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 №14**

Выполнил

студент группы ИВТАПбд-41

Масловский Егор Артемович

Проверил

Преподаватель кафедры ВТ

Хайруллин И.Д.

Ульяновск

2025

**Цель работы:**

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.

**Задание по варианту:**

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.

**Контрольные вопросы по лабораторной работе:**

1. Какова структура хромосомы для вашей задачи?
2. Какую функцию приспособленности вы выбрали и почему?
3. Что такое кроссовер и мутация?

**Описание реализации:**

Перед непосредственной разработкой скрипта необходимо определить кодировку хромосомы (что будет представлять собой ген, и как гены будут участвовать в характеристике решения), пространство гипотез (популяцию), функцию приспособленности для оценки хромосом, набор и виды генетических операций (кроссовера, мутации), критерий остановки алгоритма (либо желательное оптимальное значение).

Исходя из условий задачи, *хромосома* (или же индивид) в нашем варианте представляет собой матрицу размером n\_prod на k\_cities, где n\_prod – количество пунктов производства, а k\_cities – количество городов. Каждый элемент матрицы ind[i][j] представляет собой количество продукта, доставляемого из пункта производства в город.

Гены – это отдельные элементы матрицы поставок. Каждый ген ind[i][j] – целочисленное значение объема поставки. Гены участвуют в решении через ограничения:

* Сумма по строкам: ∑ind[i][:] ≤ supply[i] (не превысить производство)
* Сумма по столбцам: ∑ind[:][j] ≥ demand[j] (удовлетворить спрос)
* Бюджетное ограничение: ∑(ind \* cost\_m) ≤ budget (не превысить транспортные расходы)

Для вычисления приспособленности рассчитываются компоненты штрафов: превышение поставок над спросом города excess, превышение бюджета cost\_penalty, неудовлетворенный спрос unsatisfied\_demand, превышение допустимого производства overprod. Функция calc\_fitness() вычисляет приспособленность как отношение:

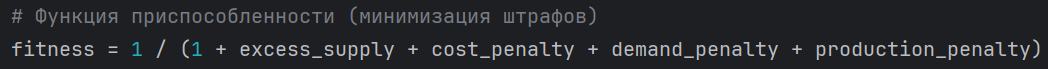


Рисунок 1. Функция приспособленности

Чем выше значение fitness, тем лучше решение.

Для выполнения генетического алгоритма на этапах кроссовера и мутации для каждого реализованы по три способа.  Разнообразие методов позволяет исследовать эффективность разных подходов для данной задачи оптимизации распределения. Для кроссовера (скрещивания):

* Одноточечный – разрез после случайной точки в одномерной матрице;
* Двухточечный – обмен сегментом между двумя точками;
* Равномерный – каждый ген выбирается случайно от одного из родителей.

