

Алгоритмы и структуры данных, 1 курс, 4 модуль ПМИ ФКН ВШЭ

Михаил Густокашин, 2022

`mgustokashin@hse.ru`

Оглавление

1	Обход в глубину: связность, раскраска в два цвета, циклы	7
1.1	Способы задания графов	7
1.2	Обход в глубину	9
1.3	Проверка на связность, компоненты связности	10
1.4	Как это запрограммировать	11
1.5	Раскраска графа в два цвета	11
1.6	Классификация ребер графа	12
1.7	Поиск циклов в графе	14
2	Обход в глубину: топологическая сортировка, мосты, точки сочленения	15
2.1	Свойства меток времени	15
2.2	Топологическая сортировка графа	16
2.3	Мосты	18
2.4	Точки сочленения	20
2.5	Сильная связность	22
3	Обход в ширину	25
3.1	Обход в ширину	25
3.2	Использование очереди	26
3.3	Несколько начальных и конечных вершин	27
3.4	Кратчайший путь на 0-1 графе	28
3.5	Кратчайший путь на 0-K графе	29
3.6	Вершины и ребра на кратчайших путях	30
3.7	Кратчайшие пути в больших графах	31
3.8	Кратчайшие пути в графах состояний	31
4	Алгоритм Дейкстры	33
4.1	Поиск кратчайшего пути во взвешенном графе	33
4.2	Алгоритм Дейкстры	33
4.3	Корректность алгоритма Дейкстры	34
4.4	Алгоритм Дейкстры для разреженных графов	35
4.5	Вершины и ребра на кратчайших путях	36
4.6	Задача о самом широком пути	37
4.7	Второй по величине путь	38
4.8	K-ый по величине путь	38

5	Алгоритмы Флойда и Форда-Беллмана	41
5.1	Алгоритм Флойда	41
5.2	Восстановление пути	42
5.3	Транзитивное замыкание графа	43
5.4	Применимость алгоритма Флойда	44
5.5	Отрицательные ребра	44
5.6	Алгоритм Форда-Беллмана	45
5.7	Доказательство корректности	45
5.8	Оптимизации и отрицательные циклы	45
5.9	Информация для размышления	46
6	Минимальное остовное дерево и СНМ	49
6.1	Минимальное остовное дерево	49
6.2	Алгоритм Борувки	50
6.3	Алгоритм Прима	51
6.4	Система непересекающихся множеств	53
6.5	Наивная реализация СНМ	53
6.6	Списки элементов множества	53
6.7	Корневые деревья	54
6.8	Алгоритм Крускала	55
6.9	Заметки о СНМ	55
7	Бинарные деревья поиска	57
7.1	Бинарное дерево поиска	57
7.2	Обход дерева поиска	59
7.3	Следующий и предыдущий элементы	59
7.4	Определение элемента по номеру	60
7.5	АВЛ-дерево	61
8	Декартово дерево	65
8.1	Декартово дерево	65
8.2	Merge и Split	66
8.3	Вставка и удаление элементов	67
8.4	Декартово дерево по неявному ключу	68
8.5	Групповые операции	69
8.6	Проталкивание обещаний	70
8.7	Избавление от y	70
9	Префиксные суммы, разреженные таблицы и другое	73
9.1	Префиксные суммы	73
9.2	Двумерные префиксные суммы	74
9.3	Разреженные таблицы	75
9.4	Дерево Фенвика	77
10	Наименьший общий предок	79
10.1	Взаимное расположение вершин в дереве	79
10.2	Наименьший общий предок	80
10.3	Двоичные подъемы	81

10.4 LCA через двоичные подъемы	82
10.5 Кратчайшие пути в дереве	83
10.6 Минимальное остовное дерево с заданным ребром	84
10.7 Дополнительная информация в вершинах	85
11 Арифметика и теория чисел	87
11.1 Простые числа	87
11.2 Факторизация числа	88
11.3 Решето Эратосфена	88
11.4 НОД и НОК	89
11.5 Полезные свойства факторизации	90
11.6 Быстрое возведение в степень	91

Лекция 1

Обход в глубину: связность, раскраска в два цвета, циклы

1.1 Способы задания графов

Что такое граф вам известно из курса дискретной математики. Договоримся обозначать множество вершин графа как V , а множество ребер как E . В случаях, где это не вызывает неоднозначностей, например, при оценке сложности, будем использовать обозначения V и E для количества вершин и ребер соответственно (вместо $|V|$ и $|E|$).

Поиск всех соседей вершины будет составной частью любого алгоритма на графах, поэтому нам необходимо выбрать эффективную структуру данных для решения этой задачи. Почти всегда в худшем случае нам придется искать соседей для каждой из вершин графа, поэтому будем оценивать эффективность этой операции.

Чаще всего в условиях задач граф будет задан примерно следующим образом:

В первой строке вводится два числа V и E — количество вершин и ребер соответственно. Вершины нумеруются числами от 1 до V . В следующих E строках задаются пары чисел F_i, T_i — номера вершин, соединенных ребром.

Из контекста задачи всегда будет понятно, является ли граф ориентированным или неориентированным. В случае, если граф не ориентированный, каждое ребро задается только один раз, т.е. в списке ребер будет присутствовать пара соединенных вершин $F_i - T_i$, но не будет пары $T_i - F_i$. Мы будем хранить неориентированный граф как ориентированный, в котором каждое неориентированное ребро превращается в пару ориентированных ребер туда и обратно.

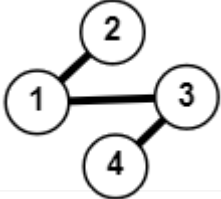
Такой способ задания графа называется «список ребер». Для решения задачи поиска всех соседей вершины нам придется написать цикл, который проходит по всем ребрам и проверяет, совпадает ли начальная вершина ребра с той, для которой мы ищем соседей. В случае, если граф неориентированный, нам придется сравнить текущую вершину и со второй вершиной в ребре. Этот способ поиска соседей будет занимать $O(E)$ для одной вершины и $O(VE)$, если мы будем искать соседей для каждой вершины графа. Его можно ускорить, например, отсортировав все ребра по начальной вершине и воспользовавшись бинарным поиском. В случае неориентированного графа его будет удобно сделать ориентированным, где каждое неориентированное ребро будет представлено как пара ориентированных (т.е. для каждого ребра $F_i - T_i$ из входных данных в набор ребер удобно добавить ребро $T_i - F_i$). Сортировка ребер займет $O(E \log E)$, поиск первого соседа в списке ребер займет $O(\log E)$ для одной вершины и $O(V \log E)$ для всех

вершин, а также в сумме мы пройдем по всем ребрам, что займет $O(E)$. Поскольку в «нормальных» графах $E \geq V$, то для них можно использовать оценку $O(E \log E)$ для поиска всех соседей всех вершин. Это медленный и трудоемкий способ.

Одним из возможных вариантов хранения графа является «матрица смежности» — таблица размером $V \times V$, где номера соединенных ребром вершин задают строку и столбец. В случае, если между вершинами i и j в графе есть ребро, то в ячейке таблицы с индексами i и j ставится единица (или true), а в случае, если ребра нет — ноль (или false). Преобразовать список смежности в матрицу смежности довольно просто. Изначально матрица смежности должна быть заполнена нулями. После считывания очередного ребра $F_i - T_i$ в ячейку матрицы $[F_i][T_i]$ записывается единица, а в случае неориентированного графа надо также записать единицу в ячейку $[T_i][F_i]$. Поскольку в задачах чаще всего используется нумерация вершин с единицы, а массивы на языке C++ нумеруются с нуля, то лучше сделать матрицу размером $(V + 1) \times (V + 1)$ — это позволит сохранить нумерацию такой же, как во входных данных, хотя нулевая строка и столбец и не будут использоваться, немного повышая расход памяти. Сложность построения составит $O(V^2)$ на выделение памяти и инициализацию матрицы нулями и $O(E)$ на заполнение единицами для существующих ребер. Сложность поиска соседа для одной вершины составит $O(V)$, а для всех соседей $O(V^2)$. Итоговая сложность составит $O(V^2)$ для поиска всех соседей всех вершин. Этот способ хранения графов прост в реализации и часто подходит для графов, в которых $E \approx V^2$ (полных или почти полных графов). Тем не менее, для большинства графов из реальной жизни $E \ll V^2$ и использование матрицы смежности неэффективно.

Наиболее часто используется способ хранения графов, называемый «список смежности». Для хранения графа в виде списка смежности создается V динамически расширяемых массивов (векторов в C++), изначально пустых. При считывании очередного ребра $F_i - T_i$ в вектор с номером F_i добавляется число T_i (в случае неориентированного графа также необходимо добавить в вектор с номером T_i число F_i). Таким образом, для каждой вершины просто хранится вектор всех ее соседей. Чтобы избежать проблемы с разницей в нумерации в условиях задач и языке C++ лучше заводить $V + 1$ вектор, при этом вектор с индексом 0 всегда будет оставаться пустым. Сложность построения списка смежности составит $O(E)$, для того чтобы для каждой вершины найти всех ее соседей потребуется $O(V + E)$ операций (т.к. мы пройдем по всем вершинам за V и в сумме посмотрим на каждое ребро за E). Для «нормальных» графов с $E \geq V$ обычно пользуются записью $O(E)$. В случае, когда $E \approx V^2$ такой способ хранения графа будет потреблять больше памяти, чем матрица смежности, т.к. вектор может иметь выделенную, но не использованную память, а также для каждого ребра нам потребуется хранение числа от 1 до V вместо хранения одного бита в матрице смежности.

Посмотрим на пример для неориентированного графа:

Список ребер	Матрица смежности	Список смежности	Рисунок																																				
<div>4 3</div> <div>3 1</div> <div>1 2</div> <div>4 3</div>	<table><tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr><tr><td>2</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>3</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr><tr><td>4</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>		0	1	2	3	4	0	0	0	0	0	0	1	0	0	1	1	0	2	0	1	0	0	0	3	0	1	0	0	1	4	0	0	0	1	0	<div>0:</div> <div>1: 3, 2</div> <div>2: 1</div> <div>3: 1, 4</div> <div>4: 3</div>	
	0	1	2	3	4																																		
0	0	0	0	0	0																																		
1	0	0	1	1	0																																		
2	0	1	0	0	0																																		
3	0	1	0	0	1																																		
4	0	0	0	1	0																																		

В некоторых случаях графы задаются довольно необычным способом. Например, в условии задачи может быть сказано, что вершины соединены ребром, если НОД их номеров больше либо равен K . В таких ситуациях иногда не имеет смысла заранее строить явное представление графа, а находить номера соседей вершины по мере необходимости, например, перебирая все вершины графа и выбирая подходящие (или, придумав более эффективный способ их нахождения).

1.2 Обход в глубину

Обход в глубину (или поиск в глубину) — один из основных алгоритмов на невзвешенных графах, предназначенный для их анализа. По-английски алгоритм называется depth-first search (DFS).

Обход в глубину начинается с одной из вершин графа. Из этой вершины мы переходим в одного из непосещенных соседей. Если все соседи вершины посещены, то мы возвращаемся вдоль пройденного пути, пока не обнаружим вершину, у которой есть непосещенные соседи. Процесс закончится, когда мы вернемся в исходную вершину и все ее соседи будут посещены.

Процесс поиска в глубину несложно представить себе как поиск выхода из лабиринта, в котором есть залы (вершины) и проходы между ними (ребра). Мы находимся в каком-то зале и хотим обойти весь лабиринт, достижимый из этого зала. Мы помечаем зал как посещенный (например, нарисовав крестик на полу) и переходим в какой-либо непосещенный соседний зал. Придя в новый зал, мы сразу же ставим стрелочку, указывающую на тот коридор, по которому мы пришли и помечаем его как посещенный. Пытаемся найти еще непосещенный соседний зал и, если такой нашелся — идем в него и повторяем процесс. Если все соседние залы посещены — возвращаемся по стрелочке. Мы обойдем весь лабиринт, когда вернемся в зал без стрелочки (это обязательно начальный зал, во всех остальных залах стрелочка есть) и все соседние залы будут помечены как посещенные.

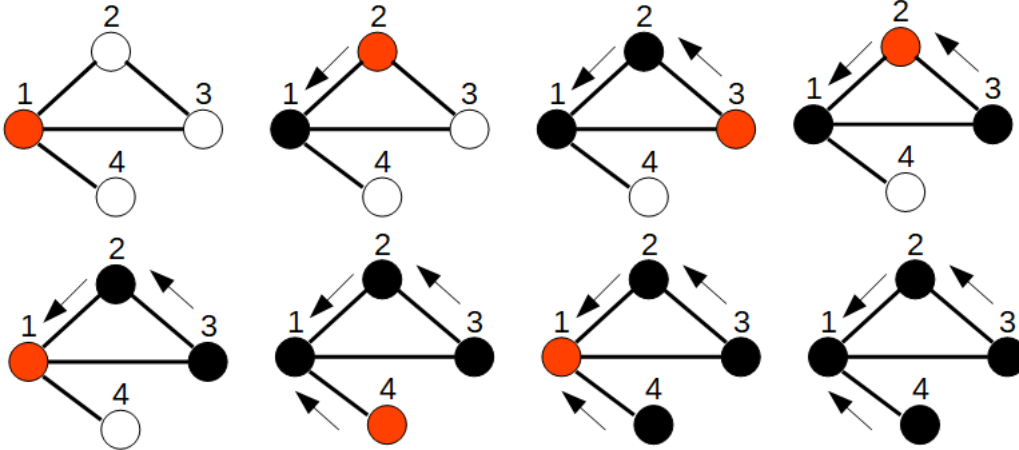
Для реализации хранения пройденного пути удобнее всего использовать рекурсию, но можно обойтись и без нее.

Рекурсивная реализация обхода в глубину очень проста:

```
void dfs(size_t now, vector<vector<size_t>> &g, vector<bool> &v) {
    v[now] = true;
    for (size_t neig : g[now])
        if (!v[neig])
            dfs(neig, g, v);
}
```

}

Здесь now — номер текущей вершины, $g(\text{graph})$ — вектор векторов, задающий список смежности для графа, а $v(\text{visited})$ — массив пометок о посещении вершины. Переменная next обозначает номер очередного соседа. Возврат по пройденному пути в случае отсутствия непосещенных соседей обеспечивается стеком рекурсии.



На рисунке изображен пример работы обхода в глубину. Белые вершины еще не посещены, черные — уже посещены, а красным цветом обозначена активная вершина. Стрелками из вершины отображается, откуда мы попали в эту вершину и куда происходит возврат после того, как все соседи посещены.

1.3 Проверка на связность, компоненты связности

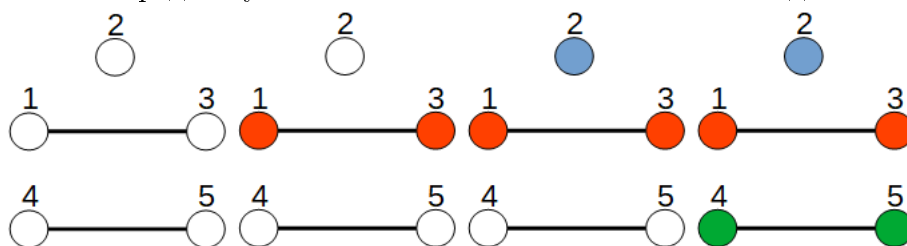
Неориентированный граф называется связным, если существует путь от любой вершины графа до любой другой вершины. Если из вершины запустить обход в глубину, то все помеченные вершины будут достижимы из этой вершины, т.е. для них будет существовать путь. Наивный алгоритм проверки графа на связность может быть реализован с помощью запуска обходов в глубину от каждой из вершин графа и его сложность составит $O(V \times E)$. Для каждого из обходов необходимо проверить, что помечены все вершины графа.

Процесс проверки графа на связность можно значительно ускорить, воспользовавшись следующим рассуждением: если от вершины 1 в неориентированном графе существует путь до вершины v и до вершины u , то также существует путь от вершины v до вершины u , например, проходящий через вершину 1. Поскольку мы можем попасть из вершины 1 в вершину v , то в неориентированном графе существует и обратный путь из v в 1, а оттуда можно попасть в вершину u . Таким образом, достаточно запустить один обход в глубину от любой вершины (чаще всего это вершина с номером 1) и просто проверить, что все вершины оказались помечены. Это можно реализовать как с помощью проверки того, что для всех вершин стоит отметка о посещении, так и с помощью подсчета числа посещенных вершин внутри функции.

Компонентой связности неориентированного графа называется такое нерасширяемое подмножество вершин и соединяющих их ребер, что существует путь из каждой вершины в каждую. Слово «нерасширяемое» в данном случае обозначает, что к этому множеству нельзя добавить ни одной вершины, чтобы оно по-прежнему было связным.

Простым примером графа с несколькими компонентами связности может служить карта автомобильных дорог России: в качестве вершин мы возьмем населенные пункты, а в качестве ребер — дороги, соединяющие их. Граф автомобильных дорог России состоит из нескольких изолированных компонент, например, от Москвы нельзя добраться по автомобильным дорогам до Южно-Сахалинска (добраться можно только используя паром) и до некоторых других населенных пунктов.

Часто встает задача «раскраски» графа на компоненты связности. Каждой вершине должен быть приписан номер компоненты связности, к которой относится эта вершина. Реализовать раскраску графа на компоненты связности можно также с помощью обхода в глубину. Для этого будет необходимо завести массив длиной V для хранения цвета каждой вершины и при входе в функцию обхода в глубину красить вершину в текущий цвет. Чтобы был раскрашен весь граф может потребоваться несколько запусков. Это реализуется с помощью цикла, перебирающего все вершины. Если очередная вершина не раскрашена, то увеличивается счетчик количества компонент и запускается функция обхода в глубину для этой вершины со значением цвета, равным текущему значению счетчика. Перед запуском счетчик количества компонент должен быть равен нулю.



На рисунке видно, в каком порядке будут закрашены компоненты. Красный цвет соответствует компоненте номер 1, синий — компоненте 2, зеленый — компоненте 3.

1.4 Как это запрограммировать

В функцию обхода в глубину передается очень много параметров: весь граф, номер текущей вершины, массив пометок о посещении и, например, информация о цвете, в который покрашена вершина (хотя это можно совместить с пометками). Кроме того, в процессе работы этой функции могут считаться различные статистики, такие как количество помеченных вершин и т.п. В некоторых более сложных задачах количество передаваемых параметров может стать еще больше.

Чтобы избежать большого количества параметров хочется воспользоваться глобальными переменными, но нельзя. Во-первых, это неприлично, а во-вторых, ваша программа не сможет работать с несколькими графами. Поэтому единственным выходом становится использование класса с полями, которые и будут играть роль глобальных переменных. Так и следует писать. Или, хотя бы, обернуть всю необходимую информацию о графе в структуру, которую и передавать в функцию.

1.5 Раскраска графа в два цвета

Рассмотрим задачу о раскраске вершин графа в два цвета так, чтобы каждое ребро графа соединяло вершины разных цветов. Если граф удалось раскрасить в два цвета, то он является двудольным: все вершины одного цвета лежат в первой доле, а друго-

го — во второй, при этом ребра соединяют только вершины из разных долей. Таким образом, то, что граф раскрашивается в два цвета и то, что он является двудольным — эквивалентные определения.

Для раскраски в два цвета нам пригодится теорема Кёнига (их несколько, это одна из них), которая гласит, что граф является двудольным тогда и только тогда, когда все циклы в графе имеют четную длину.

Если граф является двудольным, то очевидно, что необходимо пройти четное количество ребер чтобы вернуться в вершину, из которой начался путь (т.к. при проходе по ребру меняется доля, в которой мы находимся, а вернуться нужно в ту долю, из которой мы начали путь). Таким образом, можно считать доказанным, что в двудольном графе все циклы имеют четную длину.

