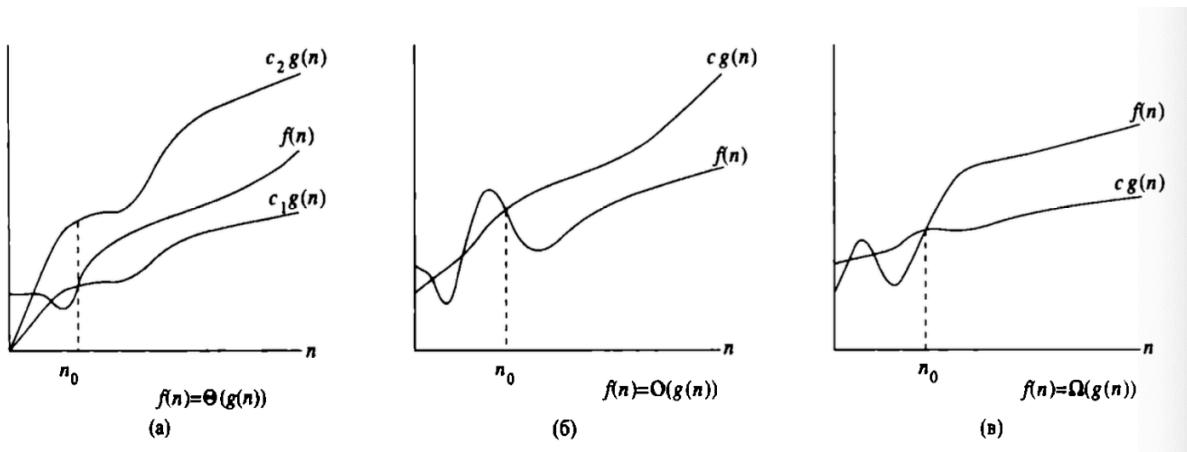


Алгоритмы

1. Анализ алгоритмов. Понятие о сложности по времени и по памяти. Асимптотика, О-символика. Доказательство корректности алгоритмов.

Рассматривая входные данные достаточно больших размеров для оценки только такой величины, как порядок роста времени работы алгоритма, мы тем самым изучаем *асимптотическую* эффективность алгоритмов. Это означает, что нас интересует только то, как время работы алгоритма растет с увеличением размера входных данных *в пределе*, когда этот размер увеличивается до бесконечности. Обычно алгоритм, более эффективный в асимптотическом смысле, будет более производительным для всех входных данных, за исключением очень маленьких.



Асимптотическая сложность: Асимптотическая сложность алгоритма описывает его поведение при стремлении размера входных данных к бесконечности. О-символика используется для обозначения асимптотической сложности (например, $O(n)$, $O(\log n)$, $O(n^2)$)

Обозначения:

Говорят, что функция $f(n)$ является асимптотически точной оценкой функции $g(n)$

Понятие о сложности по времени и по памяти: Сложность алгоритма определяет количество ресурсов (времени, памяти) необходимых для его выполнения. Временная сложность измеряется количеством элементарных операций, которые выполняет алгоритм. Пространственная сложность определяет объем памяти, который требуется для выполнения алгоритма.

Под свойством **правильности** понимается соответствие результатов работы алгоритма условию задачи.

Доказательство правильности алгоритма называется **верификацией**.

Доказательство корректности алгоритмов.

Строгое математическое доказательство правильности работы алгоритма - обычно очень трудная задача, главным образом из-за того, что трудно доказать правильность работы циклов и рекурсивных процедур. Вместе с тем демонстрация правильной работы алгоритмов на некотором наборе тестов еще не означает, что он всегда будет работать правильно. Следует помнить, что различных комбинаций входных данных бывает, как правило, бесконечно (или "практически" бесконечно) много. Поэтому необходимо сопровождать алгоритм некоторым рассуждением, которое, даже не будучи строгим доказательством, в достаточно полной мере убеждает нас в правильности алгоритма.

Конечно, оно не должно быть рассуждением в таком, например, стиле: "алгоритм перебирает все варианты, поэтому он правилен". Ведь тогда возникает вопрос, а как убедиться, что алгоритм действительно перебирает все варианты.

Пожалуй, наилучшим реальным подходом к обоснованию алгоритма является его правильность "по построению", когда используется метод пошаговой разработки. Чтобы получить правильный алгоритм, необходимо следить за правильностью детализации его шагов в ходе такого построения. Но это уже значительно более простая задача: как правило, детализация шага происходит в соответствии с определением того, что он должен делать.

При таком подходе построение алгоритма и его обоснование тесно переплетаются друг с другом. При этом следует, конечно, понимать, что если на этапе анализа задачи был выбран неверный подход к ее решению, то даже самая аккуратная последующая детализация исходной спецификации уже не позволит получить правильный алгоритм. Обоснование алгоритма будет выглядеть еще более убедительно, если его дополнить индивидуальными доказательствами, позволяющими убедиться в правильной работе хотя бы некоторых циклов.

Корректность алгоритма (на примере цикла) с помощью инварианта

Инвариантом называется логическое выражение, истинное после каждого прохода тела цикла (после выполнения фиксированного оператора) и перед началом выполнения цикла, зависящее от переменных, изменяющихся в теле цикла.

Инварианты используются в теории верификации программ для доказательства правильности выполнения цикла. Порядок доказательства работоспособности цикла с помощью инварианта сводится к следующему:

- 1.Доказывается, что выражение инварианта истинно перед началом цикла.
- 2.Доказывается, что выражение инварианта сохраняет свою истинность после выполнения тела цикла; таким образом, по индукции, доказывается, что по завершении цикла инвариант будет выполняться.
- 3.Доказывается, что при истинности инварианта после завершения цикла переменные примут именно те значения, которые требуется получить (это элементарно определяется из выражения инварианта и известных конечных значениях переменных, на которых основывается условие завершения цикла).
- 4.Доказывается (возможно — без применения инварианта), что цикл завершится, то есть условие завершения рано или поздно будет выполнено.
- 5.Истинность утверждений, доказанных на предыдущих этапах, однозначно свидетельствует о том, что цикл выполнится за конечное время и даст желаемый результат.

Также инварианты используют при проектировании и оптимизации циклических алгоритмов. Например, чтобы убедиться, что оптимизированный цикл остался корректным, достаточно доказать, что инвариант цикла не нарушен и условие завершения цикла достижимо.

Понятие инварианта также используется в объектно-ориентированном программировании для обозначения непротиворечивого состояния объекта. Подразумевается, что вызов любого метода оставляет объект в состоянии инварианта.

2. Строки и операции над ними. Представление строк. Вычисление длины, конкатенация. Алгоритмы поиска подстроки в строке.

Строковый тип — тип данных, значениями которого является произвольная последовательность (строка) символов алфавита. Каждая переменная такого типа (строковая переменная) может быть представлена фиксированным количеством байтов либо иметь произвольную длину.

Представление в памяти

Основные проблемы в машинном представлении строкового типа:

- строки могут иметь достаточно существенный размер (до нескольких десятков мегабайтов);
- изменяющийся со временем размер — возникают трудности с добавлением и удалением символов.

В представлении строк в памяти компьютера существует два принципиально разных подхода.

Представление массивом символов

В этом подходе строки представляются массивом символов; при этом размер массива хранится в отдельной (служебной) области.

Преимущества

- программа в каждый момент времени содержит сведения о размере строки, поэтому операции добавления символов в конец, копирования строки и собственно получения размера строки выполняются достаточно быстро;
- строка может содержать любые данные;
- возможно на программном уровне следить за выходом за границы строки при её обработке;
- возможно быстрое выполнение операции вида «взятие N-ого символа с конца строки».

Недостатки

- проблемы с хранением и обработкой символов произвольной длины;
- увеличение затрат на хранение строк — значение «длина строки» также занимает место и в случае большого количества строк маленького размера может существенно увеличить требования алгоритма к оперативной памяти;
- ограничение максимального размера строки. В современных языках программирования это ограничение скорее теоретическое, так как обычно размер строки хранится в 32-битовом поле, что даёт максимальный размер строки в 4 294 967 295 байт (4 гигабайта);
- при использовании алфавита с переменным размером символа (например, UTF-8), в размере хранится не количество символов, а именно размер строки в байтах, поэтому количество символов необходимо считать отдельно.

Метод «завершающего байта»

Второй метод заключается в использовании «завершающего байта». Одно из возможных значений символов алфавита (как правило, это символ с кодом 0) выбирается в качестве признака конца строки, и строка хранится как последовательность байтов от начала до конца. Есть системы, в которых в качестве признака конца строки используется не символ 0, а байт 0xFF (255) или код символа «\$».

Преимущества

- отсутствие дополнительной служебной информации о строке (кроме завершающего байта);
- возможность представления строки без создания отдельного типа данных;
- отсутствие ограничения на максимальный размер строки;
- экономное использование памяти;
- простота получения суффикса строки;
- простота передачи строк в функции (передаётся указатель на первый символ);

Недостатки

- долгое выполнение операций получения длины и конкатенации строк;
- отсутствие средств контроля за выходом за пределы строки, в случае повреждения завершающего байта возможность повреждения больших областей памяти, что может привести к непредсказуемым последствиям — потере данных, краху программы и даже всей системы;
- невозможность использовать символ завершающего байта в качестве элемента строки.

- невозможность использовать некоторые кодировки с размером символа в несколько байт (например, UTF-16), так как во многих таких символах, например Ä (0x0100), один из байтов равен нулю (в то же время, кодировка UTF-8 свободна от этого недостатка).

Представление в виде списка

Языки Erlang, Haskell, Пролог используют для строкового типа список символов. Этот метод делает язык более «теоретически элегантным» за счёт соблюдения ортогональности в системе типов, но приносит существенные потери быстродействия.

Вычисление длины строки в программировании может быть реализовано по-разному в зависимости от представления строк. Ниже приведены общие методы для вычисления длины строки в различных языках программирования:

1. Представление строк как массив символов:

- Если строка представлена как массив символов, то можно использовать следующий подход для вычисления длины строки:

- C/C++:

```
#include <stdio.h>
#include <string.h>

int main() {
    char str[] = "Hello, World!";
    int length = strlen(str);
    printf("Length of the string: %d\n", length);
    return 0;
}
```

2. Представление строк как объектов строк:

- Если строки представлены как объекты строк, то обычно язык программирования предоставляет специальные методы для вычисления длины строки:

- Python:

```
str = "Hello, World!"
length = len(str)
print("Length of the string:", length)
```

3. Использование функций/методов языка программирования:

- Многие языки программирования предоставляют встроенные функции или методы для вычисления длины строки. Например, в Python есть функция `len()`, в Java метод `length()`, в C/C++ функция `strlen()`, и т.д.

Конкатенация

Операция конкатенации определяется для типов данных, имеющих структуру последовательности (список, очередь, массив и ряд других). В общем случае, результатом конкатенации двух объектов A и B является объект C = A · B, полученный поочерёдным добавлением всех элементов объекта B, начиная с первого, в конец объекта A.

Из соображений удобства и эффективности различают две формы операции конкатенации:

- Модифицирующая конкатенация. Результат операции формируется в левом операнде.
- Немодифицирующая конкатенация. Результатом является новый объект, операнды остаются неизменными.

Алгоритмы поиска подстроки в строке.

Прямой поиск (Brute Force):

1. Идея алгоритма:

- Простой алгоритм, который проверяет каждую возможную подстроку в строке на совпадение с искомой подстрокой.

2. Шаги работы алгоритма:

- Проход по строке и сравнение каждой возможной подстроки с искомой подстрокой.
- Если найдено совпадение, возвращается индекс начала найденной подстроки.
- Если не найдено совпадение, продолжается поиск до конца строки.

3. Пример:

Поиск abc в строке ababcaa

1. Начнем сравнивать подстроку "abc" с каждой возможной подстрокой длины 3 в строке "ababcaa".

2. Сравниваем "abc" с "aba". Не совпадает.
3. Сравниваем "abc" с "bab". Не совпадает.
4. Сравниваем "abc" с "aba". Не совпадает.
5. Сравниваем "abc" с "bac". Не совпадает.
6. Сравниваем "abc" с "aca". Не совпадает.
7. Сравниваем "abc" с "caa". Не совпадает.
8. Подстрока "abc" не найдена в строке "ababcaa".

4. Временная сложность:

$O(n * m)$, где n - длина строки, m - длина искомой подстроки.

Алгоритм Кнута-Морриса-Пратта (KMP):

1. Идея алгоритма:

- Эффективный алгоритм для поиска подстроки, использующий информацию о совпадениях предыдущих символов для оптимизации поиска.

2. Шаги работы алгоритма:

- Предварительная обработка искомой подстроки для создания префиксной функции (LPS - Longest Proper Prefix which is also Suffix).
- Использование префиксной функции для оптимизации поиска и избегания повторных сравнений.
- При совпадении символов, сдвигаем позицию для продолжения поиска.

Префикс-функция строки $\pi(S, i)$ – это длина наибольшего префикса строки $S[1..i]$, который не совпадает с этой строкой и одновременно является ее суффиксом. Проще говоря, это длина наиболее длинного начала строки, являющегося также и ее концом. Для строки S удобно представлять префикс функцию в виде вектора длиной $|S|-1$. Можно рассматривать префикс-функцию длины $|S|$, положив $\pi(S, 1)=0$. Пример префикс функции для строки «abcdabcabcdabcdab»:

S[i]	a	b	c	d	a	b	c	a	b	c	d	a	b	c	d	a	b
$\pi(S, i)$	0	0	0	0	1	2	3	1	2	3	4	5	6	7	4	5	6

Предположим, что $\pi(S, i)=k$. Отметим следующие свойства префикс-функции.

1. Если $S[i+1]=S[k+1]$, то $\pi(S, i+1)=k+1$.
2. $S[1..\pi(S, k)]$ является суффиксом строки $S[1..i]$. Действительно, если строка $S[1..i]$ оканчивается строкой $S[1..\pi(S, i)]=S[1..k]$, а строка $S[1..k]$ оканчивается строкой $S[1..\pi(S, k)]$, то и строка $S[1..i]$ оканчивается строкой $S[1..\pi(S, k)]$.

3. $\forall j \in (k, i)$, $S[1..j]$ не является суффиксом строки $S[1..i]$. В противном случае было бы неверным предположение $\pi(S, i) = k$, так как $j > k$.

Рассмотренные свойства позволяют получить алгоритм вычисления префикс-функции.

Пусть $\pi(S, i) = k$. Необходимо вычислить $\pi(S, i+1)$.

1. Если $S[i+1] = S[k+1]$, то $\pi(S, i+1) = k+1$.
2. Иначе, если $k=0$, то $\pi(S, i+1)=0$.
3. Иначе положить $k := \pi(S, i)$ и перейти к шагу 1.

Ключевым моментом для понимания сути алгоритма является тот факт, что если найденный на предыдущем шаге суффикс не может быть расширен на следующую позицию, то мы пытаемся рассматривать меньшие суффиксы до тех пор, пока это возможно.

Алгоритм вычисления префикс-функции на языке Python:

```
def prefix(s)
    v = [0]*len(s)
    i = 1
    for i in xrange(1, len(s)):
        k = v[i-1]
        while k > 0 and s[k] <> s[i]:
            k = v[k-1]
        if s[k] == s[i]:
            k = k + 1
        v[i] = k
    return v
```

Покажем, что время работы алгоритма составляет $O(n)$, где $n=|S|$. Заметим, что асимптотику алгоритма определяет итоговое количество итераций цикла while. Это так, поскольку без учета цикла while каждая итерация цикла for выполняется за время, не превышающее константу. На каждой итерации цикла for k увеличивается не более чем на единицу, значит максимально возможное значение $k=n-1$. Поскольку внутри цикла while значение k лишь уменьшается, получается, что k не может суммарно уменьшиться больше, чем $n-1$ раз. Значит цикл while в итоге выполнится не более n раз, что дает итоговую оценку времени алгоритма $O(n)$.

Рассмотрим алгоритм Кнута-Морриса-Пратта, основанный на использовании префикс-функции. Как и в примитивном алгоритме поиска подстроки, образец «перемещается» по строке слева направо с целью обнаружения совпадения. Однако ключевым отличием является то, что при помощи префикс-функции мы можем избежать заведомо бесполезных сдвигов.

