МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ВОЛГОГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

На правах рукописи

Д.И. Крыжановский, П.Д.Кравченя,
А.Е. Андреев, Д.В. Завьялов, М.А. Кузнецов

Алгоритмы и структуры данных в системах искусственного интеллекта

Учебное пособие

Волгоград

2021

УДК 004.021

Крыжановский Д.И.

Алгоритмы и структуры данных в системах искусственного интеллекта: учеб. пособие / Д.И. Крыжановский, П.Д. Кравченя, А.Е. Андреев, Д.В. Завьялов, М.А. Кузнецов, ВолгГТУ. – Волгоград, 2021.–79 с.

В учебном пособии рассмотрены методические материалы по курсу «Алгоритмы и структуры данных в системах искусственного интеллекта».

Учебное пособие предназначено для магистров, обучающихся по программам магистратуры по профилю «искусственный интеллект» по направлениям 09.04.01 «Информатика и вычислительная техника», 09.04.03 «Прикладная информатика», 09.04.02 «Информационные системы и технологии». Учебное пособие выполнено в рамках реализации гранта на разработку программ бакалавриата и программ магистратуры по профилю «Искусственный интеллект», а также на повышение квалификации педагогических работников образовательных организаций высшего образования в сфере искусственного интеллекта (конкурс 2021-ИИ-01 от 10.06.2021).

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Выполнение лабораторных работ позволит студентам на практических примерах познакомится с базовыми алгоритмами и структурами данных, которые используется во множестве систем. В том числе и системах, основанных на использовании искусственного интеллекта.

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Лабораторная работа № 1. Изучение алгоритмов сортировок и базовых структур данных

Цели работы:

1. Изучить базовые структуры данных и алгоритмы сортировки.
2. Познакомиться с анализом алгоритмов и способами оценки их производительности и затрат памяти.
3. Получить навыки реализации некоторых базовых структур данных и алгоритмов сортировки.

Теоретическое введение

Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти. В обоих случаях сложность зависит от размеров входных данных: массив из 100 элементов будет обработан быстрее, чем аналогичный из 1000. При этом точное время мало кого интересует: оно зависит от процессора, типа данных, языка программирования и множества других параметров. Важна лишь асимптотическая сложность, т. е. сложность при стремлении размера входных данных к бесконечности.

Допустим, некоторому алгоритму нужно выполнить $4n^3 + 7n$ условных операций, чтобы обработать *n* элементов входных данных. При увеличении *n* на итоговое время работы будет значительно больше влиять возведение *n* в куб, чем умножение его на 4 или же прибавление 7n. Тогда говорят, что временная сложность этого алгоритма равна $O(n^3)$, т. е. она зависит от размера входных данных кубически.

Использование заглавной буквы O (или так называемая O-нотация) пришло из математики, где её применяют для сравнения асимптотического поведения функций. Формально O(f(n)) означает, что

время работы алгоритма (или объём занимаемой памяти) растёт в зависимости от объёма входных данных не быстрее, чем некоторая константа, умноженная на f(n).

Примеры различных классов асимптотических сложностей:

O(n) – **линейная сложность**. Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем *n* элементам массива, чтобы понять, какой из них максимальный.

O(log n) – **логарифмическая сложность**. Простейший пример – бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива – там его точно нет. Если же меньше, то наоборот – отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log *n* элементов.

$O(n^2)$ — **квадратичная сложность.** Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n * n, т. е. n^2.

Бывают и другие оценки по сложности, но все они основаны на том же принципе.

Также случается, что время работы алгоритма вообще не зависит от размера входных данных. Тогда сложность обозначают как O(1). Например, для определения значения третьего элемента массива не нужно ни запоминать элементы, ни проходить по ним сколько-то раз. Всегда нужно просто дождаться в потоке входных данных третий

элемент и это будет результатом, на вычисление которого для любого количества данных нужно одно и то же время.

Аналогично проводят оценку и по памяти, когда это важно. Однако, алгоритмы могут использовать значительно больше памяти при увеличении размера входных данных, чем другие, но зато работать быстрее. И наоборот. Это помогает выбирать оптимальные пути решения задач исходя из текущих условий и требований.

К основным простым структурам данных относятся:

1. Массив (Array).
2. Связный список (Linked List).
3. Стек (Stack).
4. Очередь (Queue).

Массив – это структура данных с фиксированным и упорядоченным набором однотипных элементов (компонентов). Доступ к какому-либо из элементов массива осуществляется по имени и номеру (индексу) этого элемента. Количество индексов определяет размерность массива (см. рисунок 1). Так, например, чаще всего встречаются одномерные (вектора) и двумерные (матрицы) массивы. Первые имеют один индекс, вторые – два.

Рисунок 1 – Иллюстрация массива

Для получения доступа к i-ому элементу массива потребуется указать название массива и номер требуемого элемента: A[i]. Когда A – матрица, то она представляема в виде таблицы, доступ к элементам которой

осуществляется по имени массива, а также номерам строки и столбца, на пересечении которых расположен элемент: A[i, j], где i – номер строки, j – номер столбца.

Массивы, описанного типа называются статическими, но существуют также массивы по определенным признакам отличные от них: динамические. Динамичность массивов характеризуется непостоянностью размера, т. е. по мере выполнения программы размер динамического массива может изменяться. Такая функция делает работу с данными более гибкой, но при этом приходится жертвовать быстродействием, да и сам процесс усложняется.

Список (связный список) – это структура данных, представляющая собой конечное множество упорядоченных элементов, связанных друг с другом посредствам указателей. Каждый элемент структуры содержит поле с какой-либо информацией, а также указатель на следующий элемент. В отличие от массива, к элементам списка нет произвольного доступа.

В односвязном списке, приведенном на рисунке 2, начальным элементом является Head list (голова списка), а все остальное называется хвостом. Хвост списка составляют элементы, разделенные на две части: информационную (поле info) и указательную (поле next). В последнем элементе вместо указателя, содержится признак конца списка – null.

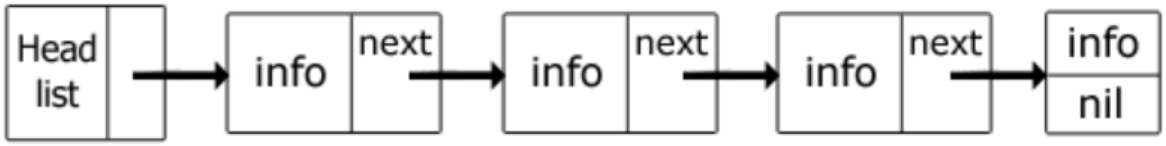


Рисунок 2 – Иллюстрация связного списка

Односвязный список не слишком удобен, т. к. из одной точки есть возможность попасть лишь в следующую точку, двигаясь тем самым в конец. Когда кроме указателя на следующий элемент есть указатель и на предыдущий, то такой список называется двусвязным. Возможность

двигаться как вперед, так и назад полезна для выполнения некоторых операций, но дополнительные указатели требуют задействования большего количества памяти, чем таковой необходимо в эквивалентном односвязном списке.

Стек характерен тем, что получить доступ к его элементам можно лишь с одного конца, называемого вершиной стека, иначе говоря: стек – структура данных, функционирующая по принципу LIFO (last in — first out, «последним пришёл — первым вышел»).

Изобразить эту структуру данных лучше в виде вертикального списка, например, стопки каких-либо вещей, где чтобы воспользоваться одной из них нужно поднять все те вещи, что лежат выше нее, а положить предмет можно лишь на вверх стопки (рисунок 3).

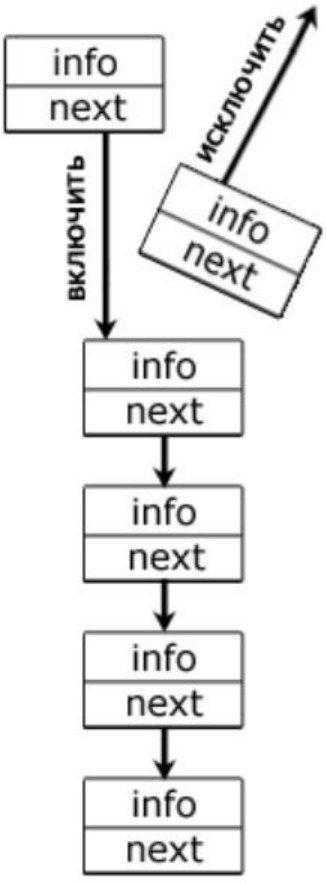


Рисунок 3 – Иллюстрация структуры данных – стека

В показанной структуре данных операции над элементами происходят

строго с одного конца: для включения нужного элемента в пятую по счету ячейку необходимо исключить тот элемент, который занимает эту позицию. Если бы было, например 6 элементов, а вставить конкретный элемент требовалось также в пятую ячейку, то исключить бы пришлось уже два элемента.

Структура данных «*Очередь*» использует принцип организации FIFO (First In, First Out — «первым пришёл – первым вышел»). В некотором смысле, такой метод более справедлив, чем тот, по которому функционирует стек, ведь простое правило, лежащее в основе привычных очередей в различные магазины, больницы считается вполне справедливым, а именно оно является базисом этой структуры. Строго говоря, очередь – это список, добавление элементов в который допустимо лишь в его конец, а их извлечение производится с другой стороны, называемой началом списка (рисунок 4).

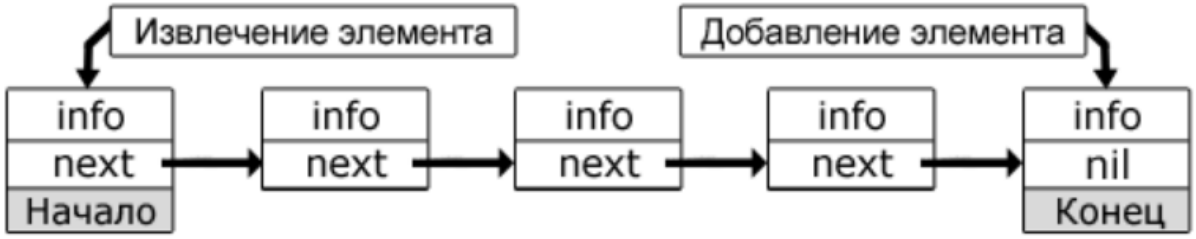


Рисунок 4 – Иллюстрация структуры данных – очереди

Алгоритм сортировки – это алгоритм для упорядочивания элементов в массиве. В случае, когда элемент в массиве имеет несколько полей, поле, служащее критерием порядка, называется *ключом сортировки*. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

Сортировка пузырьком — один из самых известных алгоритмов сортировки. Здесь нужно последовательно сравнивать значения соседних элементов и менять числа местами, если предыдущее оказывается больше последующего. Таким образом, элементы с большими значениями

оказываются в конце списка, а с меньшими остаются в начале. Реализация алгоритма сортировки приведена ниже:

```
def bubble_sort(mas):
    for i in range(len(mas)):
        # Выталкивание очередного элемента наверх
        for j in range (1, len(mas) - i):
            if mas[j-1] > mas[j]:
                mas[j-1], mas[j] = mas[j], mas[j-1]
```

Шейкерная сортировка отличается от пузырьковой тем, что она двунаправленная: алгоритм перемещается не строго слева направо, а сначала слева направо, затем справа налево:

```
def shaker_sort(mas):
    # lb, ub границы неотсортированной части массива
    k = ub = len(mas)-1
    lb = 0
    while ( lb < ub ):
        # Выталкивание очередного элемента вниз
        for j in range (ub, lb, -1):
            if mas[j-1] > mas[j]:
                mas[j-1], mas[j] = mas[j], mas[j-1]
                k = j
            lb = k
        # Выталкивание очередного элемента наверх
        for j in range (lb, ub+1):
            if mas[j-1] > mas[j]:
                mas[j-1], mas[j] = mas[j], mas[j-1]
                k = j
            ub = k
```

При *сортировке вставками* массив постепенно перебирается слева направо. При этом каждый последующий элемент размещается так, чтобы он оказался между ближайшими элементами с минимальным и максимальным значением:

```
def insert_sort(mas):
```

```
    for i in range(1, len(mas)):
        # Берем первый элемент из неотсортированной части
        x = mas[i]
        j = i - 1
        # Ищем место вставки x в сортированную часть
        while j > -1 and x < mas[j]:
            mas[j + 1], mas[j] = mas[j], x
            j -= 1
```

В *сортировке выбором* сначала нужно рассмотреть подмножество массива и найти в нём максимум (или минимум). Затем выбранное значение меняют местами со значением первого неотсортированного элемента. Этот шаг нужно повторять до тех пор, пока в массиве не закончатся неотсортированные подмассивы.

```
def choice_sort(mas):
    for i in range(len(mas)):
        # Поиск индекса наименьшего элемента в неотсортированной части
        min_index = i
        for k in range(i, len(mas)):
            if mas[k] < mas[min_index]:
                min_index = k
        # Меняем местами текущий элемент и минимальный из неотсортированной части
        mas[min_index], mas[i] = mas[i], mas[min_index]
```

Быстрая сортировка состоит из трёх шагов. Сначала из массива нужно выбрать один элемент – его обычно называют опорным. Затем другие элементы в массиве перераспределяют так, чтобы элементы меньше опорного оказались до него, а большие или равные – после. А дальше рекурсивно применяют первые два шага к подмассивам справа и слева от опорного значения.

```
def quick_sort(mas, low, high):
    if low < high:
```

```
            q = partition(mas, low, high)
            quick_sort(mas, low, q)
            quick_sort(mas, q+1, high)
```

Сортировка слиянием пригодится для таких структур данных, в которых доступ к элементам осуществляется последовательно (например, для потоков). Здесь массив разбивается на две примерно равные части и каждая из них сортируется по отдельности. Затем два отсортированных подмассива сливаются в один.

```
    def merge(mas, l, q, r):
        # Копируем подмассивы во вспомогательные массивы
        n1 = q - l + 1
        n2 = r - q
        left = mas[l:q+1]
        right = mas[q+1:r+1]
        # Добавляем сигнальные элементы к вспомогательным
массивам
        left.append(sys.maxint)
        right.append(sys.maxint)
        i = 0
        j = 0
        # Цикл слияния вспомогательных массивов в исходный
подмассив в правильном порядке
        for k in range(l, r+1):
            if left[i] <= right[j]:
                mas[k] = left[i]
                i += 1
            else:
                mas[k] = right[j]
                j += 1

    def merge_sort(mas, l, r):
        if l < r:
            q = (l+r)/2
```

```
        merge_sort(mas, l, q)
        merge_sort(mas, q+1, r)
        merge(mas, l, q, r)
```

При этой сортировке сначала строится пирамида из элементов исходного массива. Пирамида (или двоичная куча) – это способ представления элементов, при котором от каждого узла может отходить не больше двух ответвлений. А значение в родительском узле должно быть больше значений в его двух дочерних узлах.

```
    # Вспомогательная функция для получения индекса левого сына узла с индексом i в массиве mas, который пирамидально упорядочен
    def left(mas, i):
        return 2*i+1
    # Вспомогательная функция для получения индекса правого сына узла с индексом i в массиве mas, который пирамидально упорядочен
    def right(mas, i):
        return 2*i+2
    # Вспомогательная функция для упорядочения пирамиды начиная с узла i
    # Максимальный индекс элемента в пирамиде heap_size
    def max_heapify(mas, i, heap_size):
        l = left(mas, i)
        r = right(mas, i)
        if l < heap_size and mas[l] > mas[i]:
            largest = l
        else:
            largest = i
        if r < heap_size and mas[r] > mas[largest]:
            largest = r
        if largest != i:
            mas[largest], mas[i] = mas[i], mas[largest]
            max_heapify(mas, largest, heap_size)
```

```
# Вспомогательная функция построения неубывающей пирамиды
def build_max_heap(mas):
    heap_size = len(mas)
    for i in range(len(mas)/2, -1, -1):
        max_heapify(mas, i, heap_size)

def heap_sort(mas):
    build_max_heap(mas)
    heap_size = len(mas)
    for i in range(len(mas)-1, 0, -1):
        mas[0], mas[i] = mas[i], mas[0]
        heap_size -= 1
        max_heapify(mas, 0, heap_size)
```

Постановка задачи

<u>При решении заданий запрещено использовать сторонние библиотеки и STL!</u>

1. Реализуйте процедуру на языке C++, позволяющую сгенерировать массив из *N* элементов типа `double`, элементы которого равномерно распределены от –1000,0 до 1000,0.

