

16. Дискретное преобразование Фурье. Алгоритм быстрого преобразования Фурье. Использование быстрого преобразования Фурье при реализации операций над многочленами.

Дискретное преобразование Фурье (DFT, Discrete Fourier Transform) — это одно из [преобразований Фурье](#), широко применяемых в [алгоритмах цифровой обработки сигналов](#) (его модификации применяются в сжатии звука в [MP3](#), сжатии изображений в [JPEG](#) и др.), а также в других областях, связанных с анализом частот в дискретном (к примеру, оцифрованном аналоговом) сигнале. Дискретное преобразование Фурье требует в качестве входа дискретную функцию. Такие функции часто создаются путём [дискретизации](#) (выборки значений из непрерывных функций). Дискретные преобразования Фурье помогают решать частные дифференциальные уравнения и выполнять такие операции, как [свёртки](#). Дискретные преобразования Фурье также активно используются в статистике, при анализе временных рядов. Существуют многомерные дискретные преобразования Фурье.

Формулы преобразований

Прямое преобразование:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \quad k = 0, \dots, N-1$$

Обратное преобразование:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N} kn} \quad n = 0, \dots, N-1.$$

Обозначения:

- N — количество значений сигнала, измеренных за период, а также количество компонент разложения;
- $x_n, \quad n = 0, \dots, N-1,$ — измеренные значения сигнала (в дискретных временных точках с номерами $n = 0, \dots, N-1$, которые являются входными данными для прямого преобразования и выходными для обратного;
- $X_k, \quad k = 0, \dots, N-1, \quad N$ — [комплексных амплитуд](#) синусоидальных сигналов, слагающих исходный сигнал; являются выходными данными для прямого преобразования и входными для обратного; поскольку амплитуды комплексные, то по ним можно вычислить одновременно и амплитуду, и фазу;

$$\frac{|X_k|}{N}$$

- — обычная (вещественная) амплитуда k -го синусоидального сигнала;
- $\arg(X_k)$ — фаза k -го синусоидального сигнала ([аргумент комплексного числа](#));

- k — индекс частоты. Частота k -го сигнала равна $\frac{k}{T}$, где T — период времени, в течение которого брались входные данные.

Из последнего видно, что преобразование раскладывает сигнал на синусоидальные составляющие (которые называются гармониками) с частотами от N колебаний за период до одного колебания за период.

Поскольку частота дискретизации сама по себе равна N отсчётов за период, то высокочастотные составляющие не могут быть корректно отображены — возникает [муаровый эффект](#). Это приводит к тому, что вторая половина из N комплексных амплитуд, фактически, является зеркальным отображением первой и не несёт дополнительной информации.

Вывод преобразования

Рассмотрим некоторый периодический сигнал $x(t)$ с периодом равным T . Разложим его в [ряд Фурье](#):

$$x(t) = \sum_{k=-\infty}^{+\infty} c_k e^{i\omega_k t}, \quad \omega_k = \frac{2\pi k}{T}$$

Проведем дискретизацию сигнала так, чтобы на периоде было N отсчетов.

Дискретный сигнал представим в виде отсчетов: $x_n = x(t_n)$, где $t_n = \frac{n}{N}T$, тогда эти отсчеты через ряд Фурье запишутся следующим образом:

$$x_n = \sum_{k=-\infty}^{+\infty} c_k e^{i\omega_k t_n} = \sum_{k=-\infty}^{+\infty} c_k e^{\frac{2\pi i}{N} kn}$$

$$e^{\frac{2\pi i}{N}(k+mN)n} = e^{\frac{2\pi i}{N} kn}$$

Используя соотношение: $e^{\frac{2\pi i}{N}(k+mN)n} = e^{\frac{2\pi i}{N} kn}$, получаем:

$$x_n = \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N} kn}, \quad X_k = \sum_{l=-\infty}^{+\infty} c_{k+lN}$$

где

Таким образом, мы получили **обратное дискретное преобразование Фурье**.

Умножим теперь скалярно выражение для x_n на $e^{-\frac{2\pi i}{N} mn}$ и получим:

$$\sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} mn} = \sum_{k=0}^{N-1} \sum_{n=0}^{N-1} X_k e^{\frac{2\pi i}{N}(k-m)n} = \sum_{k=0}^{N-1} X_k \frac{1 - e^{2\pi i(k-m)}}{1 - e^{\frac{2\pi i(k-m)}{N}}} = \sum_{k=0}^{N-1} X_k N \delta_{km}$$

Здесь использованы: а) выражение для суммы конечного числа членов (экспонент) геометрической прогрессии, и б) выражение символа Кронекера как предела отношения функций Эйлера для комплексных чисел. Отсюда следует, что:

$$X_k = \frac{1}{N} \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}$$

Эта формула описывает **прямое дискретное преобразование Фурье**.

$$\frac{1}{N}$$

В литературе принято писать множитель $\frac{1}{N}$ в обратном преобразовании, и поэтому обычно пишут формулы преобразования в следующем виде:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}, \quad x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N} kn}$$

Матричное представление

Дискретное преобразование Фурье является линейным преобразованием,

которое переводит вектор временных отсчётов \vec{x} в вектор спектральных отсчётов той же длины. Таким образом, преобразование может быть реализовано как умножение квадратной матрицы на вектор:

$$\vec{X} = \hat{A} \vec{x}$$

матрица A имеет вид:

$$\hat{A} = \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & e^{-\frac{2\pi i}{N}} & e^{-\frac{4\pi i}{N}} & e^{-\frac{6\pi i}{N}} & \dots & e^{-\frac{2\pi i}{N}(N-1)} \\ 1 & e^{-\frac{4\pi i}{N}} & e^{-\frac{8\pi i}{N}} & e^{-\frac{12\pi i}{N}} & \dots & e^{-\frac{2\pi i}{N}2(N-1)} \\ 1 & e^{-\frac{6\pi i}{N}} & e^{-\frac{12\pi i}{N}} & e^{-\frac{18\pi i}{N}} & \dots & e^{-\frac{2\pi i}{N}3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & e^{-\frac{2\pi i}{N}(N-1)} & e^{-\frac{2\pi i}{N}2(N-1)} & e^{-\frac{2\pi i}{N}3(N-1)} & \dots & e^{-\frac{2\pi i}{N}(N-1)^2} \end{pmatrix}$$

Элементы матрицы задаются следующей формулой:

$$A(m, n) = \exp \left(-2\pi i \frac{(m-1)(n-1)}{N} \right)$$

Свойства

1. линейность

$$ax(n) + by(n) \longleftrightarrow aX(k) + bY(k)$$

2. сдвиг по времени

$$x(n-m) \longleftrightarrow X(k) e^{-\frac{2\pi i}{N} km}$$

3. периодичность

$$X(k+rN) = X(k), r \in \mathbb{Z}$$

4. выполняется [Теорема Парсеваля](#)

5. обладает спектральной плотностью

$$S(k) = |X(k)|^2$$

$$x(n) \in \mathbb{R}$$

6.

$$X(0) \in \mathbb{R}$$

$$N \bmod 2 = 0 \Rightarrow X(N/2) \in \mathbb{R}$$

Стоит отметить, что нулевая гармоника является суммой значений сигнала.

Алгоритм быстрого преобразования Фурье.

Быстрое преобразование Фурье (БПФ, FFT) — это [алгоритм](#) быстрого вычисления [дискретного преобразования Фурье](#) (ДПФ). То есть, алгоритм

вычисления за количество действий, меньшее чем $O(N^2)$, требуемых для прямого (по формуле) вычисления ДПФ. Иногда под БПФ понимается один из быстрых алгоритмов, называемый алгоритмом прореживания по частоте/времени или алгоритмом по основанию 2, имеющего сложность $O(N \log(N))$

Основной алгоритм

Покажем как выполнить дискретное преобразование Фурье за $O(N(p_1 + \dots + p_n))$ действий при $N = p_1 p_2 \dots p_n$. В частности, при $N = 2^n$ понадобится $O(N \log(N))$ действий.

Дискретное преобразование Фурье преобразует набор чисел a_0, \dots, a_{n-1} в

набор чисел b_0, \dots, b_{n-1} , такой, что $b_i = \sum_{j=0}^{n-1} a_j \varepsilon^{ij}$, где $\varepsilon^n = 1$ и $\varepsilon^k \neq 1$ при $0 < k < n$.

. Алгоритм быстрого преобразования Фурье применим к любым коммутативным ассоциативным [кольцам](#) с единицей. Чаще всего этот

алгоритм применяют к полю [комплексных чисел](#) (с $\varepsilon = e^{2\pi i/n}$) и к [кольцам вычетов](#).

Основной шаг алгоритма состоит в сведении задачи для N чисел к задаче для $p = N/q$ числам, где q — делитель N . Пусть мы уже умеем решать задачу для N/q чисел. Применим преобразование Фурье к наборам $a_i, a_{q+i}, \dots, a_{q(p-1)+i}$

для $i = 0, 1, \dots, q-1$. Покажем теперь, как за $O(Np)$ действий решить

$$b_i = \sum_{j=0}^{q-1} \varepsilon^{ij} \left(\sum_{k=0}^{p-1} a_{kq+j} \varepsilon^{kjq} \right)$$

исходную задачу. Заметим, что $i \pmod{p}$. Выражения в

скобках нам уже известны — это j -тое число после преобразования

Фурье j -той группы. Таким образом, для вычисления каждого b_i нужно $O(q)$

действий, а для вычисления всех b_i — $O(Nq)$ действий, что и требовалось получить.

Обратное преобразование Фурье

Для обратного преобразования Фурье можно применять алгоритм прямого преобразования Фурье — нужно лишь использовать ε^{-1} вместо ε (или применить операцию комплексного сопряжения в начале к входным данным, а затем к результату, полученному после прямого преобразования Фурье) и окончательный результат поделить на N .

Общий случай

Общий случай может быть сведён к предыдущему. Пусть $4N > 2^k \geq 2N$.

$$b_i = \varepsilon^{-i^2/2} \sum_{j=0}^{N-1} \varepsilon^{(i+j)^2/2} \varepsilon^{-j^2/2} a_j$$

Заметим, что

. Обозначим

$$\bar{a}_i = \varepsilon^{-i^2/2} a_i, \bar{b}_i = \varepsilon^{i^2/2} b_i, c_i = \varepsilon^{(2N-2-i)^2/2} \quad \bar{b}_i = \sum_{j=0}^{2N-2-i} \bar{a}_j c_{2N-2-i-j}$$

. Тогда

, если

$$\bar{a}_i = 0 \quad i \geq N$$

положить при

Таким образом задача сведена к вычислению [свёртки](#), но это можно сделать

с помощью трёх преобразований Фурье для 2^k элементов. Выполняем прямое

преобразование Фурье для $\{\bar{a}_i\}_{i=0}^{2^k-1}$ и $\{c_i\}_{i=0}^{2^k-1}$, перемножаем поэлементно результаты и выполняем обратное преобразование Фурье.

Вычисления всех \bar{a}_i и c_i требуют $O(N)$ действий, три преобразования Фурье требуют $O(N \log(N))$ действий, перемножение результатов преобразований

Фурье требует $O(N)$ действий, вычисление всех b_i зная значения свертки

требует $O(N)$ действий. Итого для дискретного преобразования Фурье

требуется $O(N \log(N))$ действий для любого N .

Этот алгоритм быстрого преобразования Фурье может работать над кольцом $2N$ только тогда, когда известны [первообразные корни](#) из единицы степеней 2^k и N .

Вывод преобразования из ДПФ

Дискретное преобразование Фурье для вектора \vec{x} , состоящего из N элементов, имеет вид:

$$\vec{X} = \hat{A} \vec{x}$$

$$a_N^{mn} = \exp\left(-2\pi i \frac{mn}{N}\right)$$

элементы матрицы \hat{A} имеют вид:

Пусть N чётно, тогда ДПФ можно переписать следующим образом:

$$X_m = \sum_{n=0}^{N-1} x_n a_N^{mn} = \sum_{n=0}^{N/2-1} x_{2n} a_N^{2nm} + \sum_{n=0}^{N/2-1} x_{2n+1} a_N^{(2n+1)m}$$

Коэффициенты a_N^{2nm} и $a_N^{(2n+1)m}$ можно переписать следующим образом ($M=N/2$):

$$a_N^{2nm} = \exp\left(-2\pi i \frac{2mn}{N}\right) = \exp\left(-2\pi i \frac{mn}{N/2}\right) = a_M^{nm}$$

$$a_N^{(2n+1)m} = \exp\left(-2\pi i \frac{m}{N}\right) a_M^{nm}$$

В результате получаем:

$$X_m = \sum_{n=0}^{M-1} x_{2n} a_M^{nm} + \exp\left(-2\pi i \frac{m}{N}\right) \sum_{n=0}^{M-1} x_{2n+1} a_M^{nm}$$

То есть дискретное преобразование Фурье от вектора, состоящего из N

отсчетов, свелось к линейной композиции двух ДПФ от $\frac{N}{2}$ отсчетов, и если для первоначальной задачи требовалось N^2 операций, то для полученной

композиции — $\frac{N^2}{2}$. Если M является степенью двух, то это разделение можно продолжать рекурсивно до тех пор, пока не дойдем до двухточечного преобразования Фурье, которое вычисляется по следующим формулам:

$$\begin{cases} X_0 = x_0 + x_1 \\ X_1 = x_0 - x_1 \end{cases}$$

Использование быстрого преобразования Фурье при реализации операций над многочленами.

17. Двоичные деревья поиска. Печать всех ключей. Поиск в двоичном дереве. Нахождение наибольшего и наименьшего элементов. Нахождение

следующего элемента. Добавление и удаление элемента.

Двоичное дерево поиска (англ. *binary search tree*, BST) — это двоичное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- Оба поддерева — левое и правое, являются двоичными деревьями поиска.
- У всех узлов левого поддерева произвольного узла X значения ключей данных *меньше*, нежели значение ключа данных узла X.
- У всех узлов правого поддерева произвольного узла X значения ключей данных *не меньше*, нежели значение ключа данных узла X.

Очевидно, данные в каждом узле должны обладать ключами, на которых определена операция сравнения *меньше*.

Как правило, информация, представляющая каждый узел, является записью, а не единственным полем данных. Однако, это касается реализации, а не природы двоичного дерева поиска.