Пусть $S[0..m-1]$ – образец, $T[0..n-1]$ – строка, в которой ведется поиск. Рассмотрим сравнение строк на позиции i , то есть образец $S[0..m-1]$ сопоставляется с частью строки $T[i..i+m-1]$. Предположим, первое несовпадение произошло между символами $S[j]$ и $T[i+j]$, где $i < j < m$. Обозначим $P = S[0..j-1] = T[i..i+j-1]$. При сдвиге можно ожидать, что префикс S сойдется с каким-либо суффиксом строки P . Поскольку длина наиболее длинного префикса, являющегося одновременно суффиксом, есть префикс-функция от строки S для индекса j , приходим к следующему алгоритму:

1. Построить префикс-функцию образца S , обозначим ее F .
2. Положить $k = 0$, $i = 0$.
3. Сравнить символы $S[k]$ и $T[i]$. Если символы равны, увеличить k на 1. Если при этом k стало равно длине образца, то вхождение образца S в строку T найдено, индекс вхождения равен $i - |S| + 1$. Алгоритм завершается. Если символы не равны, используем префикс-функцию для оптимизации сдвигов. Пока $k > 0$, присвоим $k = F[k-1]$ и перейдем в начало шага 3.
4. Пока $i < |T|$, увеличиваем i на 1 и переходим в шаг 3.

Возможная реализация алгоритма Кнута-Морриса-Пратта на языке Python выглядит так:

```
def kmp(s,t)
    index = -1
    f = prefix(s)
    k = 0
    for i in xrange(len(t)):
        while k > 0 and s[k] <> t[i]:
            k = f[k-1]
        if s[k] == t[i]:
            k = k + 1
        if k == len(s):
            index = i - len(s) + 1
            break
    return index
```

Временная сложность:

$O(n + m)$, где n - длина строки, m - длина искомой подстроки.

Алгоритм Рабина-Карпа:

1. Идея алгоритма:

- Алгоритм, который использует хэширование для быстрого поиска подстроки.

2. Шаги работы алгоритма:

- Вычисление хеш-значения искомой подстроки и хеш-значений всех возможных подстрок в строке.
- Сравнение хеш-значений для быстрого определения потенциальных совпадений.
- Дополнительная проверка символов в случае коллизий хеш-значений.

Устройство алгоритма

Вообще в классической реализации используется полиномиальный хеш, но подойдет в общем-то любая кольцевая хеш-функция. Но мы возьмем полиномиальную версию.

Алгоритм крайне простой: берем хеш паттерна, берем хеш куска текста равного по длине, сравниваем их. Если равны, то проверяем их посимвольно, в противном случае — идем дальше.

a ⁵	a ¹⁰	b ²¹	a	b	a	b
b ²¹	a	b	a	b		
	b ²¹	a	b	a	b	
		b ²¹	a	b	a	b

Что это все значит

У нас есть двоичный алфавит, где $a = 0$, $b = 1$

Вычислим хеш паттерна:

$$H(P) = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 21$$

Вычислим хеш подстроки $T[0, 4]$.

$$H(T[0, 4]) = 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 5$$

Они не совпадают, значитдвигаем паттерн на символ правее.

Вычислим хеш подстроки $T[1, 5]$.

$$H(T[1, 5]) = (H(T[0, 4]) - 0 * 2^4) * 2 + 0 * 2^0 = 10$$

Хеши подстроки и паттерна снова не совпадают, значитдвигаем паттерн еще на символ правее.

Вычислим хеш подстроки $T[2, 6]$.

$$H(T[2, 6]) = (H(T[1, 5]) - 0 * 2^4) * 2 + 1 * 2^0 = 21$$

Хеши совпали, значит проверяем этот участок посимвольно.

```
1 function RabinKarp(string s[1..n], string sub[1..m])
2     hsub := hash(sub[1..m])
3     hs := hash(s[1..m])
4     for i from 1 to (n-m+1)
5         if hs = hsub
6             if s[i..i+m-1] = sub
7                 return i
8             hs := hash(s[i+1..i+m])
9     return not found
```

Временная сложность:

$O(n + m)$, где n - длина строки, m - длина искомой подстроки.

3. Сортировки. Нижняя теоретико-информационная оценка сложности задачи сортировки. Алгоритмы сортировки вставками, пузырьком, быстрая сортировка, сортировка слиянием. Оценка сложности.

Информационная энтропия – это мера неопределенности или "беспорядка" в системе. Введенная в теории информации, информационная энтропия описывает количество информации, необходимое для описания состояния системы или источника данных.

Информационная энтропия измеряется в битах или натуральных единицах информации (например, нат), и чем выше энтропия, тем больше неопределенность в системе. Если все возможные состояния системы равновероятны, то информационная энтропия будет максимальной.

Информационная энтропия также связана с тем, сколько информации несет каждое новое сообщение или символ из источника данных. В случае сортировки, информационная

энтропия может быть использована для определения нижней границы сложности задачи сортировки.

Нижняя теоретико-информационная оценка сложности.

1 способ:

Предположим, что у нас есть n элементов, которые мы хотим отсортировать. Пусть каждый элемент имеет k возможных значений. Тогда общее количество возможных перестановок этих элементов будет равно k^n .

Информационная энтропия H равна логарифму от общего числа возможных перестановок:

$$H = \log(k^n) = n \cdot \log(k).$$

Теперь предположим, что у нас есть алгоритм сортировки, который может правильно упорядочить все n элементов. Если каждая перестановка элементов равновероятна (то есть каждая перестановка имеет вероятность $1/k^n$), то минимальное количество информации, необходимое для определения правильной перестановки, будет равно информационной энтропии H .

Это означает, что сложность любого алгоритма сортировки не может быть меньше $n \cdot \log(k)$, так как это минимальное количество информации, необходимое для правильной сортировки элементов.

Таким образом, нижняя теоретико-информационная оценка сложности задачи сортировки составляет $n \cdot \log(k)$.

2 способ:

Для любого алгоритма сортировки сравнениями можно создать дерево, где операции сравнения элементов соответствуют узлам, переходы между состояниями - ребрам, а конечные перестановки элементов - листьям. Необходимо доказать, что высота такого дерева для любого алгоритма сортировки сравнениями не может быть менее, чем $\Omega(n \log n)$, где n - количество элементов.

Для сортировки перестановок n элементов при сравнении двух из них есть два возможных исхода ($a_i \leq a_j$ и $a_i > a_j$), поэтому каждый узел дерева имеет не более двух сыновей. Поскольку существует $n!$ различных перестановок n элементов, количество листьев в дереве должно быть не менее $n!$ (иначе некоторые перестановки были бы недостижимы из корня).

Докажем, что двоичное дерево с не менее чем $n!$ листьями имеет глубину $\Omega(n \log n)$. Легко установить, что двоичное дерево высоты h имеет не более чем 2^h листьев. Таким образом, получаем неравенство: $n! \leq l \leq 2^h$, где l - число листьев. Прологарифмируем это неравенство и получим:

$$h \geq \log_2 n! = \log_2 1 + \log_2 2 + \dots + \log_2 n > \frac{n}{2} \log_2 \left(\frac{n}{2} \right) = \frac{n}{2} (\log_2 n - 1) = \Omega(n \log n)$$

Таким образом, для любого алгоритма сортировки сравнениями существует перестановка, на которой будет выполнено $\Omega(n \log n)$ сравнений.

Остальная информация находится в другом файле.

4. Представление матриц и векторов. Алгоритмы умножения матриц и эффективные способы их реализации. Численные методы решения систем линейных уравнений.

Представление матриц и векторов

Матрица – двумерный массив
Вектор – одномерный массив

Алгоритмы умножения матриц и эффективные способы их реализации

Умножение матриц определено только для матриц, у которых количество столбцов первой матрицы равно количеству строк второй матрицы. Результатом умножения матриц будет новая матрица, размеры которой равны количеству строк первой матрицы и количеству столбцов второй матрицы.

1) Итеративный

По определению умножения матриц для $n \times m$ матрицы A и по $m \times p$ матрицы B произведением $C = AB$ является $n \times p$ матрица, состоящая из элементов

$$c_{ij} = \sum_{k=1}^m a_{ik} b_{kj}$$

Отсюда можно построить простой алгоритм путём организации циклов по индексу i от 1 до n и j от 1 до p , осуществляя вычисления по вышеприведённой формуле с помощью вложенных циклов:

- Вход: матрицы A и B
- Пусть C будет новой матрицей нужного размера
- Для i от 1 до n :
 - Для j от 1 до p :
 - Положим $sum = 0$
 - Для k от 1 до m :
 - Положим $sum \leftarrow sum + A_{ik} \times B_{kj}$
 - Положим $C_{ij} \leftarrow sum$
 - Возвращаем C

Этот алгоритм работает за время $\Theta(nmp)$ (в асимптотических обозначениях). Обычно для упрощения анализа алгоритма предполагается, что входными матрицами являются квадратные матрицы размера $n \times n$, и в этом случае время работы составляет $\Theta(n^3)$, то есть время зависит кубически от размера матриц.

2) Алгоритм разделяй-и-властвуй

Альтернативой итерационному алгоритму для умножения матриц является алгоритм разделяй-и-властвуй. Он опирается на разложение на блоки

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},$$

что работает для всех квадратных матриц с размерностями, равными степеням двойки, то есть $2^n \times 2^n$ для некоторого n . Произведение матриц тогда равно что составляет восемь умножений пар подматриц с последующим шагом сложения.

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Алгоритм разделяй-и-властвуй вычисляет элементы рекурсивно с помощью скалярного произведения $c_{11} = a_{11}b_{11}$ как в базовом случае.

Сложность этого алгоритма $\Theta(n^3)$, ту же самую сложность, что и для итеративного алгоритма[3].

Вариант этого алгоритма, который работает для матриц произвольного размера и на практике быстрее[4], разбивает матрицы на две, а не на четыре подматрицы.

3) Алгоритм Штрассена

Если добавить к матрицам A и B одинаковые нулевые строки и столбцы, их произведение станет равно матрице AB с теми же добавленными строками и столбцами. Поэтому можно рассматривать только матрицы размера $n = 2^k$, $k \in \mathbb{N}$, а другие случаи сводить к этому добавлению нулей, отчего n может увеличиться лишь вдвое.

Пусть A, B – матрицы размера $2^k \times 2^k$. Их можно представить как **блочные матрицы** размера (2×2) из $(2^{k-1} \times 2^{k-1})$ -матриц:

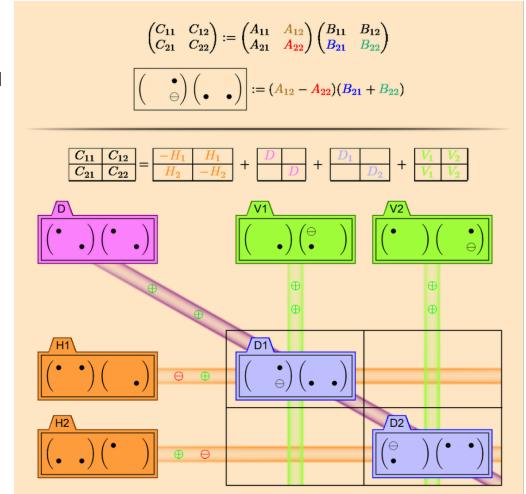
$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

По принципу **блочного умножения**, матрица AB выражается через их произведение

$$AB = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix},$$

где в правой части происходит восемь умножений матриц размера $2^{k-1} \times 2^{k-1}$. Поскольку матрицы образуют **кольцо**, то для вычисления правой части годится любой алгоритм умножения (2×2) -матриц, использующий лишь сложения, вычитания и умножения. Штрассен предложил такой алгоритм с семью умножениями:

$$\begin{aligned} D &= (A_{11} + A_{22})(B_{11} + B_{22}); \\ D_1 &= (A_{12} - A_{22})(B_{21} + B_{22}); \\ D_2 &= (A_{21} - A_{11})(B_{11} + B_{12}); \\ H_1 &= (A_{11} + A_{12})B_{22}; \\ H_2 &= (A_{21} + A_{22})B_{11}; \\ V_1 &= A_{22}(B_{21} - B_{11}); \\ V_2 &= A_{11}(B_{12} - B_{22}); \end{aligned}$$



$$\begin{aligned} AB &= \begin{pmatrix} D & 0 \\ 0 & D \end{pmatrix} + \begin{pmatrix} D_1 & 0 \\ 0 & D_2 \end{pmatrix} + \begin{pmatrix} -H_1 & H_1 \\ H_2 & -H_2 \end{pmatrix} + \begin{pmatrix} V_1 & V_2 \\ V_1 & V_2 \end{pmatrix} \\ &= \begin{pmatrix} D + D_1 + V_1 - H_1 & V_2 + H_1 \\ V_1 + H_2 & D + D_2 + V_2 - H_2 \end{pmatrix}. \end{aligned}$$

Тогда время работы алгоритма:

$$T(n) = 7T(n/2) + O(n^2) = O(n^{\log_2 7}) .$$

4) Алгоритм Кэннона

Алгоритм Кэннона^[en], известный также как алгоритм 2D, — это алгоритм, предотвращающий обмен данными^[en], который превращает каждую входную матрицу в блочную матрицу, элементами которой являются подматрицы размера $\sqrt{\frac{M}{3}} \times \sqrt{\frac{M}{3}}$, где M является размером быстрой памяти^[17]. Затем используется наивный алгоритм над блоками матриц, вычисляющий произведение подматриц полностью в быстрой памяти. Это сокращает полосу частот канала связи до $O(\frac{n^3}{\sqrt{M}})$, что асимптотически оптимально (для алгоритмов, выполняющих $\Omega(n^3)$ операций)^{[18][19]}.

Численные методы решения систем линейных уравнений.

Методы решения систем линейных уравнений делятся на две группы:

- **прямые,**
- **итерационные.**

Прямые методы используют конечные соотношения (формулы) для вычисления неизвестных.

Они дают решение после выполнения заранее известного числа операций.

Они сравнительно просты и наиболее универсальны (т.е. пригодны для решения широкого класса линейных систем).

Недостатки прямых методов:

- необходимость хранения в оперативной памяти компьютера сразу всей матрицы (при большой размерности матрицы требуется большого места в памяти);
- накапливание погрешностей в процессе решения; это особенно опасно для больших систем, а также для плохо обусловленных систем, весьма чувствительных к погрешностям

Итерационные методы – это методы последовательных приближений.

В них необходимо задать некоторое приближённое решение – начальное приближение. После этого с помощью некоторого алгоритма проводится один цикл вычислений, называемый итерацией.

В результате итерации находят новое приближение.

Итерации проводятся до получения решения с требуемой точностью.

Преимущества итерационных методов:

- требуют хранения в памяти машины не всей матрицы системы, а лишь нескольких векторов с n компонентами;
- Погрешности окончательных результатов при использовании итерационных методов не накапливаются, поскольку точность вычислений в каждой итерации определяется результатами предыдущей итерации и практически не зависит от ранее выполненных вычислений.

Прямые методы: метод Гаусса

Основан на приведении матрицы системы к треугольному виду.

Это достигается последовательным исключением неизвестных из уравнения системы.

Прямой ход:

- с помощью первого уравнения исключается x_1 из всех последующих уравнений системы;
- с помощью второго уравнения исключается x_2 из третьего и всех последующих уравнений, преобразованных на предыдущем шаге.

Этот процесс продолжается до тех пор, пока в левой части последнего (n -го) уравнения не останется лишь один член с неизвестным x_n , т.е. матрица системы будет приведена к треугольному виду.

Обратный ход - последовательное вычисление искомых неизвестных:

- решая последнее уравнение, находим единственное в этом уравнении неизвестное x_n ;
- используя это значение, из предыдущего уравнения вычисляем x_{n-1} и т.д. ;
- последним найдём x_1 из первого уравнения.

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1,$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2,$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3.$$

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1,$$

$$a'_{22}x_2 + a'_{23}x_3 = b'_2,$$

$$a''_{33}x_3 = b''_3;$$

Прямые методы: метод прогонки.