Необходимо доказать теорему в другую сторону, а именно доказать, что из того, что все циклы имеют четную длину следует то, что граф является двудольным. Пусть граф связан и не имеет циклов нечетной длины. Возьмем произвольную вершину u и разобьем вершины графа на два непересекающихся множества U и V таким образом, что в множестве U содержатся вершины, длина кратчайшего пути до которых четная, а в V — нечетная. Исходная вершина u находится в множестве U (длина пути до нее равна 0, это четное число). Докажем, что в графе нет ребра ab такого, что a и b лежат одновременно в одном из множеств U или V . Доказывать будем от противного, допустим a и b одновременно лежат в U . Пусть P_0 — кратчайший путь из начальной вершины u в a , а P_1 — из u в b , оба пути имеют четную длину. Пусть v_0 — последняя вершина из пути P_0 , которая есть в пути P_1 (общая вершина всегда существует, например, начальная вершина u). Подпути от u до v_0 в путях P_0 и P_1 имеют одинаковую длину (иначе пройдя по более короткой подцепи от u до v_0 мы сократили бы путь от u до a или от u до b и нашли бы более короткий путь чем P_0 или P_1 соответственно). Подпути от v_0 до a и от v_0 до b имеют одинаковую четность и не имеют общих вершин, отличных от v_0 . А значит эти подпути вместе с ребром ab образуют цикл нечетной длины, что противоречит условию. Из этого следует, что множества U и V образуют две доли двудольного графа.

Проверка графа на двудольность с помощью обхода в глубину осуществляется очень просто: начинаем с произвольной вершины и красим ее в цвет 1, всех ее соседей красим в цвет 2, их соседей снова в цвет 1 и т.д. Если в какой-то момент сосед уже покрашен в тот же цвет, что и наша вершина, то граф не является двудольным и в нем существуют циклы нечетной длины. Есть прием, облегчающий выбор цвета соседней вершины: если цвет текущей вершины $color$, то цвет соседа должен быть $3 - color$. Несложно убедиться, что эта формула превращает 1 в 2 и наоборот.

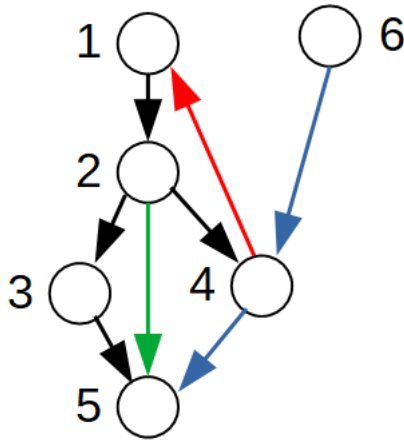
Также нужно не забыть случай, когда граф не является связным: в этом случае проверку на 2-раскрашиваемость нужно запустить для каждой компоненты связности с помощью прохода по всем вершинам (как это делается в задаче про раскраску компонент связности).

1.6 Классификация ребер графа

Рассмотрим те ребра графа, по которым прошел обход в глубину. Если, находясь в вершине u мы вызвали функцию обхода в глубину для соседней непомеченной вершины v , то ребро uv окажется в множестве ребер, по которым прошел обход в глубину.

Несложно заметить, что множество таких ребер образует дерево. Назовем такое дерево «деревом обхода в глубину». Если в графе было несколько компонент связности, то у нас получится лес обхода в глубину (несколько деревьев). Таким образом, мы смогли выделить класс ребер графа, которые можно назвать «ребра деревьев» — ребра, входящие в одно из деревьев обхода в глубину.

Однако, в графе содержатся и другие ребра, не входящие в дерево обхода в глубину. Рассмотрим на примере ориентированного графа.



На рисунке черным обозначены **ребра деревьев** (ребра 1-2, 2-3, 3-5, 2-4).

Красным обозначены **обратные ребра** — ребра, соединяющие вершину с ее предком в дереве обхода в глубину (ребро 4-1), к ним также относятся петли — ребра соединяющие вершину саму с собой.

Зеленым обозначены **прямые ребра** — ребра, соединяющие вершину с потомков в дереве обхода в глубину, но не входящие в лес обхода в глубину (ребро 2-5).

Синим обозначены **перекрестные ребра** — все остальные ребра графа (ребра 4-5 и 6-4). Если рисовать вершины слева-направо в порядке обхода, то такие ребра могут вести только справа налево (если бы они шли слева направо, то должны были бы быть ребрами дерева).

Нам необходимо научить классифицировать ребра графа. С помощью обычного обхода в глубину легко отличить ребра дерева от всех остальных, но для более точной классификации необходимо дополнить функцию поиска в глубину.

Модификация обхода в глубину будет состоять в том, что при входе в вершину мы будем красить ее в серый цвет, а при выходе из функции обхода в глубину (когда все соседи уже просмотрены) — в черный. Таким образом, в процессе выполнения обхода в глубину вершина сначала является белой (непосещенной), затем становится серой (обход в глубину вошел в вершину, но еще не обработал всех ее соседей) и, затем, черной (все соседи вершины посещены и также уже стали черными).

Пусть мы находимся в вершине u (раз мы находимся в ней, то она гарантированно серая) и соседа этой вершины v . Если вершина v белая, то ребро uv является ребром дерева, т.к. v еще не посещена и для нее надо запустить обход в глубину. Если вершина v серая, то ребро uv является обратным, т.к. серым в процессе обхода в глубину будут помечены все предки текущей вершины в дереве обхода и только они. Если же вершина v черная, то ребро uv является прямым или перекрестным. Различать эти ребра между собой нам пока не нужно.

В случае, если граф является неориентированным, то можно воспользоваться тем

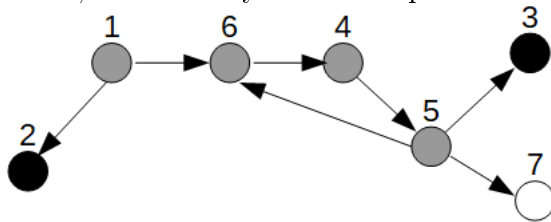
же алгоритмом. При этом мы представляем в неориентированном графе каждое ребро как пару ориентированных ребер и каждое из них может быть классифицировано как ориентированное. Если же нам нужно классифицировать именно неориентированное ребро, то мы выбираем из типов двух соответствующих этому ребру ориентированных ребер тот тип, который находится в классификации раньше. Можно заметить, что в таком случае в графе не будет прямых и перекрестных ребер — для них из двух вариантов типа выберутся обратные и ребро дерева. Таким образом, в неориентированном графе у нас существуют только ребра дерева и обратные ребра.

1.7 Поиск циклов в графе

Пользуясь трехцветной раскраской графа из предыдущего раздела, можно легко научиться проверять граф на наличие циклов и восстанавливать вершины, входящие в цикл. Критерием того, что в графе существует цикл, является существование в этом графе обратных ребер, то есть ребер, ведущих в серую вершину. В каждый момент времени серым цветом будут помечены вершины, лежащие на пути обхода в глубину от начальной вершины до текущей. Если из текущей вершины u есть ребро в серую вершину v , то в графе есть цикл, т.к. у нас существует путь от вершины v до u (v лежит на пути из начальной вершины в вершину u) и путь от вершины u до вершины v проходящий по одному ребру.

Ребро, ведущее из вершины u в черного соседа v не создает цикла, т.к. если сосед v черный, то мы можем быть уверены, что текущая вершина u не является потомком v в дереве обхода в глубину, т.к. текущая вершина серая, а все потомки v и сама эта вершина уже стали черными. Значит не существует пути из v в u и цикла не образуется.

Чтобы восстановить сами вершины, входящие в цикл, необходимо сделать следующее: находясь в вершине u , для которой обнаружился серый сосед v , необходимо запомнить номер v и выходить из рекурсивной функции, запоминая номера вершин, до тех пор, пока мы не дойдем до вершины v . Если развернуть запомненную последовательность, то мы получим все вершины от v до u , которые лежат в цикле.



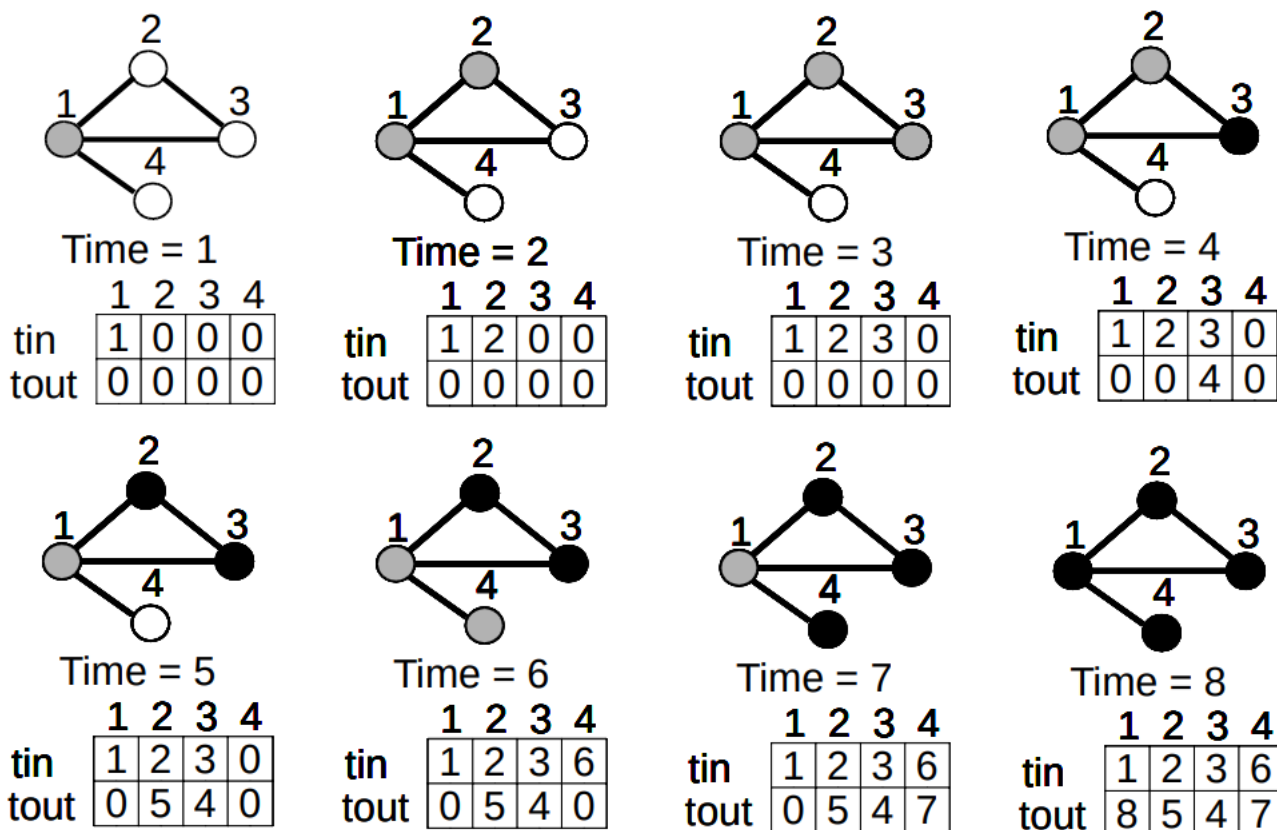
На рисунке изображен момент нахождения цикла. Обход в глубину был начат с вершины 1, соседи перебирались в порядке возрастания номеров. Сейчас функция обхода в глубину находится в вершине 5, сосед номер 3 уже обработан, и очередной сосед (вершина 6) оказывается серым. Мы запоминаем число 6 и начинаем выходить из обхода в глубину, запоминая номера вершин, до тех пор, пока не дойдем до вершины номер 6. Получится последовательность 5, 4, 6. После ее разворота получается 6, 4, 5 — последовательность номеров вершин, образующих цикл.

Лекция 2

Обход в глубину: топологическая сортировка, мосты, точки сочленения

2.1 Свойства меток времени

Функцию обхода в глубину можно дополнить таким образом, чтобы для каждой вершины определялось «время» входа и выхода в вершину. Временем здесь будем называть значение счетчика, которые увеличивается на единицу каждый раз при входе в вершину и выходе из нее. Запоминать время входа в вершину необходимо в начале функции обхода в глубину, а время выхода — в конце. Назовем массивы для хранения времени входа и выхода tin и $tout$.



Важным свойством меток времени является то, что если считать вход в вершину

открывающей скобкой, а выход — закрывающей (при этом тип скобки определяется номером вершины), то при обходе в глубину мы получим правильную скобочную последовательность. Для графа на рисунке мы получим следующую последовательность (верхний индекс обозначает номер вершины):

time	1	2	3	4	5	6	7	8
	(¹	(²	(³)) ³) ²	(⁴) ⁴) ¹

Такое свойство меток времени возникает из-за того, что выход из вершины возможен только после обработки и выхода из всех достижимых из нее вершин. То есть для всех вершин, достижимой из текущей, время входа будет больше, чем время входа в текущую, а время выхода — меньше. Строгое доказательство этого факта опирается на метод математической индукции.

Из свойств меток времени следует, что для двух вершин u и v должно выполняться ровно одно из трех свойств:

- 1) отрезки $[\text{tin}[u], \text{tout}[u]]$ и $[\text{tin}[v], \text{tout}[v]]$ не пересекаются
- 2) отрезок $[\text{tin}[u], \text{tout}[u]]$ полностью содержится в отрезке $[\text{tin}[v], \text{tout}[v]]$ — это означает, что вершина u является потомком вершины v в дереве обхода в глубину
- 3) отрезок $[\text{tin}[v], \text{tout}[v]]$ полностью содержится в отрезке $[\text{tin}[u], \text{tout}[u]]$ — это означает, что вершина v является потомком вершины u в дереве обхода в глубину

Использование меток времени совместно с раскраской вершин в белый, серый и черный цвета, позволяет полностью классифицировать все ребра графа. С помощью меток времени можно разделить обратные и перекрестные ребра: в случае обратных ребер один отрезок меток времени будет вложен в другой, а для перекрестных ребер отрезки не будут пересекаться.

2.2 Топологическая сортировка графа

Топологической сортировкой ориентированного графа без циклов называется такая перенумерация вершин графа, в которой все ребра ведут из вершины с меньшим номером в вершину с большим номером. Примером задачи топологической сортировки может служить задача о рассеянном профессоре, которому нужно одеться на работу: для пары предметов одежды указано, в каком порядке они должны быть надеты (например, носки раньше ботинок) и нужно составить какую-либо корректную последовательность, в которой профессор должен надевать вещи.

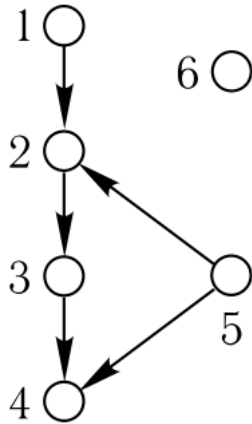
Во многих случаях задача о топологической сортировке имеет несколько решений, это можно легко представить себе на задаче о рассеянном профессоре, где, например, есть зависимость между надеванием рубашки и пиджака, а также носков и ботинок, но не важно, что будет надето раньше — пиджак или ботинки.

Наивный алгоритм топологической сортировки состоит в поиске вершины, в которую не входит ни одного ребра, удалении этой вершины вместе с исходящими ребрами и повторе этой операции раз. Если для каждой вершины поддерживать счетчик входящих ребер, то удаление вершин и исходящих из них ребер можно будет реализовать довольно быстро, за $O(E)$ для всех вершин: в этом случае при удалении ребра мы просто будем уменьшать счетчик входящих ребер на единицу для вершины, являющейся концом ребра. Для того, чтобы алгоритм работал эффективно, необходимо грамотно реализовать поиск вершин, в которые не входит ни одного ребра. В наивном алгоритме, на каждом шаге перебирающем все вершины, сложность этой операции составит

$O(V^2)$ для всех вершин. С использованием структур данных, позволяющих искать минимум, например, кучи, можно получить сложность $O((E + V) \times \log V)$ — при удалении ребра нам потребуется $\log V$ операций на изменение значения счетчика входящих ребер, также $\log V$ операций будет потрачено на удаление минимального значения из кучи при выборе очередной вершины.

Однако, выбор вершины без входящих ребер можно реализовать более эффективно. Для этого достаточно поддерживать счетчик входящих ребер для каждой вершины, а также список вершин, в которые не входит ни одного ребра. Удаление ребра будет занимать $O(1)$ (просто уменьшение счетчика), а в случае, если счетчик входящих ребер стал равен нулю, вершина просто добавляется в список вершин без входящих ребер также за $O(1)$. При такой реализации итоговая сложность будет равно $O(V + E)$.

Также для решения задачи топологической сортировки можно воспользоваться обходом в глубину. Для этого достаточно при выходе из функции обхода в глубину добавлять в конец списка с ответом номер текущей вершины. После обработки всех вершин графа этот список необходимо развернуть. При этом обход в глубину нужно запускать в цикле, для каждой непосещенной вершины, как это делалось для поиска компонент связности.



Для графа на рисунке номера вершин в порядке выхода функции обхода в глубину будет следующим: 4, 3, 2, 1, 5, 6. После разворота мы получим последовательность 6, 5, 1, 2, 3, 4, которая, в данном случае, является корректной топологической сортировкой. Кроме того, например, существует последовательность 1, 5, 2, 6, 3, 4, которая также является корректной топологической сортировкой (но такая последовательность не может быть получена с помощью обхода в глубину).

Докажем, что список номеров вершин, полученный в результате применения обхода в глубину, является корректной топологической сортировкой.

Воспользуемся методом от противного. Допустим, существует ребро (v, u) , такое что вершина u в получившемся списке стоит раньше вершины v . Но вершины в списке упорядочены по времени окончания их обработки функцией обхода в глубину, а из существования ребра (v, u) (и отсутствия пути из u в v) следует, что обработка вершины u должна закончиться раньше, чем вершины v . Вершина u может оказаться раньше вершины v в списке только в ситуации, когда существует путь из u в v , но т.к. в графе есть ребро (v, u) то это означает, что в графе есть цикл, что противоречит условию.

Часто оказывается удобно совместить топологическую сортировку с проверкой графа на наличие циклов — это можно сделать в одной функции обхода в глубину, используя и раскраску в три цвета, и запоминание номеров вершин при выходе из функции

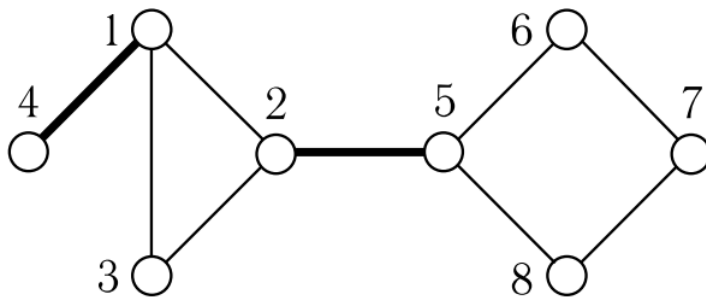
обхода в глубину.

2.3 Мосты

Мостом в неориентированном графе называется такое ребро, при удалении которого увеличивается количество компонент связности. При удалении из графа всех мостов он распадается на компоненты реберной двусвязности.

Внутри компоненты реберной двусвязности между любыми двумя вершинами этой компоненты существует не менее двух путей, не пересекающихся по ребрам. Действительно, если бы это было не так, то существовало бы ребро, через которое проходит любой путь от одной вершины до другой (это утверждение требует более строгого доказательства).

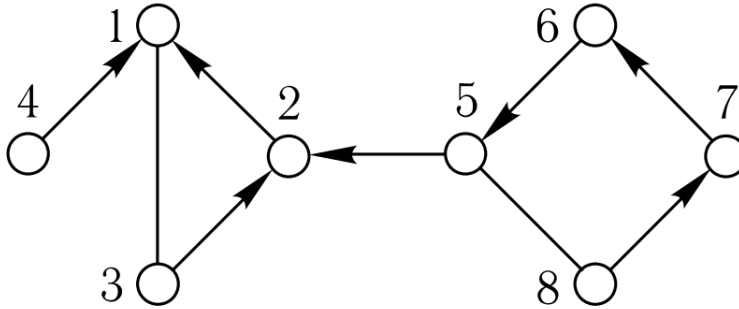
Между двумя компонентами реберной двусвязности не может существовать более одного ребра, т.к. иначе они образовали бы одну компоненту реберной двусвязности. Такие ребра, соединяющие компоненты реберной двусвязности и являются мостами.



