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 гр. ИВТАСбд-41

Павлов А.В.

**Проверил работу:**

Хайруллин И. Д.

Ульяновск,

2025

**Общее 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.

**Задача. Вариант №19:**

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.

**Теоретическая подготовка**

**Что такое генетический алгоритм?**

Генетический алгоритм (ГА) — это метод оптимизации, основанный на принципах естественного отбора и эволюции. Вместо того чтобы перебирать все возможные варианты, он **генерирует множество решений (популяцию)** и постепенно улучшает их за счёт **отбора, скрещивания и мутации**, пока не найдёт достаточно хорошее решение.

В моём коде работа ГА выглядит так:

1. **Создание популяции**
   * Алгоритм начинает с набора случайных решений.
   * Каждое решение представлено в виде матрицы N × K, где строки — это склады (пункты производства), а столбцы — города. Элементы матрицы показывают, сколько товара со склада отправляется в город.
2. **Оценка приспособленности (fitness)**
   * Каждое решение проверяется функцией fitness().
   * Она считает **транспортные расходы**, а также штрафы за **недопоставку** и **перепоставку**.
   * Чем ниже итоговые затраты, тем выше значение приспособленности.
   * Для удобства используется формула:

Fitness=1/(1+(Транспорт+Штрафы))

Таким образом задача минимизации стоимости превращается в задачу максимизации fitness.

1. **Эволюция поколений**
   * На каждом шаге алгоритм повторяет цикл:
     + **Отбор (selection)**: выбираются лучшие решения. В коде используется *турнирный отбор* — несколько случайных решений сравниваются, и в новое поколение проходит лучшее.
     + **Скрещивание (crossover)**: пары родителей обмениваются частями матриц и создают потомков.
     + **Мутация (mutation)**: потомки слегка изменяются случайным образом, чтобы сохранить разнообразие и не застрять в локальном минимуме.
2. **Результат**
   * После нескольких поколений (GENERATIONS) сохраняется лучшее найденное решение.
   * Это решение показывает оптимальное распределение поставок между складами и городами с учётом транспортных расходов и штрафов.

Почему именно ГА?  
Потому что задача относится к NP-трудным — полный перебор возможных матриц требует астрономического времени. ГА даёт возможность найти хорошее решение за разумное время, жертвуя точностью в пользу скорости.

**Скрещивание (Crossover)**

Скрещивание объединяет «гены» двух родителей, создавая новые решения. Его цель — взять лучшие свойства у разных особей.

В моём коде реализовано три вида:

* **single\_point** — матрицы разрезаются по одной строке, и нижняя часть обменивается между родителями.
* **two\_point** — разрез идёт по двум строкам, и средняя часть матриц меняется местами.
* **uniform** — для каждой строки случайно выбирается, у какого родителя её взять.

**Мутации (Mutation)**

Мутация вносит случайные изменения в матрицу, чтобы алгоритм мог исследовать новые варианты и не застревал на одном решении.

В моём коде есть три варианта:

* **single\_point** — случайная ячейка матрицы увеличивается или уменьшается на произвольное значение.
* **reset** — случайная ячейка полностью заменяется случайным числом (с последующей корректировкой по запасам).
* **redistribute** — вся строка перераспределяется заново, то есть склад случайным образом отправляет товары по городам.

**Полный перебор (brute force)**

Для маленьких размеров задачи (3×3) в коде реализован полный перебор. Он проверяет все возможные комбинации и гарантированно находит оптимум. Это используется для сравнения и проверки правильности работы ГА.