Этот метод является модификацией метода Гаусса для частного случая разрежённых систем – системы уравнений с трёхдиагональной матрицей:

$$b_1x_1 + c_1x_2 = d_1,$$

$$a_2x_1 + b_2x_2 + c_2x_3 = d_2,$$

$$a_3x_2 + b_3x_3 + c_3x_4 = d_3,$$

$$a_{n-1}x_{n-2} + b_{n-1}x_{n-1} + c_{n-1}x_n = d_{n-1},$$

$$a_nx_{n-1} + b_nx_n = d_n.$$

Метод прогонки состоит из двух этапов – прямой прогонки (аналога прямого хода метода Гаусса) и обратной прогонки (аналога обратного хода метода Гаусса).

- Прямая прогонка состоит в вычислении прогоночных коэффициентов A_i и B_i , с помощью которых каждое неизвестное x_i выражается через x_{i+1} :

$$x_i = A_i x_{i+1} + B_i, \quad i = 1, 2, \dots, n-1. \quad (8)$$

Из первого уравнения системы имеем:

$$x_1 = -\frac{c_1}{b_1}x_2 + \frac{d_1}{b_1}.$$

С другой стороны, по формуле (8):

$$x_1 = A_1x_2 + B_1$$

Приравнивая коэффициенты в обоих выражениях для x_1 , получаем:

$$A_1 = -\frac{c_1}{b_1}, \quad B_1 = \frac{d_1}{b_1}.$$

Подставим во второе уравнение системы (7) вместо x_1 его выражение через x_2 по формуле (8):

$$a_2(A_1x_2 + B_1) + b_2x_2 + c_2x_3 = d_2$$

Выразим отсюда x_2 через x_3 : $x_2 = \frac{-c_2x_3 + d_2 - a_2B_1}{a_2A_1 + b_2}$.

Или $x_2 = A_2x_3 + B_2$,

$$A_2 = -\frac{c_2}{e_2}, \quad B_2 = \frac{d_2 - a_2B_1}{e_2}, \quad e_2 = a_2A_1 + b_2$$

Аналогично вычисляются прогоночные коэффициенты для любого номера i :

$$A_i = -\frac{c_i}{e_i}, \quad B_i = \frac{d_i - a_iB_{i-1}}{e_i}, \quad e_i = a_iA_{i-1} + b_i, \quad i = 2, 3, \dots, n-1. \quad (12)$$

Обратная прогонка состоит в последовательном вычислении неизвестных x_i . Сначала нужно найти x_n .

Для этого воспользуемся выражением (8) при $i=n-1$ и последним уравнением системы (7). Запишем их:

$$\begin{aligned} x_{n-1} &= A_{n-1}x_n + B_{n-1}, \\ a_nx_{n-1} + b_nx_n &= d_n. \end{aligned}$$

Отсюда, исключая x_{n-1} , находим:

$$x_n = \frac{d_n - a_nB_{n-1}}{b_n + a_nA_{n-1}}.$$

Далее, используя формулы (8) и ранее вычисленные прогоночные коэффициенты, последовательно вычисляем все неизвестные $x_{n-1}, x_{n-2}, \dots, x_1$.

Итерационные методы

Суть итерационных методов

1. Вводятся исходные данные:

- коэффициенты уравнений,
- допустимое значение погрешности ϵ ,
- начальные приближения значений неизвестных (вектор – столбец $X(0)$).

2. Организуется циклический вычислительный процесс, каждый цикл которого представляет собой одну итерацию – переход от предыдущего приближения $X(k-1)$ к последующему $X(k)$.

3. Если оказывается, что с увеличением числа итераций приближённое решение стремится к точному:

$$\lim_{k \rightarrow \infty} X^{(k)} = X$$

то итерационный метод называется сходящимся.

На практике наличие сходимости и достижение требуемой точности обычно определяют приближённо:

при малом (с заданной допустимой погрешностью) изменении X на двух последовательных итерациях, т.е. при малом отличии X(k) от X(k-1), процесс прекращается, и происходит вывод значений неизвестных, полученных на последней итерации.

Примеры критериев окончания итерационного процесса:

$$\left| X^{(k)} - X^{(k-1)} \right| = \sqrt{\sum_{i=1}^n (x_i^{(k)} - x_i^{(k-1)})^2} < \varepsilon$$

$$\max_{1 \leq i \leq n} \left| x_i^{(k)} - x_i^{(k-1)} \right| < \varepsilon$$

$$\max_{1 \leq i \leq n} \left| \frac{x_i^{(k)} - x_i^{(k-1)}}{x_i^{(k)}} \right| < \varepsilon \quad \text{при} \quad |x_i| >> 1$$

Метод простых итераций

Запишем исходную систему уравнений в векторно-матричном виде и выполним ряд тождественных преобразований:

$$\begin{aligned} Ax &= b; & 0 &= b - Ax; & x &= b - Ax + x; \\ x &= (b - Ax)\tau + x; & x &= (E - \tau A)x + \tau b; & (18) \\ x &= Bx + \tau b, \end{aligned}$$

где $\tau \neq 0$ - некоторое число, E – единичная матрица,
 $B=E-\tau A$.

Полученная система (18) эквивалентна исходной системе и служит основой для построения метода простой итерации.

Выберем некоторое начальное приближение $x(0)$ и поставим его в правую часть системы (18):

$$x^{(1)} = Bx^{(0)} + \tau b$$

Поскольку $x(0)$ не является решением системы, в левой части (18) получится некоторый столбец $x(1)$, в общем случае отличный от $x(0)$.

Полученный столбец $x(1)$ будем рассматривать в качестве следующего (первого) приближения к решению.

По известному k-му приближению можно найти (k+1) – е приближение:

$$x^{(k+1)} = Bx^{(k)} + \tau b \quad (19)$$

Формула (19) выражает собой метод простой итерации. Для её применения нужно задать неопределённый параметр τ .

От значения τ зависит, будет ли сходиться метод, а если будет, то какова скорость сходимости, т.е. как много итераций нужно совершить для достижения требуемой точности.

Теорема.

Пусть $\det A \neq 0$.

Метод простой итерации (19) сходится тогда и только тогда, когда все собственные числа матрицы $B = A - \tau E$ по модулю меньше единицы.

Для некоторых типов матрицы A можно указать правило выбора τ , обеспечивающее сходимость метода и оптимальную скорость сходимости.

В простейшем случае τ можно положить равным некоторому постоянному числу, например, 1, 0,1 и т.д.

Метод Гаусса – Зейделя

Проиллюстрируем этот метод на примере решения системы:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1, \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2, \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3. \end{aligned} \tag{20}$$

Предположим, что диагональные элементы отличны от нуля (в противном случае можно переставить уравнения).

Выразим неизвестные x_1, x_2, x_3 соответственно из первого, второго и третьего уравнений системы (20):

$$\begin{aligned} x_1 &= \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3), \\ x_2 &= \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3), \\ x_3 &= \frac{1}{a_{33}}(b_3 - a_{32}x_2 - a_{31}x_1). \end{aligned}$$

Зададим некоторые начальные (нулевые) приближения значений неизвестных:

$$x_1^{(0)}, x_2^{(0)}, x_3^{(0)}.$$

Подставляя эти значения в правую часть первого уравнения системы (21), получаем новое (первое) приближение для x_1 :

$$x_1^{(1)} = \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(0)} - a_{13}x_3^{(0)})$$

Используя это значение для x_1 и приближение для x_3 , находим из второго уравнения системы (21) первое приближение для x_2 :

$$x_2^{(1)} = \frac{1}{a_{22}} (b_2 - a_{21}x_1^{(1)} - a_{23}x_3^{(0)})$$

И наконец, используя вычисленные значения

$$x_1 = x_1^{(1)}, \quad x_2 = x_2^{(2)},$$

находим с помощью третьего уравнения системы (21) первое приближение для x_3 :

$$x_3^{(1)} = \frac{1}{a_{33}} (b_3 - a_{31}x_1^{(1)} - a_{32}x_2^{(1)})$$

Используя значения $x_1^{(1)}$, $x_2^{(1)}$, $x_3^{(1)}$

Можно таким же способом провести вторую итерацию, в результате которой будут найдены вторые приближения к решению:

$$x_1 = x_1^{(2)}, \quad x_2 = x_2^{(2)}, \quad x_3 = x_3^{(3)}.$$

Приближение с номером k можно вычислить, зная приближение с номером $k-1$, как:

$$\begin{aligned} x_1^k &= \frac{1}{a_{11}} (b_1 - a_{12}x_2^{k-1} - a_{13}x_3^{k-1}), \\ x_2^k &= \frac{1}{a_{22}} (b_2 - a_{21}x_1^k - a_{23}x_3^{k-1}), \\ x_3^k &= \frac{1}{a_{33}} (b_3 - a_{32}x_2^k - a_{31}x_1^k). \end{aligned}$$

Итерационный процесс продолжается до тех пор, пока значения $x_1^{(k)}, x_2^{(k)}, x_3^{(k)}$ не станут близкими с заданной погрешностью к значениям $x_1^{(k-1)}, x_2^{(k-1)}, x_3^{(k-1)}$

В качестве первого приближения обычно принимают нулевые значения, т.е.

$$x_1^{(0)} = 0, \quad x_2^{(0)} = 0, \quad x_3^{(0)} = 0$$

Рассмотрим теперь общий случай для системы из n уравнений. Запишем её в виде:

$$a_{i1}x_1 + \dots + a_{i,i-1}x_{i-1} + a_{ii}x_i + a_{i,i+1}x_{i+1} + \dots + a_{in}x_n = b_i \quad ,$$

$i=1,2,\dots,n$.

Здесь также будем предполагать, что все диагональные элементы отличны от нуля.

Тогда в соответствии с методом Гаусса- Зейделя k-е приближение к решению можно представить в виде:

$$x_i^k = \frac{1}{a_{ii}}(b_{ii} - a_{i1}x_1^k - \dots - a_{i,i-1}x_{i-1}^k - a_{i,i+1}x_{i+1}^{k-1} - \dots - a_{in}x_n^{k-1}), \quad i=1,2,\dots,n.$$

Итерационный процесс продолжается до тех пор, пока все значения $x_i^{(k)}$

не станут близкими к $x_i^{(k-1)}$

Для сходимости итерационного процесса достаточно, чтобы модули диагональных коэффициентов для каждого уравнения системы были не меньше сумм модулей всех остальных коэффициентов (преобладание диагональных элементов):

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|, \quad i=1,2,\dots,n \quad (25)$$

При этом хотя бы для одного уравнения неравенство (25) должно выполняться строго.

5. Численное дифференцирование и интегрирование. Численные методы для решения систем дифференциальных уравнений.

Численное дифференцирование и интегрирование – вычисление производных и интегралов по приближенным разностным формулам. Если функция задана в виде таблицы или имеет очень сложный вид и алгоритм расчета, то ее можно дифференцировать или интегрировать только численным способом.

Простейшие квадратурные формулы. Как правило для вычисления значения определенного интеграла применяют специальные вычислительные методы. Наиболее широко используют на практике квадратурные формулы – приближенные равенства вида:

$$\int_a^b f(x) dx \approx \sum_{i=0}^N A_i f(\bar{x}_i) \quad (1)$$

В выражение (1) \bar{x}_i – некоторые точки из отрезка $[a, b]$ – узлы квадратурной формулы; A_i – числовые коэффициенты, называемые весами квадратурной формулы; $N \geq 0$ – целое число.

Сумма, которая принимается за приближенное значение интеграла, называется квадратурной формулой. Величина R , которая равна разнице интеграла и суммы, называется погрешностью (или остаточным членом) квадратурной формулы.

Разобьем отрезок $[a, b]$ на элементарные отрезки $[x_{i-1}, x_i]$ точками $a = x_0 < x_1 < \dots < x_n = b$. Интеграл I разбьется при этом на сумму элементарных интегралов:

$$I = \sum_{i=1}^n I_i, \quad (2)$$

Где $\int_{x_{i-1}}^{x_i} f(x)dx,$

Что соответствует разбиению исходной криволинейной трапеции на сумму площадей элементарных криволинейных трапеций (см. рисунок 1).

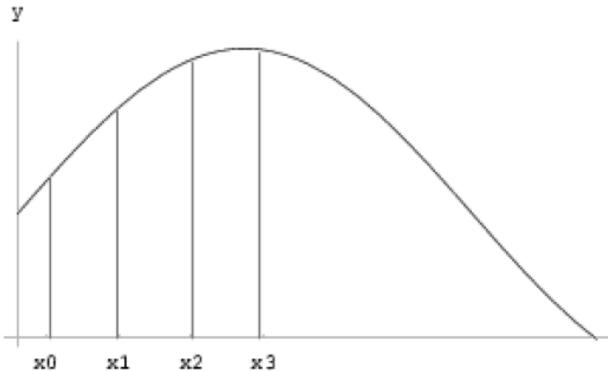


Рисунок 1

Формула прямоугольников.

Введем обозначения: $f_i = f(x_i)$, $f_{i-1/2} = f(x_{(i-1)/2})$, где $x_{(i-1)/2} = (x_{i-1} + x_i)/2$ – середина элементарного отрезка. Шаг $x_i - x_{i-1}$ будем считать постоянным.

Заменим приближенно площадь элементарной криволинейной трапеции площадью прямоугольника, основанием которого является отрезок $[x_{i-1}, x_i]$, а высота равна значению $f_{i-1/2}$, $N_{i-1/2}$ – точка с координатами $(x_{(i-1)/2}, f_{i-1/2})$. В итоге приходим к элементарной квадратурной формуле прямоугольников:

$$I_i \approx h \cdot f_{i-1/2} \quad (3)$$

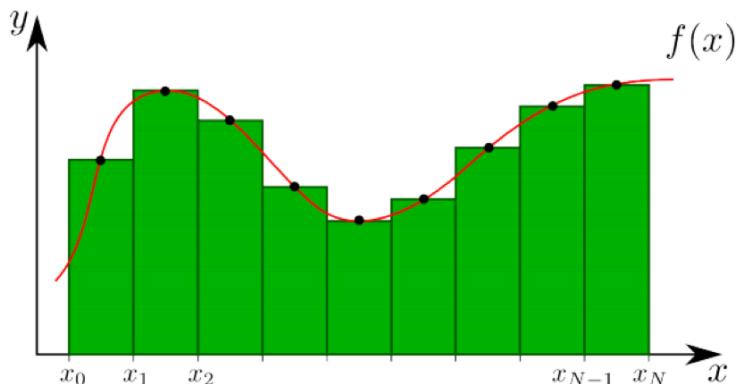


Рисунок 2

Из выражения (3) получаем составную квадратурную формулу прямоугольников:

$$I \approx I_{\text{пр}}^h = h(f_{1/2} + f_{3/2} + \dots + f_{n-1/2}) = h \sum_{i=1}^n f_{i-1/2} \quad (4)$$

Выбор в качестве значения функции средней точки интервала не принципиален, можно взять, например, левый или правый конец интервала.

$$I \approx h \sum_{i=0}^n f_i, \quad (5)$$

$$I \approx h \sum_{i=1}^n f_i, \quad (6)$$

Выражения (5) и (6) называются соответственно составными квадратурными формулами левых (см. рисунок 3а) и правых (см. рисунок 3б) прямоугольников.

