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**

**Вариант №10**

Выполнил

Студент группы ИВТАСбд-41

Ведин В.А.

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:

# №10. 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

# Задание параметров задачи

На данном этапе были инициализированы основные параметры, необходимые для решения задачи оптимизации сельскохозяйственного планирования с помощью генетического алгоритма.

Параметры задачи:

* N – определяет размер моделируемой системы, а именно количество полей (8).
* CROPS – представляет собой список сельскохозяйственных культур. Для каждой культуры задано ее название, урожайность с гектара (в тоннах) и стоимость за тонну (в долларах). Эти данные являются базовыми для последующих расчётов

Были заданы ключевые гиперпараметры, управляющие работой алгоритма:

* POP\_SIZE – размер популяции (20 особей)
* GENERATIONS – максимальное количество поколений эволюции (50)
* CROSSOVER\_RATE – вероятность скрещивания (90%)
* MUTATION\_RATE – вероятность мутации (10%)
* TOURNAMENT\_SIZE – размер турнира для селекции (3)

Для создания реалистичных исходных данных была реализована функция generate\_fields. Ее работа заключается в следующем: для каждого поля генерируется его площадь (случайное значение от 1 до 10 гектар) и поправочный коэффициент качества почвы (от 0.7 до 1.3). На основе этих данных, а также базовой урожайности и стоимости культур формируются матрицы yields\_matrix (валовый урожай для каждой культуры на каждом поле) и costs\_matrix (общая стоимость выращивания). Расчёт для каждого поля и культуры учитывает его площадь и уникальный коэффициент почвы.

Код всех параметров и функция generate\_fields представлены в листинге 1.

Листинг 1. Параметры и функция генерации данных

|  |
| --- |
| SEED = 42  random.seed(SEED)  np.random.seed(SEED)  N = 8 # field size  # crop name, yield per hectar in tons, cost per ton in dollars  CROPS = [  ("Wheat", 3.21, 173.19),  ("Corn", 6.05, 195.72),  ("Soy", 3.16, 383.52),  ("Barley", 3.10, 117.83),  ("Grapes", 22.5, 600),  ("Rice", 4.5, 450),  ("Sugar beet", 30, 550), # nerfed (yield 50 -> 30)  ("Potato", 21, 230),  ("Cassava", 10, 200)  ]  k = len(CROPS)  POP\_SIZE = 20  GENERATIONS = 50  CROSSOVER\_RATE = 0.9  MUTATION\_RATE = 0.1  TOURNAMENT\_SIZE = 3  RESULTS\_DIR = "ga\_results"  os.makedirs(RESULTS\_DIR, exist\_ok=True)  def generate\_fields(  N: int,  crops: list[tuple[str, int, int]],  seed: int = SEED  ) -> tuple[pd.DataFrame, np.ndarray, np.ndarray, list[str]]:  random.seed(seed)  np.random.seed(seed)  crop\_names = [c[0] for c in crops]  base\_yields = np.array([c[1] for c in crops])  crop\_costs = np.array([c[2] for c in crops])  areas = np.round(np.random.uniform(1.0, 10.0, size=N), 2)  soil\_factors = np.round(np.random.uniform(0.7, 1.3, size=N), 3)  yields\_matrix = np.zeros((N, len(crops)))  costs\_matrix = np.zeros((N, len(crops)))  for i in range(N):  for j in range(len(crops)):  yields\_matrix[i, j] = areas[i] \* base\_yields[j] \* soil\_factors[i]  costs\_matrix[i, j] = areas[i] \* crop\_costs[j]  fields\_df = pd.DataFrame({  "field\_id": np.arange(N),  "area": areas,  "soil\_factor": soil\_factors  })  return fields\_df, yields\_matrix, costs\_matrix, crop\_names |

# Функция приспособленности

# Для реализации функции приспособленности потребовались 2 вспомогательные функции.

# Функция total\_yield\_and\_cost вычисляет общий урожай и суммарные затраты для предложенного плана посевов (solution). Представлено это в виде массива, где каждый элемент указывает, какая культура выращивается на соответствующем поле.