На рисунке мосты выделены жирным, после их удаления граф распадется на три компоненты реберной двусвязности, состоящие из вершин (1, 2, 3), (4), (5, 6, 7, 8).

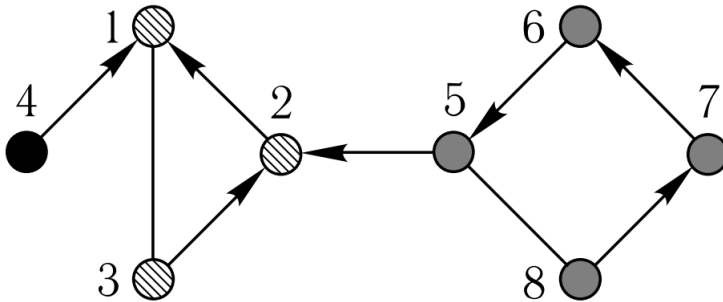
Рассмотрим алгоритм поиска мостов в неориентированном графе. Для этого требуется два поиска в глубину. Первый из них будет делать следующее: при прохождении по ребру (v, u) (то есть когда мы находимся в вершине v , а u еще не посещена) мы превращаем это ребро в ориентированное ребро (u, v) (ориентируем в обратную сторону). Те ребра, по которым обход в глубину не проходил (например, когда соседняя вершина уже помечена как посещенная), останутся в графе неориентированными. Ориентирование ребер при использовании списков смежности для хранения графа требует некоторых ухищрений. Нам необходимо, фактически, научиться удалять ориентированное ребро из v в u , оставляя ребро (u, v) . Для этого можно использовать словарь соседних вершин вместо списка, хранить в списке соседних вершин пары из номера соседней вершины и булевой пометки, удалено ребро или нет. Возможны и другие способы реализации удаления ребер из графа, позволяющие сохранить асимптотическую сложность поиска всех соседей.

После прохода обходом в глубину от вершины 1 и ориентирования ребер в противоположную сторону мы получим такой граф:



При выходе из функции обхода в глубину будем запоминать номер текущей вершины в списке так, как это делалось в топологической сортировке. В конце этот список будет развернут. В нашем случае этот список будет выглядеть таким образом: 1, 2, 3, 5, 6, 7, 8, 4.

После этого мы идем циклом по списку и, если вершина не помечена как посещенная, запускаем вторую функцию обхода в глубину, которая работает на ориентированном графе. Она аналогично функции раскраски на компоненты связности неориентированного графа и просто красит все достижимые вершины в цвет очередной компоненты. Для поиска мостов достаточно пройти по всем ребрам исходного графа и выбрать те из них, которые соединяют вершины из разных компонент.



Разберемся, почему алгоритм работает и деревья поиска, получающиеся при втором обходе в глубину, являются компонентами реберной двусвязности. Воспользуемся доказательством по индукции. Пусть очередное дерево поиска получилось в результате запуска обхода в глубину из вершины x , входящей в компоненту реберной двусвязности A . Нам необходимо доказать, что полученное дерево совпадает с компонентой A , предполагая, что все предыдущие деревья действительно образуют компоненты реберной двусвязности (это и есть предположение индукции).

Для того чтобы доказать, что очередное дерево совпадает с компонентой реберной двусвязности необходимо доказать два факта: дерево не содержит ни одной вершины из других компонент реберной двусвязности, а также факт, что дерево содержит все вершины из компоненты реберной двусвязности A .

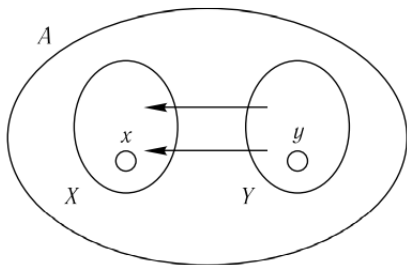
Начнем с первого (дерево не содержит вершин из других компонент двусвязности). Назовем другую компоненту реберной двусвязности B . Если между A и B в исходном графе нет ребер, то утверждение очевидно: на измененном графе мы не можем попасть из вершины x , принадлежащей A в одну из вершин множества B .

Допустим, между A и B существует более одного ребра. Тогда мы приходим к противоречию, т.к. между компонентами реберной двусвязности может существовать не более одного ребра, иначе они являются одной компонентой реберной двусвязности.

Осталось рассмотреть случай, когда между A и B есть ровно одно ребро (v, u) . Если

вершина u уже обработана (то есть компонента B найдена с помощью предыдущих операций выделения деревьев), то пройти по ребру (v, u) нельзя и вершина u не будет добавлена в дерево. Если же вершина u еще не обработана, то она стоит в списке позже вершины v . Значит, при первом поиске в глубину когда мы начали обработку вершины v вершина u была еще белой (ее обработка не началась), а значит не существовало пути из u в v и путь из v в u лежит по ребру (v, u) . Таким образом, при первом проходе это ребро будет сориентировано как (u, v) (в направлении от u к v), а значит попасть из v в u (из A в B) при втором обходе будет невозможно.

Второй факт (дерево содержит все вершины из компоненты реберной двусвязности) также будем доказывать от противного. Допустим, существует вершина y входящая в компоненту A но не входящая в дерево поиска. Поскольку по предположению индукции все ранее размеченные компоненты размечены правильно, то y не может принадлежать какой-либо уже размеченной компоненте. То есть y является непомеченной и недостижима из вершины x в графе с учетом ориентации ребер. Обозначим как X множество вершин компоненты A , достижимых из x , а как Y — множество вершин компоненты A , недостижимых из x . Поскольку X и Y лежат внутри одной компоненты реберной двусвязности A , то между ними есть как минимум два ребра.

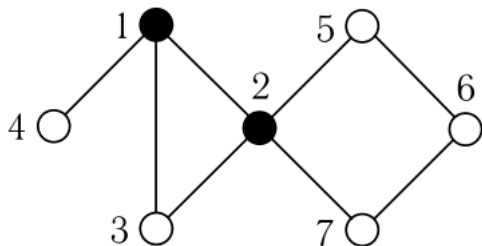


Поскольку оба ребра ориентированы, то они входили в дерево первого обхода в глубину и при первом проходе мы попадали из X в Y дважды, а из Y в X — ни разу. Обозначим за p и q вершины множества Y , соединенные ребрами с вершинами множества X . Т.к. при первом проходе в обе вершины мы попали из множества X , то пути от p до q внутри множества Y не существует, и ребра, соединяющие p и q с вершинами множества X являются мостами, что противоречит тому, что A (состоящая из X и Y) является компонентой реберной двусвязности.

Таким образом, деревья второго поиска в глубину, содержат в себе все вершины компоненты реберной двусвязности и только их.

2.4 Точки сочленения

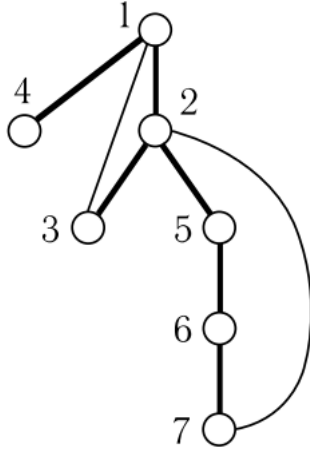
Точкой сочленения в неориентированном графе называется такая вершина, при удалении которой (а также смежных с ней ребер) увеличивается количество компонент связности.



Для графа на рисунке точками сочленения являются вершины 1 и 2.

Наивный алгоритм будет перебирать все вершины, удалять их и считать количество компонент связности, его сложность составит $O(V \times E)$. Для более быстрого поиска точек сочленения необходимо использовать модификацию обхода в глубину.

Построим лес обхода в глубину, с помощью которого для каждого ребра определим, является ли оно ребром дерева или обратным ребром (других типов ребер в неориентированных графах не существует). Рисунок соответствует графу из предыдущего рисунка, ребра дерева выделены жирным.



Корень дерева (вершина, с которой мы начали обход) является точкой сочленения тогда и только тогда, когда у него два или более сына. В нашем случае у вершины 1 два сына: вершины 2 и 4, поэтому вершина 1 является точкой сочленения. Если у корня один или ни одного сына, то, очевидно, корень не является точкой сочленения. А если сыновей два или более, то мы можем быть уверены, что при удалении корня граф распадется на несколько компонент связности (соответствующих поддеревьям корня), т.к. в неориентированном графе нет перекрестных ребер и единственный способ попасть из одного поддерева в другое — это пройти через корень.

Любая другая вершина дерева может быть точкой сочленения только если у нее есть хотя бы одно поддерево, которое «отсоединится» от графа, при удалении этой вершины. Поддерево отсоединяется от графа, только если в нем нет ни одной вершины, из которой обратное ребро ведет выше текущей вершины. Например, на рисунке вершина 5 не является точкой сочленения, т.к. в ее единственном поддереве с корнем в вершине 5 есть вершина 7, обратное ребро из которой ведет в вершину 2, которая находится выше вершины 5.

Таким образом, нам необходимо научиться для каждой вершины определять максимальную высоту, на которую мы можем подняться по обратным ребрам из вершин поддерева. Для этого введем массив up , где $up[v]$ будет обозначать минимальное время обнаружения среди всех вершин, достижимых из поддерева с корнем в вершине v при прохождении ровно по одному обратному ребру.

Вершина v дерева обхода в глубину является точкой сочленения тогда, когда у нее существует такой сын u , что $up[u] \geq tin[v]$, где $tin[v]$ — время обнаружения вершины v . Выполнение условия $up[u] \geq tin[v]$ означает, что из вершин поддерева с корнем в u все обратные ребра ведут в вершины, которые обнаружены позже v , т.е. не могут вести в вершины, лежащие выше v .

Для того чтобы решить эту задачу, необходимо считать величины tin и up для каждой вершины. Это можно сделать в одном обходе в глубину.

Нам обязательно необходимо считать величину *tin* для каждой вершины (*tout* в данном случае считать необязательно), а также подсчитать величину *up*. *up*[*v*] определяется как является минимум из всех *up*[*u*], где *u* — сын вершины *v* в дереве поиска в глубину, и всех *tin*[*w*], где *w* — достижимая по обратному ребру из *v* вершина. Изначально *up* можно заполнить числами $V + 1$, т.е. большими любого осмысленного значения.

Для того, чтобы удобнее обрабатывать корни деревьев, которые являются точками сочленения если у них больше одного ребенка, можно также для каждой вершины подсчитать количество детей у нее.

Код для решения этой задачи будет выглядеть примерно так (будем считать, что все векторы являются полями класса и инициализированы должным образом):

```
void dfs(size_t now, bool is_root) {
    visited[now] = true;
    ++time;
    tin[now] = time;
    for (auto neig : graph[now]) {
        if (!visited[neig]) {
            ++childs[now];
            dfs(neig, false);
            up[now] = min(up[now], up[neig]);
            if (up[neig] >= tin[now] && !is_root)
                ap[now] = true;
        } else {
            up[now] = min(up[now], tin[neig]);
        }
    }
}

int main() {
    ...
    for (size_t now = 1; now <= v; ++now) {
        if (!visited[now]) {
            dfs(now, true);
            if (childs[now] >= 2)
                ap[now] = true;
        }
    }
    ...
}
```

2.5 Сильная связность

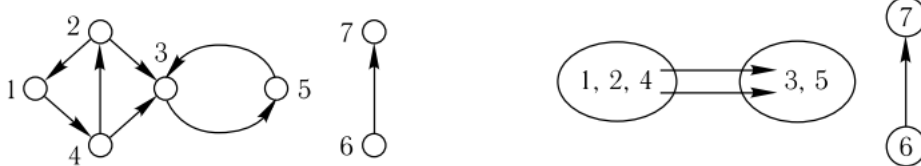
Ориентированный граф называется сильно связным если из любой его вершины существует путь в любую другую. Это определение полностью совпадает с определением связности для неориентированного графа, только в этом случае граф ориентированный. При проверке графа на связность мы пользовались идеей, что если от вершины 1 существует путь до всех вершин, то для любой пары вершин *v* и *u* существует путь из

v в u , например, проходящий через вершину 1. Для этой проверки нам было достаточно одного поиска в глубину, т.к. существование пути из вершины 1 в v автоматически означало наличие пути из v в 1. Для ориентированного графа это не так.

При запуске обхода в глубину от вершины 1 в ориентированном графе мы лишь сможем проверить, что существует путь из вершины 1 во все остальные вершины. Но также необходимо проверить наличие пути из всех вершин в вершину 1, что потребует V запусков обхода в глубину. Эта сложность совпадает со сложностью наивного решения, которое будет проверять существование пути от каждой вершины до каждой.

Чтобы получить лучшую сложность нам необходимо научиться быстро проверять существование пути от всех вершин графа до вершины номер 1. Этого можно добиться, если мы построим граф, в котором все ребра развернуты в обратную сторону и на нем уже проверим достижимость всех вершин из вершины номер 1. Действительно, если в развернутом графе есть путь из вершины 1 в вершину v , то в исходном графе существует путь из v в 1. Таким образом, мы можем проверить граф на сильную связность запустив два обхода в глубину: на прямом и на развернутом графе. При аккуратной реализации это можно сделать за $O(V + E)$.

По аналогии с компонентами связности можно ввести понятие компонент сильной связности. Компонентой сильной связности называется нерасширяемое подмножество вершин ориентированного графа, в котором каждая вершина достижима из каждой. При этом в ориентированном графе могут существовать ребра между компонентами связности (и даже несколько ребер между одной парой компонент), но все они направлены в одну сторону, т.к. если есть ребро из компоненты A в компоненту B и обратно, то объединение A и B тоже является компонентой сильной связности.



На рисунке изображены 4 компоненты сильной связности. Если заменить каждую компоненту исходного графа на вершину, то полученный граф будет называться «конденсацией» графа и будет ациклическим.

Поиск компонент сильной связности состоит из двух шагов:

1) Выполняется обход в глубину на исходном графе, при этом запоминается порядок окончания обработки вершин (как в топологической сортировке)

2) Выполняется обход в глубину на развернутом графе, при этом в цикле вершины перебираются в порядке, обратном тому, в котором они были запомнены в первом обходе. Каждое дерево второго обхода красится в свой цвет и соответствует компоненте сильной связности.

Докажем утверждение, что деревья обхода в глубину во втором поиске совпадают с компонентами сильной связности.

Сначала докажем утверждение, что если две вершины v и u входят в одну компоненту сильной связности, то они окажутся в одном дереве поиска.

Поскольку v и u находятся в одной компоненте сильной связности, то существует как путь из v в u , так и путь из u в v , это также верно и для развернутого графа. Из этого следует, что вершины попадут в одно и то же дерево поиска, независимо от порядка вызова вторых обходов в глубину.

Теперь докажем утверждение, что если две вершины v и u находятся в одном дере-

ве второго обхода в глубину, то они находятся в одной компоненте сильной связности. Пусть корнем дерева поиска, содержащем вершины v и u является вершина x . Поскольку v является потомком вершины x в дереве второго обхода (по обратному графе), то в исходном графе существует путь от v до x .

При втором обходе в глубину вершина v обнаруживается позже, чем вершина x , то есть в списке, созданном в результате первого обхода, вершина x находится раньше, чем вершина v , и ее обработка заканчивается позже, чем обработка вершины v .

Предположим, что при первом поиске в глубину v была обнаружена раньше x . Но поскольку существует путь от v до x мы получаем противоречие с их взаимным расположением в списке. Значит, предположение неверно и при первом обходе x обнаруживается раньше, чем v .

Таким образом $tin[x] < tin[v] < tout[v] < tout[x]$, т.е. v является потомком x в дереве первого обхода в глубину и в исходном графе существует путь из x в v .

Вместе с утверждением о существовании пути из v в x это позволяет утверждать, что v и x находятся в одной компоненте сильной связности. Аналогично доказывается утверждение о том, что u и x лежат в одной компоненте сильной связности, а это позволяет утверждать, что существует путь из v в u и обратно, например, проходящий через x .

После того как все вершины помечены номером компоненты сильной связности мы легко можем построить конденсацию графа, взяв в нее только те ребра, которые соединяют разные компоненты сильной связности.

Лекция 3

Обход в ширину

3.1 Обход в ширину

Кроме обхода в глубину существует еще один способ обхода вершин графа, чаще всего применяемый для поиска кратчайших путей в невзвешенных графах — обход в ширину или волновой алгоритм (breadth-first search). Будем решать задачу о поиске кратчайшего пути от одной вершины графа до всех остальных. Алгоритм для ориентированных и неориентированных графов одинаковый.

Наивный алгоритм поиска кратчайшего пути можно сформулировать следующим образом: создадим массив, хранящий для каждой вершины расстояние от начальной вершины до текущей. Сначала заполним этот массив бесконечностью (числом $v + 1$), а для начальной вершины запишем 0. Выполним $v - 1$ шаг (максимальная длина пути в графе равна v , она достигается когда все вершины входят в путь). Каждый шаг заключается в переборе всех вершин, выборе вершин, находящихся на расстоянии, равном номеру шага и пометке всех соседних непомяченных вершин числом на единицу больше, чем номер текущего шага. На нулевом шаге выберется только начальная вершина и все ее соседи пометятся числом 1. Затем выберутся вершины, находящиеся на расстоянии 1 и их непомяченные соседи пометятся числом 2 и т.д. Сложность такого наивного алгоритма при использовании списков смежности составит $O(V^2 + E)$ (V шагов, перебор V вершин, а также, в сумме, будут просмотрены все ребра).

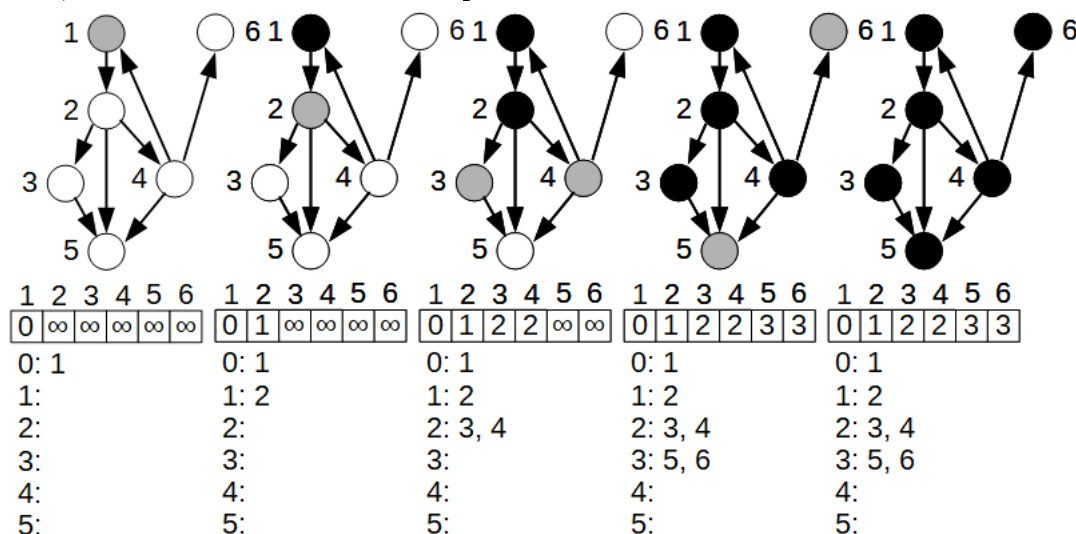
В наивном алгоритме легко найти место, которое требует выполнения лишних операций. Это перебор всех вершин на каждом шаге. Действительно, единственный случай, когда вершина оказывается на расстоянии X от начала, это случай, когда ее сосед находился на расстоянии $X - 1$. Таким образом, можно создать V динамически расширяемых массивов, i -ый из которых будет содержать номера всех вершин, находящихся на расстоянии i . В начале работы алгоритма начальная вершина помещается в массив с индексом 0, т.к. она находится на расстоянии 0. Затем, так же как и в наивном алгоритме, перебираются все соседи, непомяченные помечаются числом 1, а их номера помещаются в список с номером 1. На следующем шаге перебираются вершины из массива вершин, находящихся на расстоянии 1 и т.д.