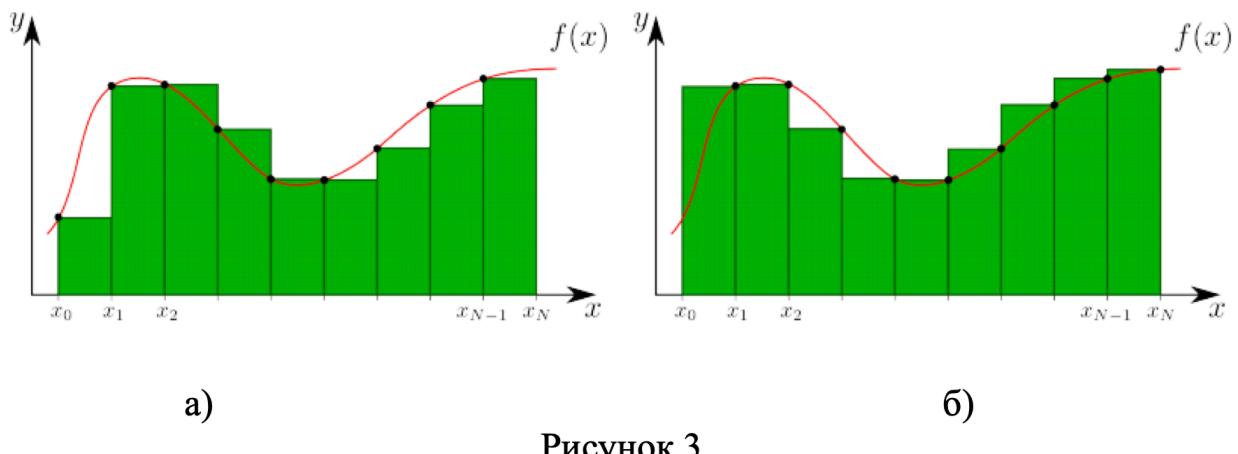


Рисунок 3

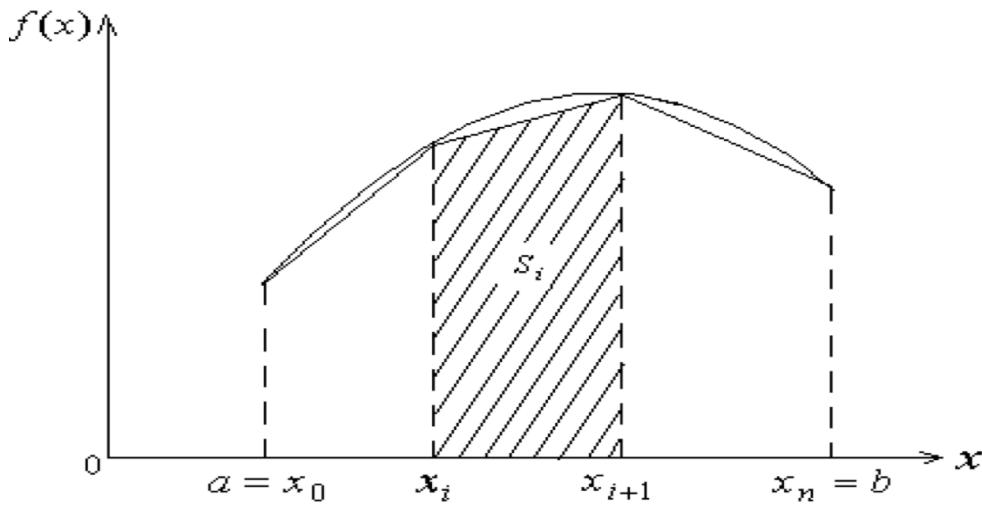
Квадратурная формула трапеций.

Соединив отрезком точки $N_{i-1}(x_{i-1}, f_{i-1})$ и $N_i(x_i, f_i)$ на графике функции $y = f(x)$, получим трапецию. Заменим теперь приближенно площадь элементарной криволинейной трапеции площадью построенной фигуры. Тогда получим элементарную квадратурную формулу трапеций:

$$I_i \approx \frac{h}{2} (f_{i-1} + f_i) \quad (7)$$

Из выражения (7) получаем составную квадратурную формулу трапеций:

$$I \approx I_{\text{тр}}^h = h \left(\frac{f_0 + f_n}{2} + \sum_{i=1}^n f_i \right) \quad (8)$$



Квадратурная формула Симпсона.

Если площадь элементарной криволинейной трапеции заменить площадью фигуры, расположенной под параболой, проходящей через точки x_{i-1} , $x_{i-1/2}$ и x_i , то получим приближенное равенство

$$I_i \approx \int_{x_{i-1}}^{x_i} P_2(x) dx \quad (9)$$

Здесь $P_2(x)$ – интерполяционный многочлен второй степени с узлами x_{i-1} , $x_{i-1/2}$, x_i .

$$P_2(x) = f_{i-1/2} + \frac{f_i - f_{i-1}}{h} (x - x_{i-1/2}) + \frac{f_i - 2f_{i-1/2} + f_{i-1}}{h^2/2} (x - x_{i-1/2})^2 \quad (10)$$

Интерполяционный многочлен – это многочлен, который аппроксимирует набор точек данных, таким образом, что он проходит через все эти точки. Интерполяционный многочлен используется для представления функции, когда известны значения функции в конечном наборе точек.

Интегрирование $P_2(x)$ приводит к равенству:

$$\int_{x_{i-1}}^{x_i} P_2(x) dx = \frac{h}{6} (f_{i-1} + 4f_{i-1/2} + f_i) \quad (11)$$

В результате получаем элементарную квадратичную формулу Симпсона:

$$I_i \approx \frac{h}{6} (f_{i-1} + 4f_{i-1/2} + f_i) \quad (12)$$

Применяя формулу (12) на каждом элементарном отрезке, выводим составную квадратичную формулу Симпсона:

$$I \approx I_C^h = \frac{h}{6} \left(f_0 + f_n + 4 \sum_{i=1}^n f_{i-1/2} + 2 \sum_{i=1}^{n-1} f_i \right) \quad (13)$$

Задача численного дифференцирования приближенно заданных функций

Простейшие формулы численного дифференцирования. Пусть функция f дифференцируема достаточноное количество раз в окрестности точки x . Из определения производной:

$$f'(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}, \quad (15)$$

получим две простейшие приближенные формулы:

$$f'(x) \approx \frac{f(x + h) - f(x)}{h}, \quad (16)$$

$$f'(x) \approx \frac{f(x) - f(x - h)}{h}, \quad (17)$$

соответствующие выбору фиксированных значений $dx = h$ и $dx = -h$, $h > 0$ – малый параметр (шаг). Разностные отношения в правых частях формул (16) и (17) называют правой и левой разностными производными. Данные формулы имеют первый порядок точности. Для повышения точности можно использовать формулу центральной разностной производной:

$$f'(x) \approx \frac{f(x + h) - f(x - h)}{2h}. \quad (18)$$

Для вычисления производной $f'(x)$ можно получить формулы любого порядка точности, но в таких формулах с ростом порядка точности возрастает и число используемых значений функции. Например, формула, имеющая четвертый порядок точности, будет иметь вид:

$$f'(x) \approx \frac{f(x - 2h) - 8f(x - h) + 8f(x + h) - f(x + 2h)}{12h}. \quad (19)$$

Численные методы для решения систем дифференциальных уравнений.

1. Метод Эйлера: Этот метод является одним из простейших численных методов для решения обыкновенных дифференциальных уравнений (ОДУ). Он основан на

аппроксимации производной функции разностным отношением и используется для решения начальных значений задач.

2. Метод Рунге-Кутты: Этот метод представляет собой семейство численных методов для решения ОДУ, которые обеспечивают более точное приближение к решению по сравнению с методом Эйлера. Метод Рунге-Кутты различных порядков позволяет контролировать точность численного решения.

Метод Эйлера

Пусть дана следующая система обыкновенных дифференциальных уравнений:

$$\begin{cases} \frac{dy_1}{dx} = f_1(x, y_1, y_2) \\ \frac{dy_2}{dx} = f_2(x, y_1, y_2) \end{cases} \quad (3)$$

с начальными условиями:

$$\begin{aligned} y_1|_{x=x_0} &= y_{01} \\ y_2|_{x=x_0} &= y_{02} \end{aligned} \quad (4)$$

При использовании метода Эйлера, расчетные формулы примут следующий вид:

$$\begin{cases} y_{(i),1} = y_{i-1,1} + h \cdot f_1(x_{i-1}, y_{(i-1),1}, y_{(i-1),2}) \\ y_{(i),2} = y_{i-1,2} + h \cdot f_2(x_{i-1}, y_{(i-1),1}, y_{(i-1),2}) \\ x_i = x_{i-1} + h \end{cases} \quad (5)$$

где h – шаг интегрирования; $f_1(x_i, y_{i,1}, y_{i,2})$ и $f_2(x_i, y_{i,1}, y_{i,2})$ – правые части дифференциальных уравнений.

Метод Рунге-Кутты

Пусть дана следующая система обыкновенных дифференциальных уравнений:

$$\begin{cases} \frac{dy_1}{dx} = f_1(x, y_1, y_2) \\ \frac{dy_2}{dx} = f_2(x, y_1, y_2) \end{cases} \quad (6)$$

с начальными условиями:

$$\begin{aligned} y_1|_{x=x_0} &= y_{01} \\ y_2|_{x=x_0} &= y_{02} \end{aligned} \quad (7)$$

Метод Рунге-Кутты

При использовании метода Рунге-Кутты, расчетные формулы примут следующий вид:

$$\begin{cases} y_{i,1} = y_{(i-1),1} + h/6 \cdot (k_{1,1} + 2 \cdot k_{2,1} + 2 \cdot k_{3,1} + k_{4,1}) \\ y_{i,2} = y_{(i-1),2} + h/6 \cdot (k_{1,2} + 2 \cdot k_{2,2} + 2 \cdot k_{3,2} + k_{4,2}) \\ x_i = x_{i-1} + h \end{cases} \quad (8)$$

где

$$\begin{aligned} k_{1,1} &= f_1(x, y_{(i-1),1}, y_{(i-1),2}); & k_{1,2} &= f_2(x, y_{(i-1),1}, y_{(i-1),2}); \\ k_{2,1} &= f_1\left(x + \frac{h}{2}, y_{(i-1),1} + k_{1,1} \cdot \frac{h}{2}, y_{(i-1),2} + k_{1,2} \cdot \frac{h}{2}\right); & k_{2,2} &= f_2\left(x + \frac{h}{2}, y_{(i-1),1} + k_{1,1} \cdot \frac{h}{2}, y_{(i-1),2} + k_{1,2} \cdot \frac{h}{2}\right); \\ k_{3,1} &= f_1\left(x + \frac{h}{2}, y_{(i-1),1} + k_{2,1} \cdot \frac{h}{2}, y_{(i-1),2} + k_{2,2} \cdot \frac{h}{2}\right); & k_{3,2} &= f_2\left(x + \frac{h}{2}, y_{(i-1),1} + k_{2,1} \cdot \frac{h}{2}, y_{(i-1),2} + k_{2,2} \cdot \frac{h}{2}\right); \\ k_{4,1} &= f_1(x + h, y_{(i-1),1} + k_{3,1} \cdot h, y_{(i-1),2} + k_{3,2} \cdot h); & k_{4,2} &= f_2(x + h, y_{(i-1),1} + k_{3,1} \cdot h, y_{(i-1),2} + k_{3,2} \cdot h). \end{aligned} \quad (9)$$

где h – шаг интегрирования; $f_1(x_i, y_{(i-1),1}, y_{(i-1),2})$ и $f_2(x_i, y_{(i-1),1}, y_{(i-1),2})$ – правые части дифференциальных уравнений, $k_{1,j}$, $k_{2,j}$, $k_{3,j}$, $k_{4,j}$ – параметры метода Рунге-Кутты для j -го уравнения.

Пример метод Эйлера

Пусть требуется решить систему дифференциальных уравнений первого порядка:

$$\begin{cases} \frac{dy_1}{dx} = y_2 \\ \frac{dy_2}{dx} = e^{-x \cdot y_1} \end{cases}$$

методом Эйлера на отрезке $[0, 1]$ с шагом $h = 0.1$. Начальные условия: $x_0 = 0$; $y_1(0) = 0$; $y_2(0) = 0$. Воспользуемся формулой (5) и запишем выражения для $y_{i,1}$ и $y_{i,2}$:

$$\begin{cases} y_{i,1} = y_{(i-1),1} + 0.1 \cdot y_{(i-1),2} \\ y_{i,2} = y_{(i-1),2} + 0.1 \cdot e^{-x_{i-1} \cdot y_{(i-1),1}} \\ x_i = x_{i-1} + h \end{cases}$$

Пример метод Рунге-Кутты

Решим методом Рунге-Кутты пример, приведенный на слайде 7. Воспользуемся формулами (8), (9) и запишем выражения для нахождения значений искомых переменных $y_{i,1}$ и $y_{i,2}$:

$$\begin{aligned} k_{1,1} &= y_{(i-1),2}; & k_{1,2} &= \exp(-x_i \cdot y_{(i-1),1}); \\ k_{2,1} &= y_{(i-1),2} + k_{1,2} \cdot \frac{h}{2}; & k_{2,2} &= \exp\left[-\left(x_i + \frac{h}{2}\right) \cdot \left(y_{(i-1),1} + k_{1,1} \cdot \frac{h}{2}\right)\right] \\ k_{3,1} &= y_{(i-1),2} + k_{2,2} \cdot \frac{h}{2}; & k_{3,2} &= \exp\left[-\left(x_i + \frac{h}{2}\right) \cdot \left(y_{(i-1),1} + k_{2,1} \cdot \frac{h}{2}\right)\right] \\ k_{4,1} &= y_{(i-1),2} + k_{3,2} \cdot h; & k_{4,2} &= \exp\left[-(x_i + h) \cdot (y_{(i-1),1} + k_{3,1} \cdot h)\right] \\ \\ \begin{cases} y_{i,1} = y_{(i-1),1} + \frac{0.1}{6} \cdot (k_{1,1} + 2 \cdot k_{2,1} + 2 \cdot k_{3,1} + k_{4,1}) \\ y_{i,2} = y_{(i-1),2} + \frac{0.1}{6} \cdot (k_{1,2} + 2 \cdot k_{2,2} + 2 \cdot k_{3,2} + k_{4,2}) \\ x_i = x_{i-1} + 0.1 \end{cases} \end{aligned}$$

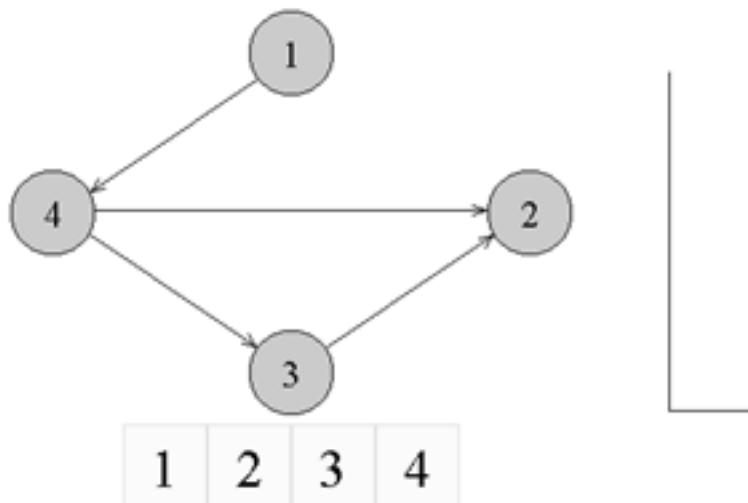
6. Граф. Ориентированный граф. Представления графа. Обход графа в глубину и в ширину. Топологическая сортировка. Подсчет числа путей в орграфе.

Смотри другой файл

Топологическая сортировка — один из основных алгоритмов на графах, который применяется для решения множества более сложных задач.

Задача топологической сортировки графа состоит в следующем: указать такой линейный порядок на его вершинах, чтобы любое ребро вело от вершины с меньшим номером к вершине с большим номером. Очевидно, что если в графе есть циклы, то такого порядка не существует.

Ориентированной сетью называют бесконтурный ориентированный граф. В задачах подобного плана рассматриваются только конечные сети.



↑ Пример ориентированного неотсортированного графа, к которому применима топологическая сортировка

Из рисунка видно, что граф не отсортирован, так как ребро из вершины с номером 4 ведет в вершину с меньшим номером (2).

Существует несколько способов топологической сортировки — из наиболее известных:

- Алгоритм Демукрона
- Метод сортировки для представления графа в виде нескольких уровней
- Метод топологической сортировки с помощью обхода в глубину

Я остановлюсь на рассмотрении последнего, поскольку он наиболее популярен, нагляден и прост в реализации.