# Функция compute\_theoretical\_bounds вычисляет минимально и максимально возможные значения общего урожая для затрат для всей системы полей. Эти значения используются для последующей нормаизации фактических показателей и приведения их к безразмерному виду в интервале [0, 1].

# Сама же функция приспособленности fitness\_of\_solution оценивает качество решения, комбинируя два противоречивых критерия – максимизацию урожая и минимизациюеевыавы затрат. Алгоритм ее работы следующий:

# 1. Сначала вычисляются фактический общий урожай (total\_y) и затраты (total\_c) для решения с помощью функции total\_yield\_and\_cost.

# 2. Оба показателя нормализуются относительно своих теоретических минимумов и максимумов, который вычисляются с помощью функции compute\_theoretical\_bounds, а нормализуются по формуле:

# Нормлизованная стоимость высчитывается по соответствующей формуле.

# Итоговая приспособленность рассчитывается по формуле:

# Где ny – нормализованный урожай, w – вес, регулирующий баланс между двумя целями (по умолчанию 0.5), nc – нормализованная стоимость.

# Функции описанные выше представлены в листинге 2.

# Листинг 2. Функция приспособленности

|  |
| --- |
| def generate\_fields(N: int,crops: list[tuple[str, int, int]],seed: int = SEED) -> tuple[pd.DataFrame, np.ndarray, np.ndarray, list[str]]:random.seed(seed)np.random.seed(seed)crop\_names = [c[0] for c in crops]base\_yields = np.array([c[1] for c in crops])crop\_costs = np.array([c[2] for c in crops])areas = np.round(np.random.uniform(1.0, 10.0, size=N), 2)soil\_factors = np.round(np.random.uniform(0.7, 1.3, size=N), 3)yields\_matrix = np.zeros((N, len(crops)))costs\_matrix = np.zeros((N, len(crops)))for i in range(N):for j in range(len(crops)):yields\_matrix[i, j] = areas[i] \* base\_yields[j] \* soil\_factors[i]costs\_matrix[i, j] = areas[i] \* crop\_costs[j]fields\_df = pd.DataFrame({"field\_id": np.arange(N),"area": areas,"soil\_factor": soil\_factors})return fields\_df, yields\_matrix, costs\_matrix, crop\_namesdef total\_yield\_and\_cost(solution: np.ndarray,yields\_matrix: np.ndarray,costs\_matrix: np.ndarray) -> tuple[float, float]:n = len(solution)idx = (np.arange(n), solution)total\_yield = float(yields\_matrix[idx].sum())total\_cost = float(costs\_matrix[idx].sum())return total\_yield, total\_costdef compute\_theoretical\_bounds(yields\_matrix: np.ndarray,costs\_matrix: np.ndarray) -> tuple[float, float, float, float]:min\_yield = float(yields\_matrix.min(axis=1).sum())max\_yield = float(yields\_matrix.max(axis=1).sum())min\_cost = float(costs\_matrix.min(axis=1).sum())max\_cost = float(costs\_matrix.max(axis=1).sum())return (min\_yield, max\_yield, min\_cost, max\_cost)def fitness\_of\_solution(solution: np.ndarray,yields\_matrix: np.ndarray,costs\_matrix: np.ndarray,mode: str = "weighted",w: float = 0.5,bounds: Optional[tuple[float, float, float, float]] = None) -> float:total\_y, total\_c = total\_yield\_and\_cost(solution, yields\_matrix, costs\_matrix)if bounds is None:bounds = compute\_theoretical\_bounds(yields\_matrix, costs\_matrix)min\_y, max\_y, min\_c, max\_c = boundsny = (total\_y - min\_y) / (max\_y - min\_y) if (max\_y - min\_y) > 0 else 0.5nc = (total\_c - min\_c) / (max\_c - min\_c) if (max\_c - min\_c) > 0 else 0.5return ny \* math.exp(-w \* nc) |

# Создание популяции

# Создан класс Individual, который инкапсулирует отдельное решение (особь) в популяции, каждая особь содержит:

# genome – геном решения в виде массива.

# yield\_val, cost\_val – рассчитанные значения общего урожая и затрат для данного решения.

# fiteness – значение функции приспособленности

# Формирование же популяции происходит с помощью функции create\_random\_individual, которое генерирует решение, создавая геном фиксированной длины N (количество полей), и функции create\_initial\_pop, которая создает начальную популяцию заданного размера (POP\_SIZE), последовательно вызывая функцию, упомянутую выше.