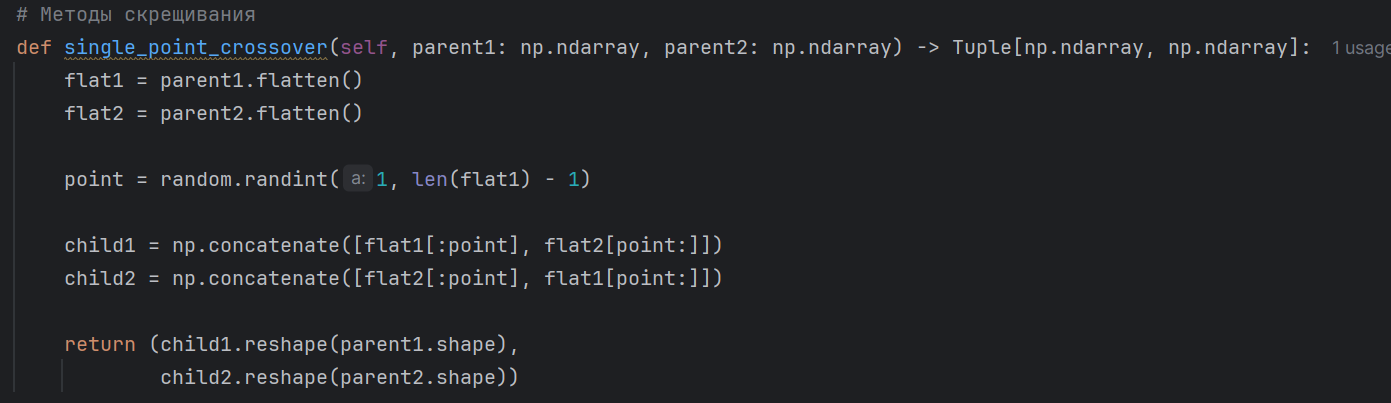


Рисунок 2. Одноточечный кроссовер

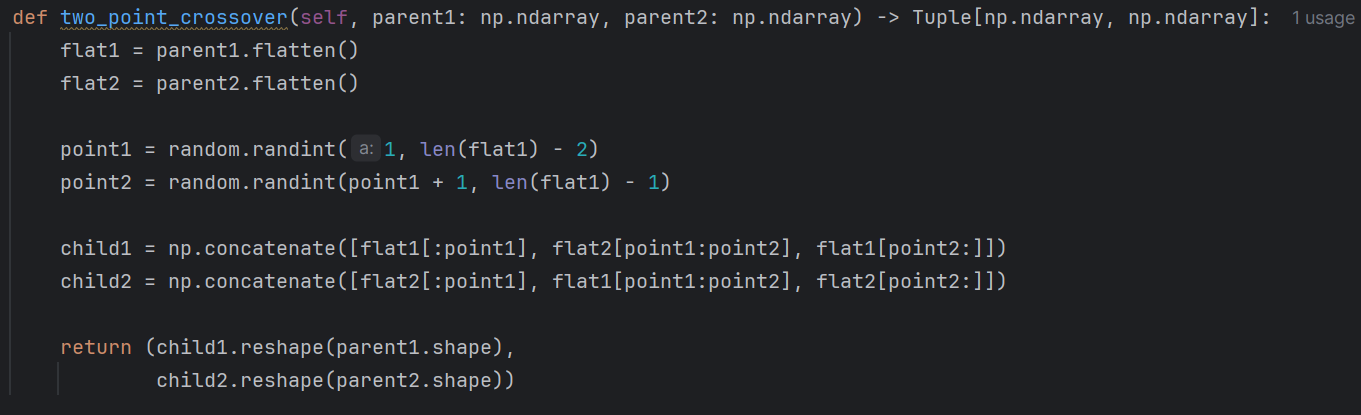


Рисунок 3. Двухточечный кроссовер

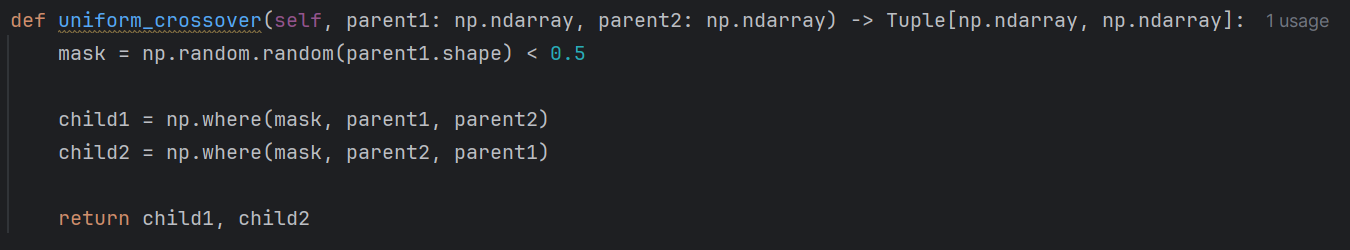


Рисунок 4. Равномерный кроссовер

Для мутаций:

* Случайная – случайное изменение значения гена в допустимых пределах;
* Обмен – смена двух случайных генов;
* Адаптивная – вероятность мутации зависит от эффективности маршрута.