Сложность такого алгоритма составить $O(V + E)$, т.к. каждое ребро и вершина будут обработаны лишь один раз. Также выполнение алгоритма можно завершить досрочно, если очередной список оказался пустым (то есть уже нет вершин, находящихся на заданном расстоянии). Во всех последующих списках вершин также не будет.

Несложно заметить, что в этом алгоритме в каждый момент времени используется

всего два массива: список вершин, находящихся на расстоянии x (на текущем шаге) и находящихся на расстоянии $x + 1$ (на следующем шаге). Поэтому можно организовать кольцевой буфер из массивов длиной 2 для хранения только списка вершин на текущем и на следующем шаге. В среднем это даст некоторую экономию памяти, однако мы все равно тратим $O(V)$ дополнительной памяти на хранения массива пометок и избавиться от него нельзя.

На рисунке серым помечены вершины, кратчайшее расстояние до которых равно номеру шага алгоритма, а также содержимое массивов кратчайших расстояний и списки вершин, находящихся на заданном расстоянии.



Для восстановления кратчайших путей можно использовать один из двух методов. В любом из них путь восстанавливается от конца к началу. В случае, если мы не используем дополнительной информации, для каждой вершины необходимо перебрать все соседние вершины, из которых достижима текущая и выбрать ту, расстояние до которой на один меньше, чем для текущей (это может вызвать определенные сложности в случае ориентированного графа). Второй способ состоит в запоминании для каждой вершины из какой вершины мы попали в текущую на этапе поиска кратчайших путей. Это требует $O(V)$ дополнительной памяти, однако значительно упрощает восстановление пути, особенно в случае ориентированных графов.

Кроме поиска кратчайших путей, обход в ширину может использоваться для решения части задач, для которых мы использовали обход в глубину, например, для проверки графа на связности, выделения компонент связности, проверки графа на раскрашиваемость в два цвета. Однако, дерево обхода в ширину не обладает таким количеством интересных свойств и чаще всего обход в ширину все же используется для поиска кратчайших путей. При этом дерево обхода в ширину обладает приятным свойством: оно является деревом кратчайших путей от начальной вершины, то есть длина кратчайшего пути в этом дереве и в графе совпадает для любой вершины графа.

3.2 Использование очереди

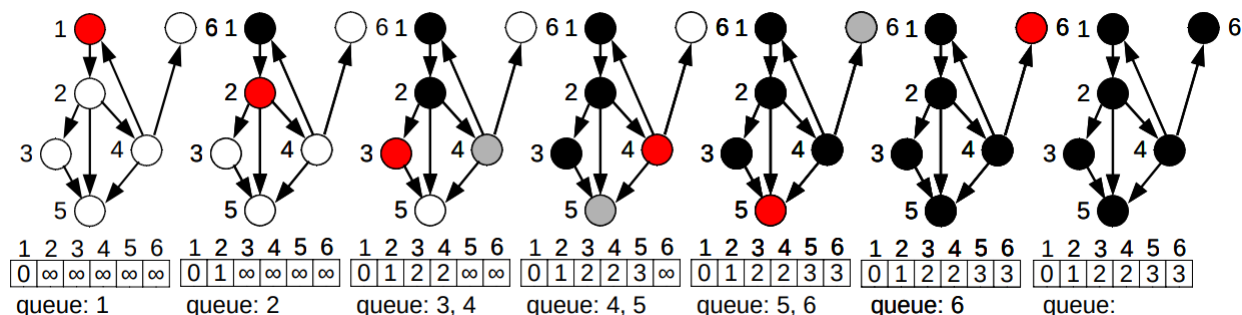
Несмотря на то, что способ, предложенный в предыдущем разделе прост в понимании и программировании, исторически для реализации обхода в ширину использовалась очередь.

Как мы уже заметили в прошлом разделе, можно не хранить списки вершин не для всех возможных расстояний: достаточно хранить лишь список вершин на «текущем» шаге и формировать по нему список вершин на «следующем» шаге из всех непо отмеченных соседей вершин текущего шага. Можно заменить эти два списка на одну очередь, в начале которой будут лежать вершины текущего шага, а в конец будут добавляться вершины следующего шага. Таким образом, в начале очереди находятся вершины текущего шага, в конце — следующего. Верность этого инварианта в процессе работы алгоритма легко доказывается по индукции.

Перед началом работы алгоритма нужно поместить начальную вершину в очередь, а в массиве расстояний указать для нее ноль.

При реализации обхода в ширину с помощью очереди нам все равно нужно хранить массив, позволяющий определить расстояние до вершины по ее номеру, а максимальный размер очереди может достигать V (когда все вершины соединены ребром с той, от которой начинается обход в ширину). Выполнение алгоритма завершается, когда очередь станет пуста.

Процесс работы обхода в ширину показан на рисунке. Белым помеченные еще не посещенные вершины, черным — уже посещенные, красным показана вершина, которая обрабатывается в данный момент (находящаяся в начале очереди), а серым — вершины, находящиеся в очереди, но не в ее начале.



3.3 Несколько начальных и конечных вершин

Обход в глубину достаточно часто возникает в задачах реальной жизни. Например, с некоторыми допущениями, мы можем считать схему метро неориентированным невзвешенным графом, где станции — это вершины, а перегоны и переходы между ними — ребра. В реальности время проезда по перегону или переход может быть разным, но сейчас мы этим пренебрежем.

Если использовать метрополитен как модель, то вполне естественна такая задача: найти кратчайший путь от одной станции до нескольких других. Такая задача возникает, когда цель путешествия на метро находится примерно на одинаковом расстоянии от нескольких станций.

Эта задача решается очень просто, ведь алгоритм обхода в ширину ищет кратчайшее расстояние от одной вершины до всех остальных, а значит нам необходимо просто запустить обход в ширину от начальной вершины и среди всех конечных вершин выбрать ту, расстояние до которой минимально.

Если ехать на метро в обратную сторону, то у нас возникает обратная задача: несколько начальных вершин и одна конечная. В случае неориентированного графа

мы можем просто запустить обход в ширину от конечной вершины и свести задачу к предыдущей. Если есть необходимость восстановить путь, то нужно не забыть развернуть его. Если же граф был ориентированным, то перед запуском обхода в ширину от конечной вершины необходимо развернуть все ребра в нем.

Самой сложной из этого класса задач является задача о поиске пути от нескольких начальных вершин до нескольких конечных. Есть два способа осмыслить и записать решение этой задачи.

Во-первых, мы можем создать виртуальную вершину и соединить ее ребрами со всеми начальными вершинами. Тогда от найденной длины пути нужно будет отнять единицу и не забыть удалить эту вершину если требуется восстановить путь.

Во-вторых, можно не создавать виртуальную вершину, а просто перед началом шагов алгоритма обхода в ширину добавить в очередь все начальные вершины и в качестве расстояния до них в массиве расстояний записать ноль. На рисунках были рассмотрены примеры, когда несколько вершин находились на расстоянии два и нас это не смущало. Аналогично нас не должно смущать наличие нескольких вершин на расстоянии ноль. Если же представить такое сложно, то можно представлять себе виртуальную вершину, соединенную с начальными, но явно ее в графе не создавать.

3.4 Кратчайший путь на 0-1 графе

В этом разделе мы научимся искать кратчайший путь на взвешенном графе, веса ребер в котором могут принимать только значения 0 или 1. В реальной жизни такому графу может соответствовать какая-либо система общественного транспорта, в которой есть проезда между остановками по цене билета (это ребра с весом 1) и бесплатного проезда (это ребра с весом 0) и стоит задача найти самый дешевый способ проезда.

Несмотря на то, что существуют универсальные алгоритмы поиска кратчайшего пути во взвешенном графе, которые будут изучены позднее, конкретно в этом случае можно использовать более эффективный алгоритм, основанный на обходе в ширину.

Сначала рассмотрим версию алгоритма обхода в ширину, в котором используются списки вершин, находящиеся на заданном расстоянии. Пусть у нас уже есть список вершин, находящихся на расстоянии x мы рассматриваем очередную вершину v . Если очередное ребро в непосещенную соседнюю с v вершину имеет вес 1, то все происходит как в обычном поиске в ширину — соседняя вершина попадает в список вершин на расстоянии $x + 1$. Если же ребро имеет вес 0, то вершина попадает в конец того же самого списка вершин, находящихся на расстоянии x .

Для того чтобы этот способ работал корректно, необходимо учесть ряд моментов. При проходе по всем вершинам, находящимся на расстоянии x , может получиться так, что обнаружится соседняя непомеченная вершина, соединенная ребром веса 1. Эта вершина попадет в список вершин на расстоянии $x + 1$. Затем, при дальнейшей обработке вершин на расстоянии x может произойти ситуация, что вершина u соединена с очередной вершиной ребром веса 0.

Во-первых, необходимо научиться замечать возникновение такой ситуации, ведь в обычном обходе в ширину признаком непосещенной вершины была бесконечность в соответствующей ей ячейке массива кратчайших расстояний, а теперь там будет записано число $x + 1$. Эту проблему легко решить, просто введя дополнительную проверку.

Во-вторых, вершина u уже есть в списке вершин на расстоянии $x + 1$ и после до-

добавления этой вершины в список вершин на расстоянии x она окажется одновременно в двух списках. Если не учесть этой ситуации, то алгоритм поиска в ширину будет работать долго и неправильно. В такой ситуации очень хочется удалить вершину из списка $x + 1$, но в случае использования `vector` для хранения списка вершин удаление будет происходить медленно. Если же использовать `unordered set` или обычный `set`, то удаление произойдет быстро, но после добавления вершины u в список вершин на расстоянии x может инвалидироваться итератор, указывающий на очередную обрабатываемую вершину, или может оказаться так, что вершину u вставится раньше этого итератора.

Наиболее простое решение этой проблемы состоит в том, чтобы все же хранить вершину u и в списке вершин на расстоянии x и в списке вершин на расстоянии $x + 1$, т.е. никаким образом не удалять ее из списка вершин на расстоянии $x + 1$. Однако, при обработке вершин необходимо будет проверять, что расстояние до нее в массиве расстояний совпадает с номером списка — если это не так, то вершину надо просто пропускать (она была обработана на предыдущем шаге).

Такой способ поиска кратчайшего пути требует определенной аккуратности, однако он не очень сложен.

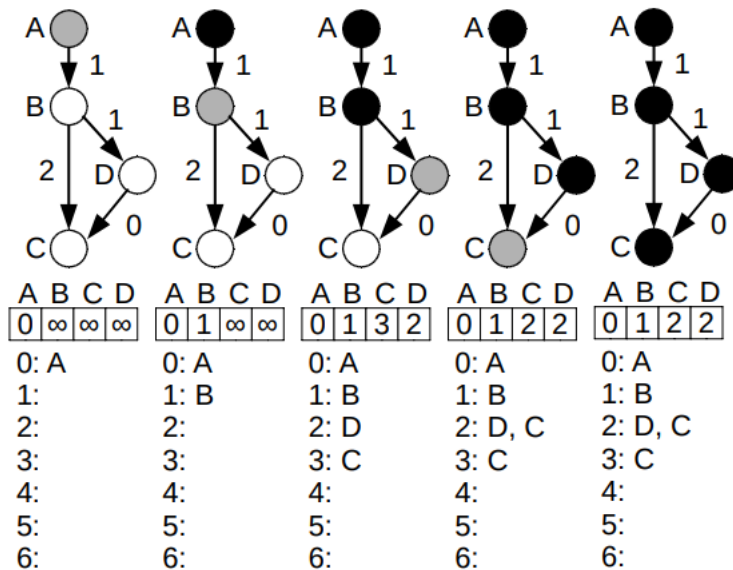
Альтернативой ему является использование аналога обхода в ширину с очередью. Здесь в случае наличия ребра весом ноль соседа необходимо добавлять не в конец, а в начало очереди (не забыв перед этим удалить обрабатываемую вершину из очереди). Очередь, в которой можно добавлять не только в конец, но и в начало, называется деком. При таком способе реализации обхода в ширину для 0-1 графа, вершина может дважды оказаться в деке и требуется соблюдать все те же меры по борьбе с проблемами, что и в случае использования списков вершин на заданном расстоянии.

3.5 Кратчайший путь на 0-K графе

Идею поиска кратчайшего пути на 0-1 графе можно расширить для 0-K графа, т.е. такого графа, где веса ребер могут только целыми числами от 0 до K включительно.

Эту задачу уже нельзя решить с помощью одной очереди или дека, поэтому удобнее пользоваться списками вершин, находящихся на заданном расстоянии. При этом нужно отметить, что максимальное расстояние в таком графе может быть равно $(V - 1) \times K$, это происходит, когда все вершины выстроены в ряд и соседи соединены ребрами весом K . Также в этом алгоритме могут возникать пустые списки вершин (т.е. может не быть вершин на расстоянии x , но могут быть вершины на расстоянии большем чем x), поэтому использовать пустоту очередного списка нельзя использовать в качестве критерия остановки алгоритма. Нуно придумать другой критерий остановки, например, всегда идти до списка с номером $(V - 1) \times K$ или запоминать максимальный номер списка, в котором есть вершины (его нужно обновлять при добавлении новых вершин). При этом использование в качестве критерия количества посещенных вершин нужно использовать аккуратно: в случае, если не все вершины достижимы из начальной, такой критерий может привести к вечному циклу и выходу за пределы массива.

На рисунке показан пример обработки 0-2 графа. Серым показана вершина, которая обрабатывается в данный момент.



Если вместо списков вершин, находящихся на заданном расстоянии, использовать, например, *ordered map*, где в качестве ключа будет расстояние до вершины, а в качестве значения — список номеров вершин, то можно добиться сложности $O(E \times \log K + V)$. Одновременно в *map* будет находиться не более K различных элементов (мы будем выбрасывать уже опустевшие списки, а никаких вершин на расстоянии более чем K от текущей быть в списках не может), а добавление вершины в какой-нибудь список будет происходить при обработке каждого из E ребер за $\log K$. Также в сумме будут обработаны все V вершин, что и даст нужную сложность.

Алгоритмы поиска на 0- K графе используются довольно редко, но в случае очень малых K могут дать некоторое незначительное преимущество по сравнению с универсальными алгоритмами поиска кратчайших путей во взвешенных графах.

3.6 Вершины и ребра на кратчайших путях

Рассмотрим две родственные задачи: найти все ребра и все вершины в графе, лежащие на каком-либо кратчайшем пути от вершины a до вершины b . Можно попытаться найти такие ребра и вершины, сохраняя длину пути от вершины a для каждой из вершин и двигаясь от вершины b обходом в ширину, который будет ходить по обратным ребрам и только в ситуации, когда расстояние, записанное для соседней вершины, на 1 меньше, чем для текущей.

Однако существует более изящное решение этой задачи. Запустим два обхода в ширину: от вершины a и от вершины b (в случае, если граф ориентированный, обход в ширину от вершины b должен идти по обратным ребрам) и для каждой вершины сохраним кратчайшее расстояние как от вершины a , так и от вершины b . Вершина x будет лежать на каком-либо кратчайшем пути от вершины a до b тогда и только тогда, когда длина пути от a до x в сумме с длиной пути от x до b совпадает с длиной пути от a до b . Таким образом, достаточно запустить два обхода в ширину, насчитать кратчайшие расстояния, перебрать все вершины графа и выбрать среди них подходящие.

Задача с ребрами решается также с помощью подсчета расстояний от a и до b для каждой вершины. Переберем все ребра, допустим очередное ориентированное ребро ведет из вершины v в вершину u . Ребро (v, u) лежит на каком-либо кратчайшем пути

тогда и только тогда, когда сумма расстояний от a до v и от u до b на единицу меньше, чем расстояние от a до b . Единицу прибавляет, собственно, ребро от v до u . В случае, если ребро неориентированное, его необходимо проверить как в порядке от v к u так и от u к v .

3.7 Кратчайшие пути в больших графах

В реальной жизни возникают очень большие графы, в которых нужно найти кратчайший путь от одной вершины до другой. Типичным примером такого графа является граф дружбы в социальной сети. Запрос всех соседей вершины в таком графе делается довольно медленно, через API социальной сети, а количество вершин в нем измеряется сотнями миллионов и даже миллиардами. Согласно теории шести рукопожатий расстояние между любыми двумя людьми в графе социальных связей не превосходит 6, а согласно теории Данбара количество прямых социальных связей примерно равно 200. То есть у каждой вершины есть примерно 200 соседей, на расстоянии 2 находится до 40000 соседей, на расстоянии 3 до 8 000 000 соседей и т.д. вплоть до 200^5 . Реально, общие друзья значительно снизят эти числа, но за 5 шагов придется обойти всю социальную сеть, что невозможно сделать физически.

В таких ситуациях применяется метод, когда шаги от начальной и конечной вершины делаются поочередно и на каждом шаге проверяется пересечение множеств вершин достижимых от каждой из начальных вершин.

В среднем от каждой из вершин придется совершить по 3 шага и всего будет обработано не более чем 2×200^3 вершин, что значительно ускорит работу такого алгоритма.

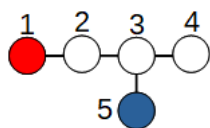
Эта идея может применяться и в других очень больших графах.

3.8 Кратчайшие пути в графах состояний

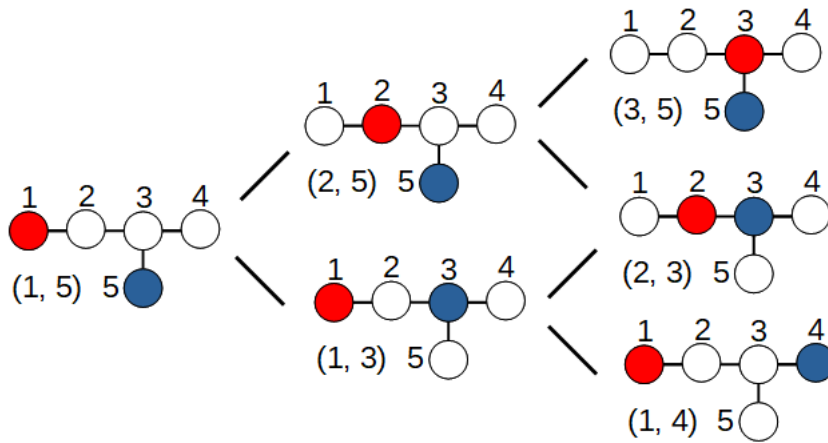
Существует большой класс задач, в которых состояние системы можно представить в виде вершины графа, а переход в другое состояние в виде ребра. В таком случае можно воспользоваться обходом в ширину, а в случае, если граф состояний очень большой — двумя обходами в ширину от начального и конечного состояния как в предыдущем разделе.

Рассмотрим на примере. Пусть двум машинам необходимо поменяться местами в узком дворе. Представим двор в виде графа, где вершины — это перекрестки, а ребра — проезды между ними. За один ход одна машина может переместиться в соседнюю вершину, при этом две машины не могут находиться в одной вершине одновременно.

На рисунке показан пример начального расположения вершин. Красная машина находится в вершине 1, а синяя — в вершине 5. Назовем это состоянием $(1, 5)$. Целевым состоянием является состояние $(5, 1)$.



В явном виде граф состояний обычно не строится, соседние состояния определяются динамически. Соседями состояния $(1, 5)$ будут состояния $(2, 5)$ и $(1, 3)$. Часть графа состояний для этой задачи представлена на следующем рисунке:



Количество вершин в графе состояний для этой задачи оценить достаточно легко: их не более чем V^2 . Оценка количества ребер немного сложнее: поочередно будем фиксировать первую из двух вершин исходного графа и перебирать вторую вершину. Всего будет V позиций для первой вершины и для каждой позиции первой вершины $V - 1$ позиция второй. При фиксированной первой вершине исходного графа для всех подходящих вершин графа состояний количество ребер будет равно $(V - 2) \times K + E$, где K — количество соседей первой вершины. Такая формула получается, т.к. в каждом из $V - 1$ состояний с первой фиксированной вершиной мы можем переместиться в состояние, где произошло изменение первой вершины на любого из K ее соседей, а вторая вершина поочередно будет принимать любое из $V - 1$ оставшихся после фиксации первой вершины и суммарное количество ребер, исходящих из этих вершин будет равно $E - K$. Перебирая все первые вершины, мы просуммируем их K и получим E . Таким образом, итоговая оценка на количество ребер будет равна $O(E \times V)$.