Для целей реализации двоичное дерево поиска можно определить так:

- Двоичное дерево состоит из узлов (вершин) — записей вида (data, left, right), где data — некоторые данные привязанные к узлу, left и right — ссылки на узлы, являющиеся детьми данного узла - левый и правый сыновья соответственно. Для оптимизации алгоритмов конкретные реализации предполагают также определения поля parent в каждом узле (кроме корневого) - ссылки на родительский элемент.
- Данные (data) обладают ключом (key), на котором определена операция сравнения "меньше". В конкретных реализациях это может быть пара (key, value) - (ключ и значение), или ссылка на такую пару, или простое определение операции сравнения на необходимой структуре данных или ссылке на неё.
- Для любого узла X выполняются свойства дерева поиска: $\text{key}[\text{left}[X]] < \text{key}[X] \leq \text{key}[\text{right}[X]]$, т. е. ключи данных родительского узла больше ключей данных левого сына и нестрогие меньше ключей данных правого.

Основным преимуществом двоичного дерева поиска перед другими структурами данных является возможная высокая эффективность реализации основанных на нём алгоритмов поиска и сортировки.

Двоичное дерево поиска применяется для построения более абстрактных структур, таких как множества, мультимножества, ассоциативные массивы.

Основные операции в двоичном дереве поиска

Базовый интерфейс двоичного дерева поиска состоит из трех операций:

- FIND(K) — поиск узла, в котором хранится пара (key, value) с $\text{key} = K$.
- INSERT(K,V) — добавление в дерево пары (key, value) = (K, V).
- REMOVE(K) — удаление узла, в котором хранится пара (key, value) с $\text{key} = K$.

Этот абстрактный интерфейс является общим случаем, например, таких интерфейсов, взятых из прикладных задач:

- «Телефонная книжка» — хранилище записей (имя человека, его телефон) с операциями поиска и удаления записей по имени человека, и операцией добавления новой записи.
- Domain Name Server — хранилище пар (доменное имя, IP адрес) с операциями модификации и поиска.
- Namespase — хранилище имен переменных с их значениями, возникающее в трансляторах языков программирования.

По сути, двоичное дерево поиска — это структура данных, способная хранить таблицу пар (key, value) и поддерживающая три операции: FIND, INSERT, REMOVE. Кроме того, интерфейс двоичного дерева включает ещё три дополнительных операции обхода узлов дерева: INFIX_TRAVERSE, PREFIX_TRAVERSE и POSTFIX_TRAVERSE. Первая из них позволяет обойти узлы дерева в порядке неубывания ключей.

Печать всех ключей, входящих в дерево T с корнем root[T]:

- INORDER-TREE-WALK(*x*)
- 1 **if** *x* != NIL
- 2 **then** INORDER-TREE-WALK (*left*[*x*])
- 3 print *key*[*x*]
- 4 INORDER-TREE-WALK (*right*[*x*])
-

Поиск элемента (FIND)

Дано: дерево T и ключ K.

Задача: проверить, есть ли узел с ключом K в дереве T, и если да, то вернуть ссылку на этот узел.

Алгоритм:

- Если дерево пусто, сообщить, что узел не найден, и остановиться.
- Иначе сравнить K со значением ключа корневого узла X.
 - Если K=X, выдать ссылку на этот узел и остановиться.
 - Если K>X, рекурсивно искать ключ K в правом поддереве T.
 - Если K<X, рекурсивно искать ключ K в левом поддереве T.

TREE-SEARCH (*x*, *k*)

```
1 if x = NIL or k = key[x]  
2 then return x  
3 if k < key[x]  
4 then return TREE-SEARCH (left[x], k)  
5 else return TREE-SEARCH (right[x], k)
```

ITERATIVE-TREE-SEARCH (*x*,*k*)

```
1 while x != NIL and k!= key[x]  
2 do if k < key[x]  
3 then x = left[x]  
4 else x = right[x]  
5 return x
```

Нахождение минимума и максимума:

TREE-MINIMUM (*x*)

```
1 while left[x] != NIL  
2 do x = left[x]  
3 return x
```

TREE-MAXIMUM(*x*)

```
1 while right[x] !=NIL  
2 do x = right[x]  
3 return x
```

Нахождение следующего элемента:

TREE SUCCESSOR(x)

```
1 if  $right[x] \neq NIL$   
2 then return TREE-MINIMUM( $right[x]$ )  
3  $y = p[x]$   
4 while  $y \neq NIL$  and  $x = right[y]$   
5 do  $x = y$   
6  $y = p[y]$   
7 return  $y$ 
```

Добавление элемента (INSERT)

Дано: дерево T и пара (K, V) .

Задача: добавить пару (K, V) в дерево T .

Алгоритм:

- Если дерево пусто, заменить его на дерево с одним корневым узлом $((K, V), null, null)$ и остановиться.
- Иначе сравнить K с ключом корневого узла X .
 - Если $K \geq X$, рекурсивно добавить (K, V) в правое поддереву T .
 - Если $K < X$, рекурсивно добавить (K, V) в левое поддереву T .

Удаление узла (REMOVE)

Дано: дерево T с корнем n и ключом K .

Задача: удалить из дерева T узел с ключом K (если такой есть).

Алгоритм:

- Если дерево T пусто, остановиться;
- Иначе сравнить K с ключом X корневого узла n .
 - Если $K > X$, рекурсивно удалить K из правого поддерева T ;
 - Если $K < X$, рекурсивно удалить K из левого поддерева T ;
 - Если $K = X$, то необходимо рассмотреть три случая.
 - Если обоих детей нет, то удаляем текущий узел и обнуляем ссылку на него у родительского узла;
 - Если одного из детей нет, то значения полей ребёнка m ставим вместо соответствующих значений корневого узла, затирая его старые значения, и освобождаем память, занимаемую узлом m ;
 - Если оба ребёнка присутствуют, то
 - найдём узел m , являющийся самым левым узлом правого поддерева с корневым узлом $Right(n)$;
 - скопируем данные (кроме ссылок на дочерние элементы) из m в n ;
 - рекурсивно удалим узел m .

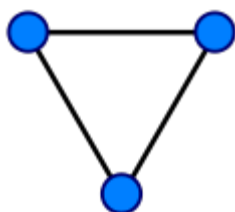
18. Определение графа и способы представления графов. Поиск в ширину. Поиск в глубину.

В математической [теории графов](#) и [информатике](#) **граф** — это совокупность непустого [множества](#) вершин и множества пар вершин (связей между вершинами). Объекты представляются как **вершины**, или **узлы** графа, а связи — как **дуги**, или **рёбра**. Для разных областей применения виды графов могут различаться направленностью, ограничениями на количество связей и дополнительными данными о вершинах или рёбрах.

Определения

Теория графов не обладает устоявшейся терминологией. В различных статьях под одними и теми же терминами понимаются разные вещи. Ниже приведены наиболее часто встречаемые определения.

Граф



Граф, или **неориентированный граф** G — это [упорядоченная пара](#) $G := (V, E)$, для которой выполнены следующие условия:

- V — это непустое [множество](#) **вершин**, или **узлов**,
- E — это множество пар (в случае неориентированного графа — неупорядоченных) вершин, называемых **рёбрами**.

V (а значит и, E , иначе оно было бы [мультимножеством](#)) обычно считаются конечными множествами. Многие хорошие результаты, полученные для конечных графов, неверны (или каким-либо образом отличаются) для *бесконечных графов*. Это происходит потому, что ряд соображений становится ложным в случае бесконечных множеств.

Вершины и рёбра графа называются также **элементами** графа, число вершин в

графе $|V|$ — **порядком**, число рёбер $|E|$ — **размером** графа.

Вершины u и v называются **концевыми** вершинами (или просто **концами**) ребра $e = \{u, v\}$

. Ребро, в свою очередь, **соединяет** эти вершины. Две концевые вершины одного и того же ребра называются **соседними**.

Два ребра называются **смежными**, если они имеют общую концевую вершину.

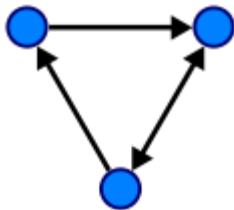
Два ребра называются **кратными**, если множества их концевых вершин совпадают.

Ребро называется **петлёй**, если его концы совпадают, то есть $e = \{v, v\}$.

Степенью $\deg v$ вершины v называют количество инцидентных ей рёбер (при этом петли считают дважды).

Вершина называется **изолированной**, если она не является концом ни для одного ребра; **висячей** (или **листом**), если она является концом ровно одного ребра.

Ориентированный граф



Основная статья: [Ориентированный граф](#)

Ориентированный граф (сокращённо **орграф**) G — это [упорядоченная пара](#) $G := (V, A)$

, для которой выполнены следующие условия:

- V — это непустое [множество](#) **вершин** или **узлов**,
- A — это множество (упорядоченных) пар различных вершин, называемых **дугами** или **ориентированными рёбрами**.

Дуга — это упорядоченная пара вершин (v, w) , где вершину v называют началом, а w — концом дуги. Можно сказать, что дуга $v \rightarrow w$ ведёт от вершины v к вершине w .

Смешанный граф

Смешанный граф G — это граф, в котором некоторые рёбра могут быть ориентированными, а некоторые — неориентированными. Записывается

$G := (V, E, A)$, где V , E и A определены так же, как выше. Ориентированный и неориентированный графы являются частными случаями смешанного.

Изоморфные графы

Граф G называется изоморфным графу H , если существует [биекция](#) f из множества вершин графа G в множество вершин графа H , обладающая следующим свойством: если в графе G есть ребро из вершины A в вершину B , то в графе H должно быть ребро из вершины $f(A)$ в вершину $f(B)$ и наоборот — если в графе H есть ребро из вершины A в вершину B , то в графе G должно быть ребро из вершины $f^{-1}(A)$ в вершину $f^{-1}(B)$. В случае [ориентированного графа](#) эта биекция также должна сохранять ориентацию ребра. В случае [взвешенного графа](#) биекция также должна сохранять вес ребра.

Прочие связанные определения

Путём (или **цепью**) в графе называют конечную последовательность вершин, в которой каждая вершина (кроме последней) соединена со следующей в последовательности вершин ребром.

Ориентированным путём в орграфе называют конечную последовательность

$$v_i (i = 1, \dots, k) \quad (v_i, v_{i+1}) (i = 1, \dots, k-1)$$

вершин, для которой все пары являются (ориентированными) рёбрами.

Циклом называют путь, в котором первая и последняя вершины совпадают. При этом **длиной** пути (или цикла) называют число составляющих его рёбер. Заметим, что если вершины u и v являются концами некоторого ребра, то согласно данному

$$(u, v, u)$$

определению, последовательность является циклом. Чтобы избежать таких «вырожденных» случаев, вводят следующие понятия.

Путь (или цикл) называют **простым**, если ребра в нём не повторяются;

элементарным, если он простой и вершины в нём не повторяются. Несложно видеть, что:

- Всякий путь, соединяющий две вершины, содержит элементарный путь, соединяющий те же две вершины.
- Всякий простой *неэлементарный* путь содержит элементарный *цикл*.
- Всякий *простой* цикл, проходящий через некоторую вершину (или ребро), содержит *элементарный* (под-)цикл, проходящий через ту же вершину (или ребро).
- Петля — элементарный цикл.

Бинарное отношение на множестве вершин графа, заданное как «существует путь

из u в v », является отношением эквивалентности и, следовательно, разбивает это множество на классы эквивалентности, называемые **компонентами связности** графа. Если у графа ровно одна компонента связности, то граф связный. На компоненте связности можно ввести понятие **расстояния** между вершинами как минимальную длину пути, соединяющего эти вершины.

Всякий максимальный связный подграф графа G называется **связной компонентой**

(или просто компонентой) графа G . Слово «максимальный» означает максимальный относительно включения, то есть не содержащийся в связном подграфе с большим числом элементов

Ребро графа называется **мостом**, если его удаление увеличивает число компонент.

Дополнительные характеристики графов

Граф называется:

- связным, если для любых вершин u, v , есть путь из u в v .
- **сильно связным** или **ориентированно связным**, если он ориентированный, и из любой вершины в любую другую имеется ориентированный путь.
- деревом, если он связный и не содержит *простых* циклов.
- полным, если любые его две (различные, если не допускаются петли) вершины соединены ребром.
- двудольным, если его вершины можно разбить на два непересекающихся подмножества V_1 и V_2 так, что всякое ребро соединяет вершину из V_1 с вершиной из V_2 .
- **k-дольным**, если его вершины можно разбить на k непересекающихся подмножества V_1, V_2, \dots, V_k так, что не будет рёбер, соединяющих вершины одного и того же подмножества.

- **полным двудольным**, если каждая вершина одного подмножества соединена ребром с каждой вершиной другого подмножества.
- **планарным**, если граф можно изобразить диаграммой на плоскости без пересечений рёбер.
- **взвешенным**, если каждому ребру графа поставлено в соответствие некоторое число, называемое весом ребра.

Также бывает:

- k-раскрашиваемым
- k-хроматическим

Обобщение понятия графа

Простой граф является одномерным симплициальным комплексом.

Более абстрактно, граф можно задать как тройку (V, E, φ) , где V и E — некоторые множества (вершин и рёбер, соотв.), а φ — **функция инцидентности** (или **инцидентор**), сопоставляющая каждому ребру $e \in E$ (упорядоченную или неупорядоченную) пару вершин u и v из V (его *концов*). Частными случаями этого понятия являются:

- ориентированные графы (орграфы) — когда $\varphi(e)$ всегда является упорядоченной парой вершин;
- неориентированные графы — когда $\varphi(e)$ всегда является неупорядоченной парой вершин;
- смешанные графы — в котором встречаются как ориентированные, так и неориентированные рёбра и петли;
- Эйлеровы графы — граф в котором существует циклический эйлеров путь (Эйлеров цикл).
- мультиграфы — графы с *кратными* рёбрами, имеющими своими концами одну и ту же пару вершин;
- псевдографы — это мультиграфы, допускающие наличие петель;
- простые графы — не имеющие петель и кратных рёбер.

Под данное выше определение не подходят некоторые другие обобщения:

- гиперграф — если ребро может соединять более двух вершин.
- ультраграф — если между элементами x_i и x_j существуют бинарные отношения инцидентности.

Представление графа

Итак, разобравшись, что такое граф, приступим к «обучению» компьютера работы с ним. От того, как хранить граф в той или иной задаче очень много зависит... После того, как придумано полное решение, но, написав его, получаешь только половину баллов, задумываешься, а правильно ли храниться граф?

Проблема правильного хранения графа в памяти компьютера действительно актуальна в сегодняшние дни. Давайте выясним, какие существуют методы хранения графа в памяти компьютера.

