

# Построение и анализ алгоритмов

[Ссылка](#) на .tex файлы и все-такое

3 июня 2023 г.

## Содержание

<b>1 Поиск с возвращением. Метод Монте-Карло</b>	<b>3</b>
1.1 Идея поиска с возвращением (backtracking) . . . . .	3
1.1.1 Пример на задаче о ферзях . . . . .	3
1.1.2 Подходы метода ветвей и границ по отсеканию вариантов . . . . .	3
1.2 Метод Монте-Карло . . . . .	4
1.2.1 Пример на размере дерева . . . . .	4
1.2.2 Пример на площади фигуры . . . . .	5
1.2.3 Ограничения и условия применимости . . . . .	6
<b>2 Раскраска в 3 цвета</b>	<b>7</b>
2.1 Алгоритм полного перебора . . . . .	7
2.2 Перебор с учётом выбора только из 2 цветов . . . . .	7
2.3 Перебор подмножеств размера $\leq \frac{n}{3}$ . . . . .	7
2.4 Вероятностный алгоритм. Сведение к задаче выполнимости . . . . .	7
2.5 Применение раскраски на практике . . . . .	8
<b>3 Задача коммивояжера</b>	<b>9</b>
3.1 Сложность полного перебора . . . . .	9
3.2 Метод ветвей и границ: . . . . .	9
3.2.1 Отсечка по текущему найденному пути . . . . .	9
3.2.2 Отсечка по весу МОД . . . . .	10
3.3 Локальный поиск . . . . .	10
3.3.1 2-окружение . . . . .	10
3.3.2 Имитация отжига . . . . .	10
3.4 Приближённое решение (2-приближённый алгоритм) . . . . .	11
<b>4 Минимальный разрез в графе (Каргер-Штейн)</b>	<b>12</b>
4.1 Примеры практических задач . . . . .	12
4.2 Алгоритм Каргера . . . . .	12
4.3 Оптимизация Штейна алгоритма Каргера . . . . .	13
<b>5 Динамическое программирование</b>	<b>14</b>
5.1 Примеры вычисления чисел Фибоначчи . . . . .	14
5.2 Редакционное расстояние . . . . .	15
5.3 Задача о порядке перемножения матриц . . . . .	16

<b>6 Поиск подстроки в строке (Кнут-Моррис-Пратт)</b>	<b>17</b>
6.1 Основные определения . . . . .	17
6.2 Задача точного поиска образца в строке . . . . .	17
6.3 Наивный алгоритм и его сложность . . . . .	17
6.4 Алгоритм КМП . . . . .	17
6.5 Наивное построение префикс-функции и его сложность . . . . .	18
6.6 Построение префикс-функции за линейное время . . . . .	19
<b>7 Поиск кратчайших путей. Эвристические алгоритмы</b>	<b>20</b>
7.1 Алгоритм Дейкстры . . . . .	20
7.2 A* . . . . .	21
7.3 ALT . . . . .	23
7.4 Reach . . . . .	24
7.5 REALT . . . . .	24
7.6 Иерархии путей . . . . .	25
7.7 Arc flags . . . . .	26
<b>8 Максимальный поток в графе. Алгоритм проталкивания предпотока</b>	<b>27</b>
8.1 Идея push-relabel алгоритмов . . . . .	27
8.2 Формализмы и инварианты . . . . .	27
8.3 Доказательство корректности . . . . .	29
8.4 Сравнение сложности с Фордом-Фалкерсоном . . . . .	29
<b>9 Поиск набора строк в тексте (Ахо-Корасик)</b>	<b>31</b>
9.1 Задача точного поиска наборов образцов . . . . .	31
9.2 Trie . . . . .	31
9.3 Задача о словаре . . . . .	31
9.4 Алгоритм Ахо-Корасик . . . . .	32
<b>10 Сложность алгоритмов</b>	<b>34</b>
10.1 Виды сложности . . . . .	34
10.2 Понятие вычислительной сложности в зависимости от размера входа . . . . .	34
10.3 Константная сложность на примере vector и unordered_set:	36
10.3.1 Амортизированная . . . . .	36
10.3.2 В среднем . . . . .	37
<b>11 Изоморфизм графов</b>	<b>37</b>
11.1 Задача изоморфизма: . . . . .	38
11.1.1 Точный изоморфизм . . . . .	38
11.1.2 Поиск подграфа в графе . . . . .	38
11.2 Алгоритм Ульмана (переборный с матрицей) . . . . .	39
11.3 Применение изоморфизма . . . . .	40
<b>12 Суффиксные деревья</b>	<b>41</b>

# 1 Поиск с возвращением. Метод Монте-Карло

## 1.1 Идея поиска с возвращением (backtracking)

*Поиск с возвращением* (backtracking) является алгоритмическим подходом к решению задач комбинаторной оптимизации, перебирающим все возможные варианты решения. Основная идея заключается в пошаговом итеративном переборе всех возможных вариантов решения и откате назад (возвращении), если текущий вариант не приводит к правильному или оптимальному решению.

Процесс backtracking можно представить в виде дерева, где каждая ветвь представляет собой очередной шаг в решении задачи, а узлы представляют возможные варианты выбора на каждом шаге. Алгоритм начинает с начального состояния и перебирает все возможные варианты выбора на каждом шаге. Если текущий вариант выбора не приводит к правильному решению, алгоритм откатывается назад и пробует другой вариант.

---

### Algorithm 1 backtracking

---

1. Проверка условия завершения. Если задача решена (например, все ферзи размещены), возвращается результат.
2. Генерация следующего варианта выбора.
3. Проверка допустимости выбранного варианта.
4. Рекурсивный вызов функции для следующего шага.
5. Если рекурсивный вызов вернул правильное решение, возвращается результат.
6. Если рекурсивный вызов не вернул правильного решения, отменяются все изменения и пробуется другой вариант выбора.

Процесс продолжается до тех пор, пока не будут перебраны все варианты или найдено правильное решение.

---

### 1.1.1 Пример на задаче о ферзях

В *Задаче о восьми ферзях* требуется разместить восемь ферзей на шахматной доске таким образом, чтобы они не били друг друга. Алгоритм backtracking может последовательно размещать ферзей на каждом шаге, проверяя, что они не находятся под атакой друг друга. Если ферзь находится под атакой, алгоритм откатывается назад и пробует другой вариант.

### 1.1.2 Подходы метода ветвей и границ по отсеканию вариантов

Подход метода ветвей и границ по отсеканию вариантов включает в себя использование границ для эффективного отсечения неперспективных вариантов решения. Это позволяет сократить количество перебираемых вариантов и ускорить поиск оптимального решения.

---

**Algorithm 2** Метод ветвей и границ

---

1. Создание начальной вершины дерева поиска.
  2. Определение верхней границы для текущей вершины. Верхняя граница представляет наилучшую известную оценку оптимального решения.
  3. Если нижняя граница текущей вершины превышает верхнюю границу, текущая вершина и все ее потомки отсекаются, так как они не могут содержать оптимальное решение.
  4. Если текущая вершина является листом (не имеет потомков), она становится новым оптимальным решением, если ее значение лучше, чем текущее оптимальное решение.
  5. Если текущая вершина не является листом, генерируются ее потомки, каждый из которых представляет возможный вариант решения на следующем шаге.
  6. Для каждого потомка рассчитывается его нижняя граница. Если нижняя граница потомка превышает верхнюю границу, этот потомок и все его потомки отсекаются.
  7. Потомки, которые не были отсечены, рекурсивно проходят через шаги 3-7.
  8. По окончании процесса метода ветвей и границ, оптимальное решение будет содержаться в последней вершине дерева, которая не была отсечена.
- 

## 1.2 Метод Монте-Карло

*Метод Монте-Карло* — это численный метод решения математической задачи при помощи моделирования случайной величины.

Этот метод основывается на законе больших чисел, согласно которому с увеличением числа случайных выборок статистическая оценка будет приближаться к точному значению.

### 1.2.1 Пример на размере дерева

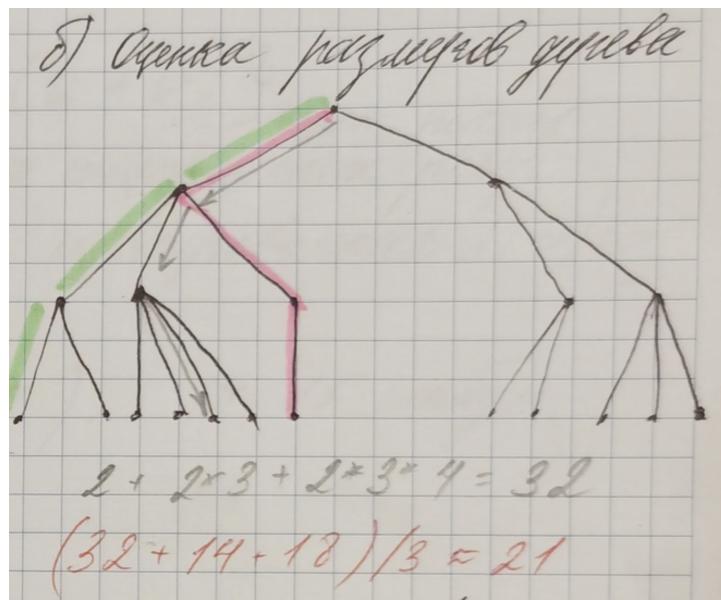
*Пример Калишенко метода Монте-Карло на размере дерева*

Пусть существует некоторого дерева  $G$ .

Проведем метод некоторое число раз.

Каждую итерацию будем выбирать случайный узел, по которому будем идти дальше, после чего будем смотреть, число потомков текущего узла и "отражать данный путь" (считать, что каждого ребра, находящихся на одном уровне одинаковое число потомков), из чего не трудно посчитать общее число узлов. Таким образом, каждую итерацию мы будем получать "симметричное дерево".

Когда доходим до *листа*, сохраняем число узлов в некоторый *контейнер исходов*, и по заверешению алгоритма, делим сумму всех исходов на число итераций



**Рис. 1:** Пример Калишенко метода Монте-Карло на размере дерева (на первой итерации, выделенной разноцветными маркерами, получили 32 узла, после чего по такой же схеме совершают еще 2 итерации, в которых были получены 14 и 18 узлов)

\* Ниже рассматриваются примеры, отличные от примера Калишенко, однако, также имеющих древовидную модель эксперимента.

Предположим, у нас есть дерево решений для задачи принятия решений. Каждый узел дерева представляет собой определенное решение, а ребра дерева - возможные исходы или решения, которые могут быть приняты после каждого решения. Наша цель - оценить вероятность успеха задачи, то есть вероятность достижения конечного узла дерева, где считается, что задача успешно решена.

Чтобы использовать метод Монте-Карло, мы можем случайным образом выбрать большое количество путей через дерево. Каждый путь представляет собой последовательность принятых решений, начиная с корневого узла и двигаясь по ребрам дерева, пока не достигнем конечного узла. Затем мы считаем, сколько из выбранных путей успешно решают задачу, и оцениваем вероятность успеха как отношение успешных путей к общему количеству выбранных путей.

### Пример на крестиках-ноликах

Предположим, у нас есть дерево решений для игры в крестики-нолики. В корневом узле мы выбираем, кто будет ходить первым - крестики или нолики. Затем мы проходим через дерево, выбирая случайные ходы для каждого игрока, пока не достигнем конечного узла, представляющего окончательное состояние игры. Если в конечном узле игра выиграна крестиками, мы считаем путь успешным.