Лекция 4

Алгоритм Дейкстры

4.1 Поиск кратчайшего пути во взвешенном графе

С помощью модифицированного обхода в ширину нам удалось решить, в том числе, задачу о поиске кратчайшего пути во взвешенном графе с целочисленными весами вершин, не превосходящими K . Однако, такой подход неэффективен при больших K и не работает, например, для вещественных чисел.

Задачу поиска кратчайшего пути во взвешенном графе мы будем решать в следующей формулировке: задан взвешенный граф (неважно, ориентированный или нет) с ребрами неотрицательного веса, необходимо найти кратчайший путь от начальной вершины до всех остальных (на английском это называется *single-source shortest paths problem*). Требование неотрицательности ребер очень важно, т.к. рассматриваемый в лекции алгоритм не умеет работать с отрицательными ребрами.

Перед решением задачи, рассмотрим подробнее механизм хранения взвешенного графа для его удобной обработки. Удобнее всего использовать списки смежности, а каждое ребро записывать как пару чисел: номер соседней вершины и вес этого ребра. В случае использования матрицы смежности можно использовать таблицу, где в случае наличия ребра из вершины v в вершину u на пересечение строки номер v и столбца номер u будет записан вес этого ребра. При этом отсутствие ребра надо помечать каким-либо специальным признаком, например бесконечностью или числом -1 .

4.2 Алгоритм Дейкстры

В 1959 году голландский ученый Эдсгер Дейкстра (на английском фамилия пишется как *Dijkstra*) предложил следующий алгоритм поиска кратчайшего пути:

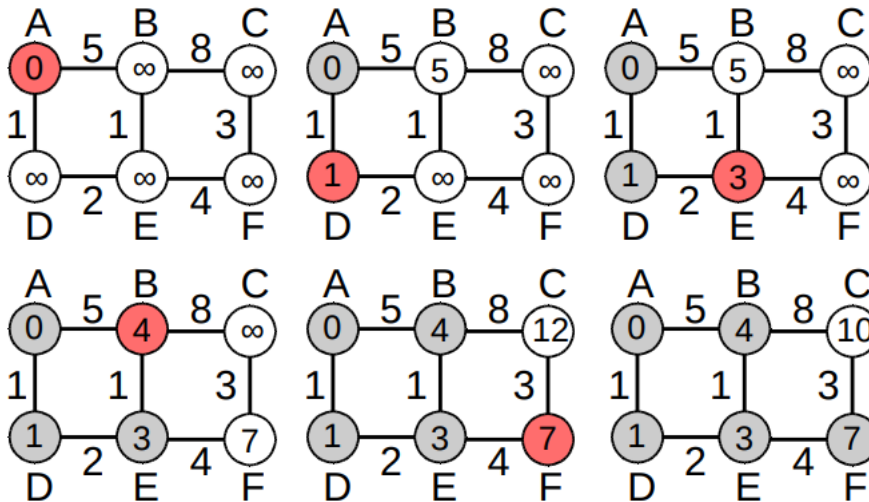
Заведем массив `dist` размером v (или $v + 1$ для более удобной нумерации вершин, начиная с единицы), в котором для каждой вершины будет храниться текущую длину кратчайшего пути от начальной вершины s . Изначально $d[s] = 0$, а для всех остальных вершин длина пути равна бесконечности (в качестве бесконечности следует выбирать число, большее чем вес максимального ребра, умноженного на $v - 1$). Также заведем массив пометок `visited` того же размера, в котором будут стоять пометки об обработке вершины. Изначально все вершины помечены как необработанные.

Алгоритм Дейкстры состоит из $v - 1$ шага. На каждом шаге происходит два действия:

1. Выбирается необработанная вершина v с минимальным расстоянием от начальной и помечается как обработанная.
2. Производится «релаксация»: рассматриваем все соседние с вершиной v вершины u (вес ребра (v, u) обозначим за len) и для соседней вершины u мы пытаемся улучшить значение $dist[u]$. То есть, записываем в $dist[u]$ минимум из текущего значения $dist[u]$ и $dist[v] + len$.

Если граф не является связным (или, в случае ориентированного графа, не все вершины достижимы из начальной), то на нескольких последних шагах в алгоритме Дейкстры будут выбираться вершины, расстояние до которых равно бесконечности. Эти действия никакой пользы не несут и алгоритм можно прервать досрочно. Если выбрать слишком большое число в качестве бесконечности, то при сложении с весом ребра может произойти переполнение, поэтому иногда проще прервать работу алгоритма, чтобы не столкнуться с этой проблемой.

На рисунке показан $v - 1$ шаг алгоритма Дейкстры и финальное состояние графа. Красным помечены вершины, для которых будет происходить релаксация на данном шаге, а серым выделены обработанные вершины.



В случае, если минимальное расстояние до непомеченной вершины ищется с помощью линейного поиска, а ребра хранятся с помощью списков смежности, сложность алгоритма составит $O(V^2 + E)$. На каждом из $V - 1$ шага нам придется искать минимум из V чисел для вершин и за все время работы каждое ребро будет просмотрено только один раз. Использование матрицы смежности не ухудшит асимптотическое время работы алгоритма, оно составит $O(V^2)$.

Восстановление пути в алгоритме Дейкстры делается аналогично тому, как это происходило для обхода в ширину. Можно либо пытаться обойтись без дополнительной информации, либо для каждой вершины хранить номер вершины-предка. Номер вершины-предка должен обновляться для соседней вершины, если удалось сократить путь до соседней вершины при релаксации.

4.3 Корректность алгоритма Дейкстры

Основное утверждение, которое нужно доказать для доказательства корректности алгоритма Дейкстры формулируется так: после того, как какая-либо вершина v ста-

новится помеченной, текущее расстояние до неё $\text{dist}[v]$ уже является наименьшим и не будет изменяться в дальнейшем.

Докажем по индукции. Для начальной вершины s $\text{dist}[s] = 0$, что и является кратчайшим путем. Допустим, что утверждение верно для всех предыдущих шагов алгоритма Дейкстры (то есть для всех уже помеченных вершин кратчайший путь найден верно). Пусть v — вершина, которая выбирается на очередном шаге алгоритма Дейкстры, необходимо доказать, что $\text{dist}[v]$ действительно является длиной кратчайшего пути до v .

Рассмотрим путь P от начальной вершины, до вершины v . Допустим, что он содержит непомеченную вершину u , отличную от v . Поскольку все ребра в графе неотрицательные, то $\text{dist}[v] \geq \text{dist}[u]$, что противоречит утверждению, что $\text{dist}[v]$ минимальное число среди всех непомеченных вершин (если же расстояния для v и u равны, то мы можем провести аналогичные рассуждения для u и найти для нее путь той же длины, не проходящий через v). Таким образом, путь до вершины v состоит только из уже помеченных вершин. Поскольку для всех помеченных вершин была проведена релаксация, то для вершины v был выбран самый короткий путь.

4.4 Алгоритм Дейкстры для разреженных графов

Рассмотренная ранее реализация алгоритма Дейкстры за $O(V^2 + E)$ хорошо подходит для плотных, почти полных графов. В реальной жизни гораздо чаще встречаются разреженные графы, в которых количество ребер значительно меньше, чем V^2 .

Сложность алгоритма складывается из двух частей: просмотр всех ребер за $O(E)$ и выбор минимальной вершины за $O(V)$ на каждом из $V - 1$ шагов. На все ребра так или иначе придется посмотреть, поэтому ускорять имеет смысл лишь ту часть алгоритма, в которой выбирается минимум.

Нам необходима структура данных, которая позволяет быстро находить минимум, а также достаточно быстро менять значение элемента (изменение расстояния до вершины происходит при соркащении пути во время релаксации). Одной из подходящих структур данных является *ordered set*: узнать минимум в нем можно за $O(1)$ (но после этого минимум надо удалить за $O(\log N)$), а изменить элемент (путем удаления старого значения и добавления нового) за $O(\log N)$. В *set*'е мы будем хранить пары из расстояния до вершины и ее номера. Таким образом, с использованием *ordered set* мы сможем добиться сложности $O(E \times \log V + V \log V)$ (для большинства графов, просто $O(E \times \log V)$) т.к. каждое из E ребер может привести к уменьшению пути и изменению значения в *ordered set*.

Основная сложность возникает при обновлении кратчайшего расстояния до вершины. Чтобы понять, требуется ли обновление расстояния до вершины, необходимо каким-то образом узнать старое расстояние до нее. Сделать это с помощью одного *ordered set* невозможно, поэтому нам придется дополнительно хранить массив, в котором для каждой вершины будет храниться текущее кратчайшее расстояние до нее. Если после релаксации появился более короткий путь, то необходимо удалить из *ordered set* пару из старого значения расстояния и номера вершины и добавить туда пару из нового расстояния и того же номера вершины. Также необходимо обновить расстояние до вершины в массиве расстояний по номеру.

Дополнительные затраты памяти на поддержку структур данных составят $O(V)$.

С учетом неасимптотической сложности более эффективным оказывается использование кучи минимумов. К сожалению, встроенный контейнер `priority_queue` не поддерживает операцию изменения элемента, однако, реализовав двоичную кучу вручную, можно добавить в нее операцию изменения элемента. Для этого достаточно поддерживать массив, в котором по номеру вершины в графе можно будет определить ее позицию в куче, а при изменении расстояния просеивать изменившийся элемент вверх. Асимптотическая сложность не изменится и по-прежнему будет составлять $O(E \log V)$, однако, константа станет меньше и можно ожидать ускорения работы программы примерно вдвое.

Еще один вариант состоит в использовании кучи без изменения элементов (например, `priority_queue`): старые значения пар «расстояние - номер вершины» можно не удалять из кучи, а просто игнорировать появления номера вершины в качестве минимума в том случае, если она уже была обработана. В таком случае размер кучи можно достигать E и сложность составит $O(E \log E)$, однако за счет низкой константы этот вариант оказывается более быстрым и может быть использован, если можно удовлетворить потребности в $O(E)$ памяти на хранение кучи. К тому же, поскольку $E \leq V^2$ и, следовательно, $\log E \leq 2 \times \log V$, то оценку сложности можно представить и как $O(E \log V)$ и единственной реальной проблемой этого метода становится лишь потребление памяти. При использовании `priority_queue` из STL следует помнить, что по умолчанию она реализует очередь максимумов и необходимо либо переопределять операторы сравнения, либо прибегать к каким-либо хитростям, например, добавляя в нее расстояние со знаком минус.

Довольно экзотическим вариантом является использование Фибоначчиевой кучи. Ее использование позволяет находить минимум за $O(\log V)$, а обновлять значение за $O(1)$. Таким образом можно достичь сложности $O(V \log V + E)$, однако очень большая константа делает такую реализацию на практике более медленной, чем использование альтернативной, более простой, структуры данных.

4.5 Вершины и ребра на кратчайших путях

Рассмотрим две родственные задачи: для каждой вершины (или ребра) необходимо определить, лежит ли оно на кратчайшем пути от вершины A до вершины B . Эта задача решается так же, как и задача для невзвешенного графа: для каждой из вершин посчитаем кратчайшее расстояние от вершины A до нее и от нее до вершины B . Для этого понадобится два запуска алгоритма Дейкстры: от вершины A и от вершины B (в случае, если граф ориентированные, перед запуском от вершины B все ребра необходимо развернуть).

Обозначим расстояние от вершины A до вершины v как $\text{distA}[v]$, от вершины v до вершины B как $\text{distB}[v]$. Длина пути от A до B при этом будет равна $\text{distA}[B]$. Вершина v лежит на каком-либо кратчайшем пути от A до B если $\text{distA}[v] + \text{distB}[v] == \text{distA}[B]$. Ребро (v, u) длиной len лежит на каком-либо кратчайшем пути, если $\text{distA}[v] + \text{distB}[u] + \text{len} == \text{distA}[B]$. Для неориентированных ребер необходимо перебрать два варианта ориентации этого ребра.

Посмотрим на одну задачу, в которой можно применить эту идею. Пусть сеть полетов авиакомпании состоит из V аэропортов и E односторонних перелетов между ними. Для каждого сегмента известна стоимость перелета. Вам необходимо наиболее деше-

вым способом попасть из аэропорта A в аэропорт B (возможно, с пересадками) и, при этом, у вас есть купон на один бесплатный перелет.

Интуитивная идея найти кратчайший путь и воспользоваться купоном на самом дорогом перелете на нем легко опровергается простым контрпримером: начальная и конечная вершины соединены прямым, но очень дорогим ребром и существует дешевый путь, например, по ребрам единичной стоимости. Правильным ответом будет число 0 (бесплатный прямой перелет), но предложенное решение выдаст другой ответ.

Наивный алгоритм решения состоит в поочередном обнулении всех ребер графа и поиска кратчайшего пути в оставшемся графе. То есть, мы поочередно перебираем сегмент, на который действует купон, делаем вес соответствующего ребра равным нулю и запускаем на этом графе алгоритм Дейкстры. Затем возвращаем старое значение, обнуляем очередное ребро и т.д. Сложность такого алгоритма составит $O(E^2 \log V)$, что довольно много.

Эту задачу можно решить быстрее, если использовать идею поиска ребер, лежащих на кратчайших путях. А именно, точно также насчитаем $\text{dist}A$ и $\text{dist}B$ для всех вершин, а затем переберем все ребра (v, u) . Если обнулить это ребро, то длина пути с использованием купона на бесплатный перелет составит $\text{dist}A[v] + \text{dist}B[u]$. Среди всех таких сумм необходимо выбрать минимальную. В итоге решение задачи будет состоять из двух запусков алгоритма Дейкстры и перебора всех ребер, сложность составит $O(E \log V)$.

Еще один способ состоит в «раздваивании» вершин. А именно, каждую вершину исходного графа v следует разделить на две: «с купоном» и «без купона», обозначим их за v^0 и v^1 . Каждое ребро исходного графа (v, u, len) превратится в три ребра: (v^1, u^1, len) , (v^0, u^0, len) и $(v^1, u^0, 0)$. То есть можно за деньги переместиться по ребру сохраняя состояние купона или воспользоваться купоном и из состояния наличия купона попасть в соседнюю вершину без купона, зато бесплатно. В таком графе с помощью одного запуска алгоритма Дейкстры нужно найти путь из состояния A^1 в состояние B^0 . Сложность такого алгоритма составит $O(E \log V)$, в явном виде новый граф (по крайней мере, его ребра) можно не строить.

4.6 Задача о самом широком пути

С помощью модификаций алгоритма Дейкстры можно решать не только задачу о кратчайшем пути в графе, но и целый класс задач о путях, с ребрами, обладающими определенным свойством. Доказывать корректность для каждой задачи нужно по той же схеме, по которой доказывается корректность алгоритма Дейкстры для кратчайших путей, но все эти задачи объединяет одно свойство: характеристика пути не может стать лучше в процессе работы алгоритма (например, путь до вершины не может стать короче в будущем, если речь идет о задаче поиска кратчайшего пути).

Рассмотрим следующую задачу: необходимо найти путь от вершины A до вершины B , такой, чтобы минимальное ребро на нем было как можно большим. На английском эта задача называется *maximum bottleneck edge problem* — задача о максимальном бутылочном горлышке. Она имеет прикладное значение, например, для логистики, когда мы хотим перевезти какой-то очень тяжелый груз в условиях ограничений на вес машины на дорогах.

Рассуждения в этой задаче такие же, как в алгоритме Дейкстры. Вместо массива

расстояний dist будем использовать массив bottleneck , который означает вес минимального ребра на пути. Для начальной вершины значение в этом массиве будет равно бесконечности, для всех остальных вершин — 0. Релаксацию будем проводить следующим образом: если вершина v уже обработана и мы попадаем в u по ребру (v, u, len) , то $\text{bottleneck}[u] = \min(\text{bottleneck}[v], \text{len})$. И если в обычном алгоритме Дейкстры мы старались минимизировать суммарную величину пути, то при поиске самого широкого пути нужно среди всех вариантов попасть в вершину выбирать максимальный. При этом на каждом шаге необходимо выбирать ту непомеченную вершину, значение bottleneck для которой максимально.

4.7 Второй по величине путь

Ежедневно, садясь в свою машину и прокладывая на навигаторе маршрут, мы получаем не только самый быстрый вариант проезда, но и несколько альтернатив. Научимся искать второй по величине простой путь в графе (то есть путь без циклов).

Для поиска второго по величине пути в первую очередь необходимо найти первый, самый короткий путь и запомнить все ребра, по которым он проходит. Второй путь отличается от первого хотя бы по одному ребру. Будем поочередно удалять из графа ребра первого пути по одному (т.е. на каждом шаге из исходного графа удалено ровно одно ребро кратчайшего пути) и искать на оставшемся графе кратчайший путь. Среди всех найденных путей выберем минимальный — он и будет вторым по величине путем. Этот путь будет отличаться от первого хотя бы по одному ребру, а его минимальность доказывается от противного.

4.8 К-ый по величине путь

Идею поиска второго по величине пути можно обобщить для поиска K -го кратчайшего пути. Алгоритм поиска K -го кратчайшего пути называется алгоритмом Йена (Yen's algorithm).

Наша задача состоит в том, чтобы найти K -ый по длине путь от вершины s до вершины t . Будем поддерживать два множества путей: A^i , в котором хранятся i первых кратчайших путей, и множество B , в котором хранятся потенциальные кратчайшие пути.

Множество A^1 состоит из одного элемента — кратчайшего пути, найденного, например, с помощью алгоритма Дейкстры. После первого запуска алгоритма Дейкстры множество B пусто.

Пусть мы уже построили $K - 1$ путь (они хранятся в множестве A^{K-1}) и хотим построить K по величине путь. Нам необходимо, чтобы этот путь отличался от всех путей в множестве A^{K-1} .

Чтобы сгенерировать все пути, которые отличаются от первых $K - 1$ пути, мы возьмем $K - 1$ по величине путь и с помощью переменной i будем перебирать, сколько первых вершин из $K - 1$ по величине пути мы оставляем в графе. Вырежем первые i вершин из $K - 1$ пути и назовем их rootPath , а i по счету вершину в $K - 1$ пути назовем точкой разветвления (spurNode). Поскольку путь должен отличаться от всех уже найденных $K - 1$ пути, то мы переберем все уже найденные пути и выберем из них те,

первый i вершин которых совпадают с первыми i вершинами $K - 1$ пути (их начало совпадает с `rootPath`), обозначим множество этих путей за C . Наш новый путь не должен совпадать ни с одним из них, поэтому для каждого пути из C удалим в графе ребро, соединяющую i по счету вершину пути (`spurNode`) со следующей, $i + 1$ вершиной в этом пути. Таким образом мы сможем гарантировать, что на $i + 1$ шаге наш путь будет отличаться от всех уже найденных. Чтобы избежать появления путей с циклами, удалим все из графа все вершины из `rootPath`. На оставшемся после удаления ребер и вершин графе мы найдем кратчайший путь от `spurNode` до конечной вершины t (назовем этот путь как `spurPath`). Таким образом, весь путь будет состоять из `rootPath` + `spurPath`. Добавим этот путь в множество путей-кандидатов B . Восстановим все удаленные ребра и вершины и продолжим итерировать i .

После окончания этого шага выберем самый короткий путь из B — он и будет K по величине путем.

Доказательство корректности этого алгоритма мы приводить не будем.

Сложность одного запуска алгоритма Дейкстры составляет $O(E \log V)$. Для поиска кратчайших путей нам потребуется $K \times l$ запусков алгоритма Дейкстры, где l — длина `spurPath`. Поскольку длина `spurPath` гарантированно не превышает V , то можно оценить сложность всего алгоритма Йена как $O(KV \times E \log V)$.