2. Реализуйте процедуру на языке C++, позволяющую сгенерировать почти полностью отсортированный массив (на 95%) из *N* элементов типа `double` со значениями от –1000,0 до 1000,0.

3. Реализуйте две процедуры на языке C++, каждая из которых осуществляет сортировку массива с использованием алгоритма в соответствии с индивидуальным заданием по вариантам.

4. Проведите исследования зависимости времени работы алгоритмов сортировок на созданных массивах от их размера *N*. Результаты занесите в таблицу 1.

5. На одном графике постройте четыре кривые, соответствующие полученным результатам. При необходимости, используйте

полулогарифмическую шкалу.

6. Сделайте выводы по результатам исследований. Коды разработанных программ, таблицу 1 и построенные по ней графики занесите в протокол лабораторной работы.

Варианты индивидуальных заданий

Номер варианта	Первый тип сортировки	Второй тип сортировки
1	Сортировка выбором	Radix-сортировка
2	Сортировка вставками	Быстрая сортировка
3	Сортировка пузырьком	Сортировка Шелла
4	Шейкер-сортировка	Сортировка слиянием
5	Сортировка вставками	Пирамидальная сортировка
6	Шейкер-сортировка	Radix-сортировка
7	Сортировка пузырьком	Быстрая сортировка
8	Сортировка выбором	Сортировка Шелла
9	Сортировка вставками	Сортировка слиянием
10	Сортировка пузырьком	Пирамидальная сортировка

Таблица 1 — Результаты исследований времени работы алгоритмов сортировки от размерности массива

N	Время работы сортировки **первого** типа, с		Время работы сортировки **второго** типа, с	
	Неотсортированный массив	Частично отсортированный массив	Неотсортированный массив	Частично отсортированный массив
1000				
2000				
4000				
8000				
…				
10^6				

Вопросы к отчету лабораторной работы

1. Понятие алгоритма, свойства алгоритмов. Понятие RAM-машины и положения, положенные в основу ее функционирования.

2. Асимптотический анализ алгоритмов. Определение O, **Ω**, **Θ**-нотаций и их смысл.

3. Амортизационный анализ. Понятие и назначение. Примеры использования.

4. Статический и динамический массивы как структуры данных. Размещение в памяти, основные операции над массивами, их асимптотический анализ по времени работы и занимаемой памяти.

5. Односвязный и двусвязный списки как структуры данных. Размещение в памяти, основные операции над списками, их асимптотический анализ по времени работы и занимаемой памяти.

6. Стек, очередь и двусторонняя очередь как структуры данных. Основные операции над ними, их асимптотический анализ по времени работы и занимаемой памяти. Стек и рекурсия.

7. Двоичная куча как структура данных. Представление двоичной кучи как массива. Основные операции над кучей, их асимптотический анализ по времени работы и занимаемой памяти. Процедура `heapify`. Очередь с приоритетом.

8. Сортировка выбором: алгоритм работы и его асимптотический анализ.

9. Сортировка вставками: алгоритм работы и его асимптотический анализ.

10. Сортировка пузырьком и шейкер-сортировка: алгоритмы работы и их асимптотический анализ.

11. Сортировка Шелла: алгоритм работы и его асимптотический анализ.

12. Пирамидальная сортировка: алгоритм работы и его асимптотический анализ.

13. Быстрая сортировка: алгоритм работы и его асимптотический анализ. Понятие опорного элемента и его влияние на процедуру сортировки.

14. Radix-сортировка: алгоритм работы и его асимптотический анализ.

15. Сортировка слиянием: алгоритм работы и его асимптотический анализ.

16. Понятие устойчивости сортировок. Примеры устойчивых сортировок.

Лабораторная работа № 2. Матричные вычисления (на примере метода вращений)

Описание метода

Всякая ненулевая квадратная матрица A может быть записана в виде

$$A = QR, \qquad (1)$$

где Q – ортогональная матрица;

R – верхняя треугольная матрица.

Напомним, что ортогональной является матрица Q, для которой справедливо: $Q^{-1}=Q^{T}$ и $Q^{T}Q = E$, где E – единичная матрица. Произведение ортогональных матриц также является ортогональной матрицей. Верхней треугольной называется матрица, у которой все элементы ниже главной диагонали равны 0.

Представление матрицы A в виде произведения QR называется треугольным QR–разложением. Это типовое преобразование широко используется в линейной алгебре при решении систем линейных алгебраических уравнений, нахождении собственных значений матриц и получении матриц специального вида.

Рассмотрим решение систем линейных алгебраических уравнений с помощью QR–разложения, выполняемого методом вращений Гивенса.

Этот метод является методом исключения неизвестных и, как и метод Гаусса, выполняется в два этапа. На первом этапе («прямой ход») система линейных алгебраических уравнений

$$Ax = b\ , \qquad (2)$$

где A – матрица коэффициентов при неизвестных;

x – вектор-столбец неизвестных;

b – вектор-столбец свободных членов,

приводится к виду :

$$Rx = Q^T b\ , \qquad (3)$$

где R – верхняя треугольная матрица.

На втором этапе («обратный ход») метода вращений решается система (3) с треугольной матрицей коэффициентов. Обратный ход метода вращений проводится так же, как и для метода Гаусса.

Рассмотрим реализацию первого этапа решения.

Пусть задана система линейных алгебраических уравнений :

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ &a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ &\quad\ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \\ &a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n\,. \end{aligned} \qquad (4)$$

Эту систему можно записать также в матричной форме:

$$A\,x = b\ , \qquad (4`)$$

где

$$A=\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}, \qquad (5)$$

$$x=\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix}, \qquad b=\begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{pmatrix} \qquad (6)$$

На первом шаге прямого хода исключают последовательно x_1 из всех уравнений, кроме первого.

Для исключения x_1 из второго уравнений находят числа

$$s_{12}=\frac{a_{21}}{\sqrt{a_{11}^2+a_{21}^2}}; \quad c_{12}=\frac{a_{11}}{\sqrt{a_{11}^2+a_{21}^2}} \qquad (7)$$

и заменяют первое уравнение системы линейной комбинацией первого и второго уравнений с коэффициентами c_{12} и s_{12}, а второе уравнение – линейной комбинацией первого и второго уравнений с коэффициентами s_{12} и c_{12}.

Так как

$$-s_{12}a_{11}+c_{12}a_{21}=0, \qquad (8)$$

в результате получают систему:

$$a_{11}^{(1)}x_1 + a_{12}^{(1)}x_2 + a_{13}^{(1)}x_3 + \dots + a_{1n}^{(1)}x_n = b_1^{(1)},$$
$$a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \dots + a_{2n}^{(1)}x_n = b_2^{(1)},$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \ldots + a_{3n}x_n = b_3 \quad , \qquad (9)$$

$$\ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots$$

$$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n = b_n \quad ,$$

в которой

$$a_{1j}^{(1)} = c_{12}\,a_{1j} + s_{12}\,a_{2j}\,,\ a_{2j}^{(1)} = -s_{12}\,a_{1j} + c_{12}\,a_{2j}\ ,\ (1 \le j \le n), \qquad (10)$$

$$b_1^{(1)} = c_{12}\,b_1 + s_{12}\,b_2\,,\ b_2^{(1)} = -s_{12}\,b_1 + c_{12}\,b_2\,. \qquad (11)$$

Преобразование исходной системы (4) к виду (9) равносильно умножению матричного уравнения (4`) слева на матрицу P_{12}, имеющую вид

$$P_{12} = \begin{pmatrix} c_{12} & s_{12} & 0 & 0 & \ldots & 0 \\ -s_{12} & c_{12} & 0 & 0 & \ldots & 0 \\ 0 & 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 0 & 1 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & 0 & \ldots & 1 \end{pmatrix}. \qquad (12)$$

Таким образом, систему (9) можно записать в виде

$$P_{12}\,Ax = P_{12}\,b\,. \qquad (13)$$

Для исключения неизвестного x_1 из третьего уравнения находят числа

$$s_{13} = \frac{a_{31}}{\sqrt{(a_{11}^{(1)})^2 + a_{31}^2}}; \quad c_{13} = \frac{a_{11}^{(1)}}{\sqrt{(a_{11}^{(1)})^2 + a_{31}^2}} \quad . \qquad (14)$$

Первое уравнение системы (9) заменяют линейной комбинацией первого и третьего уравнений с коэффициентами c_{13} и s_{13}, а третье уравнение – линейной комбинацией этих же уравнений с коэффициентами $(-s_{13})$ и c_{13}. Это преобразование эквивалентно умножению матричного уравнения (12) слева на матрицу

$$P_{13} = \begin{pmatrix} c_{13} & 0 & s_{13} & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ -s_{13} & 0 & c_{13} & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 1 \end{pmatrix} . \qquad (15)$$

Аналогично исключают x_1 из уравнений системы с номерами $i = 4, \dots n$. В результате 1-го шага, состоящего их $n-1$ элементарных шагов, исходная система приводится к виду

$$\begin{aligned} a_{11}^{(n-1)} x_1 + a_{12}^{(n-1)} x_2 + a_{13}^{(n-1)} x_3 + \dots + a_{1n}^{(n-1)} x_n &= b_1^{(n-1)} , \\ a_{22}^{(1)} x_2 + a_{23}^{(1)} x_3 + \dots + a_{2n}^{(1)} x_n &= b_2^{(1)} , \\ a_{32}^{(1)} x_2 + a_{33}^{(1)} x_3 + \dots + a_{3n}^{(1)} x_n &= b_3^{(1)} , \\ \dots \quad \dots \quad \dots \quad \dots \quad \dots & \\ a_{n2}^{(1)} x_2 + a_{n3}^{(1)} x_3 + \dots + a_{nn}^{(1)} x_n &= b_n^{(1)} . \end{aligned} \qquad (16)$$

В матричной записи эта система имеет вид

$$A^{(1)} x = b^{(1)}, \qquad (17)$$

где $A^{(1)} = P_{1n} \dots P_{13} P_{12} A$, $b^{(1)} = P_{1n} \dots P_{13} P_{12} b$. (18)

Через P_{1j} обозначена матрица элементарного преобразования, отличающаяся от единичной матрицы E только четырьмя элементами. В ней элементы с индексами (1, 1) и (j, j) равны c_{1j}, элемент с индексами (1, j) равен s_{1j}, а элемент с индексами (j, 1) равен ($-s_{1j}$).

Таким образом, матрицу P_{1j} можно записать в виде

$$P_{1j} = \begin{pmatrix} c_{1j} & 0 & 0 & \dots & 0 & s_{1j} & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ -s_{1j} & 0 & 0 & \dots & 0 & c_{1j} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 1 \end{pmatrix}, \qquad (19)$$

где $$s_{1j} = \frac{a_{j1}}{\sqrt{(a_{11}^{(j-2)})^2 + a_{j1}^2}}; \quad c_{1j} = \frac{a_{11}^{(j-2)}}{\sqrt{(a_{11}^{(j-2)})^2 + a_{j1}^2}}. \qquad (20)$$

При этом выполняются условия

$$c_{1j}^2 + s_{1j}^2 = 1, \quad -s_{1j} a_{11}^{(j-2)} + c_{1j} a_{j1} = 0. \qquad (21)$$

Действие матрицы P_{1j} на вектор x эквивалентно его повороту вокруг оси, перпендикулярной плоскости Ox_1x_j на угол $\cup_{1j}$ такой, что

$$c_{1j} = \cos \cup_{1j}, \quad s_{1j} = \sin \cup_{1j}. \qquad (22)$$

(Существование такого угла гарантируется первым из равенств (21)). Эта геометрическая интерпретация и дала название методу вращений. Так как $P_{1j}^{T} = P_{1j}^{-1}$, то матрица P_{1j} является ортогональной.

На втором шаге метода вращений, состоящем из $n - 2$ элементарных шагов, из уравнений системы (16) с номерами i =3, 4, … n исключают неизвестное x_2 . Для этого каждое i–ое уравнение комбинируют со вторым уравнением. В результате получают систему

$$\begin{aligned} a_{11}^{(n-1)} x_1 + a_{12}^{(n-1)} x_2 + a_{13}^{(n-1)} x_3 + \ldots + a_{1n}^{(n-1)} x_n &= b_1^{(n-1)}, \\ a_{22}^{(n-1)} x_2 + a_{23}^{(n-1)} x_3 + \ldots + a_{2n}^{(n-1)} x_n &= b_2^{(n-1)}, \\ a_{33}^{(2)} x_3 + \ldots + a_{3n}^{(2)} x_n &= b_3^{(2)}, \\ \ldots \quad \ldots \quad \ldots \quad \ldots & \\ a_{n3}^{(2)} x_3 + \ldots + a_{nn}^{(2)} x_n &= b_n^{(2)}. \end{aligned} \qquad (23)$$

В матричной форме записи система (23) имеет вид

$$A^{(2)} x = b^{(2)}, \qquad (24)$$

где $A^{(2)} = P_{2n} \ldots P_{24} P_{23} A^{(1)}$, $b^{(2)} = P_{2n} \ldots P_{24} P_{23} b^{(1)}$. (25)

Матрица P_{2j} отличается от единичной матрицы E четырьмя элементами. В ней элементы с индексами (2, 2) и (j, j) равны c_{2j}, элемент с индексами (2, j) равен s_{2j}, а элемент с индексами (j, 2) равен ($-s_{2j}$). При этом

$$s_{2j} = \frac{a_{j2}^{(1)}}{\sqrt{(a_{22}^{(j-2)})^2 + (a_{j2}^{(1)})^2}}; \quad c_{2j} = \frac{a_{22}^{(j-2)}}{\sqrt{(a_{22}^{(j-2)})^2 + (a_{j2}^{(1)})^2}} \quad \text{и} \qquad (26)$$

$$c_{2j}^2 + s_{2j}^2 = 1, \quad -s_{2j} a_{22}^{(j-2)} + c_{2j} a_{j2}^{(1)} = 0. \qquad (27)$$

Легко проверить, что матрицы P_{2j} являются ортогональными. После завершения $(n - 1)$ – го шага система принимает вид

$$
\begin{aligned}
a_{11}^{(n-1)} x_1 + a_{12}^{(n-1)} x_2 + a_{13}^{(n-1)} x_3 + \ldots + a_{1n}^{(n-1)} x_n &= b_1^{(n-1)}, \\
a_{22}^{(n-1)} x_2 + a_{23}^{(n-1)} x_3 + \ldots + a_{2n}^{(n-1)} x_n &= b_2^{(n-1)}, \\
a_{33}^{(n-1)} x_3 + \ldots + a_{3n}^{(n-1)} x_n &= b_3^{(n-1)}, \\
\ldots \quad \ldots \quad \ldots \quad \ldots & \\
a_{nn}^{(n-1)} x_n &= b_n^{(n-1)},
\end{aligned}
\qquad (28)
$$

или в матричной форме записи

$$A^{(n-1)}x = b^{(n-1)}, \qquad (29)$$

где $A^{(n-1)} = P_{n-1,\,n}\, A^{(n-2)}$, $b^{(n-1)} = P_{n-1,\,n}\, b^{(n-2)}$. (30)

Таким образом, «прямой ход» или первый этап решения исходной системы завершен и найдена матрица $R = A^{(n-1)}$.

Матрица коэффициентов преобразованной системы R связана с исходной матрицей системы A равенством

$$R = PA, \qquad (31)$$

где $P = P_{n-1,\,n} \ldots P_{2n} \ldots P_{23}\, P_{1n} \ldots P_{12}$ (32)

– матрица результирующего преобразования. Для реализации первого этапа решения (и нахождения матрицы R) потребовалось выполнение $(n-1) + (n-2) + \ldots + 1 = n\,(n-1)/2$ элементарных преобразований с помощью матриц P_{ij}.