### Еще пример (менее душный)

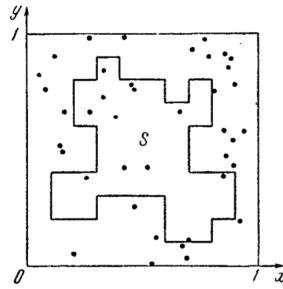
Из корня несколько раз случайным образом выбираем направления, по которым будем идти и после смотрим, является ли лист этого направления успешным (В аналогии с площадью: Попали или Не попали в область). Так делаем  $n$  раз, после чего считаем отношение Успешных Листьев на  $n$ .

## 1.2.2 Пример на площади фигуры

Преположим, что надо вычислить площадь фигуры  $S$ , которая расположена внутри некоторого области (в данном случае - единичного квадрата).

Выберем внутри этой области  $N$  число случайных точек. Точки, попавшие в  $S$  обозначим как  $N^*$ .

Геометрически очевидно, что  $\lim_{N \rightarrow \infty} \frac{N^*}{N} = S$



**Рис. 2:** Пример метода Монте-Карло на площади фигуры

### 1.2.3 Ограничения и условия применимости

Чтобы применять метод Монте-Карло необходимо знать распределение в задачи, а также предел генерации случайных величин.

Также, ошибка вычислений, как правило пропорциональная  $\sqrt{\frac{D}{N}}$ , где  $D$  – некоторая постоянная, а  $N$  – число испытаний.

## 2 Раскраска в 3 цвета

Source: Конспект лекции по раскраскам

Когда говорят о раскраске графов, почти всегда подразумевают под этим раскраску их вершин, то есть присвоение цветовых меток вершинам графа так, чтобы любые две вершины, имеющие общее ребро, имели разные цвета. Другими словами, **необходимо найти правильную раскраску графа для 3-х цветов.**

### 2.1 Алгоритм полного перебора

Пусть дан график и дана его раскраска. Чтобы проверить её корректность, нужно обойти все ребра и проверить, если одинаковые цвета на концах. Значит, сложность проверки - линейная.

Самый простой вариант - генерация всех возможных три-раскрасок и проверка каждой на корректность. Сложность такого алгоритма -  $O(3^n)$ , т.к. это число возможных три-раскрасок графа.

### 2.2 Перебор с учётом выбора только из 2 цветов

Пусть есть некоторая вершина. Мы красим её в первый цвет. Для инцидентных вершин выбор идет не из 3-х цветов, а из 2-х. Сложность такого алгоритма -  $O(2^n)$ .

Другими словами, используется знание, что инцидентные данной вершине нельзя красить в цвет данной.

### 2.3 Перебор подмножеств размера $\leq \frac{n}{3}$

Два соображения:

- Если график уже раскрашен, то вершины разделяются на множества - первого, второго и третьего цвета. Очевидно, что внутри этих множеств не существует ребер. Если есть  $n$  вершин и 3 множества, очевидно, что одно из этих множеств будет иметь мощность  $\leq \frac{n}{3}$  вершин.
- Пусть нам заранее известно одно из этих множеств. Если это так, то все оставшиеся вершины будут краситься гораздо проще - с линейной сложностью.

Значит, нужно найти такое множество. Способов выбрать такое множество:

$$C_n^0 + C_n^1 + C_n^2 + \dots + C_n^{\frac{n}{3}} \leq n * C_n^{\frac{n}{3}} \leq 1.9^n - \text{радиус шара Хэмминга.}$$

Значит, сложность алгоритма -  $O(1.9^n)$

### 2.4 Вероятностный алгоритм. Сведение к задаче выполнимости

Посмотрим все цвета для вершин, в которые можно их покрасить. Можно для каждой вершины случайным образом выкинуть один цвет и красить из оставшихся. В таком случае каждая вершина может быть покрасить в один из двух цветов.

Такую задачу можно свести к задаче выполнимости булевых формул (2SAT).

Сведение такое:

Пусть каждый цвет обозначается переменной  $a_1, a_2, a_3$ . Нужно покрасить все вершины в какие-то цвета. Это значит, что нужно выбрать хотя бы (и только) один из этих цветов. Т.е. для одной вершины верно следующее

$$(a_1 \vee a_2 \vee a_3) \wedge (\overline{a_1} \vee \overline{a_2}) \wedge (\overline{a_1} \vee \overline{a_3}) \wedge (\overline{a_2} \vee \overline{a_3})$$

1-й дизъюнкт - вершину нужно покрасить

2, 3, 4-е дизъюнкты - вершину можно покрасить только в один цвет.

Ещё нужно учесть ограничение три-раскраски, т.е. добавить ограничение на неодинаковость цвета инцидентных вершин.

$$\wedge (\overline{a_1} \vee \overline{b_1}) \wedge (\overline{a_2} \vee \overline{b_2}) \wedge (\overline{a_3} \vee \overline{b_3})$$

Вычеркнув неиспользуемые цвета получаем **задачу 2SAT**, где каждый дизъюнкт содержит не больше чем два литерала. Эту задачу можно решить за полиномиальное время.

Задачу 2SAT возможно свести к графу, и решением будет поиск компонент связности.

Если найдено решение задачи, то граф точно может быть раскрашен, но обратное неверно - возможно, мы вычеркнули нужные цвета для раскраски. Вероятность того, что раскраска выживет после вычеркивания –  $(\frac{2}{3})^n$ . **Это очень мало**

Но если прогнать алгоритм  $(\frac{2}{3})^n$  раз, вероятность ошибки –  $\frac{1}{e}$

Если после этого прогнать ещё 100 раз, то вероятность ошибки –  $\frac{1}{e^{100}}$

Сложность алгоритма –  $O(1.5^n)$

## 2.5 Применение раскраски на практике

1. Задача планирования. Пусть есть список заказов, которые начинаются и заканчиваются в определенное время. Нужно понять, какое минимальное количество ресурсов нужно выделить для решения этих заказов. Сводится так: пересечение заказов по времени - ребро графа. Хроматическое число графа и есть ответ.
2. Есть карта - отображение информации, которое предполагает, что между объектами есть границы (например - политическая карта мира). Задача - определить минимальное число цветов раскрасить карту, чтобы были видны границы между объектами. Задача очевидным образом сводится к раскраске графа.
3. Задача оптимального распределения переменных по регистрам.

### 3 Задача коммивояжера

Source: [Лекция 1 | Алгоритмы для задачи коммивояжёра | Александр Куликов | Лекториум](#)

Задача коммивояжера (Traveling Salesman Problem, TSP) является классической задачей оптимизации комбинаторного типа. В ней требуется найти самый короткий путь, проходящий через все заданные города и возвращающийся в исходный город. (Найти гамильтонов цикл минимальной длины)

#### 3.1 Сложность полного перебора

Сложность полного перебора для задачи коммивояжера растет экспоненциально с увеличением количества городов.

Предположим, у нас есть  $N$  городов. Количество возможных путей для посещения всех городов и возврата в исходный город равно  $(N - 1)!$ , поскольку первый город выбирается из  $N$  возможных, второй - из оставшихся  $N - 1$  возможных и так далее, до последнего города, который автоматически определяется, так как остается только один непосещенный город.

Таким образом, время выполнения полного перебора вариантов для задачи коммивояжера составляет  $O((N - 1)!)$ , что является факториальной сложностью. Это означает, что с увеличением числа городов, время выполнения растет очень быстро и становится практически неосуществимым для больших  $N$ .

#### 3.2 Метод ветвей и границ:

- Метод ветвей и границ (Branch and Bound) - это алгоритм для решения задачи коммивояжера и других комбинаторных задач оптимизации.

- Основные две эвристики:

1. *Ветви*: в каком порядке выбирать еще не посещенные вершины (например, начинать с ближайших)
2. *Границы*: нижняя оценка на длину решений.

- Особенности метода:

1. Находит оптимальное решение
2. Время работы зависит и от эвристик, и от входных данных

##### 3.2.1 Отсечка по текущему найденному пути

Если *текущий вес* гамильтонова цикла больше, чем *лучший вес* (нижняя оценка) - отсекаем ветку решений.

### 3.2.2 Отсечка по весу МОД

Вес оптимального цикла коммивояжера не меньше:

1.  $\frac{1}{2} \sum_{v \in V}$  (два мин. ребра, смежных с  $v$ )
2. веса минимального покрывающего дерева (при выкидывании ребра из оптимального цикла получается покрывающее дерево)

## 3.3 Локальный поиск

---

### Algorithm 3 Локальный поиск

---

```
 $S \leftarrow$  какое-нибудь начальное решение  
while в окрестности  $S$  есть решение  $S^*$  большего веса do  
    Заменить  $S$  на  $S^*$   
end while  
return  $S$ 
```

---

### 3.3.1 2-окружение

Суть: В каждом решении пытаемся стереть *два ребра* и заменить на какие-нибудь другие.

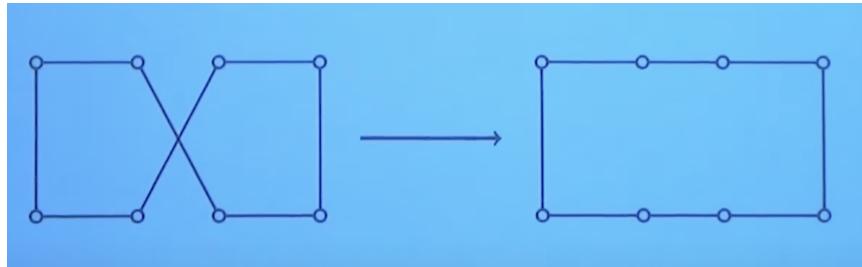


Рис. 3: 2-окружение

### 3.3.2 Имитация отжига

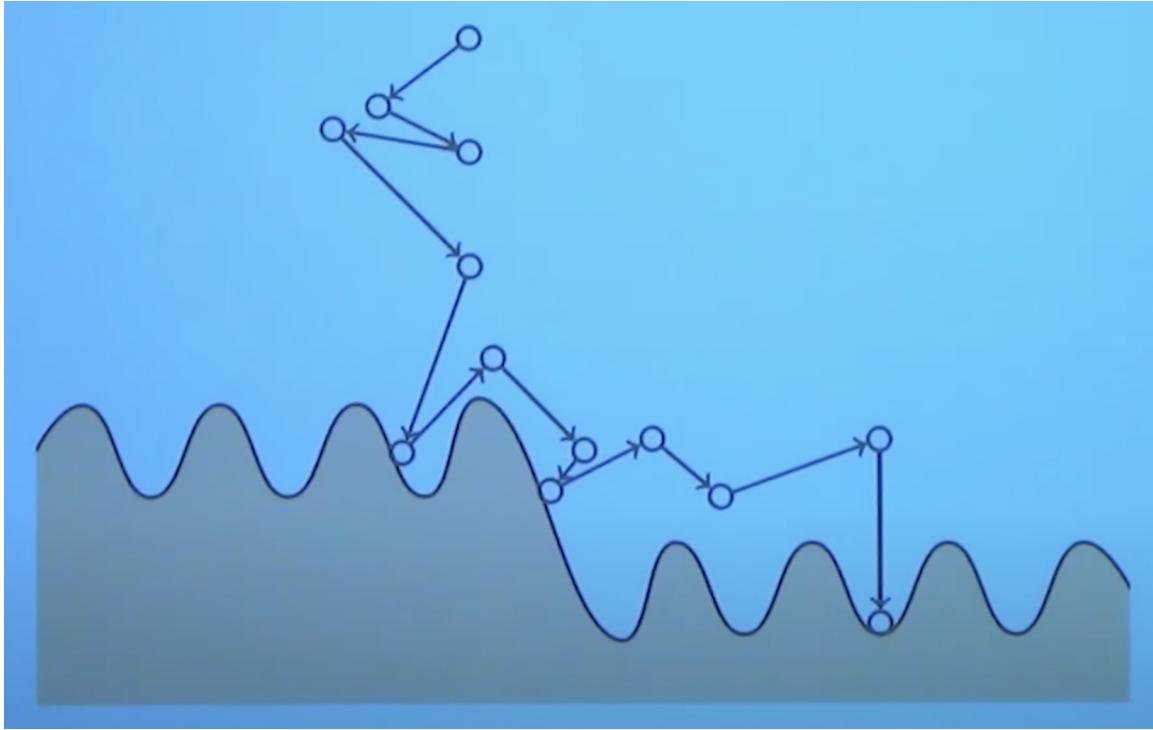
---

### Algorithm 4 Имитация отжига

---

```
 $S \leftarrow$  какое-нибудь начальное решение  
loop  
     $S^* \leftarrow$  случайное решение из окружения  $S$   
     $\Delta \leftarrow cost(S^*) - cost(S)$   
    if  $\Delta < 0$  then  
        заменить  $S$  на  $S^*$   
    else  
        заменить  $S$  на  $S^*$  с вероятностью  $e^{-\frac{\Delta}{T}}$ , где  $T$  - температура или время  
    end if  
end loop
```