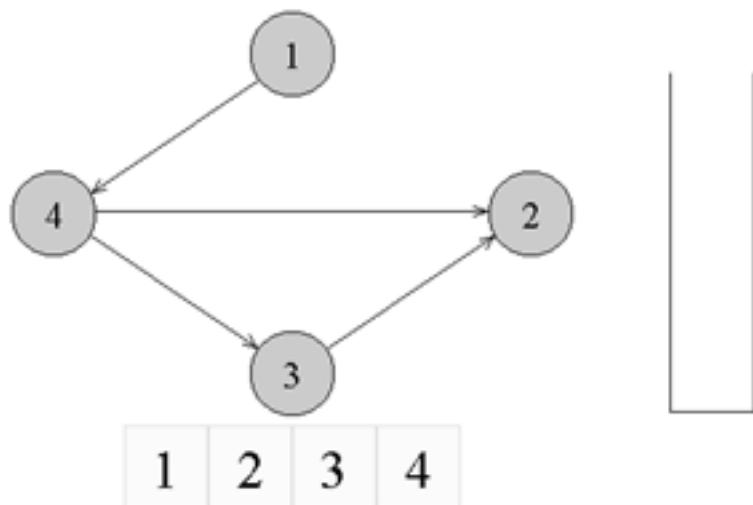
Суть алгоритма

Поиск в глубину или обход в глубину — один из методов обхода графа. Алгоритм поиска описывается следующим образом: для каждой не пройденной вершины необходимо найти все не пройденные смежные вершины и повторить поиск для них.

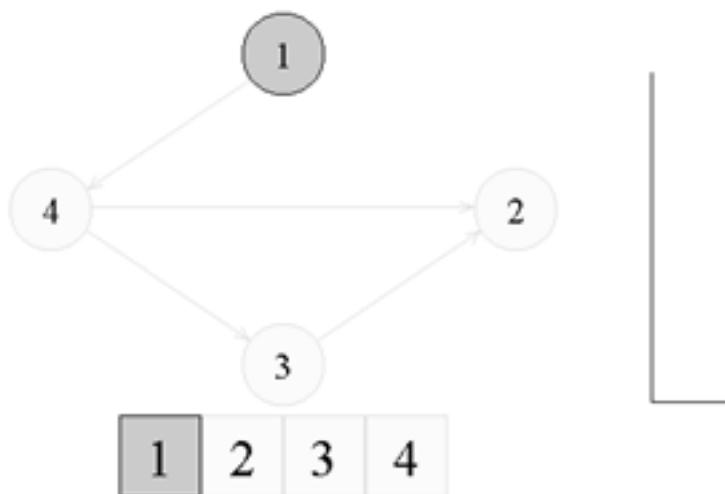
Запускаем обход в глубину, и когда вершина обработана, заносим ее в стек. По окончании обхода в глубину вершины достаются из стека. Новые номера присваиваются в порядке вытаскивания из стека.

Цвет: во время обхода в глубину используется 3 цвета. Изначально все вершины белые. Когда вершина обнаружена, красим ее в серый цвет. Когда просмотрен список всех смежных с ней вершин, красим ее в черный цвет.

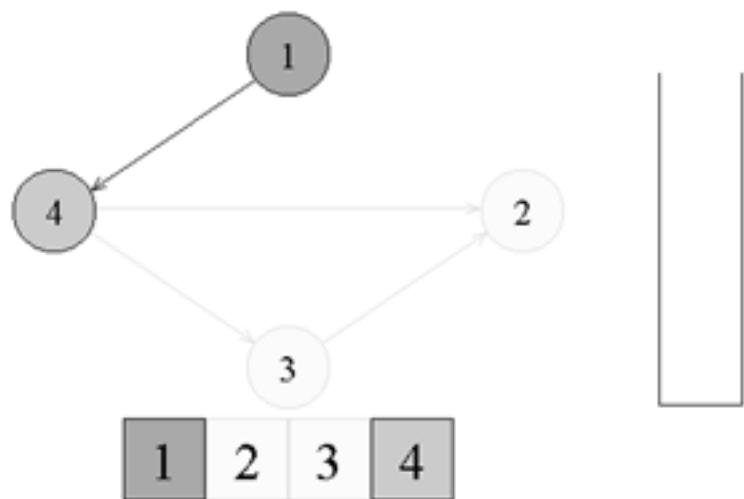
Думаю будет проще рассмотреть данный алгоритм на примере:



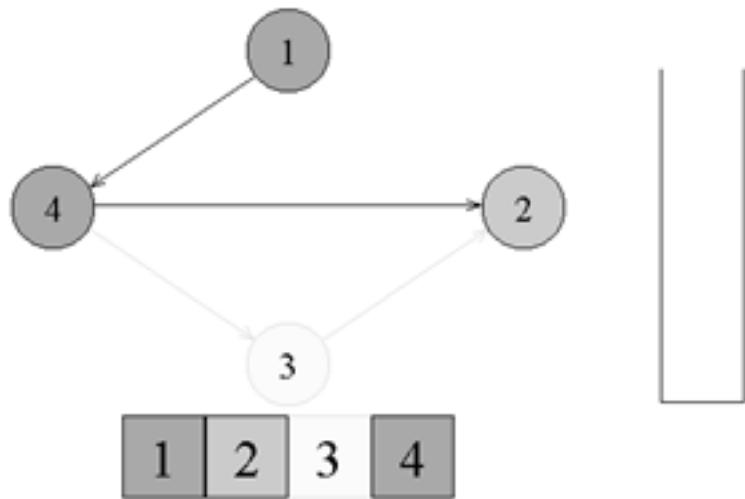
↑ Имеем бесконтурный ориентированный граф. Изначально все вершины белые, а стек пуст. Начнем обход в глубину с вершины номер 1.



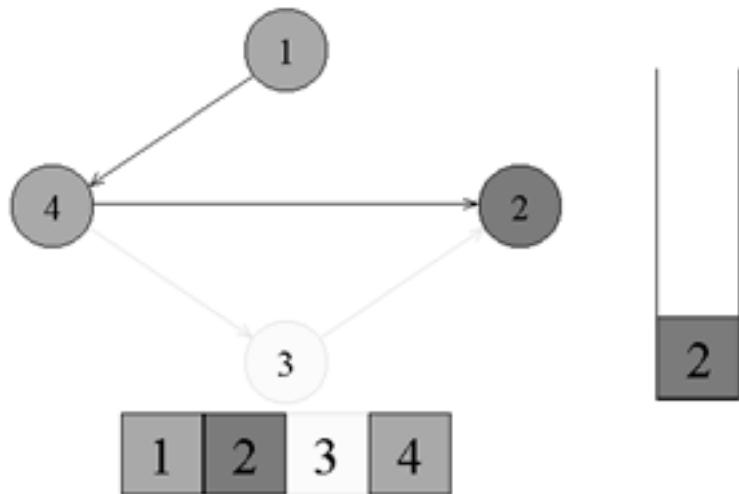
↑ Переходим к вершине номер 1. Красим ее в серый цвет.



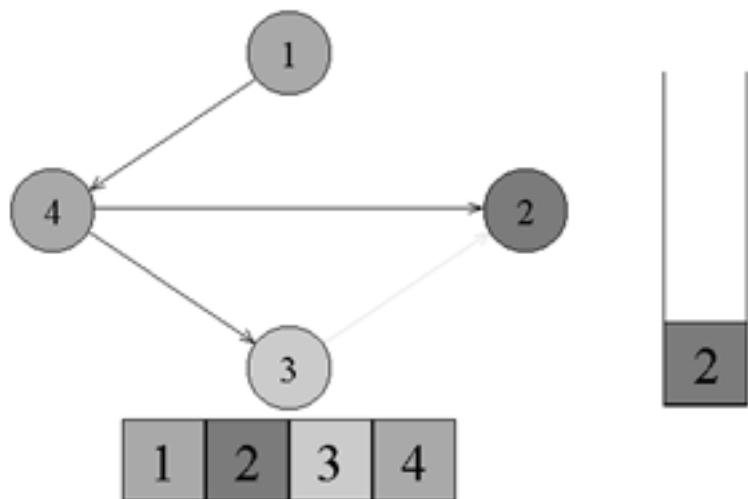
↑ Существует ребро из вершины номер 1 в вершину номер 4. Переходим к вершине номер 4 и красим ее в серый цвет.



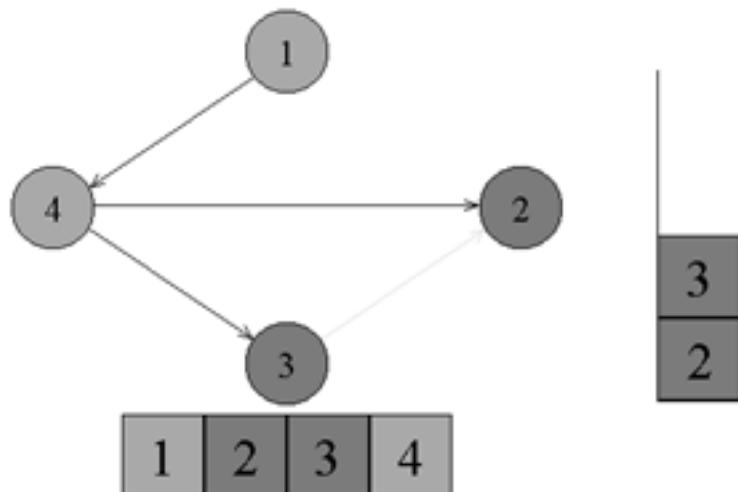
↑ Существует ребро из вершины номер 4 в вершину номер 2. Переходим к вершине номер 2 и красим ее в серый цвет.



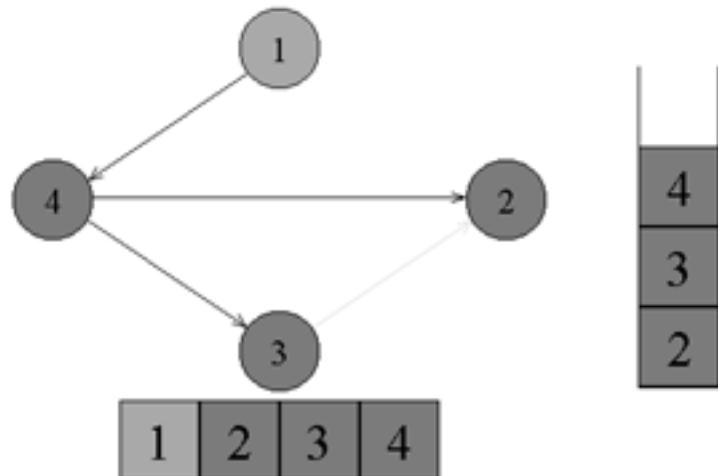
↑ Из вершины номер 2 нет рёбер, идущих не в черные вершины. Возвращаемся к вершине номер 4. Красим вершину номер 2 в черный цвет и кладем ее в стек.



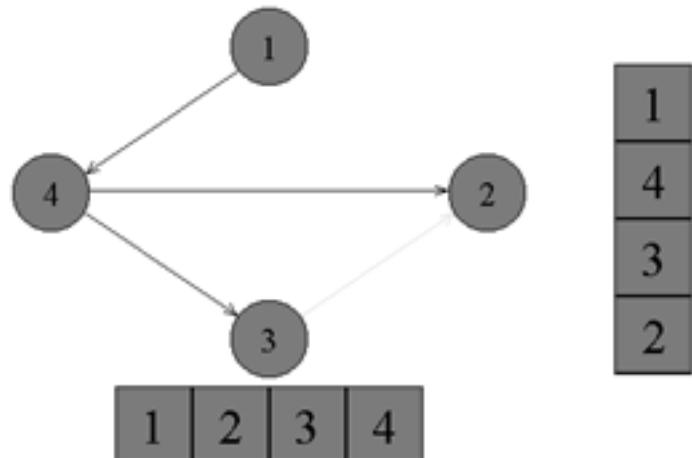
↑ Существует ребро из вершины номер 4 в вершину номер 3. Переходим к вершине номер 3 и красим ее в серый цвет.



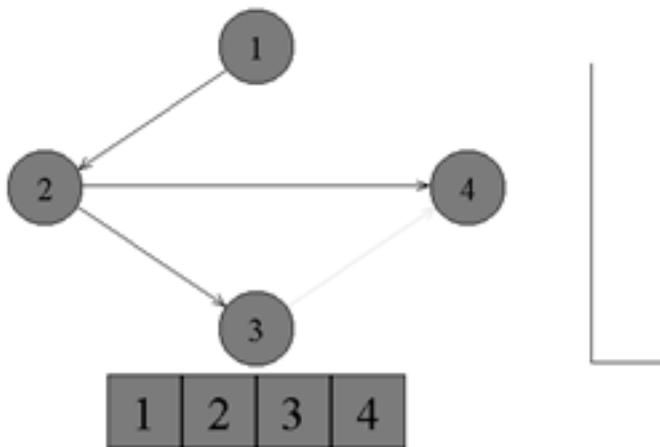
↑ Из вершины номер 3 нет рёбер, идущих не в черные вершины. Возвращаемся к вершине номер 4. Красим вершину номер 3 в черный цвет и кладем ее в стек.



↑ Из вершины номер 4 нет рёбер, идущих не в черные вершины. Возвращаемся к вершине номер 1. Красим вершину номер 4 в черный цвет и кладем ее в стек.



↑ Из вершины номер 1 нет рёбер, идущих не в черные вершины. Красим её в черный цвет и кладем в стек. Обход точек закончен.



↑ По очереди достаем все вершины из стека и присваиваем им номера 1, 2, 3, 4 соответственно. Алгоритм топологической сортировки завершен. Граф отсортирован.

Применение

Топологическая сортировка применяется в самых разных ситуациях, например при распараллеливании алгоритмов, когда по некоторому описанию алгоритма нужно составить граф зависимостей его операций и, отсортировав его топологически, определить, какие из операций являются независимыми и могут выполняться параллельно (одновременно). Примером использования топологической сортировки может служить создание карты сайта, где имеет место древовидная система разделов.

Задача о числе путей в ациклическом графе

Задан ациклический граф и две вершины s и t . Необходимо посчитать количество путей из s в t .

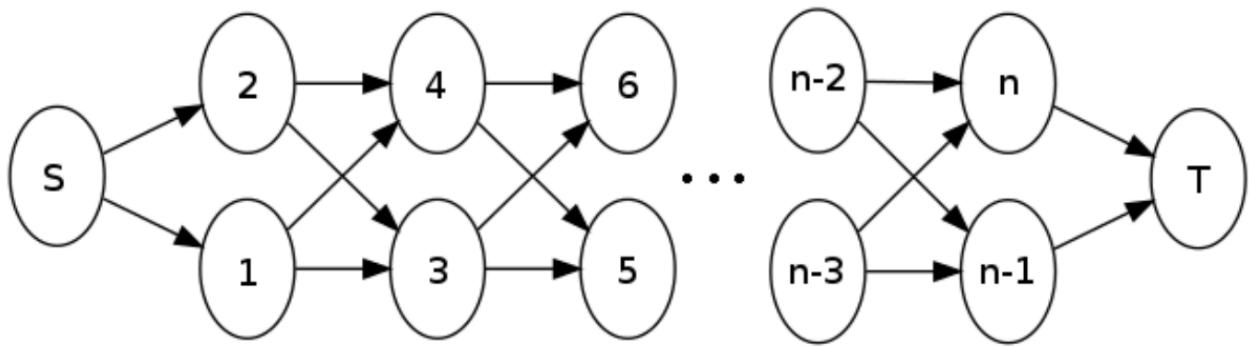
Перебор всех возможных путей

Небольшая модификация алгоритма обхода в глубину. Запустим обход в глубину от вершины s . При каждом посещении вершины v проверим, не является ли она искомой вершиной t . Если это так, то ответ увеличивается на единицу и обход прекращается. В противном случае производится запуск обхода в глубину для всех вершин, в которые есть ребро из v , причем он производится независимо от того, были эти вершины посещены ранее, или нет.

