YIELDS = np.array([  
 [1.72, 4.13, 2.95, 3.47, 1.59, 4.88],  
 [2.34, 3.21, 1.87, 4.45, 2.76, 3.99],  
 [4.67, 1.98, 2.43, 3.88, 4.12, 1.34],  
 [3.56, 2.67, 4.21, 1.76, 2.91, 3.44],  
 [1.88, 4.55, 3.12, 2.34, 1.99, 4.01],  
 [2.77, 3.89, 1.45, 4.67, 2.18, 3.22],  
 [4.33, 2.12, 3.76, 1.54, 4.88, 2.65],  
 [3.11, 1.67, 4.44, 2.23, 3.55, 1.88],  
 [2.56, 4.22, 3.33, 1.99, 2.77, 4.66],  
 [1.45, 3.88, 2.12, 4.55, 3.01, 2.34]  
])  
  
N\_FIELDS = FIELD\_YIELDS.shape[0]  
N\_CROPS = len(CROPS)  
  
ALPHA, BETA = 0.6, 0.4  
MUTATION\_PROBABILITY = 0.5  
CROSSOVER\_PROBABILITY = 0.7  
POPULATION\_SIZE = 20  
GENERATIONS = 50  
  
MAX\_YIELD = np.sum(np.max(FIELD\_YIELDS, axis=1))  
MAX\_COST = N\_FIELDS \* np.max(CROP\_COSTS)  
  
  
class Individual:  
 def \_\_init\_\_(self, genome: np.ndarray):  
 self.genome = genome  
 self.fitness, self.total\_yield, self.total\_cost = self.calculate\_fitness()  
  
 def calculate\_fitness(self) -> Tuple[float, float, float]:  
 total\_yield = np.sum(FIELD\_YIELDS[np.arange(N\_FIELDS), self.genome])  
 total\_cost = np.sum(CROP\_COSTS[self.genome])  
  
 norm\_yield = total\_yield / MAX\_YIELD  
 norm\_cost = total\_cost / MAX\_COST  
  
 score = ALPHA \* norm\_yield - BETA \* norm\_cost  
 return score, total\_yield, total\_cost  
  
  
def create\_random\_individual() -> Individual:  
 genome = np.random.randint(0, N\_CROPS, size=N\_FIELDS)  
 return Individual(genome)  
  
  
def select\_parents(population: List[Individual]) -> Tuple[Individual, Individual]:  
 candidates = random.sample(population, 5)  
 candidates.sort(key=lambda x: x.fitness, reverse=True)  
 return candidates[0], candidates[1]  
  
  
def single\_point\_crossover(parent1: Individual, parent2: Individual) -> Tuple[Individual, Individual]:  
 point = np.random.randint(1, N\_FIELDS)  
 child1 = np.concatenate([parent1.genome[:point], parent2.genome[point:]])  
 child2 = np.concatenate([parent2.genome[:point], parent1.genome[point:]])  
 return Individual(child1), Individual(child2)  
  
  
def two\_point\_crossover(parent1: Individual, parent2: Individual) -> Tuple[Individual, Individual]:  
 p1, p2 = sorted(np.random.randint(1, N\_FIELDS, size=2))  
 child1 = parent1.genome.copy()  
 child2 = parent2.genome.copy()  
 child1[p1:p2], child2[p1:p2] = parent2.genome[p1:p2], parent1.genome[p1:p2]  
 return Individual(child1), Individual(child2)  
  
  
def uniform\_crossover(parent1: Individual, parent2: Individual) -> Tuple[Individual, Individual]:  
 mask = np.random.randint(0, 2, size=N\_FIELDS).astype(bool)  
 child1 = np.where(mask, parent1.genome, parent2.genome)  
 child2 = np.where(mask, parent2.genome, parent1.genome)  
 return Individual(child1), Individual(child2)  
  
  
def random\_reset(ind: Individual) -> Individual:  
 genome = ind.genome.copy()  
 pos = np.random.randint(N\_FIELDS)  
 genome[pos] = np.random.randint(N\_CROPS)  
 return Individual(genome)  
  