---



**Рис. 4:** Метод имитации отжига абстрактно (выпрыгиваем из локального оптимума)

### 3.4 Приближённое решение (2-приближённый алгоритм)

\* **Коммивояжер в метрическом пространстве:**

Частный случай для неориентированных графов, веса ребер которых удовлетворяют неравенству треугольника:

$$w(i,j) \leq w(i,k) + w(k,j)$$

---

#### Algorithm 5 2-приближенный алгоритм

---

1. Построить минимальное покрывающее дерево (МОД)  $T$
  2. Продублировать каждое ребро дерева  $T$
  3. В полученном графе найти эйлеров цикл
  4. Выкинуть из этого цикла все повторения вершин
- 

*Доказательство.*

1. Пусть  $W_T$  – вес МОД, а  $W_{opt}$  – вес оптимального гамильтонова цикла.
2.  $W_T \leq W_{opt}$ , поскольку при выкидывании ребра из гамильтонова цикла получается оствное дерево.
3. Каждое ребро построенного гамильтонова цикла заменяет какой-то путь эйлерова цикла, длина которого по неравенству треугольника не менее длины этого ребра
4. Значит, длина найденного пути не превосходит  $2W_T$ , а следовательно, и  $2W_{opt}$

□

## 4 Минимальный разрез в графе (Каргер-Штейн)

Source:

1. Конспект ИТМО
2. \* [Лекция 2 | Вероятностные алгоритмы | Иван Михайлин | Лекториум](#)

### 4.1 Примеры практических задач

### 4.2 Алгоритм Каргера

---

#### Определение 4.1.

---

**Стягиванием** ребра  $uv$  назовем последовательность действий:

1. Добавляем новую вершину  $w$
2. Для всех ребер  $xu$  и  $xv$  (где  $x \in V$ ,  $x \neq v$ ,  $x \neq u$ ) добавляем новые ребра  $xw$ . При этом, если получаются кратные ребра — оставляем их.
3. Удаляем вершины  $u$  и  $v$  и все инцидентные им ребра.

---

#### Algorithm 6 Алгоритм Каргера

---

```
minCut = ∞
G = (V, E) // Некоторый граф с V вершинами и E ребрами
i = 0
for i ≤ n2 ln n do
    G* = G
    while |V*| > 2 do
        Стягиваем случайное ребро из G*
    end while
    if |E*| < minCut then
        minCut = |E*|
    end if
end for
```

---

Т.к. внутренний цикл ( $while |V^*| > 2$ ) выполняется  $n^2 \ln n$  раз, то вероятность нахождения неверного ответа  $\leq \frac{1}{n^2}$ . Даже при небольших значениях  $n$  алгоритм практически гарантированно выдает правильный ответ.

- Оценка времени работы алгоритма  
Время работы внутреннего цикла —  $O(n^2)$ , а внешнего —  $O(n^2 \log(n))$ , из чего следует, что время работы всего алгоритма —  $O(n^4 \log(n))$ .

### 4.3 Оптимизация Штейна алгоритма Каргера

- × Будем называть внутренний цикл *Алгоритма Каргера* функцией *GetCut*, а внешний – функцией *MinCut*.

- **Оптимизация заключается в следующем:**

Заметим, что вероятность стягивания вершины, принадлежащей минимальному разрезу, в начале выполнения функции *GetCut* довольно мала, в то время, как вероятность стянуть ребро, которое не следует стягивать, ближе к концу работы функции существенно возрастает. Тогда будем использовать следующую рекурсивную версию алгоритма:

1. Запускаем функцию *GetCut* и стягиваем ребра до тех пор, пока не останется  $\frac{n}{\sqrt{2}}$  вершин.
2. Запускаем независимо эту же функцию для получившегося графа дважды и возвращаем минимальный из ответов.

Такая модификация алгоритма выдает правильный ответ с точностью, не менее  $\frac{1}{\log(n)}$ . Время работы функции *GetCut* вычисляется рекурсивной функцией:

$$T(n) = O(n^2) + 2 * T\left(\frac{n}{\sqrt{2}}\right) = O(n^2 \log(n))$$

Это медленнее, чем оригинальный алгоритм, однако вероятность нахождения разреза минимального веса экспоненциально выше. Достаточно запустить алгоритм  $c \log^2(n)$  раз, где  $c$  – некоторая константа. Действительно, рассчитаем вероятность неправильного ответа также, как раньше:

$$\left(1 - \frac{1}{\log(n)^{c \log^2(n)}}\right) \leq e^{c \log^2(n) - \frac{1}{\log(n)}} = \frac{1}{n^c}.$$

Итоговое время работы –  $O(n^2 \log(n)) \cdot c \log^2(n) = O(n^2 \log^3(n))$

## 5 Динамическое программирование

Динамическое программирование (DP) - это метод оптимизации, который позволяет решать задачи, разбивая их на более мелкие подзадачи и сохраняя результаты этих подзадач для последующего использования.

### 5.1 Примеры вычисления чисел Фибоначчи

Одна из классических задач, которую можно решить с использованием динамического программирования, - это вычисление чисел Фибоначчи. Рассмотрим решение в лоб, а именно с помощью использования рекурсивной функции.

Листинг 1: Первый пример вычисления чисел Фибоначчи рекурсией

```
1 def fibonacci(n):
2     if n==1:
3         return 1
4     else:
5         return fibonacci(n-1)+fibonacci(n-2)
6
```

В чем недостаток данного метода. В том, что мы несколько раз будем пересчитывать одно и тоже значение, которое нами уже посчитано. Для этого и существует динамическое программирование, деление задачи на множество маленьких подзадач, единожды последовательно решив которые, мы получим результат.

Листинг 2: Второй пример вычисления чисел Фибоначчи Динамическим программированием

```
1 def fibonacci(n):
2     fib = [0, 1]
3
4     for i in range(2, n + 1):
5         fib.append(fib[i - 1] + fib[i - 2])
6
7     return fib[n]
8
```

Главным недостатком данного подхода может показаться не рациональный расход памяти, однако этого можно заметить, что для вычисления следующего значения чисел фибоначчи, нам нужно 2 предыдущих, используем этот факт и получим следующий код.

Листинг 3: Примеры вычисления чисел Фибоначчи Динамическим программированием-2

```
1 def fibonacci(n):
2     prev_fib=0
3     this_fib=1
4     next_fib=0
5     for i in range(2, n + 1):
6         next_fib=this_fib+prev_fib
7         prev_fib=this_fib
8         this_fib=next_fib
9
10    return next_fib
11
```

Такое решение имеет линейную сложность.

## 5.2 Редакционное расстояние

### Определение 5.1.

*Расстояние Левенштейна (редакционное расстояние, дистанция редактирования)* — метрика, измеряющая по модулю разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (а именно вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую.

$$\begin{array}{c} \text{exponential} \\ \times \times \dots \uparrow\downarrow . + \uparrow \\ \text{potential} \end{array}$$

Здесь и далее будем рассматривать  $\times$ -удаление,  $\uparrow\downarrow$ -замена,  $+$ -добавление.

Решим эту задачу с помощью динамического программирования. Введем функцию  $E(i,j)$ , где  $i$  — первые  $i$  и  $j$  длины префиксов первой и второй строки. Тогда

$$E(i,j) = \min \begin{cases} E(i-1,j) + 1 & \text{вставка} \\ E(i,j-1) + 1 & \text{удаление} \\ E(i-1,j-1) + \text{diff}(i,j) & \text{замена} \end{cases}$$
$$\text{diff}(i,j) = \begin{cases} 1 & i \neq j \\ 0 & i = j \end{cases}$$

Теперь если учесть, что  $E(i,0) = i$ , а  $E(j,0) = j$ , то можно написать псевдокод.

Листинг 4: Пример нахождения минимального количества операций преобразования одной строки в другую

```
1 def diff(i,j):
2     if i==j:
3         return 0
4     else:
5         return 1
6 def count_operation_string(old_string,to_string,len_old_string,len_to_string):
7     E = [[0] * len_old_string for i in range(len_to_string)] //
8     for i in range(len_to_string): //
9         E[i][0]=i
10    for j in range(len_old_string):
11        E[0][j]=j
12    for i in range(len_to_string): //
13        for j in range(len_old_string):
14            E[i][j]=min(E[i-1][j]+1,E[i][j-1]+1,E[i-1][j-1]+diff(i,j))
15    return E[len_old_string-1][len_to_string-1]
```

### 5.3 Задача о порядке перемножения матриц

Сложность перемножения двух матриц  $O(n^3)$ . Рассмотрим пример перемножения трех матриц  $A(10 \times 100), B(100 \times 5), C(5 \times 50)$ .

Количество итераций при:

$$(A * B) * C = 10 * 100 * 5 + 10 * 5 * 50 = 7500 \text{ итераций}$$
$$A * (B * C) = 100 * 5 * 50 + 100 * 10 * 100 = 75000 \text{ итераций}$$

Очевидно, что при изменении порядка перемножения матриц результат не изменится, а количество затраченных итераций на перемножения может быть разным. Возникает вопрос о таком перемножении матриц, при котором количество операций будет минимально, так как перемножение матриц сам по себе сложный процесс, имеющий кубическую сложность.

Попробуем решить задачу с помощью динамического программирования. Введем функцию

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} (m[i,k] + m[k+1,j] + p_{i-1} * p_k * p_j) & i < j \end{cases}$$

Будем обозначать произведение матриц  $A_i * A_{i+1} * \dots * A_{j-1} * A_j$  как  $A_{i..j}$ . Данная функция обозначает минимальное количество операций скалярного умножения

Тогда  $m[i,j]$  - минимальное количество скалярных умножений матриц  $A_{i..j}$

В этой формуле  $m[i,k] + m[k+1,j]$  уже перемноженные на предыдущих шагах матрицы  $A_{i..k}, A_{k+1..j}$   
 $p_{i-1} * p_k * p_j$  - это количество строк  $i-1$  матрицы \* количество столбцов  $k$  матрицы \* количество столбцов  $j$  матрицы.

$p$  - это строка размерностей наших матриц. Для примера выше  $p=(10,100,5)$ .

Как нетрудно заметить, 1 элемент этого вектора - количество строк первой матрицы, последний элемент этого вектора - количество столбцов последней матрицы, а остальные элементы - количество строк правой и столбцов левой матрицы.