Также алгоритм Йена потребляет довольно много дополнительной памяти: нам необходимо хранить множества путей A и множество путей B . Множество B может быть довольно большим, однако, мы можем воспользоваться тем, что суммарный размер множеств A и B может не превосходить K (нам не нужны пути, длиннее, чем K -ый). Поскольку длина пути не превосходит V , то оценка сверху на дополнительную память составляет $O(KV)$.

Лекция 5

Алгоритмы Флойда и Форда-Беллмана

5.1 Алгоритм Флойда

Помимо задачи поиска кратчайшего пути от одной вершины до всех остальных существует задача поиска кратчайших путей от каждой вершины графа до каждой. Мы уже можем решить эту задачу вызвав алгоритм Дейкстры для каждой из V вершин, асимптотическая сложность составит $O(V^3)$ или $O(V \times E \log V)$ в зависимости от типа реализации алгоритма Дейкстры. Однако, для решения задачи поиска кратчайших путей от каждой вершины до каждой существует гораздо более простой в написании алгоритм.

Алгоритм Флойда (Floyd) был предложен в 1962 году одновременно Флойдом и Уоршеллом (Warshall), поэтому иногда называется алгоритмом Флойда-Уоршелла. Реально первым похожий алгоритм предложил Рой (Roy) в 1959 году, но его публикация осталась незамеченной.

Реализация алгоритма Флойда очень проста: мы будем постепенно превращать матрицу смежности графа в матрицу кратчайших расстояний между вершинами. Можно разбить алгоритм Флойда на V шагов, при этом перед шагом номер k в ячейки матрицы $\text{dist}[\text{from}][\text{to}]$ будет храниться длина кратчайшего пути из from в to , проходящего только через промежуточные вершины с номерами, меньшими чем k (начало и конец пути могут быть любыми, не только меньшими k).

Перед первым шагом dist будет полностью соответствовать матрице смежности графа — в ней нет никаких промежуточных вершин. В матрице смежности удобно пометить отсутствующие ребра бесконечностью.

При переходе от $k - 1$ шага к k нам необходимо пересчитать матрицу расстояний, добавив в нее кратчайшие пути, проходящие через вершину k . При пересчете кратчайшего пути из from в to возможны два случая:

1. кратчайший путь от from до to не проходит через вершину k и остается неизменным
2. кратчайший путь от from до to проходит через вершину k . В таком случае разобьем путь от from до to на две части: от from до k и от k до to . Каждая из этих частей посчитана корректно на предыдущих шагах алгоритма. Длина нового пути в таком случае будет равна $\text{dist}[\text{from}][k] + \text{dist}[k][\text{to}]$

Очевидно, что для подсчета кратчайшего пути нужно просто выбрать лучший из двух вариантов. Таким образом, мы можем получить из матрицы кратчайших расстояний с использованием промежуточных вершин до $k - 1$ матрицу расстояний с использованием вершин до k за время $O(V^2)$ (перебор всевозможных from и to). Для хранения всех матриц размеров V^2 на V шагах алгоритма потребуется V^3 памяти. Можно хранить лишь две матрицы для предыдущего и текущего шагов, тогда потребление памяти сократится до $O(V^2)$. Однако, все действия можно проводить и на одной матрице, на ответ это не повлияет.

Таким образом, весь алгоритм Флойда выглядит так:

```
for (size_t k = 1; k <= v; ++k)
    for (size_t from = 1; from <= v; ++from)
        for (size_t to = 1; to <= v; ++to)
            dist[from][to] = min(dist[from][to], dist[from][k] + d[k][to]);
```

Обратите внимание, что матрица имеет размер $(V + 1) \times (V + 1)$ для реализации нумерации. Также необходимо очень аккуратно выбирать значение бесконечности, чтобы не возникло переполнения при сложении бесконечности с весом ребра.

5.2 Восстановление пути

Получившаяся в результате применения алгоритма Флойда матрица позволяет определить длину пути между двумя вершинами, но пользуясь только ей восстановить путь довольно сложно. Поэтому для восстановления пути намного удобнее сохранять некоторую вспомогательную информацию. Есть два способа хранения вспомогательной информации: хороший и похуже. Начнем со способа похуже.

Довольно часто в интернете можно найти предложение хранить для каждой пары вершин from и to промежуточную вершину — ту самую, через которую последний раз был сокращен путь. То есть, в ситуации, когда путь через промежуточную вершину k оказался более коротким, чем текущий путь от from до to , то в специальном двумерном массиве *via* будем сохранять номер промежуточной вершины: $\text{via}[\text{from}][\text{to}] = k$. Сначала весь массив *via* должен быть заполнен нулями — это будет признак того, что вершины соединены прямым ребром и никаких промежуточных вершин на нем нет. При восстановлении пути от from до to мы будем рекурсивно восстанавливать путь от from до k и, затем, от k до to . Во многих источниках написано, что рекурсивная функция восстановления «несложная» и это действительно так, однако, ее реализация требует большой аккуратности, чтобы избежать, например, дублирования вершин. В худшем случае длина пути будет составлять $O(V)$ и при определенных обстоятельствах глубина рекурсии также составит $O(V)$. Например, когда одна первая состоит из одного ребра, а вторая из всех оставшихся на пути вершин, и подобная ситуация возникает на каждом шаге рекурсии.

Хороший способ восстановления пути пользуется заметно меньшей популярностью, хоть и опирается на классическую логику восстановления путей в графах в частности и восстановления решений задач динамического программирования вообще. В нем также используется матрица p того же размера, что и матрица смежности. При этом $p[\text{from}][\text{to}]$ хранит номер предпоследней вершины на пути от from до to . Изначально для каждого ребра исходного графа (from, to) в матрице записывается $p[\text{from}][\text{to}] = \text{from}$. И действительно, в случае пути по одному ребру, предпоследней вершиной на этом пути

является начальная вершина. Во время работы алгоритма Флойда в ситуации, когда путь через промежуточную вершину k оказался короче существующего пути из from в to , то мы должны заменить значение $p[\text{from}][\text{to}]$. Путь из from в to состоит из первой части из from в k и второй части из k в to . При этом предпоследняя вершина пути из from в to совпадает с предпоследней вершиной второй части пути (из k в to) и достаточно просто сделать присваивание $p[\text{from}][\text{to}] = p[k][\text{to}]$. Когда возникает необходимость восстановить путь, то это делается стандартным образом, начиная с конца. Вершина from при восстановлении не изменяется, а to на каждом шаге заменяется на предпоследнюю вершину пути, т.е. $\text{to} = p[\text{from}][\text{to}]$. Повторять эти шаги необходимо до тех пор, пока значение to не достигнет from . Шагов, по-прежнему, может потребоваться до V , а также необходимо $O(V)$ памяти для хранения и последующего разворота последовательности номеров вершин на пути.

Из хорошего способа довольно легко получить почти идеальный: достаточно хранить не предпоследнюю вершину на пути, а вторую. В таком случае можно избавиться от необходимости разворачивать последовательность вершин, а в случае, когда их сразу можно выводить ответ по одному числу, и вовсе избавиться от необходимости хранить эту последовательность. Такой способ восстановления пишется значительно проще рекурсивного восстановления пути, имеет меньшую константу как по времени, так и по памяти и, при определенных обстоятельствах, позволяет и вовсе обойтись $O(1)$ дополнительной памяти непосредственно во время восстановления пути.

5.3 Транзитивное замыкание графа

Перейдем к невзвешенному ориентированному графу, заданному матрицей смежности. Для него существует похожая на поиск кратчайших путей, но более простая задача: для каждой вершины найти множество достижимых из нее вершин.

Задачу о транзитивном замыкании графа можно решить точно таким же способом, как и задачу о кратчайших путях, однако если в случае кратчайшего пути мы выбирали минимум из существующего пути и пути через промежуточную вершину, то в случае транзитивного замыкания нам достаточно того, чтобы существовал хотя бы один из путей (либо уже существует путь из from в to , либо существует путь из from в k и путь из k в to).

В случае, если в матрице смежности наличие ребра обозначалось единицей, а отсутствие — нулем, мы можем построить матрицу достижимости просто написав три цикла как в алгоритме Флойда. Внутри циклов достаточно написать $d[\text{from}][\text{to}] = d[\text{from}][k] \text{ ||| } (d[\text{from}][k] \ \&\& \ d[k][\text{to}])$.

Мы можем использовать для хранения матрицы смежности и матрицы достижимости не двумерные массивы, состоящие из переменных типа `bool`, а их сжатые представления. То есть, для хранения каждого `true` или `false` мы будем использовать не один байт, а всего один бит внутри 32 или 64-битной переменной. Это даст нам экономию памяти в 8 раз.

Кроме экономии памяти можно добиться значительного роста производительности. Ведь мы можем работать одновременно не только с одним битом числа, а сразу со всеми 32 или 64 — для этого достаточно использовать побитовое и и или. Этим мы добьемся очень большого роста производительности, хоть он и не будет асимптотическим.

5.4 Применимость алгоритма Флойда

Задача определения длины кратчайшего пути за $O(1)$ (с любым предподсчетом) может быть решена для довольно специфичного класса графов. Независимо от того, каким образом мы подсчитали кратчайшие пути, для их хранения потребуется $O(V^2)$ памяти, что невозможно для больших графов, таких как карта или социальная сеть.

Кроме того, время подсчета также важно: для алгоритма Флойда оно составит $O(V^3)$, а для V алгоритмов Дейкстры $O(V \times E \log V)$. В случае разреженных графов использование алгоритма Дейкстры может быть более предпочтительно.

При этом алгоритм Флойда обладает одной очень приятной особенностью: на очередном шаге внешнего цикла по k все данные внутри этого цикла независимы. То есть, порядок перебора *from* и *to* не имеет абсолютно никакого значения. А значит, имея V^2 вычислительных устройств, умеющих совершать примитивные операции сложения и выбора минимума, мы можем реализовать шаг алгоритма Флойда за $O(1)$ и итоговая сложность составит всего $O(V)$. Алгоритм Дейкстры не удастся распараллелить в той же степени, т.к. в нем нужно каждый раз дожидаться выполнения предыдущего шага.

Современные видеокарты содержат в себе тысячи исполнительных устройств, способных выполнять операции сложения и выбора минимума, что позволяет неасимптотически ускорить алгоритм Флойда в сотни и даже тысячи раз. Таким образом, проблема с вычислительной сложностью алгоритма в $O(V^3)$ может быть в значительной степени преодолена и основной проблемой становится потребление памяти в $O(V^2)$.

5.5 Отрицательные ребра

Ребра с отрицательным весом встречаются довольно редко и в случае, когда вес ребра обозначает расстояние или время проезда по ребру не имеют физического смысла. Однако, существуют задачи, где отрицательный вес ребер может иметь смысл. Например, если вес ребра — это стоимость бензина для проезда на машине, но на некоторых ребрах мы можем взять попутчиков, которые оплатят свой проезд, то отрицательный вес ребер имеет смысл, когда вместо расхода мы получаем прибыль.

Алгоритм Флойда не имеет требования к обязательной неотрицательности ребер и будет корректно искать кратчайшие пути в графах с отрицательными весами ребер. Но в графах с циклами отрицательного веса длина кратчайшего пути может быть минус бесконечностью и задача поиска кратчайшего пути теряет смысл.

Таким образом, необходимо научиться диагностировать в графе отсутствие циклов отрицательного веса (если их нет, то пути будут подсчитаны корректно).

Здесь необходимо вспомнить, что цикл — это путь из вершины в саму себя. Таким путям соответствуют клетки на главной диагонали матрицы кратчайших расстояний, т.е. клетки вида $\text{dist}[i][i]$. Если перед запуском алгоритма Флойда заполнить эти клетки нулями, то после выполнения всех шагов в случае наличия цикла отрицательного веса, в который входит определенная вершина, значение в клетке, соответствующей ей, станет отрицательным.

Восстановить сам цикл можно стандартным методом для восстановления пути.

5.6 Алгоритм Форда-Беллмана

В некоторых случаях нам необходимо решить задачу поиска кратчайшего пути от одной вершины до всех остальных на разреженном графе. В такой ситуации использование алгоритма Флойда нецелесообразно и, в случае отсутствия отрицательных ребер в графе следует пользоваться алгоритмом Дейкстры. Но если в графе есть отрицательные ребра, алгоритм Дейкстры не сможет найти кратчайшие пути даже в случае, если в графе нет циклов отрицательного веса.

Для этих целей можно использовать алгоритм Форда-Беллмана (Ford-Bellman), разработанный в 1956-1958 годах. Этот алгоритм также как и алгоритм Дейкстры использует релаксацию ребер, но делает это по-другому.

Перед работой алгоритма нам необходимо создать массив кратчайших расстояний dist , где для каждой вершины, кроме начальной, будет записана бесконечность, а для начальной вершины будет записано число 0.

Для поиска кратчайшего пути нам потребуется $V-1$ шаг алгоритма Форда-Беллмана. На каждом из шагов мы будем перебирать все ребра и проводить релаксацию по этому ребру.

То есть, если в графе есть ребро (v, u) длиной len , то мы заменим $\text{dist}[u]$ на $\min(\text{dist}[u], \text{dist}[v] + \text{len})$.

Итоговая сложность алгоритма составит $O(V \times E)$, что заметно хуже, чем у алгоритма Дейкстры, но заметно лучше, чем у алгоритма Флойда (в случае, если граф разреженный).

5.7 Доказательство корректности

Для доказательства корректности алгоритма Форда-Беллмана докажем, что после k шагов алгоритма будут корректно найдены все пути, длина которых по числу ребер не превосходит k .

Доказательство будем проводить по индукции. Перед началом алгоритма расстояния определены верно. Пусть для каждой вершины верно рассчитано кратчайшее расстояние для $k-1$ шага. Рассмотрим произвольную вершину v и множество вершин U , из которых можно попасть в v по ребру. Поскольку мы перебираем все ребра графа, то будут рассмотрены все ребра из всех вершин U в вершину v и выбран кратчайший путь. Таким образом, для вершины v будет определен кратчайший путь, содержащий не более k ребер.

5.8 Оптимизации и отрицательные циклы

Поскольку максимальная длина пути в графе по количеству ребер не может превосходить $V-1$, то нам необходимо и достаточно выполнить $V-1$ шаг алгоритма Форда-Беллмана. Однако, в реальных графах кратчайшие пути редко включают в себя все вершины и их длина по ребрам значительно меньше, чем $V-1$.

Алгоритм Форда-Беллмана определяет кратчайший путь, содержащий не более k ребер. Поэтому если после очередного шага алгоритма не произошло никаких изменений в массиве dist , то мы можем остановить алгоритм. Для этого достаточно завести переменную типа bool , которую в начале шага устанавливать в false , а в случае, если

произошла хотя бы одна релаксация — устанавливать в true. Это позволит значительно сократить количество шагов алгоритма для большинства графов и очень незначительно ухудшит константу для худшего случая.

При этом нельзя полагаться исключительно на эту переменную, сигнализирующую о наличии изменений в массиве расстояний, все равно нужно совершать максимум $V - 1$ шаг. При наличии в графе циклов отрицательного веса, достижимых из начальной вершины, релаксации будут происходить и на шагах после $V - 1$. С помощью этого факта можно проверять наличие достижимых циклов отрицательного веса: достаточно совершить еще один шаг алгоритма Форда-Беллмана после $V - 1$ шага и, если произошла хотя бы одна релаксация, то обязательно существует достижимый цикл отрицательного веса.

Для восстановления путей в алгоритме Форда-Беллмана используется стандартный прием с запоминанием для каждой вершины номера той вершины, из которой мы попали в текущую. Этот номер меняется в случае релаксации по ребру (v, u) , для вершины u запоминается предыдущая вершина v .

С помощью этой информации можно восстановить цикл отрицательного веса. Действительно, достаточно запомнить номер хотя бы одной вершины, для которой произошла релаксация на шаге номер V . Эта вершина будет либо лежать на цикле отрицательного веса, либо быть достижима из вершин, лежащих на таком цикле. Достаточно восстановить V предков этой вершины и найти среди них ту вершину, которая встречается как минимум дважды. Все вершины, лежащие от первого вхождения этой вершины до второго и будут составлять цикл.

Такой способ позволяет найти достижимый цикл отрицательного веса, но можно модифицировать алгоритм таким образом, чтобы он искал любой цикл отрицательного веса. Для этого достаточно объявить все вершины начальными — на задачу поиска цикла отрицательного веса это никак не повлияет.

5.9 Информация для размышления

Мир алгоритмов поиска кратчайших путей в графах велик и удивителен и охватить его за три лекции невозможно.

Существует множество оптимизаций алгоритма Форда-Беллмана, которые не входят в наш курс алгоритмов. Если вы заинтересованы в них, то ключевые слова для поиска:

- Алгоритм Левита (Levit's algorithm) — обратите внимание, что реализация с сайта e-maxx работает за экспоненциальное время в некоторых случаях
- Pallottino's algorithm
- Goldfarb-Hao-Kai algorithm
- Goldberg-Radzik algorithm

Особого внимания заслуживает алгоритм Джонсона: с помощью алгоритма Форда-Беллмана определяются потенциалы вершин, которые позволяют модифицировать веса ребер и избавиться от отрицательных ребер. После этого можно запускать алгоритм Дейкстры и на графе с отрицательными ребрами, что значительно лучше в случае разреженных графов.

Алгоритмы поиска кратчайшего пути в графах с отрицательными ребрами очень востребованы при решении задачи поиска максимального потока минимальной стоимости, который используется для решения задачи о назначениях, в некоторых алгоритмах машинного обучения и других отраслях народного хозяйства.

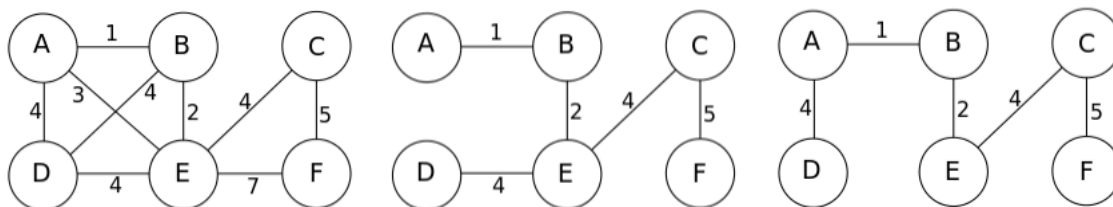
Лекция 6

Минимальное остовное дерево и СНМ

6.1 Минимальное остовное дерево

Минимальным остовным деревом (Minimum Spanning Tree, MST) связного взвешенного неориентированного графа называется множество ребер этого графа, такое что по этим ребрам существует путь от любой вершины до любой и их суммарный вес минимален.

В графе может существовать больше одного минимального остовного дерева. Например, если все ребра имеют одинаковый вес, то любое остовное дерево (то есть соединяющее все вершины) будет минимальным. На рисунке показан пример графа и двух его минимальных остовных деревьев.



Понятие минимального остовного дерева имеет смысл только для графа с положительными весами ребер, т.к. в случае наличия ребер отрицательного веса подмножество ребер минимального веса может не быть деревом (в нем допустимы циклы, полностью состоящие из ребер отрицательного веса).

В случае, если все ребра имеют различный вес, то минимальное остовное дерево единственно. Этот факт доказывается несложно: допустим, существует два различных минимальных остовных дерева, тогда выберем такое ребро e_1 с минимальным весом, которое присутствует в одном дереве, но отсутствует в другом. Договоримся называть дерево, содержащее ребро e_1 первым. Добавим это ребро во второе дерево, в результате чего в нем образуется цикл. В этом цикле обязательно найдется ребро e_2 , которое принадлежит второму дереву, но не принадлежит первому (если бы такого ребра не нашлось, то весь цикл целиком должен был быть в первом дереве, что невозможно, т.к. в дереве нет циклов). Поскольку e_2 не содержится в первом дереве, то его вес больше, чем вес e_1 (т.е. e_1 минимально среди всех ребер, не содержащихся хотя бы в одном дереве). Таким образом, мы можем исключить ребро e_2 и заменить его ребром e_1 во втором дереве, в результате чего вес дерева уменьшится, что противоречит тому, что второе дерево является минимальным остовом.

