



Control Work Algorithms



Содержание:

- | | |
|--------------------|------------------------------------|
| 1. Хеширование | 3. Сеть и поток |
| 2. Графы | — Фолда-Фалкерсона |
| — Эйлеров граф | — Эдмондса-Карпа |
| — Гамильтонов граф | — Диницы |
| — Union-Find | — Дерево Гомори-Ху |
| — Флёри | — Хопркофта-Карпа |
| — Косарайю | — Венгерский |
| — Тарьяна | — Вырезание соцветий |
| — Кана | 4. Раскраска, планарность, укладка |
| — Прима | — Хроматический многочлен |
| — Краскала | — Greedy-Coloring |
| — Борувки | — Гамма-алгоритм |
| — Дейкстра | |
| — Беллмана-Форда | |
| — Флойда-Уоршелла | |
| — Джонсона | |
| — A* | |

Хеширование

Хеширование — процесс отображения объекта на целое число в диапазоне $[0; M - 1]$.

Хеш-таблицы используют хеш-функцию вместе с некоторым механизмом обработки коллизий

Свойства хорошей хеш-функции:

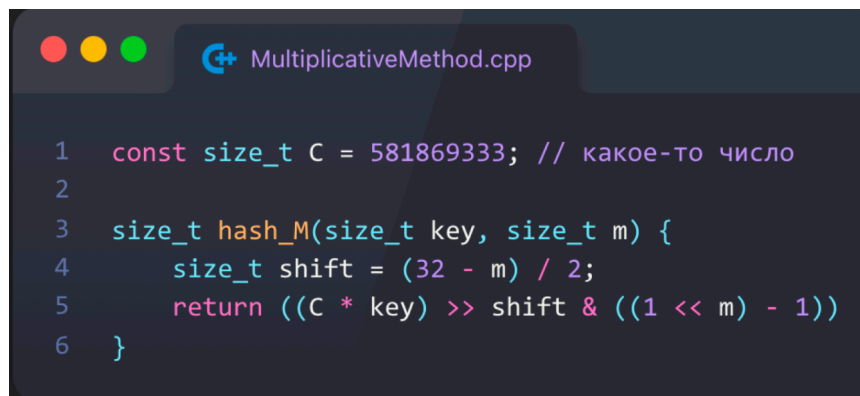
1. Быстрый процесс $O(1)$.
2. Распределение вычисляемых значений стремится к равномерному.

3. Детерминированность: если $x = y$, то $hash(x) = hash(y)$.
4. Хеш должен определяться всеми битами (разрядами) key .

Mapping Down — метод отображения в структурах данных, где индекс в структуре данных определяется только младшими разрядами целочисленного ключа.

Mapping Down с использованием mod определяется младшими разрядами. Преимущества: быстро, просто. Минус: плохое распределение, если N — степень 2 или кратно ключам.

В мультипликативном методе Mapping Down хеш-значение формируется на основе дробной части произведения ключа на константу. Участвуют все разряды числа. Преимущества: лучше распределяет, меньше коллизий. Минусы: медленнее, нужна хорошая константа.



```
1  const size_t C = 581869333; // какое-то число
2
3  size_t hash_M(size_t key, size_t m) {
4      size_t shift = (32 - m) / 2;
5      return ((C * key) >> shift & ((1 << m) - 1))
6  }
```

Обфускация разрядов в целом числе — процесс искажения или маскирования битов числа с целью усложнения его анализа, предсказания или восстановления исходного значения.

Универсальное семейство:

H — множество хеш-функций. Такое семейство универсальное тогда и только тогда, когда $\forall x, y \in U : x \neq y \quad |h \in H : h(x) = h(y)| = \frac{|H|}{M}$.

В универсальном множестве вероятность коллизий — $\frac{1}{m}$.

создание универсального семейства	
ШАГ 1 условие	выбрать некоторое простое число в качестве M
ШАГ 2 пред-обработка	разбить ключ key на $r + 1$ разрядов $key = \langle k_0, k_1, \dots, k_r \rangle$, где $k_i \in [0, M)$
ШАГ 3 случайность	выбрать случайные коэффициенты $a = \langle a_0, a_1, \dots, a_r \rangle$, где $a_i \in [0, M)$
ШАГ 4 хеш-функция	составить хеш-функцию $hash_M(key) = \left(\sum_{i=0}^r a_i \cdot r_i \right) \bmod M$

Идеальное хеширование полностью исключает возможность коллизий.

Идеальное хеширование разумно применять для статичных наборов данных.

Двухэтапный процесс хеширования — отображения множества объектов на множество индексов.

Нельзя хешировать изменяемые структуры (`list`, `dict`, `set`).

Два основных способа взлома пароля по значению хеш-функции в случае получения доступа к хешам:

1. Грубая сила — полный перебор случайных паролей до получения совпадения по значению.
2. Радужные таблицы — большой набор предварительно вычисленных хешей.

Хеширование с солью — добавление соли перед хешированием. Соль — дополнительная случайная строка некоторой длины, которая приписывается к паролю перед тем, как вычислить его хеш.

Хеширование кукушкой — наличие дополнительной хеш-функции, которая используется, если по первой происходит коллизия.

Вычисление значения хеш-функции не является примером стохастического алгоритма.

`std::hash<...>`

Имеет специализации для фундаментальных типов `bool`, `char`, `int`, `float`, `double`, ...

Возвращает значение типа `std::size_t` и не порождает исключений, обладает свойством детерминированности. Вероятность коллизий стремится к

$$\frac{1.0}{std::numeric_limits::size_t::max()}.$$

1. Закрытая адресация

- a. Формируется список объектов с одним и тем же хешем.
- b. Хеш-таблица представляет собой массив связанных списков.
- c. Коэффициент заполненности **load factor** $\lambda = \frac{\text{количество элементов}}{\text{размер хеш-таблицы}}.$
- d. Если **load factor** превышает некоторый порог, надо сделать перехэширование. Обычно увеличивают размер таблицы в 2 раза.
- e. Слабости метода цепочек: выделение дополнительной памяти, операции могут выполняться за $O(n)$.
- f. Можно использовать вместо цепочек сбалансированные деревья поиска.

2. Открытая адресация

- a. При коллизии объект напрямую добавляется в таблицу.
- b. w.h.p. (*with high probability*) — с высокой вероятностью. Время работы не гарантировано $O(\dots)$ в худшем случае, но с вероятностью, стремящейся к 1 при больших n , оно такое.
- c. Математическое ожидание даёт усреднённую оценку времени работы алгоритма, а w. h. p. даёт относительно точные гарантии того, как будет работать алгоритм.
- d. Линейное пробирование:
 - i. Требуется $O(\log n)$ времени w. h. p.
 - ii. При поиске либо находим объект, либо находим пустую ячейку, либо просмотрен весь массив при $\lambda = 1$.
 - iii. Появляются кластеры.
 - iv. Среднее количество проб при успешном поиске: $\frac{1}{2}(1 + \frac{1}{1-\lambda})$.
 - v. Среднее количество проб при неуспешном поиске или вставке: $\frac{1}{2}(1 + \frac{1}{(1-\lambda)^2})$.
 - vi. Нельзя просто удалить объект.

vii. При ленивом удалении надо помечать ячейку, тогда при поиске считать её заполненной, при вставке — пустой.

viii. Умное удаление:

1) Если

`currentIndex < emptyCell`, сдвинуть A на пустое место при условии `hash(A) <= emptyCell` и `hash(A) > currentIndex`.