# Начальная популяцию представляет собой набор полностью случайных распределений культур по полям.

# Код класса и функций, упомянутых выше, представлен в листинге 3.

# Листинг 3. Создание популяции

|  |
| --- |
| class Individual:def \_\_init\_\_(self,genome: np.ndarray,yields\_matrix: np.ndarray,costs\_matrix: np.ndarray) -> None:self.genome = genomeself.yield\_val, self.cost\_val = total\_yield\_and\_cost(genome,yields\_matrix,costs\_matrix)self.fitness = fitness\_of\_solution(genome, yields\_matrix, costs\_matrix)def \_\_repr\_\_(self):return f"Individual(Y={self.yield\_val:.2f}, C={self.cost\_val:.2f}, F={self.fitness:.4f})"def create\_random\_individual(yields\_matrix: np.ndarray,costs\_matrix: np.ndarray) -> Individual:genome = np.random.randint(0, k, size=N)return Individual(genome, yields\_matrix, costs\_matrix)def create\_initial\_pop(pop\_size: int,yields\_matrix: np.ndarray,costs\_matrix: np.ndarray) -> list[Individual]:return [create\_random\_individual(yields\_matrix, costs\_matrix) for \_ in range(pop\_size)] |

# Селекция и эволюция

# Для выбора родительских особей применяется турнирная селекция. Из популяции случайно выбирает несколько особей (размер турнира), и среди них отбирается особь с наивысшей приспособленностью.

# Для скрещивания били реализованы три оператора кроссовера:

# Одноточечный кроссовер – выбирается случайная точка раздела, после чего, до точки раздела передаются гены 1-го родителя, после точки - 2-го.

# Двухточечный кроссовер – выбираются 2 случайные точки раздела, после чего, до 1-ой точки и после 2-ой точки берутся гены 1-го родителя, после 1-ой точки и до 2-ой точки берутся гены 2-го родителя.

# Равномерный кроссовер – каждый ген потомка случайно наследуется от одного из родителей с заданной вероятностью (обычно 50 на 50).