Матрица смежности

Один из самых распространённых способов хранения графа - матрица смежности. Она представляет собой двумерный массив. Если в клетке i, j (i - строка, j - столбец) установлено значение пусто (как правило, это очень

большая величина или величина, которой заведомо не может равняться вес ребра), то дуги, начинающейся в вершине i и кончающейся в вершине j , нет. Иначе дуга есть. Если она есть, то в соответствующую ячейку записывают ее вес. Если граф не взвешенный, то вес дуги считается равным единице. Составим матрицу смежности для нашего графа:

	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	2	0	0	0
3	0	0	0	3	0	0
4	0	4	0	0	0	0
5	3	7	0	0	0	0
6	0	0	0	0	0	0

Если нам дан неориентированный граф, то ребро можно заменить двумя дугами, т.е. если у нас есть ребро (1,3), то мы можем заменить его на дуги (1,3) и (3,1) - так мы сможем пройти в любом направлении в любое время.

Как вы уже заметили, в матрице смежности нам часто нужно хранить большое количество нулей. Например, в нашей матрице "нужных" значений только 6, а остальные 30 - нули, не представляющие для нас почти никакой нужной информации.

Для представления графа матрицей смежности нужно V^2 (где V - количество вершин) ячеек. Если граф почти полный, т.е. $E \approx V^2$ (где E - количество дуг), этот способ хранения графа наиболее выгоден, т.к. его очень просто реализовывать и память будет заполнена "нужными" значениями.

Но если это условие не выполняется, например, вершин до 10000 и ребер до 1000, то нам понадобится $\approx 2 * 100000000 / 1024^2$ Мбайт памяти (по 2 байта на ячейку), что примерно равно 190 Мбайт, памяти. А на олимпиадах, как правило, ограничения на ее использование составляет 64 Мбайт. Значит, этот метод не годится.

Кроме большого количества требуемой памяти и медленной работы на разреженных графах (графах, у которых $E \ll V^2$) у матрицы смежности есть ещё один важный недостаток. Иногда в задачах нужно выводить не номера вершин, а номера дуг (рёбер) на вводе. Хранить эти номера матрица смежности «не умеет». Нужно реализовывать восстановление номера дуги (ребра) как-то иначе, а это совсем неудобно.

Пусть G - помеченный граф порядка n , $V=\{1,2,\dots,n\}$.

Матрица смежностей неориентированного графа - это матрица $B=[B_{ij}]$ размерности $|V| \times |V|$, где

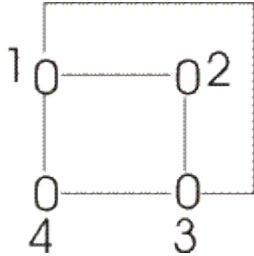
$$B_{i,j} = \begin{cases} 1, & \text{если существует ребро, идущее из вершины } i \text{ в вершину } j; \\ 0, & \text{в противном случае.} \end{cases}$$

Матрица смежностей ориентированного графа - это матрица $B=[B_{ij}]$ размерности $|V| \times |V|$, где

$$B_{i,j} = \begin{cases} 1, & \text{если существует дуга, идущая из вершины } i \text{ в вершину } j; \\ 0, & \text{в противном случае.} \end{cases}$$

Легко видеть, что матрица смежностей неориентированного графа *всегда симметрична*, поэтому для ее представления достаточно хранить только ее верхний треугольник. Приведем пример:

Неориентированный граф

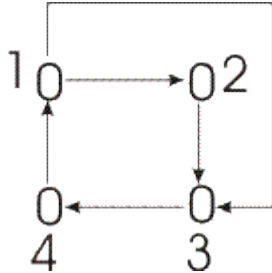


Матрица смежностей

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Конечно, для ориентированного графа это не так. Приведем пример:

Ориентированный граф



Матрица смежностей

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Количество единиц в строке матрицы смежностей равно степени вершины, соответствующей данной строке.

Представление графа в виде матрицы смежностей "**удобно**" для тех алгоритмов на графах, которым часто необходимо "знать", есть ли в графе данное ребро, ибо время, необходимое для поиска ребра, фиксировано и не зависит от $|V|$ и $|E|$.

Приведем расчеты временной сложности хранения графа матрицей смежности:

Операция

Временная сложность

Проверка смежности вершин x и y	$O(1)$
Перечисление всех вершин смежных с x	$O(V)$
Определение веса ребра (x, y)	$O(1)$
Перечисление всех ребер (x, y)	$O(V^2)$

Список дуг

Следующий тип хранения графа в памяти компьютера - список дуг. Чаще всего это двумерный массив размером $3 \times E$, в первой строке которого хранится информация, из какой вершины начинается дуга, во второй - в какой кончается, а в третьей строке - вес дуги. Опять же разберёмся на примере (все тот же граф):

	1	2	3	4	5	6
1	1	2	3	4	5	5
2	2	3	4	2	1	2
3	1	2	3	4	3	7

Мы чётко видим, что почти вся таблица заполнена "нужными" значениями, а не нулями. Это уже хорошо, значит, память экономится.

Приведем расчеты временной сложности хранения графа списком дуг:

Операция

Временная сложность

Проверка смежности вершин x и y	$O(E)$
Перечисление всех вершин смежных с x	$O(E)$
Определение веса ребра (x, y)	$O(E)$

Перечисление всех ребер (x, y)
Поиск i-ой дуги

$O(E)$
 $O(1)$

Как видно, этот способ, в отличие от матрицы смежности, хранит информацию о номере дуги. Также ясно, что этот способ нам выгоден, если чаще всего нам нужно будет узнать что-то (вес, вершины начала или конца) о i-ой дуге. Однако, такие задачи в практическом программировании встречаются довольно редко.

Если в предыдущих представлениях одно ребро мы заменяли двумя дугами, то список дуг может хранить и дуги и рёбра (в зависимости от реализации). Это довольно удобно и может требовать в 2 раза меньше памяти.

Матрица инцидентности

Классическим способом представления графа (V, E) служит *матрица инцидентности*, представляющая собой матрицу с $|V|$ строками, соответствующих вершинам, и $|E|$ столбцами, соответствующих ребрам (или дугам) графа, причем :

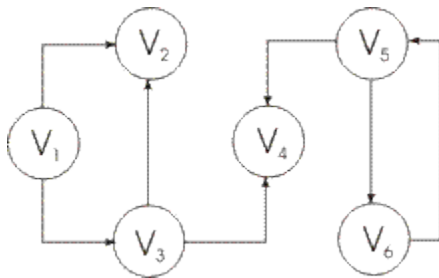
- 1) в случае *неориентированного графа* столбец, соответствующий ребру $\{x, y\}$, содержит 1 в строках, соответствующих вершинам x и y , и 0 - в остальных строках;
- 2) для *ориентированного графа*, столбец, соответствующий дуге $(x, y) \in E$, содержит:
 - (a) (-1) в строке, соответствующей вершине x ;
 - (b) 1 в строке, соответствующей вершине y ,
 - (c) 0 во всех остальных строках (петлю в вершине x удобно представлять иным значением в строке, соответствующей вершине x , например, 2).

Отметим, что:

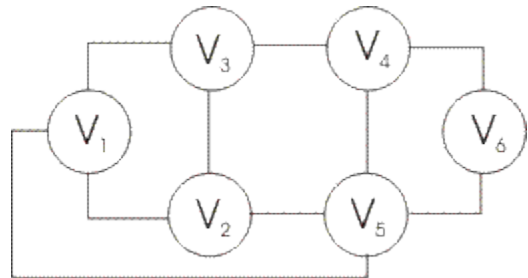
- (a) никакие два столбца матрицы инцидентности не идентичны,
- (b) каждый столбец в матрице содержит две единицы,
- (c) количество единиц в i-ой строке равно степени вершины i в графе.

Рассмотрим два графа [Бурковский, Холопкина, Райхель, Кравец, 1996]:

(а) ориентированный граф;



(б) неориентированный граф.



Матрицы инцидентности для данных графов имеют следующий вид:

Номера дуг

	1	2	3	4	5	6	7
1	-1	-1	0	0	0	0	0
2	1	0	1	0	0	0	0
3	0	1	-1	-1	0	0	0
4	0	0	0	1	1	0	0
5	0	0	0	0	-1	-1	1
6	0	0	0	0	0	1	-1

Номера ребер

	1	2	3	4	5	6	7	8	9
1	1	1	1	0	0	0	0	0	0
2	1	0	0	1	1	0	0	0	0
3	0	1	0	1	0	1	0	0	0
4	0	0	0	0	0	1	1	1	0
5	0	0	1	0	1	0	1	0	1
6	0	0	0	0	0	0	0	1	1

С алгоритмической точки зрения матрица инцидентности является, вероятно, **самым худшим способом представления графа**. Этот способ требует $|E| \cdot |V|$ ячеек памяти, причем большинство этих ячеек занято нулями. Неудобен также доступ к информации. Ответ на элементарные вопросы типа: "Существует ли дуга

(x,y)?", "К каким вершинам ведут ребра из вершины x?" требует в худшем случае перебора всех столбцов матрицы, т.е. $|E|$ шагов. Однако матрица инцидентности **полезна** при решении задач о циклах (или контурах) в теории графов.

Поиск в глубину

Поиском в глубину называется способ обхода вершин графа, который, начавшись от какой-либо вершины, рекурсивно применяется ко всем вершинам, в которые можно попасть из текущей. Пусть граф с N вершинами задан матрицей смежности $A[N][N]$. Создадим одномерный массив `int visited[N]` и заполним его нулями, считая, что ни одна вершина до начала выполнения алгоритма не была посещена. Функция обхода записывается следующим образом:

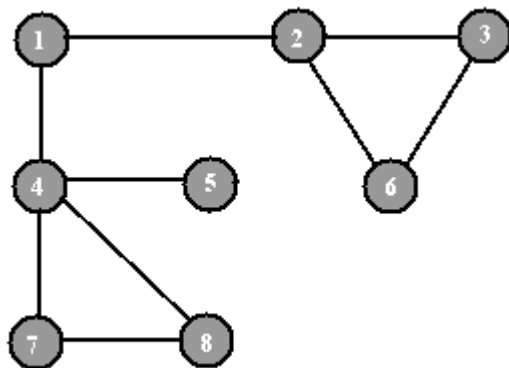
```
void go(int curr)
{
    visited[curr] = 1; /* помечаем текущую вершину как пройденную */
    for (int i = 0; i < N; i++)
        if (!visited[i] && A[curr][i])
            go(i);
}
```

...

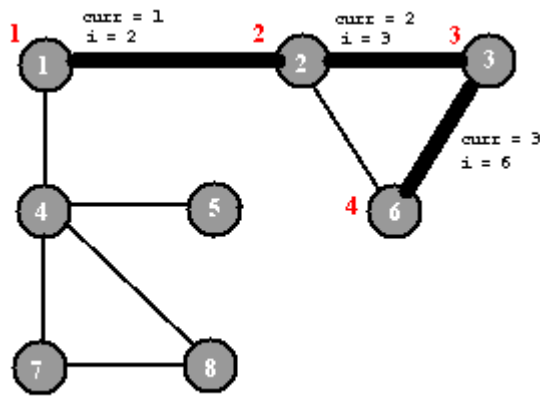
/* в тексте программы */

go(start);

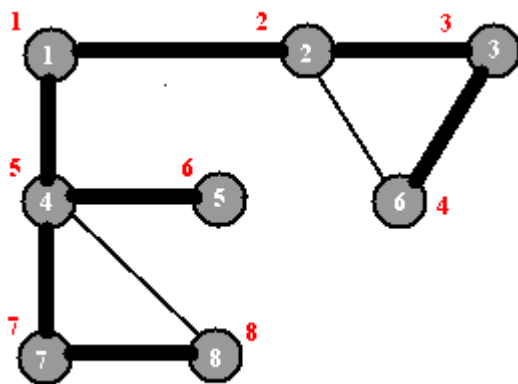
Распишем подробно, как будет работать алгоритм на графе, изображенном на рисунке. Пусть `start = 1`, т.е. начинаем с первой вершины.



При вызове `go(1)` первая вершина помечается как пройденная (`visited[1] = 1`), и в цикле по i перебираются все вершины, куда можно попасть из текущей (т.е. из первой). При $i = 2$ мы видим, что во второй вершине мы не были (`visited[2] == 0`) и что в графе есть ребро, соединяющее первую вершину и вторую, следовательно, рекурсивно вызываем ту же функцию `go(int curr)` с параметром `curr`, равным 2. Аналогично, `go(2)` вызывает `go(3)`, а последняя вызывает `go(6)`. Здесь мы обнаруживаем, что из шестой вершины некуда идти: и во второй, и в третьей мы уже побывали. Следовательно, `go(6)` дорабатывает до конца, а за ней заканчиваются `go(3)` и `go(2)`. Таким образом, мы оказываемся в функции `go(1)` в конце цикла `for` при $i = 2$. На рисунке изображена эта ситуация, красным цветом отмечен порядок обхода вершин, жирным выделены ребра, по которым мы «ходили».



Цикл for в go(1) продолжает работу, и при $i = 4$ мы уходим в рекурсию go(4), и т.д. пока управление снова не вернется в go(1) (и произойдет выход из функции, т.к. больше нет непосещенных вершин, связанных ребром с первой). Окончательная картинка выглядит так:



В результате работы алгоритма все вершины графа оказались пройденными (при условии, что граф связный). Необходимо отметить, что ребра, по которым мы «ходили» — так называемые *прямые ребра* (они выделены жирным) — образуют один из возможных *каркасов* исходного графа.

Поиск в глубину ([англ. Depth-first search, DFS](#)) — один из методов обхода [графа](#). Алгоритм поиска описывается следующим образом: для каждой непройденной вершины необходимо найти все не пройденные смежные вершины и повторить поиск для них. Используется в качестве подпрограммы в алгоритмах поиска одно- и двусвязных компонент, топологической сортировки.

Алгоритм поиска в глубину

Пусть задан граф $G = (V, E)$, где V — множество вершин графа, E — множество ребер графа. Предположим, что в начальный момент времени все вершины графа окрашены в *белый* цвет. Выполним следующие действия:

1. Из множества всех *белых* вершин выберем любую вершину, обозначим её v_1 .
2. Выполняем для неё процедуру DFS(v_1).
3. Повторяем шаги 1-3 до тех пор, пока множество *белых* вершин не пусто.