# 6 Поиск подстроки в строке (Кнут-Моррис-Пратт)

Source:

1. Алгоритм Кнута-Морриса-Пратта (ИТМО конспект)
2. Префикс-функция (ИТМО конспект)
3. Лекция 5 | Алгоритмы в биоинформатике | Николай Вяххи | CSC | Лекториум

## 6.1 Основные определения

### Определение 6.1.

**Префикс функция** – длина наибольшего префикса строки, которая не совпадает с этой строкой и одновременно является ее суффиксом:

$$\begin{aligned} S &= abacaba \\ P(S) &= 0010123 \end{aligned}$$

## 6.2 Задача точного поиска образца в строке

Дана некоторая строка  $pattern$  длины  $n$  и строка  $text$  длины  $m$ . Найти все вхождения  $pattern$ 'а в  $text$ .

## 6.3 Наивный алгоритм и его сложность

Полный перебор всех элементов  $text$ , в котором сравниваются  $pattern$  и  $text[i : n]$ .

**Сложность** –  $O(n * m)$

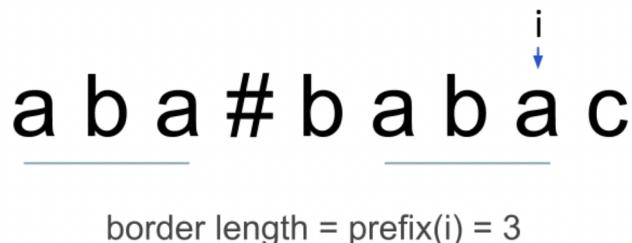
## 6.4 Алгоритм КМП

Дана строка текста  $T$  и шаблон  $P$ .

Построим строку  $S = P\#T$ , где  $\#$  – любой символ, которого нету ни в  $T$ , ни в  $P$ . Посчитаем по ней значение префикс функции  $p$ . Из-за символа-разделителя выполняется  $\forall i : p[i] \leq |P|$ .

Заметим, что по определению префикс-функции при  $i > |P|$  и  $p[i] = |P|$  подстроки длины  $P$ , начинающиеся с позиций  $0$  и  $i - |P| + 1$ , совпадают.

Соберем все такие позиции  $i - |P| + 1$  строки  $S$ , вычтем из каждой позиции  $|P| + 1$ , это и будет ответ. Другими словами, если в какой-то позиции  $i$  выполняется условие  $p[i] = |P|$ , то в этой позиции начинается очередное вхождение образца в цепочку.



---

**Algorithm 7** КМП

---

```
int[] kmp(string P, string T):
    int pl = P.length
    int tl = T.length
    int[] answer
    int[] p = prefixFunction(P + "#" + T)
    int count = 0
    for i = 0 .. tl - 1
        if p[pl + i + 1] == pl
            answer[count++] = i - pl
    return answer
```

---

**Время работы**

Префикс-функция от строки  $S$  строится за  $O(S) = O(P + T)$ . Проход цикла по строке  $S$  содержит  $O(T)$  итераций. Итого, время работы алгоритма оценивается как  $O(P + T)$

## 6.5 Наивное построение префикс-функции и его сложность

Наивный алгоритм вычисляет префикс-функцию непосредственно по определению, сравнивая префиксы и суффиксы строк. Обозначим длину строки за  $n$ . Будем считать, что префикс-функция хранится в массиве  $p$ .

---

**Algorithm 8** Наивное построение префикс-функции

---

```
int[] prefixFunction(string s):
    int[] p = int[s.length]
    fill(p, 0)
    for i = 0 to s.length - 1
        for k = 0 to i - 1
            if s[0..k] == s[i - k..i]
                p[i] = k
    return p
```

---

**Пример**

Рассмотрим строку  $abcabcd$ , для которой значение префикс-функции равно  $[0, 0, 0, 1, 2, 3, 0]$

Шаг	Строка	Значение функции
1	a	0
2	ab	0
3	abc	0
4	abca	1
5	abcab	2
6	abcabc	3
7	abcabcd	0

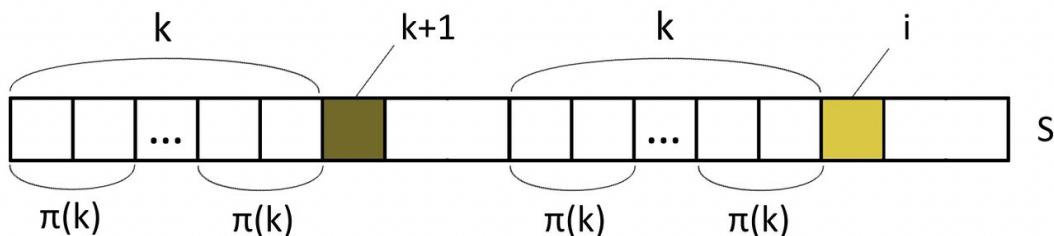
**Время работы**

Всего  $O(n^2)$  итераций цикла, на каждой из которых происходит сравнение строк за  $O(n)$ , что дает в итоге  $O(n^3)$ .

## 6.6 Построение префикс-функции за линейное время

Вносятся несколько важных замечаний:

- \* Заметим, что  $p[i + 1] \leq p[i] + 1$ . Чтобы показать это, рассмотрим суффикс, оканчивающийся на позиции  $i + 1$  и имеющий длину  $p[i + 1]$ , удалив из него последний символ, мы получим суффикс, оканчивающийся на позиции  $i$  и имеющий длину  $p[i + 1] - 1$ , следовательно неравенство  $p[i + 1] > p[i] + 1$  неверно.
- \* Избавимся от явных сравнений строк. Пусть мы вычислили  $p[i]$ , тогда, если  $s[i + 1] = s[p[i]]$ , то  $p[i + 1] = p[i] + 1$ . Если окажется, что  $s[i + 1] \neq s[p[i]]$ , то нужно попытаться попробовать подстроку меньшей длины. Хотелось бы сразу перейти к такому бордеру наибольшей длины, для этого подберем такое  $k$ , что  $k = p[i] - 1$ . Делаем это следующим образом. За исходное  $k$  необходимо взять  $p[i - 1]$ , что следует из первого пункта. В случае, когда символы  $s[k]$  и  $s[i]$  не совпадают,  $p[k - 1]$  – следующее потенциальное наибольшее значение  $k$ , что видно из рисунка. Последнее утверждение верно, пока  $k > 0$ , что позволит всегда найти его следующее значение. Если  $k = 0$ , то  $p[i] = 1$  при  $s[i] = s[1]$ , иначе  $p[i] = 0$ .




---

**Algorithm 9** Эффективный алгоритм вычисления префикс-функции

---

```

int[] prefixFunction(string s):
    int[] p = int[s.length]
    p[0] = 0
    for i = 1 to s.length - 1
        int k = p[i - 1]
        while k > 0 and s[i] != s[k]
            k = p[k - 1]
        if s[i] == s[k]
            k++
        p[i] = k
    return p
  
```

---

### Время работы

Время работы алгоритма составит  $O(n)$ .

Для доказательства этого нужно заметить, что итоговое количество итераций цикла *while* определяет асимптотику алгоритма. Теперь стоит отметить, что  $k$  увеличивается на каждом шаге не более чем на единицу, значит максимально возможное значение  $k = n - 1$ . Поскольку внутри цикла *while* значение  $k$  лишь уменьшается, получается, что  $k$  не может суммарно уменьшиться больше, чем  $n - 1$  раз. Значит цикл *while* в итоге выполнится не более  $n$  раз, что дает итоговую оценку времени алгоритма  $O(n)$ .

# 7 Поиск кратчайших путей. Эвристические алгоритмы

Source:

1. Алгоритмы Дейкстра и Флойда-Уоршелла
2. Алгоритм A\* (ИТМО конспект)
3. Эвристики для поиска кратчайших путей (ИТМО конспект)
4. Для ATL, Reach, REALT, Иерархии путей, Arc flags использовал gpt, тк в интернете мало инфы, да и мне как-то лень сегодня
5. \* Полезная статья на хабре про методы поиска мин пути в графе

## 7.1 Алгоритм Дейкстры

*Алгоритм Дейкстры* находит кратчайший путь от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса.

### Идея алгоритма

1. Каждый раз, когда мы хотим посетить новый узел, мы выберем узел с наименьшим известным расстоянием.
2. Как только мы переместились в узел, мы проверяем каждый из соседних узлов. Мы вычисляем расстояние от соседних узлов до корневых узлов, суммируя стоимость ребер, которые ведут к этому новому узлу.
3. Если расстояние до узла меньше известного расстояния, мы обновим самое короткое расстояние.

На каждом шаге существует множество уже обработанных вершин и еще не обработанных.

### Сложность алгоритма

Худший случай - каждый путь содержит в себе все остальные ( $v$  вершин). Если каждый такой путь будет хранится к каждой вершине, память будет  $v^2$ . Для оптимизации в каждой вершине хранится не весь путь, а только предыдущую вершину, из которой можно попасть в текущую.

Общая сложность алгоритма:

1.  $O(v^2)$  – при работе на массиве;
2.  $O(\ln(v))$  – при работе на куче.

Если количество ребер небольшое, выгоднее использовать реализацию на куче, если же ребер намного больше, чем вершин, лучше использовать работу на массиве.

### Применение алгоритма

Одно из применений алгоритма – маршрутизация. Например, алгоритм OSPF (Open Shortest Path First). Каждый маршрутизатор строит некоторый график и использует алгоритм Дейкстры в чистом виде.

---

**Algorithm 10** Алгоритм Дейкстры

---

```
for  $v \in V$  do
     $Dist[v] = \infty$ 
     $Prev[v] = \emptyset$ 
end for
 $Dist[v' \in V] = 0$  // Стартовая вершина
// Первый шаг:
 $H \leftarrow MakeQueue()$  // формирование очереди с приоритетами для вершины  $v'$ . Все инцидентные
вершины попадают сюда
while  $H \neq \emptyset$  do
     $v \leftarrow \min(H)$  // из очереди с приоритетами выбирается минимальный  $Dist[v]$ 
    for  $vu \in E$  // для каждого инцидентного ребра do
        if ( $Dist[u] > Dist[v] + w(v,u)$  // если расстояние до вершины больше, чем то, по которому
мы проходим - условие релаксации then
             $Dist[u] \leftarrow Dist[v] + w(v,u)$ 
             $Prev[u] \leftarrow UpdatePriorities(H)$ 
        end if
    end for
end while
```

---

## 7.2 A\*

**Алгоритм поиска A\*** — относится к эвристическим алгоритмам поиска по первому лучшему совпадению на графе с положительными весами ребер, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной). Алгоритм использует вспомогательную функцию (эвристика), чтобы направлять поиск решения и сокращать его длительность. Алгоритм полный в том смысле, что всегда находит оптимальное решение, если он существует.

### Описание алгоритма

В процессе работы алгоритма для вершин рассчитывается функция  $f(v) = g(v) + h(v)$ , где

1.  $g(v)$  — наименьшая стоимость пути в  $v$  из стартовой вершины,
2.  $h(v)$  — эвристическое приближение стоимости пути от  $v$  до конечной цели.

Фактически, функция  $f(v)$  — длина пути до цели, которая складывается из пройденного расстояния  $g(v)$  и оставшегося расстояния  $h(v)$ . Исходя из этого, чем меньше значение  $f(v)$ , тем раньше мы откроем вершину  $v$ , так как через неё мы предположительно достигнем расстояние до цели быстрее всего. Открытые алгоритмом вершины можно хранить в очереди с приоритетом по значению  $f(v)$ . A\* действует подобно алгоритму Дейкстры и просматривает среди всех маршрутов ведущих к цели сначала те, которые благодаря имеющейся информации (эвристическая функция) в данный момент являются наилучшими.