Матрица P_{ij} отличается от единичной матрицы E четырьмя элементами. В ней элементы с индексами (i, i) и (j, j) равны c_{ij}, элемент с индексами (i, j) равен s_{ij}, а элемент с индексами (j, i) равен $(-s_{ij})$, где

$$s_{ij} = \frac{a_{ji}^{(i-1)}}{\sqrt{(a_{ii}^{(j-2)})^2 + (a_{ji}^{(i-1)})^2}}; \quad c_{ij} = \frac{a_{ii}^{(j-2)}}{\sqrt{(a_{ii}^{(j-2)})^2 + (a_{ji}^{(i-1)})^2}}. \qquad (33)$$

При этом

$$c_{ij}^2 + s_{ij}^2 = 1, \quad -s_{ij} a_{ii}^{(j-2)} + c_{ij} a_{ji}^{(i-1)} = 0. \qquad (34)$$

Действие матрицы P_{ij} на любой n – мерный вектор-столбец эквивалентно его повороту вокруг оси, перпендикулярной плоскости Ox_ix_j на угол $\cup_{ij}$ такой, что

$$c_{ij} = \cos \cup_{ij}, \quad s_{ij} = \sin \cup_{ij}. \qquad (35)$$

Матрицу P_{ij} называют матрицей вращения Гивенса, а операцию умножения на матрицу P_{ij} – плоским вращением или вращением Гивенса. Легко проверить, что каждая из матриц вращения Гивенса ортогональна, т.е. $P_{ij}^{\mathrm{T}} = P_{ij}^{-1}$. Поэтому и матрица P является ортогональной как произведение ортогональных матриц. Таким образом, система (28) может быть записана в виде

$$R\,x = P\,b, \qquad (36)$$

где R – верхняя треугольная матрица, а P – ортогональная матрица.

Обозначая $P^{-1} = P^{\mathrm{T}} = Q$, получаем QR – разложение матрицы A. Следует отметить, что при реализации метода вращений на ЭВМ матрица P, как правило, фактически не вычисляется и не запоминается. В случае, если требуется найти QR–разложение матрицы A для фактического определения матрицы P можно применить рассмотренный алгоритм

решения системы уравнений, задавая соответствующим образом столбец свободных членов.

Рассмотрим численный пример. Решим систему уравнений, приведенную в [1] :

$$\begin{aligned} 2x_1 &- 9x_2 + 5x_3 = -4, \\ 1.2x_1 &- 5.3999x_2 + 6x_3 = 0.6001, \\ x_1 &- x_2 - 7.5x_3 = -8.5 . \end{aligned}$$

Прямой ход. Первый шаг. Исключим x_1 из второго уравнения. Для этого вычислим $c_{12} = \cos\theta_{12}$, $s_{12} = \sin\theta_{12}$ по формулам (7):

$$c_{12} = \frac{2}{\sqrt{2^2 + 1.2^2}} \approx 0.857493; \quad s_{12} = \frac{1.2}{\sqrt{2^2 + 1.2^2}} \approx 0.514495 .$$

Преобразуя коэффициенты первого и второго уравнения в соответствии с (9), приходим к системе :

$$\begin{aligned} 2.33238\,x_1 &- 10.4957\,x_2 + 7.37444\,x_3 = -3.12122, \\ &7.85493*10^{-5}\,x_2 + 2.57248\,x_3 = 2.57256, \\ x_1 &- x_2 - 7.5\,x_3 = -8.5 \end{aligned}$$

Далее вычисляем коэффициенты c_{13}, s_{13} по формулам (14) :

$$c_{13} = \frac{2.33238}{\sqrt{2.33238 + 1^2}} \approx 0.919087; \quad s_{13} = \frac{1}{\sqrt{2.33238^2 + 1^2}} \approx 0.394055$$

Преобразуя коэффициенты первого и третьего уравнения как в (16), получаем систему:

$$\begin{aligned} 2.53772\,x_1 \;-\; & 10.0405\;x_2 \;+\; 3.82234\,x_3 \;=\; -6.21814, \\ & 7.85493*10^{-5}\,x_2 \;+\; 2.57248\,x_3 \;=\; 2.57256, \\ & 3.21680\,x_2 \;-\; 9.79909\,x_3 \;=\; -6.58231 \end{aligned}$$

В первом столбце поддиагональные элементы равны 0.

Второй шаг. В полученной системе имеем $a`_{22} = 7.85493*10^{-5}$, $a`_{32} = 3.2168$. Поэтому

$$c_{12} = \frac{a'_{22}}{\sqrt{(a'_{22})^2 + (a'_{32})^2}} \approx 2.44185*10^{-5}; \quad s_{12} \approx 1$$

Преобразуя коэффициенты второго и третьего уравнения, получаем систему :

$$\begin{aligned} 2.53772\,x_1 \;-\; & 10.0405\;x_2 \;+\; 3.82234\,x_3 \;=\; -6.21814, \\ & 3.21680\;x_2 \;-\; 9.79903\,x_3 \;=\; -6.58225, \\ & -\; 2.57272\,x_3 \;=\; -2.57272. \end{aligned}$$

Обратный ход дает последовательно значения $x_3 = 1$, $x_2 = 0.999994$, $x_1 = -1.58579*10^{-5}$.

Трудоемкость и оценка погрешности

Известны оценки трудоемкости методов Гаусса и вращений. Для реализации прямого хода метода Гаусса по схеме единственного деления

требуется примерно 2/3 m^3 арифметических операций, для реализации прямого хода метода Гаусса с выбором главного элемента по столбцу (по схеме частичного выбора) дополнительно требуется примерно m^2 действий (m^2 операций сравнения и перезаписи информации), а для реализации прямого хода метода Гаусса по схеме полного выбора дополнительно к $(2/3)m^3$ арифметических операций требуется примерно $m^3/3$ операций сравнения. Реализация прямого хода метода вращений требует примерно $2m^3$ арифметических операций, так что метод вращений является более трудоемким по сравнению с методом Гаусса.

Известно, что для реализации на ЭВМ методов исключения (Гаусса и вращений) справедлива следующая оценка относительной погрешности полученного численного решения:

$$\frac{\left\|x - x^*\right\|_2}{\left\|x\right\|_2} \le f(n) cond_E(A)\varepsilon_M \quad , \qquad (37)$$

где x^* – вычисленное на ЭВМ решение системы, $cond_E(A) = \|A\|_E \left\|A^{-1}\right\|_E$ – число обусловленности матрицы A, ε_M – машинное эпсилон и $f(n) = c(n)\Pi(n)$, где $c(n)$ – некоторая медленно растущая функция, зависящая от порядка n системы, $\Pi(n)$ – коэффициент роста, $\|x\|_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$ – евклидова норма вектора, а $\|A\|_E = \sqrt{\sum_{i,j=1}^{n} |a_{ij}|^2}$ – евклидова норма матрицы A. Число обусловленности матрицы A можно интерпретировать как отношение максимального коэффициента растяжения векторов под действием матрицы A к минимальному коэффициенту.

Для метода Гаусса по схеме частичного выбора

$$П(n) = 2^{n-1}. \quad (38)$$

Для метода Гаусса по схеме полного выбора

$$П(n) \delta 1{,}8n^{0{,}25 \ln n}. \quad (39)$$

Для метода вращений $f(n) = 6n$. (40)

Таким образом, для систем с большим числом неизвестных метод Гивенса обладает большей численной устойчивостью по сравнению с методом Гаусса и позволяет находить решение даже для плохо обусловленных матриц.

Метод Гивенса более сложен при реализации, поэтому на ЭВМ общего назначения его выполнение занимает больше времени. Однако при реализации решения в виде схемы специализированного устройства (матричный процессор) метод оказывается более подходящим, чем методы Гаусса. Помимо своей численной устойчивости, метод предпочтительнее тем, что может допускать распараллеливание лучше, чем метод Гаусса. Выше при описании метода приведена одна из возможных *схем аннулирования* элементов в столбце, при которой пары (i, j) выбираются таким образом, что i всегда равно номеру диагонального элемента. В то же время при реализации на параллельной вычислительной системе можно выбирать схему аннулирования, при которой весь столбец разбивается на непересекающиеся пары (i, j) , что дает за один параллельный шаг обнуление половины элементов, в следующем – половины от половины и так далее – по пирамидальной схеме. Таким образом, для обнуления всех поддиагональных элементов столбца потребуется не k последовательных

шагов, как в первой схеме аннулирования, а $log_2 k$, где k – число поддиагональных элементов.

Алгоритм решения СЛАУ методом Гивенса

Алгоритм решения СЛАУ методом Гивенса представлен на рис.5. Алгоритм состоит из ввода (определения) данных, вычисления верхнетреугольной матрицы и правых частей преобразованной системы по методу Гивенса и обратной подстановки. В свою очередь, вычисление верхнетреугольной матрицы выполняется по алгоритму, представленному на рис. 6. Внешний цикл (по i) выполняется по столбцам, вложенный цикл (по j) – по строкам для выбора обнуляемой компоненты столбца.

Рисунок 5 - Алгоритм решения СЛАУ методом Гивенса

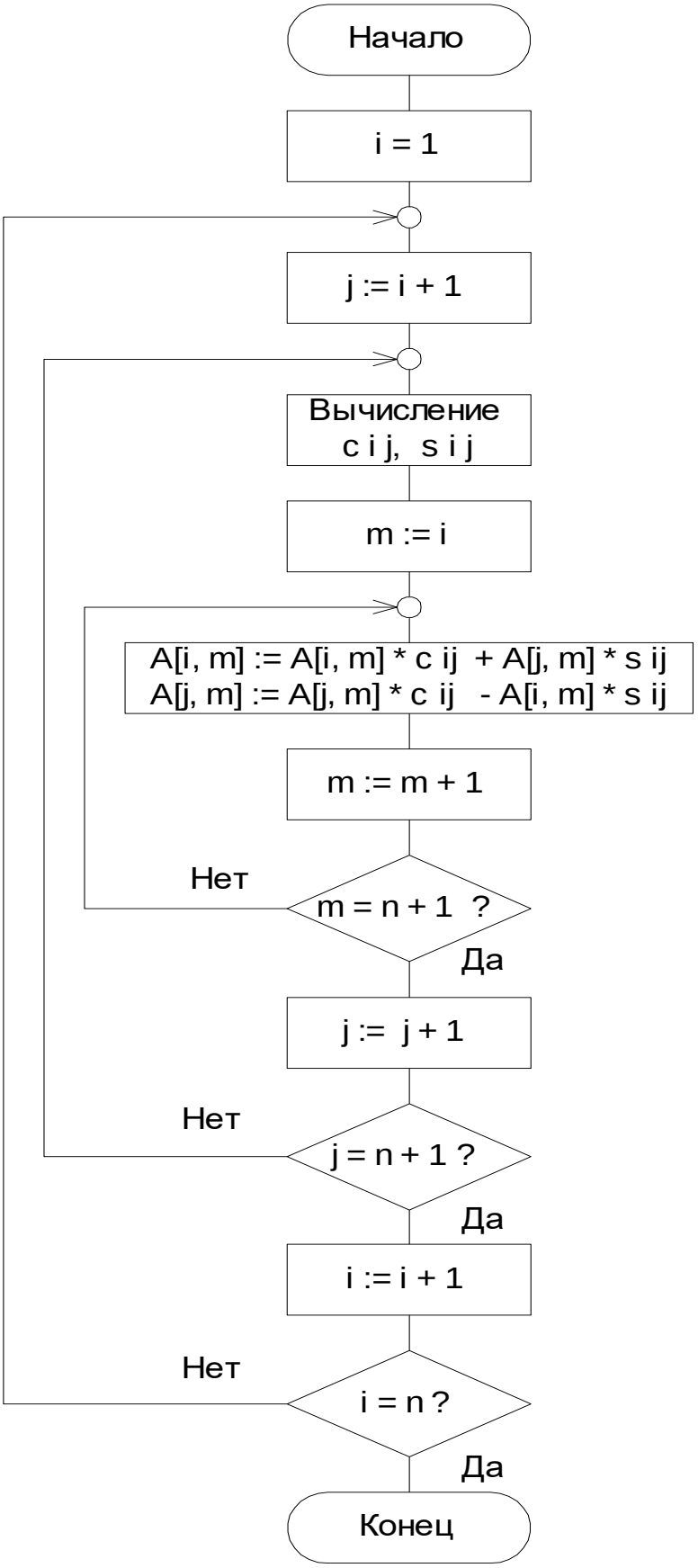


Рисунок 6 - вычисление верхнетреугольной матрицы

Порядок выполнения лабораторной работы

При выполнении лабораторной работы необходимо составить программу для решения заданной системы линейных уравнений методом Гивенса по приведенному во втором пункте алгоритму. В программе необходимо предусмотреть проверку решения его подстановкой в исходную систему с вычислением и выводом на печать (или экран) невязок.

Порядок выполнения работы следующий:

1. Получить задание у преподавателя.
2. Составить программу на выбранном языке высокого уровня.
3. Произвести расчет и сохранить результаты прогона программы.

Дополнительное задание:

составить программу для решения той же системы методом Гаусса и сравнить затраты времени (для этого необходимо воспользоваться встроенными функциями используемого языка высокого уровня для определения времени, затраченного на вычисления).

Варианты заданий

Решите систему уравнений

$Ax = b$, где

1

$$A = \begin{pmatrix} -6 & -1 & -3 & 1 & 7 \\ 9 & -7 & 3 & -3 & -9 \\ -3 & 1 & -7 & -7 & 8 \\ 3 & 7 & -9 & 6 & 1 \\ 8 & 3 & -6 & 2 & -7 \end{pmatrix}; \quad b = \begin{pmatrix} -2 \\ 5 \\ -1 \\ 6 \\ -7 \end{pmatrix}$$

2

$$A = \begin{pmatrix} 4 & -7 & 6 & -9 & 6 \\ 7 & 2 & 6 & -2 & -9 \\ -8 & -6 & -3 & 9 & -2 \\ 7 & -6 & 3 & 2 & 6 \\ 3 & 1 & 4 & -3 & 7 \end{pmatrix}; \quad b = \begin{pmatrix} 7 \\ -1 \\ -4 \\ -2 \\ -2 \end{pmatrix}$$

3

$$A = \begin{pmatrix} 9 & -9 & -5 & -4 & -7 \\ 9 & -5 & 1 & 9 & -6 \\ -5 & 7 & 5 & 5 & 6 \\ 2 & 8 & -1 & 3 & -2 \\ -4 & -9 & -8 & -6 & 5 \end{pmatrix}; \quad b = \begin{pmatrix} 5 \\ -4 \\ -6 \\ -9 \\ -7 \end{pmatrix}$$

4

$$A = \begin{pmatrix} 5 & 3 & -9 & -4 & -2 \\ -2 & 2 & -8 & 8 & -8 \\ 1 & 2 & -5 & -4 & -5 \\ 1 & 8 & 6 & 5 & 6 \\ 7 & 8 & -1 & -3 & -4 \end{pmatrix}; \quad b = \begin{pmatrix} -9 \\ -6 \\ 8 \\ -9 \\ -7 \end{pmatrix}$$

5

$$A = \begin{pmatrix} -4 & 2 & -6 & 1 & -7 \\ -3 & 6 & -4 & 7 & -1 \\ 8 & -4 & 5 & -9 & 1 \\ -8 & 7 & -8 & -3 & 8 \\ 9 & -9 & 6 & 8 & 9 \end{pmatrix}; \quad b = \begin{pmatrix} -6 \\ 6 \\ 9 \\ -4 \\ -4 \end{pmatrix}$$

6

$$A = \begin{pmatrix} 1 & 9 & 1 & -9 & -8 \\ -4 & 2 & 5 & 3 & -6 \\ -4 & 9 & 3 & 5 & 7 \\ 9 & -9 & -7 & 2 & -6 \\ -9 & 3 & 5 & -4 & 3 \end{pmatrix}; \quad b = \begin{pmatrix} 4 \\ -3 \\ 1 \\ 6 \\ -7 \end{pmatrix}$$

7

$$A = \begin{pmatrix} -8 & 1 & 5 & -8 & 9 \\ -6 & -9 & 5 & -6 & 4 \\ 9 & -7 & 8 & 2 & 6 \\ 3 & -2 & -8 & -7 & -7 \\ -3 & -7 & 8 & 8 & 2 \end{pmatrix}; \quad b = \begin{pmatrix} 3 \\ -2 \\ 4 \\ 1 \\ -8 \end{pmatrix}$$