**Типы скрещивания (crossover)**

В моём коде реализованы три оператора кроссовера:

1. **Одноточечное скрещивание (single\_point\_crossover)**
   * **Как работает:** В коде выбирается случайная строка в матрице поставок. Все строки выше берутся от первого родителя, а все строки ниже — от второго. Таким образом формируются два потомка.
   * **Плюсы:** Простая реализация, сохраняет целые блоки распределений (все поставки со склада), что помогает передавать хорошие решения.
   * **Минусы:** Если лучшие значения разбросаны по всей матрице, то они могут потеряться.
2. **Двухточечное скрещивание (two\_point\_crossover)**
   * **Как работает:** В коде выбираются две случайные строки. Средняя часть матрицы меняется местами между родителями. Потомки формируются комбинацией блоков сверху, из середины и снизу.
   * **Плюсы:** Позволяет обмениваться сразу несколькими блоками, создавая больше разнообразия.
   * **Минусы:** Всё ещё ограничено блочными структурами (обмениваются только целые строки).
3. **Равномерное скрещивание (uniform\_crossover)**
   * **Как работает:** Для каждой строки (то есть для одного склада) случайно подбрасывается "монетка": берём всю строку от первого или второго родителя.
   * **Плюсы:** Дает самое глубокое смешивание, позволяет потомкам получить разные комбинации складов.
   * **Минусы:** Может разрушать полезные наборы строк (например, удачное распределение нескольких складов вместе).

**Типы мутаций (mutation)**

В коде реализовано три способа мутации:

1. **Одноточечная мутация (single\_point\_mutation)**
   * **Как работает:** Случайная ячейка матрицы выбирается, и к её значению прибавляется или вычитается случайное число. Потом матрица корректируется, чтобы не превысить производственные мощности.
   * **Плюсы:** Мягкие изменения, которые тонко настраивают решение.
   * **Минусы:** Может быть слишком "слабой", чтобы серьёзно изменить ситуацию.
2. **Мутация сброса (reset\_mutation)**
   * **Как работает:** Случайная ячейка полностью заменяется новым случайным значением (с учётом ограничений).
   * **Плюсы:** Вносит сильное разнообразие, помогает выйти из тупика.
   * **Минусы:** Может резко ухудшить текущее решение.
3. **Мутация перераспределения (redistribute\_mutation)**
   * **Как работает:** Выбирается одна строка (один склад). Все его поставки сбрасываются, и товары распределяются заново случайным образом между городами, но так, чтобы не превысить мощности склада.
   * **Плюсы:** Вносит большие изменения, но при этом учитывает одно из главных ограничений (supply). Это делает мутацию «умной».
   * **Минусы:** Вычислительно тяжелее, чем простая точечная мутация.

**Отбор (selection)**

В моём коде используется **турнирный отбор**:

* **Как работает:**
  1. Выбирается несколько случайных решений из популяции.
  2. Среди них выбирается то, у которого fitness выше (т.е. суммарные расходы и штрафы меньше).
  3. Этот процесс повторяется, пока не будет набрано нужное количество родителей для кроссовера.
* **Почему именно он:**
  1. Турнирный отбор прост в реализации.
  2. Он даёт шанс не только «лидерам», но и средним особям, что сохраняет разнообразие популяции.
  3. В коде это критично: если бы мы всегда брали только лучших, популяция быстро «схлопнулась» бы, и алгоритм застрял бы в локальном минимуме.

## Описание набора данных

В моём коде набор данных строится вокруг двух ключевых элементов: **пункты производства** и **города**.

* **Производственные пункты (N=5):**  
  Каждый пункт имеет:
  + координаты (x, y) на плоскости,
  + максимальную производственную мощность (**supply**) — объём продуктов, который склад может отгрузить.
* **Города (K=8):**  
  Каждый город описывается:
  + координатами (x, y),
  + фиксированным спросом (**demand**) — количество продуктов, которое город должен получить.

Координаты позволяют вычислить **расстояния между складами и городами**. Эти расстояния используются для расчёта **транспортных расходов**.

Дополнительно набор данных включает:

* price\_per\_unit\_distance — стоимость перевозки одной единицы продукта на единицу расстояния,
* penalty\_per\_shortage\_unit — штраф за недопоставку,
* penalty\_per\_excess\_unit — штраф за излишек.

Причём штраф за **недопоставку выше**, чем за излишек. Это вынуждает алгоритм в первую очередь удовлетворять спрос всех городов, а уже потом оптимизировать маршруты и издержки.