Процедура DFS (параметр — вершина $u \in V$)

1. Перекрашиваем вершину u в *черный* цвет.
 2. Для всякой вершины w , смежной с вершиной u и окрашенной в белый цвет, выполняем процедуру DFS(w).
-

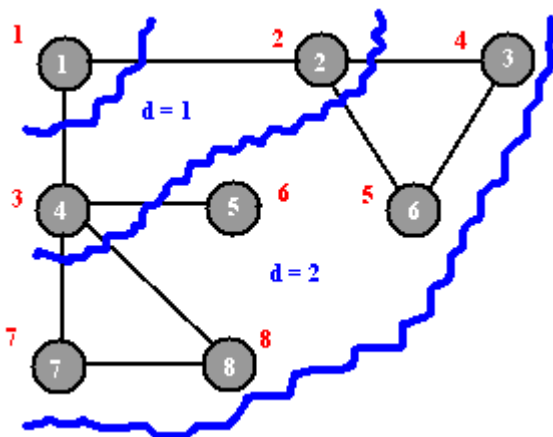
Поиск в ширину

При поиске в ширину вместо стека рекурсивных вызовов хранится очередь, в которую записываются вершины в порядке удаления от начальной. Опять же, введем массив `int visited[N]`, а также создадим очередь для хранения вершин (реализуем ее в виде простого массива; естественно, возможны другие варианты). В начало очереди запишем начальную вершину:

```
int queue[N];
queue[0] = start;
visited[start] = 1;
int r = 0, w = 1;
```

Переменная r будет указывать позицию очереди, из которой мы читаем данные, переменная w — позицию, куда данные будем писать. Тогда обход может быть написан следующим образом:

```
while (r < w) {
    int curr = queue[r++];
    for (int i = 0; i < N; i++) {
        if (!visited[i] && A[curr][i]) {
            visited[i] = 1;
            queue[w++] = i;
        }
    }
}
```



Находясь в первой вершине, в очередь добавляются вторая и четвертая (они удалены от первой на расстояние $d = 1$). Затем из очереди читается очередной элемент (только что добавленная вторая вершина) и добавляются вершины, в которые можно попасть из второй: третья и шестая. То же самое делается для следующей в очереди вершины (четвертой): в очередь добавляются вершины 5, 7 и 8. Так как все вершины графа уже оказались пройденными, больше в очередь ничего добавлено не будет, и алгоритм завершится после чтения из очереди последней вершины (8).

Красным цветом на рисунке помечен порядок обхода вершин при поиске в ширину. Так как вершины перебираются в порядке удаления от начальной, работу алгоритма

можно представить в виде набегающей волны, что и дает ему второе название: «метод волны». Ясно, что таким образом можно искать кратчайшие пути от одной вершины до всех остальных в невзвешенном графе (сравните с [алгоритмом Дейкстры](#) для взвешенных графов).

Поиск в ширину

Поиск в ширину (обход в ширину, breadth-first search) — это один из основных алгоритмов на графах.

В результате поиска в ширину находится путь кратчайшей длины в невзвешенном графе, т.е. путь, содержащий наименьшее число рёбер.

Алгоритм работает за $O(n + m)$, где n — число вершин, m — число рёбер.

Описание алгоритма

На вход алгоритма подаётся заданный граф (невзвешенный), и номер стартовой вершины s . Граф может быть как ориентированным, так и неориентированным, для алгоритма это не важно.

Сам алгоритм можно понимать как процесс "поджигания" графа: на нулевом шаге поджигаем только вершину s . На каждом следующем шаге огонь с каждой уже горящей вершины перекидывается на всех её соседей; т.е. за одну итерацию алгоритма происходит расширение "кольца огня" в ширину на единицу (отсюда и название алгоритма).

Более строго это можно представить следующим образом. Создадим очередь q , в которую будут помещаться горящие вершины, а также заведём булевский массив $used[]$, в котором для каждой вершины будем отмечать, горит она уже или нет (или иными словами, была ли она посещена).

Изначально в очередь помещается только вершина s , и $used[s] = true$, а для всех остальных вершин $used[] = false$. Затем алгоритм представляет собой цикл: пока очередь не пуста, достать из её головы одну вершину, просмотреть все рёбра, исходящие из этой вершины, и если какие-то из просмотренных вершин ещё не горят, то поджечь их и поместить в конец очереди.

В итоге, когда очередь опустеет, обход в ширину обойдёт все достижимые из вершины, причём до каждой дойдёт кратчайшим путём. Также можно посчитать длины кратчайших путей (для чего просто надо завести массив длин путей $d[]$), и компактно сохранить информацию, достаточную для восстановления всех этих

кратчайших путей (для этого надо завести массив "предков" $p[]$, в котором для каждой вершины хранить номер вершины, по которой мы попали в эту вершину).

19. Алгоритмы Крускала, Примы, Дейкстры.

Алгоритм Дейкстры

Алгоритм Дейкстры решает задачу о кратчайшем пути из одной вершины во взвешенном ориентированном графе $G = (V, E)$ в том случае, когда веса ребер неотрицательны. Поэтому в настоящем разделе предполагается, что для всех ребер $(u, v) \in E$ выполняется неравенство $w(u, v) \geq 0$. При хорошей реализации алгоритм Дейкстры производительнее, чем

алгоритм Беллмана-Форда.

В алгоритме Дейкстры поддерживается множество вершин S , для которых уже вычислены окончательные веса кратчайших путей к ним из истока s . В этом алгоритме поочередно выбирается вершина $u \in V - S$, которой на данном этапе соответствует минимальная оценка кратчайшего пути. После добавления этой вершины u в множество S производится ослабление всех исходящих из нее ребер. В приведенной ниже реализации используется неубывающая очередь с приоритетами Q , состоящая из вершин, в роли ключей для которых выступают значения d .

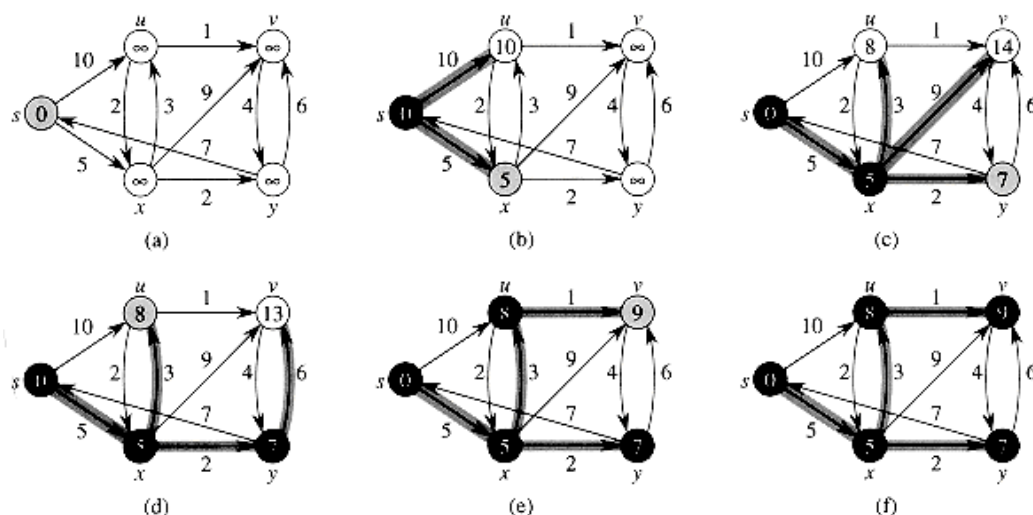
Dijkstra(G, w, s)

1. Initialize _Single_Source(G, s)
2. $S \leftarrow \emptyset$
3. $Q \leftarrow V[G]$
4. while $Q \neq \emptyset$
5. do $u \leftarrow \text{Extract Min}(Q)$
6. $S \leftarrow S \cup \{u\}$
7. for (для) каждой вершины $v \in \text{Adj}[u]$
8. do Relax(u, v, w)
- 9.

Процесс ослабления ребер в алгоритме Дейкстры проиллюстрирован на

Рисунке. Исток s расположен на рисунке слева от остальных вершин. В каждой вершине приведена оценка кратчайшего пути к ней, а выделенные ребра указывают предшественников. Черным цветом обозначены вершины, добавленные в множество S , а белым — содержащиеся в неубывающей очереди с приоритетами $Q = V - S$. В части а рисунка проиллюстрирована ситуация, сложившаяся непосредственно перед выполнением первой итерации цикла while в строках 4-8. Выделенная серым цветом вершина имеет минимальное значение d и выбирается в строке 5 в качестве вершины u для следующей итерации. В частях б-е изображены

ситуации после выполнения очередной итерации цикла while. В каждой из этих частей выделенная серым цветом вершина выбирается в качестве вершины u в строке 5. В части e приведены конечные значения величин d и π .



Выполнение алгоритма Дейкстры

Время выполнения алгоритма Дейкстры зависит от реализации неубывающей очереди с приоритетами. Сначала рассмотрим случай, когда неубывающая очередь с приоритетами поддерживается за счет того, что все вершины пронумерованы от 1 до $|V|$. Атрибут $d[v]$ просто помещается в элемент массива с индексом v . Каждая операция Insert и Decrease_Key занимает время $O(1)$, а каждая операция Extract_Min — время $O(V)$ (поскольку в ней производится поиск по всему массиву); в результате полное время работы алгоритма равно $O(V^2 + E) = O(V^2)$.

Фактически время работы алгоритма может достигать значения $O(V \lg V + E)$, если неубывающая очередь с приоритетами реализуется с помощью пирамиды Фибоначчи.

Алгоритм Крускала

Дан взвешенный неориентированный граф. Требуется найти такое поддерево этого графа, которое бы соединяло все его вершины, и при этом обладало наименьшим весом (т.е. суммой весов рёбер) из всех возможных. Такое поддерево называется минимальным остовным деревом или просто минимальным остовом.

Здесь будут рассмотрены несколько важных фактов, связанных с минимальными остовами, затем будет рассмотрен алгоритм Крускала в его простейшей реализации.

Свойства минимального остова

- Минимальный остов **уникален**, если веса всех рёбер различны. В противном случае, может существовать несколько минимальных остовов (конкретные алгоритмы обычно получают один из возможных остовов).
- Минимальный остов является также и **остовом с минимальным произведением** весов рёбер.

(доказывается это легко, достаточно заменить веса всех рёбер на их логарифмы)

- Минимальный остов является также и **остовом с минимальным весом самого тяжелого ребра**.

(это утверждение следует из справедливости алгоритма Крускала)

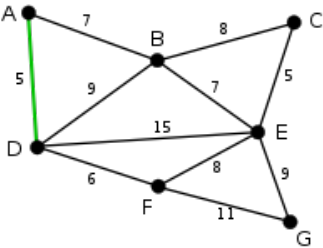
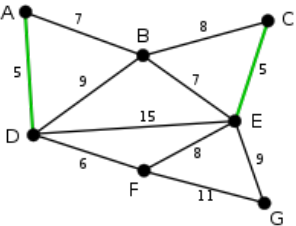
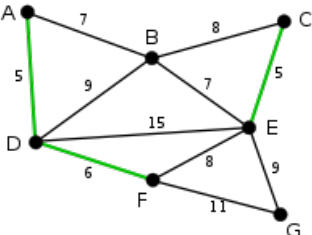
- **Остов максимального веса** ищется аналогично остову минимального веса, достаточно поменять знаки всех рёбер на противоположные и выполнить любой из алгоритм минимального остова.

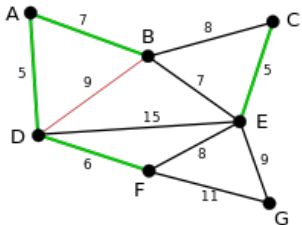
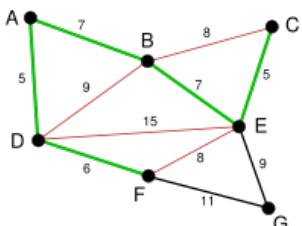
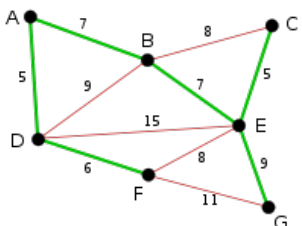
Алгоритм Крускала

Данный алгоритм был описан Крускалом (Kruskal) в 1956 г.

Алгоритм Крускала изначально помещает каждую вершину в своё дерево, а затем постепенно объединяет эти деревья, объединяя на каждой итерации два некоторых дерева некоторым ребром. Перед началом выполнения алгоритма, все рёбра сортируются по весу (в порядке неубывания). Затем начинается процесс объединения: перебираются все рёбра от первого до последнего (в порядке сортировки), и если у текущего ребра его концы принадлежат разным поддеревьям, то эти поддеревья объединяются, а ребро добавляется к ответу. По окончании перебора всех рёбер все вершины окажутся принадлежащими одному поддереву, и ответ найден. Подграф данного графа, содержащий все его вершины и найденное множество рёбер, является его остовным деревом минимального веса.

Пример

Изображение	Описание
	Ребра AD и CE имеют минимальный вес, равный 5. Произвольно выбирается ребро AD (выделено на рисунке).
	Теперь наименьший вес, равный 5, имеет ребро CE . Так добавление CE не образует цикла, то выбираем его в качестве второго ребра.
	Аналогично выбираем ребро DF , вес которого равен 6.

	<p>Следующие ребра — AB и BE с весом 7. Произвольно выбирается ребро AB, выделенное на рисунке. Ребро BD выделено красным, так уже существует путь (зеленый) между B и D, поэтому, если бы это ребро было выбрано, то образовался бы цикл ABD.</p>
	<p>Аналогичным образом выбирается ребро BE, вес которого равен 7. На этом этапе красным выделено гораздо больше ребер: BC, потому что оно создаст цикл BCE, DE, потому что оно создаст цикл DEBA, и FE, потому что оно сформирует цикл FEBAD.</p>
	<p>Алгоритм завершается добавлением ребра EG с весом 9. Минимальное остовное дерево построено.</p>

Оценка

До начала работы алгоритма необходимо отсортировать рёбра по весу, это требует $O(E \times \log(E))$ времени. После чего компоненты связности удобно хранить в виде [системы непересекающихся множеств](#). Все операции в таком случае займут $O(E \times \alpha(E, V))$, где α — функция, обратная к [функции Аккермана](#). Поскольку для любых практических задач $\alpha(E, V) < 5$, то можно принять её за константу, таким образом общее время работы алгоритма Крускала можно принять за $O(E * \log(E))$.

Алгоритм Прима

Дан взвешенный неориентированный граф G с n вершинами и m рёбрами. Требуется найти такое поддерево этого графа, которое бы соединяло все его вершины, и при этом обладало наименьшим возможным весом (т.е. суммой весов рёбер). Поддерево — это набор рёбер, соединяющих все вершины, причём из любой вершины можно добраться до любой другой ровно одним простым путём.

Такое поддерево называется минимальным остовным деревом или просто **минимальным остовом**. Легко понять, что любой остов обязательно будет содержать $n - 1$ ребро.

В **естественной постановке** эта задача звучит следующим образом: есть n городов, и для каждой пары известна стоимость соединения их дорогой (либо известно, что

соединить их нельзя). Требуется соединить все города так, чтобы можно было доехать из любого города в другой, а при этом стоимость прокладки дорог была бы минимальной.