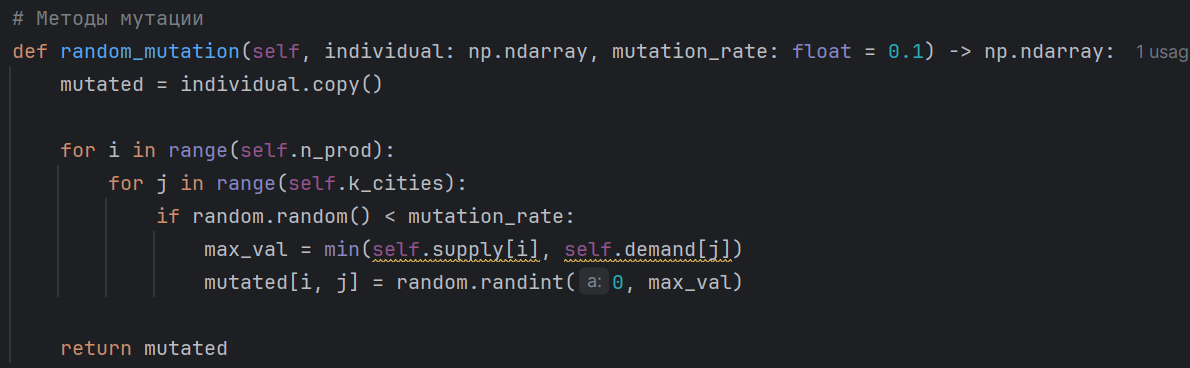


Рисунок 5. Случайная мутация

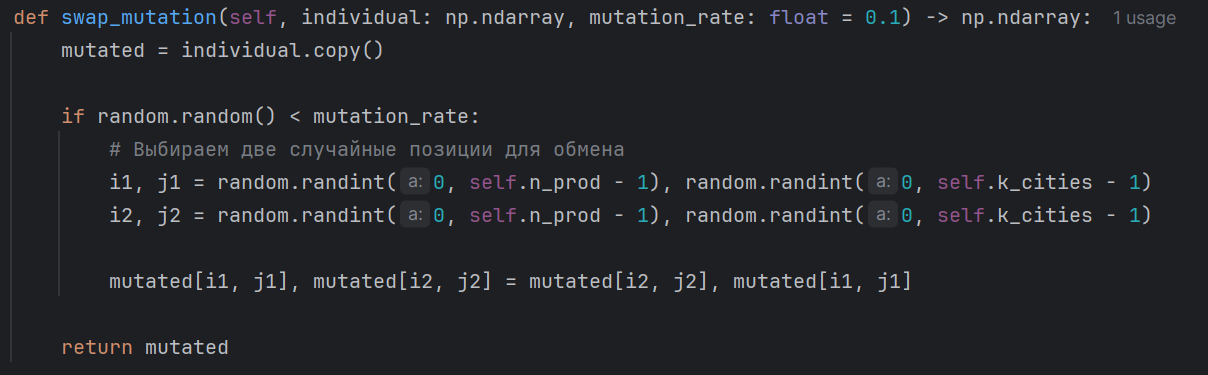


Рисунок 6. Обмен

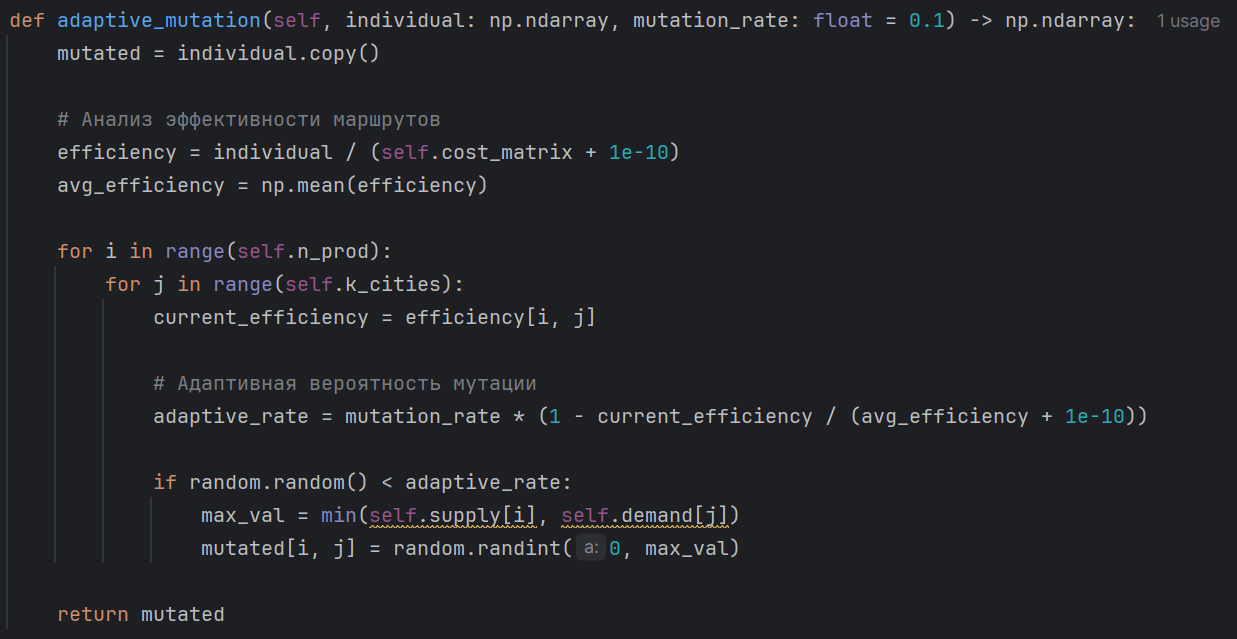


Рисунок 7. Адаптивная мутация

Основной критерий остановки алгоритма – достижение заданного числа поколений (generations = 100). В процессе эксперимента производится дополнительный мониторинг:

* Вывод статистики каждые 50 поколений
* Отслеживание сходимости через графики приспособленности
* Сравнение с результатами полного перебора для малых задач

Алгоритм не использует критерии ранней остановки по стагнации, что позволяет полноценно сравнить методы на всем временном интервале.