8

$$A = \begin{pmatrix} -1 & 1 & -1 & -2 & 1 \\ 2 & 5 & 1 & -1 & 8 \\ 8 & -8 & -2 & 1 & -3 \\ -6 & 7 & 9 & 6 & -9 \\ -9 & -4 & 6 & 9 & -9 \end{pmatrix}; \quad b = \begin{pmatrix} 3 \\ -2 \\ -3 \\ -2 \\ 7 \end{pmatrix}$$

9

$$A = \begin{pmatrix} 4 & -1 & -9 & -7 & -5 \\ 2 & 6 & -6 & 2 & 5 \\ -7 & 4 & -6 & -3 & 1 \\ -6 & -6 & 5 & 4 & 1 \\ -9 & -9 & 3 & -9 & -6 \end{pmatrix}; \quad b = \begin{pmatrix} 6 \\ 9 \\ 3 \\ -9 \\ -8 \end{pmatrix}$$

1

$$A = \begin{pmatrix} 1 & 4 & 2 & -6 & -3 \\ 9 & 9 & 3 & -5 & 5 \\ 2 & -2 & -4 & -8 & -6 \\ 2 & 4 & 9 & -5 & -9 \\ 4 & -9 & 4 & -3 & 6 \end{pmatrix}; \quad b = \begin{pmatrix} 5 \\ -4 \\ -6 \\ 3 \\ -2 \end{pmatrix}$$

$$\frac{1}{A}\begin{pmatrix} -9 & 5 & 9 & 4 & 5 \\ 3 & 2 & 1 & 5 & -1 \\ -4 & 8 & -6 & 8 & 1 \\ -5 & -6 & -7 & 8 & -7 \\ 4 & -3 & -6 & -4 & -7 \end{pmatrix}; \quad \begin{pmatrix} -1 \\ -9 \\ 9 \\ -3 \\ -8 \end{pmatrix} \frac{1}{A} \begin{pmatrix} -6 & 9 & 9 & 1 & -9 \\ 9 & -8 & 8 & -5 & -7 \\ -5 & -7 & -8 & -1 & 6 \\ -4 & -3 & 6 & 3 & 5 \\ -2 & -8 & 6 & 1 & -2 \end{pmatrix}; \quad \begin{pmatrix} 8 \\ -7 \\ 7 \\ -5 \\ 7 \end{pmatrix}$$

$$\frac{1}{A}\begin{pmatrix} -6 & -4 & -3 & 3 & 4 \\ -9 & 6 & -2 & -2 & 8 \\ 5 & -9 & -4 & -5 & -1 \\ -8 & 3 & -7 & 7 & -1 \\ 3 & 9 & -4 & 9 & 3 \end{pmatrix}; \quad \begin{pmatrix} 3 \\ 6 \\ -1 \\ -2 \\ -5 \end{pmatrix} \frac{1}{A} \begin{pmatrix} -7 & 2 & 8 & 8 & -1 \\ -3 & -7 & -8 & -5 & 6 \\ 8 & -3 & -2 & -1 & -8 \\ -3 & 3 & 7 & 9 & 4 \\ -3 & -5 & -9 & 7 & -7 \end{pmatrix}; \quad \begin{pmatrix} -3 \\ -5 \\ 4 \\ 3 \\ -9 \end{pmatrix}$$

$$\frac{1}{A}\begin{pmatrix} 2 & -5 & -2 & 4 & 7 \\ 3 & -7 & -1 & 1 & 7 \\ 2 & 5 & 4 & -1 & -3 \\ -8 & -8 & 1 & 2 & 3 \\ -6 & -5 & -3 & -1 & 6 \end{pmatrix}; \quad \begin{pmatrix} 4 \\ 3 \\ -8 \\ -6 \\ -9 \end{pmatrix} \frac{1}{A} \begin{pmatrix} -3 & -8 & 2 & -6 & 9 \\ -4 & -9 & -1 & 5 & -9 \\ -6 & 2 & 3 & 3 & -2 \\ -2 & 3 & 9 & 3 & 3 \\ 7 & 8 & -8 & 9 & -7 \end{pmatrix}; \quad \begin{pmatrix} 3 \\ -9 \\ 6 \\ 9 \\ -4 \end{pmatrix}$$

$$\frac{1}{A}\begin{pmatrix} 1 & -9 & -6 & -1 & -9 \\ -7 & 4 & -2 & -2 & 6 \\ 4 & -8 & 4 & 9 & -5 \\ -6 & 9 & 7 & 1 & 7 \\ 4 & -5 & 4 & -5 & -1 \end{pmatrix}; \quad \begin{pmatrix} 9 \\ 9 \\ 4 \\ -8 \\ -4 \end{pmatrix} \frac{1}{A} \begin{pmatrix} 1 & -2 & 9 & 6 & -2 \\ 8 & -1 & 2 & 5 & -3 \\ -9 & 8 & -7 & -1 & 1 \\ -4 & -2 & 5 & 8 & 2 \\ -8 & 1 & -7 & -5 & -1 \end{pmatrix}; \quad \begin{pmatrix} 1 \\ 5 \\ -5 \\ -5 \\ -7 \end{pmatrix}$$

$$\frac{1}{A}\begin{pmatrix} 4 & 2 & -9 & -2 & -3 \\ 1 & -5 & 8 & -1 & 4 \\ 9 & -3 & -3 & -4 & 1 \\ 3 & -9 & 8 & 2 & 1 \\ -6 & -6 & 6 & 3 & 3 \end{pmatrix}; \quad \begin{pmatrix} 7 \\ -9 \\ 6 \\ 3 \\ -5 \end{pmatrix} \frac{2}{A} \begin{pmatrix} -7 & -5 & -1 & -3 & -1 \\ 4 & 6 & 9 & -2 & -7 \\ 3 & 9 & 3 & 1 & -3 \\ 6 & -5 & -5 & -6 & 8 \\ -6 & 4 & -9 & 4 & -6 \end{pmatrix}; \quad \begin{pmatrix} -5 \\ -7 \\ 8 \\ -3 \\ -5 \end{pmatrix}$$

$$\frac{2}{A}\begin{pmatrix} 1 & -7 & 2 & -9 & 5 \\ -3 & -4 & 5 & -1 & -6 \\ 9 & -5 & -5 & -3 & -4 \\ 2 & 1 & 4 & -1 & 6 \\ -2 & 8 & -1 & 5 & -2 \end{pmatrix}; \quad \begin{pmatrix} -6 \\ -8 \\ 4 \\ 7 \\ 9 \end{pmatrix} \frac{2}{A} \begin{pmatrix} 8 & -4 & 6 & -3 & 5 \\ 7 & -3 & 1 & 1 & -2 \\ -4 & -1 & -6 & -5 & 1 \\ 3 & 9 & -9 & 8 & -6 \\ -5 & 1 & -3 & 5 & -9 \end{pmatrix}; \quad \begin{pmatrix} 1 \\ 5 \\ 7 \\ 8 \\ 6 \end{pmatrix}$$

$$A \quad ; \qquad A \quad ;$$

2

$$\begin{pmatrix} -8 & 6 & -7 & -1 & 1 \\ 2 & 5 & 4 & 4 & -6 \\ 6 & -5 & -6 & -9 & 5 \\ 1 & 9 & 4 & 2 & 3 \\ -2 & -8 & 8 & -8 & 2 \end{pmatrix} \quad \begin{pmatrix} -6 \\ -6 \\ -8 \\ 2 \\ 4 \end{pmatrix}$$

2

$$\begin{pmatrix} 7 & -6 & -4 & -1 & -5 \\ 6 & 6 & 9 & 4 & 7 \\ -1 & 4 & 5 & 5 & -2 \\ 9 & 4 & 6 & 6 & 9 \\ 7 & -3 & -7 & 1 & -6 \end{pmatrix} \quad \begin{pmatrix} -9 \\ -5 \\ -4 \\ -6 \\ 6 \end{pmatrix}$$

2

$$\begin{pmatrix} 2 & -1 & -5 & -7 & 4 \\ -3 & -6 & 2 & 6 & -2 \\ 5 & 2 & 8 & -9 & 2 \\ 7 & -2 & 4 & -3 & 7 \\ -5 & -4 & 1 & 9 & 2 \end{pmatrix} \quad \begin{pmatrix} 3 \\ -9 \\ 5 \\ 4 \\ -8 \end{pmatrix}$$

2

$$\begin{pmatrix} 3 & 6 & -9 & 2 & 4 \\ 2 & 6 & -3 & 3 & -8 \\ 2 & -6 & 3 & -1 & 9 \\ 4 & -4 & 7 & -8 & -4 \\ -5 & -8 & 6 & -8 & 4 \end{pmatrix} \quad \begin{pmatrix} -2 \\ 6 \\ -4 \\ 5 \\ -4 \end{pmatrix}$$

2

$$\begin{pmatrix} -3 & -7 & -8 & 2 & 2 \\ 5 & 9 & 9 & 6 & -9 \\ -8 & 3 & 9 & 6 & 9 \\ -8 & -7 & 8 & -4 & 4 \\ 4 & -3 & 3 & -2 & -2 \end{pmatrix} \quad \begin{pmatrix} -1 \\ 7 \\ 9 \\ 9 \\ -6 \end{pmatrix}$$

2

$$\begin{pmatrix} -8 & -1 & -9 & -2 & 5 \\ 9 & -3 & -4 & 4 & 6 \\ -2 & 4 & -8 & -1 & 4 \\ -8 & -6 & 2 & 3 & 6 \\ 8 & -5 & 9 & -9 & -5 \end{pmatrix} \quad \begin{pmatrix} -2 \\ -8 \\ -8 \\ -3 \\ -1 \end{pmatrix}$$

Требования к оформлению протокола по лабораторной работе

Протокол выполнения лабораторной работы должен включать:

1. Цель работы.
2. Задание.
3. Алгоритм работы программы.
4. Текст программы.
5. Результаты прогона программы.
6. При выполнении дополнительного задания протокол может содержать результат прогона программы решения СЛАУ методом Гаусса, алгоритм и текст этой программы, а также – показатели времени

выполнения программы, реализующей метод Гивенса и программы метода Гаусса.

Отчет должен иметь титульный лист, на котором указывается предмет, наименование лабораторной работы, фамилия и номер группы студента, фамилия преподавателя.

Контрольные вопросы к отчету по лабораторной работе

1. Что такое *QR*– разложение ? Для чего оно применяется ?
2. Опишите метод вращений Гивенса для решения СЛАУ.
3. Какие еще методы решения СЛАУ вы знаете ?
4. Преимущества и недостатки метода вращений Гивенса.
5. Дайте определение ортогональной, единичной, треугольной, верхнетреугольной матрицам, матрице вращения Гивенса.
6. Приведите и поясните алгоритм для реализации метода Гивенса на ЭВМ.
7. Что такое схема аннулирования в методе Гивенса ?
8. Как можно получить ортогональную матрицу *Q*, входящую в *QR*– разложение ?

Лабораторная работа № 3. Изучение теоретико-числовых алгоритмов на примере расширенного алгоритма Евклида и алгоритма Миллера-Рабина

Цели работы:

1. Изучить сущность и принцип работы расширенного алгоритма Евклида.
2. Изучить сущность и принцип работы алгоритма Миллера-Рабина.
3. Получить навыки реализации теоретико-числовых алгоритмов на языке программирования C++.

Теоретическое введение

Делитель натурального числа – это такое целое натуральное число, на которое делится данное число без остатка. Если у натурального числа больше двух делителей, его называют составным. Общий делитель нескольких целых чисел – это такое число, которое может быть делителем каждого числа из указанного множества.

Наибольшим общим делителем двух целых неотрицательных чисел a и b называется наибольшее число, которое является делителем одновременно и a, и b. На английском языке «наибольший общий делитель» пишется «greatest common divisor», и распространённым его обозначением является $gcd(a, b)$.

Одним из эффективных алгоритмов нахождения наибольшего общего делителя (НОД) пары целых чисел является алгоритм Евклида.

Алгоритм нахождения НОД делением

1. Большее число делим на меньшее.
2. Если делится без остатка, то меньшее число и есть НОД (следует выйти из цикла).
3. Если есть остаток, то большее число заменяем на остаток от деления.

4. Переходим к пункту 1.

Пример:

Найти НОД для 30 и 18.

30 / 18 = 1 (остаток 12)

18 / 12 = 1 (остаток 6)

12 / 6 = 2 (остаток 0)

Конец: НОД – это делитель 6.

НОД (30, 18) = 6

Блок-схема алгоритма Евклида показана на рисунке 7.

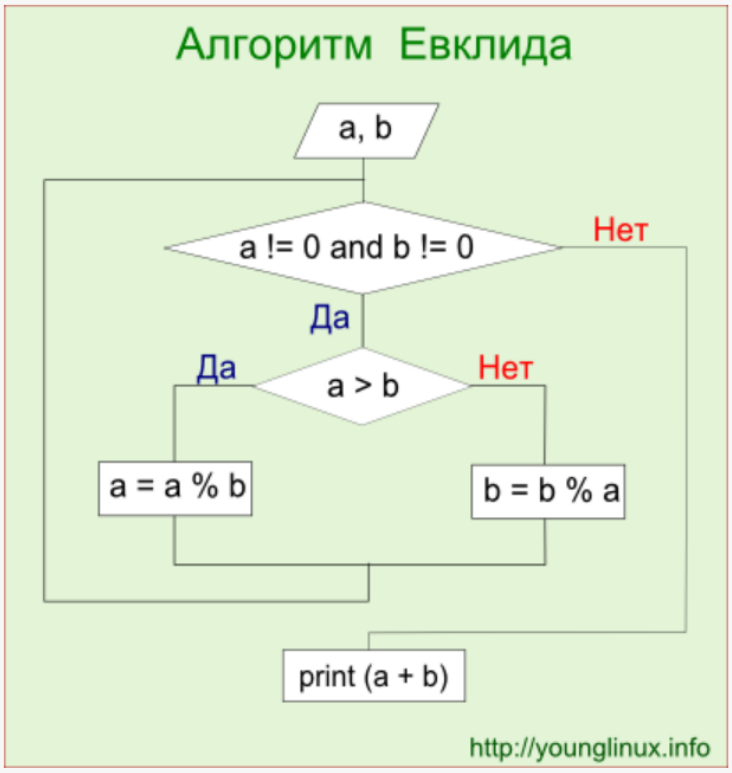


Рисунок 7 – Блок-схема алгоритма Евклида

Листинг процедуры Евклида с примером расчета НОД на языке C++ приведен ниже.

```
a = 50
b = 130
while a != 0 and b != 0:
    if a > b:
```

```
        a = a % b
    else:
        b = b % a
print(a + b)
```

В цикле в переменную a или b записывается остаток от деления. Цикл завершается, когда хотя бы одна из переменных равна нулю. Это значит, что другая переменная содержит НОД. Однако, какая именно, мы не знаем. Поэтому для НОД находим сумму этих переменных. Поскольку в одной из переменных ноль, он не оказывает влияние на результат.

Алгоритм Евклида можно *расширить* для нахождения по заданным a и b таких целых x и y, что ax + by = d, где d – наибольший общий делитель a и b.

Лемма. Пусть для положительных целых чисел a и b ($a > b$) известны d = НОД(a, b) = НОД(b, a mod b), а также числа x* и y*, для которых:

$$d = x^* \cdot b + y^* \cdot (a \bmod b).$$

Тогда значения x и y, являющиеся решениями уравнения ax + by = d, находятся из соотношений:

$$x = y^*,$$

$$y = x^* - y^* \cdot \left\lfloor \frac{a}{b} \right\rfloor$$

Через $\left\lfloor \frac{a}{b} \right\rfloor$ обозначена целая часть частного чисел a и b.

Доказательство. Поскольку a mod b = a – $\left\lfloor \frac{a}{b} \right\rfloor$ · b, то

d = x* · b + y* · (a – $\left\lfloor \frac{a}{b} \right\rfloor$ · b) = y* · a + (x* – y* · $\left\lfloor \frac{a}{b} \right\rfloor$) · b = x · a + y · b,

где обозначено x = y*, y = x* – y* · $\left\lfloor \frac{a}{b} \right\rfloor$.