2) Если

`currentIndex > emptyCell`, сдвинуть A на пустое место при условии `hash(A) <= emptyCell` или `hash(A) > currentIndex`.

е. **Квадратичное пробирование:**

i. M — простое число p . Гарантируется, что простое квадратичное пробирование посетит $\frac{p}{2}$ ячеек хеш-таблицы.

ii. Среднее количество проб при успешном поиске $\frac{\ln(\frac{1}{1-\lambda})}{\lambda}$ и неуспешном $\frac{1}{1-\lambda}$.

ф. **При двойном хешировании** сдвиг вычисляется с помощью второй хеш-функции.

Двойное хеширование

Двойное хеширование (англ. double hashing) — метод борьбы с коллизиями, возникающими при открытой адресации, основанный на использовании двух хеш-функций для построения различных последовательностей исследования хеш-таблицы.

Принцип двойного хеширования

При двойном хешировании используются две независимые хеш-функции $h_1(k)$ и $h_2(k)$. Пусть k — это наш ключ, m — размер нашей таблицы, $n \bmod m$ — остаток от деления n на m , тогда сначала исследуется ячейка с адресом $h_1(k)$, если она уже занята, то рассматривается $(h_1(k) + h_2(k)) \bmod m$, затем $(h_1(k) + 2 \cdot h_2(k)) \bmod m$ и так далее. В общем случае идёт проверка последовательности ячеек $(h_1(k) + i \cdot h_2(k)) \bmod m$ где $i = (0, 1, \dots, m-1)$

Таким образом, операции вставки, удаления и поиска в лучшем случае выполняются за $O(1)$, в худшем — за $O(m)$, что не отличается от обычного **линейного разрешения коллизий**. Однако в среднем, при грамотном выборе хеш-функций, двойное хеширование будет выдавать лучшие результаты, за счёт того, что вероятность совпадения значений сразу двух независимых хеш-функций ниже, чем одной.

$$\forall x \neq y \exists h_1, h_2 : p(h_1(x) = h_1(y)) > p((h_1(x) = h_1(y)) \wedge (h_2(x) = h_2(y)))$$

Алгоритм

Основная идея хеширования кукушки — использование двух хеш-функций вместо одной (далее $h_1(x)$ и $h_2(x)$). Также есть вариант алгоритма, в котором используются две хеш-таблицы, и первая хеш-функция указывает на ячейку из первой таблицы, а вторая — из второй. Рассмотрим алгоритмы функций $add(x)$, $remove(x)$ и $contains(x)$.

Выберем 2 хеш-функции $h_1(x)$ и $h_2(x)$ (из универсального семейства хеш-функций).

Add

Добавляет элемент с ключом x в хеш-таблицу

1. Если одна из ячеек с индексами $h_1(x)$ или $h_2(x)$ свободна, кладем в нее элемент.
2. Иначе произвольно выбираем одну из этих ячеек, запоминаем элемент, который там находится, помещаем туда новый.
3. Смотрим в ячейку, на которую указывает другая хеш-функция от элемента, который запомнили, если она свободна, помещаем его в нее.
4. Иначе запоминаем элемент из этой ячейки, кладем туда старый. Проверяем, не зациклились ли мы.
5. Если не зациклились, то продолжаем данную процедуру поиска свободного места пока не найдем свободное место или зациклимся.
6. Иначе выбираем 2 новые хеш-функции и переходим к добавлению элементов.
7. Также после добавления нужно увеличить размер таблицы в случае если она заполнена.

Remove

Удаляет элемент с ключом x из хеш-таблицы.

1. Смотрим ячейки с индексами $h_1(x)$ и $h_2(x)$.
2. Если в одной из них есть искомым элемент, просто помечаем эту ячейку как свободную.

Contains

Проверяет на наличие элемента x в хеш-таблице

1. Смотрим ячейки с индексами $h_1(x)$ и $h_2(x)$.
2. Если в одной из них есть искомым элемент, возвращаем true.
3. Иначе возвращаем false.

Защипывание

Защипывание может возникнуть при добавлении элемента. Пусть мы добавляем элемент x . И обе ячейки $h_1(x)$ и $h_2(x)$ заняты. Элемент x положили изначально в ячейку $h_1(x)$. Если в ходе перемещений элементов в таблице на очередном шаге мы опять хотим переместить элемент x в ячейку $h_1(x)$, чтобы в ячейку $h_2(x)$ ($i \neq j$) мы смогли поместить какой-то y (это может произойти, если в ходе перемещений элемент x был перемещен в ячейку $h_j(x)$), то произошло защипывание.

Например, защипывание возникнет, если добавить в хеш-таблицу 3 элемента x, y, z у которых $h_1(x) = h_1(y) = h_1(z)$ и $h_2(x) = h_2(y) = h_2(z)$.

Одним из способов решения проблемы защипывания является смена хеш-функций, что было доказано Джоном Траппом^[1]

Время работы алгоритма

Удаление и проверка происходит за $O(1)$ (что является основной особенностью данного типа хеширования), добавление в среднем происходит за $O(1)$. Первые два утверждения очевидны: требуется проверить всего лишь 2 ячейки таблицы.



3. Фильтр Блума

- а. Требуется $1.44 \cdot n \cdot \log_2\left(\frac{1}{\epsilon}\right)$ бит. Размер объектов никак не влияет.
- б. Позволяет отфильтровать $1 - \epsilon$ запросов $q \notin S$.
- в. Чтобы не было ложно-положительного срабатывания, достаточно одного ложного бита.
- г. С ростом заполненности возрастает вероятность ложно-положительного срабатывания.
- д. Не поддерживается удаление.
- е. Затраты по памяти никак не соотносятся с размером объектов.
- ж. Внутри: пусть мощность множества S составляет n , а вероятность ложно-положительного срабатывания ϵ :
 - для идентификации объектов используется $k = \log_2 \frac{1}{\epsilon}$ хеш-функций h_1, h_2, \dots, h_n .
 - информация о принадлежности хранится в битовом массиве размера $m = nk \cdot \log_2 e = 1.44 \cdot n \cdot \log_2 \frac{1}{\epsilon}$.

Если используем k хеш-функций вероятность ложного срабатывания равна $(1 - e^{-\frac{mk}{n}})^k$, оптимальное количество хеш-функций равно $\frac{n}{m} \cdot \ln(2)$, где m — количество элементов, n — размер фильтра.