# Также, были реализованы три оператора мутации:

# Случайная замена – отдельные гены заменяются на случайные значения из допустимого диапазона

# Обменная мутация – случайно выбранные гены в хромосоме меняются местами

# Перемешивающая мутация – случайный сегмент хромосомы перемешивается.

# И функция, которая объединяет все вышеупомянутые – функция эволюции:

# 1. Сохраняет лучшие особи текущего поколения

# 2. Попарный отбор родителей через турнирную селекцию

# 3. Скрещивание выбранных родителей с заданной вероятностью

# 4. Применение оператора мутации к потомкам с заданной вероятностью

# 5. Формирование новой популяции из лучшей особи и полученных потомков

# Функции, упомянутые выше, представлены в листинге 4.

# Листинг 4. Отбор и эволюция

|  |
| --- |
| def one\_point\_crossover(parent1: np.ndarray,parent2: np.ndarray) -> tuple[np.ndarray, np.ndarray]:n = len(parent1)if n < 2:return parent1.copy(), parent2.copy()cp = random.randint(1, n-1)child1 = np.concatenate([parent1[:cp], parent2[cp:]])child2 = np.concatenate([parent2[:cp], parent1[cp:]])return child1, child2def two\_point\_crossover(parent1: np.ndarray,parent2: np.ndarray) -> tuple[np.ndarray, np.ndarray]:n = len(parent1)if n < 3:return one\_point\_crossover(parent1, parent2)a = random.randint(1, n-2)b = random.randint(a+1, n-1)child1 = parent1.copy()child2 = parent2.copy()child1[a:b], child2[a:b] = parent2[a:b], parent1[a:b]return child1, child2def uniform\_crossover(parent1: np.ndarray,parent2: np.ndarray,p: float = 0.5) -> tuple[np.ndarray, np.ndarray]:n = len(parent1)mask = np.random.rand(n) < pchild1 = np.where(mask, parent1, parent2)child2 = np.where(mask, parent2, parent1)return child1, child2def random\_reset\_mutation(child: list[int],k: int,mutation\_rate: float) -> list[int]:for i in range(len(child)):if random.random() < mutation\_rate:new = random.randrange(k)while new == child[i] and k > 1:new = random.randrange(k)child[i] = newreturn childdef swap\_mutation(child: list[int],k: int,mutation\_rate: float) -> list[int]:if random.random() < mutation\_rate and len(child) >= 2:i, j = random.sample(range(len(child)), 2)child[i], child[j] = child[j], child[i]return childdef scramble\_mutation(child: list[int],k: int,mutation\_rate: float) -> list[int]:n = len(child)if random.random() < mutation\_rate and n >= 3:a = random.randint(0, n-2)b = random.randint(a+1, n-1)sub = child[a:b+1].copy()np.random.shuffle(sub)child[a:b+1] = subreturn childdef tournament\_selection(pop: list[np.ndarray],fintesses: list[float],tournament\_size: int) -> np.ndarray:i\_idxs = np.random.randint(0, len(pop), size=tournament\_size)best\_idx = i\_idxs[0]best\_fit = fintesses[best\_idx]for idx in i\_idxs[1:]:if fintesses[idx] > best\_fit:best\_fit = fintesses[idx]best\_idx = idxreturn pop[best\_idx].copy()  def evolve\_pop(pop: list[Individual],yields\_matrix: np.ndarray,costs\_matrix: np.ndarray,crossover\_fn: Callable,mutation\_fn: Callable,crossover\_rate: float = CROSSOVER\_RATE,mutation\_rate: float = MUTATION\_RATE) -> list[Individual]:fitnesses = [ind.fitness for ind in pop]genomes = [ind.genome for ind in pop]new\_pop = []best\_individual = pop[np.argmax(fitnesses)]new\_pop.append(best\_individual)while len(new\_pop) < POP\_SIZE:parent1 = tournament\_selection(genomes, fitnesses, TOURNAMENT\_SIZE)parent2 = tournament\_selection(genomes, fitnesses, TOURNAMENT\_SIZE)if random.random() < crossover\_rate:child1, child2 = crossover\_fn(parent1, parent2)else:child1, child2 = parent1.copy(), parent2.copy()child1 = mutation\_fn(child1.tolist(), k, mutation\_rate)child2 = mutation\_fn(child2.tolist(), k, mutation\_rate)new\_pop.append(Individual(np.array(child1), yields\_matrix, costs\_matrix))if len(new\_pop) < POP\_SIZE:new\_pop.append(Individual(np.array(child2), yields\_matrix, costs\_matrix))return new\_pop[:POP\_SIZE] |

# Запуск эксперимента

# Функция run\_experiment выполняет один полный запуск генетического алгоритма для конкретной пары операторов скрещивания и мутации. Она создает начальную популяции, затем в цикле на протяжении заданного количества поколений применяет операторы эволюции, отслеживая историю изменения лучшей приспособленности.

# Основной эксперимент осуществляется функцией main\_experiment, которая генерирует исходные данные о полях и культурах, после чего тестирует все возможные комбинации из 3 типов скрещивания и трех типов мутации. Для каждой комбинации запускается отдельный прогон алгоритма в процессе которого выводятся промежуточные результаты и строится график динамики изменения приспособленности.

# После завершения всех экспериментов результаты агрегируются в сводную таблицу, которая включает информацию о лучшей достигнутой приспособленности, значениях урожая и затрат, а также о самом решении (распределения культур). Данные сохраняются в CSV-файл для последующего анализа. Дополнительно строится сравнительная гистограмма, наглядно демонстрирующая эффективность каждой комбинации операторов.