  
def swap\_mutation(ind: Individual) -> Individual:  
 genome = ind.genome.copy()  
 i, j = np.random.choice(N\_FIELDS, 2, replace=False)  
 genome[i], genome[j] = genome[j], genome[i]  
 return Individual(genome)  
  
  
def inversion\_mutation(ind: Individual) -> Individual:  
 genome = ind.genome.copy()  
 i, j = sorted(np.random.choice(N\_FIELDS, 2, replace=False))  
 genome[i:j + 1] = genome[i:j + 1][::-1]  
 return Individual(genome)  
  
  
def evolve\_population(population: List[Individual],  
 crossover: Callable,  
 mutation: Callable) -> List[Individual]:  
 new\_population = []  
  
 while len(new\_population) < len(population):  
 parent1, parent2 = select\_parents(population)  
  
 # Скрещивание  
 if random.random() < CROSSOVER\_PROBABILITY:  
 child1, child2 = crossover(parent1, parent2)  
 else:  
 child1, child2 = parent1, parent2  
  
 # Мутация  
 if random.random() < MUTATION\_PROBABILITY:  
 child1 = mutation(child1)  
 if random.random() < MUTATION\_PROBABILITY:  
 child2 = mutation(child2)  
  
 new\_population.extend([child1, child2])  
  
 return new\_population[:len(population)]  
  
  
def run\_experiment(crossover: Callable, mutation: Callable,  
 label: str, color: str, ax: plt.Axes) -> None:  
 population = [create\_random\_individual() for \_ in range(POPULATION\_SIZE)]  
 best\_fitness\_over\_time = []  
 best\_individuals = []  
  
 for generation in range(GENERATIONS):  
 population = evolve\_population(population, crossover, mutation)  
 best\_individual = max(population, key=lambda ind: ind.fitness)  
 best\_fitness\_over\_time.append(best\_individual.fitness)  
 best\_individuals.append(best\_individual)  
  
 ax.plot(best\_fitness\_over\_time, label=label, color=color, linewidth=2)  
  
 best = best\_individuals[-1]  
 crop\_assignment = [CROPS[crop] for crop in best.genome]  
  
 print(f"{label}:")  
 print(f" Урожай = {best.total\_yield:.2f}, Стоимость = {best.total\_cost:.2f}, Фитнес = {best.fitness:.3f}")  
 print(f" Назначение культур: {crop\_assignment}")  
 print("-" \* 70)  
  
  
def main():  
 fig, ax = plt.subplots(figsize=(12, 8))  
  
 experiments = [  
 (single\_point\_crossover, random\_reset, "Single-point + Reset", "red"),  
 (two\_point\_crossover, random\_reset, "Two-point + Reset", "green"),  
 (uniform\_crossover, random\_reset, "Uniform + Reset", "blue"),  
 (single\_point\_crossover, swap\_mutation, "Single-point + Swap", "orange"),  
 (two\_point\_crossover, swap\_mutation, "Two-point + Swap", "purple"),  
 (uniform\_crossover, swap\_mutation, "Uniform + Swap", "brown"),  
 (single\_point\_crossover, inversion\_mutation, "Single-point + Inversion", "pink"),  
 (two\_point\_crossover, inversion\_mutation, "Two-point + Inversion", "cyan"),  
 (uniform\_crossover, inversion\_mutation, "Uniform + Inversion", "gray")  
 ]  
  
 for crossover, mutation, label, color in experiments:  
 run\_experiment(crossover, mutation, label, color, ax)  
  
 ax.set\_title("Эволюция функции приспособленности для различных операторов", fontsize=14)  
 ax.set\_xlabel("Поколение")  
 ax.set\_ylabel("Приспособленность ")  
 ax.legend(bbox\_to\_anchor=(1.05, 1), loc='upper left')  
 ax.grid(True, alpha=0.3)  
 plt.tight\_layout()  
 plt.show()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()