Функция `countPaths(g,s,t)` принимает граф g в виде списка смежности, начальную вершину s и конечную вершину t .

```
countPaths(g, v, t)
    if v == t
        return 1
    else
        s = 0
        for to in g[v]
            s += countPaths(g, to, t)
    return s
```

Время работы данного алгоритма в худшем случае $O(Ans)$, где Ans — число путей в графе из s в t . Например, на следующем графе данный алгоритм будет иметь время работы $O(2^{n/2})$. Если же использовать метод динамического программирования, речь о котором пойдет ниже, то асимптотику можно улучшить до $O(n)$.



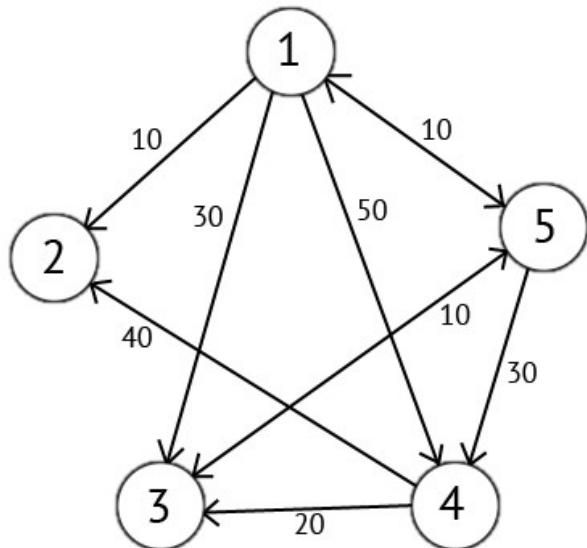
Метод динамического программирования

Пусть $P(v)$ — число путей от вершины s до вершины v . Тогда $P(v)$ зависит только от вершин, ребра из которых входят в v . Тогда $P(v) = \sum P(c)$ таких c , что есть ребро из s в v . Мы свели нашу задачу к меньшим подзадачам, причем мы также знаем, что $P(s)=1$. Это позволяет решить задачу методом динамического программирования.

7. Алгоритмы поиска кратчайших путей в графе. Алгоритм Дейкстры. Алгоритм Форда-Беллмана. Алгоритм Флойда. Алгоритм A*.

Алгоритм Дейкстры.

Для примера возьмем такой ориентированный граф G :



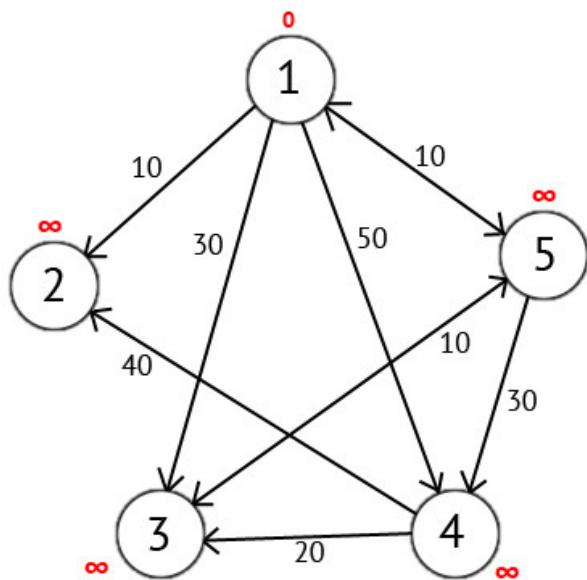
Этот граф мы можем представить в виде матрицы C :

	1	2	3	4	5
1		10	30	50	10
2					
3					10
4		40	20		
5	10		10	30	

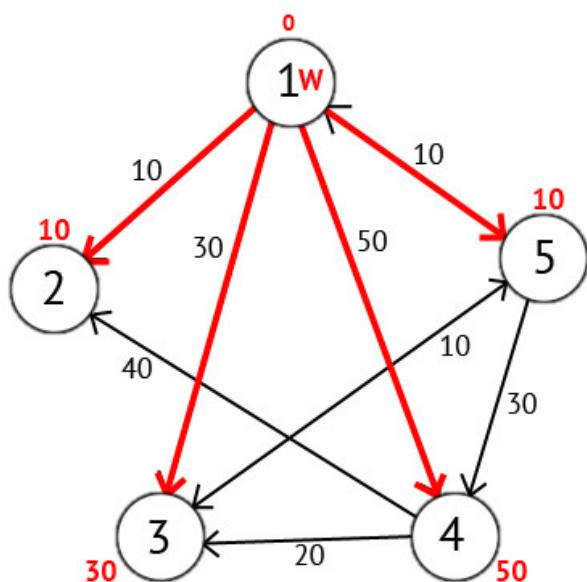
Возьмем в качестве источника вершину 1. Это значит что мы будем искать кратчайшие маршруты из вершины 1 в вершины 2, 3, 4 и 5.

Данный алгоритм пошагово перебирает все вершины графа и назначает им метки, которые являются известным минимальным расстоянием от вершины источника до конкретной вершины. Рассмотрим этот алгоритм на примере.

Присвоим 1-й вершине метку равную 0, потому как эта вершина — источник. Остальным вершинам присвоим метки равные бесконечности.

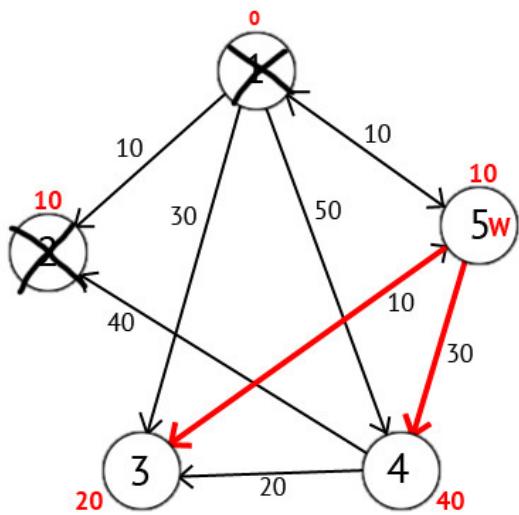


Далее выберем такую вершину W , которая имеет минимальную метку (сейчас это вершина 1) и рассмотрим все вершины в которые из вершины W есть путь, не содержащий вершин посредников. Каждой из рассмотренных вершин назначим метку равную сумме метки W и длины пути из W в рассматриваемую вершину, но только в том случае, если полученная сумма будет меньше предыдущего значения метки. Если же сумма не будет меньше, то оставляем предыдущую метку без изменений.



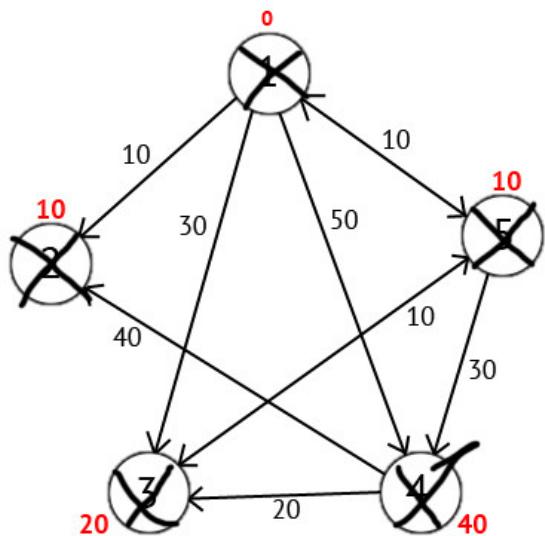
После того как мы рассмотрели все вершины, в которые есть прямой путь из W , вершину W мы отмечаем как посещённую, и выбираем из ещё не посещенных такую, которая имеет минимальное значение метки, она и будет следующей вершиной W . В данном случае это вершина 2 или 5. Если есть несколько вершин с одинаковыми метками, то не имеет значения какую из них мы выберем как W .

Мы выберем вершину 2. Но из нее нет ни одного исходящего пути, поэтому мы сразу отмечаем эту вершину как посещенную и переходим к следующей вершине с минимальной меткой. На этот раз только вершина 5 имеет минимальную метку. Рассмотрим все вершины в которые есть прямые пути из 5, но которые еще не помечены как посещенные. Снова находим сумму метки вершины W и веса ребра из W в текущую вершину, и если эта сумма будет меньше предыдущей метки, то заменяем значение метки на полученную сумму.



Исходя из картинки мы можем увидеть, что метки 3-й и 4-ой вершин стали меньше, то есть был найден более короткий маршрут в эти вершины из вершины источника. Далее отмечаем 5-ю вершину как посещенную и выбираем следующую вершину, которая имеет минимальную метку. Повторяем все перечисленные выше действия до тех пор, пока есть непосещенные вершины.

Выполнив все действия получим такой результат:



Также есть вектор P , исходя из которого можно построить кратчайшие маршруты. По количеству элементов этот вектор равен количеству вершин в графе. Каждый элемент содержит последнюю промежуточную вершину на кратчайшем пути между вершиной-источником и конечной вершиной. В начале алгоритма все элементы вектора P равны вершине источнику (в нашем случае $P = \{1, 1, 1, 1, 1\}$). Далее на этапе пересчета значения метки для рассматриваемой вершины, в случае если метка рассматриваемой вершины меняется на меньшую, в массив P мы записываем значение текущей вершины W .

Например: у 3-ей вершины была метка со значением «30», при $W=1$. Далее при $W=5$, метка 3-ей вершины изменилась на «20», следовательно мы запишем значение в вектор P — $P[3]=5$. Так же при $W=5$ изменилось значение метки у 4-й вершины (было «50», стало «40»), значит нужно присвоить 4-му элементу вектора P значение $W - P[4]=5$. В результате получим вектор $P = \{1, 1, 5, 5, 1\}$.

Зная что в каждом элементе вектора P записана последняя промежуточная вершина на пути между источником и конечной вершиной, мы можем получить и сам кратчайший маршрут.

Алгоритм Форда- Беллмана.

Входные данные: Граф и начальная вершина src .

Выходные данные: Кратчайшее расстояние до всех вершин от src . Если попадается цикл отрицательного веса, то самые короткие расстояния не вычисляются, выводится сообщение о наличии такого цикла.

1. На этом шаге инициализируются расстояния от исходной вершины до всех остальных вершин, как бесконечные, а расстояние до самого src принимается равным 0.
Создается массив $dist[]$ размера $|V|$ со всеми значениями равными бесконечности, за исключением элемента $dist[src]$, где src — исходная вершина.
2. Вторым шагом вычисляются самые короткие расстояния. Следующие шаги нужно выполнять $|V|-1$ раз, где $|V|$ — число вершин в данном графе.
 - Произведите следующее действие для каждого ребра $u-v$:
Если $dist[v] > dist[u] + \text{вес ребра } uv$, то обновите $dist[v]$
 $dist[v] = dist[u] + \text{вес ребра } uv$
3. На этом шаге сообщается, присутствует ли в графе цикл отрицательного веса. Для каждого ребра $u-v$ необходимо выполнить следующее:
 - Если $dist[v] > dist[u] + \text{вес ребра } uv$, то в графе присутствует цикл отрицательного веса.

Идея шага 3 заключается в том, что шаг 2 гарантирует кратчайшее расстояние, если график не содержит цикла отрицательного веса. Если мы снова переберем все ребра и получим более короткий путь для любой из вершин, это будет сигналом присутствия цикла отрицательного веса.

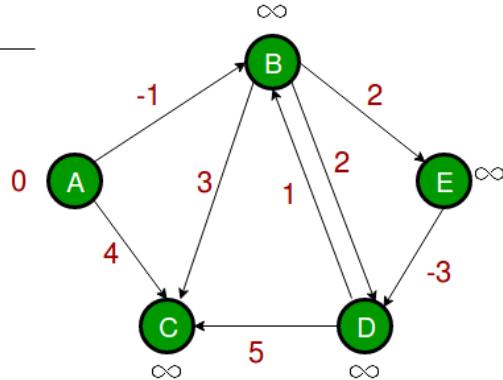
Как это работает? Как и в других задачах динамического программирования, алгоритм вычисляет кратчайшие пути снизу вверх. Сначала он вычисляет самые короткие расстояния, то есть пути длиной не более, чем в одно ребро. Затем он вычисляет кратчайшие пути длиной не более двух ребер и так далее. После i -й итерации внешнего цикла вычисляются кратчайшие пути длиной не более i ребер. В любом простом пути может быть максимум $|V|-1$ ребер, поэтому внешний цикл выполняется именно $|V|-1$ раз. Идея заключается в том, что если мы вычислили кратчайший путь с не более чем i ребрами, то итерация по всем ребрам гарантирует получение кратчайшего пути с не более чем $i+1$ ребрами

Пример

Давайте разберемся в алгоритме на следующем примере графа. Изображения взяты отсюда.

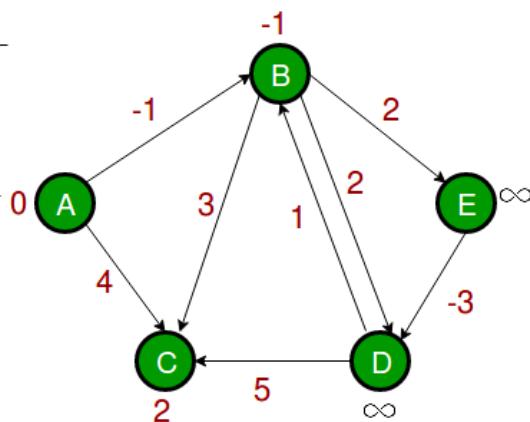
Пусть начальная вершина равна 0. Примите все расстояния за бесконечные, кроме расстояния до самой src . Общее число вершин в графике равно 5, поэтому все ребра нужно пройти 4 раза.

A	B	C	D	E
0	∞	∞	∞	∞



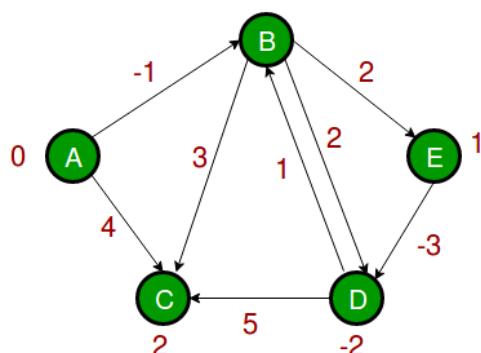
Пусть ребра отрабатываются в следующем порядке: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). Мы получаем следующие расстояния, когда проход по ребрам был совершен первый раз. Первая строка показывает начальные расстояния, вторая строка показывает расстояния, когда ребра (B, E), (D, B), (B, D) и (A, B) обрабатываются. Третья строка показывает расстояние при обработке (A, C). Четвертая строка показывает, что происходит, когда обрабатываются (D, C), (B, C) и (E, D).

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



Первая итерация гарантирует, что все самые короткие пути будут не длиннее пути в 1 ребро. Мы получаем следующие расстояния, когда будет совершен второй проход по всем ребрам (в последней строке показаны конечные значения).

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



Вторая итерация гарантирует, что все кратчайшие пути будут иметь длину не более 2 ребер. Алгоритм проходит по всем ребрам еще 2 раза. Расстояния минимизируются после второй итерации, поэтому третья и четвертая итерации не обновляют значения расстояний.

Алгоритм Флойда.

Рассмотрим граф G с вершинами V , пронумерованными от 1 до N . Алгоритм Флойда — Уоршелла сравнивает все возможные пути через граф между каждой парой вершин. Он может сделать это за $\Theta(|V|^3)$ сравнений в графе, даже если в графе может быть до $\Omega(|V|^2)$ ребер, и каждая комбинация ребер проверяется. Это достигается путем постепенного улучшения оценки кратчайшего пути между двумя вершинами, пока оценка не станет оптимальной.

Далее рассмотрим функцию $\text{shortestPath}(i, j, k)$, которая возвращает кратчайший возможный путь от i до j с использованием вершин только из множества $\{1, 2, \dots, k\}$ в качестве промежуточных точек на этом пути.

Теперь, учитывая эту функцию, наша цель — найти кратчайший путь от каждого i до каждого j , используя любую вершину в $\{1, 2, \dots, N\}$.

Для каждой из этих пар вершин $\text{shortestPath}(i, j, k)$ может быть либо

(1) путь, который не проходит через k (использует только вершины из набора $\{1, \dots, k - 1\}$),

или

(2) путь, который проходит через k (от i до k и затем от k до j , в обоих случаях используются только промежуточные вершины в $\{1, \dots, k - 1\}$).