---

**Определение 7.1.**

---

Говорят, что эвристическая оценка  $h(v)$  **допустима**, если для любой вершины  $v$  значение  $h(v)$  меньше или равно весу кратчайшего пути от  $v$  до цели.

Допустимая оценка является оптимистичной, потому что она предполагает, что стоимость решения меньше, чем оно есть на самом деле. Второе, более сильное условие — функция  $h(v)$  должна быть **монотонной** (т.е для любой вершины  $v_1$  и ее потомка  $v_2$  разность  $h(v_1)$  и  $h(v_2)$  не превышает фактического веса ребра  $c(v_1, v_2)$  от  $v_1$  и  $v_2$ , а эвристическая оценка целевого состояния равна нулю).

## Реализация

1.  $Q$  — множество вершин, которые требуется рассмотреть,
2.  $U$  — множество рассмотренных вершин,
3.  $f[x]$  — значение эвристической функции "расстояние + стоимость" для вершины  $x$ ,
4.  $g[x]$  — стоимость пути от начальной вершины до  $x$ ,
5.  $h(x)$  — эвристическая оценка расстояния от вершины  $x$  до конечной вершины.

На каждом этапе работы алгоритма из множества  $Q$  выбирается вершина с наименьшим значением эвристической функции и просматриваются её соседи. Для каждого из соседей обновляется расстояние, значение эвристической функции и он добавляется в множество  $Q$ .

---

### Algorithm 11 A\*

---

```

 $U = \emptyset$ 
 $Q = \emptyset$ 
 $Q.push(start)$ 
 $g[start] = 0$ 
 $f[start] = g[start] + h(start)$ 
while  $Q.size() \neq 0$  do
    current = вершина из  $Q$  с минимальным значением  $f$ 
    if current = goal then
        return Нашли путь до нужной вершины
    end if
     $Q.remove(current)$ 
     $U.push(current)$ 
    for  $v$  : смежных с current вершины do
         $tentativeScore = g[current] + d(current, v)$  //  $d(current, v)$  — стоимость пути между current и  $v$ 
        if  $v \in U$  and  $tentativeScore \geq g[v]$  then
            continue
        end if
        if  $v \notin U$  or  $tentativeScore < g[v]$  then
             $parent[v] = current$ 
             $g[v] = tentativeScore$ 
             $f[v] = g[v] + h(v)$ 
            if  $v \notin Q$  then
                 $Q.push(v)$ 
            end if
        end if
    end for
end while

```

---

## Примеры эвристик:

- Если мы можем перемещаться в четырех направлениях, то в качестве эвристики стоит выбрать **манхэттенское расстояние**

$$h(v) = |v.x - goal.x| + |v.y - goal.y|$$

- Расстояние Чебышева** применяется, когда к четырем направлениям добавляются диагонали:

$$h(v) = \max(|v.x - goal.x|, |v.y - goal.y|)$$

- Если передвижение не ограничено сеткой, то можно использовать **евклидово расстояние** по прямой:

$$h(v) = \sqrt{(v.x - goal.x)^2 + (v.y - goal.y)^2}$$

## 7.3 ALT

**Алгоритм ATL** ( $A^*$  + Landmarks + Triangle inequality) - это комбинированный алгоритм поиска кратчайшего пути, который объединяет идеи из трех различных подходов: алгоритма  $A^*$  (A-star), использования ориентиров (landmarks) и неравенства треугольника.

*Ориентиры* – это ключевые вершины графа, выбранные для вычисления приближенных расстояний до остальных вершин.

Вот основные шаги алгоритма ATL:

- Подготовка данных:** Перед началом поиска кратчайшего пути производится предварительная обработка графа. Это может включать в себя определение ориентиров (landmarks) – ключевых вершин, для которых будут вычислены расстояния до всех остальных вершин.
- Вычисление расстояний до ориентиров:** Используя выбранные ориентиры, вычисляются приближенные расстояния от каждой вершины графа до каждого ориентира. Это можно сделать с помощью алгоритма поиска кратчайшего пути, такого как алгоритм Дейкстры.
- Вычисление нижних оценок:** На основе полученных приближенных расстояний строятся нижние оценки расстояний от каждой вершины до целевой вершины. Это выполняется путем комбинирования приближенных расстояний до ориентиров с использованием неравенства треугольника, чтобы получить наиболее оптимистичные оценки.

Будем использовать неравенство треугольника для нижних оценок пути. Пусть  $A$  – один из ориентиров, тогда:

- $dist(v, w) \geq dist(A, w) - dist(A, v)$
- $dist(v, w) \geq dist(v, A) - dist(w, A)$
- $dist(v, w) \geq \max\{dist(A, w) - dist(A, v), dist(v, A) - dist(w, A)\}$

- Использование  $A^*$ :** Запускается алгоритм  $A^*$  для поиска кратчайшего пути от исходной вершины к целевой. При этом для выбора следующей вершины используются нижние оценки, вычисленные на предыдущем шаге.

5. **Повторение процесса:** Если алгоритм A\* не достиг целевой вершины, можно повторить шаги 2-4, чтобы улучшить оценки и продолжить поиск пути.

Алгоритм ATL пытается улучшить производительность алгоритма A\* путем использования ориентиров и неравенства треугольника для получения более точных и оптимистичных оценок расстояний. Это позволяет ускорить поиск пути, особенно в больших графах или в случаях с длинными путями.

## 7.4 Reach

**Алгоритм Reach** (или Reachability Algorithm) - это алгоритм, используемый для определения достижимости между вершинами в ориентированном графе. Его основная цель - определить, можно ли достичь одну вершину из другой по заданному направлению ребер.

**Основные шаги алгоритма Reach:**

1. **Инициализация:** Устанавливаются начальные значения для каждой вершины графа. Обычно все вершины помечаются как недостижимые.
2. **Установка начальных вершин:** Выбирается начальная вершина или группа вершин, для которых будем искать достижимость. Это может быть одна вершина или набор вершин, в зависимости от требуемых условий.
3. **Обход графа:** Начиная с выбранных начальных вершин, выполняется обход графа с помощью поиска в глубину (DFS) или поиска в ширину (BFS). При обходе графа помечаются достижимые вершины.
4. **Проверка достижимости:** После завершения обхода графа проверяется, достижимы ли целевые вершины из выбранных начальных вершин. Если достижимость обнаружена, соответствующие вершины помечаются как достижимые.
5. **Завершение:** Алгоритм Reach завершается после проверки достижимости целевых вершин и обновления их меток.

Алгоритм Reach широко используется для определения достижимости и проверки связности в графах. Он может быть применен в различных областях, включая анализ социальных сетей, оптимизацию маршрутов и выявление взаимосвязей в данных.

## 7.5 REALT

Основная идея алгоритма REALT заключается в поиске кратчайшего пути от целевой вершины к исходной вершине, в то время как большинство классических алгоритмов поиска пути ищут путь от исходной вершины к целевой. Алгоритм REALT работает в обратном направлении, начиная с целевой вершины и двигаясь к исходной.

**Основные шаги алгоритма REALT:**

1. **Инициализация:** Устанавливаем расстояние до целевой вершины равным нулю, а расстояние до остальных вершин - бесконечностью.

2. **Установка приоритетов:** Для каждой вершины графа вычисляем приоритет, который определяется как минимальное расстояние от данной вершины до целевой вершины.
3. **Итерации:** В каждой итерации выбираем вершину с наибольшим приоритетом и обновляем расстояния до ее соседей. Если новое расстояние меньше текущего, обновляем его.
4. **Завершение:** Повторяем итерации до тех пор, пока не достигнем исходной вершины или пока все вершины не будут обработаны.
5. **Восстановление пути:** Если мы достигли исходной вершины, восстанавливаем кратчайший путь, используя информацию о предыдущих вершинах.

Алгоритм REALT имеет преимущества в скорости работы и потребляемых ресурсах для больших графов. Однако он может быть ограничен использованием только взвешенных ориентированных графов и не является универсальным для всех типов графов.

## 7.6 Иерархии путей

**Алгоритм Иерархии путей** (Pathfinding Hierarchy Algorithm) – это алгоритм поиска кратчайшего пути в графе с использованием предварительной обработки иерархической структуры.

### Основная идея

Основная идея алгоритма Иерархии путей заключается в разделении графа на несколько уровней или слоев, где каждый уровень представляет собой абстракцию вершин более низкого уровня. Таким образом, граф становится иерархическим, где вершины на более высоких уровнях объединяют группы вершин на более низких уровнях.

### Основные шаги:

1. **Построение иерархии:** Исходный график разбивается на уровни или слои. Это может быть достигнуто различными способами, например, с помощью кластеризации вершин или использования методов снижения размерности.
2. **Предварительная обработка:** Для каждого уровня графа вычисляются кратчайшие пути между всеми парами вершин на этом уровне. Это можно сделать с помощью любого эффективного алгоритма поиска пути, например, алгоритма Дейкстры или алгоритма A\*.
3. **Поиск пути:** Для поиска кратчайшего пути между исходной и целевой вершинами используется иерархический подход. Сначала находятся ближайшие вершины на высоком уровне, затем на следующем более низком уровне, и так далее. Затем выполняется поиск пути на каждом уровне, используя предварительно вычисленные кратчайшие пути.
4. **Комбинирование путей:** Когда кратчайшие пути найдены на каждом уровне, они комбинируются для получения общего кратчайшего пути между исходной и целевой вершинами. Это может быть достигнуто с помощью слияния путей на разных уровнях и устранения дублирования вершин.

Алгоритм Иерархии путей позволяет ускорить поиск кратчайшего пути за счет использования предварительной обработки и разделения графа на более простые уровни. Он особенно полезен для

## 7.7 Arc flags

**Алгоритм Arc-Flag** – это эффективный алгоритм для поиска минимального пути в графе, основанный на использовании специальных флагов (arc flags) на ребрах графа. Он позволяет сократить количество обрабатываемых ребер и ускорить поиск минимального пути.

**Основные шаги:**

1. **Предварительная обработка:** Для каждого ребра в графе вычисляются и сохраняются дополнительные данные, такие как вес ребра или другая информация, необходимая для определения минимального пути.
2. **Установка флагов на ребрах:** На некоторых ребрах графа устанавливаются флаги, которые указывают на определенные свойства ребер. Флаги могут отражать различные характеристики ребер, например, наличие определенных ограничений или особенностей.
3. **Поиск пути:** Алгоритм выполняет поиск минимального пути от одной вершины графа к другой, используя флаги и предварительно вычисленные данные о ребрах. При поиске пути алгоритм учитывает только те ребра, на которых установлены соответствующие флаги.
4. **Обновление флагов:** По мере продвижения вдоль пути алгоритм обновляет флаги на ребрах в соответствии с посещенными вершинами и другой информацией о пути. Это может включать установку или снятие флагов в зависимости от изменений в пути или посещенных вершинах.
5. **Оптимизация и повторный поиск:** После завершения первого поиска пути алгоритм может проанализировать результаты и выполнить дополнительные оптимизации, например, сократить количество флагов или переиспользовать предварительно вычисленные данные. Затем может быть выполнен повторный поиск пути для получения окончательного минимального пути.

Алгоритм Arc-Flag позволяет существенно сократить количество ребер, которые требуется рассмотреть при поиске минимального пути, путем использования флагов и выборочной обработки только некоторых ребер. Это делает его эффективным в случаях, когда графы очень большие или имеют большое количество ребер, и позволяет достичь более быстрых временных характеристик алгоритма поиска минимального пути.