4. Фильтр кукушки

- С ростом заполненности потребуется больше времени для поиска, поэтому надо делать перехеширование.
- Можно удалять объекты из таблицы.
- Хеширование кукушки требует $O(\log n)$ обменов w.h.p.

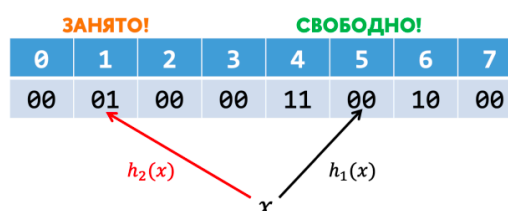
Устройство фильтра кукушки

ε — доля ложно-положительных срабатываний

- k хеш-функций h_1, h_2, \dots, h_k , где k не зависит от ε
- функция-отпечаток [fingerprint] $f: U \rightarrow \{1, 1/\varepsilon\}$
предположив, что $1 + 1/\varepsilon = 2^f$, мы можем зарезервировать f бит для хранения отпечатка
- хеш-таблица на m слотов, каждый из которых хранит f бит
 m также не зависит от ε — обычно определяется как $m = C \cdot n$, где n — число объектов

ПРИМЕР КОНФИГУРАЦИИ

- $n = 4$ объекта и $m = 2n = 8$
- $\varepsilon = 1/3$ и $f = 2$ бита на слот
- $k = 2$ хеш-функции
- $f(x) = 2_{10} = 10_2$



$$i = h_1(x) = \text{hash}(x)$$

$$j = h_2(x) = h_1(x) \oplus \text{hash}(\text{fingerprint}(x))$$

фильтр Блума VS фильтр кукушки

Фильтр Блума	Фильтр кукушки
вставка и проверка принадлежности объектов требует вычисления значений k различных хеш-функций	на практике используется особая схему с двумя хеш-функциями
время вставки остается неизменным вне зависимости от заполненности битового вектора(-ов)	с ростом заполненности хеш-таблицы хеширование кукушки потребует больше времени для поиска свободной ячейки — потребуется перехеширование
с ростом заполненности битового вектора(-ов) фильтра Блума значительно возрастает вероятность ложно-положительного срабатывания	целевой порог вероятности ложно-положительного ответа может остаться неизменным до заполнения на 95.5% (практические данные)
не поддерживается удаление объектов, так как иначе возможны ложно-отрицательные ответы	объекты, о которых точно известно, что они были добавлены в фильтр, могут быть удалены

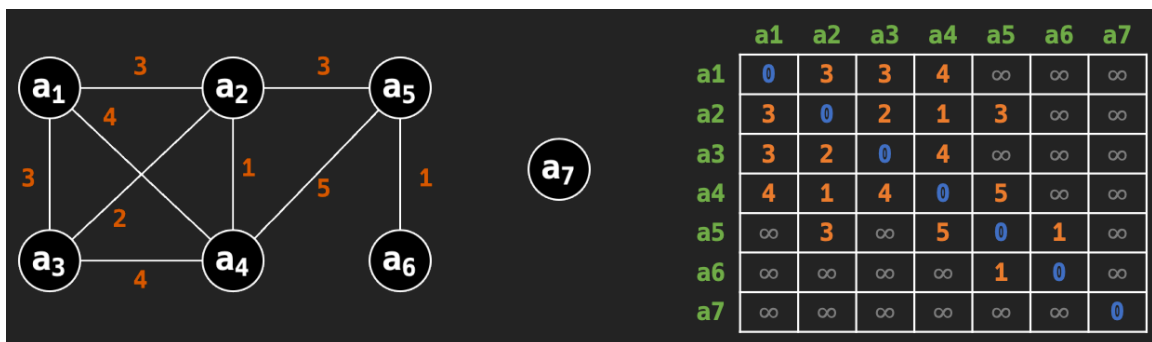
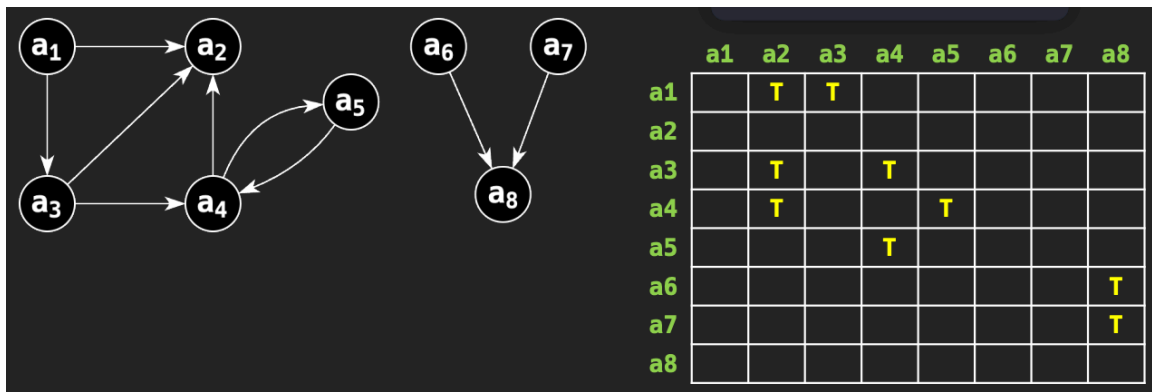
5. Метод цепочек

- Хеш-таблица представляет собой массив связанных списков.

- б. Минусы: Использование дополнительной памяти для организации линейных одно(дву)связных списков; Основные операции ADT словаря деградируют до операций на связных списках.
6. **Ошибка первого рода**: ложно-положительное срабатывание. Объекта нет, но говорим, что есть.
 7. **Ошибка первого рода**: ложно-отрицательное срабатывание. Объект есть, но говорим, что его нет.
 8. Чем больше значение ϵ (вероятность ложно-положительного срабатывания), тем меньше памяти.
 9. **Список с пропусками Skip-List** — $O(\log n)$. В идеальном случае каждый уровень содержит половину ключей от предыдущего. Представляет вероятностную реализацию ADT Словаря — INSERT, SEARCH, DELET.