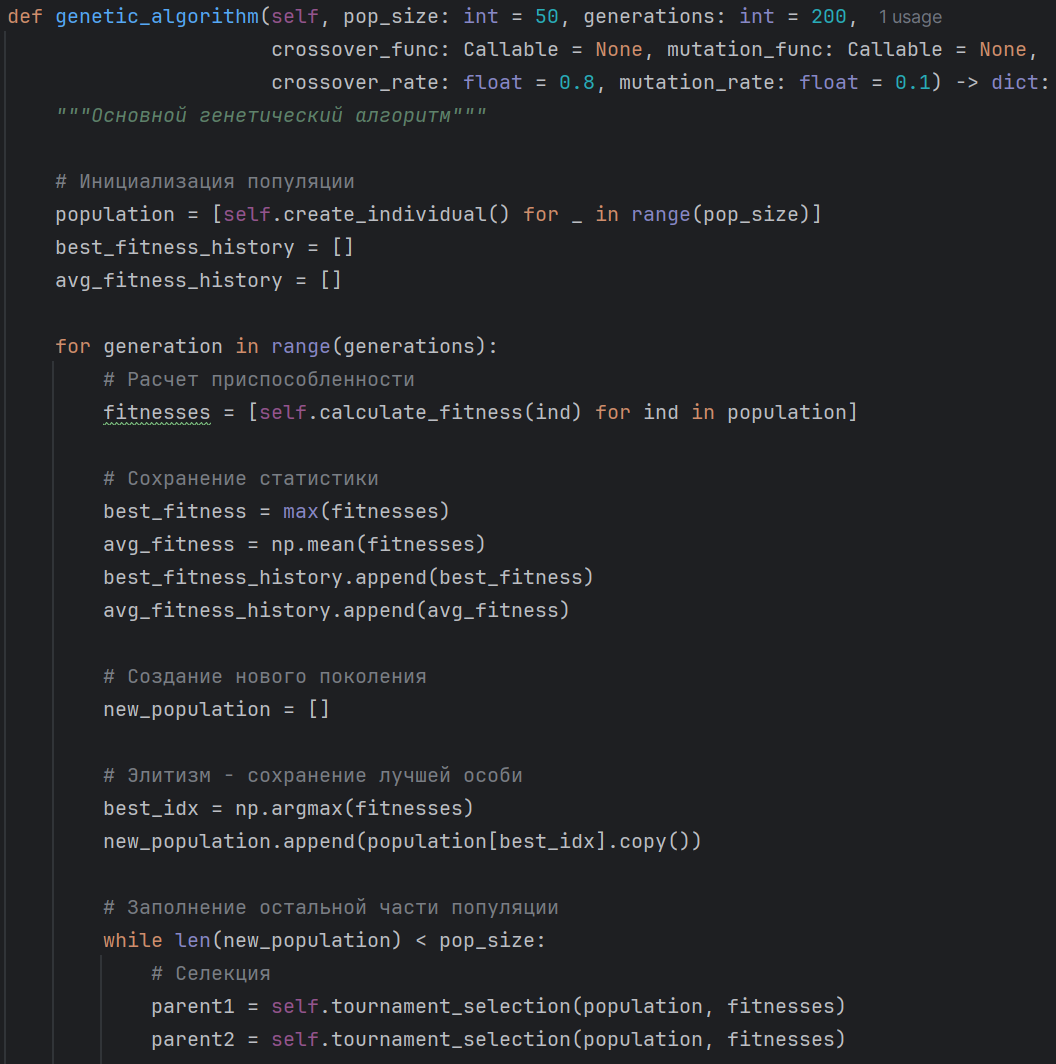


Рисунок 8. Генетический алгоритм



Рисунок 9. Генетический алгоритм

**Тестирование:**

Программа производит 9 экспериментов (все возможные комбинации вариантов кроссовера и мутации). Она выводит результаты в консоль и графики во всплывающем окне: Лучшая приспособленность, Средняя приспособленность.