# 8 Максимальный поток в графе. Алгоритм проталкивания предпотока

Source:

1. Метод проталкивания предпотока (ИТМО конспект)
2. Запись лекции "Алгоритм Гольдберга"
3. \* Лекция 1 | Избранные главы теории потоков | Максим Бабенко | Лекториум
4. \* Лекция 4 | Избранные главы теории потоков | Максим Бабенко | Лекториум
5. \* Лекция 5 | Избранные главы теории потоков | Максим Бабенко | Лекториум

## 8.1 Идея push-relabel алгоритмов

Для понимания идеи алгоритма представим, что наша сеть — система из резервуаров, находящихся на определенной высоте, и соединяющих их трубы с заданными пропускными способностями, соответствующих вершинам и рёбрам в исходной сети. Сам алгоритм можно представить как процесс поочередного "переливания" жидкости (операция проталкивания) из одного резервуара в другие, находящиеся на меньшей высоте, до тех пор пока не перестанет существовать резервуар, соответствующий переполненной вершине. Может случиться ситуация, что все трубы, выходящие из переполненной вершины  $u$ , ведут к вершинам, находящимся на такой же высоте что и  $u$  или выше её. В таком случае поднимем резервуар (операция подъёма), соответствующий данной вершине, таким образом, чтобы его высота стала на единицу больше, чем высота самого низкого из смежных резервуаров. После подъёма будет существовать по крайней мере одна труба, по которой можно пропустить жидкость.

В итоге, у нас не останется ни одной переполненной вершины, та часть потока, которая могла пройти к стоку, окажется там, остальная же вернется в исток. В такой ситуации предпоток превращается в обычный поток, так как для каждой вершины выполняется условие сохранения потока. Как будет показано далее, предпоток становится не только обычным, но и максимальным потоком.

## 8.2 Формализмы и инварианты

---

### Определение 8.1.

**Предпотоком** (Preflow) будем называть функцию  $f : V \times V \rightarrow \mathbb{R}$ , удовлетворяющую следующим свойствам:

1.  $f(u,v) = -f(v,u)$  (антисимметричность)
2.  $f(u,v) \leq c(u,v)$  (ограничение пропускной способностью)
3.  $\forall u \in V \setminus \{s,t\}, \sum_{v \in V} f(v,u) \geq 0$  (ослабленное условие сохранения потока)

\* Как можно заметить, по своим свойствам предпоток очень похож на поток и отличается лишь тем, что для него не выполняется закон сохранения потока.

## Определение 8.2.

**Избыточным потоком**, входящим в вершину  $u$ , назовем величину  $e(u) = \sum_{v \in V} f(v,u)$ .

Тогда вершина  $u \in V \setminus \{s,t\}$  будет называться **переполненной**, если  $e(u) > 0$ .

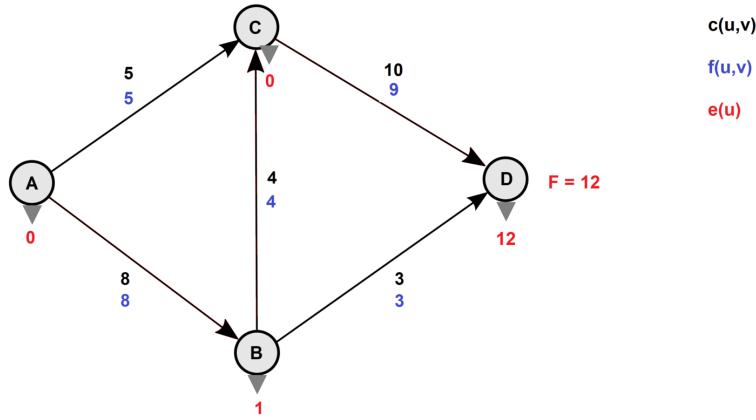


Рис. 5: Пример сети с истоком  $A$  и стоком  $D$ . Вершина  $B$  является переполненной.

## Определение 8.3.

Функция  $h : V \rightarrow \mathbb{Z}^+$  называется **высотой вершины**, если она удовлетворяет условиям:

1.  $h(s) = |V|$
2.  $h(t) = 0$
3.  $\forall (u,v) \in E, h(u) \leq h(v) + 1$

- **Проталкивание (push)**

Операция проталкивания из вершины  $u$  в вершину  $v$  может применяться тогда, когда  $e(u) > 0$ , то есть вершина  $u$  является переполненной,  $c_f(u,v) > 0$  и  $h(u) = h(v) + 1$ .

Данная операция работает следующим образом: по ребру  $(u,v)$  пропускается максимально возможный поток, то есть минимум из избытка вершины  $u$  и остаточной пропускной способности ребра  $(u,v)$ , вследствие чего избыток вершины  $u$ , остаточная пропускная способность ребра  $(u,v)$  и поток по обратному ребру  $(v,u)$  уменьшаются на величину потока, а избыток вершины  $v$ , поток по ребру  $(u,v)$  и остаточная пропускная способность обратного ребра  $(v,u)$  увеличиваются на эту же величину.

### Algorithm 12 Push Function

```
Function push(Node u, Node v)
    d = min(e(u), c(u, v) - f(u, v))
    f(u, v) += d
    f(v, u) = -f(u, v)
    e(u) -= d
    e(v) += d
```

По своему результату все проталкивания можно разделить на 2 группы. Будем называть проталкивание из вершины  $u$  в вершину  $v$  **насыщающим**, если после него остаточная пропускная способность ребра  $(u,v)$  стала равна нулю. Все остальные проталкивания будем называть **ненасыщающими**. Подобная классификация проталкиваний понадобится нам при оценке их количества.

- **Подъем (relabel)**

Операция подъёма применима для вершины  $u$ , если  $e(u) > 0$  и для всех  $(u,v) \in E$  выполнено  $h(u) \leq h(v)$ .

То есть, для переполненной вершины  $u$  применима операция подъёма, если все вершины, для которых в остаточной сети есть рёбра из  $u$ , расположены не ниже  $u$ . Следовательно, операцию проталкивания для вершины  $u$  произвести нельзя.

В результате подъёма высота текущей вершины становится на единицу больше высоты самой низкой смежной вершины в остаточной сети, вследствие чего появляется как минимум одно ребро, по которому можно протолкнуть поток.

---

#### Algorithm 13 Relabel function

---

**Function** relabel(**Node**  $u$ )  

$$h(u) \leftarrow \min\{h(v) : f(u, v) - c(u, v) < 0\} + 1$$


---

### 8.3 Доказательство корректности

Доказательство корректности алгоритма проталкивания предпотока (Push-Relabel) включает несколько ключевых шагов. Вот общий обзор доказательства корректности этого алгоритма:

1. **Лемма о высоте:** Доказывается, что высота вершин в остаточной сети удовлетворяет условию леммы о высоте. Это означает, что высота вершины является допустимой и соответствует количеству ребер от источника до данной вершины в остаточной сети.
2. **Лемма о насыщенных вершинах:** Доказывается, что все вершины, кроме источника и стока, либо насыщены (то есть имеют положительный предпоток, превышающий исходящий поток), либо являются просачивающимися (то есть имеют положительный предпоток, равный исходящему потоку). Эта лемма гарантирует, что не будет "застривания" предпотока в вершинах.
3. **Лемма о промежуточном потоке:** Доказывается, что в любой момент времени все вершины, кроме источника и стока, имеют промежуточный поток, который является допустимым потоком в остаточной сети.
4. **Теорема о корректности:** На основе лемм, описанных выше, доказывается, что алгоритм проталкивания предпотока находит максимальный поток в сети. Это означает, что алгоритм правильно находит оптимальное значение максимального потока между источником и стоком.

### 8.4 Сравнение сложности с Фордом-Фалкерсоном

Идея алгоритма Форда-Фалкерсона заключается в следующем. Изначально величине потока присваивается значение 0:  $f(u,v) = 0$  для всех  $u,v$  из  $V$ . Затем величина потока итеративно увеличивается посредством поиска увеличивающего пути (путь от источника  $s$  к стоку  $t$ , вдоль которого можно

послать ненулевой поток). Обычно этот поиск осуществляется с помощью обхода в глубину (DFS). Процесс повторяется, пока можно найти увеличивающий путь.

**Сложность алгоритма Форда-Фалкерсона** –  $O(|E| * f)$ , где  $f$  - максимальный поток в графе.

**Сложность push-reliable** –  $O(|V|^2 * |E|)$

**Сложность push-reliable с оптимизациями** –  $O(|V|^3)$

## 9 Поиск набора строк в тексте (Ахо-Корасик)

Source:

1. Алгоритм Ахо-Корасик (ИТМО конспект)
2. Бор (ИТМО конспект)
3. Запись лекции "Алгоритм Ахо-Корасика"
4. \* Алгоритм Ахо-Корасик (Хабр)

Дан набор строк в алфавите размера  $k$  суммарной длины  $m$ . Необходимо найти для каждой строки все её вхождения в текст.

### 9.1 Задача точного поиска наборов образцов

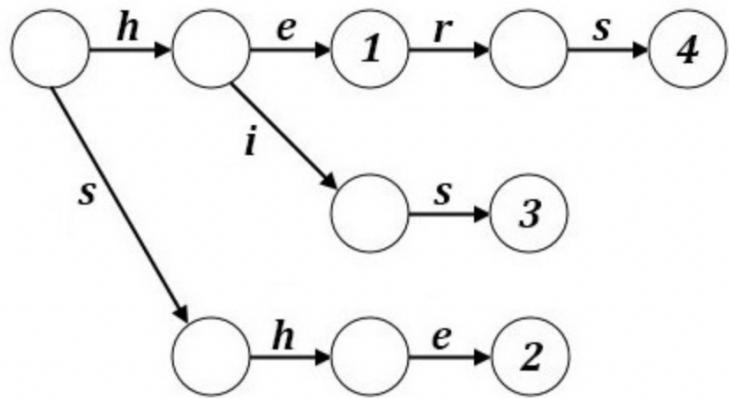
В данной задаче имеется набор образцов (также называемых ключевыми словами или паттернами) и текст, в котором нужно найти все вхождения этих образцов.

### 9.2 Trie

**Бор** (англ. trie, луч, нагруженное дерево) — структура данных для хранения набора строк, представляющая из себя подвешенное дерево с символами на рёбрах. Строки получаются последовательной записью всех символов, хранящихся на рёбрах между корнем бора и терминальной вершиной. Размер бора линейно зависит от суммы длин всех строк, а поиск в бору занимает время, пропорциональное длине образца.

#### Пример

Бор для набора образцов {he, she, his, hers}:



### 9.3 Задача о словаре

Задача о словаре также решается с помощью алгоритма Ахо-Корасик. В этом случае набор образцов представляет собой словарь, а текст проверяется на наличие слов из этого словаря. Примером такой задачи может быть проверка текста на наличие запрещенных слов или поиск ключевых слов в большом объеме данных.

## 9.4 Алгоритм Ахо-Корасик

### 1 Шаг: *Построение бора*

Строим бор из строк.

Построение выполняется за время  $O(m)$ , где  $m$  — суммарная длина строк.

### 2 Шаг: *Преобразование бора*

Обозначим за  $[u]$  слово, приводящее в вершину  $u$  в боре.

Узлы бора можно понимать как состояния автомата, а корень как начальное состояние.

Узлы бора, в которых заканчиваются строки, становятся терминальными.

