Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Ульяновский государственный технический университет»
Кафедра «Вычислительная техника»

Системы искусственного интеллекта

Лабораторная работа №1

«Генетические алгоритмы»

Вариант №4

Выполнил:
студент группы ИВТАСбд-41
Рубцов Денис Алексеевич
Проверил:
Хайруллин И. Д.

Ульяновск
2025

СОДЕРЖАНИЕ

Постановка задачи

Общее задание:

Необходимо разработать программу на языке python, реализующую генетический алгоритм по предложенному вариантом заданию.

Провести эксперименты по разным способам скрещивания (не менее 3-х), разным способам мутирования (не менее трех). Результат отобразить в виде графиков

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.

Предоставить отчет о проделанной работе.

Вариант 4: на языке Python разработайте скрипт, который с помощью генетического алгоритма и полного перебора решает следующую задачу. Дано n пунктов производства продуктов и k городов, которые в них нуждаются.

Каждый город может потребить x продуктов, а каждый пункт произвести y продуктов. Необходимо получить оптимальный маршрут, так, чтобы все города получили нужный им объем продуктов с минимальным его превышением, а транспортные расходы укладывались в определенные рамки.

Ход работы

Первым делом в программе заданы количество пунктов производства n, количество городов k, мощность каждого пункта supply [n], потребности каждого города demand [k], бюджет на транспортные расходы transport_budget и матрица стоимости перевозки transport_costs.

Особь представляет собой матрицу распределения товаров для каждого города.

Функция приспособленности вводит такие штрафы за неудовлетворение спроса города, за превышение поставок над потребностями, за превышение производственных мощностей, за превышение транспортного бюджета.

В программе присутствуют 3 вида мутаций: случайное перераспределение, SWAP и гауссова мутация.

Случайное перераспределение берёт случайный завод (строку матрицы), считает, сколько он всего производит сумма и полностью заново перераспределяет этот объём между городами случайным образом.

В SWAP 2 случайных элемента меняются между собой.

В гауссовой мутации к значению прибавляется случайный шум из нормального распределения, не превышающий производства завода.

Также в программе приведены 3 вида скрещивания:

Одноточечное (single-point crossover) - случайная точка разреза хромосомы, где для потомка первая часть берётся у родителя 1, вторая – у родителя 2.

Двухточечное (two-point crossover) - выбираются две точки, участок между ними берётся от одного родителя, остальное – от другого.

Однородное / равномерное (uniform crossover)-каждый ген берётся случайно у одного из родителей с вероятностью 50%

В качестве метода селекции используется турнирный отбор: каждый раз из популяции случайным образом отбирается несколько претендентов (от двух и более). Затем, среди отобранных участников выбирается наиболее

приспособленный (с наибольшим значением функции принадлежности). Он и переходит в новую выборку.

В качестве параметров генетического метода выступили:

- Размерность популяции 300.
- Количество генераций 300.
- Вероятность мутаций 50%.

Итогом программы является график зависимости приспособленности от поколения для всех возможных комбинаций видов скрещивания и мутаций