Алгоритм Прима

Этот алгоритм назван в честь американского математика Роберта Прима (Robert Prim), который открыл этот алгоритм в 1957 г. Впрочем, ещё в 1930 г. этот алгоритм был открыт чешским математиком Войтеком Ярником (Vojtěch Jarník). Кроме того, Эдгар Дейкстра (Edsger Dijkstra) в 1959 г. также изобрёл этот алгоритм, независимо от них.

Описание алгоритма

Сам **алгоритм** имеет очень простой вид. Искомый минимальный остов строится постепенно, добавлением в него рёбер по одному. Изначально остов полагается состоящим из единственной вершины (её можно выбрать произвольно). Затем выбирается ребро минимального веса, исходящее из этой вершины, и добавляется в минимальный остов. После этого остов содержит уже две вершины, и теперь ищется и добавляется ребро минимального веса, имеющее один конец в одной из двух выбранных вершин, а другой — наоборот, во всех остальных, кроме этих двух. И так далее, т.е. всякий раз ищется минимальное по весу ребро, один конец которого — уже взятая в остов вершина, а другой конец — ещё не взятая, и это ребро добавляется в остов (если таких рёбер несколько, можно взять любое). Этот процесс повторяется до тех пор, пока остов не станет содержать все вершины (или,

что то же самое, $n - 1$ ребро).

В итоге будет построен остов, являющийся минимальным. Если граф был изначально не связан, то остов найден не будет (количество выбранных рёбер

останется меньше $n - 1$).

Доказательство

Пусть граф G был связным, т.е. ответ существует. Обозначим через T остов, найденный алгоритмом Прима, а через S — минимальный остов. Очевидно, что T действительно является остовом (т.е. поддеревом графа G). Покажем, что веса S и T совпадают.

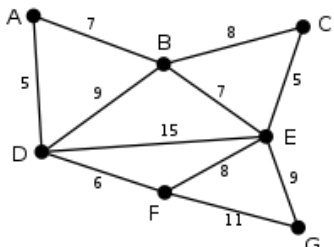
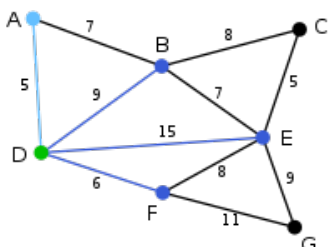
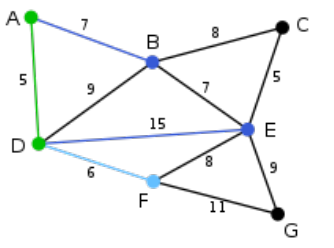
Рассмотрим первый момент времени, когда в T происходило добавление ребра, не входящего в оптимальный остов S . Обозначим это ребро через e , концы его — через a и b , а множество входящих на тот момент в остов вершин — через V (согласно алгоритму, $a \in V$ $b \notin V$, либо наоборот). В оптимальном остове S вершины a и b соединяются каким-то путём P ; найдём в этом пути любое ребро g , один конец которого лежит в V , а другой — нет. Поскольку алгоритм Прима выбрал ребро e вместо ребра g , то это значит, что вес ребра g больше либо равен весу ребра e .

Удалим теперь из S ребро g , и добавим ребро e . По только что сказанному, вес остова в результате не мог увеличиться (уменьшиться он тоже не мог, поскольку S было оптимальным). Кроме того, S не перестало быть остовом (в том, что связность не нарушилась, нетрудно убедиться: мы замкнули путь P в цикл, и потом удалили из этого цикла одно ребро).

Итак, мы показали, что можно выбрать оптимальный остов S таким образом, что он будет включать ребро e . Повторяя эту процедуру необходимое число раз, мы получаем, что можно выбрать оптимальный остов S так, чтобы он совпадал с T .

Следовательно, вес построенного алгоритмом Прима T минимален, что и требовалось доказать.

Пример

Изображение	Множество выбранных вершин U	Ребро (u,v)	Множество невыбранных вершин $V \setminus U$	Описание
	$\{\}$		$\{A,B,C,D,E,F,G\}$	Исходный взвешенный граф. Числа возле ребер показывают их веса, которые можно рассматривать как расстояния между вершинами.
	$\{D\}$	$(D,A) = 5$ $(D,B) = 9$ $(D,E) = 15$ $(D,F) = 6$	$\{A,B,C,E,F,G\}$	В качестве начальной произвольно выбирается вершина D . Каждая из вершин A , B , E and F соединена с D единственным ребром. Вершина A — ближайшая к D , и выбирается как вторая вершина вместе с ребром AD .
	$\{A,D\}$	$(D,B) = 9$ $(D,E) = 15$ $(D,F) = 6$ $(A,B) = 7$	$\{B,C,E,F,G\}$	Следующая вершина — ближайшая к любой из выбранных вершин D или A . B удалена от D на 9 и от A — на 7. Расстояние до E равно 15, а до F —

				6. F является ближайшей вершиной, поэтому она включается в дерево F вместе с ребром DF .
	{A,D,F}	$(D,B) = 9$ $(D,E) = 15$ $(A,B) = 7$ V $(F,E) = 8$ $(F,G) = 11$	{B,C,E,G}	Аналогичным образом выбирается вершина B , удаленная от A на 7.
	{A,B,D,F}	$(B,C) = 8$ $(B,E) = 7$ V $(D,B) = 9$ цикл $(D,E) = 15$ $(F,E) = 8$ $(F,G) = 11$	{C,E,G}	В этом случае есть возможность выбрать либо C , либо E , либо G . C удалена от B на 8, E удалена от B на 7, а G удалена от F на 11. E — ближайшая вершина, поэтому выбирается E и ребро BE .
	{A,B,D,E,F}	$(B,C) = 8$ $(D,B) = 9$ цикл $(D,E) = 15$ цикл $(E,C) = 5$ V $(E,G) = 9$ $(F,E) = 8$ цикл $(F,G) = 11$	{C,G}	Здесь доступны только вершины C и G . Расстояние от E до C равно 5, а до G — 9. Выбирается вершина C и ребро EC .
	{A,B,C,D,E,F}	$(B,C) = 8$ цикл $(D,B) = 9$ цикл $(D,E) = 15$ цикл $(E,G) = 9$ V $(F,E) = 8$ цикл $(F,G) = 11$	{G}	Единственная оставшаяся вершина — G . Расстояние от F до нее равно 11, от E — 9. E ближе, поэтому выбирается вершина G и ребро EG .
	{A,B,C,D,E,F,G}	$(B,C) = 8$ цикл $(D,B) = 9$ цикл $(D,E) = 15$ цикл $(F,E) = 8$ цикл $(F,G) = 11$ цикл	{}	Выбраны все вершины, <u>минимальное остовное дерево</u> построено (выделено зеленым). В этом случае его вес равен 39.

Оценка

Асимптотика алгоритма зависит от способа хранения графа и способа хранения

вершин, не входящих в дерево. Если приоритетная очередь Q реализована как обычный массив d , то $Extract.Min(Q)$ выполняется за $O(n)$, а стоимость операции $d[u] \leftarrow w(v, u)$ составляет $O(1)$. Если Q представляет собой бинарную пирамиду, то $Extract.Min(Q)$ выполняется за $O(\log n)$, а стоимость $d[u] \leftarrow w(v, u)$ снижается до $O(\log n)$, а стоимость $d[u] \leftarrow w(v, u)$ возрастает до $O(1)$. При использовании фибоначчиевых пирамид операция $Extract.Min(Q)$ выполняется за $O(\log n)$, а $d[u] \leftarrow w(v, u)$ за $O(1)$.

20. Реляционная модель данных. Операции реляционной алгебры над отношениями. Теорема Хита. Теория нормальных форм: 1НФ, 2НФ, 3НФ, нормальная форма Бойса-Кодда (НФБК), 4НФ, 5НФ. Теорема Фейгина.

Реляционная модель данных (РМД) — [логическая модель данных](#), прикладная [теория](#) построения [баз данных](#), которая является приложением к задачам обработки данных таких разделов [математики](#) как [теории множеств](#) и [логика первого порядка](#).

На реляционной модели данных строятся [реляционные базы данных](#).

Реляционная модель данных включает следующие компоненты:

- [Структурный](#) аспект (составляющая) — данные в базе данных представляют собой набор [отношений](#).
- Аспект (составляющая) [целостности](#) — отношения (таблицы) отвечают определенным условиям [целостности](#). РМД поддерживает декларативные [ограничения целостности](#) уровня [домена](#) (типа данных), уровня отношения и уровня базы данных.
- Аспект (составляющая) обработки (манипулирования) — РМД поддерживает операторы манипулирования отношениями ([реляционная алгебра](#), [реляционное исчисление](#)).

В реляционной модели каждый вид объектов описывается при помощи таблицы. Таблицы в реляционной модели принято называть отношениями (отсюда и название: отношение по-английски – relation).

Атрибуты отношения – это столбцы таблицы.

Каждая строка таблицы соответствует одному экземпляру объектов данного вида. Строки принято называть кортежами или записями.

Для лучшего понимания РМД следует отметить три важных обстоятельства:

- модель является логической, то есть отношения являются логическими (абстрактными), а не физическими (хранимыми) структурами;
- для реляционных баз данных верен [информационный принцип](#): всё информационное наполнение базы данных представлено одним и только одним способом, а именно — явным заданием значений атрибутов в [кортежах](#) отношений; в частности, нет никаких указателей (адресов), связывающих одно значение с другим;
- наличие реляционной алгебры позволяет реализовать [декларативное программирование](#) и декларативное описание ограничений целостности, в дополнение к навигационному (процедурному) программированию и процедурной проверке условий.

Наиболее известными альтернативами реляционной модели являются [иерархическая модель](#), и [сетевая модель](#). Некоторые системы, использующие эти старые архитектуры, используются до сих пор. Кроме того, можно упомянуть об [объектно-ориентированной модели](#), на которой строятся так называемые [объектно-ориентированные СУБД](#), хотя однозначного и общепринятого определения такой модели нет.

Кардинальность отношения – это количество кортежей (записей) удовлетворяющих данному отношению.

Степень отношения – это количество атрибутов отношения.

Домен, которому принадлежит атрибут – совокупность допустимых значений, которые может принимать данный атрибут.

Операции реляционной алгебры над отношениями.

Работа с отношениями (таблицами) в идеале должна осуществляться только посредством операций реляционной алгебры.

Выборка

Пусть задано отношение $A(X, Y, \dots)$ и операция $Q: (X, Y) \rightarrow \{T|F\}$. Тогда Q -выборкой из отношения A по атрибутам X, Y называется новое отношение B , с теми же атрибутами и содержащее все такие кортежи отношения A , для атрибутов X, Y которых $Q(X, Y) = T$.

Проекция

Пусть задано отношение $A(X, Y, \dots)$. Тогда проекцией отношения A по атрибутам X, Y называется новое отношение B , состоящее только из атрибутов X и Y и содержащее все кортежи вида $\{X:x, Y:y\}$, такие, что множество кортежей отношения A содержит хотя бы один кортеж вида $\{X:x, Y:y, \dots\}$.

Произведение

Пусть заданы отношение $A(A_1, A_2, \dots A_n)$ и отношение $B(B_1, B_2, \dots B_m)$. Тогда произведением отношения A на отношение B называется новое отношение C ,

состоящее только из множества атрибутов $A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m$ и содержащее все кортежи, полученные путем дописывания к каждому кортежу отношения A каждый кортеж отношения B .

Объединение

Пусть заданы отношение $A(X_1, X_2, \dots, X_n)$ и отношение $B(X_1, X_2, \dots, X_n)$. Тогда объединением отношения A и B называется новое отношение C , состоящее из тех же атрибутов X_1, X_2, \dots, X_n , и содержащее все такие кортежи k , для которых верно, что k принадлежит множеству кортежей отношения A или множеству кортежей отношения B .

Пересечение

Пусть заданы отношение $A(X_1, X_2, \dots, X_n)$ и отношение $B(X_1, X_2, \dots, X_n)$. Тогда пересечением отношения A и B называется новое отношение C , состоящее из тех же атрибутов X_1, X_2, \dots, X_n , и содержащее все такие кортежи k , для которых верно, что k принадлежит множеству кортежей отношения A и множеству кортежей отношения B одновременно.

Разность

Пусть заданы отношение $A(X_1, X_2, \dots, X_n)$ и отношение $B(X_1, X_2, \dots, X_n)$. Тогда разностью отношения A и B называется новое отношение C , состоящее из тех же атрибутов X_1, X_2, \dots, X_n , и содержащее все такие кортежи k , для которых верно, что k принадлежит множеству кортежей отношения A и не принадлежит множеству кортежей отношения B .

Естественное соединение

Пусть заданы отношение $A(A_1, A_2, \dots, A_n, X_1, \dots, X_k)$ и отношение $B(X_1, \dots, X_k, B_1, B_2, \dots, B_m)$. Тогда естественным соединением отношения A и отношения B называется новое отношение C , состоящее только множества атрибутов $A_1, A_2, \dots, A_n, X_1, \dots, X_k, B_1, B_2, \dots, B_m$ и содержащее все кортежи, полученные путем дописывания к каждому кортежу отношения A вида $(A_1:a_1, A_2:a_2, \dots, A_n:a_n, X_1:x_1, \dots, X_k:x_k)$ недостающей части вида $(B_1:b_1, B_2:b_2, \dots, B_m:b_m)$ каждого кортежа отношения B , у которого атрибуты X_1, \dots, X_k принимают те же самые значения $x_1 \dots x_k$.

Θ-соединение

Соединением отношений $A(A_1, A_2 \dots A_n)$ и $B(B_1, B_2 \dots B_m)$ по операции $\Theta: A_1 \times A_2 \times \dots \times A_n \times B_1 \times B_2 \times \dots \times B_m \rightarrow \{T|F\}$ называется новое отношение, содержащее все кортежи, получаемые путем соединения кортежей из исходных A и B и удовлетворяющие условию $\Theta(A:B) = T$.

Пример:

Θ : Заказы.id_клиента == Клиенты.id && Заказы.Статус != Доставлено

Эквисоединение

Если операция Θ представляет из себя равенство каких-либо полей (наборов полей), то такое соединение называется эквисоединением.

Пример:

Θ : Заказы.Id_клиента = Клиента.Id

Деление

Результатом деления отношения $A(A_1, A_2 \dots A_n)$ на отношение $B(B_1, B_2 \dots B_m)$ по отношению $C(A_1, A_2 \dots A_n, B_1, B_2 \dots B_m)$ называется новое отношение,

содержащее все такие кортежи x из отношения A для которых выполняется условие:

для любого y , принадлежащего множеству кортежей B , отношение C содержит кортеж $x:y$.