Для переходов по автоматау заведём в узлах несколько функций:

- $\text{parent}(u)$  — возвращает родителя вершины  $u$ ;
- $\pi(u) = \delta(\text{parent}(u), c)$  — **суффиксная ссылка**, и существует переход из  $\text{parent}(u)$  в  $u$  по символу  $c$ ;
- $\delta(u, c) = \begin{cases} v, & \text{if } v \text{ is son by symbol } c \text{ in trie;} \\ \text{root}, & \text{if } u \text{ is root and } u \text{ has no child by symbol } c \text{ in trie;} \\ \delta(\pi(u), c), & \text{else.} \end{cases}$  — функция перехода

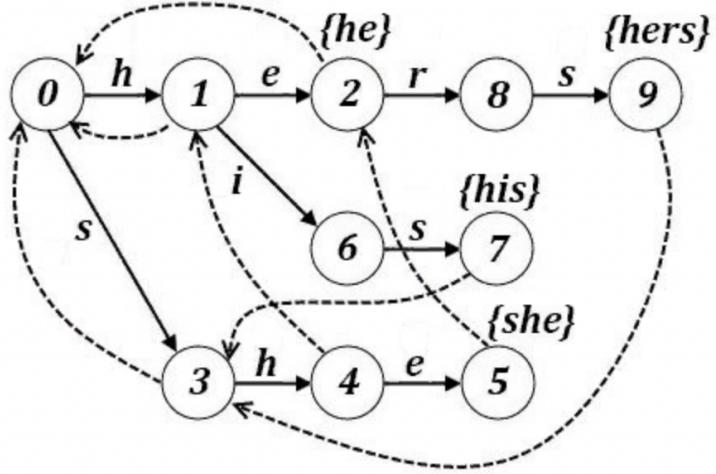
Мы можем понимать рёбра бора как переходы в автомата по соответствующей букве. Однако одними только рёбрами бора нельзя ограничиваться. Если мы пытаемся выполнить переход по какой-либо букве, а соответствующего ребра в боре нет, то мы тем не менее должны перейти в какое-то состояние. Для этого нам и нужны суффиксные ссылки.

Суффиксная ссылка  $\pi(u) = v$ , если  $[v]$  — максимальный суффикс  $[u]$ ,  $[v] \neq [u]$ . Функции перехода и суффиксные ссылки можно найти либо алгоритмом обхода в глубину с ленивыми вычислениями, либо с помощью алгоритма обхода в ширину.

Из определений выше можно заметить два следующих факта:

- функция перехода определена через суффиксную ссылку, а суффиксная ссылка — через функцию переходов;
- для построения суффиксных ссылок необходимо знать информацию только выше по бору от текущей вершины до корня.

Это позволяет реализовать функции поиска переходов по символу и суффиксных ссылок ленивым образом при помощи взаимной рекурсии.



**Рис. 6:** Пример автомата Ахо-Корасик

Суффиксные ссылки обозначены пунктиром. Из вершин, для которых они не показаны, суффиксные ссылки ведут в корень.

Суффиксная ссылка для каждой вершины  $u$  — это вершина, в которой оканчивается наименнейший собственный суффикс строки, соответствующей вершине  $u$ . Единственный особый случай — корень бора: для удобства суффиксную ссылку из него проведем в себя же. Например, для вершины 5 с соответствующей ей строкой "she" максимальным подходящим суффиксом является строка "he". Видим, что такая строка заканчивается в вершине 2. Следовательно, суффиксной ссылкой для вершины 5 является вершина 2.

### 3 Шаг: Построение сжатых суффиксных ссылок

При построении автомата может возникнуть такая ситуация, что ветвление есть не на каждом символе. Тогда можно маленький бамбук заменить одним ребром. Для этого и используются сжатые суффиксные ссылки.

$$up(u) = \begin{cases} \pi(u), & \text{если } \pi(u) \text{ — терминальная вершина,} \\ \emptyset, & \text{если } \pi(u) \text{ — корень,} \\ up(\pi(u)), & \text{иначе.} \end{cases}$$

Где  $up(u)$  — сжатая суффиксная ссылка, т.е. ближайшее допускающее состояние (терминал) перехода по суффиксным ссылкам. Аналогично обычным суффиксным ссылкам, сжатые суффиксные ссылки могут быть найдены при помощи ленивой рекурсии.

# 10 Сложность алгоритмов

Source:

1. Дискретная математика и информатика - С.В. Рыбин (2021 г.) – Глава 3.5.8
2. \* [Оригинал шпаргалки в высоком разрешении](#)

## 10.1 Виды сложности

Существует несколько видов сложности алгоритмов, которые обычно используются для анализа и оценки эффективности алгоритмов. Некоторые из наиболее распространенных видов сложности включают:

### 1. Временная сложность (*Time Complexity*):

Это мера количества времени, необходимого для выполнения алгоритма в зависимости от размера входных данных. Оценка временной сложности позволяет предсказать, как будет изменяться время выполнения алгоритма при увеличении размера входных данных. Обычно время работы алгоритма выражается в терминах "O-нотации" (Big O notation), которая описывает его асимптотическое поведение при стремлении размера входных данных к бесконечности. Примеры временной сложности включают  $O(1)$  (константная сложность),  $O(n)$  (линейная сложность),  $O(n^2)$  (квадратичная сложность) и т. д.

### 2. Пространственная сложность (*Space Complexity*):

Пространственная сложность алгоритма определяет, сколько дополнительной памяти требуется для выполнения алгоритма в зависимости от размера входных данных. Она измеряется в терминах объема памяти, необходимого для хранения структур данных и временных переменных во время выполнения алгоритма. Пространственная сложность также может быть выражена в О-нотации, аналогично временной сложности.

### 3. Вычислительная сложность (*Computational Complexity*):

Вычислительная сложность алгоритма определяет количество вычислительных ресурсов, таких как операции с плавающей точкой, умножения, сравнения и другие, необходимых для выполнения алгоритма. Эта сложность связана с конкретной аппаратной платформой и может варьироваться в зависимости от реализации алгоритма.

### 4. Сложность по другим параметрам:

В зависимости от конкретной задачи или алгоритма могут быть определены и другие виды сложности, специфичные для этой задачи. Например, в некоторых случаях может быть важной оценка энергетической сложности алгоритма или сложности связи/коммуникации при параллельном выполнении.

## 10.2 Понятие вычислительной сложности в зависимости от размера входа

### • Константный – $T(n) = O(1)$

Любой алгоритм, всегда требующий независимо от размера данных одного и того же времени, имеет константную сложность.

Если не считать эффективность в наилучшем случае, в этот класс попадает небольшое число алгоритмов.

- **Логарифмический** –  $T(n) = O(\log(n))$

Эффективность алгоритма логарифмическая. Алгоритм начинает работать намного медленее с увеличением размера выходных данных ( $n$ ).

Такое время работы характерно для алгоритмов, в которых большая задача делится на маленькие, каждая из которых решается по отдельности.

- **Линейный** –  $T(n) = O(n)$

Эффективность алгоритма меняется линейно в зависимости от размера входных данных (обычно такое наблюдается, когда каждый элемент входных данных требуется обработать линейное число раз).

К этому классу относятся алгоритмы, выполняющие сканирование списка, состоящего из  $n$  элементов (Например, алгоритм поиска методом последовательного перебора).

- **Линейно-логарифмический** –  $T(n) = O(n \log(n))$

Такая эффективность характерна для алгоритмов, которые делят большую проблему на маленькие, а затем, решив их, соединяют их решения.

К этому классу относятся большое количество алгоритмов декомпозиции, таких как алгоритмы сортировки, быстрого преобразования Фурье.

- **Квадратичный** –  $T(n) = O(n^2)$

Обычно, подобная зависимость характеризует эффективность алгоритма, содержащих два встроенных цикла (Например, ряд операций, выполняемых над матрицами размера  $n \times n$ ).

- **Кубический** –  $T(n) = O(n^3)$

Обычно, подобная зависимость характеризует эффективность алгоритма, содержащих три встроенных цикла.

- **Экспоненциальный** –  $T(n) = O(2^n)$

Данная зависимость типична для алгоритмов, выполняющих обработку всех подмножеств некоторого  $n$ -элементного множества.

Часто термин "экспоненциальный" применяется в широком смысле и означает очень высокий порядок роста (аргумент Big-O).

- **Факториальный** –  $T(n) = O(n!)$

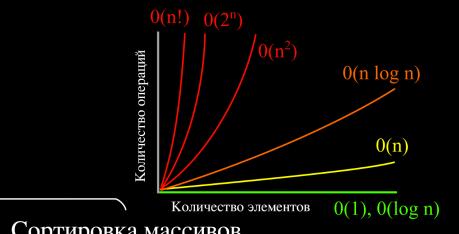
Характеризует алгоритмы, выполняющих обработку всех перестановок некоторого  $n$ -элементного множества.

## Легенда

Временная сложность vs. Пространственная сложность

Низкая Средняя Высокая  
 Низкая Средняя Высокая

## < Шпаргалка по BIG "O" >



### Структуры данных

#### Операции

	Временная сложность			Пространственная сложность		
	Низкая	Средняя	Высокая	Низкая	Средняя	Высокая
Массив						
Стек						
Очередь						
Односвязный список						
Двусвязный список						
Список с пропусками						
Хеш-таблица						
дерево поиска						
Декартово дерево						
В-дерево						
Красно-чёрное дерево						
Косое дерево						
AVL-дерево						
K-мерное дерево						

### Сортировка массивов

#### Алгоритмы

	Временная сложность	Пространственная сложность
Быстрая сортировка		
Сортировка слиянием		
Тимсорт		
Параллельная сортировка		
Сортировка пузырьком		
Сортировка вставками		
Сортировка выбором		
Сортировка деревом		
Сортировка Шелла		
Блочная сортировка		
Параллельная сортировка		
Сортировка подсчетом		
Cubesort		

Рис. 7: Полезная шпаргалка по Big-O

## 10.3 Константная сложность на примере `vector` и `unordered_set`:

\* Сделал через гpt, поэтому могут быть ошибки

`std::vector` - это последовательный контейнер, реализованный как динамический массив.

`std::unordered_set` - это ассоциативный контейнер, реализованный в виде хэш-таблицы.

### 10.3.1 Амортизированная

Использование термина "амортизированная константная сложность" означает, что в среднем время выполнения операции является константным, но иногда может быть немного больше, но все равно ограничено.

В случае с `std::vector`, вставка нового элемента в конец вектора выполняется за **амортизированное время  $O(1)$** . Это связано с тем, что вектор имеет динамическую структуру данных, которая увеличивает свою внутреннюю емкость при необходимости. Когда вектор заполняется, он выделяет новый блок памяти большего размера и копирует все существующие элементы в новую область. Этот

процесс, известный как "реаллокация" происходит реже по мере роста вектора, и поэтому в среднем сложность вставки в конец вектора является амортизированной константой.

В `std::unordered_set`, операции вставки, удаления и поиска также имеют **амортизированную константную сложность  $O(1)$** . Они основаны на использовании хэш-таблицы, которая разрешает коллизии путем создания цепочек элементов с одинаковыми хэшами. Хотя некоторые операции могут потребовать времени  $O(n)$  в худшем случае, в среднем время выполнения операций остается константным.

**Таким образом, как для `std::vector`, так и для `std::unordered_set`, операции вставки новых элементов в среднем выполняются за амортизированную константную сложность  $O(1)$** , что обеспечивает эффективность при работе с этими контейнерами.

### 10.3.2 В среднем

Для `std::vector`, вставка нового элемента в конец вектора в среднем имеет сложность  $O(1)$ , то есть постоянное время. Однако, если вектор достигает своей текущей емкости и требуется выполнить операцию реаллокации, то сложность вставки становится  $O(n)$ , где  $n$  - количество элементов в векторе. Это связано с необходимостью копирования всех элементов в новую область памяти большего размера. В среднем, при последовательной вставке элементов, время выполнения вставки будет близко к  $O(1)$ , но среди операций может быть несколько дорогих операций реаллокации, что повышает общую сложность.