## Описание реализации

Код реализует два подхода:

1. **Генетический алгоритм (ГА)** — основной инструмент поиска приближённого решения.
2. **Полный перебор (brute force)** — используется только для маленьких примеров, чтобы проверить правильность работы ГА.

Программа разделена на несколько логических блоков:

### 1. Инициализация данных

* Задаются параметры задачи: количество складов и городов, их координаты, объёмы производства и потребности.
* Рассчитывается матрица расстояний между всеми парами «пункт–город».
* Фиксируются экономические параметры: стоимость перевозки, штрафы.  
  Эти данные формируют основу, на которой работает алгоритм.

### 2. Функция приспособленности

Ключевая функция — **calculate\_fitness()**.  
Она принимает матрицу потоков (распределение поставок) и вычисляет:

* **стоимость перевозки**: сумма произведений объёма поставки × расстояние × стоимость,
* **штрафы**:
  + за недопоставку (если город получил меньше спроса),
  + за излишек (если город получил больше),
  + за превышение мощностей склада (если отгружено больше supply).

Далее итоговая стоимость преобразуется в fitness по формуле:

Fitness=11+(Транспортныерасходы+Штрафы)Fitness = \frac{1}{1 + (Транспортные расходы + Штрафы)}Fitness=1+(Транспортныерасходы+Штрафы)1​

Так задача минимизации стоимости превращается в задачу максимизации fitness.

### 3. Операторы генетического алгоритма

Алгоритм использует стандартные «эволюционные шаги»:

* **Создание популяции (create\_initial\_population)** — генерируется первое поколение случайных решений.
* **Отбор (selection)** — применяется турнирный отбор: из случайной группы решений выбирается то, у которого fitness выше.
* **Скрещивание (crossover):**
  + single\_point\_crossover — обмен блоками строк после одной точки разреза,
  + two\_point\_crossover — обмен средними блоками после двух точек разреза,
  + uniform\_crossover — для каждой строки случайно выбирается родитель.
* **Мутации (mutation):**
  + single\_point\_mutation — небольшое изменение случайной ячейки,
  + reset\_mutation — случайная ячейка полностью заменяется новым значением,
  + redistribute\_mutation — вся строка перераспределяется заново.

Таким образом, код реализует полный цикл работы генетического алгоритма: от создания популяции до постепенного улучшения решений через десятки поколений.

**4. Основной цикл ГА**

Функция **genetic\_algorithm** объединяет все операторы и реализует полный цикл эволюции.  
Алгоритм работает в течение заданного числа поколений (**GENERATIONS**) и на каждом шаге выполняет:

1. **Оценка приспособленности**  
   Для каждой особи в популяции рассчитывается fitness с учётом транспортных расходов и штрафов.
2. **Выбор родителей**  
   Используется турнирный отбор: из случайной подгруппы особей выбирается лучший кандидат.
3. **Скрещивание и мутация**  
   Для выбранных родителей применяются выбранные операторы скрещивания (одноточечное, двуточечное или равномерное) и мутации (одноточечная, сброс или перераспределение).
4. **Формирование новой популяции**  
   Потомки заменяют старую популяцию. На каждом поколении отслеживается и сохраняется лучшее решение.

В конце цикла функция возвращает:

* лучшую найденную матрицу поставок,
* её стоимость и штрафы,
* динамику изменения fitness по поколениям.

**5. Полный перебор**

Функция **brute\_force\_small** показывает, насколько сложно решать задачу полным перебором.

* Она решает ту же задачу распределения, но для **сильно уменьшенного примера** (например, 3 склада, 3 города и маленькие мощности).
* Перебираются **все возможные комбинации** потоков поставок, чтобы найти абсолютно оптимальное решение.

Это наглядно демонстрирует, что для маленьких задач полный перебор возможен, но для реальных размеров (5×8 матрица и большие мощности) — вычислительная сложность становится экспоненциальной, и такой подход становится практически невозможным.

**6. Запуск и анализ результатов**

В последнем блоке программы запускаются **все комбинации операторов** (3 вида скрещивания × 3 вида мутации). Для каждой комбинации:

* Измеряется время работы,
* Сохраняется лучшее решение,
* Фиксируется динамика улучшения fitness по поколениям.