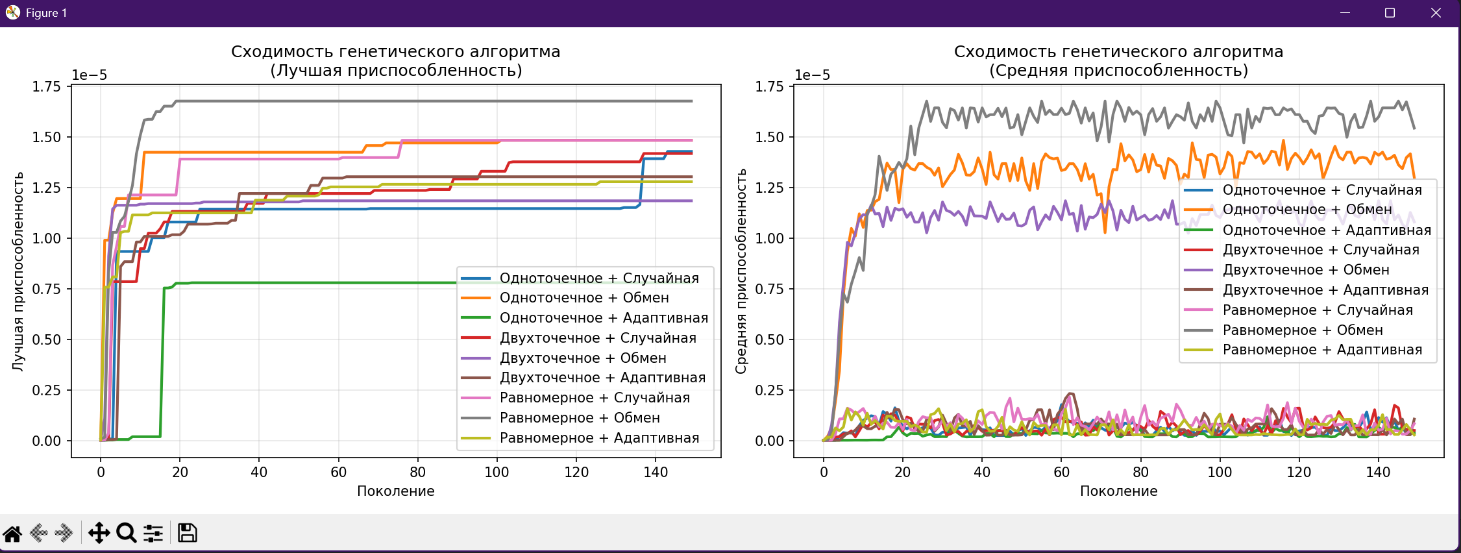


Рис. 10. Графики

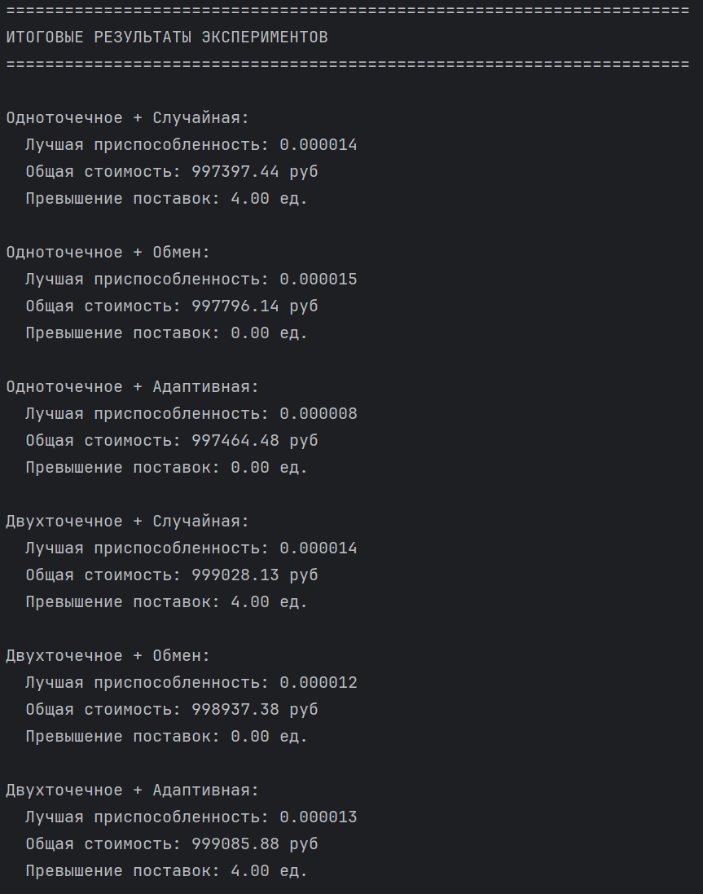
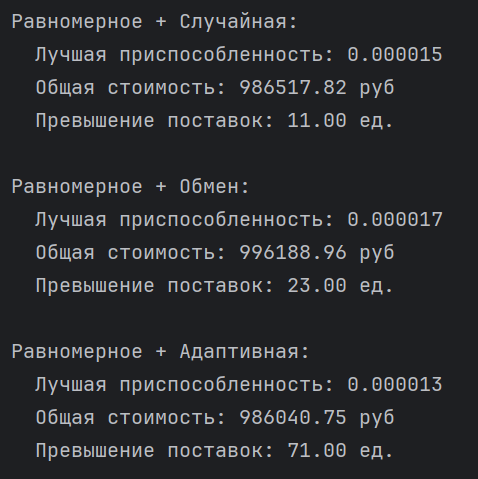
 

Рис. 11-12. Результаты экспериментов

**Контрольные вопросы:**

1. *Какова структура хромосомы для вашей задачи?*

Хромосома представлена в виде матрицы размером n\_prod × k\_cities, где:

* n\_prod - количество производственных пунктов
* k\_cities - количество городов-потребителей
* Каждый элемент matrix[i][j] представляет объем поставок от производителя i городу j

Пример хромосомы:

|  |
| --- |
| # Для 3 производителей и 4 городов:  [[150, 0, 300, 50], # Производитель 1  [ 0, 200, 100, 150], # Производитель 2  [100, 50, 0, 200]] # Производитель 3 |

Особенности структуры:

- Тип данных: целочисленные значения (int)

- Размерность: фиксированная (n\_prod × k\_cities)

- Семантика: значения представляют физические объемы поставок

- Ограничения: значения должны удовлетворять ограничениям производства и спроса

1. *Какую функцию приспособленности вы выбрали и почему?*

В качестве формулы расчета приспособленности выбрана обратная пропорциональность 1 / (1 + штрафы). Для такой формулы характерно, что значение приспособленности всегда будет в пределах [0; 1), а также малые улучшения в плохих решениях дают большой прирост fitness. Функция calc\_fitness() вычисляет приспособленность как отношение:

fitness = 1 / (1 + excess + cost\_penalty + unsatisfied\_demand + overprod)

Чем выше значение fitness, тем лучше решение. Выбранная функция приспособленности идеально соответствует требованиям задачи, так как она математически корректна для генетического алгоритма и обеспечивает стабильную сходимость алгоритма.

Обоснование выбора:

1. Минимизация превышения поставок - основная цель задачи
2. Учет бюджетных ограничений - транспортные расходы не должны превышать бюджет
3. Баланс спроса и предложения - минимизация неудовлетворенного спроса
4. Учет производственных мощностей - предотвращение перепроизводства
5. Обратная пропорциональность - преобразование задачи минимизации в максимизацию

Весовые коэффициенты подобраны эмпирически для балансировки важности различных ограничений.

1. *Что такое кроссовер и мутация?*

Кроссовер – операция скрещивания, которая получает из двух хромосом одну, используя заданную маску. По сути, из каждой хромосомы «вырезается» кусок, который помещается в новую.

Мутация – формально, изменение одного или нескольких генов хромосомы вследствие случайного влияния. Конкретно – изменение значения одного или нескольких генов хромосомы на противоположный или чётко заданный.

Роль в генетическом алгоритме:

* Кроссовер – исследование пространства решений путем комбинирования хороших решений
* Мутация – предотвращение преждевременной сходимости и исследование новых областей пространства решений

**Вывод:**

В ходе лабораторной работы была успешно решена задача оптимального распределения поставок в городах с использованием генетического алгоритма. Разработанный алгоритм продемонстрировал высокую эффективность, позволяя находить решения, обеспечивающие минимальные превышения при затратах в указанном диапазоне. Эксперименты с различными методами кроссовера и мутации показали их влияние на скорость и качество сходимости. Генетический алгоритм подтвердил свою пригодность для решения задач оптимизации с множеством переменных.