Для `std::unordered_set`, операции вставки, удаления и поиска имеют среднюю сложность  $O(1)$  в амортизированном смысле. В хэш-таблице коллизии разрешаются путем создания цепочек элементов, и в большинстве случаев эти операции выполняются за постоянное время. Однако в худшем случае, когда происходит большое количество коллизий, время выполнения операций может достигать  $O(n)$ , где  $n$  - количество элементов в хэш-таблице.

Таким образом, в среднем время выполнения операций вставки, удаления и поиска элементов в `std::unordered_set` и вставки элементов в `std::vector` является постоянным  $O(1)$ . Но стоит отметить, что в редких случаях, сложность операций может быть выше, особенно при операциях реаллокации вектора или большом количестве коллизий в хэш-таблице.

## 11 Изоморфизм графов

Source:

1. Дискретная математика и информатика - С.В. Рыбин (2021 г.) – Глава 2.1.15
2. [Проверка изоморфности двух графов и поиск изоморфных подграфов: подход на основе анализа NB-Paths | Хабр](#)
3. [Быстрый поиск изоморфных подграфов | Хабр](#)

## 11.1 Задача изоморфизма:

### 11.1.1 Точный изоморфизм

---

**Определение 11.1.**

Графы  $G_1 = (V_1, E_1)$ ,  $G_2 = (V_2, E_2)$  называются **изоморфными**, если между ними существует взаимо однозначное отображение  $\varphi : G_1 \rightarrow G_2$  ( $V_1 \rightarrow V_2$ ,  $E_1 \rightarrow E_2$ ), которое сохраняет соответствие между ребрами графов, т.е. для любого ребра  $e = (v, u)$  верно:

$$e' = \varphi(e) = (\varphi(v), \varphi(u)), \quad (e \in E_1, e' \in E_2)$$

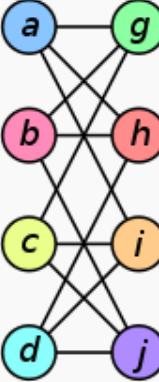
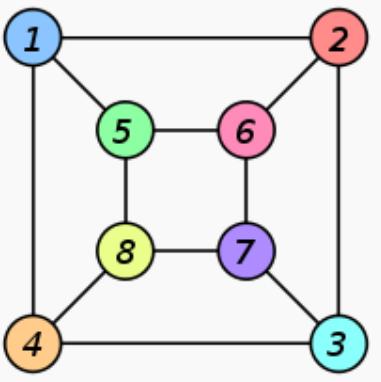
Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$

Рис. 8: Пример изоморфных графов

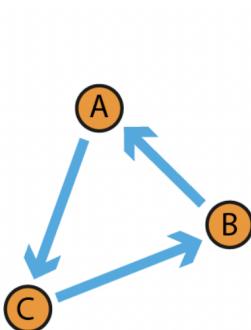
**Задача:**

Проверить, являются ли графы  $G_1$  и  $G_2$  изоморфными.

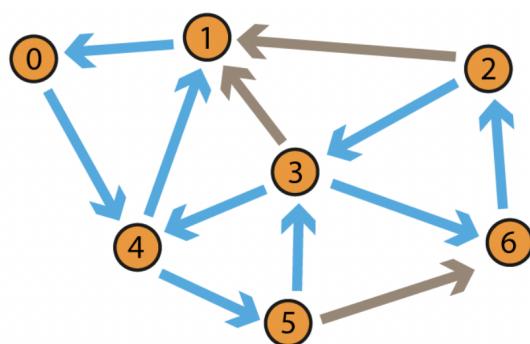
Другими словами, проблема подграфового изоморфизма состоит в определении, являются ли два заданных графа изоморфными, то есть можно ли переставить вершины одного графа так, чтобы получить другой граф.

### 11.1.2 Поиск подграфа в графе

Пусть у нас определено два графа: большой и малый. Нам требуется найти все такие подграфы большого графа, что они будут изоморфны малому. Большой граф будем называть **дата-графом**, малый - **паттерном**.



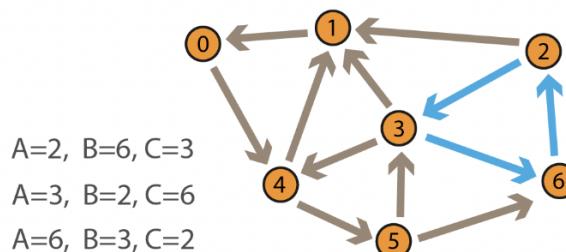
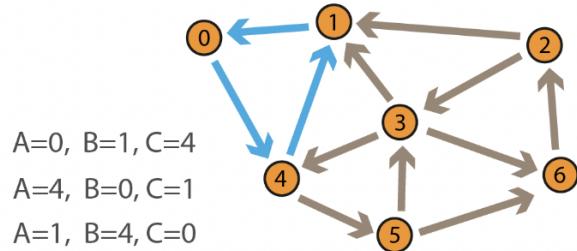
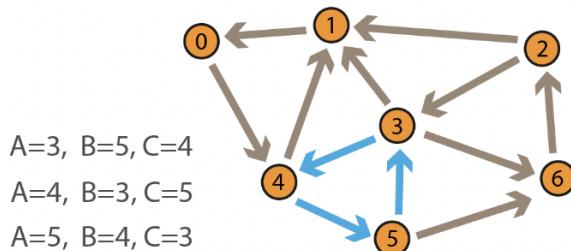
Паттерн



Дата-граф

На рисунке слева изображён паттерн ABC, справа - дата-граф 0123456.

В дата-графе паттерну изоморфны три подграфа:



При этом каждый подграф связан тремя разными комбинациями точек. Такое число комбинаций обусловлено тем, что паттерн симметричен. Если бы в нём совсем не было симметрии, то на каждый подграф приходилась бы ровно одна комбинация. Будем называть такие комбинации изоморфными.

Чтобы найти все изоморфные подграфы перебором, нужно перебрать все возрастающие комбинации вершин дата-графа длины, равной количеству вершин паттерна. Затем нужно построить по ним матрицы смежности и проверить их на изоморфность матрице смежности паттерна.

## 11.2 Алгоритм Ульмана (переборный с матрицей)

Алгоритм Ульмана основан на методе перебора с использованием матрицы соответствия. Он работает в два этапа: предварительная обработка и рекурсивный поиск.

1. *Предварительная обработка:*

- Создается матрица соответствия размером  $N \times M$ , где N - количество вершин в первом графе, а M - количество вершин во втором графе. Изначально все ячейки матрицы установлены в значение "false".
- Для каждой пары вершин  $(u, v)$ , где u - вершина первого графа, v - вершина второго графа, проверяется их изоморфность:
  - ★ Если вершины u и v имеют одинаковую степень и нет других вершин, которые уже были сопоставлены с u и v, то устанавливается значение "true" в соответствующей ячейке матрицы.
- Повторяется предыдущий шаг до тех пор, пока больше нельзя установить новые соответствия.

## 2. Рекурсивный поиск:

- Для каждой пары вершин  $(u, v)$ , где u - вершина первого графа, v - вершина второго графа, для которых значение "true" установлено в матрице соответствия:
  - ★ Помечаем вершины u и v как сопоставленные.
  - ★ Рекурсивно продолжаем поиск со следующей парой вершин.
  - ★ Если поиск успешен для всех пар вершин, то графы считаются изоморфными.
  - ★ Если поиск неуспешен, отменяем сопоставление вершин u и v и продолжаем с другой парой вершин.

Алгоритм Ульмана может быть реализован с использованием рекурсии и обратного отслеживания, чтобы определить все возможные изоморфные соответствия между вершинами двух графов.

Важно отметить, что алгоритм Ульмана имеет экспоненциальную сложность в худшем случае, поэтому он может столкнуться с проблемами при работе с большими графиками. Однако, на практике он может быть эффективен для небольших графов или при оптимизациях, таких как использование эвристик и проверка дополнительных условий для исключения некоторых неправильных сопоставлений.

## 11.3 Применение изоморфизма

### 1. Генетические Алгоритмы:

Для поиска генов в геноме.

### 2. Распознавание образов и компьютерное зрение:

Изоморфизм графов используется для сопоставления и распознавания образов. Это может включать сравнение и поиск схожих графов или детектирование объектов на изображении с использованием сопоставления графов.

### 3. Социальные сети и рекомендательные системы:

Анализ изоморфных графов может быть полезным для исследования социальных сетей, выявления общих паттернов и структурных связей между участниками сети. Кроме того, изоморфные графы могут быть использованы для рекомендации контента или товаров на основе схожих интересов и предпочтений.

## 12 Суффиксные деревья

Source:

1. Сжатое суффиксное дерево (ИТМО конспект)
2. Запись лекции "Алгоритм Ахо-Корасика"

---

### Определение 12.1.

---

**Суффиксное дерево** (сжатое суффиксное дерево)  $T$  для строки  $s$  (где  $|s| = n$ ) — дерево с  $n$  листьями, обладающее следующими свойствами:

- Каждая внутренняя вершина дерева имеет не меньше двух детей.
- Каждое ребро помечено непустой подстрокой строки  $s$ .
- Никакие два ребра, выходящие из одной вершины, не могут иметь пометок, начинающихся с одного и того же символа.
- Дерево должно содержать все суффиксы строки  $s$ , причем каждый суффикс заканчивается точно в листе и нигде кроме него.

Данное определение порождает следующую проблему:

Рассмотрим дерево для строки  $habxa$ : суффикс  $ha$  является префиксом суффикса  $habxa$ , а, значит, этот суффикс не заканчивается в листе. Для решения проблемы в конце строки  $s$  добавляют символ, не входящий в исходный алфавит: **защитный** символ. Обозначим его как  $\$$  (защитный символ). Любой суффикс строки с защитным символом действительно заканчивается в листе и только в листе, т.к. в такой строке не существует двух различных подстрок одинаковой длины, заканчивающихся на  $\$$ .

Далее  $n$  — длина строки  $s$  с защитным символом.

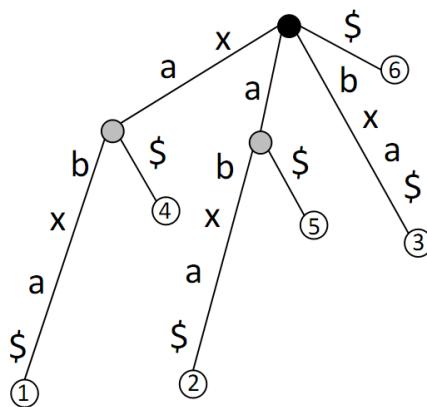


Рис. 9: Суффиксное дерево для строки  $habxa$  с защитным символом

### Алгоритм построения суффиксного дерева:

Основная идея алгоритма построения суффиксного дерева состоит в том, чтобы последовательно добавлять суффиксы строки в дерево. Построение дерева выполняется в линейном времени относительно размера входной строки.

1. Создание корневого узла дерева.
2. Для каждого суффикса строки:
  - Начиная от корневого узла, просматриваем существующие пути по символам суффикса.
  - Если путь совпадает с текущим символом суффикса, переходим по пути и движемся к следующему символу суффикса.
  - Если путь не совпадает с текущим символом суффикса, создаем новую ветвь от текущего узла, представляющую оставшийся суффикс.
  - Если ветвь уже существует, разделяем ее на две части и добавляем новый узел для вставки оставшегося суффикса.
  - Повторяем процесс для каждого суффикса.
3. Построение дерева завершается, когда все суффиксы добавлены.