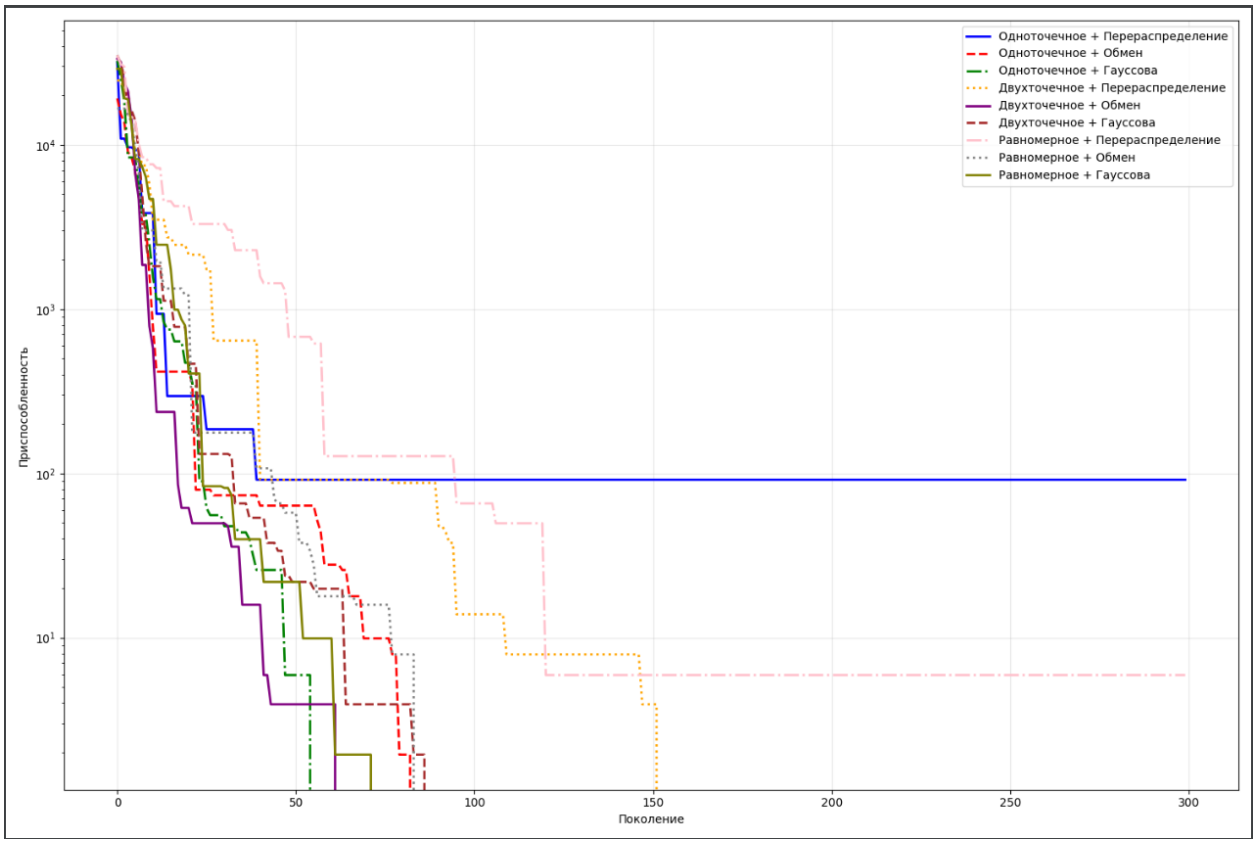

Рис. 1. Сравнение комбинаций скрещивания и мутаций в генетическом алгоритме.

Хуже всего показали себя комбинации с мутацией случайное перераспределение. Одноточечное гауссово показало лучший результат.

Заключение

В рамках выполнения лабораторной работы были приобретены умения и навыки в реализации генетических алгоритмов.