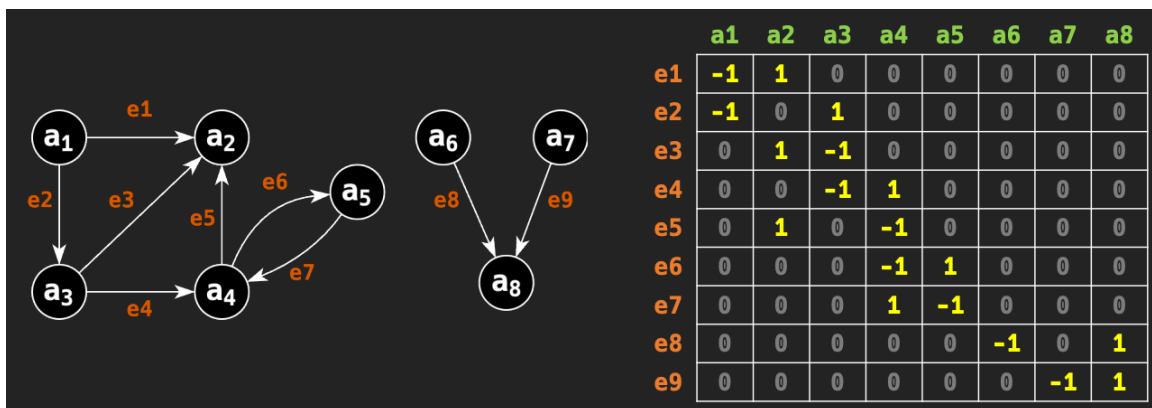
Графы

1. Максимальное число рёбер в графе — $\frac{V \cdot (V-1)}{2} = O(V^2)$.
2. В связанном графе $G = (V, E)$ без циклов $|V| = |E| + 1$.
3. Степень вершины в неориентированном графе — количество смежных вершин.
4. Степень исхода в ориентированном графе — количество исходящих дуг. Вершина с нулевой степенью исхода — сток. Вершина с нулевой степенью захода — исток.
5. Граф связанный, если между любой парой вершин есть путь.
6. Список рёбер: по памяти $O(E)$, все базовые операции $O(E)$.
7. **Матрица смежности**:



8. Матрица инцидентности:

Начало дуги **-1**, конец дуги **1**.



9. Список смежности — список, в котором для каждой вершины записываются все достижимые вершины.

структура	память	проверить ребро	добавить ребро	удалить ребро	получить соседей
список ребер	$O(E)$	$O(E)$	$O(1)$	$O(E)$	$O(E)$
матрица смежности	$O(V^2)$	$O(1)$	$O(1)$	$O(1)$	$O(V)$
матрица инцидентности	$O(V \cdot E)$	$O(E)$	$O(E)$	$O(E)$	$O(E)$
список смежности	$O(V + E)$	$O(k)$	$O(1)$	$O(k)$	$O(1)$

Топологический порядок — порядок на вершинах графа, при котором для любой дуги (A, B) вершина B следует после вершины A . Если граф имеет топологический порядок на вершинах, то в нем нет циклов.

Компоненты связности:

- В **неориентированном** графе — максимальные подграфы, в которых любые две вершины соединены путём.
- В **ориентированном** графе компоненты слабой связности — максимальные группы вершин, которые остаются соединёнными, если убрать направления рёбер.
- В **ориентированном** графе компоненты сильной связности — максимальные подграфы, в которых из каждой вершины можно попасть в любую другую по направлению рёбер.

Мост — ребро, при удалении которого количество компонент связности увеличивается.

Точка сочленения — вершина, удаление которой приводит к увеличению числа компонент связности.

Проверить граф на двудольность можно с помощью раскраски в 2 цвета.

Эйлеров граф

Эйлеров путь — путь в графе, который посещает каждое ребро графа 1 раз.

Эйлеров цикл — эйлеров путь, который начинается и заканчивается в одной и той же вершине.

Неориентированный граф содержит эйлеров цикл, если он связанный и степени всех вершин чётные.

Неориентированный граф содержит эйлеров путь, если он связанный и нечётную степень имеют 2 вершины. Если вершин с нечетной степенью нет, то существует эйлеров цикл.

В любом графе число вершин с нечётной степенью чётно.

Гамильтонов граф

Гамильтонов путь — путь, который посещает каждую вершину один раз.

Гамильтонов цикл — гамильтонов путь, который начинается и заканчивается в одной и той же вершине.

	Алгоритм	Инструмент	Сложность
Связанность, мосты, точки сочленения	Kosaraju	DFS	$O(V + E)$
Связанность, мосты, точки сочленения	Tarjan	DFS	$O(V + E)$
Упорядочивание вершин	Kahn	Обход вершин по степеням захода	$O(V + E)$
Эйлеровость	Heirholzer	DFS с поиском циклов	$O(V + E)$
Эйлеровость	Fluery	Обход рёбер с проверкой мостов	$O(E^2)$
Гамильтоновость	Roberts & Flores	Перебор с возвратом	$O(V!)$

Остовое дерево соединяет все вершины графа.

Разрез графа — разбиение множества вершин на 2 дизъюнктивных множества.

Light-ребро пересекает линию разреза и имеет наименьший вес среди всех пересекающих рёбер.

Граф конденсации

Граф конденсации (граф компоненты сильной связности) — новый граф, получаемый из исходного ориентированного графа путем сокращения всех компонент сильной связности в отдельные вершины.

Процесс построения:

1. **Нахождение компонент сильной связности:** Разбиваем исходный граф на компоненты сильной связности. Каждая компонента сильной связности является подграфом, в котором существует путь между любыми двумя вершинами, причем все вершины компоненты достижимы друг от друга.
2. **Сокращение компонент:** Каждую компоненту сильной связности заменяем на одну вершину. Это значит, что каждая компонента становится одной вершиной в графе конденсации.
3. **Добавление рёбер:** Если существует ребро, которое соединяет вершины разных компонент, то в графе конденсации добавляется ребро между соответствующими вершинами компонент.

Union-Find

Используется: проверка, принадлежат ли два элемента одному множеству.

Можно реализовывать на основе массива, списка или деревьев.

Алгоритм Флёри

Сложность: $O(E^2)$.

Используется: для нахождения эйлерова пути и цикла в графе.

Идея:

1. Проверяем, что граф эйлеров:
 - Все вершины имеют чётную степень → есть эйлеров цикл.
 - Ровно две вершины имеют нечётную степень → есть эйлеров путь.
2. Стартуем с вершины:
 - Если путь → с одной из двух нечётных вершин.
3. Идём по рёбрам, удаляя их, но не разрывая граф:

- Если есть несколько вариантов, не выбираем мост, пока есть альтернативы.
- Удаляем ребро и продолжаем путь.

4. Когда рёбра закончились — путь найден.

Алгоритм Косарайю

Сложность: $O(V + E)$.

Используется: для нахождения компонент сильной связности в ориентированном графе, поиск циклов.

Идея:

1. Проход по графу в глубину (DFS) и запоминание порядка выхода:
 - Обходим граф DFS, записывая вершины в стек по мере завершения их обработки.
2. Транспонирование графа (разворачиваем все рёбра)
3. Второй обход DFS по новому графу:
 - Берём вершины в порядке из стека и запускаем DFS
 - Все вершины, достижимые из текущей — это одна КСС.

| Результат не зависит от стартовой вершины.

Алгоритм Тарьяна

Сложность: $O(V + E)$.

Используется: для нахождения компонент сильной связности в ориентированном графе, поиск циклов.

Идея:

1. Запускаем DFS и каждому узлу присваиваем:
 - ID (порядок посещения)
 - low-link (минимальный ID, достижимый через обратные рёбра)

2. Используем стек для отслеживания текущей КСС.