Функция `gcdext(int a, int b, int *d, int *x, int *y)`, приведенная ниже, по входным числам a и b находит d = НОД(a, b) и такие x, y, что d = a · x + b · y. Для поиска неизвестных x и y необходимо рекурсивно запустить функцию gcdext(b, a mod b, d, x, y) и пересчитать значения x и y по выше приведенной формуле. Рекурсия заканчивается,

когда b = 0. При b = 0 НОД(a, 0) = a и a = a · 1 + 0 · 0, поэтому полагаем x = 1, y = 0.

```
void gcdext (int a, int b, int *d, int *x, int *y)
{
  int s;
  if (b == 0)
  {
    *d = a; *x = 1; *y = 0;
    return;
  }
  gcdext(b, a % b, d, x, y);
  s = *y;
  *y = *x - (a / b) * (*y);
  *x = s;
}
```

Для ручного выполнения расширенного алгоритма Евклида удобно воспользоваться таблицей с четырьмя столбцами (как показано на рисунке 8), соответствующими значениям *a*, *b*, *x*, *y*. Процесс вычисления разделим на три этапа:

а) Сначала вычислим НОД(a, b), используя первые два столбца таблицы и соотношение (1). В последней строке в колонке a будет находиться значение d = НОД(a, b).

б) Занесем значения 1 и 0 соответственно в колонки x и y последней строки.

в) Будем заполнять последовательно строки для колонок x и y снизу вверх. Значения x и y в последнюю строку уже занесены на этапе б). Считая, что в текущей заполненной строке уже находятся значения (x*, y*), мы будем пересчитывать и записывать значения (x, y) в соответствующие ячейки выше стоящей строки.

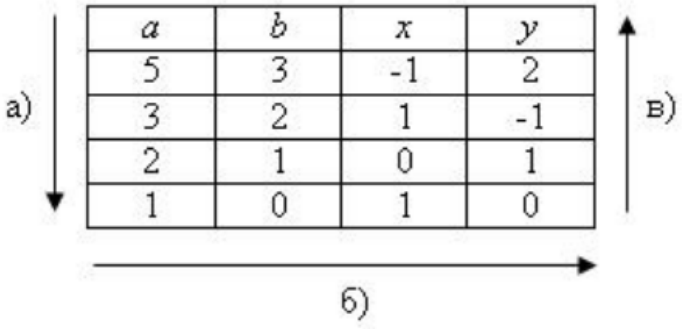


Рисунок 8 – Иллюстрация примера использования расширенного алгоритма Евклида

Найдем решение уравнения 5x + 3y = 1. Вычисление НОД(5, 3) и нахождение соответствующих x, y произведем в таблице, показанной на рисунке 2. Порядок и направление вычислений указаны стрелками и буквами. Из таблицы находим, что НОД(5, 3) = 5 · (-1) + 3 · 2 = 1, то есть x = -1, y = 2.

Рассмотрим, например, как получены значения (x, y) для первой строки. Со второй строки берем значения (x*, y*) = (1, -1). Отсюда получим:

x = y* = -1,

$y = x^* - y^* \cdot \lfloor \frac{a}{b} \rfloor = 1 - (-1) \cdot \lfloor \frac{5}{3} \rfloor = 1 + 1 = 2.$

Линейным сравнением называется уравнение вида ax = b (mod n). Оно имеет решение тогда и только тогда, когда *b* делится на d = НОД(a, n). Если d > 1, то уравнение можно упростить, заменив его на a*x = b* (mod n*), где a* = a / d, b* = b / d, n* = n / d. После такого преобразования числа a* и n* являются взаимно простыми.

Алгоритм решения уравнения a*x = b* (mod n*) со взаимно простыми a* и n* состоит из двух частей:

1. Решаем уравнение a*x = 1 (mod n*). Для этого при помощи расширенного алгоритма Евклида ищем решение (x0, y0) уравнения a*x

+ n*y = 1. Взяв по модулю n* последнее равенство, получим a*x0 = 1 (mod n’).

2. Умножим на b* равенство a*x0 = 1 (mod n*). Получим a*(b*x0) = b* (mod n*), откуда решением исходного уравнения a*x = b* (mod n*) будет x = b*x0 (mod n*).

Лемма. Если НОД(k, m) = 1, то равенство ak = bk (mod m) эквивалентно a = b (mod m).

Доказательство. Из ak = bk (mod m) следует, что (a – b) · k делится на m. Но поскольку k и m взаимно просты, то a – b делится на m, то есть a = b (mod m).

Пример. Решить уравнение 18x = 6 (mod 8).

Значение d = НОД(18, 8) = 2 является делителем 6, поэтому решение существует. После упрощения получим уравнение: 9x = 3 (mod 4). Что согласно лемме эквивалентно 3x = 1 (mod 4). Решая уравнение 3x + 4y = 1 с помощью расширенного алгоритма Евклида, получим x = -1, y = 1. Откуда x = -1 (mod 4) = 3. То есть x = 3 будет как решением уравнения 3x = 1 (mod 4), так и 18x = 6 (mod 8).

Выполнять проверку числа на простоту требуется часто. Данная операция распространена и в задачах, и в алгоритмах. Небольшое число легко проверить на простоту – достаточно в цикле перебрать все числа от 2 до N – 1 и проверить, не делится ли N на них без остатка.

Но когда речь заходит об очень больших числах (содержащих в своем составе десятки разрядов), такой подход не годится: вычисления займут большое время (часы, сутки). В данном случае применяют специальные алгоритмы, называемые *тестами на простоту*. Рассмотрим один из таких тестов, он называется Тест Миллера – Рабина.

Тест *Миллера-Рабина* – вероятностный полиномиальный тест простоты. Тест Миллера-Рабина позволяет эффективно определять, является ли данное число составным. Однако, с его помощью нельзя

строго доказать простоту числа. Тем не менее, тест Миллера-Рабина часто используется в криптографии для получения больших случайных простых чисел.

Алгоритм был разработан Гари Миллером в 1976 году и модифицирован Майклом Рабином в 1980 году.

Пусть m – нечётное число большее 1. Число $m-1$ однозначно представляется в виде $m-1=2^s \cdot t$, где t нечётно.

Целое число a, $1 < a < m$, называется свидетелем простоты числа m, если выполняется одно из условий:

1. $a^t \equiv 1(mod\ m)$,
2. или существует целое число k, $0 \leq k < s$, такое, что $a^{2^k t} \equiv -1(mod\ m)$.

Теорема Рабина утверждает, что составное нечётное число m имеет не более $\frac{\varphi(m)}{4}$ различных свидетелей простоты, где $\varphi(m)$ – функция Эйлера. Алгоритм Миллера-Рабина параметризуется количеством раундов r. Рекомендуется брать r порядка величины $log_2 m$, где m – проверяемое число.

Для данного m находятся такие целое число s и целое нечётное число t, что $m-1=2^s \cdot t$. Выбирается случайное число a, $1 < a < m$. Если a не является свидетелем простоты числа m, то выдается ответ «m составное», и алгоритм завершается. Иначе, выбирается новое случайное число a, и процедура проверки повторяется. После нахождения r свидетелей простоты, выдается ответ «m, вероятно, простое», и алгоритм завершается.

Как и для теста Ферма, все числа $n > 1$, которые не проходят этот тест, – составные, а числа, которые проходят, могут быть простыми. И, что важно, для этого теста нет аналогов чисел Кармайкла.

В 1980 году было доказано, что вероятность ошибки теста Рабина-Миллера не превышает 1/4. Таким образом, применяя тест Рабина-Миллера раз для разных оснований, мы получаем вероятность ошибки 2^{-2t}.

Исходный код процедуры проверка числа n на простоту методом Миллера-Рабина (вероятностный тест) приведен ниже.

```
def miller_rabin_pass(a, s, d, n):
    a_to_power = pow(a, d, n)
    if a_to_power == 1:
        return True
    for i in xrange(s-1):
        if a_to_power == n - 1:
            return True
        a_to_power = (a_to_power * a_to_power) % n
    return a_to_power == n - 1
```

Реализация теста Миллера-Рабина на простоту числа на языке C# может быть составлена в виде (производится *k* раундов проверки числа n на простоту):

```
public bool MillerRabinTest(BigInteger n, int k)
{
    // если n == 2 или n == 3 - эти числа простые, возвращаем true
    if (n == 2 || n == 3)
        return true;

    // если n < 2 или n четное - возвращаем false
    if (n < 2 || n % 2 == 0)
        return false;
```

// представим n − 1 в виде (2^s)·t, где t нечётно, это можно сделать последовательным делением n - 1 на 2

```
        BigInteger t = n - 1;
        int s = 0;

        while (t % 2 == 0)
        {
            t /= 2;
            s += 1;
        }

        // повторить k раз
        for (int i = 0; i < k; i++)
        {
```
// выберем случайное целое число a в отрезке [2, n − 2]
```
            RNGCryptoServiceProvider rng = new
RNGCryptoServiceProvider();
            byte[] _a = new
byte[n.ToByteArray().LongLength];
            BigInteger a;
            do
            {
                rng.GetBytes(_a);
                a = new BigInteger(_a);
            }
            while (a < 2 || a >= n - 2);

            // x ← a^t mod n, вычислим с помощью возведения в
степень по модулю
            BigInteger x = BigInteger.ModPow(a, t, n);
```
// если x == 1 или x == n − 1, то перейти на следующую итерацию цикла
```
            if (x == 1 || x == n - 1)
                continue;
```
// повторить s − 1 раз

```
        for (int r = 1; r < s; r++)
        {
            // x ← x^2 mod n
            x = BigInteger.ModPow(x, 2, n);

            // если x == 1, то вернуть "составное"
            if (x == 1)
                return false;

      // если x == n − 1, то перейти на следующую итерацию внешнего цикла
            if (x == n - 1)
                break;
        }

        if (x != n - 1)
            return false;
    }

    // вернуть "вероятно простое"
    return true;
```

Постановка задачи

<u>При решении заданий запрещено использовать сторонние библиотеки!</u>

1. Составьте процедуру на языке C++, реализующую расширенный алгоритм Евклида.

2. Выполните поиск наибольшего общего делителя и двух сопутствующих коэффициентов для пары чисел в соответствии с индивидуальным вариантом. Проверьте корректность полученных чисел прямым подсчетом. Сделайте выводы.

3. Составьте процедуру на языке C++, реализующую алгоритм Миллера-Рабина.

4. Выполните проверку трех чисел (в соответствии с индивидуальным вариантом) на простоту с помощью реализованной процедуры. Проверьте корректность результата. Сделайте выводы.

Варианты индивидуальных заданий

Номер варианта	Пары чисел для нахождения НОД алгоритмом Евклида		Числа для проверки на простоту алгоритмом Миллера-Рабина		
1	12 и 240	420 и 25556243	3331	24569	23341
2	15 и 261	681 и 67921363	2833	65893	63973
3	11 и 314	314 и 62465258	3539	85447	15841
4	13 и 811	911 и 38697812	3449	22127	62765
5	11 и 566	571 и 29731656	3457	54599	41041
6	19 и 177	829 и 11254687	3319	95437	6601
7	13 и 289	937 и 34579524	3037	75411	10585
8	14 и 399	311 и 59771345	2467	54219	75361
9	16 и 482	103 и 40862554	2521	55239	29341
10	17 и 951	249 и 38914562	3407	75241	8911

Вопросы к отчету лабораторной работы

1. Понятие целых и натуральных чисел. Понятие делителей числа.
2. Понятие частного и остатка при целочисленном делении. Теорема о делении. Простые и составные числа.
3. Наибольший общий делитель двух чисел, способы его вычисления. Взаимно простые числа.
4. ϕ-функция Эйлера и ее свойства. Константа Эйлера.

5. Теорема о единственности разложения числа на простые множители.

6. Алгоритм Евклида: принцип работы и расчетная процедура.

7. Расширенный алгоритм Евклида: принцип работы и расчетная процедура.

8. Определение группы как алгебраической структуры. Понятие абелевой группы.

9. Модульная арифметика. Аддитивная и мультипликативная группы по модулю n. Модульные линейные уравнения.

10. Теорема Эйлера. Малая теорема Ферма. Понятие дискретного логарифма. Теорема о распределении простых чисел.

11. Алгоритм проверки чисел на простоту пробным делением. Псевдопростые числа. Числа Кармайкла.

12. Алгоритм Миллера-Рабина: принцип работы и расчетная процедура.

Лабораторная работа № 4. Изучение алгоритмов вычислительной геометрии

Цели работы:

1. Изучить сущность и принцип работы алгоритма определения взаимного расположения отрезков на плоскости.

2. Изучить сущность и принцип работы алгоритма построения выпуклой оболочки для заданного множества точек.

3. Изучить сущность и принцип работы алгоритма поиска наиближайшей друг к другу пары точек на плоскости.

4. Получить навыки реализации алгоритмов вычислительной геометрии на языке программирования C++.

Теоретическое введение

Выпуклое множество – такое множество точек, что, для любых двух точек множества все точки на отрезке между ними тоже принадлежат этому множеству.

Выпуклой оболочкой множества точек называется пересечение всех выпуклых множеств, содержащих все заданные точки. Выпуклая оболочка – такое выпуклое множество точек, что все точки фигуры также лежат в нем.

Минимальная выпуклая оболочка множества точек – это минимальная по площади выпуклая оболочка.

Для определения выпуклой оболочки множества точек часто пользуются алгоритмами Джарвиса и Грехэма.

Алгоритм *Джарвиса* («gift wrapping algorithm», «заворачивание подарка») является одним их наиболее простых алгоритмов построения выпуклой оболочки. Его идея заключается в последовательном построении оболочки. Сначала нам необходимо найти точку, которая должна быть в оболочке. Для этого можно взять самую нижнюю из самых левых точек. Далее будем каждый раз добавлять такую точку, чтобы угол образуемый следующей, текущей и предыдущей точками был максимален. Для этого совсем не обязательно находить сам угол. Мы можем использовать ориентированную площадь треугольника: пусть p_0 – текущая точка, p_n - уже выбранная следующая точка и p_i - текущая точка в переборе точек. Тогда если Sign Area Triangle (p_0, p_i, p_n) > 0, то точка p_i находится справа от вектора $\overrightarrow{p_0 p_n}$ и, значит, она составляет больший угол в выпуклой оболочке, поэтому мы ей дадим большее предпочтение, чем точке p_n (см. рисунок 9).

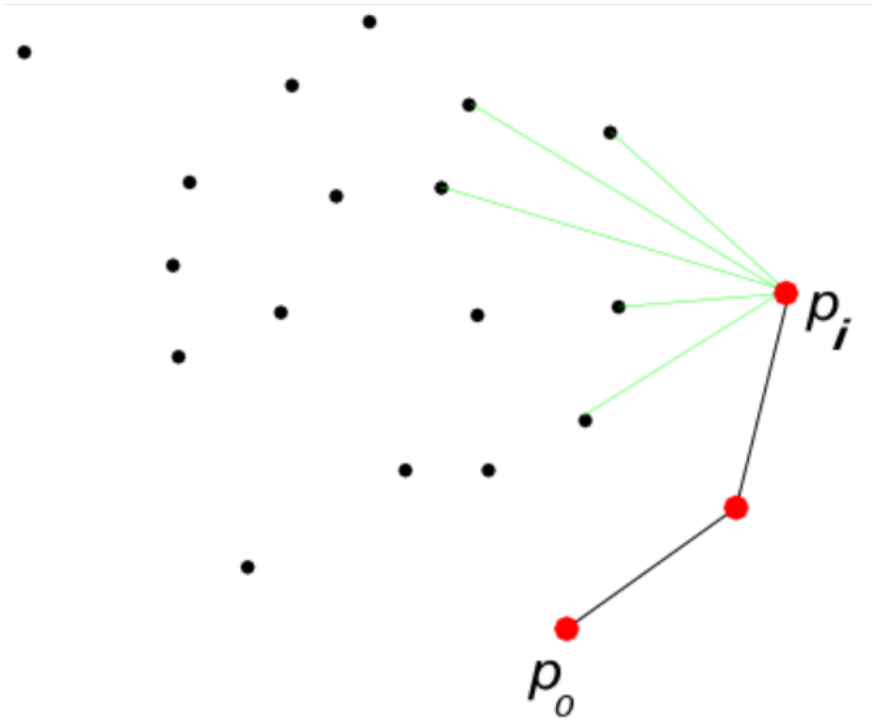


Рисунок 9 – Иллюстрация построения выпуклой оболочки методом Джарвиса

Также нам необходимо рассмотреть вырожденный случай – все три точки (p_0, p_i, p_n) лежат на одной прямой (ориентированная площадь треугольника равна 0). В таком случае нам нужно выбрать ту точку, которая расположена дальше от p0. Для этого тоже можно не использовать сложных вычислений, достаточно узнать, лежит ли pn в прямоугольнике с диагональю $\overrightarrow{p_0 p_i}$ (это делается простым сравнением координат точек). Если p_n лежит в этом прямоугольнике, то p_i лежит дальше от p_0, и мы точке p_i вследствии этого отдадим большее предпочтение, чем p_n.