A называется делимым, B – делителем, а C – посредником.

Пример:

A		B		C				A divide by B per C	
F1	F2	F3	F4	F1	F2	F3	F4	F1	F2
1	2	1	1	1	2	1	1	1	2
2	3	1	1	1	2	2	2	1	2
		2	2	2	3	1	1		
				1	2	4	5		

Реляционная БД

Реляционная база данных – это набор реляционных отношений (таблиц) и только их. Никаким другим образом (переменные, массивы) данные не представлены.

Идеальная реляционная СУБД – это система управления реляционной БД, в которой все действия представляются в виде комбинации операций реляционной алгебры.

Теория нормализации

Нормализация данных - это разложение отношений на большее количество более простых таблиц.

Цели нормализации:

1. Основной целью теории нормальных форм первоначально была экономия места на диске.
2. Данные должны быть устроены так, чтобы при их редактировании или удалении, необходимо было исправлять только в одном месте БД.
3. Группировка данных по содержанию (Каждая таблица - определенная тематика).
4. Принцип модульности (Несколько унифицированных независимых блоков).

Декомпозиция.

Основой нормализации является процесс разбиения - или декомпозиции.

Причем нас будет интересовать не просто процесс декомпозиции, а процесс декомпозиции без потерь.

Процесс декомпозиции будем называть декомпозицией без потерь, если из полученных отношений можно полностью восстановить исходное отношение.

По своей сути декомпозиция представляет собой проекцию. А обратное преобразование - операция соединения.

Теорема Хита

Пусть дано отношение $R(A, B, C)$ где A, B и C – непересекающиеся подмножества множества атрибутов R . Причем множество атрибутов R равно объединению подмножеств A, B, C . Если R удовлетворяет функциональной зависимости $A \rightarrow B$, то R равно соединению его проекций $\{A, B\}$ и $\{A, C\}$.

Доказательство:

Обозначим проекции за $R1$ и $R2$. Доказательство разделим на два шага: доказательство того, что естественное соединение $R1$ и $R2$ содержит все

кортежи отношения R и доказательство того, что среди результата данного естественного соединения нет лишних кортежей.

Шаг 1. Пусть кортеж $\{a, b, c\}$ принадлежит R. Тогда по определению операции взятия проекции кортеж $\{a, b\}$ принадлежит проекции $\{A, B\}$ и кортеж $\{a, c\}$ принадлежит проекции $\{A, C\}$. Следовательно по определению естественного соединения, кортеж $\{a, b, c\}$ принадлежит соединению отношений R1 и R2.

Шаг 2. Надо доказать, что если кортеж $\{a, b, c\}$ принадлежит соединению отношений R1 и R2, то кортеж $\{a, b, c\}$ принадлежит R. Если кортеж $\{a, b, c\}$ принадлежит соединению отношений R1 и R2, то существуют кортеж $\{a, b\}$, принадлежащий $\{A, B\}$, и кортеж $\{a, c\}$, принадлежащий $\{A, C\}$. Может ли оказаться, что $\{a, b\}$ соединится с неправильной парой $\{a, c\}$? Последнее условие может выполняться в том и только в том случае, когда существует кортеж $\{a, b_2, c\}$, принадлежащее R. Но поскольку отношение R удовлетворяет функциональной зависимости $A \rightarrow B$, то $b=b_2$ и, следовательно, $\{a, b, c\} = \{a, b_2, c\}$. Теорема доказана.

Пример

Рассмотрим отношение R:

Группа	Кол-во	Специальность
4311	15	5201
4361	15	3515
2311	21	5201

Декомпозиция без потерь:

Группа	Кол-во	Группа	Специальность
4311	15	4311	5201
4361	15	4361	3515
2311	21	2311	5201

Декомпозиция с потерями

Группа	Кол-во	Кол-во	Специальность
4311	15	15	5201
4361	15	15	3515
2311	21	21	5201

1-я нормальная форма.

Отношение R находится в первой нормальной форме (1НФ) тогда и только тогда, когда значения всех его атрибутов атомарны.

Атрибут атомарен, если его значение теряет смысл при *любом* разбиении на части или переупорядочивании. Следовательно, если какой-либо способ разбиения на части не лишает атрибут смысла, то атрибут *не* атомарен.

Например, значение «4286» является

- **атомарным**, если его смысл — «пин-код кредитной карты» (при разбиении на части или переупорядочивании смысл теряется)
- **неатомарным**, если его смысл — «набор цифр» (при разбиении на части или переупорядочивании смысл **не** теряется)

Пример

не удовлетворяет 1НФ

Преподаватели-предметы	
Преподаватель	Предмет
Иванов	Физика
	Химия
Петров	Физ-ра
	Ин-яз

1НФ

Преподаватели-предметы	
Преподаватель	Предмет
Иванов	Физика
Иванов	Химия
Петров	Физ-ра
Петров	Ин-яз

Недостатки 1НФ

У рассмотренной таблицы легко заметить следующие проблемы:

1. Добавление - нельзя сделать запись о том, что студент учится на какой-то специальности, не добавив информацию о хотя бы одном уроке.
2. Удаление - обратное, если мы удалим все записи о предметах для какого-либо студента, то потеряем информацию и о его группе и специальности.
3. Изменение - при изменении группы у студента, надо исправить не одно значение, а значение всех записей, соответствующих его урокам.

2-я нормальная форма.

Отношение находится во второй нормальной форме тогда и только тогда, когда оно находится в 1НФ и каждый неключевой* атрибут неприводимо зависит от первичного ключа.

* Атрибут называется неключевым, если он не входит в состав ни одного возможного ключа. Иначе он называется ключевым.

Замечание

Определение 2НФ было дано только для отношения с одним возможным ключом.

Если говорить простым языком, то оно может быть трактовано следующим образом.

Если для каждой функциональной зависимости вида:

ключ → неключ. атрибуты

слева нельзя опустить ни один атрибут, без потери части смысла, то отношение находится во 2НФ.

Определение для отношений с несколькими ключами

Отношение находится во второй нормальной форме тогда и только тогда, когда оно находится в 1НФ и каждый неключевой атрибут неприводимо зависит от любого возможного ключа.

Пример:

Персоны		
Студент	Специальность	Группа
Иванов	5102	1311
Петров	3515	1361

Проблемы:

1. Добавление - нельзя добавить информацию о том, что студенты группы такой-то учатся по специальности такой-то, если нет ни одной записи о студентах этой группы.
2. Удаление - при удалении всех студентов группы, теряется информация о том, какую специальность изучала данная группа.

Изменение - при изменении номера специальности надо проводить изменения ни одной записи, а всех, соответствующих студентам данной группы.

Причина заключается в наличие транзитивных зависимостей неключевых атрибутов от ключевых.

Или, проще говоря, с наличием взаимозависимостей между неключевыми атрибутами группа и специальность.

3-я нормальная форма.

Отношение находится в 3НФ тогда и только тогда, когда оно находится во 2НФ и отсутствуют транзитивные зависимости неключевых атрибутов от ключевых.

Пояснение: транзитивной зависимостью неключевых атрибутов от ключевых будем считать следующее:

$$A \rightarrow B \quad B \rightarrow C,$$

где A - набор ключевых атрибутов (ключ), B и C - различные множества неключевых атрибутов.

Поэтому мы можем сказать, что необходимо убедиться лишь в отсутствии взаимозависимостей между неключевыми атрибутами.

Персона-группа		Группа-специальность	
Студент	Группа	Специальность	Группа
Иванов	1311	5102	1311
Петров	1361	3515	1361

Нормальная форма Бойса-Кодда.

Хотя данное определение для третьей нормальной формы применимо как для случая с одним потенциальным ключом, так и для случаев с большим количеством потенциальных ключей, тем не менее, во втором случае некоторые проблемы еще остаются.

Поэтому в случае, когда отношение содержит два или более потенциальных ключа обычно говорят не о 3НФ, а о нормальной форме Бойса-Кодда.

3 условия.

1. Отношение имеет два (или более) потенциальных ключа.
2. Эти ключи – составные.
3. Два или более потенциальных ключа перекрываются.

Если хотя бы одно из этих условий не выполняется, то НФБК совпадает с 3НФ.

Определение НФБК:

Определение: отношение находится в НФБК тогда и только тогда, когда детерминанты всех его ФЗ являются потенциальными ключами.

Пример:

Предположим, создаётся таблица бронирования для теннисных кортов на день: **{Номер корта, Время начала, Время окончания, Тариф, Член клуба}**. Тариф зависит от выбранного корта и членства в клубе.

Таким образом, возможны следующие составные первичные ключи: **{Номер корта, Время начала}**, **{Номер корта, Время окончания}**, **{Тариф, Время начала}**, **{Тариф, Время окончания}**.

Таблица соответствует второй и третьей нормальной форме, так как атрибуты, не входящие в состав первичного ключа, зависят от составного первичного ключа целиком ([2NF](#)) и нет транзитивных зависимостей ([3NF](#)).

Тем не менее, существует функциональная зависимость тарифа от номера корта. То есть, по ошибке можно нарушить логическую целостность и, например, приписать тариф *Premium* для первого корта, хотя тариф *Premium* может относиться только ко второму корту.

Можно улучшить структуру, разбив таблицу на две: **{Номер корта, Время начала, Время окончания, Член клуба}** и **{Тариф, Номер корта, Член клуба}**. Данное отношение будет соответствовать BCNF.

Многозначная зависимость.

Пусть задано отношение R и три его произвольных подмножества атрибутов A, B, C. Тогда говорят, что B многозначно зависит (МЗЗ) от A ($A \twoheadrightarrow B$) (или есть многозначная зависимость B от A) тогда и только тогда, когда множество значений атрибута B, соответствующих паре значений атрибутов A и C, зависит только от значения атрибута A, но не зависит от значения атрибута C.

4-я нормальная форма.

Отношение R находится в четвёртой нормальной форме тогда и только тогда, когда оно находится в первой нормальной форме и для любой его нетривиальной многозначной зависимости $A \twoheadrightarrow B$ выполняется, что все атрибуты отношения R функционально зависят от A.

Пример

Предположим, что рестораны производят разные виды пиццы, а службы доставки ресторанов работают только в определенных районах города. Составной первичный ключ соответствующей переменной отношения включает три атрибута: **{Ресторан, Вид пиццы, Район доставки}**. Такая переменная отношения не соответствует 4НФ, так как существует следующая многозначная зависимость:

- **{Ресторан} \twoheadrightarrow {Вид пиццы}**
- **{Ресторан} \twoheadrightarrow {Район доставки}**

То есть, например, при добавлении нового вида пиццы придется внести по одному новому кортежу для каждого района доставки. Возможна логическая аномалия, при которой определенному виду пиццы будут соответствовать лишь некоторые районы доставки из обслуживаемых рестораном районов. Для предотвращения аномалии нужно декомпозировать отношение, разместив независимые факты в разных отношениях. В данном примере следует выполнить декомпозицию на **{Ресторан, Вид пиццы}** и **{Ресторан, Район доставки}**.

Однако если к исходной переменной отношения добавить атрибут, функционально зависящий от потенциального ключа, например цену с учетом стоимости доставки (**{Ресторан, Вид пиццы, Район доставки} \rightarrow Цена**), то полученное отношение будет находиться в 4НФ и его уже нельзя

подвергнуть декомпозиции без потерь. Указанные выше многозначные зависимости в данном случае называются *внедрёнными зависимостями*.

5-я нормальная форма.

Пример:

Поставщики		
Поставщик	Товар	Проект
Иванов и Ко	Гайки	Лекарство от рака
Иванов и Ко	Болты	Лекарство от рака
Иванов и Ко	Гайки	Станция Мир
Петров и Ко	Гайки	Станция Мир
...

Отношение «Поставщики» содержит три атрибута «Поставщик», «Товар» и «Проект». Предполагается что каждый поставщик умеет поставлять строго определённый набор товаров. Для каждого проекта требуется также строго определённый набор товаров. Причём, если известно, что поставщик z работает с проектом x , и, что для проекта x требуются товар y и, что поставщик z умеет поставлять товар y , то отношение «Поставщики» обязательно содержит запись $\{z, y, x\}$.

Можно догадаться, что отношение декомпозируется без потерь на три более простых.

Декомпозиция без потерь

Поставщик-товар		Товар-проект	
Поставщик	Товар	Товар	Проект
Иванов и Ко	Гайки	Гайки	Лекарство от рака
Иванов и Ко	Болты	Болты	Лекарство от рака
Петров и Ко	Гайки	Гайки	Станция Мир
...

Поставщик-проект	
Поставщик	Проект
Иванов и Ко	Лекарство от рака
Иванов и Ко	Станция Мир
Петров и Ко	Станция Мир
...	...

При большом количестве записей и хотя бы на порядок меньшем разнообразии значений атрибутов такая декомпозиция даст преимущество с точки зрения занимаемого дискового пространства. Экономия дискового пространства может, в свою очередь, привести и к другим улучшениям. Но

откуда взялась идея такой декомпозиции? Чтобы понять это введём несколько определений.

Зависимость соединения

Пусть задано отношение R и подмножества A_1, A_2, \dots, A_n множества его атрибутов. Тогда говорят, что отношение R удовлетворяет зависимости соединения $\{A_1, A_2, \dots, A_n\}$ тогда и только тогда, когда любое допустимый набор кортежей отношения R эквивалентен набору кортежей отношения, получаемого путем соединения его проекций по подмножествам атрибутов A_1, A_2, \dots, A_n .

Замечание: зависимость соединения – это ещё более общая форма многозначной зависимости.

Тривиальная зависимость соединения

Зависимость соединения $\{A_1, A_2, \dots, A_n\}$ для отношения R называется тривиальной тогда и только тогда, когда одно из подмножеств атрибутов A_1, A_2, \dots, A_n совпадает с самим отношением R.

Очевидно, что для любого отношения можно найти более одной тривиальной зависимости соединения.

Подразумевается ключами

Говорят, что зависимость соединения $\{A_1, A_2, \dots, A_n\}$ отношения R подразумевается возможными ключами, если каждое из подмножеств атрибутов A_1, A_2, \dots, A_n включает в себя какой-либо из возможных ключей целиком.

Определение:

Отношение R находится в пятой нормальной форме (5НФ) тогда и только тогда, когда оно находится в первой нормальной форме и каждая нетривиальная зависимость соединения в нём подразумевается возможными ключами.

Пример когда декомпозиция бессмысленна:

Заказы		
Номер заказа	Клиент	Сумма
12345	Пупкин В.А.	1500
12346	Козлов К.К.	2500
12347	Свинкина Х.П.	1500
12348	Пупкин В.А.	1300
...