3. Во время обратного прохода DFS:

- Если встречаем вершину, у которой `low == ID`, значит, нашли КСС.
- Достаём вершины из стека, пока не дойдём до текущей.

4. Повторяем для всех вершин, пока не разберём граф.

```
index := 0
stack := []
for each v in V do
  if v.index = null then
    strongconnect(v)

function strongconnect(v)
  v.index := index
  v.lowlink := index
  index := index + 1
  stack.push(v)
  v.onStack := true

  for each (v, w) in E do
    if w.index = null then
      strongconnect(w)
    v.lowlink := min(v.lowlink, w.lowlink)
    else if w.onStack then
      v.lowlink := min(v.lowlink, w.index)

  if v.lowlink = v.index then
    создать новую компоненту сильной связности
    repeat
      w := stack.pop()
      w.onStack := false
      добавить w в текущую компоненту сильной связности
    while w ≠ v
    вывести текущую компоненту сильной связности
```

Алгоритм Кана

Сложность: $O(V + E)$.

Используется: для топологической сортировки ориентированного ациклического графа.

Идея:

1. Считаем входящие степени вершин (сколько рёбер в них входит).
2. Добавляем в очередь все вершины с входной степенью 0 (они не зависят от других).
3. Берём вершину из очереди → добавляем в топологический порядок → удаляем её рёбра → уменьшаем входные степени у её соседей.
4. Если у соседа входная степень стала 0 → кладём его в очередь.
5. Повторяем, пока не обработаем все вершины.

Алгоритм Прима

Сложность: $O(E \cdot \log V)$ при использовании кучи или $O(E + V \cdot \log V)$ с матрицей смежности.

Используется: поиск минимального остового дерева (MST) в взвешенном неориентированном графе.

Идея:

1. Начинаем с любой вершины.
2. Добавляем в остовое дерево ребро с минимальным весом, которое соединяет уже включённые вершины с новыми.
3. Обновляем доступные рёбра и повторяем шаг 2, пока не покроем все вершины.

Алгоритм Краскала

Сложность: $O(E \cdot \log E)$.

Используется: поиск минимального остового дерева (MST) в взвешенном неориентированном графе.

Идея:

1. Сортируем рёбра по весу (от меньшего к большему).
2. Добавляем рёбра одно за другим, если они не создают цикл (используем Union-Find).
3. Повторяем, пока не добавим $(V - 1)$ рёбер (где V — число вершин).

Алгоритм Борувки

Сложность: $O(E \cdot \log V)$.

Используется: поиск минимального остового дерева (MST) в взвешенном неориентированном графе.

Идея:

1. Каждая вершина является отдельным компонентом.
2. На каждом шаге для каждой компоненты выбирается ребро минимального веса, которое соединяет её с другой компонентой.
3. Все найденные минимальные рёбра добавляются в остовое дерево.
4. Слияние компонентов: После того как минимальные рёбра для всех компонент найдены, компоненты объединяются.
5. Повторяем шаги 2-4, пока не останется только одна компонента (остовное дерево покрывает все вершины).

Алгоритм Дейкстры

Сложность: $O((E + V) \cdot \log V)$ при использовании приоритетной очереди или $O(E + V \cdot \log V)$ при использовании матрицы смежности.

Используется: жадный алгоритм для нахождения кратчайших путей от одной вершины до всех остальных вершин в взвешенном графе с неотрицательными весами рёбер.

Идея:

1. Инициализация:

- Устанавливаем расстояние до начальной вершины равным 0, а до всех остальных — бесконечность.
- Используем приоритетную очередь (или кучу) для хранения вершин с минимальными расстояниями.

2. Поиск кратчайшего пути:

- Выбираем вершину с минимальным расстоянием, которая ещё не была посещена.
- Для каждой соседней вершины проверяем, может ли кратчайший путь до неё быть улучшен через текущую вершину. Если да — обновляем расстояние.
- Повторяем этот процесс до тех пор, пока все вершины не будут обработаны.

3. Завершение:

- После завершения алгоритма для каждой вершины будет найдено минимальное расстояние от начальной вершины.

Алгоритм Беллмана-Форда

Сложность: $O(V \cdot E)$.

Используется: для нахождения кратчайших путей в графе, который может иметь отрицательные веса рёбер.

Идея:

Основан на динамическом программировании и постепенно улучшает кратчайшие пути от начальной вершины к остальным вершинам. Он выполняет несколько итераций, в ходе которых обновляются расстояния до всех вершин.

1. Инициализация:

- Устанавливаем расстояние от начальной вершины до самой себя равным 0, а до всех остальных вершин — бесконечность.

2. Основной цикл:

- Выполняем $V - 1$ итераций (где V — количество вершин в графе). На каждой итерации мы рассматриваем все рёбра и проверяем,

можем ли мы улучшить расстояние до вершины, используя это ребро.

- Если для ребра (u, v) с весом w выполняется условие $distance[u] + w < distance[v]$, то обновляем расстояние до вершины v .

3. Проверка на отрицательные циклы:

- После выполнения $V - 1$ итераций проверяем все рёбра ещё раз. Если для какого-либо ребра можно улучшить расстояние, это означает, что граф содержит отрицательный цикл.

Алгоритм Флойда-Уоршелла

Сложность: $O(V^3)$.

| Сложность линейно не зависит от количества рёбер.

Используется: для нахождения кратчайших путей между всеми парами вершин в взвешенном графе. Он работает для графов с отрицательными весами рёбер, но не работает с графами, содержащими отрицательные циклы.

Идея:

Использует динамическое программирование для нахождения кратчайших путей между всеми парами вершин графа. Он постепенно улучшает кратчайшие пути, используя другие вершины как промежуточные точки.

1. Инициализация:

- Строится матрица расстояний, где $dist[i][j]$ — это кратчайшее расстояние от вершины i до вершины j . Изначально:
 - $dist[i][j] = \text{вес ребра между } i \text{ и } j$, если такое ребро существует.
 - $dist[i][j] = \infty$, если рёбер между i и j нет.
 - $dist[i][i] = 0$, для всех вершин i .

2. Основной цикл:

- Алгоритм выполняет V итераций (где V — количество вершин). На каждой итерации рассматриваем все возможные промежуточные вершины, чтобы улучшить кратчайшие пути. Для каждой пары вершин (i, j) проверяем, можем ли мы улучшить путь от i до j через промежуточную вершину k , где:

$$dist[i][j] = \min(dist[i][j], dist[i][k] + dist[k][j])$$

- Если через вершину k путь короче, обновляем $dist[i][j]$.

3. Завершение:

- После V итераций матрица $dist$ будет содержать кратчайшие расстояния между всеми парами вершин.

Внешний цикл по k .

Алгоритм Джонсона

Сложность: $O(V^2 \cdot \log V + V \cdot E)$ на фибначчиевой куче и $O(V \cdot (V + E) \cdot \log V + V \cdot E)$ на бинарной куче.