Мы знаем, что лучший путь от i до j , это путь который использует только вершины с 1 по $k - 1$, определяется как $\text{shortestPath}(i, j, k - 1)$, и ясно, что если бы существовал лучший путь от i до k до j , тогда длина этого пути была бы цепочкой состоящей из самого короткого пути от i до k (только с использованием промежуточных вершин в $\{1, \dots, k - 1\}$) и кратчайшего пути от k до j (только с использованием промежуточных вершин в $\{1, \dots, k - 1\}$).

Если $w(i, j)$ — вес ребра между вершинами i и j , мы можем определить $\text{shortestPath}(i, j, k)$ в терминах следующей [рекурсивной](#) формулой:

базовый случай

$$\text{shortestPath}(i, j, 0) = w(i, j)$$

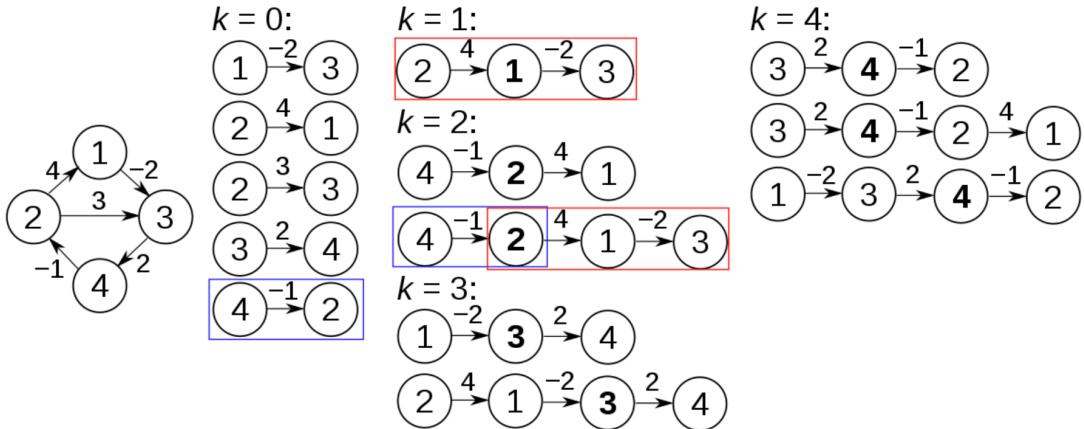
и рекурсивный случай

$$\begin{aligned} \text{shortestPath}(i, j, k) = \\ \min \left(\text{shortestPath}(i, j, k - 1), \right. \\ \left. \text{shortestPath}(i, k, k - 1) + \text{shortestPath}(k, j, k - 1) \right). \end{aligned}$$

Эта формула составляет основу алгоритма Флойда — Уоршелла. Алгоритм работает, сначала вычисляя $\text{shortestPath}(i, j, k)$ для всех пар (i, j) для $k = 1$, а затем $k = 2$, и так далее. Этот процесс продолжается до тех пор, пока $k = N$ не будет найден кратчайший путь для всех пар (i, j) с использованием любых промежуточных вершин. Псевдокод для этой базовой версии следующий:

```
let dist be a  $|V| \times |V|$  массив минимальных расстояний, инициализированный как  $\infty$ 
(бесконечность)
for each edge  $(u, v)$  do
    dist[u][v]  $\leftarrow w(u, v)$  // Вес ребра  $(u, v)$ 
for each vertex  $v$  do
    dist[v][v]  $\leftarrow 0$ 
for  $k$  from 1 to  $|V|$ 
    for  $i$  from 1 to  $|V|$ 
        for  $j$  from 1 to  $|V|$ 
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j]  $\leftarrow$  dist[i][k] + dist[k][j]
            end if
```

Алгоритм выше выполняется на графе слева внизу:



До первой рекурсии внешнего цикла, обозначенного выше $k = 0$, единственные известные пути соответствуют отдельным ребрам в графе. При $k = 1$ находятся пути, проходящие через вершину 1: в частности, найден путь $[2,1,3]$, заменяющий путь $[2,3]$, который имеет меньше ребер, но длиннее (с точки зрения веса). При $k = 2$ находятся пути, проходящие через вершины 1,2. Красные и синие прямоугольники показывают, как путь $[4,2,1,3]$ собирается из двух известных путей $[4,2]$ и $[2,1,3]$, встреченных в предыдущих итерациях, с 2 на пересечении. Путь $[4,2,3]$ не рассматривается, потому что $[2,1,3]$ — это кратчайший путь, встреченный до сих пор от 2 до 3. При $k = 3$ пути, проходящие через вершины 1,2,3 найдены. Наконец, при $k = 4$ находятся все кратчайшие пути.

Матрица расстояний на каждой итерации k , обновленные расстояния выделены **жирным шрифтом**, будет иметь вид:

		j				
		1	2	3	4	
i	$k = 0$	1	0	∞	-2	∞
	$k = 1$	1	0	∞	-2	∞
	$k = 2$	1	0	∞	-2	∞
	$k = 3$	1	0	∞	-2	0
i		1	0	∞	2	∞
		2	4	0	2	∞
		3	∞	∞	0	2
		4	∞	-1	∞	0
i		1	0	-1	∞	0
		2	4	0	2	4
		3	5	1	0	2
		4	3	-1	1	0
i		1	0	-1	-2	0
		2	4	0	2	4
		3	5	1	0	2
		4	3	-1	1	0

Алгоритм A^* .

Представим, что нам необходимо попасть из точки А в точку В. Прямой путь между этими двумя точками разделён стеной, как показано на рисунке 1.

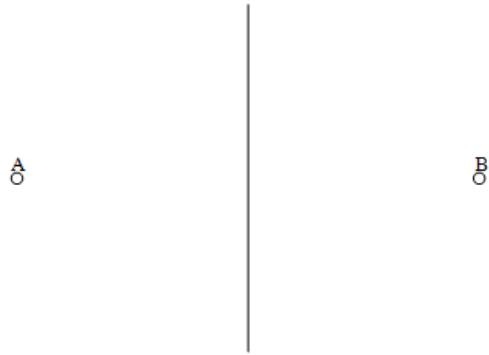


Рисунок 1

Для упрощения области поиска пути, её необходимо разделить на сетку с квадратными ячейками (смотри рисунок 2). Это позволит выразить область поиска пути через двумерный массив, каждый элемент которого будет представлять одну из клеток получившейся сетки. Одним из значений каждой ячейки массива будет проходимость соответствующей ей клетки (проходима или непроходима).

	0	1	2	3	4	5	6
0							
1							
2		A				B	
3							
4							

Рисунок 2

Для нахождения пути необходимо определить, какие именно клетки нужны для перемещения из точки А в точку В. Как только путь будет найден, можно начинать перемещение из центра клетки А в центр следующей клетки пути, до тех пор, пока не будет достигнута конечная точка В. Замечание: Сетка не обязательно строится из квадратных ячеек. Она может быть построена из прямоугольных ячеек, из шестиугольных ячеек, из треугольных ячеек, или из любых других ячеек. При этом центральные точки ячеек называют вершинами. Вершины могут располагаться как в центре, так и вдоль граней, или ещё где-нибудь.

Поиск пути начинаем со следующих шагов:

1. Добавляем стартовую клетку, где находится точка А, в «открытый список». В данный момент в этом списке будет находиться только одна ячейка, но позже в него будут добавляться и другие ячейки. Клетки, находящиеся в открытом списке это клетки, которые необходимо проверить и решить, будут ли они являться частью искомого пути к конечной клетке.
2. Ищем проходимые клетки, граничащие со стартовой клеткой, игнорируя непроходимые клетки (со стенами, водой и прочим), и добавляем их в открытый список. Для каждой из этих клеток сохраняем клетку А как «родительскую».
3. Удаляем стартовую клетку А с открытого списка и добавляем её в «закрытый список» клеток, которые больше не нужно проверять.

После этих шагов должно получиться нечто похожее на то, что изображено на рисунке 3. На нём стартовая клетка выделена оранжевым цветом для отображения того, что она находится в закрытом списке. Все соседние клетки в данный момент находятся в открытом списке, – они выделены синим цветом. Каждая из этих клеток имеет указатель, направленный на родительскую клетку, которая в данном случае является стартовой клеткой А.

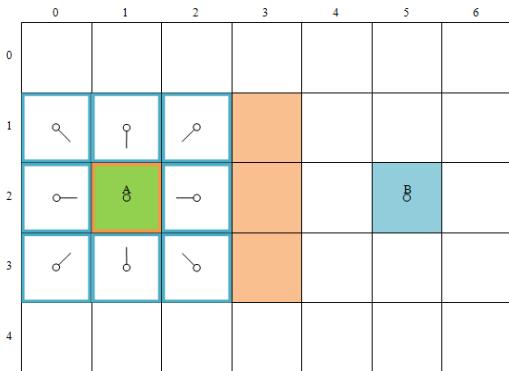


Рисунок 3

Далее необходимо выбрать одну клетку из находящихся в открытом списке клеток, и, практически, повторить вышеописанный процесс. Но следующая клетка из открытого списка, выбирается не случайно, а в зависимости от её величины F.

Величина F вычисляется по формуле 1:

$$F = G + H \quad (1)$$

где

- G – энергия, затрачиваемая на передвижение из стартовой клетки A в текущую рассматриваемую клетку, следуя найденному пути к этой клетке;
- H – примерное количество энергии, затрачиваемое на передвижение от текущей клетки до целевой клетки B. Изначально эта величина равна предположительному значению, такому, что если бы мы шли напрямую, игнорируя препятствия (но исключив диагональные перемещения). В процессе поиска она корректируется в зависимости от встречающихся на пути преград.

Обычно, энергия, затрачиваемая на прохождение в соседнюю клетку по горизонтали, берётся равной 10 единицам, а по диагонали – 14 единицам. Для вычисления величины G текущей рассматриваемой клетки, необходимо величину G её родительской клетки сложить с 10 или 14 (в зависимости от диагонального или ортогонального расположения текущей клетки относительно родительской клетки). Величина H обычно вычисляется методом Манхэттена. Суть его заключается в том, чтобы сосчитать общее количество клеток, необходимых для достижения целевой клетки B, от текущей рассматриваемой клетки, причём игнорируя диагональные перемещения между клетками, а также любые препятствия. Затем полученное количество умножается на 10. Рассчитав величину F по формуле 1, для всех клеток в открытом списке, получим результат похожий на то, что изображено на рисунке 4.

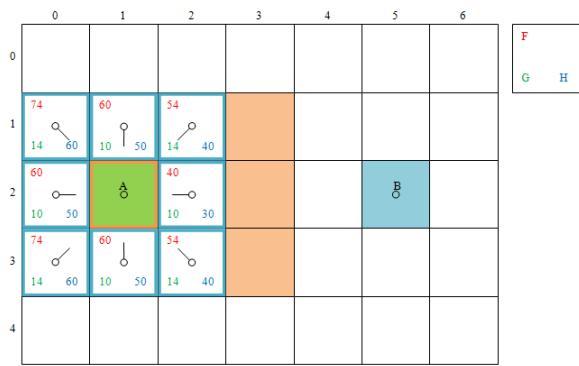


Рисунок 4

Клетки, расположенные ортогонально к стартовой клетке, имеют значение G равное 10, а клетки, расположенные диагонально к стартовой клетке – G равное 14. Значение H равняется Манхэттенскому расстоянию от центра текущей клетки до центра целевой клетки B, умноженное на 10. Например, для клетки с индексами [2 , 2], расстояние от её центра до центра целевой – 3 клетки (рисунок 5).

	0	1	2	3	4	5	6
0							
1	74 14 60	60 10 50	54 14 40				
2	60 10 50	8 60	40 10 30			B 8	
3	74 14 60	60 10 50	54 14 40				
4							

Рисунок 5

А для клеток с индексами [1 , 0] и [3 , 0], расстояние от их центра до центра целевой клетки – 6 клеток (рисунок 6).

	0	1	2	3	4	5	6
0							
1	74 14 60	60 10 50	54 14 40				
2	60 10 50	8 60	40 10 30			B 8	
3	74 14 60	60 10 50	54 14 40				
4							

Рисунок 6

Величина F для каждой клетки вычисляется по формуле 1, как сумма величин G и H. Для продолжения поиска кратчайшего пути выбирается ячейка, из открытого списка, с наименьшим значением F. И для этой клетки выполняются следующие действия (продолжение действий описанных выше):

1. Выбранную из открытого списка клетку удаляем из него и добавляем в закрытый список.
2. Добавляем в открытый список все соседние к ней клетки, если они еще не находятся в нём (при этом игнорируя непроходимые клетки и клетки, которые содержатся в закрытом списке). Предварительно указав, что текущая клетка является родительской для клеток, добавленных в открытый список, а также вычислив их значения G, H и F.
3. Если соседняя клетка уже находится в открытом списке, то сравниваем значение величин G у клетки в открытом списке и текущей проверяемой клетки. Если прежнее значение (в открытом списке) меньше нового, то ничего не делаем. В обратном случае, у клетки в открытом списке меняем значение G на новое, также меняем указатель на родителя, чтобы он указывал на текущую проверяемую клетку.

Рассмотрим описанные шаги. Сейчас в открытом списке находится 8 клеток, а стартовая клетка – в закрытом списке. Из открытого списка следующей клеткой для рассмотрения будет выбрана клетка с наименьшим значением F – это клетка с индексами [2 , 2]. Смотри рисунок 7.

	0	1	2	3	4	5	6
0							
1	74 14 60	60 10 50	54 14 40				
2	60 10 50	8 60	40 10 30			B 8	
3	74 14 60	60 10 50	54 14 40				
4							

Рисунок 7

Вначале текущую клетку (с индексами [2 , 2]) удаляем из открытого списка, и помещаем в закрытый список (поэтому на рисунке 7 она отмечена оранжевым цветом). Затем проверяем соседние клетки. Четыре из них, игнорируем, – это три непроходимые клетки стены и стартовая клетка, находящаяся в закрытом списке. Оставшиеся четыре клетки уже расположены в открытом списке, а значит необходимо сравнить их значения G со значением G таким, что если бы мы к ним дошли через текущую клетку. Значение G у клетки ниже текущей (с индексами [3 , 2]) равно 14, а значение G, полученное при проходе через текущую клетку равно 20 ($G = 10$ у текущей клетки, и плюс 10 путь до клетки ниже текущей). Значение 14 меньше 20, поэтому значение G у клетки ниже текущей не нужно обновлять. У клетки слева внизу (с индексами [3 , 1]) $G = 10$, а значение G, полученное при проходе через текущую клетку 24 ($G = 10$ у текущей клетки, и плюс 14 – путь от текущей до проверяемой клетки). 24 больше 10, значит обновлять значение G у этой клетки не нужно. Соответственно делаются проверки и для клетки выше текущей (с индексами [1 , 2]), и для клетки слева вверху от текущей клетки (с индексами [1 , 1]). После того, как все соседние клетки рассмотрены, можно двигаться далее. На данный момент в открытом списке находятся 7 клеток, две из которых имеют одинаковое наименьшее значение F равное 54. Какую клетку выбрать следующей текущей из этих двух клеток, для алгоритма не имеет значения, поэтому представим, что случайным образом выбрали клетку находящуюся справа внизу от стартовой клетки (с индексами [3 , 2]), как показано на рисунке 8.