Также была написана программа, реализующая все пункты задания

Приложение А.

```
from random import randint, choices, choice, random, gauss
import numpy as np
import matplotlib.pyplot as plt

n = 4  # количество пунктов
k = 5  # количество городов
supply = [200, 300, 250, 350]  #  мощности каждого пункта
demand = [150, 200, 180, 220, 250]  # потребности каждого города
transport_budget = 32000  # бюджет на транспортные расходы

# стоимость перевозки одного продукта из пункта i в город j
transport_costs = [
    [15, 20, 25, 30, 35],
    [18, 22, 28, 32, 38],
    [12, 18, 20, 28, 32],
    [20, 25, 30, 35, 40]
]

best_fitness_history = []
current_method_name = ""

def create_individual():
    """Случайное решение (особь)"""
    individual = []
    for i in range(n):
        # Распределяем производство пункта i между городами
        max_production = supply[i]
        production = randint(0, max_production)
        distribution = []

        for j in range(k - 1):
            if production > 0:
                alloc = randint(0, production)
                distribution.append(alloc)
                production -= alloc
            else:
                distribution.append(0)

        # Последний город получает остаток
        distribution.append(max(0, production))
```

```
        individual.extend(distribution)

    return individual


def decode_individual(individual):
    """Декодирует в матрицу"""
    matrix = []
    idx = 0
    for i in range(n):
        row = []
        for j in range(k):
            row.append(individual[idx])
            idx += 1
        matrix.append(row)
    return matrix


def fitness_function(individual):
    """Функция приспособленности"""
    matrix = decode_individual(individual)

    # Рассчитываем общие транспортные расходы
    total_cost = 0
    for i in range(n):
        for j in range(k):
            total_cost += matrix[i][j] * transport_costs[i][j]

    # Рассчитываем распределение по городам
    city_supply = [0] * k
    for j in range(k):
        for i in range(n):
            city_supply[j] += matrix[i][j]

    # Рассчитываем использование производственных мощностей
    factory_usage = [0] * n
    for i in range(n):
        factory_usage[i] = sum(matrix[i])

    # Комплексная функция приспособленности
    fitness = 0
```

```
    # Штраф за неудовлетворение спроса
    for j in range(k):
        if city_supply[j] < demand[j]:
            fitness += (demand[j] - city_supply[j]) * 100

    # Штраф за превышение поставок над потребностями
    for j in range(k):
        if city_supply[j] > demand[j]:
            fitness += (city_supply[j] - demand[j]) * 2

    #Штраф за превышение производственных мощностей
    for i in range(n):
        if factory_usage[i] > supply[i]:
            fitness += (factory_usage[i] - supply[i]) * 50

    # Штраф за превышение транспортного бюджета
    if total_cost > transport_budget:
        fitness = float('inf')

    # Поощрение за эффективное использование мощностей
    for i in range(n):
        if factory_usage[i] <= supply[i] and factory_usage[i] > 0:
            fitness -= factory_usage[i] * 0.01

    return max(0, fitness)


def mutation_random_reallocate(individual):
    mutated = individual.copy()

    # случайный пункт производства для мутации
    factory_idx = randint(0, n - 1)

    # Индексы в массиве, соответствующие этому пункту
    start_idx = factory_idx * k
    end_idx = start_idx + k

    current_production = sum(mutated[start_idx:end_idx])

    # Генерируем новое распределение
    new_distribution = []
    remaining = current_production
```

```
    for j in range(k - 1):
        if remaining > 0:
            alloc = randint(0, remaining)
            new_distribution.append(alloc)
            remaining -= alloc
        else:
            new_distribution.append(0)

    new_distribution.append(max(0, remaining))

    mutated[start_idx:end_idx] = new_distribution

    return mutated


def mutation_swap(individual):
    """обмен значений между двумя случайными позициями"""
    mutated = individual.copy()

    idx1 = randint(0, len(mutated) - 1)
    idx2 = randint(0, len(mutated) - 1)


    mutated[idx1], mutated[idx2] = mutated[idx2], mutated[idx1]

    return mutated


def mutation_gaussian(individual):
    mutated = individual.copy()
    num_mutations = randint(1, 3)

    for _ in range(num_mutations):
        pos = randint(0, len(mutated) - 1)

        # Определяем, к какому производству относится эта позиция
        factory_idx = pos // k

        # Гауссово изменение с ограничением в пределах производственных мощностей
        change = int(gauss(0, supply[factory_idx] * 0.1))
        new_value = max(0, mutated[pos] + change)