Используется: для нахождения кратчайших путей между всеми парами вершин в графе. Преимущество заключается в эффективности работы с графами, которые могут содержать отрицательные рёбра, но не содержат отрицательных циклов.

Идея:

1. Применение **Алгоритма Беллмана-Форда** для перераспределения весов рёбер, чтобы все веса стали неотрицательными.
2. Применение **Алгоритма Дейкстры** для нахождения кратчайших путей для каждой вершины с перераспределёнными весами.

Шаги:

1. Добавление вспомогательной вершины:
 - Добавляем новую вершину s в граф и соединяем её рёбрами с каждой из существующих вершин с нулевым весом. Таким образом, новая вершина будет подключена ко всем остальным, и её можно будет использовать для перераспределения весов.

2. Использование Беллмана-Форда:

- Запускаем Алгоритм Беллмана-Форда из новой вершины s . Это позволит нам вычислить кратчайшие расстояния от s до всех других вершин с учётом исходных весов рёбер. Эти значения используются для перераспределения весов рёбер. Если на последней итерации Беллмана-Форда обнаружатся улучшения, значит, в графе есть отрицательные циклы, и алгоритм не может быть выполнен.

3. Перераспределение весов рёбер:

- После выполнения алгоритма Беллмана-Форда, для каждой вершины v вычисляем значение $h[v]$, которое будет равно расстоянию от вершины s до вершины v . Мы перераспределяем веса рёбер с учётом значений h :

$$w'(u, v) = w(u, v) + h[u] - h[v]$$

Где

$w(u, v)$ — это исходный вес ребра, а $w'(u, v)$ — перераспределённый вес. Таким образом, все веса рёбер становятся неотрицательными, и можно применять алгоритм Дейкстры.

4. Применение алгоритма Дейкстры:

- Для каждой вершины u графа выполняем Алгоритм Дейкстры с перераспределёнными весами, чтобы найти кратчайшие пути от u ко всем остальным вершинам.

5. Восстановление исходных расстояний:

- После применения алгоритма Дейкстры, восстанавливаем исходные веса путём обратного перераспределения:

$$d(u, v) = d'(u, v) + h[v] - h[u]$$

Где

$d'(u, v)$ — кратчайшее расстояние с учётом перераспределённых весов, а $d(u, v)$ — искомое расстояние с исходными весами рёбер.

Алгоритм A*

Сложность: $O(b^d)$.

- b — это количество соседей каждой вершины (степень графа).
- d — это глубина поиска, или максимальное количество шагов до целевой вершины.

Используется: поиск кратчайшего пути в графе, который используется для нахождения оптимального пути между двумя вершинами, часто применяемый в задачах навигации и искусственном интеллекте. A* сочетает в себе преимущества алгоритмов поиска в ширину (BFS) и жадных алгоритмов, используя эвристику для ускорения поиска пути.

Идея:

Алгоритм A* строит путь от начальной вершины к целевой, при этом оценивает не только расстояние до текущей вершины, но и прогнозирует, как далеко находится цель, используя эвристическую функцию.

Оценка пути:

Для каждой вершины v алгоритм A* вычисляет функцию стоимости $f(v)$, которая определяется как сумма двух составляющих:

$$f(v) = g(v) + h(v)$$

где:

- $g(v)$ — это стоимость пути от начальной вершины до вершины v .
- $h(v)$ — это эвристическая оценка (прогноз) стоимости пути от вершины v до целевой вершины. Это функция, которая должна быть невозрастающей (то есть никогда не переоценивать фактическую стоимость пути). Например, для задачи нахождения пути на плоскости это может быть эвристика в виде евклидова расстояния или Манхэттенского расстояния.

Цель алгоритма — минимизировать $f(v)$ для всех возможных вершин, выбирая в первую очередь те, для которых сумма $g(v) + h(v)$ наименьшая.

Шаги:

1. Инициализация:

- Начинаем с начальной вершины и присваиваем её g -стоимость равной 0, а h -стоимость — эвристическому расстоянию до целевой вершины. Функция $f(v)$ для начальной вершины будет $f(start) = g(start) + h(start)$.

2. Расширение вершин:

- Помещаем начальную вершину в открытый список (open list), который будет хранить вершины, которые ещё нужно рассмотреть.
- Создаём закрытый список (closed list), где будут храниться уже обработанные вершины.

3. Основной цикл:

- Пока открытый список не пуст, выбираем вершину с минимальным значением $f(v)$ из открытого списка (это будет вершина, которая на данный момент кажется наиболее перспективной).
- Если эта вершина является целевой, то путь найден. Мы можем восстановить путь, двигаясь от целевой вершины к начальной через сохранённые ссылки на предыдущие вершины.
- Для каждой соседней вершины:
 - Вычисляем новую стоимость g -стоимости.
 - Если соседняя вершина ещё не в открытом списке, добавляем её в список и вычисляем её f -стоимость.
 - Если соседняя вершина уже в открытом списке и новый путь через текущую вершину дешевле, обновляем её стоимость и её родителя.

4. Конец работы:

- Если открытый список опустел, это значит, что целевая вершина недостижима (например, граф содержит изолированные компоненты или препятствия).

	Алгоритм	Инструмент	Сложность
Single-Source Shortest Path	Dijkstra	BFS	$O((E + V) \cdot \log V)$ или $O(E + V \cdot \log V)$

	Алгоритм	Инструмент	Сложность
Single-Source Shortest Path	A*	BFS	$O(b^d)$
Single-Source Shortest Path	Bellman-Ford	Динамическое программирование	$O(V \cdot E)$
All-Pairs Shortest Path	Floyd-Warshall	Динамическое программирование	$O(V^3)$
All-Pairs Shortest Path	Jonson	Bellman-Ford + Dijkstra	$O(V \cdot E + V \cdot (V + E) \cdot \log V)$

Сеть и поток

Сеть — это направленный граф, в котором каждое ребро имеет два параметра:

- Пропускную способность (или вес ребра) $c(u, v)$, которая указывает максимальный объем потока, который может пройти по ребру от вершины u к вершине v .
- Направление: рёбра имеют направление, что означает, что поток может двигаться только в одну сторону.

Поток — это функция, которая присваивает каждому ребру значение, представляющее количество потока, которое проходит по этому ребру, с учетом следующих ограничений:

- Скорость потока на ребре $e = (u, v)$ не может превышать пропускной способности $c(u, v)$: $0 \leq flow(u, v) \leq c(u, v)$.
- Сохранение потока: для каждой вершины, кроме исходной и целевой, суммарный поток, входящий в вершину, должен равняться суммарному потоку, выходящему из неё.

Критическое ребро на пути из истока S в сток T в сети — это ребро с минимальной остаточной пропускной способностью.

Паросочетание — произвольная выборка рёбер, в которой никакие два ребра не имеют общей вершины.

Паросочетание максимальное, если к нему невозможно добавить ни одно ребро.