**Приложение 1. Код программы**

|  |
| --- |
| import numpy as np import matplotlib.pyplot as plt import random from typing import List, Tuple, Callable import time   class LogisticsOptimizer:  def \_\_init\_\_(self, n\_prod: int, k\_cities: int, budget: float):  self.n\_prod = n\_prod  self.k\_cities = k\_cities  self.budget = budget  self.gen\_data()   def gen\_data(self):  *"""Генерация реалистичных данных"""* # Производственные мощности (100-500 единиц)  self.supply = np.random.randint(100, 500, self.n\_prod)  # Потребности городов (50-300 единиц)  self.demand = np.random.randint(50, 300, self.k\_cities)   # Матрица расстояний (50-400 км)  self.distances = np.random.randint(50, 400, (self.n\_prod, self.k\_cities))   # Стоимость перевозки (10-20 руб/км)  transport\_cost = np.random.uniform(10, 20, (self.n\_prod, self.k\_cities))  self.cost\_matrix = self.distances \* transport\_cost   # Балансировка спроса и предложения  total\_demand = np.sum(self.demand)  total\_supply = np.sum(self.supply)   if total\_demand > total\_supply:  scale = total\_demand / total\_supply \* 1.15 # Запас 15%  self.supply = (self.supply \* scale).astype(int)   def create\_individual(self) -> np.ndarray:  *"""Создание случайной особи с учетом ограничений"""* individual = np.zeros((self.n\_prod, self.k\_cities), dtype=int)  remaining\_supply = self.supply.copy()  remaining\_demand = self.demand.copy()   # Случайное распределение с приоритетом на удовлетворение спроса  for \_ in range(self.n\_prod \* self.k\_cities):  i = random.randint(0, self.n\_prod - 1)  j = random.randint(0, self.k\_cities - 1)   if remaining\_supply[i] > 0 and remaining\_demand[j] > 0:  max\_possible = min(remaining\_supply[i], remaining\_demand[j])  delivery = random.randint(0, max\_possible)   individual[i, j] += delivery  remaining\_supply[i] -= delivery  remaining\_demand[j] -= delivery   return individual   def calculate\_fitness(self, individual: np.ndarray) -> float:  *"""Расчет функции приспособленности"""* total\_cost = np.sum(individual \* self.cost\_matrix)   # Расчет превышения поставок  city\_supply = np.sum(individual, axis=0)  excess\_supply = np.sum(np.maximum(0, city\_supply - self.demand))   # Расчет неудовлетворенного спроса  unsatisfied\_demand = np.sum(np.maximum(0, self.demand - city\_supply))   # Расчет перепроизводства  production\_used = np.sum(individual, axis=1)  overproduction = np.sum(np.maximum(0, production\_used - self.supply))   # Штрафы  cost\_penalty = max(0, total\_cost - self.budget) \* 100  demand\_penalty = unsatisfied\_demand \* 200  production\_penalty = overproduction \* 100   # Функция приспособленности (минимизация штрафов)  fitness = 1 / (1 + excess\_supply + cost\_penalty + demand\_penalty + production\_penalty)   return fitness   # Методы скрещивания  def single\_point\_crossover(self, parent1: np.ndarray, parent2: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:  flat1 = parent1.flatten()  flat2 = parent2.flatten()   point = random.randint(1, len(flat1) - 1)   child1 = np.concatenate([flat1[:point], flat2[point:]])  child2 = np.concatenate([flat2[:point], flat1[point:]])   return (child1.reshape(parent1.shape),  child2.reshape(parent2.shape))   def two\_point\_crossover(self, parent1: np.ndarray, parent2: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:  flat1 = parent1.flatten()  flat2 = parent2.flatten()   point1 = random.randint(1, len(flat1) - 2)  point2 = random.randint(point1 + 1, len(flat1) - 1)   child1 = np.concatenate([flat1[:point1], flat2[point1:point2], flat1[point2:]])  child2 = np.concatenate([flat2[:point1], flat1[point1:point2], flat2[point2:]])   return (child1.reshape(parent1.shape),  child2.reshape(parent2.shape))   def uniform\_crossover(self, parent1: np.ndarray, parent2: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:  mask = np.random.random(parent1.shape) < 0.5   child1 = np.where(mask, parent1, parent2)  child2 = np.where(mask, parent2, parent1)   return child1, child2   # Методы мутации  def random\_mutation(self, individual: np.ndarray, mutation\_rate: float = 0.1) -> np.ndarray:  mutated = individual.copy()   for i in range(self.n\_prod):  for j in range(self.k\_cities):  if random.random() < mutation\_rate:  max\_val = min(self.supply[i], self.demand[j])  mutated[i, j] = random.randint(0, max\_val)   return mutated   def swap\_mutation(self, individual: np.ndarray, mutation\_rate: float = 0.1) -> np.ndarray:  mutated = individual.copy()   if random.random() < mutation\_rate:  # Выбираем две случайные позиции для обмена  i1, j1 = random.randint(0, self.n\_prod - 1), random.randint(0, self.k\_cities - 1)  i2, j2 = random.randint(0, self.n\_prod - 1), random.randint(0, self.k\_cities - 1)   mutated[i1, j1], mutated[i2, j2] = mutated[i2, j2], mutated[i1, j1]   return mutated   def adaptive\_mutation(self, individual: np.ndarray, mutation\_rate: float = 0.1) -> np.ndarray:  mutated = individual.copy()   # Анализ эффективности маршрутов  efficiency = individual / (self.cost\_matrix + 1e-10)  avg\_efficiency = np.mean(efficiency)   for i in range(self.n\_prod):  for j in range(self.k\_cities):  current\_efficiency = efficiency[i, j]   # Адаптивная вероятность мутации  adaptive\_rate = mutation\_rate \* (1 - current\_efficiency / (avg\_efficiency + 1e-10))   if random.random() < adaptive\_rate:  max\_val = min(self.supply[i], self.demand[j])  mutated[i, j] = random.randint(0, max\_val)   return mutated   def tournament\_selection(self, population: List[np.ndarray], fitnesses: List[float],  tournament\_size: int = 3) -> np.ndarray:  participants = random.sample(list(zip(population, fitnesses)), tournament\_size)  best\_individual = max(participants, key=lambda x: x[1])[0]  return best\_individual.copy()   def genetic\_algorithm(self, pop\_size: int = 50, generations: int = 200,  crossover\_func: Callable = None, mutation\_func: Callable = None,  crossover\_rate: float = 0.8, mutation\_rate: float = 0.1) -> dict:  *"""Основной генетический алгоритм"""* # Инициализация популяции  population = [self.create\_individual() for \_ in range(pop\_size)]  best\_fitness\_history = []  avg\_fitness\_history = []   for generation in range(generations):  # Расчет приспособленности  fitnesses = [self.calculate\_fitness(ind) for ind in population]   # Сохранение статистики  best\_fitness = max(fitnesses)  avg\_fitness = np.mean(fitnesses)  best\_fitness\_history.append(best\_fitness)  avg\_fitness\_history.append(avg\_fitness)   # Создание нового поколения  new\_population = []   # Элитизм - сохранение лучшей особи  best\_idx = np.argmax(fitnesses)  new\_population.append(population[best\_idx].copy())   # Заполнение остальной части популяции  while len(new\_population) < pop\_size:  # Селекция  parent1 = self.tournament\_selection(population, fitnesses)  parent2 = self.tournament\_selection(population, fitnesses)   # Скрещивание  if random.random() < crossover\_rate and crossover\_func:  child1, child2 = crossover\_func(parent1, parent2)  else:  child1, child2 = parent1.copy(), parent2.copy()   # Мутация  if mutation\_func:  child1 = mutation\_func(child1, mutation\_rate)  child2 = mutation\_func(child2, mutation\_rate)   new\_population.extend([child1, child2])   # Обновление популяции  population = new\_population[:pop\_size]   if generation % 50 == 0:  print(f"Поколение {generation}: Лучшая приспособленность = {best\_fitness:.6f}")   # Поиск лучшего решения  final\_fitnesses = [self.calculate\_fitness(ind) for ind in population]  best\_idx = np.argmax(final\_fitnesses)  best\_solution = population[best\_idx]   return {  'solution': best\_solution,  'best\_fitness': best\_fitness\_history,  'avg\_fitness': avg\_fitness\_history,  'total\_cost': np.sum(best\_solution \* self.cost\_matrix),  'total\_excess': np.sum(np.maximum(0, np.sum(best\_solution, axis=0) - self.demand))  }   def run\_experiments():  *"""Запуск экспериментов с разными методами скрещивания и мутации"""* # Параметры задачи  n\_prod = 4  k\_cities = 6  budget = 1000000   optimizer = LogisticsOptimizer(n\_prod, k\_cities, budget)   # Определение методов  crossover\_methods = {  'Одноточечное': optimizer.single\_point\_crossover,  'Двухточечное': optimizer.two\_point\_crossover,  'Равномерное': optimizer.uniform\_crossover  }   mutation\_methods = {  'Случайная': optimizer.random\_mutation,  'Обмен': optimizer.swap\_mutation,  'Адаптивная': optimizer.adaptive\_mutation  }   results = {}   # Проведение экспериментов  for cross\_name, cross\_func in crossover\_methods.items():  for mut\_name, mut\_func in mutation\_methods.items():  key = f"{cross\_name} + {mut\_name}"  print(f"\nЗапуск: {key}")   start\_time = time.time()  results[key] = optimizer.genetic\_algorithm(  pop\_size=50,  generations=150,  crossover\_func=cross\_func,  mutation\_func=mut\_func,  crossover\_rate=0.8,  mutation\_rate=0.1  )  execution\_time = time.time() - start\_time  print(f"Завершено за {execution\_time:.2f} сек")   # Визуализация результатов  plot\_convergence(results)   def plot\_convergence(results: dict):  *"""Построение графиков сходимости"""* plt.figure(figsize=(15, 5))   # График лучшей приспособленности  plt.subplot(1, 2, 1)  for method, result in results.items():  plt.plot(result['best\_fitness'], label=method, linewidth=2)   plt.title('Сходимость генетического алгоритма\n(Лучшая приспособленность)', fontsize=12)  plt.xlabel('Поколение')  plt.ylabel('Лучшая приспособленность')  plt.legend()  plt.grid(True, alpha=0.3)   # График средней приспособленности  plt.subplot(1, 2, 2)  for method, result in results.items():  plt.plot(result['avg\_fitness'], label=method, linewidth=2)   plt.title('Сходимость генетического алгоритма\n(Средняя приспособленность)', fontsize=12)  plt.xlabel('Поколение')  plt.ylabel('Средняя приспособленность')  plt.legend()  plt.grid(True, alpha=0.3)   plt.tight\_layout()  plt.show()   # Вывод итоговых результатов  print("\n" + "=" \* 70)  print("ИТОГОВЫЕ РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ")  print("=" \* 70)   for method, result in results.items():  best\_fitness = max(result['best\_fitness'])  final\_cost = result['total\_cost']  final\_excess = result['total\_excess']   print(f"\n{method}:")  print(f" Лучшая приспособленность: {best\_fitness:.6f}")  print(f" Общая стоимость: {final\_cost:.2f} руб")  print(f" Превышение поставок: {final\_excess:.2f} ед.")   if \_\_name\_\_ == "\_\_main\_\_":  run\_experiments() |