```

```
            new_value = min(new_value, supply[factory_idx])

            mutated[pos] = new_value

    return mutated


def mutation(individual, method='random_reallocate'):
    if method == 'random_reallocate':
        return mutation_random_reallocate(individual)
    elif method == 'swap':
        return mutation_swap(individual)
    elif method == 'gaussian':
        return mutation_gaussian(individual)
    else:
        return mutation_random_reallocate(individual)


def crossover_one_point(parent1, parent2):
    child = []

    for i in range(n):
        start_idx = i * k
        end_idx = start_idx + k

        if random() < 0.5:
            # Берем распределение от первого родителя
            child.extend(parent1[start_idx:end_idx])
        else:
            # Берем распределение от второго родителя
            child.extend(parent2[start_idx:end_idx])

    return child


def crossover_two_point(parent1, parent2):
    child = parent1.copy()

    # Выбираем две случайные точки разрыва
    point1 = randint(1, len(parent1) - 2)
    point2 = randint(point1 + 1, len(parent1) - 1)
```

```
    # Заменяем сегмент между точками разрыва
    child[point1:point2] = parent2[point1:point2]

    return child


def crossover_uniform(parent1, parent2):
    """Равномерное скрещивание"""
    child = []

    for i in range(len(parent1)):
        # С вероятностью 50% берем ген от одного из родителей
        if random() < 0.5:
            child.append(parent1[i])
        else:
            child.append(parent2[i])

    return child


def crossover(parent1, parent2, method='one_point'):
    if method == 'one_point':
        return crossover_one_point(parent1, parent2)
    elif method == 'two_point':
        return crossover_two_point(parent1, parent2)
    elif method == 'uniform':
        return crossover_uniform(parent1, parent2)
    else:
        return crossover_one_point(parent1, parent2)


def selection(population, fitnesses):
    # Турнирный отбор
    selected = []
    tournament_size = 3

    for _ in range(len(population)):
        # Выбираем случайных особей для турнира
        tournament = choices(range(len(population)), k=tournament_size)
        # Выбираем лучшую из турнира
        winner = min(tournament, key=lambda x: fitnesses[x])
        selected.append(population[winner])
```

```
    return selected


def genetic_algorithm(population_size=100, generations=200, mutation_rate=0.1,
                      crossover_method='one_point', mutation_method='random_reallocate'):
    population = [create_individual() for _ in range(population_size)]

    best_individual = None
    best_fitness = float('inf')

    global best_fitness_history, current_method_name
    best_fitness_history = []
    current_method_name = f"{crossover_method}_{mutation_method}"

    for generation in range(generations):
        # Вычисление приспособленности
        fitnesses = [fitness_function(ind) for ind in population]

        # Обновление лучшего решения
        current_best = min(fitnesses)
        if current_best < best_fitness:
            best_fitness = current_best
            best_individual = population[fitnesses.index(current_best)]

        best_fitness_history.append(best_fitness)

        # Отбор
        selected = selection(population, fitnesses)

        # Скрещивание
        new_population = []
        for i in range(0, len(selected), 2):
            if i + 1 < len(selected):
                child1 = crossover(selected[i], selected[i + 1], crossover_method)
                child2 = crossover(selected[i + 1], selected[i], crossover_method)
                new_population.extend([child1, child2])

        # Мутация
        for i in range(len(new_population)):
            if random() < mutation_rate:
                new_population[i] = mutation(new_population[i], mutation_method)
```

```
        population = new_population

        if generation % 50 == 0:
            print(f"Поколение {generation}, лучшая приспособленность: {best_fitness}")

    return best_individual, best_fitness


def plot_fitness_progress(all_methods_data):
    colors = ['blue', 'red', 'green', 'orange', 'purple', 'brown', 'pink', 'gray', 'olive']
    line_styles = ['-', '--', '-.', ':', '-', '--', '-.', ':', '-']
    crossover_methods = ['one_point', 'two_point', 'uniform']
    mutation_methods = ['random_reallocate', 'swap', 'gaussian']
    plt.figure(figsize=(15, 10))
    color_idx = 0
    for crossover_method in crossover_methods:
        for mutation_method in mutation_methods:
            method_key = f"{crossover_method}_{mutation_method}"
            if method_key in all_methods_data:
                data = all_methods_data[method_key]
                generations = range(len(data))

                crossover_name = {
                    'one_point': 'Одноточечное',
                    'two_point': 'Двухточечное',
                    'uniform': 'Равномерное'
                }[crossover_method]

                mutation_name = {
                    'random_reallocate': 'Перераспределение',
                    'swap': 'Обмен',
                    'gaussian': 'Гауссова'
                }[mutation_method]

                label = f"{crossover_name} + {mutation_name}"

                plt.plot(generations, data,
                         color=colors[color_idx], linestyle=line_styles[color_idx],
                         linewidth=2, label=label)
                color_idx += 1
```

```
    plt.xlabel('Поколение')
    plt.ylabel('Приспособленность')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.yscale('log')
    plt.tight_layout()
    plt.savefig('combined_comparison.png', dpi=300)
    plt.show()

def brute_force():
    if n > 2 or k > 3:
        print("Слишком большая задача для полного перебора")
        return None, float('inf')

    # Генерируем все возможные распределения
    from itertools import product

    # Определяем диапазоны для каждого элемента
    ranges = []
    for i in range(n):
        for j in range(k):
            ranges.append(range(0, supply[i] + 1))

    best_solution = None
    best_fitness = float('inf')
    count = 0

    for solution in product(*ranges):
        count += 1
        fitness = fitness_function(solution)

        if fitness < best_fitness:
            best_fitness = fitness
            best_solution = solution

    print(f"Перебрано {count} решений")
    return best_solution, best_fitness



def print_solution(solution):
    if solution is None:
        print("Решение не найдено")
```

```
        return

    matrix = decode_individual(solution)

    print(f"{'Привезено':<12}", end="")
    for j in range(k):
        actual_demand = sum(matrix[i][j] for i in range(n))
        print(f"{actual_demand:<10}", end="")
    print()

    print(f"{'Требуется':<12}", end="")
    for j in range(k):
        print(f"{demand[j]:<10}", end="")
    print()

    total_cost = 0
    for i in range(n):
        for j in range(k):
            total_cost += matrix[i][j] * transport_costs[i][j]

    print(f"\nОбщие транспортные расходы: {total_cost}")
    print(f"Бюджет: {transport_budget}")
    print(f"Превышение бюджета: {max(0, total_cost - transport_budget)}")



if __name__ == "__main__":
    print(f"Пунктов производства: {n}, Городов: {k}")
    print(f"Производственные мощности: {supply}")
    print(f"Потребности городов: {demand}")
    print(f"Транспортный бюджет: {transport_budget}")


    crossover_methods = {
        'one_point': 'Одноточечное скрещивание',
        'two_point': 'Двухточечное скрещивание',
        'uniform': 'Равномерное скрещивание'
    }

    mutation_methods = {
        'random_reallocate': 'Случайное перераспределение',
        'swap': 'Обмен значениями',
```

```
    'gaussian': 'Гауссова мутация'
}

results = {}
all_methods_data = {}


for crossover_key, crossover_name in crossover_methods.items():
    for mutation_key, mutation_name in mutation_methods.items():
        method_key = f"{crossover_key}_{mutation_key}"
        method_display_name = f"{crossover_name} + {mutation_name}"

        print(f"\n" + "=" * 70)
        print(f"КОМБИНАЦИЯ: {method_display_name}")
        print("=" * 70)

        ga_solution, ga_fitness = genetic_algorithm(
            population_size=300,
            generations=300,
            mutation_rate=0.5,
            crossover_method=crossover_key,
            mutation_method=mutation_key
        )

        results[method_key] = {
            'solution': ga_solution,
            'fitness': ga_fitness,
            'name': method_display_name
        }

        # Сохраняем данные для графика
        all_methods_data[method_key] = best_fitness_history.copy()

        print(f"\nЛучшая приспособленность: {ga_fitness}")
        print_solution(ga_solution)

plot_fitness_progress(all_methods_data)

# # Полный перебор
# print("\n" + "=" * 50)
# print("ПОЛНЫЙ ПЕРЕБОР")
# print("=" * 50)
```

```
#
# bf_solution, bf_fitness = brute_force()
#
# if bf_solution is not None:
#     print(f"\nЛучшая приспособленность перебора: {bf_fitness}")
#     print_solution(bf_solution)
#
#     # Сравнение результатов
#     print("\n" + "=" * 50)
#     print("СРАВНЕНИЕ РЕЗУЛЬТАТОВ")
#     print("=" * 50)
#     print(f"Генетический алгоритм: {ga_fitness}")
#     print(f"Полный перебор: {bf_fitness}")
#     print(f"Разница: {ga_fitness - bf_fitness}")
```