Наибольшее паросочетание включает в себя максимально возможное число рёбер.

Полное паросочетание покрывает все вершины исходного графа.

Поиск максимального паросочетания займёт $O(V \cdot E)$.

Чередующийся путь — путь, рёбра вдоль которого поочерёдно входят и не входят в M .

Увеличивающий путь — чередующийся путь, который начинается и заканчивается в свободных вершинах. Длина увеличивающего пути всего нечётна.

Двудольный граф имеет циклы только чётной длины.

Алгоритм Фолда-Фалкерсона

Сложность: $O((V + E) \cdot V \cdot C)$.

Используется: для нахождения максимального потока в сетях потоков. Этот алгоритм решает задачу нахождения максимального потока от источника s к стоку t в сети с заданными пропускными способностями рёбер.

Идея:

Использует жадный подход, постепенно увеличивая поток по сети, пока это возможно. Он ищет пути из источника в сток, по которым ещё можно провести поток (пути увеличения потока), и увеличивает поток по этим путям.

1. Инициализация: Установим начальный поток на всех рёбрах сети равным нулю.
2. Поиск пути увеличения: Ищем путь от источника s до стока t , по которому ещё можно провести поток (путь увеличения). Для этого обычно используется поиск в глубину (DFS) или поиск в ширину (BFS).

3. Увеличение потока: По найденному пути увеличиваем поток на минимальное значение из пропускных способностей рёбер на этом пути. Пропускные способности рёбер, по которым прошёл поток, уменьшаются, а пропускные способности обратных рёбер увеличиваются.
4. Повторение: Повторяем шаги 2-3, пока не найдём путь увеличения. Если путь не найден, значит, максимальный поток найден.

Алгоритм Эдмондса-Карпа

Сложность: $O(V \cdot E^2)$.

Используется: улучшенная версия алгоритма Форда-Фалкерсона для нахождения максимального потока в поточной сети.

Идея:

Алгоритм использует поиск в ширину (BFS) для нахождения путей увеличения потока и, таким образом, ускоряет процесс нахождения максимального потока по сравнению с базовым алгоритмом Форда-Фалкерсона.

1. Инициализация: Начальный поток на всех рёбрах сети устанавливается равным нулю.
2. Поиск пути увеличения: Используем поиск в ширину (BFS) для нахождения пути от источника s к стоку t , по которому можно провести поток. Путь должен проходить по рёбрам, у которых ещё есть оставшаяся пропускная способность (не заполнены).
3. Увеличение потока: После нахождения пути увеличиваем поток по всем рёбрам этого пути на величину, равную минимальной пропускной способности на пути. Уменьшаем пропускную способность по рёбрам на этом пути, а для обратных рёбер увеличиваем пропускную способность, чтобы учесть возможность возврата потока.

4. Повторение: Повторяем шаги 2-3, пока существует путь увеличения, т.е. пока можно провести ещё поток по сети.

Когда больше нет путей увеличения (BFS не находит пути от источника к стоку), алгоритм завершён, и максимальный поток найден.

Алгоритм Диницы

Сложность: $O(V^2 \cdot E)$.

Используется: для нахождения максимального потока в сети.

Идея:

Использует уровневый граф (или граф, в котором вершины разбиты на уровни) для поиска пути увеличения потока. Это позволяет значительно улучшить производительность, поскольку поиск путей увеличения происходит не по всему графу, а только по определённым рёбрам, которые могут быть использованы для увеличения потока. В отличие от алгоритма Эдмондса-Карпа, который использует поиск в ширину для каждого пути, алгоритм Диницы сокращает количество ненужных обходов и улучшает время работы.

1. Построение уровневого графа:

- Для поиска путей увеличения потока алгоритм строит уровневый граф, в котором вершины разделяются на уровни. Уровень вершины определяется как минимальное количество рёбер, которое нужно пройти от источника, чтобы попасть в эту вершину.
- Строится этот граф с использованием поиска в ширину (BFS) от источника. Рёбра, которые идут от вершины на одном уровне к вершине на следующем уровне, могут быть использованы для поиска пути увеличения потока.

2. Поиск пути увеличения с ограничениями на потоки:

- После того как построен уровневый граф, алгоритм использует поиск в глубину (DFS) для нахождения путей увеличения потока. Однако DFS используется только по рёбрам, которые идут от одной вершины к вершине следующего уровня (из уровня i на уровень $i + 1$).

- Это ограничение на выбор рёбер с уровня в уровень значительно ускоряет процесс поиска путей увеличения.

3. Увеличение потока:

- Как только путь увеличения найден, поток увеличивается вдоль этого пути на минимальное значение из пропускных способностей рёбер на пути.
- Пропускная способность рёбер по пути уменьшается, а для обратных рёбер увеличивается пропускная способность.

4. Повторение:

- Шаги 1-3 повторяются, пока существует путь увеличения. Как только не удаётся найти путь увеличения (поиск в ширину не находит пути), алгоритм завершает свою работу и возвращает максимальный поток.

Алгоритм	Инструмент	Сложность
For-Fulkerson	-	$O((V + E) \cdot V \cdot C)$
Масштабирование	Ford-Fulkerson и Δ	$O((V + E) \cdot E \cdot \log C)$
Edmonds-Karp	Ford-Fulkerson и BFS	$O((V + E) \cdot E \cdot V) \rightarrow O(V \cdot E^2)$
Dinic	Ford-Fulkerson и блокирующие потоки	$O((V + E) \cdot E \cdot V) \rightarrow O(V^2 \cdot E)$
Push-Relabel	Поток по отдельным дугам	$O(V^2 \cdot E)$

Дерево Гомори-Ху

Сложность: $O(V^2 \cdot E)$.

Идея:

Строится для неориентированного графа с пропускными способностями рёбер и позволяет быстро вычислять минимальный разрез для каждой пары вершин. Оно сводит задачу нахождения минимальных разрезов для всех пар вершин к построению единственного дерева, где для каждой

пары вершин в графе мы можем определить минимальный разрез за время $O(1)$ после построения дерева.

1. Начало построения: Для каждой пары вершин u и v из графа нужно найти минимальный разрез, который разделяет эти вершины.
2. Алгоритм: Строится дерево с помощью многократного использования алгоритма для нахождения максимального потока (например, алгоритма Эдмондса-Карпа или алгоритма Диницы).
 - На каждом шаге алгоритм находит минимальный разрез для пары вершин u и v с помощью максимального потока.
 - Этот минимальный разрез является ребром в дереве Гомори-Ху.
 - После нахождения минимального разреза, этот разрез делит граф на два компонента, и с помощью этого разреза продолжается построение дерева для оставшихся компонент.
3. Ребра дерева: В дереве Гомори-Ху каждое ребро соответствует минимальному разрезу между двумя вершинами. Вес ребра — это пропускная способность минимального разреза, который разделяет соответствующие компоненты графа.
4. Результат: Когда дерево построено, минимальный разрез между любой парой вершин u и v можно быстро получить. Он будет равен весу ребра на пути от u до v в дереве Гомори-Ху.