Теорема Фейгина

Пусть дано отношение $R(A, B, C)$, где A, B и C - непересекающиеся подмножества множества атрибутов R. Причем множество атрибутов R равно объединению подмножеств A, B, C. Отношение R может быть декомпозировано без потерь на его проекции $\{A, B\}$ и $\{A, C\}$ тогда и только тогда, когда оно удовлетворяет многозначной зависимости $A \twoheadrightarrow B$.

21. Основы языка SQL. Транзакции и контрольные точки. Индексные структуры: простые индексы, вторичные индексы, В-деревья, хэш-таблицы, индексы с несколькими ключами.

SQL (Structured Query Language) представляет собой подязык данных, содержащий конструкции для создания и обработки базы данных. Язык SQL был разработан компанией IBM и был принят Американским национальным институтом стандартов (ANSI) в качестве национального стандарта США в 1992 году.

Основу языка SQL составляют операторы, условно разбитые на несколько групп по выполняемым функциям:

- Операторы DDL (Data Definition Language) – операторы определения объектов базы данных.
- Операторы DML (Data Manipulation Language) – операторы манипулирования данными.
- Операторы защиты и управления данными, и др.

Операторы DDL (Data Definition Language) - операторы определения объектов базы данных

CREATE TABLE

Оператор CREATE TABLE служит для создания отношений.

Иногда вместе с оператором CREATE TABLE используется оператор ALTER для указания первичных и вторичных ключей.

CREATE TABLE ПРОЕКТ

(ИдПроекта Integer Primary Key.

Название Char(25) Unique Not Null,

Отдел VarChar(100) Null,

МаксТрудозатраты Numeric(6,1) Default 100);

Каждый столбец описывается тремя характеристиками: именем, типом данных и необязательными ограничениями. Есть пять типов ограничений:

1. PRIMARY KEY,
2. UNIQUE,
3. NULL/NOT NULL (по умолчанию подразумевается ограничение NULL),
4. FOREIGN KEY,
5. CHECK.

Первичные ключи ни при каких условиях не могут иметь пустых значений, и ограничение PRIMARY KEY включает в себя ограничение NOT NULL.

Определение первичных и альтернативных ключей с помощью оператора **ALTER**.

После того как таблица определена, ее структуру, свойства и ограничения можно изменить, используя оператор ALTER. В частности, с помощью этого оператора можно создавать первичные и альтернативные ключи.

Пример:

ALTER TABLE СОТРУДНИК ADD CONSTRAINT СотрудникПК PRIMARY KEY (ТабельныйНомер);

Операторы DROP

В языке SQL имеется множество других операторов и конструкций для определения данных. Одним из самых полезных среди них является оператор DROP TABLE. Но он одновременно является и одним из самых

опасных, поскольку удаляет таблицу из базы данных вместе со всеми содержащимися в ней данными.

Оператор DROP TABLE не выполняется, если таблица содержит или может содержать значения, необходимые для соблюдения ограничений ссылочной целостности.

Пример: DROP TABLE <ИМЯ ТАБЛИЦЫ>

Транзакции и контрольные точки.

Под *транзакцией* понимается некоторая группа DML команд. Все изменения сделанные ими, сохраняются в отдельной области памяти до окончательного подтверждения изменений (успешное завершение транзакции), либо до их отмены. Если во время транзакции делается запрос на выборку данных, то создается отдельное представление. Для чего нужны транзакции? В англоязычной литературе концепция транзакций описывается аббревиатурой ACID:

- атомарность - выполнение или не выполнение всех DML команд входящих в транзакцию;
- целостность БД - завершение транзакции не должно нарушать целостность БД;
- изоляция - можно отображать либо исходные данные, которые были до начала транзакции, либо новые данные после выполнения транзакции;
- сохранность данных - если пользователю пришло подтверждение выполнения транзакции, то его изменения не будут отменены по каким-либо причинам.

В стандарте предусмотрены следующие команды управления транзакциями:

- *START TRANSACTION* - явное начало транзакции. Команда не поддерживается в Oracle. В MySQL и PostgreSQL можно использовать синоним *begin* (не путать с блоковым оператором *begin*, после которого нет разделителя). Если начало транзакции явно не указано, то PostgreSQL считает каждую DML команду отдельной транзакцией. В Oracle транзакции следуют одна за другой. То есть первая DML команда открывает транзакцию, следующие команды становятся частью этой транзакции, пока не будет вызвана команда *commit* или *rollback*. Если режим автоподтверждения (*autocommit*) включен, то MySQL работает также как PostgreSQL, иначе как Oracle;
- *COMMIT* - завершить транзакцию, применяя все сделанные изменения;
- *ROLLBACK* - завершить транзакцию, отменяя все сделанные изменения. Если точка отката не указана, то отменяется вся текущая транзакция;
- *SAVEPOINT* - сохранить точку отката;
- *RELEASE SAVEPOINT* - уничтожить точку отката, что позволяет освободить часть ресурсов до завершения транзакции;
- *SET TRANSACTION* - устанавливает характеристики текущей транзакции. Если транзакция не начата явно, то эта команда игнорируется в PostgreSQL.

Журнализация изменений — функция [СУБД](#), которая сохраняет информацию, необходимую для восстановления [базы данных](#) в предыдущее консистентное состояние в случае логических или физических отказов. В простейшем случае журнализация изменений заключается в последовательной записи во [внешнюю память](#) всех изменений, выполняемых в базе данных. Записывается следующая информация:

- порядковый номер, тип и время изменения;
- идентификатор [транзакции](#);
- объект, подвергшийся изменению (номер хранимого файла и номер блока данных в нём, номер строки внутри блока);
- предыдущее состояние объекта и новое состояние объекта.

Формируемая таким образом информация называется **журнал изменений** базы данных. Журнал содержит отметки начала и завершения транзакции, и отметки принятия [контрольной точки](#) (см. ниже).

В [СУБД с отложенной записью](#) блоки данных внешней памяти снабжаются отметкой порядкового номера последнего изменения, которое было выполнено над этим блоком данных. В случае сбоя системы эта отметка позволяет узнать какая версия блока данных успела достичь внешней памяти.

СУБД с отложенной записью периодически выполняет контрольные точки. Во время выполнения этого процесса все незаписанные данные переносятся на внешнюю память, а в журнал пишется отметка принятия контрольной точки. После этого содержимое журнала, записанное до контрольной точки может быть удалено.

Журнал изменений может не записываться непосредственно во внешнюю память, а аккумулироваться в оперативной. В случае подтверждения транзакции СУБД дожидается записи оставшейся части журнала на внешнюю память. Таким образом гарантируется, что все данные, внесённые после сигнала подтверждения, будут перенесены во внешнюю память, не дожидаясь переписи всех изменённых блоков из [дискового кэша](#). СУБД дожидается записи оставшейся части журнала так же при выполнении контрольной точки.

В случае логического отказа или сигнала отката одной [транзакции](#) журнал сканируется в обратном направлении, и все записи отменяемой транзакции извлекаются из журнала вплоть до отметки начала транзакции. Согласно извлеченной информации выполняются действия, отменяющие действия транзакции, а в журнал записываются **компенсирующие записи**. Этот процесс называется **откат** (rollback).

В случае физического отказа, если ни журнал, ни сама база данных не повреждена, то выполняется процесс **прогонки** (rollforward). Журнал сканируется в прямом направлении, начиная от предыдущей контрольной точки. Все записи извлекаются из журнала вплоть до конца журнала. Извлеченная из журнала информация вносится в блоки данных внешней памяти, у которых отметка номера изменений меньше, чем записанная в журнале. Если в процессе прогонки снова возникает сбой, то сканирование журнала вновь начнется сначала, но фактически восстановление продолжится с той точки, откуда оно прервалось.

Контрольная точка - момент синхронизации между базой данных и журналом регистрации транзакций. В этот момент все буфера системы принудительно записываются во вторичную память системы. Контрольные точки организуются через установленный интервал времени и предусматривают выполнение следующих действий.

- Перенос всех имеющихся в оперативной памяти записей журнала во вторичную память.
- Запись всех модифицированных блоков в буферах базы данных во вторичную память.
- Помещение в файл журнала записи контрольной точки. Эта запись содержит идентификаторы всех транзакций, которые были активны в момент создания контрольной точки.

Обычно создание контрольных точек представляет собой относительно недорогую операцию, поэтому часто оказывается возможным создать три или даже четыре контрольные точки в час. В результате при сбое потребуется восстановить работу, выполненную всего лишь за последние 15-20 минут.

Основные виды индексов

- Простые индексы для упорядоченных файлов
- Вторичные индексы для неупорядоченных файлов
- В-деревья (В+-деревья)
- Хэш-таблицы

Индексы для последовательных файлов

Плотный индекс

- Размер файла индекса значительно меньше
- Возможность бинарного поиска
- Есть вероятность загрузки индекса в память

Разреженный индекс

Более сложные варианты

- Многоуровневый индекс
- Индексы для файлов с дубликатами ключей
 - Плотный с дублированием
 - Плотный
 - Разреженный с наименьшими значениями
 - Разреженный с наименьшими новыми значениями

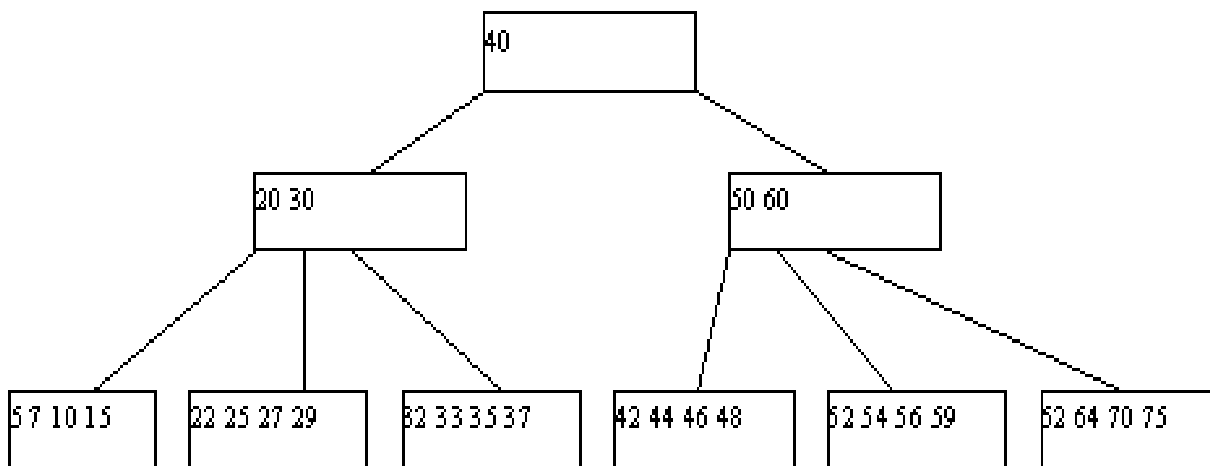
Операции с индексами

- Удаление
 - Может быть модифицировано путем добавления “мертвых” записей, остающихся на месте удаленных
- Вставка
 - Может быть модифицирована путем использования блоков переполнения

Вторичные индексы

- Плотный вторичный индекс
- Двухуровневый плотный индекс
- Многоуровневые

В-деревья



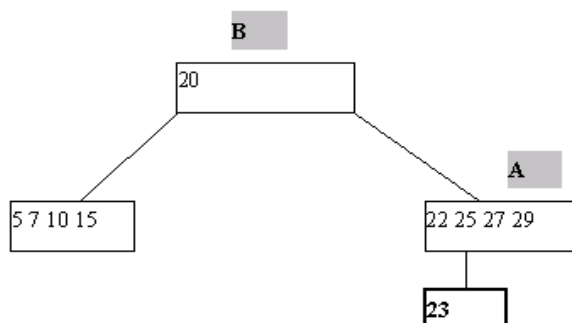
Классическое В-дерево порядка 2

Поиск в В-дереве

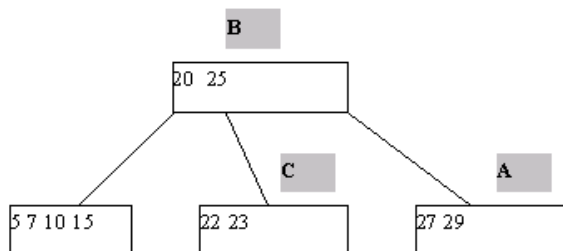
1. Если в считанной странице обнаруживается пара ключей k_i и $k_{(i+1)}$ такая, что $k_i < K < k_{(i+1)}$, то поиск продолжается на странице p_i .
2. Если обнаруживается, что $K > k_m$, то поиск продолжается на странице p_m .
3. Если обнаруживается, что $K < k_1$, то поиск продолжается на странице p_0 .

Вставка в В-дерево

Пытаемся вставить:

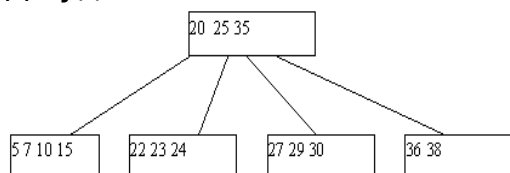


Вставка путём расщепления Страницы A:

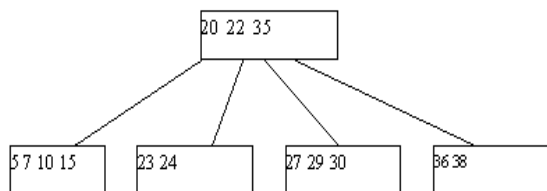


Исключение из В-дерева

До удаления:

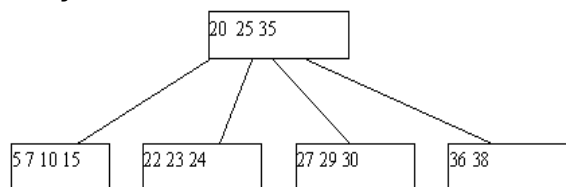


После удаления 25:

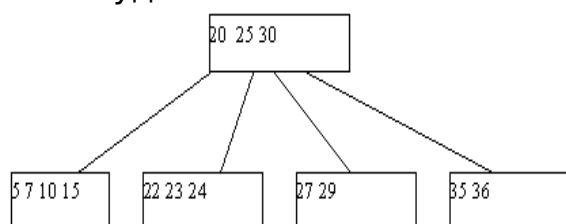


Исключение с переливанием ключей

До удаления:

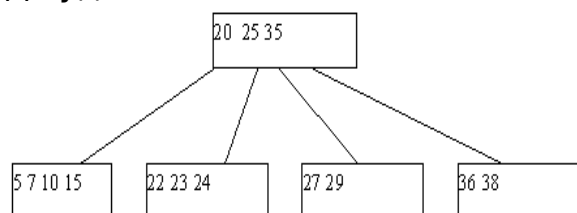


после удаления 38:

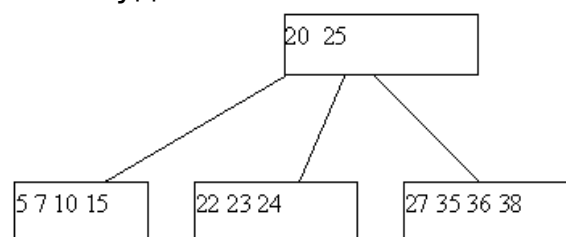


Исключение со слиянием страниц

До удаления:



После удаления 29:



B+ деревья

- 1) Не листовые вершины содержат только ключи и ссылки на дочерние страницы.
- 2) Листовые вершины содержат все множество ключей отношения, сами указатели на записи, плюс указатель на следующий по порядку лист.
- 3) Ключи в листовых вершинах отсортированы по возрастанию.