Результаты анализируются следующим образом:

* На экран выводится таблица с итоговыми значениями: транспортные расходы, штрафы, общая стоимость, время работы.
* Лучшие найденные решения сохраняются в отдельные файлы.
* Строится **график изменения fitness по поколениям** для всех комбинаций. Этот график позволяет сравнить, какие комбинации операторов дают более быстрый рост качества решений.

Также проводится сравнение с **полным перебором** (на маленьком примере). Результаты показывают, что генетический алгоритм способен находить хорошие решения значительно быстрее и на задачах реального размера, где перебор невозможен.

## Общие вопросы по заданию

### Какова структура хромосомы для вашей задачи?

Хромосома в моей задаче — это **матрица распределения потоков** размером **N × K**, где:

* **N = 5** — количество пунктов производства (склады/фабрики).
* **K = 8** — количество городов.

Каждый элемент матрицы individual[i, j] — это **целое число**, показывающее, сколько продукта отправляется из пункта производства **i** в город **j**.

Таким образом, вся матрица описывает **одно решение** задачи: какой склад сколько продукции отправляет в каждый город.  
Это и есть **особь (chromosome)** в терминах генетического алгоритма.

### Какую функцию приспособленности вы выбрали и почему?

В моей задаче я использую функцию:

fitness=11+(Общая стоимость доставки+Штрафы)fitness = \frac{1}{1 + (\text{Общая стоимость доставки} + \text{Штрафы})}fitness=1+(Общая стоимость доставки+Штрафы)1​

Где штрафы учитывают:

* **Недопоставку** (shortage penalty) — если город получил меньше, чем требуется.
* **Перепоставку** (excess penalty) — если город получил больше, чем нужно.
* **Превышение производства** — если склад отгружает больше, чем может. В этом случае решение считается абсолютно непригодным и получает очень большой штраф.

**Почему именно так:**

* Генетические алгоритмы по природе **максимизируют приспособленность**. Но в моей задаче требуется **минимизировать стоимость**. Использование обратной пропорции (1/(1+cost)1 / (1 + cost)1/(1+cost)) позволяет превратить задачу минимизации в задачу максимизации.
* Чем ниже стоимость (включая штрафы), тем выше значение fitness.
* Если решение нарушает ограничения, оно получает близкий к нулю fitness и исключается из эволюции.

Это делает функцию удобной, понятной и хорошо работающей для отбора лучших решений.

### Что такое кроссовер и мутация?

#### Кроссовер (скрещивание)

Кроссовер — это процесс, при котором два родителя (матрицы распределений) **обмениваются частями генов**, чтобы создать новых потомков. Это позволяет объединять удачные свойства разных решений.

В моём коде реализовано три типа кроссовера:

1. **Одноточечный** (single-point)
   * Матрицы разрезаются в одной случайной строке.
   * Первая часть от одного родителя соединяется со второй частью другого.
2. **Двухточечный** (two-point)
   * Матрицы разрезаются в двух случайных строках.
   * Средняя часть одной матрицы меняется местами с соответствующей частью другой.
3. **Равномерный** (uniform)
   * Для каждой строки случайно выбирается родитель, от которого она наследуется.
   * Таким образом, потомок может состоять из строк, взятых "вперемешку" от обоих родителей.

#### Мутация

Мутация — это процесс **внесения случайных изменений** в потомка. Она поддерживает разнообразие популяции и помогает алгоритму исследовать новые варианты решений.

В моём коде реализовано три типа мутации:

1. **Одноточечная мутация** (single\_point)
   * Выбирается случайная ячейка матрицы, и её значение немного изменяется (увеличивается или уменьшается).
2. **Мутация сброса** (reset)
   * Выбранная ячейка получает новое случайное значение в пределах допустимого лимита.
3. **Мутация перераспределения** (redistribute)
   * Для одного пункта производства (строки матрицы) все потоки сбрасываются и заново распределяются между городами случайным образом.
   * При этом строго соблюдается ограничение на общий объём производства.