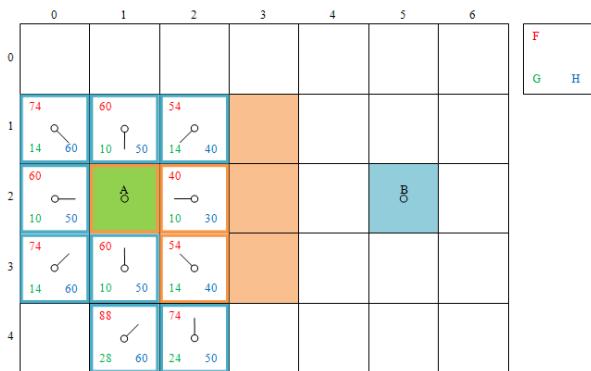


Рисунок 9

Проверяя клетки, соседние к текущей клетке, с индексами [2 , 3] и [3 , 3], игнорируем, так как они непроходимы. Клетку с индексами [2 , 2] и стартовую клетку также игнорируем, – они находятся в закрытом списке. Клетку с индексами [4 , 3] также игнорируем, по причине того, что к ней невозможно добраться без среза угла ближайшей стены. – Сначала необходимо перейти на клетку с индексами [4 , 2], а потом уже переходить на клетку с индексами [4 , 3]. (Правило запрещающее срезать углы у препятствий необязательно к выполнению, его применение зависит от расположения ваших вершин.) Клетка с индексами [3 , 1] уже находится в открытом списке, поэтому сравниваем её значение F со значением F таким, что если бы мы пришли на неё через текущую клетку. Это значения 60 и 64, соответственно, а значит, данные проверяемой клетки не нужно обновлять. Клетки с индексами [4 , 1] и [4 , 2] добавляем в открытый список, предварительно вычислив их значения величин G, H и F, а также установив указатель на родительскую клетку (как проиллюстрировано на рисунке 8). Повторяем описанную выше методику поиска пути до тех пор, пока не добавим целевую клетку в открытый список. Следующая клетка в открытом списке с наименьшим значением F, равным 54, это клетка с индексами [1 , 2]. Удаляем её из открытого списка, добавляем в закрытый список, и проверяем её соседей (при этом добавляем две клетки с индексами [0 , 1] и [0 , 2]). Итог операций проиллюстрирован на рисунке 9.

	0	1	2	3	4	5	6	
0	88 28 60	74 14 60	24 10 50					F G H
1	74 14 60	60 10 50	54 14 40					
2	60 10 50	88 28 60	40 10 30					B
3	74 14 60	60 10 50	54 14 40					
4		88 28 60	74 14 50					

Рисунок 9

Сейчас в открытом списке находится две клетки с наименьшим значением F равным 60. Случайным образом выбираем клетку с индексами [3, 1]. Удаляем её из открытого списка, добавляем в закрытый список. Проверяем её соседей (добавляем в открытый список клетку с индексами [4, 0] и у клетки с индексами [4, 1], уже находящейся в открытом списке, обновляем значение F и меняем указатель на родителя, теперь он ссылается на текущую клетку, с индексами [3, 1]). Смотри рисунок 10.

	0	1	2	3	4	5	6	
0	88 28 60	74 14 60	24 10 50					F G H
1	74 14 60	60 10 50	54 14 40					
2	60 10 50	88 28 60	40 10 30					B
3	74 14 60	60 10 50	54 14 40					
4	94 24 70	80 20 60	74 24 50					

Рисунок 10

Следующую из открытого списка обрабатываем клетку с индексами [2, 0]. Удаляем её из открытого списка и добавляем в закрытый список. При этом никакие из её соседних клеток не нуждаются в обновлении (смотри рисунок 11).

	0	1	2	3	4	5	6	
0	88 28 60	74 14 60	24 10 50					F G H
1	74 14 60	60 10 50	54 14 40					
2	60 10 50	88 28 60	40 10 30					B
3	74 14 60	60 10 50	54 14 40					
4	94 24 70	80 20 60	74 24 50					

Рисунок 11

Далее обрабатываем клетку с индексами [1, 1]. Удаляем её из открытого списка, добавляем в закрытый список. При этом обновляем данные её соседней клетки с индексами [0, 1] и добавляем в открытый список клетку с индексами [0, 0]. Смотри рисунок 12.

	0	1	2	3	4	5	6
0	94 24 70	80 20 60	74 24 50	○			
1	74 14 60	60 10 50	54 14 40				
2	60 10 50	◊	40 10 30			◊	
3	74 14 60	60 10 50	54 14 40				
4	94 24 70	80 20 60	74 24 50				

Рисунок 12

Следующие клетки в открытом списке имеют наименьшее значение F равное 74. Случайным образом выбираем клетку с индексами [4 , 2]. Удаляем её из открытого списка, добавляем в закрытый список. Проверяем её соседей – добавляем в открытый список клетку с индексами [4 , 3]. Смотри рисунок 13.

	0	1	2	3	4	5	6
0	94 24 70	80 20 60	74 24 50				
1	74 14 60	60 10 50	54 14 40				
2	60 10 50	◊	40 10 30			◊	
3	74 14 60	60 10 50	54 14 40				
4	94 24 70	80 20 60	74 24 50	74 34 40			

Рисунок 13

Предположим, что следующей клеткой с наименьшим значением F выбрана клетка ближе к той, которую проверяли перед этим – клетка с индексами [4 , 3]. Удаляем её из открытого списка, добавляем в закрытый список. Обрабатываем её соседние клетки (рисунок 14).

	0	1	2	3	4	5	6
0	94 24 70	80 20 60	74 24 50				
1	74 14 60	60 10 50	54 14 40				
2	60 10 50	◊	40 10 30			◊	
3	74 14 60	60 10 50	54 14 40				
4	94 24 70	80 20 60	74 24 50	74 34 40	74 44 30		

Рисунок 14

Далее, выбираем клетку с индексами [4 , 4], удаляем её из открытого списка, добавляем в закрытый список. Обрабатываем её соседние клетки – добавляем три клетки в открытый список (клетки с индексами [3 , 4], [3 , 5] и [5 , 4]) (рисунок 15).

	0	1	2	3	4	5	6
0	94 24 70	80 20 60	74 24 50				
1	74 14 60	60 10 50	54 14 40				
2	60 10 50	8 10 50	40 10 30			B 68	
3	74 14 60	60 10 50	54 14 40		74 54 20	68 58 10	
4	94 24 70	80 20 60	74 24 50	74 34 40	74 44 30	74 54 20	

Рисунок 15

Следующая клетка с наименьшим значением F это клетка с индексами [3 , 5]. Выбираем её, удаляем из открытого списка, добавляем в закрытый список. Обрабатываем её соседние клетки, при этом в открытый список добавляются клетки с индексами [2 , 4], [3 , 6], [2 , 6] и целевая клетка с индексами [2 , 5]. Смотри рисунок 16.

	0	1	2	3	4	5	6
0	94 24 70	80 20 60	74 24 50				
1	74 14 60	60 10 50	54 14 40				
2	60 10 50	8 10 50	40 10 30		82 72 10	B 68 00	82 72 10
3	74 14 60	60 10 50	54 14 40		74 54 20	68 58 10	88 68 20
4	94 24 70	80 20 60	74 24 50	74 34 40	74 44 30	74 54 20	102 72 30

Рисунок 16

Целевая клетка находится в открытом списке, а это значит, что был найден путь от стартовой до финишной клетки. Теперь следуя указателям на родителей можно пройти от финишной клетки до стартовой клетки, а сохранённый путь в обратном направлении – путь от стартовой до целевой клетки, это и будет найденный кратчайший путь (рисунок 17).

	0	1	2	3	4	5	6
0	94 24 70	80 20 60	74 24 50				
1	74 14 60	60 10 50	54 14 40				
2	60 10 50	8 10 50	40 10 30		82 72 10	B 68 00	82 72 10
3	74 14 60	60 10 50	54 14 40		74 54 20	68 58 10	88 68 20
4	94 24 70	80 20 60	74 24 50	74 34 40	74 44 30	74 54 20	102 72 30

Рисунок 17

Пошаговое представление метода:

- Добавить стартовую клетку в открытый список (при этом её значения G, H и F равны 0).
- Повторять следующие шаги:
 - Ищем в открытом списке клетку с наименьшим значением величины F, делаем её текущей.
 - Удаляем текущую клетку из открытого списка и помещаем в закрытый список.
 - Для каждой из соседних, к текущей клетке, клеток:
 - Если клетка непроходима или находится в закрытом списке, игнорируем её.
 - Если клетка не в открытом списке, то добавляем её в открытый список, при этом рассчитываем для неё значения G, H и F, и также устанавливаем ссылку родителя на текущую клетку.

- Если клетка находится в открытом списке, то сравниваем её значение G со значением G таким, что если бы к ней пришли через текущую клетку. Если сохранённое в проверяемой клетке значение G больше нового, то меняем её значение G на новое, пересчитываем её значение F и изменяюм указатель на родителя так, чтобы она указывала на текущую клетку.
 - Останавливаемся, если:
 - В открытый список добавили целевую клетку (в этом случае путь найден).
 - Открытый список пуст (в этом случае к целевой клетке пути не существует).
3. Сохраняем путь, двигаясь назад от целевой точки, проходя по указателям на родителей до тех пор, пока не дойдём до стартовой клетки.

8. Недетерминированные конечные автоматы, различные варианты определения. Детерминированные конечные автоматы. Их эквивалентность. Машина Тьюринга.

Конечный автомат (КА) в теории алгоритмов — математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных. Является частным случаем абстрактного дискретного автомата, число возможных внутренних состояний которого конечно.

При работе на вход КА последовательно поступают входные воздействия, а на выходе КА формирует выходные сигналы. Обычно под входными воздействиями принимают подачу на вход автомата символов одного алфавита, а на выход КА в процессе работы выдаёт символы в общем случае другого, возможно даже не пересекающегося со входным, алфавита.

Помимо конечных автоматов существуют и бесконечные дискретные автоматы — автоматы с бесконечным числом внутренних состояний.

Переход из одного внутреннего состояния КА в другое может происходить не только от внешнего воздействия, но и самопроизвольно.

Общее формальное описание [[править](#) | [править код](#)]

Формально КА определяется в виде упорядоченной пятёрки элементов некоторых множеств:

$$A = (S, X, Y, \delta, \lambda),$$

где S — конечное множество [состояний](#) автомата;

X, Y — конечные входной и выходной алфавиты соответственно, из которых формируются строки, считываемые и выдаваемые автоматом;

$\delta : S \times X \rightarrow S$ — функция переходов;

$\lambda : S \times X \rightarrow Y$ — функция выходов.



Абстрактный автомат с некоторым выделенным состоянием s_0 (это состояние называют начальным состоянием) называется инициальным автоматом. Так как каждый КА имеет конечное число состояний, и любое из его состояний может быть назначено начальным состоянием, одному и тому же автомата соответствует N инициальных автоматов, N – число внутренних состояний КА. Таким образом, абстрактный КА определяет семейство инициальных автоматов. Если не указано начальное состояние, то поведение автомата всегда недетерминировано, выходное слово автомата зависит от начального состояния, поэтому полное детерминированное описание автомата будет:

$$A = (S, X, Y, \delta, \lambda, s_0).$$

Различают два класса КА: автоматы Мура – КА, у которых выходной сигнал зависит только от внутреннего состояния, по рисунку у автомата Мура нет связи от входа $x(t)$ к функции выхода λ , и автоматы Мили – выходной сигнал зависит как от внутреннего состояния, так и от состояния входа.

Недетерминированный конечный автомат можно определить как пятерку $(N, \Sigma, \delta, q_0, F)$,

где

- N – конечное множество состояний,
- Σ – алфавит входных символов,
- δ – функция перехода, которая задает правила перехода между состояниями,
- q_0 – начальное состояние,
- F – множество конечных состояний.

Другие способы задания функционирования КА

4. Диаграмма состояний (или иногда граф переходов) – графическое представление множества состояний и функции переходов. Представляет собой размеченный ориентированный граф, вершины которого – состояния КА, дуги – переходы из одного состояния в другое, а метки дуг – символы, по которым осуществляется переход из одного состояния в другое. Если переход из состояния q_1 в q_2 может быть осуществлён по одному из нескольких символов, то все они должны быть надписаны над дугой диаграммы.
5. Таблица переходов – табличное представление функции δ . Обычно в такой таблице каждой строке соответствует одно состояние (исходное и результатное), а столбцу – один допустимый входной символ. В ячейке на пересечении строки и столбца записывается результатное состояние, в которое должен перейти автомат, если в данном исходном состоянии он считал данный входной символ. Пример таблицы переходов для автомата, заданного в виде графа на рисунке 1, приведён справа.

Исходное состояние	Следующее состояние		
	Входной символ a	Входной символ b	Любой другой символ
p0	p1	p0	p0
p1	p1	p2	p1
p2	p3	p4	p2
p3	p3	p5	p3
p4	p4	p4	p4
p5	p3	p5	p5

Конечные автоматы подразделяются на детерминированные и недетерминированные.

- Детерминированным конечным автоматом (ДКА) называется такой автомат, в котором нет дуг с меткой ϵ (предложение, не содержащее ни одного символа), и из любого состояния по любому символу возможен переход не более, чем в одно состояние.
- Недетерминированный конечный автомат (НКА) является обобщением детерминированного. Недетерминированность автоматов может достигаться двумя способами: либо могут существовать переходы из состояния в состояние, вызываемые пустой цепочкой символов, то есть самопроизвольные переходы без внешних воздействий, либо из одного состояния КА может переходить в разные состояния под воздействием одного и того же символа.

Эквивалентность НКА и ДКА заключается в том, что для любого НКА существует эквивалентный ДКА, который принимает тот же язык. Это означает, что любой язык, который может быть распознан НКА, также может быть распознан и ДКА.

Теорема о детерминизации утверждает, что для любого конечного автомата может быть построен эквивалентный ему детерминированный конечный автомат (два конечных автомата называют эквивалентными, если их языки совпадают). Однако поскольку количество состояний в эквивалентном ДКА в худшем случае растёт экспоненциально с ростом количества состояний исходного НКА, на практике подобная детерминизация не всегда возможна. Кроме того, конечные автоматы с выходом в общем случае не поддаются детерминизации.

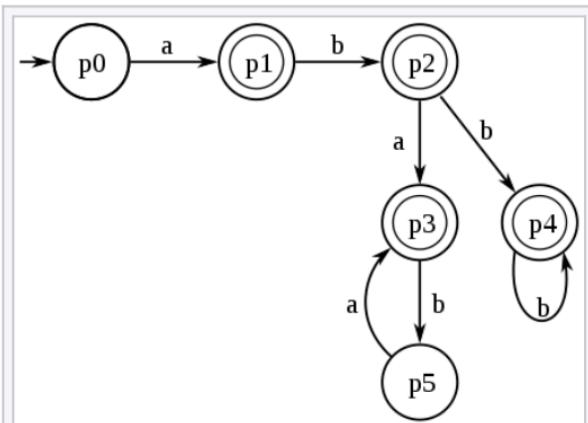


Рисунок 1. Пример графа
переходов детерминированного КА.

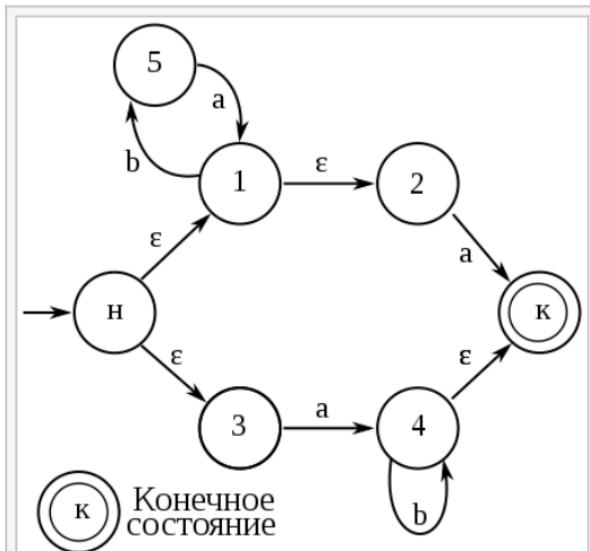


Рисунок 2. Пример графа
переходов недетерминированного
КА с самопроизвольными
переходами

МашинаТьюринга - это математическая модель вычислительного устройства, предложенная Аланом Тьюрингом. МашинаТьюринга состоит из бесконечной ленты, на которой записаны символы, головки чтения/записи и конечного числа состояний. МашинаТьюринга способна выполнять различные операции на ленте в зависимости от текущего символа на ленте и текущего состояния. МашинаТьюринга является универсальной моделью вычислений и может решать широкий спектр задач.