Асимптотическая сложность алгоритма: $O(nk)$, где k – количество точек в оболочке, а n – количество заданных точек.

Программный код процедуры построения выпуклой оболочки по методу Джарвиса на языке C++:

```
    void hull_jarvis (vector <Point> p, vector <int> &ip)   {
    int n = p.size();
    int first, q, next;
    double sign;
    // находим самую нижнюю из самых левых точек
    first = 0;
    for (int                                  i = 1; i < n; ++i)
        if (p[i].x < p[first].x ||
           (p[i].x == p[first].x && p[i].y < p[first].y))
                                                    first = i;

    q = first; // текущая точка
    // добавляем точки в оболочку
    do
    {
        ip.push_back(q);
        next = q;
        // ищем следующую точку
        for (int                          i = n - 1; i >= 0; --i)
            if (p[i].x != p[q].x || p[i].y != p[q].y)       {
                sign = area_triangle (p[q], p[i], p[next]);
                if (next == q || sign > 0 ||
               (sign == 0 && point_in_box (p[next],
                    p[q], p[i])))
                                                      next = i;
            }
        q = next;
    }
    while (q != first);
}
```

Алгоритм *Грехэма* заключается в том, точки выпуклой оболочки ищутся последовательно, с использованием стека. Первым шагом определяем самую нижнюю-правую точку, пусть это – точка 0. Она наверняка лежит на оболочке (рисунок 10, шаг А). Следующая стадия – сортировка всех точек по углу между осью X и линией (точка 0). Точка с самым большим углом отправляется в стек за точкой 0 (шаг В).

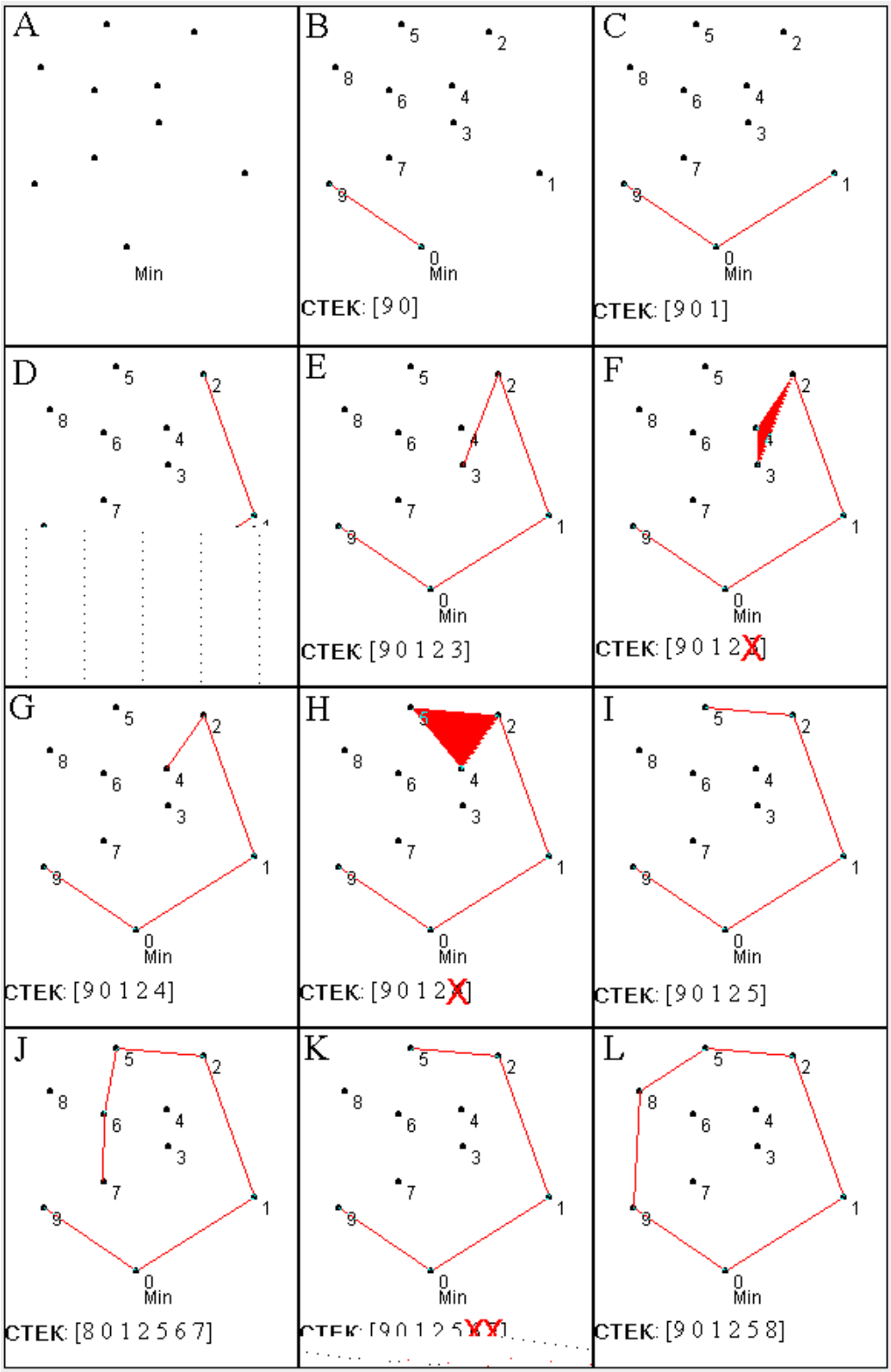


Рисунок 10 – Иллюстрация построения выпуклой оболочки методом Грехэма

Дальше все точки обрабатываются по очереди, начиная от самого меньшего угла (точка 1) и до тех пор, пока не достигнута точка 9. Процесс

состоит из теста, является ли строго левым поворот к новой точке на пути: верхняя – 1 точка стека – верхняя точка стека – тестируемая точка. Если это так, тогда она идет в стек, а мы переходим к следующей точке. Если нет – то убираем вершину стека и повторяем проверку с той же точкой. Это проиллюстрировано в шагах C-L. Иногда из стека приходится убирать несколько точек подряд, так как они последовательно не проходят проверки: обнаруживается правый поворот. Вообще говоря, в алгоритме еще нужна проверка на то, нет ли поворота обратно, и в этом случае точку следует оставить: мы имеем прямую.

Псевдокод процедуры Грехэма:

1. Найти нижнюю (правую) точку.

   Пометить ее p(0)

   Отсортировать все остальные точки по углу относительно p(0),

   Если одинаковый угол - по расстоянию от p(0).

   помечаем их p(1),...,p(n-1)

2. Заметим, что получившиеся точки образуют простой многоугольник, то есть ломаную без самопересечений.

Применяем «обход Грэхема»:

```
Стек S=(p(n-1),p(0))=(p(t-1),p(i)); t - индекс вершины
i = 1
while i < n do
  if p(i) строго слева от ( p(t-1),p(t) )
  then Push(S,i) and i = i + 1
  else Pop(S)
```

Обход Грэхема - основная часть метода, которая используется и в других алгоритмах. Он позволяет получить из простого многоугольника его выпуклую оболочку за $\Theta(n)$ операций.

Если многоугольник хранится в виде кольцевого списка точек, то более удобен следующий псевдокод:

```
v := НАЧАЛО кольца; w := ПРЕД(v); f:= false;
while (СЛЕД(v) =/= НАЧАЛО or f == false ) do {
   if (СЛЕД(v)==w) then f:= true;
   if (три точки v, СЛЕД(v), СЛЕД(СЛЕД(v))
      образуют левый поворот) then v:=СЛЕД(v);
   else {
      Удалить СЛЕД(v);
      v := ПРЕД(v);
   }
}
```

Функция, определяющая наличие левого поворота:

```
isLeft( coord *p0, coord *p1, coord *p2 ) {
    return (p1->x - p0->x)*(p2->y - p0->y) -
           (p2->x - p0->x)*(p1->y - p0->y);
}
```

При реализации необходимо правильно поддерживать случай удаления начальной вершины. Она обязательно должна проверяться в конце алгоритма (что может привести к многочисленным отходам назад с удалением), но при этом должно корректно проводиться сравнение в первом цикле while.

Множество точек прямой, проходящей через две точки с координатами (x1; y1) и (x2; y2), удовлетворяет уравнению:

$$(x–x1)*(y2–y1) – (y–y1)*(x2–x1) = 0.$$

Это значит, что если имеется точка с координатами (x0; y0) и (x0–x1)*(y2–y1) – (y0–y1)*(x2–x1) = 0, то эта точка лежит на прямой. В дальнейшем, вместо выражения (x–x1)*(y2–y1) – (y–y1)*(x2–x1) мы иногда будем использовать для краткости обозначение Ax + By + C или

f(x1, y1, x2, y2, x, y). Прямая Ax + By +C = 0, проходящая через две заданные точки с координатами (x1; y1) и (x2; y2),разбивает плоскость на две полуплоскости. Рассмотрим возможные значения выражения Ax + By +C.

1. Ax + By + C = 0 – определяет геометрическое место точек, лежащих на прямой. Запишем алгоритм для определения, лежит ли точка с координатами (x3; y3) на прямой, проходящей через точки (x1; y1) и (x2; y2). Переменная P – переменная логического типа, которая имеет значение «истина», если точка лежит на прямой, и «ложь» -- в противном случае.

P := «ложь»

если (x3–x1)*(y2–y1) – (y3–y1)*(x2–x1) = 0, то

P := «истина»

2. Ax + By + C > 0 – определяет геометрическое место точек, лежащих по одну сторону от прямой.

3. Ax + By + C < 0 – определяет геометрическое место точек, лежащих по другую сторону от прямой.

Это значит, что если для двух точек с координатами (x3; y3) и (x4; y4) значения выражений Ax3 + By3 + C и Ax4 + By4 + C имеют разные знаки, то эти точки лежат по разные стороны от прямой, проходящей через точки с координатами (x1; y1) и (x2; y2), а если одинаковые, то эти точки лежат по одну стороны от прямой. При этом число 0 имеет знак и «+» и «–». На рисунке 3 точки (x3; y3) и (x4; y4) лежат по одну сторону от прямой, точки (x3; y3) и (x5; y5) по разные стороны от прямой, а точка (x6; y6) лежит на прямой.

Пусть нам теперь необходимо определить взаимное расположение двух отрезков. Отрезки на плоскости заданы координатами своих концевых точек. Предположим, что концевые точки одного из отрезков имеют координаты (x1; y1) и (x2; y2), а концевые точки другого – (x3; y3)

и (x4; y4). Пусть общее уравнение первой прямой, проходящей через точки (x1;y1) и (x2;y2), имеет вид A1x + B1y + C1= 0, а второй прямой, проходящей через точки (x3;y3) и (x4;y4), A2x + B2y + C2= 0.

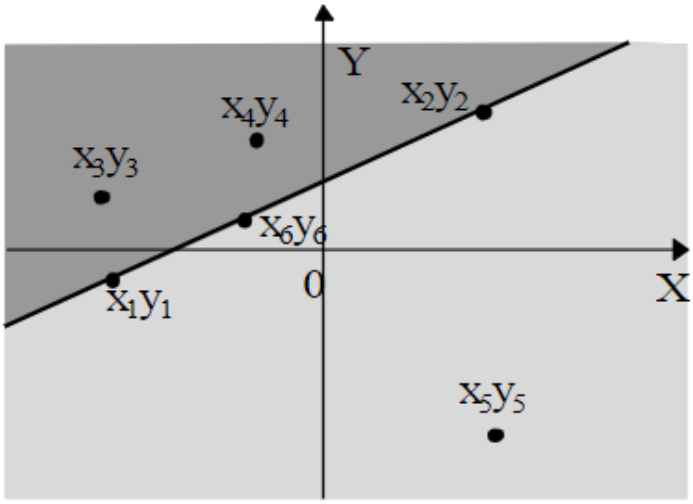


Рисунок 11 – Иллюстрация взаимного расположения прямой и точек

Определим расположение точек (x3; y3) и (x4; y4) относительно первой прямой. Если они расположены по одну сторону от прямой, то отрезки не могут пересекаться. Аналогично можно определить положение точек (x1; y1) и (x2; y2) относительно другой прямой. Таким образом, если значения пары выражений Z1 = A1x3 + B1y3 + C1 и Z2 = A1x4 + B1y4 + C1 имеют разные знаки, или Z1*Z2=0, а также пары Z3 = A2x1 + B2y1 + C2 и Z4 = A2x2 + B2y2 + C2 имеют разные знаки, или Z3*Z4=0, то отрезки пересекаются. Если же значения пар выражений Z1и Z2, или Z3и Z4, имеют одинаковые знаки, то отрезки не пересекаются. Различные случаи расположения отрезков показаны на рисунке 12.

На этом рисунке отрезки с концами в точках (x1; y1), (x2; y2) и (x4;y4), (x5; y5) пересекаются, отрезки с концами в точках (x1; y1), (x2;y2) и (x3; y3), (x4; y4) не пересекаются, а отрезки с концами в точках (x3; y3), (x4; y4) и (x4; y4) и (x5; y5) имеют общую вершину, что можно считать частным случаем пересечения.

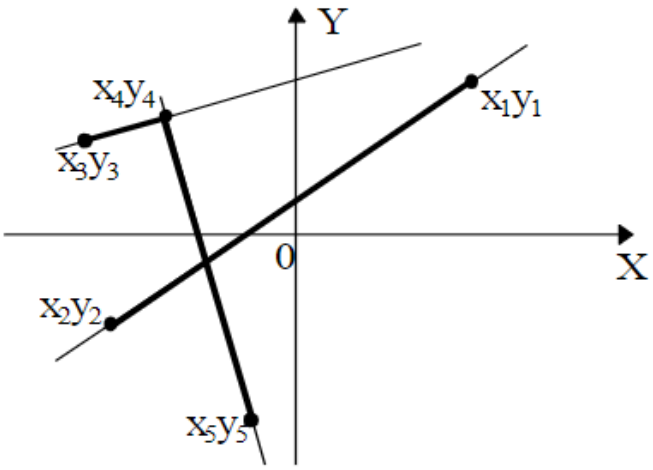


Рисунок 12 – Различные случаи взаимного расположения отрезков

Алгоритм для определения, пересекаются ли два отрезка с концами в точках (x1; y1), (x2; y2) и (x3; y3), (x4; y4), будет следующим:

P := «истина»

Z1 := (x3–x1)*(y2–y1) – (y3–y1)*(x2–x1)

Z2 := (x4–x1)*(y2–y1) – (y4–y1)*(x2–x1)

если Z1*Z2 > 0, то

P := «ложь»

Z3 := (x1–x3)*(y4–y3) – (y1–y3)*(x4–x3)

Z4 := (x2–x3)*(y4–y3) – (y2–y3)*(x4–x3)

если Z3*Z4 > 0, то

P:="ложь"

Приведенный фрагмент алгоритма не учитывает крайней ситуации, когда два отрезка лежат на одной прямой. В этом случае (x3–x1)*(y2–y1) – (y3–y1)*(x2–x1) = 0 и (x4–x1)*(y2–y1) – (y4–y1)* (x2–x1) = 0.