Алгоритм Хопкрофт-Карпа

Сложность: $O(E\sqrt{V})$.

Используется: для поиска наибольшего паросочетания в двудольном графе.

Идея:

1. Построение слоев (BFS)
 - Используется обход в ширину (BFS) для построения многослойного графа, где:
 - Свободные вершины левой доли U находятся в первом слое.

- В следующем слое — их соседние вершины из правой доли V , и так далее.
- Обход продолжается, пока не найдётся хотя бы один увеличивающий путь.

2. Поиск увеличивающих путей (DFS)

- Затем алгоритм выполняет поиск в глубину (DFS), чтобы найти максимальное количество независимых увеличивающих путей.
- Как только путь найден, меняется паросочетание.

3. Обновление и повтор

- Если хотя бы один увеличивающий путь был найден, процесс повторяется.
- Когда BFS уже не находит новых увеличивающих путей, алгоритм завершается.

Венгерский алгоритм

Сложность: $O(n^3)$.

Используется: задачу о назначениях – находит оптимальное паросочетание в двудольном графе с весами. Он применяется, когда есть n заданий и n исполнителей, и нужно распределить их так, чтобы суммарная стоимость (или время) назначения была минимальной.

Идея:

Задача заключается в нахождении максимального паросочетания с минимальной суммой весов в двудольном графе. Это делается за счёт преобразования матрицы стоимости, чтобы найти паросочетание в графе с нулевыми весами.

1. Вычитание минимального элемента в строках и столбцах

- Из каждой строки матрицы вычитаем минимальный элемент этой строки.
- Затем из каждого столбца полученной матрицы вычитаем минимальный элемент этого столбца.

- В результате в матрице появляются нулевые элементы, которые помогут сформировать паросочетание.

2. Построение покрытия нулей минимальным числом линий

- Используется жадный алгоритм: покрываем все нули наименьшим числом строк и столбцов.
- Если число линий равно n (размеру матрицы), то можно построить оптимальное паросочетание и завершить алгоритм.

3. Модификация матрицы (если паросочетание не найдено)

- Если покрытие нулей не удалось сделать за n линий, то:
 - Находим минимальный непокрытый элемент.
 - Вычитаем его из всех непокрытых элементов.
 - Добавляем его ко всем элементам, пересекающимся двумя линиями.
 - Повторяем шаги, пока паросочетание не будет найдено.

Вырезание соцветий

Сложность: $O(V^4)$.

Используется: поиска максимального паросочетания в общем (не обязательно двудольном) графе.

Идея:

В не двудольных графах могут появляться нечётные циклы (blossoms, "соцветия"), которые мешают стандартным алгоритмам поиска увеличивающих путей.

Алгоритм использует два ключевых приёма:

1. Сжатие нечётных циклов – если найдён нечётный цикл, его рассматривают как одну вершину (сжатие соцветия).
2. Поиск увеличивающих путей – после сжатия графа алгоритм применяет поиск увеличивающих путей (как в алгоритме Хопкрофта-Карпа).

Шаги:

1. Поиск увеличивающих путей с чередованием

- Стартуем с непокрытой вершины и строим двудольное разбиение (чередование слоёв).
- Если находим свободную вершину, то увеличивающий путь найден, и мы обновляем паросочетание.
- Если находим цикл нечётной длины (соцветие), переходим к шагу 2.

2. Сжатие нечётных циклов

- Нечётный цикл (blossom) заменяется одной вершиной, уменьшая размер графа.
- Продолжаем поиск увеличивающего пути в сжатом графе.

3. Восстановление пути

- После нахождения увеличивающего пути разворачиваем сжатие и корректируем паросочетание.

Раскраска, планарность, укладка

Правильная k -раскраска графа $G = (V, E)$ — функция $f : V \rightarrow \{1, 2, \dots, k\}$, для которой верно $\forall \{a, b\} \in E : f(a) \neq f(b)$.

Хроматический многочлен:

Хроматический многочлен графа G — многочлен, который описывает количество способов раскрасить вершины графа в k цветов, так чтобы соседние вершины имели разные цвета.

- Простая цепь W_n : $P(W_n, k) = k \cdot (k - 1)^{n-1}$
- Дерево T_n : $P(T_n, k) = P(W_n, k) = k \cdot (k - 1)^{n-1}$
- Треугольник C_3 : $P(C_3, k) = k \cdot (k - 1) \cdot (k - 2)$
- Цикл C_n : $P(C_n, k) = P(W_n, k) - P(C_{n-1}, k)$

Теорема Deletion-Contraction: $P(G, k) = P(G - e, k) - P(\frac{G}{e}, k)$

$G - e$ — граф без ребра e

G/e — граф, в котором концы ребра e склеены

Одного цвета достаточно только для правильной раскраски нуль-графа O_n , который не имеет рёбер. $P(O_n, k) = k^n$.

$\chi(G)$ не превосходит 2, если каждую долю графа можно раскрасить в один из двух доступных цветов. Применимо к циклам чётной длины.

$\chi(G)$ как минимум 3, если есть цикл нечётной длины.

$\chi(G)$ равно как минимум числу вершин для полного графа.

Алгоритм Greedy-Coloring

Сложность: $O(V + E)$, если список смежности, и $O(V^2)$, если матрица смежности.

Используется: для раскрашивания вершин графа, используя минимальное количество цветов. Не всегда даёт оптимальное решение, но работает быстро и просто.

Идея:

Проходим по вершинам в некотором порядке и назначаем наименьший доступный цвет, который не используется у соседних вершин.

1. Берём первую вершину и назначаем ей первый цвет.
2. Идём по остальным вершинам:
 - Смотрим на цвета соседей.
 - Выбираем наименьший цвет, который ещё не использован у соседей.
3. Повторяем процесс для всех вершин.

Планарный граф — это граф, который можно нарисовать на плоскости так, чтобы рёбра не пересекались.

Формула Эйлера: $V - E + F = 2$

- V — количество вершин
- E — количество рёбер

- F — количество граней, включая внешнюю область.

6-color theorem

Для любого планарного графа существует правильная 6-раскраска.

Теорема Понтрягина-Куратовского

Граф $G = (V, E)$ является планарным тогда и только тогда, когда он не содержит подграфов, гомеоморфных K_5 и $K_{3,3}$.

Гамма-алгоритм

Сложность: $O(V^2)$.

Используется: нарисовать граф на плоскости, избегая пересечений рёбер.

Идея:

1. Выбираем произвольный простой цикл в графе и размещаем его на плоскости.
2. Разбиваем граф на сегменты – части, которые ещё не уложены.
3. Последовательно добавляем сегменты, выбирая такие способы размещения, которые минимизируют количество пересекаемых граней.
4. Обновляем конфигурацию граней, следя за тем, чтобы граф оставался планарным.