Эффективность B+ деревьев

Возьмем значение блока равным 4096 байт(что часто встречается на практике).

Пусть указатель занимает 8 байт, а ключ 4. Тогда блок вмещает максимум 340 индексов. Кол-во индексов в блоке в среднем = 255. Тогда дерево глубиной в 3 уровня может адресовать 255^3 записей (16 581 375).

Хеш-таблицы

Хеш-функция вычисляет по ключу номер сегмента, где должна быть размещена запись.

Особенности хеш-функций для внешней памяти:

- 1) В роли сегментов выступают блоки
- 2) Каждый сегмент содержит возможность для создания блока переполнения

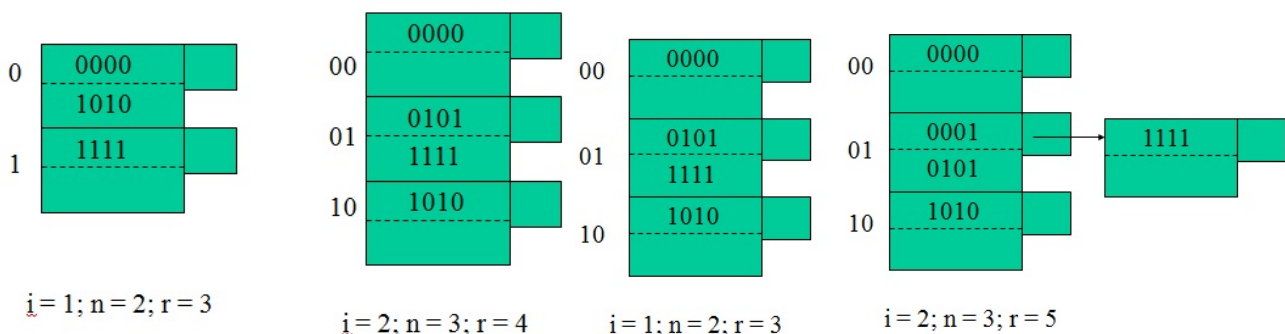
Динамические хеш-таблицы

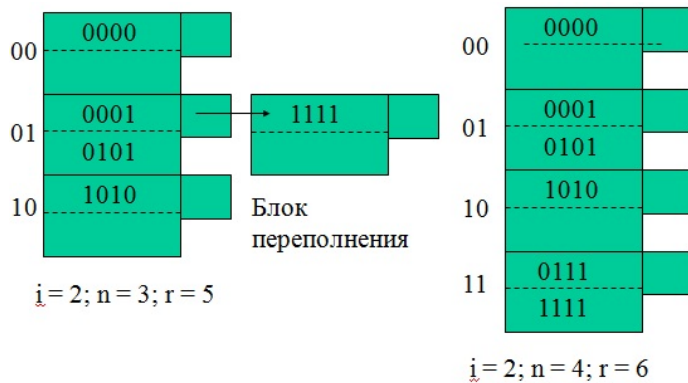
- Расширяемые хеш-таблицы
- Линейные хеш-таблицы

Линейные хеш-таблицы

- Количество сегментов линейной хеш-таблицы (n) всегда выбирается так, чтобы отношение среднего числа записей в блоке к максимально допустимому оставалось постоянным (например, 80%)
- Допустимо применять блоки переполнения, но их число, отнесенное к общему числу сегментов должно быть существенно меньше единицы
- Количество битов, используемых для нумерации элементов массива сегментов (i) составляет $\lceil \log_2 n \rceil$. Биты выбираются из младших разрядов двоичного представления числа, возвращаемого хеш-функцией ($h(K)$)
- Выбор сегмента для размещения ссылки на запись с ключом K осуществляется следующим образом: если число m образуемое младшими i битами $h(K) < n$, то запись кладется в сегмент с соответствующим номером. Иначе в двоичном представлении числа m заменяем старшие биты на 0 до тех пор, пока это условие не выполнится. Полученное в результате число и есть номер сегмента, куда надо распределить запись.

Пример:





24. Интерполяция и аппроксимация кривых и поверхностей. Аппроксимация сплайнами.

СПЛАЙН-АППРОКСИМАЦИЯ

приближенное представление функции или приближенное восстановление функции из заданного класса по неполной информации (напр., по значениям на сетке) с помощью *сплайнов*.

Как и в классич. теории приближения функций, изучаются линейные методы С.-а., включая *сплайн-интерполяцию*, наилучшие методы, а также аппроксимации классами нелинейных сплайнов, напр. сплайнами с нефиксированными узлами.

Наилучшие приближения сплайнами. Изучаются вопросы существования, единственности, характеристич. свойства наилучшего сплайна (н. с.) (см. *Наилучшего приближения элемент*), а также порядки, асимптотика и точные верхние грани уклонений сплайнов от заданного класса функций. Сплайны с фиксированными узлами не образуют *Чебышева систему*, поэтому в $C[a, b]$ нет единственности н. с. и характеристич. свойства н. с. сложнее, чем характеристич. свойства *Наилучшего приближения многочлена* (см. [8]). Однако в $L[a, b]$ для подкласса непрерывных, функций н. с., если они склеиваются из гладких функций, образующих систему Чебышева на $[a, b]$, обладают свойствами единственности [2]. Сплайны с фиксированной гладкостью, но с нефиксированными узлами (предполагается, что число узлов не превосходит заданного числа) не образуют замкнутого множества, поэтому здесь может не существовать н. с. Порядок приближения может быть охарактеризован следующим результатом [6]:

$$\begin{aligned} & \|f^{(i)}(x) - S_{m, \Delta_n}^{(i)}(x)\|_{L_p[a, b]} \leq \\ & \leq c \|\Delta_n\|^{1 - \frac{1}{p} + \frac{1}{q}} \omega_{m-i}(f^{(i+1)}, \|\Delta_n\|)_q, \quad (1) \\ & 1 \leq p \leq q \leq \infty, \end{aligned}$$

$$S_{m, \Delta_n}(x)$$

где $S_{m, \Delta_n}(x)$ - полиномиальный сплайн степени m с узлами в точках сетки Δ_n

$$\Delta_n: a = x_0^{(n)} \leq x_1^{(n)} < \dots < x_n^{(n)} = b, \\ \|\Delta_n\| = \max_{0 \leq i \leq n-1} (x_{i+1} - x_i),$$

$$\omega_k(f, \delta)_q$$

- модуль гладкости порядка k в $L_q[a, b]$ и функция $f(x)$ имеет L_q ($1 \leq l \leq m$),

абсолютно непрерывную $(l-1)$ -ю производную и l -ю из

$$1 \leq q \leq p \leq \infty$$

$i=0, 1, \dots, l-1$. При $i=0$ в (1) можно заменить на $i-1$ и убрать

$$\|\Delta_n\|^{1-\frac{1}{p} + \frac{1}{q}}.$$

множитель

Более слабые аналоги неравенства (1)

$$f \in W_2^k(\Omega) \quad (W_2^k(\Omega))$$

получены для многомерных сплайнов. Напр., если

$$S_h^k$$

пространство Соболева) и S_h^k - совокупность сплайнов (степени не выше k по каждой переменной) с равномерными узлами и шагом h на области

$$\Omega$$

удовлетворяет строгому условию конуса (см. Вложения теоремы), то

$$\inf_{S \in S_h^k} \|f - S\|_{W_2^j(\Omega)} \leq c \cdot h^{k-j} \|f\|_{W_2^k(\Omega)}, \quad 0 \leq j \leq k.$$

$$(\|\Delta_n\| = 1/n)$$

$$W_q^{m+1}$$

Для равномерной сетки

и класса

порядок правой

$$1 \leq p \leq q \leq \infty \quad n^{\frac{1}{p} - \frac{1}{q} - m - 1 + i}.$$

части в (1) при

равен

Если

рассматривать приближение сплайнами степени $m-1$ с нефиксированными узлами, число k -рых не превосходит n , то за счет выбора узлов можно добиться [7], чтобы порядок аппроксимации был равен n^{-m-1+i} . Для наилучшего равномерного приближения нек-рых классов периодических функций полиномиальными сплайнами с равномерными узлами имеется ряд окончательных результатов. Напр.,

$$\tilde{W}^r H_\omega, \quad \omega(\delta)$$

для класса $\tilde{W}^r H_\omega$, где $\omega(\delta)$ - выпуклый модуль непрерывности, подсчитана верхняя грань уклонения от сплайнов степени r [4]. Она совпадает с соответствующим поперечником этого класса. Изучаются также наилучшие приближения сплайнами при дополнительных ограничениях на его старшую Производную [13]. В связи с изучением наилучших квадратурных формул естественно возникает задача наилучшего приближения специальной функции $(b-t)^r$ (см. Монослайн). Линейные методы приближения сплайнам и начали изучать раньше наилучших приближении. При этом преимущественно изучались приближения интерполяционными сплайнами (и. с.) (см. [1],

[3], [5]). И. с. часто дают тот же порядок приближения, что и наилучшие; это является одним из преимуществ перед интерполированием многочленами. Так, если функция $f(x)$ имеет

$(-\infty, \infty)$,

непрерывную r -ю производную на $(-\infty, \infty)$, то для приближения

полиномиальными и. с. $S_n(x, h)$ степени $n \geq r$ с равномерными узлами

$i=0, \pm 1, \pm 2, \dots$,

интерполяции $x_i = ih$, и узлами сплайна

справедлива оценка [6].

$$\|f^{(i)}(x) - S_n^{(i)}(x, h)\|_{C(-\infty, \infty)} \leq c \cdot \omega_{r+1-i}(f^{(i)}, h), \quad i=0, 1, \dots, r.$$

При изучении и. с. с произвольными узлами в качестве параметра приближения выбирается максимальное расстояние между узлами интерполяции; обычно узлы интерполяции и узлы сплайна тесно связаны между собой. В приложениях наиболее широко используются полиномиальные и. с. $S_3(x)$ 3-й степени - кубические сплайны. Это связано с тем, что построение таких сплайнов сводится в большинстве случаев к решению системы линейных уравнений с трехдиагональной матрицей, имеющей доминирующую главную диагональ. Решение таких систем легко реализуется на ЭВМ. Кроме того, если функция

$0 \leq k \leq 3$,

$f(x)$ имеет непрерывную k -ю, производную на $[a, b]$, то имеют место оценки

$$\|f^{(i)}(x) - S_3^{(i)}(x)\|_{C[a, b]} \leq c \|\Delta_n\|^{k-i} \omega(f^{(k)}, \|\Delta_n\|), \quad 0 \leq i \leq k,$$

$\{x_i^{(n)}\}$

где $\{x_i^{(n)}\}$ - узлы интерполяции. При $k=1, 2$ константа $c > 0$ не зависит от

Δ_n .

Δ_n

f и от сеток Δ_n . При $k=0$ и $k=3$ на последовательность сеток налагаются дополнительные ограничения. Аналог этого результата имеет место также для многомерных кубических сплайнов, а также для сплайнов большей степени.

И. с. нечетной степени обладает рядом экстремальных свойств. Напр., среди всех функций, имеющих абсолютно непрерывную $(\tau-1)$ -ю производную на $[a, b]$ и m -ю производную из $L_2[a, b]$ и принимающих в точках $\{x_i\}$, $a < x_0 < x_1 < \dots < x_n < b$, заданные значения $\{y_i\}$, полиномиальный сплайн $S_{2m-1}(x)$ с узлами $\{x_i\}$, принимающий в точках $\{x_i\}$ значения $\{y_i\}$, имеющий непрерывную $(2\tau-2)$ -ю производную на $[a, b]$ и совпадающий на $[a, x_0]$ и $[x_n, b]$ с многочленами степени не выше $\tau-1$, имеет наименьшую норму τ -й производной в $L_2[a, b]$. Это свойство послужило основой для многочисленных обобщений сплайнов. Для некоторых классов функций верхняя грань уклонов от и. с. совпадает с

$\tilde{W}^r H_\omega$

верхней гранью уклонов для и. с., напр. для класса $\tilde{W}^r H_\omega$ при

$\omega(\delta) = \delta$.

Сплайны играют важную роль в задаче сглаживания [3], [5] сеточной функции, заданной с погрешностью. С помощью сплайнов строятся

базисы [5] и ортонормированные базисы [9], Лебега константы к-рых ограничены.

Методы С.-а. тесно связаны с численным решением уравнений в частных производных методом конечных элементов, в основе к-рого лежит Ритца метод при специальном выборе базисных функций. В методе конечных элементов в качестве базисных функций выбираются

кусочно полиномиальные функции, т. е. сплайны. Пусть, напр., $\Omega \subset \mathbb{R}^2$, -

ограниченная область из \mathbb{R}^2 , к-рую можно разложить на конечное число $1 \leq i \leq N$.

правильных треугольных подобластей T_i , Для фиксированного i многочлен

$$P_i(p_{ij}) = \alpha_1 + \alpha_2 x_1 + \alpha_3 x_2 + \alpha_4 x_1^2 + \alpha_5 x_1 x_2 + \alpha_6 x_2^2$$

определяется из условий

$$P_i(p_{ij}) = f(p_{ij}), \quad P_i(q_{ij}) = f(q_{ij}), \quad j = 1, 2, 3,$$

где функция $f(p)$ непрерывна на $\bar{\Omega}$ и p_{ij} - вершины треугольника T_i , а q_{ij} - середины его сторон. Пусть $S(p) = P_i(p)$ при $p \in T_i$, $i = 0, 1, \dots, N$.
 $f \in W_2^3(\Omega)$,

Если то

$$\|f - S\|_{W_2^j(\Omega)} \leq ch^{3-j} \|f\|_{W_2^3(\Omega)}, \quad j = 0, 1,$$

где h - длина стороны треугольника T_i и c -абсолютная постоянная.