На рисунке 5а отрезки, лежащие на одной прямой, не пересекаются, а на рисунке 5б – пересекаются. Для того чтобы определить взаимное

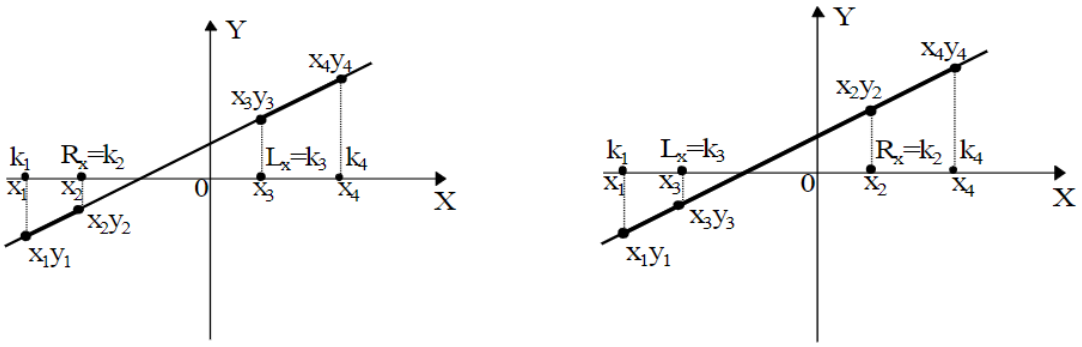
расположение таких отрезков, поступим следующим образом. Обозначим:

k1 = min (x1; x2);

k2 = max (x1; x2);

k3 = min(x3; x4);

k4 = max (x3; x4);



а) б)

Рисунок 13 – Иллюстрация пересекающихся (а) и непересекающихся (б) отрезков, лежащих на одной прямой

Здесь k1 является левой, а k2 – правой точкой проекции первого отрезка (отрезка, заданного координатами (x1; y1), (x2; y2)) на ось Ox. Аналогично, k3 является левой, а k4 – правой точкой проекции второго отрезка (отрезка, заданного координатами (x3; y3), (x4; y4)) на ось Ox. Аналогично, ищем проекции на ось OY. Отрезки, лежащие на одной прямой, будут пересекаться тогда, когда их проекции на каждую ось пересекаются (следует заметить, что если проекции двух произвольных отрезков пересекаются, то это не значит, что и сами отрезки пересекаются, что видно на рисунке 13).

Для определения взаимного расположения проекций на ось OX воспользуемся следующим фактом (см. рисунок 14): координата левой точки пересечения проекций Lx равна max(k1; k3), т. е. максимальной из

координат левых точек проекций. Рассуждая аналогично для правых точек проекций, получим, что координата правой точки Rx пересечения равна min(k2;k4).

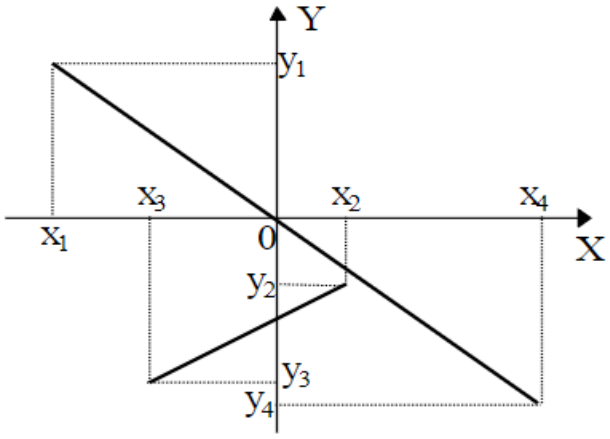


Рисунок 14 – Иллюстрация проекций отрезков

Для того чтобы отрезки пересекались, необходимо, чтобы левая координата пересечения проекций была не больше правой координаты пересечения отрезков (такой случай имеет место на рисунке 5а, когда Lx = x3, а Rx = x2). Поэтому, условием пересечения проекций является выполнение неравенства $L_x \leq R_x$. Аналогично, можно вычислить величины Ly и Ry, беря соответствующие проекции на ось Oy.

Постановка задачи

<u>При решении заданий запрещено использовать сторонние библиотеки!</u>

1. Составьте процедуру на языке C++, определяющую взаимное расположение отрезков на плоскости, заданных парой точек (в соответствии с индивидуальным вариантом). Визуализируйте расположение отрезков с помощью любого программного пакета визуализации данных или языка программирования (например, Python и Matplotlib).

2. Сгенерируйте случайным образом 10 точек, лежащих внутри прямоугольника 10×10. Составьте процедуру на языке C++, позволяющую определить выпуклую оболочку для данного множества точек одним из

способов (в соответствии с индивидуальным вариантом).

3. Составьте процедуру на языке C++, позволяющую определять пару точек из некоторого множества, расположенных ближе всего друг к другу (имеющую асимптотическое время работы $O(n\,logn)$). С помощью данной процедуры определите ближайшие точки на множестве, сгенерированном в п.2.

4. Визуализируйте расположение точек, пары ближайших точек и выпуклой оболочки, полученных в пп. 2-3, с помощью любого программного пакета визуализации данных или языка программирования (например, Python и Matplotlib).

5. Повторите пункты 2-4 данной работы, увеличив количество точек до 100.

6. Проанализируйте полученные результаты. В протокол лабораторной работы поместите исходные коды разработанных программ с комментариями и изображения, полученные в пп.4,5.

7. Сделайте выводы.

Варианты индивидуальных заданий

Номер варианта	Координаты концов отрезков $\underline{AB}$ и $\underline{CD}$				Алгоритм поиска выпуклой оболочки
	A	B	C	D	
1	(1, 1)	(8, 4)	(4, 3)	(5, 5)	Грэхема
2	(4, 1)	(2, 8)	(2, 5)	(6, 7)	Джарвиса
3	(8, 5)	(2, 1)	(4, 5)	(5, 3)	Джарвиса
4	(5, 8)	(9, 9)	(8, 8)	(3, 3)	Грэхема
5	(2, 5)	(1, 2)	(3, 8)	(4, 11)	Грэхема
6	(2, 7)	(5, 1)	(3, 5)	(2, 3)	Джарвиса
7	(2, 3)	(12, 5)	(4, 7)	(6, 4)	Джарвиса
8	(3, 4)	(2, 2)	(2, 3)	(6, 0)	Грэхема
9	(4, 10)	(3, 8)	(1, 4)	(5, 12)	Джарвиса
10	(6, 1)	(1, 6)	(6, 8)	(5, 3)	Грэхема

Вопросы к отчету лабораторной работы

1. Понятие отрезка и направленного отрезка. Конечные точки. Выпуклая комбинация точек.
2. Понятие векторного произведения, его смысл и правила вычисления.
3. Возможные варианты взаимного расположения отрезков на плоскости. Алгоритм определения взаимного расположения отрезков на плоскости.
4. Понятие выпуклой оболочки. Алгоритмы Грэхема и Джарвиса поиска выпуклой оболочки множества точек.
5. Алгоритм разбиения для поиска пары ближайших точек на плоскости из множества заданных.

Лабораторная работа № 5 Изучение алгоритмов нечеткого множества

Хотя это звучит как тавтология, но понятие «неопределенность» имеет неопределенный смысл без применения контекста, в котором решается задача. Основной смысл этого термина — неизвестность. Если уточнить принцип проявления неопределенности, то задача получает методы и алгоритмы решения. При этом неопределенность не исчезает, а формализуется. Некоторые приемы решения задач можно классифицировать по источнику появления неопределенности (см. рис. 15). Дли таких задач существуют теории и пути решения. В частности, при невозможности получить четкие исходные данные, которые поддаются измерению, это класс задач с физической неопределенностью. Если неопределенность кроется в знаниях, выраженных естественным или формальным языком, то это лингвистическая неопределенность. Класс задач, в которых значения слов четко не определены формируют задачи, решаемые с помощью нечеткого множества.

Теоретическое обоснование

Теория нечетких множеств описывает неточные категории, представления и знания, позволяет оперировать ими и делать на основе обработки информации заключения и выводы. Все это делает доступным создание информационных моделей на качественном, понятийном уровне и ведет к возможности организации обработки информации на основе применения методов нечеткого вывода. Программные системы на нечетких моделях позволяют реализовывать интеллектуальное управление неполными представлениями, а также извлекать новые знания.

Термин нечеткое множество определяется с помощью использования функции принадлежности. Данная функция задает характеристическую меру обычному множеству. Каждый его элемент оценивается значением меры принадлежности «привязанной» к функции лингвистическому

понятию. В общем смысле можно сказать, что набором значений можно задать отношение множества к рассматриваемому на качественном уровне понятию.

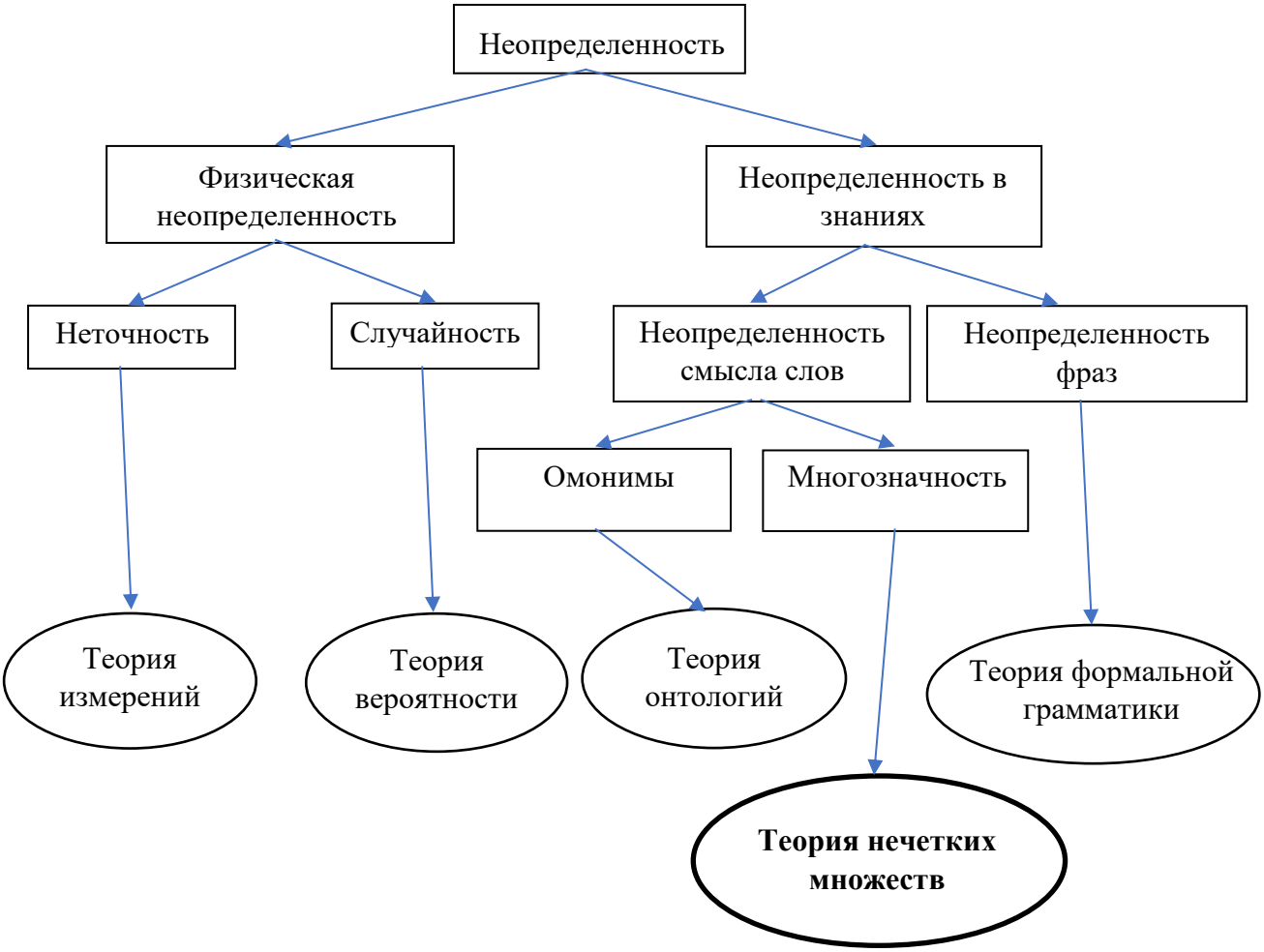


Рисунок 15 – Виды неопределенности

Характеристическая функция χ_A, определяющая множество A в полном пространстве X, представляет собой отображение, для которого X есть область определения, а $\{0,1\}$ (двузначное множество из 0 и 1) есть область значений:

$\chi_A : X \to \{0,1\}$ или

$$x \to \chi_A(x) = \begin{cases} 0, x \notin A, \\ 1, x \in A. \end{cases}$$

При этом $\chi_A(x) = 1$, если элемент x удовлетворяет свойствам A, и 0, если не удовлетворяет. Таким образом отношение к свойству A задается четко.

Нечеткое множество является расширением представленного множества. Функция принадлежности описывает нечеткое множество и принимает свои значения в интервале [0,1]. Такое представление отражает субъективную оценку степени принадлежности отдельных элементов базовой шкалы соответствующему нечеткому множеству:

$$A = (\mu_A(x), x), \ x \in X, \ \mu_A(x) \in [0,1],$$

где A - определяемое нечеткое множество; X - исходная базовая шкала (область определения); $\mu_A(x)$ - функция принадлежности.

Основные определения

Нечёткое множество — понятие, расширяющее классическое понятие множества, допускающее, что характеристическая функция (функция принадлежности элемента множеству) может принимать любые значения в интервале [0,1], а не только значения 0 или 1.

Определение: Под нечётким множеством A понимается совокупность $A = \{(x, \mu_A(x)) | x \in X\}$, где X— универсальное множество, а $\mu_A(x)$— функция принадлежности (характеристическая функция), характеризующая степень принадлежности элемента x нечёткому множеству A.

Функция $\mu_A(x)$ принимает значения в некотором вполне упорядоченном множестве M. Множество M называют множеством

принадлежностей, часто в качестве M выбирается отрезок $[0,1]$. Если $M=\{0,1\}$, то нечёткое множество может рассматриваться как обычное, чёткое множество.

Пример: рассмотрим, как с помощью нечеткого множества определить выражение "он еще молодой". Для наглядности приведем характеристическую функцию множества молодых людей.

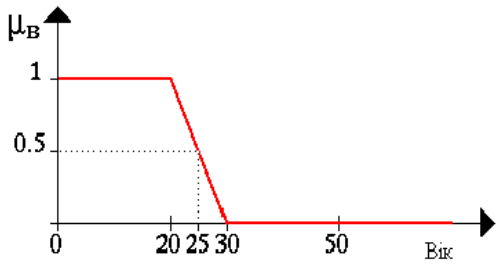


Рисунок 16 – Иллюстрация функции принадлежности

Пусть E = {x1, x2, x3, x4, x5 }, M = [0,1]; A - нечеткое множество, для которого μA(x1)=0,3; μA(x2)=0; μA(x3)=1; μA(x4)=0,5; μA(x5)=0,9. Тогда A можно представить в виде:

A = {0,3/x1; 0/x2; 1/x3; 0,5/x4; 0,9/x5 } или A = 0,3/x1 + 0/x2 + 1/x3 + 0,5/x4 + 0,9/x5, (знак "+" является операцией не сложения, а объединения) или

	x1	x 2	x 3	x4	x5
A =	0, 3	0	1	0, 5	0, 9

Основные характеристики нечетких множеств

Пусть A нечёткое множество с элементами из универсального множества Xи множеством принадлежностей $M=[0,1]$. Тогда

Носителем (суппортом) нечёткого множества $suppA$ называется множество $\{x|x \in X, \mu_A(x) > 0\}$.

1. Величина $\sup\limits_{x \in X} \mu_A(x) = \max\limits_{x \in X} \mu_A(x),$ называется **высотой** нечёткого множества A.

2. Нечёткое множество A **нормально**, если его высота равна 1.

3. Если высота строго меньше 1, нечёткое множество называется **субнормальным**.

4. Нечёткое множество **пусто**, если $\forall x \in X \ \mu_A(x) = 0$.

5. Непустое субнормальное нечёткое множество можно **нормализовать** по формуле: $\mu'_A(x) = \dfrac{\mu_A(x)}{\sup \mu_A(x)}$.