Также из интересных свойств остовных деревьев:

- минимальный остов также является остовом с минимальным произведением весов ребер (доказывается через замену весов ребер на их логарифмы)
- минимальный остов также является остовом с минимальным весом самого большого ребра (удобно доказывать через алгоритм Крускала)
- легко найти максимальный остов — достаточно заменить знаки у весов ребер

Задача поиска минимальных остовных деревьев имеет важное практическое значение для планирования линий электропередач, каналов связи и т.п. Впервые алгоритм поиска минимального остовного дерева предложил чехословацкий математик Отокар Боровка (Borůvka) в 1926 году для планирования линий электропередач в Моравии. В 1930 году другой чехословацкий математик, Войтех Ярник (Jarník) предложил алгоритм, впоследствии переоткрытый Примом (Prim) в 1957 году и Дейкстрой в 1959 году (он получил название «алгоритм Прима», но иногда в его названии упоминается и Ярник). В 1956 году был предложен алгоритм Крускала (Kruskal).

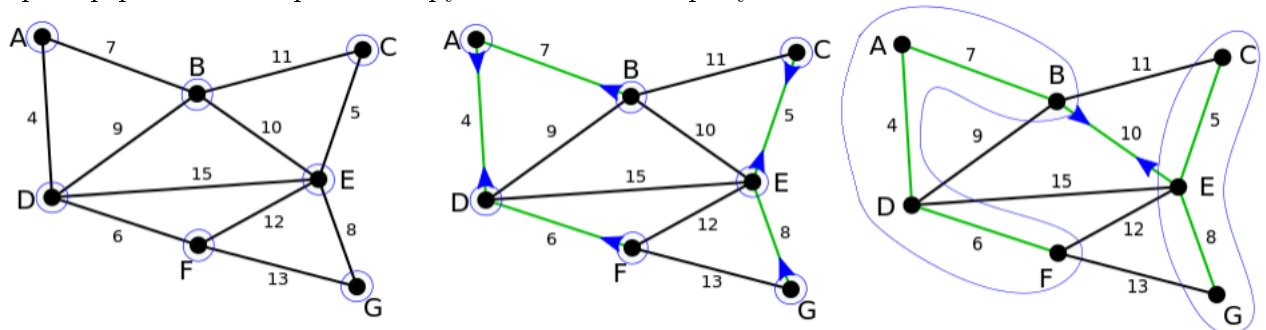
6.2 Алгоритм Боровки

Алгоритм Боровки переоткрывался много раз, в том числе в 1965 году Соллином (Sollin) и иногда встречается под названием «алгоритм Соллина», особенно часто в параллельном программировании.

Сначала в алгоритме для каждой вершины выбирается минимальное инцидентное ей ребро и все эти ребра добавляются в ответ, в результате чего получается лес. Затем для каждого получившегося дерева выбирается минимальное ребро, то есть ребро, инцидентное одной из вершин дерева, второй конец которого не входит в это дерево. За один шаг алгоритма количество деревьев уменьшается как минимум вдвое и процесс заканчивается, когда осталось всего одно дерево. Количество деревьев может уменьшаться и быстрее, в два раза оно уменьшается только в случае, когда для двух деревьев было выбрано одно и то же ребро, соединяющее именно их.

Если веса ребер не уникальны, то может случиться ситуация, когда образуется цикл. Самый простой пример: треугольник, в котором вершины соединены ребрами веса 1. При неудачном выборе минимальных ребер для каждой компоненты все три ребра попадут в ответ на первом шаге. Чтобы избежать этой ситуации достаточно выбирать среди всех минимальных ребер ребро с минимальным номером (при этом можно обойтись просто списком ребер) или ввести какой-либо другой порядок просмотра ребер — главное, чтобы он был одинаковым для всех вершин и прямые и обратные ребра обрабатывались одновременно.

Пример работы алгоритма Боровки показан на рисунке:



На каждом шаге алгоритма Борувки можно заново проводить разметку на деревья, используя для этого обычный обход в глубину, который ходит только по уже добавленным в дерево ребрам. Граф в виде списков смежности с добавленными ребрами также можно строить на каждом шаге заново. Результирующая сложность составит $O(E \log V)$.

Схема доказательства корректности алгоритма Борувки совпадает со схемой доказательства существования единственного минимального остовного дерева для графа с уникальными весами.

6.3 Алгоритм Прима

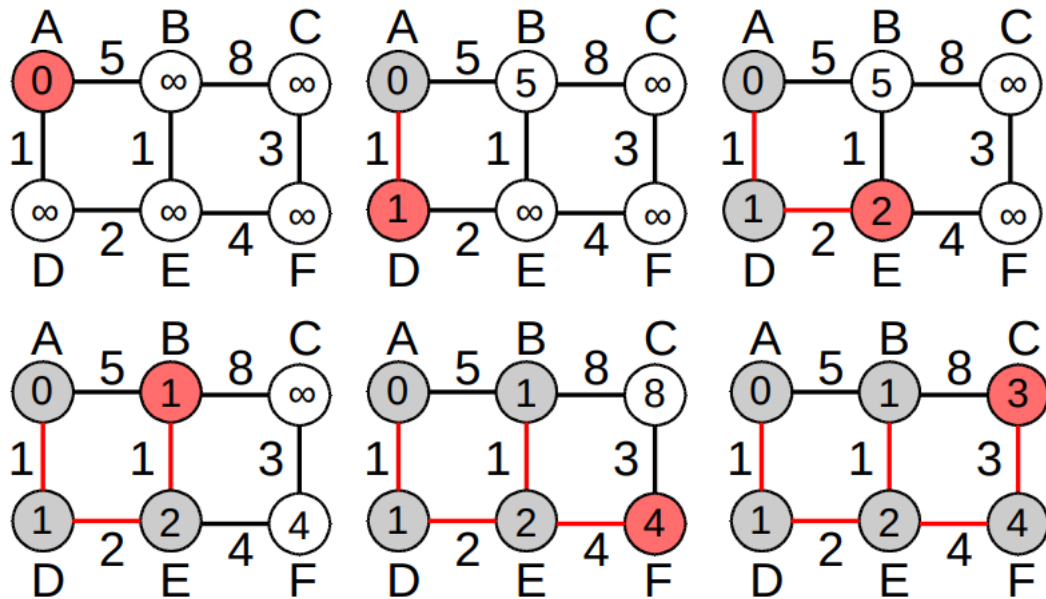
Алгоритм Прима строит минимальный остов, начиная с одной вершины и, постепенно, расширяя множество вершин, для которых уже построено минимальное остовное дерево.

Перед началом выполнения алгоритма Прима можно выбрать произвольную вершину в графе. На первом шаге перебираются все ребра, исходящие из этой вершины, и выбирается ребро с минимальным весом. Минимальный остов содержит уже две вершины и теперь ищется минимальное ребро, один из концов которого является одной из двух вершин, а другой, наоборот, не является ни одной из двух этих вершин.

Таким образом, на каждом шаге алгоритма Прима имеется множество вершин, минимальный остов для которых уже построен, и к этому множеству добавляется вершина не из этого множества, соединенная ребром с какой-либо вершиной из множества, причем вес ребра, соединяющий эту вершину с какой-либо вершиной из множества, наименьший среди всех таких ребер (если таких несколько — можно взять любое из ребер).

Всего необходимо совершить $V - 1$ шаг. При этом для каждой вершины графа, не входящей в множество, для которого построен остов, удобно хранить число, обозначающее минимальный вес ребра, соединяющий эту вершину с одной из вершин множества. При добавлении очередной вершины в множество достаточно пересчитать эти расстояния только для вершин, еще не входящих в множество и смежных с добавленной.

Пример работы алгоритма Прима показан на рисунке:



Доказательство алгоритма Прима опирается на ту же идею, что и доказательство единственности минимального остова для графа с уникальными весами. Рассмотрим дерево, построенное алгоритмом Прима T и минимальное остовное дерево S . Если деревья различны, то в T найдется ребро e_1 не входящее в S . Обозначим концы этого ребра за u и v , а множество вершин, входящих в остов перед добавлением ребра e_1 за X . Тогда u находится в X , а v — нет. В дереве S вершины u и v соединяются каким-либо путем, найдем на нем ребро e_2 , такое, что один конец этого ребра лежит в множестве X , а другой — нет. Такое ребро обязательно существует, т.к. u лежит в X , а v — нет, а значит какое-то из ребер на пути из u в v переходит из X в не- X . Алгоритм Прима выбирает минимальное ребро и выбрал ребро e_1 , а значит вес ребра e_2 больше либо равен весу ребра e_1 . Если вес e_2 больше веса e_1 , то при добавлении ребра e_1 в дерево S образуется цикл, содержащий ребра e_1 и e_2 , мы удалим из него ребро e_2 , уменьшим суммарный вес ребер в S и придем к противоречию. Если же вес ребер равен, то алгоритм Прима просто построил другое минимальное остовное дерево с тем же весом, что и минимальное.

Несложно заметить, что алгоритм Прима похож на алгоритм Дейкстры: на каждом шаге выбирается минимальная вершина из множества непомяченных и производится «релаксация» ребер. Отличие состоит лишь в том, что в алгоритме Дейкстры мы поддерживали для каждой вершины длину кратчайшего пути от начальной вершины, а в алгоритме Прима — длину ребра, соединяющего вершину с одной из вершин множества. Таким образом, различие алгоритмов Дейкстры и Прима заключается лишь в одной строке: там где при релаксации ребра из вершины v в u , длиной len , в Дейкстре мы записывали $dist[u] = \min(dist[u], dist[v] + len)$, в алгоритме Прима достаточно написать $dist[u] = \min(dist[u], len)$. Таким образом, все методы, использованные для алгоритма Дейкстры, могут быть использованы и для алгоритма Прима, и его сложность можно оценить как $O(V^2)$ для плотного графа, $O(E \log V)$ для разреженного графа с использованием кучи или ordered set и $O(V \log V + E)$ при использовании Фибоначчиевой кучи.

Обратите внимание, что алгоритм Прима ни в какой момент не требует положительности ребер для доказательства и будет строить корректные минимальные остовные

деревья в графах, в которых нет циклов со всеми отрицательными ребрами.

6.4 Система непересекающихся множеств

Алгоритм Крускала требует для своей реализации структуры данных, называемой системой непересекающихся множеств, сокращенно СНМ (Disjoint Set Union, DSU). Эта структура бывает полезна и в некоторых других алгоритмах и задачах.

Пусть у нас есть N элементов, занумерованных числами от 1 до N и изначально каждый из них находится в своем множестве (их мы также занумеруем от 1 до N). Нам необходимо реализовать две операции:

- $\text{find}(x)$ — найти номер множества, к которому относится элемент x
- $\text{union}(x, y)$ — объединить множества, содержащие элементы x и y

В классическом варианте есть и третья операция $\text{make}(x)$, создающая новое множество, содержащее элемент x , но нам она не понадобится, а ее добавление к рассматриваемым нами вариантам реализации СНМ довольно очевидно.

Оценивать эффективность реализации СНМ мы будем по тому, какое количество операций потребуется для объединения всех элементов в одно множество в худшем случае.

6.5 Наивная реализация СНМ

Простейший способ хранения СНМ заключается в создании массива с индексами от 1 до N , где индекс обозначает номер элемента, а содержимое — номер множества, к которому относится этот элемент. Операция $\text{find}(x)$ в этом случае очевидна: достаточно вернуть содержимое ячейки с номером x , ее сложность $O(1)$.

Операция union , однако, будет работать довольно медленно. Для объединения множеств, содержащих элементы x и y достаточно узнать номера множеств, содержащих эти элементы (пусть они оказались a и b соответственно), затем пройти по всему массиву и заменить значения b на a . Одна операция объединения множеств имеет сложность $O(N)$, а объединение всех множеств в одной займет $O(N^2)$ времени.

6.6 Списки элементов множества

Очевидная оптимизация этого метода состоит в том, чтобы для каждого множества хранить список элементов этого множества — это избавляет от необходимости проходить по всему массиву и искать элементы, входящие в множество b . Однако, в худшем случае сложность всех операций по-прежнему будет равна $O(N^2)$. Для этого достаточно постепенно расширять множество, каждый раз приписывая его к множеству, состоящему из одного элемента. Таким образом, в среднем одна операция объединения будет занимать $O(N)$.

Избавиться от этой проблемы поможет хранение длин списков и присоединение короткого списка к длинному. Тогда вариант с постепенным прибавлением элементов по одному к длинному списку будет работать за $O(N)$ в сумме, но есть другой худший

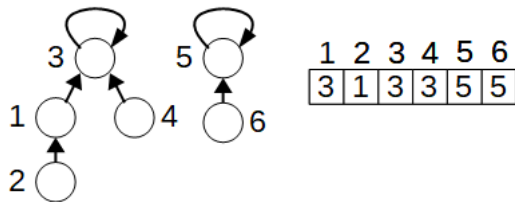
вариант: оптимизация не будет срабатывать, когда списки имеют равную длину. Если всегда объединять списки равной длины, то нам потребуется $O(\log N)$ шагов и на каждом шаге $O(N)$ для объединения множеств. Итоговая сложность объединения всех множеств составит $O(N \log N)$.

6.7 Корневые деревья

Еще один способ хранения множеств позволяет обойтись без массива с номером множества. Этот способ заключается в том, что каждое множество будет храниться в виде дерева, корень которого называется «представителем». Для каждого элемента множества нужно хранить ссылку на его предка, а корень дерева (представитель) будет хранить ссылку на самого себя.

Реализовать корневое дерево можно на одномерном массиве: индекс будет являться номером элемента, а значение — номером предка.

На рисунке показан пример деревьев и содержимое массива предков для них:

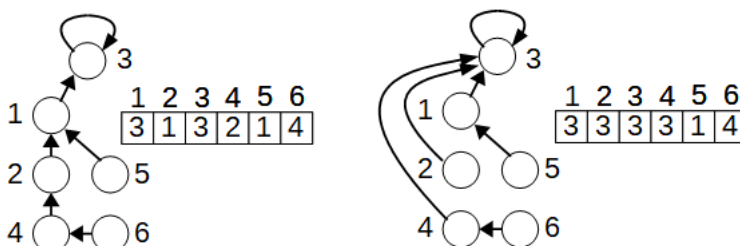


Для реализации операции `find` необходимо дойти до представителя множества, номер которого и будет определять номер множества. Это можно сделать просто с помощью цикла `while` идя до тех пор, пока номер предка не совпадет с текущим номером. Для реализации операции `union` необходимо дойти до представителей множеств обоих элементов и подвесить одного к другому.

Такая реализация множеств неэффективна, можно придумать пример, где выполнение операций `find` и `union` будут происходить за $O(N)$, а объединение всех элементов в одно множество займет $O(N^2)$. Для ускорения работы этой структуры необходимо применить две эвристики.

Первая эвристика называется «сжатие путей». Эта эвристика очень простая: при вызове `find` запомним все пройденные вершины и, после того как узнаем представителя, для всех пройденных вершин установим в качестве предка представителя. Это наглядно делается с помощью рекурсивной функции, но также можно использовать цикл `while` для прохода до корня, `vector` для запоминания всех вершин и еще один цикл для переподвешивания всех вершин на пути напрямую к корню дерева. Существенной разнице в производительности нет.

На рисунке показано дерево до сжатия путей и после вызова `find(4)`:



Применение этой эвристики ведет к тому, что среднее время выполнения операции

`find` становится равным $O(\log N)$. Доказательство этого факта можно прочитать на сайте `e-maxx`.

Вторая эвристика похожа по своей идее на объединение списков: а именно, нужно подвешивать дерево с меньшей глубиной к дереву с большей глубиной. Для этого для каждого корня дерева необходимо хранить его ранг (глубину), а поскольку в начале работы алгоритма каждый элемент хранится в отдельном дереве, то разумно хранить ранги деревьев в массиве. Эта оптимизация **отдельно** от сжатия путей делает сложность выполнения операции `find` равной $O(\log N)$.

Сочетание двух эвристик делает сложность операции `find` равной $O(\log * N)$ (где $\log * N$ — «итерированный логарифм», количество операций логарифмирования, необходимых для того, чтобы получить из N единицу). Доказательство этого факта можно найти в Кормене. Существует и более строгая оценка сложности, а именно $O(\alpha(N))$, где $\alpha(N)$ — обратная функция Аккермана. Для всех разумных чисел, соизмеримых с памятью современных компьютеров, значение обратной функции Аккермана не превосходит 4, так что при разумных ограничениях мы можем говорить, что `find` работает за $O(1)$.

Впервые доказательство оценки сложности реализации СНМ снизу и сверху обратной функцией Аккермана было доказано Тарьяном в 1975 и с тех пор продолжается увлекательный процесс поиска и исправления ошибок в этом доказательстве и попытки доказать сложность $O(1)$.

6.8 Алгоритм Крускала

Несмотря на сложные рассуждения, реализация обеих функций СНМ занимает 3-5 строк и сложность выполнения этих функций на практике можно считать равной $O(1)$.

Это позволяет эффективно использовать СНМ при реализации алгоритма Крускала для поиска минимального остовного дерева, который формулируется очень просто: отсортируем все ребра по возрастанию и будем брать в остовное дерево те ребра, которые соединяют разные множества вершин. Изначально каждая вершина находится в своем множестве, а при добавлении ребра происходит объединение этих множеств.

Доказывается алгоритм Крускала стандартно, с отличающимся ребром и образованием цикла.

6.9 Заметки о СНМ

Вместо эвристики с рангом дерева (на самом деле за счет сжатия путей ранг хранит верхнюю оценку на глубину дерева) часто можно встретить эвристику с размером дерева — эта характеристика не изменяется от сжатия путей и, на практике, работает не хуже. Еще один способ, позволяющий сэкономить память — полный отказ от ранговой эвристики и случайный выбор того, какое дерево будет присоединяться к другому. Этот способ также работает хорошо на практике.

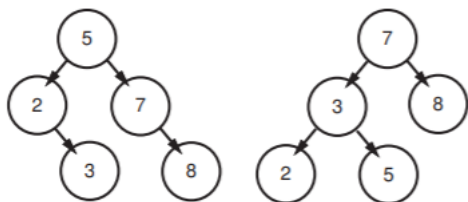
СНМ может применяться для решения задачи о подсчете связных областей одноцветных пикселей на изображении, в таком случае даже не обязательно хранить изображение целиком — достаточно всего двух последовательных строк изображения и СНМ, что значительно экономит память в случае, если строки изображения поступают последовательно.

Лекция 7

Бинарные деревья поиска

7.1 Бинарное дерево поиска

Во втором модуле уже был рассмотрен пример бинарного дерева, обладающего некоторым свойством (кучу). Теперь рассмотрим бинарное дерево с другим свойством: левый сын имеет ключ, меньший, чем в данном узле, а правый сын — больший. Такие деревья называются бинарными деревьями поиска. Из этого свойства следует, что все элементы в левом поддереве меньше данного элемента, а в правом, наоборот, больше. Обычно, в бинарных деревьях поиска нет двух элементов с одинаковыми ключами, но если это условие не выполнено, то можно либо в каждом узле хранить количество элементов с таким ключом или добавлять его в произвольное поддерево (иногда строго в правое поддерево). Ключами в бинарном дереве поиска могут быть любые сравнимые на больше-меньше элементы.



На рисунке приведены примеры двух бинарных деревьев поиска. Как видно, деревья могут отличаться, хотя и содержат одинаковые элементы. Это зависит от порядка добавления элементов, хотя для разных порядков добавления могут получиться одинаковые деревья.

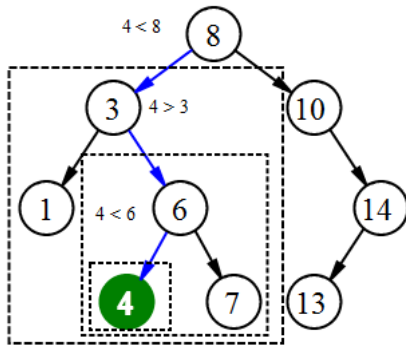
В бинарном дереве поиска легко найти минимальный и максимальный элемент: это самый левый и самый правый элементы (для их нахождения нужно идти от корня «до упора влево» и «до упора вправо» соответственно).