# Тестирование

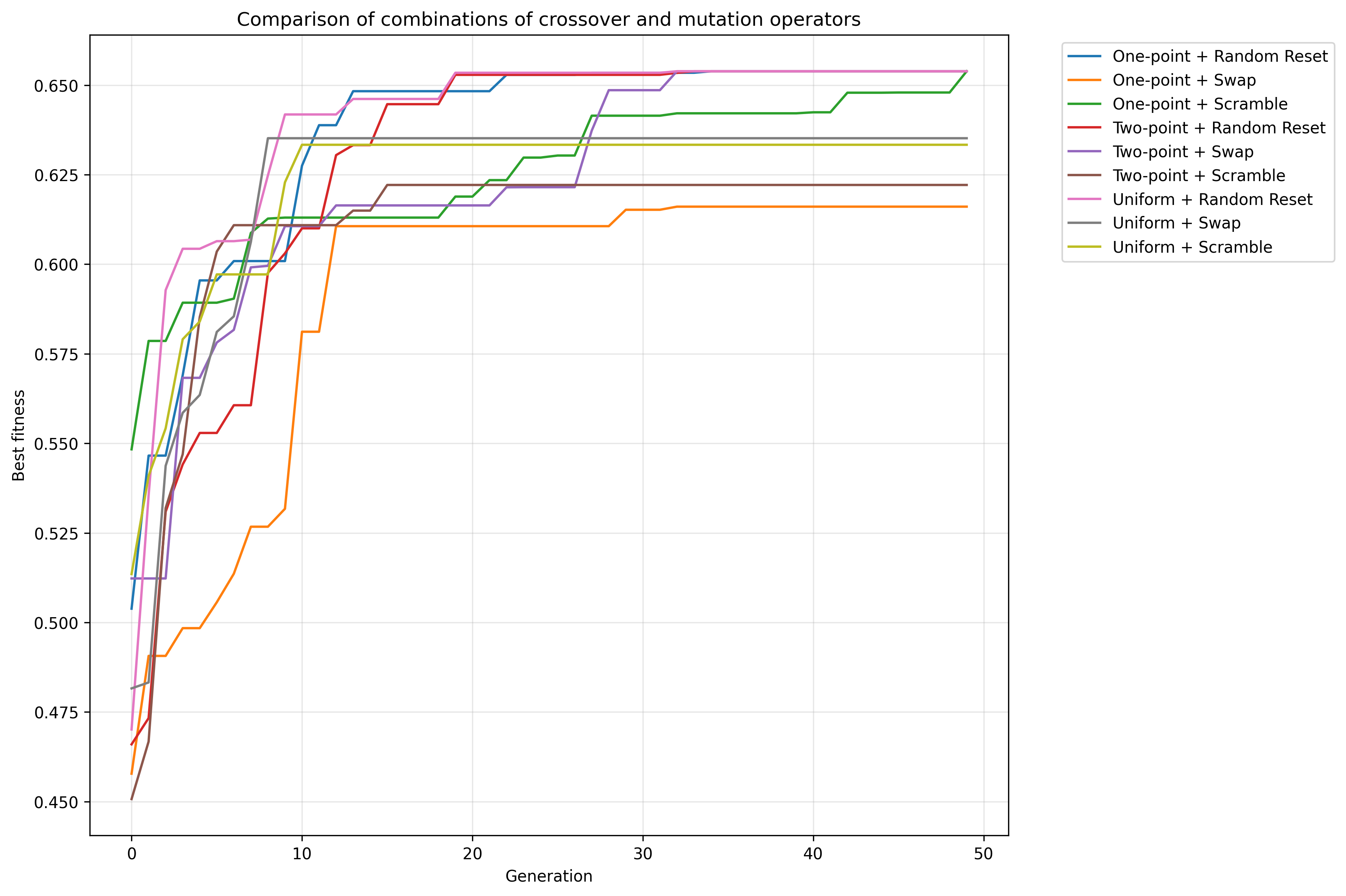


Рис. 1. График сравнения всех комбинаций всех операторов

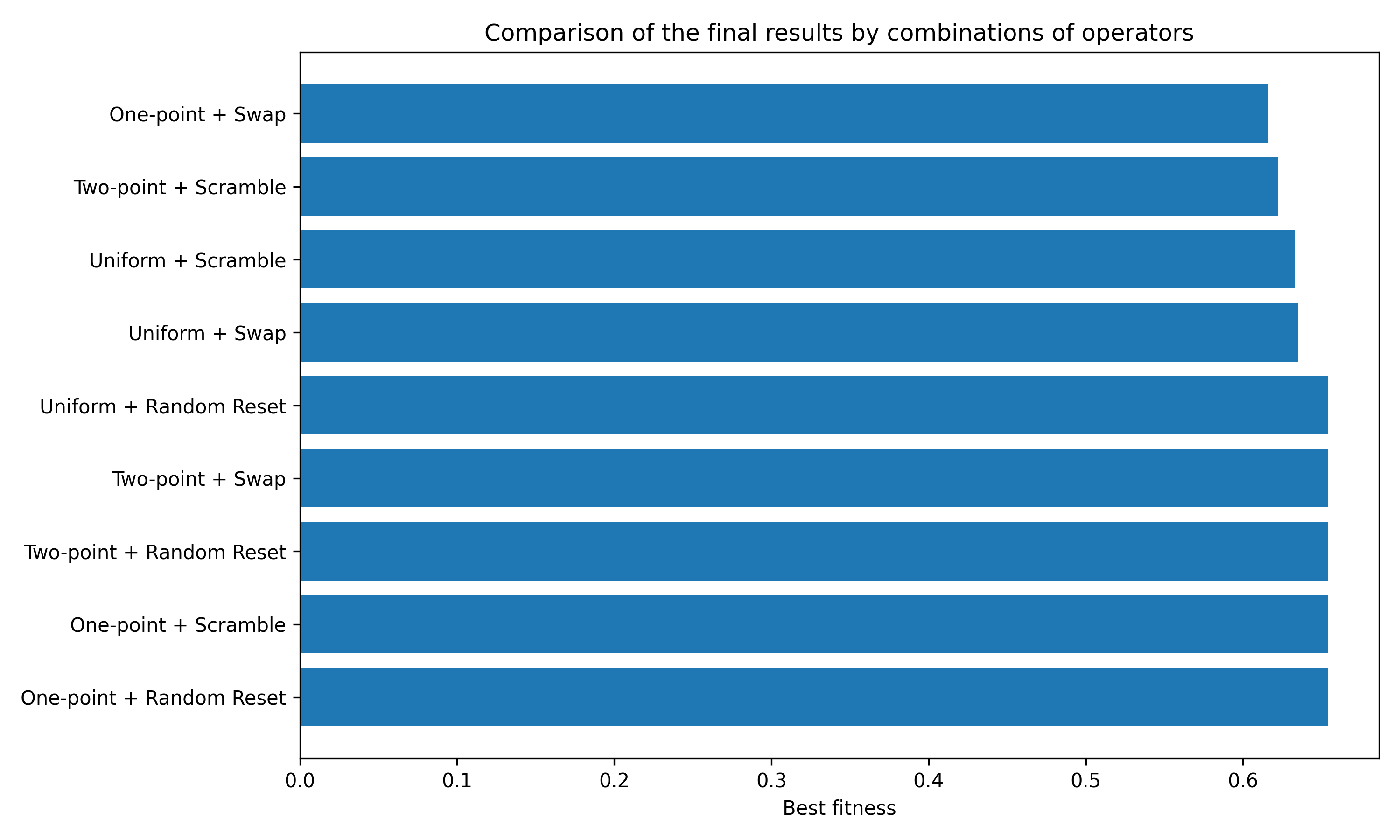


Рис. 2. Гистограмма итоговых результатов эксперимента

# 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..

# Приложение.

# Ссылка на репозиторий с кодом лабораторной работы: <https://github.com/Tsaranchik/Artifical_Intelligence_Systems/tree/master/lab1>