6. Нечёткое множество **унимодально**, если $\mu_A(x) = 1$ только на одном x из X.

7. Элементы $x \in X$, для которых $\mu_A(x) = 0,5$, называются **точками перехода** нечёткого множества A.

Операции над нечеткими множествами

Пусть A и B - нечеткие множества на универсальном множестве E. Говорят, что A содержится в B, если $\forall x \in E$ $\mu A(x) < \mu B(x)$. Обозначение: $A \subset B$.

Иногда используют термин "доминирование", то есть в случае, если $A \subset B$, говорят, что B доминирует A.

Равенство: A и B равны, если $\forall x \in E$ $\mu A(x) = \mu B(x)$. Обозначение: A = B.

Дополнение: Отрицанием множества A при $M = [0, 1]$ называется множество $\overline{A}$ с функцией принадлежности: $\mu_{\overline{A}}(x) = 1 - \mu_A(x)$, для каждого $x \in X$.

Пересечение: Пересечением нечётких множеств A и B называется наибольшее нечёткое подмножество, содержащееся одновременно в A и B:

$$\mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x)).$$

Объединение: Объединением нечётких множеств A и B называется наименьшее нечёткое подмножество, содержащее одновременно A и B:

$$\mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x)).$$

Разность: A - B = A$\cap \overline{B}$ с функцией принадлежности:

$$\mu A\text{-}B(x) = \mu A \cap \overline{B}(x) = \min(\mu A(x), 1 - \mu B(x)).$$

Дизъюнктивная сумма: A$\oplus$B = (A - B)$\cup$(B - A) = (A $\cap \overline{B}$) $\cup$($\overline{A} \cap$ B) с функцией принадлежности:

$$\mu A\text{-}B(x) = \max\{[\min\{\mu A(x), 1 - \mu B(x)\}];[\min\{1 - \mu A(x), \mu B(x)\}]\}.$$

Произведение: Произведением нечётких множеств A и B называется нечёткое подмножество с функцией принадлежности:

$$\mu_{AB}(x) = \mu_A(x)\mu_B(x).$$

Сумма: Суммой нечётких множеств A и B называется нечёткое подмножество с функцией принадлежности:

$$\mu_{AB}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x)\mu_B(x).$$

Примеры

Пусть:

A = 0,4/ x1 + 0,2/ x2+0/ x3+1/ x4;

B = 0,7/ x1+0,9/ x2+0,1/ x3+1/ x4;

C = 0,1/ x1+1/ x2+0,2/ x3+0,9/ x4.

Здесь:

1. $A \subset B$, то есть A содержится в B или B доминирует A, C несравнимо ни с A, ни с B, то есть пари {A, C} и {A, C} - пары недоминируемых нечетких множеств.

2. $A \neq B \neq C$.

3. $\overline{A}$ = 0,6/ x1 + 0,8/x2 + 1/x3 + 0/x4; $\overline{B}$ = 0,3/x1 + 0,1/x2 + 0,9/x3 + 0/x4.

4. $A \cap B$ = 0,4/x1 + 0,2/x2 + 0/x3 + 1/x4.

5. $A \cup C$ = 0,7/x1 + 0,9/x2 + 0,1/x3 + 1/x4.

6. A - B = $A \cap \overline{B}$ = 0,3/x1 + 0,1/x2 + 0/x3 + 0/x4;

B - A = $\overline{A} \cap B$ = 0,6/x1 + 0,8/x2 + 0,1/x3 + 0/x4.

7. $A \oplus B$ = 0,6/x1 + 0,8/x2 + 0,1/x3 + 0/x4.

Наглядное представление операций над нечеткими множествами

Для нечетких множеств можно применить визуальное представление. Рассмотрим прямоугольную систему координат, на оси ординат которой

откладываются значение μA(x), на оси абсцисс в произвольном порядке расположены элементы E. Если E по своей природе упорядочено, то этот порядок желательно сохранить в расположении элементов на оси абсцисс. Такое представление делает наглядными простые операции над нечеткими множествами.

Пусть A нечеткий интервал между 5 до 8 и B нечеткое число около 4, как показано на рисунке.

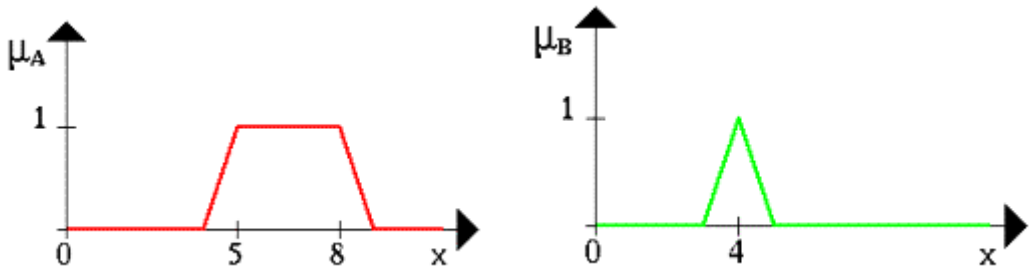


Рисунок 17 – Варианты задания функции принадлежности

Проиллюстрируем нечеткое множество между 5 и 8 И (AND) около 4 (синяя линия).

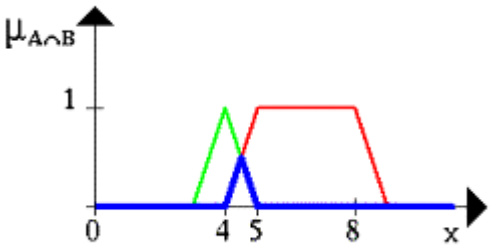


Рисунок 18 – Иллюстрация операции AND над нечеткими множествами

Нечеткое множество между 5 и 8 ИЛИ (OR) около 4 показано на следующем рисунке (снова синяя линия).

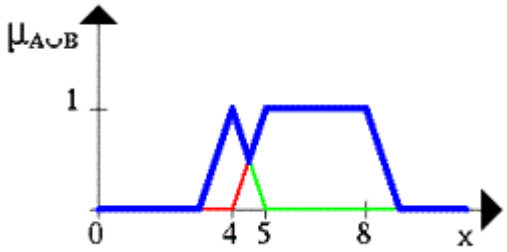


Рисунок 19 – Иллюстрация операции OR над нечеткими множествами

Следующий рисунок иллюстрирует операцию отрицания. Синяя линия - это ОТРИЦАНИЕ нечеткого множества A.

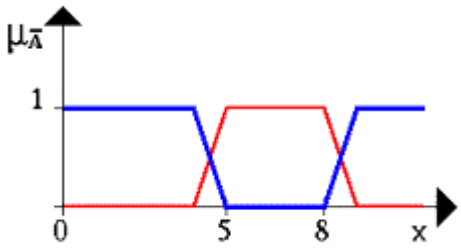
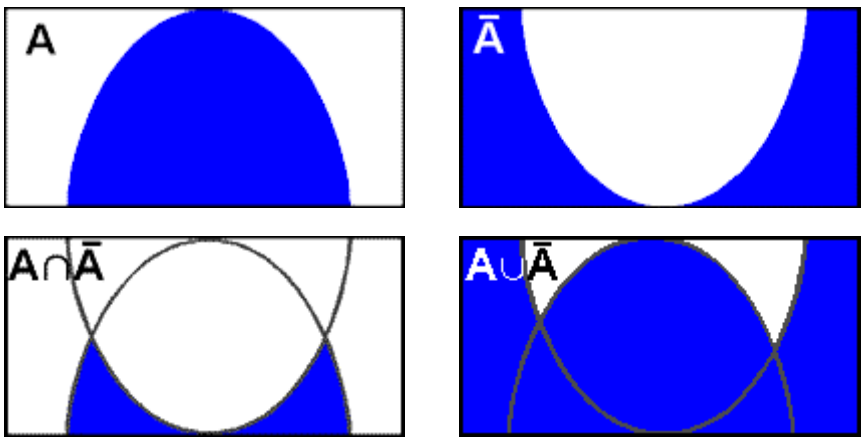


Рисунок 20 – Иллюстрация операции NOT над нечетким множеством

На следующем рисунке заштрихованная часть соответствует нечеткому множеству A и изображает область значений A и всех нечетких множеств, содержащихся в A. Остальные рисунки изображают соответственно $\overline{A}$, $A \cap \overline{A}$, $A \cup \overline{A}$.



Свойства операций □ i □

Пусть A, B, C - нечеткие множества, тогда выполняются следующие свойства:

- $\left.\begin{array}{l} A\cap B = B\cap A \\ A\cup B = B\cup A \end{array}\right\}$ - коммутативность;
- $\left.\begin{array}{l} (A\cap B)\cap C = A\cap (B\cap C) \\ (A\cap B)\cap C = A\cap (B\cap C) \end{array}\right\}$ - ассоциативность;
- $\left.\begin{array}{l} A\cap A = A \\ A\cup A = A \end{array}\right\}$ - идемпотентность;
- $\left.\begin{array}{l} A\cap (B\cup C) = (A\cap B)\cup (A\cap C) \\ A\cup (B\cap C) = (A\cup B)\cap (A\cup C) \end{array}\right\}$ - дистрибутивность;
- A∪∅ = A, где ∅ - пустое множество, то есть m∅(x) = 0 ∀x∈E;
- A∩∅ = ∅;
- A∩E = A, где E - универсальное множество;
- A∪E = E;
- $\left.\begin{array}{l} \overline{A\cap B} = \overline{A}\cup\overline{B} \\ \overline{A\cup B} = \overline{A}\cap\overline{B} \end{array}\right\}$ - теоремы де Моргана.

В отличие от четких множеств, для нечетких множеств в общем случае: $A\cap\overline{A}\neq\emptyset$, $A\cup\overline{A}\neq E$.

Умножение на число

Если α - положительное число, такое, что $\alpha \max_{x\in A} \mu A(x)\leq 1$, тогда нечеткое множество □A имеет функцию принадлежности: μαA(x) = αμA(x).

Сравнение нечётких множеств

Пусть A и B нечёткие множества, заданные на универсальном множестве X.

A содержится в B, если для любого элемента из X функция его принадлежности множеству A будет принимать значение меньшее либо равное, чем функция принадлежности множеству B:

$$A \subset B \Leftrightarrow \forall x \in X \ \mu_A(x) \leq \mu_B(x).$$

В случае, если условие $\mu_A(x) \leq \mu_B(x)$ выполняется не для всех $x \in X$, говорят о степени включения нечёткого множества A в B, которое определяется так: $l(A \subset B) = \min_{x \in T} \mu_B(x)$, где

$$T = \{x \in X; \mu_A(x) \leq \mu_B(x), \mu_A(x) > 0\}.$$

Два множества называются равными, если они содержатся друг в друге:

$$A = B \Leftrightarrow \forall x \in X \ \mu_A(x) = \mu_B(x).$$

В случае, если значения функций принадлежности $\mu_A(x)$ и $\mu_B(x)$ почти равны между собой, говорят о степени равенства нечётких множеств A и B, например, в виде

$$E(A = B) = 1 - \max_{x \in T} |\mu_A(x) - \mu_B(x)|,$$

где $T = \{x \in X; \mu_A(x) \neq \mu_B(x)\}$.

Свойства нечётких множеств

α-разрезом нечёткого множества $A \subseteq X$, обозначаемым как A_α, называется следующее чёткое множество:

$$A_\alpha = \{x \in X; \mu_A(x) \geq \alpha\},$$

то есть множество, определяемое следующей характеристической функцией (функцией принадлежности): $\chi_{A_\alpha}(x) = \begin{cases} 0, & \mu_A(x) < \alpha, \\ 1, & \mu_A(x) \geq \alpha. \end{cases}$

Для α-разреза нечёткого множества истинна импликация

$\alpha_1 < \alpha_2 \Rightarrow A_{\alpha_1} \supset A_{\alpha_2}$

Нечёткое множество $A \subseteq \mathbf{R}$ является выпуклым тогда и только тогда, когда выполняется условие

$$\mu_A[\gamma x_1 + (1-\gamma)x_2] \geq \langle \mu_A(x_1) \wedge \mu_A(x_2) = \min\{\mu_A(x_1), \mu_A(x_2)\} \rangle,$$

для любых $x_1, x_2 \in \mathbf{R}$ и $\gamma \in [0,1]$.

Нечёткое множество $A \subseteq \mathbf{R}$ является вогнутым тогда и только тогда, когда выполняется условие

$$\mu_A[\gamma x_1 + (1-\gamma)x_2] \leq \langle \mu_A(x_1) \vee \mu_A(x_2) = \max\{\mu_A(x_1), \mu_A(x_2)\} \rangle,$$

для любых $x_1, x_2 \in \mathbf{R}$ и $\gamma \in [0,1]$.

Индивидуальное задание

Постройте структуру данных для описания нечеткого множества и реализуйте алгоритмы для основных операций над нечёткими множествами. Реализованные функции проверьте на следующих данных.

№	A	x_1	x_2	x_3	x_4	x_5	B	x_1	x_2	x_3	x_4	x_5	α	β
1		0,1	0,2	0,6	0	1		0	0,5	0,2	0,1	1	0,6	0,4
2		0,8	0	0,7	0,2	1		0,6	0,4	0	0,3	0,8	0,4	0,3
3		0	0,6	0,4	1	0,1		1	0,8	1	0,6	0,3	0,8	0,7
4		0,9	0,1	0,6	0	0,5		0	0,1	0,4	0,2	1	0,3	0,6
5		0,5	0,1	0	1	0		0,4	0,9	0,3	0,2	0	0,7	0,8
6		0,9	0,3	1	0,3	0,5		1	0	1	0,4	0,7	0,5	0,6
7		0,1	0,5	0,3	0,4	1		0,5	0,2	0,6	0,7	0,3	0,9	0,5
8		0,2	0,5	0,4	0	0,3		0,1	0,7	0,6	1	0,3	0,3	0,4
9		0,3	0,1	0,3	0	1		0,5	1	0,4	0,7	0	0,2	0,9
10		0,2	0,6	0,1	0,3	0		1	0,8	0,9	0,1	0	0,4	0,7

Варианты:

1. Равенство
2. Дополнение
3. Пересечение
4. Объединение
5. Разность

6. Дизъюнктивная сумма
7. Произведение
8. Сумма

Рекомендуемая литература

1. Павлов, Л. А. Структуры и алгоритмы обработки данных : учебник для вузов / Л. А. Павлов, Н. В. Первова. — 3-е изд., стер. — Санкт-Петербург : Лань, 2021. — 256 с.
2. Вирсански, Э. Генетические алгоритмы на Python : руководство / Э. Вирсански ; перевод с английского А. А. Слинкина. — Москва : ДМК Пресс, 2020. — 286 с.
3. Гулаков, В. К. Структуры и алгоритмы обработки многомерных данных : монография / В. К. Гулаков, А. О. Трубаков, Е. О. Трубаков. — 2-е изд., стер. — Санкт-Петербург : Лань, 2021. — 356 с.
4. Нидхем, М. Графовые алгоритмы : руководство / М. Нидхем, Э. Холдер ; перевод с английского В. С. Яценкова. — Москва : ДМК Пресс, 2020. — 258 с.
5. Тюкачев, Н. А. C#. Алгоритмы и структуры данных : учебное пособие для вузов / Н. А. Тюкачев, В. Г. Хлебостроев. — 4-е изд., стер. — Санкт-Петербург : Лань, 2021. — 232 с.
6. Кораблин, Ю. П. Структуры и алгоритмы обработки данных : учебно-методическое пособие / Ю. П. Кораблин, В. П. Сыромятников, Л. А. Скворцова. — Москва : РТУ МИРЭА, 2020. — 219 с.
7. Сыромятников, В. П. Структуры и алгоритмы обработки данных: Практикум : учебное пособие / В. П. Сыромятников. — Москва : РТУ МИРЭА, 2020. — 244 с.

Учебное издание

Дмитрий Иванович Крыжановский
Павел Дмитриевич Кравченя
Андрей Евгеньевич Андреев
Дмитрий Викторович Завьялов
Михаил Андреевич Кузнецов

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ В СИСТЕМАХ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Учебное пособие

Волгоградский государственный технический университет.
400005, г. Волгоград, просп. В. И. Ленина, 28, корп. 1.