Обозначим максимальную глубину бинарного дерева за H . В случае, если элементы добавлялись в случайном порядке $H = O(\log N)$. Бинарное дерево поиска позволяет искать элемент по ключу за $O(H)$ в среднем (как и бинарный поиск в отсортированном массиве) и, в отличие от отсортированного массива, позволяет производить вставку элемента за $O(H)$ в среднем (для отсортированного массива эта операция занимает $O(N)$). Кроме того, из бинарного дерева поиска можно за $O(N)$ получать отсортированный массив. Также существуют такие полезные операции с бинарным деревом поиска, как поиск порядковой статистики за $O(H)$ или определение порядкового номера элемента по значению также за $O(H)$.

Для хранения узла дерева необходима структура, с полем для ключа, а также с двумя ссылками на левого и правого сыновей.

Для поиска элемента в бинарном дереве поиска удобно использовать рекурсивную функцию, которая сначала вызывается для корня дерева. Если ключ в текущем элементе совпадает с искомым, то нужно вернуть ссылку на текущий узел. Если искомое значение меньше ключа в текущем узле, то нужно вызвать функцию от левого поддерева, а если больше — от правого, и вернуть результат, возвращенный этой функцией. Если в какой-то момент необходимо вызвать функцию от несуществующего ребенка, то следует вернуть специальный признак того, что не нашлось элемента с искомым ключом. Сложность функции поиска также составит $O(H)$.

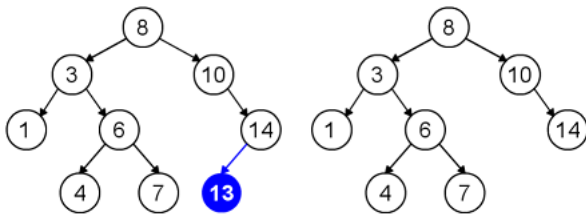
На рисунке показан процесс поиска ключа 4:



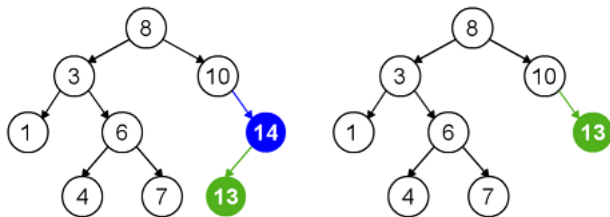
Вставка элемента в дерево осуществляется точно так же, как поиск, но при обнаружении отсутствия у элемента нужного потомка осуществляется вставка добавляемого элемента на его место.

Несколько более интересно удаление элемента из дерева.

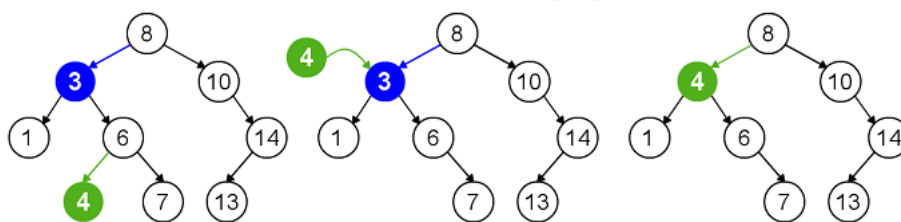
Листья удаляются очень просто, путем обнуления ссылки у их предка.



Узлы с одним потомком также удаляются несложно — в предке ссылка перекидывается на этого потомка.



Несколько сложнее дело обстоит с удалением узла, у которого два потомка. Для этого нужно запомнить ключ из самого левого потомка в правом поддереве (или из самого правого потомка в левом поддереве), затем удалить этого потомка (он либо не будет иметь потомков, либо будет иметь всего одного потомка, иначе он не может оказаться самым левым или правым) и записать ключ в текущий узел.



7.2 Обход дерева поиска

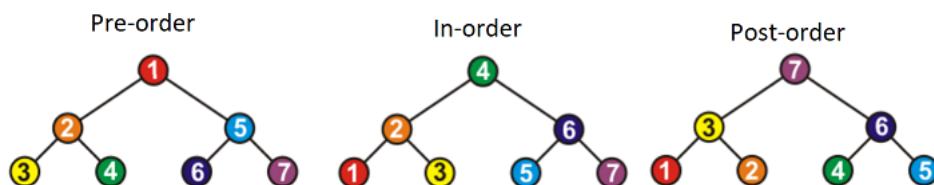
Существует три основных способа обойти дерево поиска, один из которых имеет практический смысл. На русский язык их названия переводятся самым причудливым образом, поэтому будем использовать английские названия. Все эти способы реализуются с помощью рекурсивных функций и имеют сложность $O(N)$ и потребляют $O(H)$ памяти.

Pre-order — сначала мы выводим значение ключа в текущем узле, затем посещаем сначала левого сына, а потом — правого.

In-order — сначала мы посещаем левого сына, затем выводим значение ключа в текущем узле, а затем посещаем правого сына:

Post-order — сначала посещаем левого сына, затем правого, а затем выводим значение ключа в текущем узле.

На рисунке числами обозначен порядок обхода узлов при разных способах обхода:



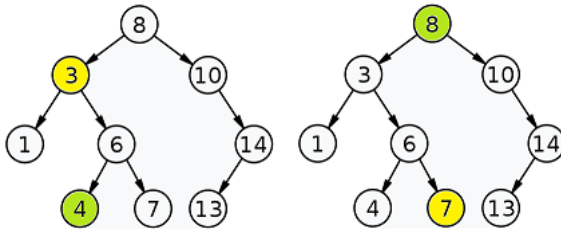
In-order обход дерева даст нам возрастающую последовательность ключей, так что в некотором смысле можно считать это методом сортировки. Можно изменить порядок посещения сыновей в In-order обходе и сначала посещать правого, а затем — левого. В таком случае у нас получится упорядоченная по убыванию последовательность.

7.3 Следующий и предыдущий элементы

Найчимся искать следующий по величине элемент, если у нас уже есть ссылка на текущий узел. Для облегчения этой задачи удобно хранить в структуре, описывающей узел дерева, не только ссылки на детей, но и на родителя.

Если у узла есть правое поддерево, то следующий по величине элементом будет являться самый левый узел правого поддерева. Если же правого поддерева нет, то нам нужно подниматься вверх до тех пор, пока не окажется, что текущий узел является левым сыном своего предка (т.е. мы переходим к предку, если являемся его правым сыном и идем до тех пор, пока не станем левым сыном). Предок такого узла и будет являться первым большим элементом.

На рисунке желтым показан текущий элемент, а зеленым — следующий за ним, для случая наличия правого поддерева и его отсутствия:



Перыдущий элемент ищется полностью аналогично: необходимо взять самый правый элемент в левом поддереве, а в случае его отсутствия — подниматься вверх до тех пор, пока не окажется, что текущий узел является правым сыном своего предка.

Поиск следующего элемента в худшем случае осуществляется за $O(H)$, однако сложность в среднем составит $O(1)$. Действительно, если мы начнем перебирать все элементы, начиная с минимального, с помощью перехода к следующему по величине элемента, то каждый узел дерева будет посещен всего дважды: при проходе «вниз» когда мы двигались через него при наличии правого поддерева и при проходе «вверх», когда уже обработаны все узлы в поддереве, корнем которого он является. В итоге мы потратим $O(N)$ операций при обходе всех узлов.

В случае, если мы не храним ссылку на предка мы все равно можем найти следующий элемент по значению ключа. Пусть мы ищем первое значение, большее чем x . Запустим обычную функцию поиска по ключу в бинарном дереве, запоминая наименьшее значение, большее x на пути. Если у узла дерева с ключом x есть правое поддерево, то мы просто ищем в нем самый левый элемент. Если же правого поддерева нет, то ответом будет запомненное значение. Такой способ поиска следующего элемента имеет сложность $O(H)$.

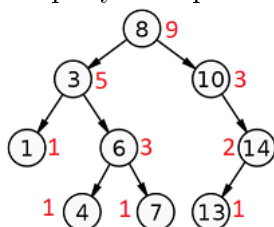
7.4 Определение элемента по номеру

С помощью хранения дополнительной информации можно научиться находить порядковую статистику в дереве, а также определять порядковый номер элемента по значению.

Для этого нужно поддерживать размер поддеревьев с корнем в каждом из узлов. При добавлении нового элемента нам нужно прибавить единицу к счетчику размеров поддеревьев во всех узлах, находящихся на пути от корня до этого элемента. Это легко сделать прибавляя единицу к размеру поддерева при проходе через узел. Делать это можно как на рекурсивном спуске, так и на рекурсивном подъеме.

В случае удаления элемента необходимо пересчитывать размер каждого поддерева для каждого из узлов на пути как сумму размеров двух поддеревьев и единицы. Делать это нужно в конце рекурсивной функции удаления элемента, когда поддерева уже обработаны.

На рисунке красным указан размер поддеревьев



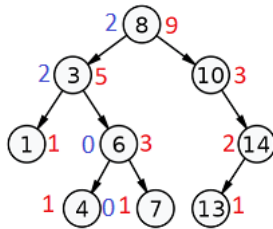
Обозначим за *leftsize* размер левого поддерева текущего узла (если левого подде-

рева нет, то будем считать $leftsize$ равным нулю). При поиски k -ой порядковой статистики в бинарном дереве поиска с подсчитанными размерами поддеревьев необходимо действовать следующим образом:

- если $k = leftsize$, то ключ текущего узла и есть k -я порядковая статистика
- если $k < leftsize$, то будем искать k -ю порядковую статистику в левом поддереве
- если $k > leftsize$, то будем искать порядковую статистику с номером $k - leftsize - 1$ в правом поддереве

Этот способ поиска порядковой статистики основывается на том, что в левом поддереве все элементы меньше текущего, а в правом — больше. Если $k < leftsize$, то мы можем исключить из рассмотрения текущий элемент и все элементы с ключом, больше чем у текущего (т.е. исключить сам элемент и все его правое поддерево). Если $k = leftsize$, то в рассматриваемом поддереве ровно k меньших элементов и текущий элемент и является искомой порядковой статистикой. Если же $k > leftsize$, то мы можем исключить из рассмотрения первые $leftsize + 1$ элемент (все правое поддерево и текущий элемент) и искать порядковую статистику только в правом поддереве. При этом надо вычесть из числа k количество исключенных из рассмотрения элементов.

Допустим, мы хотим вторую (при нумерации с нуля) порядковую статистику в дереве с рисунка. Значения k в каждом узле подписаны синим, будет найден элемент с ключом 4:



Также размеры поддеревьев можно использовать для определения порядкового номера элемента по его значению. Достаточно в обычной функции поиска по значению в бинарном дереве поиска прибавлять $leftsize + 1$ каждый раз, когда мы переходим направо. Действительно, при переходе в правого сына мы понимаем, что искомое значение больше, чем в текущем узле и чем в любом узле левого поддерева, а количество таких значений как раз равно $leftsize + 1$. При переходе левого сына ничего делать не нужно, ведь мы не исключаем никаких элементов.

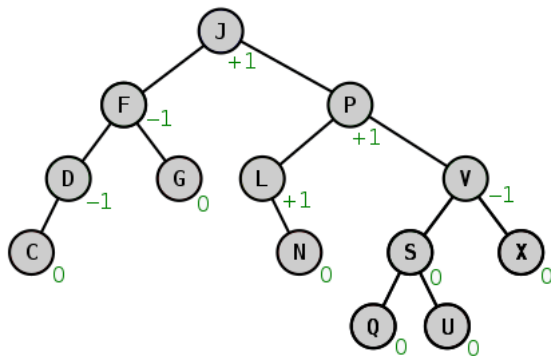
Сложность обоих этих алгоритмов равна $O(H)$.

7.5 АВЛ-дерево

В случае, если элементы добавляются в бинарное дерево поиска в случайном порядке, то оно получится сбалансированным, т.е. $H = \log N$. Но если добавлять элементы, например, по возрастанию, то дерево вырождается в одну длинную ветвь направо и H будет равно N . Чтобы избежать этой ситуации существует несколько алгоритмов балансировки дерева, позволяющих сохранять его высоту пропорциональной $\log N$.

Впервые алгоритм балансировки предложили советские ученые Георгий Адельсон-Вельский и Евгений Ландис в 1962 году. В честь них такое дерево называется AVL-дерево.

Для того чтобы поддерживать дерево сбалансированным, требуется менять его структуру при возникновении слишком длинных нераздваивающихся ветвей. В AVL-дереве каждому узлу запрещено иметь потомков, высоты поддеревьев, корнями которых они являются, различаются больше чем на 1. На самом деле, глубину в явном виде можно не хранить: достаточно для каждого узла знать «баланс»: разницу высот для правого и левого поддеревьев. На рисунке показано корректное AVL-дерево, где каждому узлу дополнительно приписан баланс:



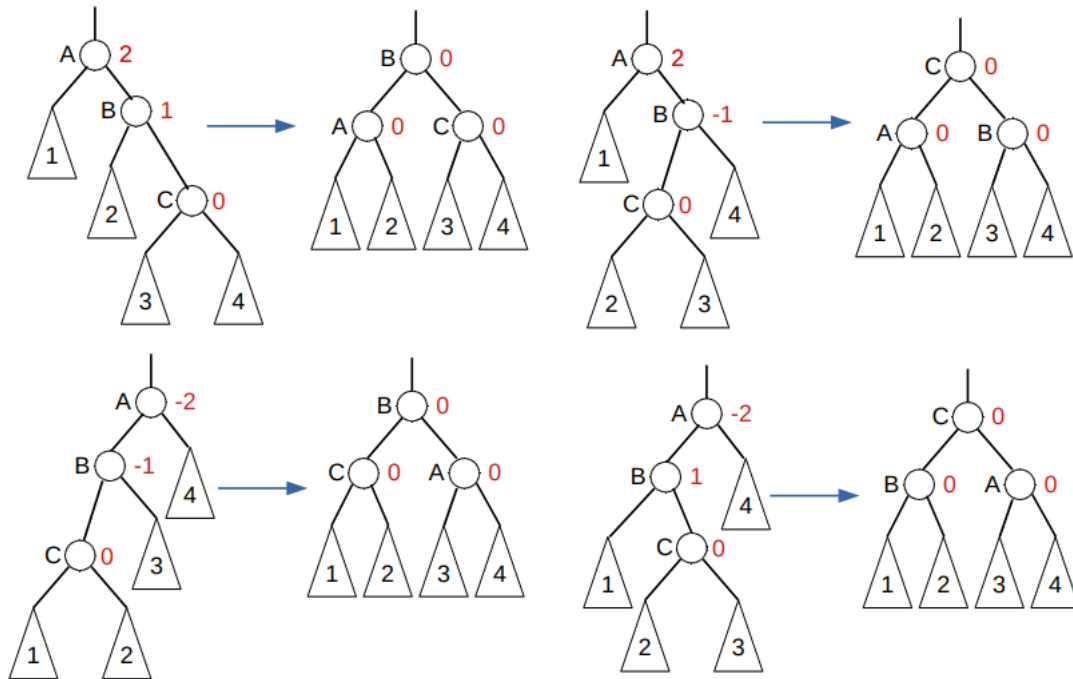
Поскольку AVL-дерево является бинарным деревом поиска, то поиск в нем происходит совершенно обычным образом, только этот процесс занимает $O(\log N)$.

Добавление вершины влечет за собой пересчет балансов и, при необходимости, балансировку. Рассмотрим подробнее этот процесс.

При добавлении новой вершины она становится листом и ее баланс равен нулю. Функция добавления значения должна возвращать булевское значение, которое будет истинным, если глубина поддерева увеличилась. Очевидно, что при добавлении листа функция должна вернуть истину, т.к. высота была равна нулю, а стала равна единице.

После выхода из функции, добавившей лист, мы будем возвращаться по дереву, двигаясь к его корню. Допустим, очередной элемент был добавлен в левое поддерево. Если функция вернула ложь, то балансировки не требуется, ведь глубина поддерева не изменилась, а дерево до этого было сбалансированным. Если же функция вернула истину (размер левого поддерева увеличился), то необходимо рассмотреть несколько вариантов, некоторые из которых могут вызвать потребность в балансировке. Если баланс вершины был равен +1 (правое поддерево было на 1 глубже), то баланс вершины станет равным нулю (левое поддерево увеличилось). Балансировка при этом не требуется. Если баланс был 0, то он станет равным -1, но это допустимая ситуация и балансировка по-прежнему не требуется. Если же баланс уже был равен -1, то он станет равным -2 и потребуются балансировка. Полностью аналогичные рассуждения можно провести для добавления в правое поддерево.

На рисунке треугольниками обозначены поддеревья одинаковой высоты, а красным шрифтом — баланс для каждой из вершин. Всего возможно возникновение четырех случаев нарушения балансы и показаны примеры вращения дерева, которые сохраняют свойство дерева поиска и делают баланс по модулю не превышающем единицы:



Вообще говоря, баланс для вершины C не обязательно равен нулю: одно из прикрепленных к ней поддеревьев может быть на единицу меньше, чем другое. При этом вес другого поддерева обязательно должен совпадать с весом поддерева, прикрепленным к вершине B . То есть до поворота вершина C может также иметь баланс 1 или -1, что приведет к тому, что и после поворота некоторые вершины будут иметь баланс 1 или -1. Но это не противоречит свойству АВЛ-дерева.

Максимальная глубина поддеревьев в АВЛ-дерева может отличаться не более чем в 2 раза.

При удалении элемента из АВЛ-дерева также может нарушиться баланс. Восстанавливается он с помощью таких же поворотов.

Лекция 8

Декартово дерево

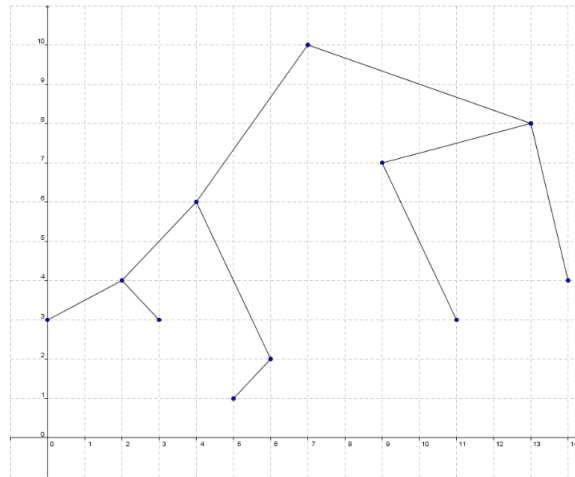
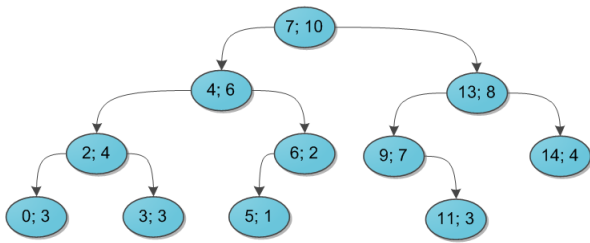
8.1 Декартово дерево

Декартово дерево имеет к Декарту еще меньшее отношение, чем Фибоначчиева куча, такое название появилось из-за удобного отображения дерева с использованием декартовых координат x и y (то есть самых обычных координат на плоскости). Декартово дерево использует два ключа, по одному из которых оно является бинарным деревом поиска, а по-другому — обычной двоичной кучей. По-английски Декартово дерево называется *Cartesian tree*, но именно для изучаемой структуры данных более точное название — *treap* (сокращение от *tree* и *heap*). На русском Декартово дерево также иногда называется «дерاميда» (от «дерево» и «пирамида») или «дуча» и это тоже более точные определения. В целом Декартово дерево (*Cartesian tree*) несколько более широкое понятие, чем *treap*, но в русском языке под Декартовым деревом чаще всего понимают именно *treap*.

Декартово дерево было впервые описано Raimund Seidel и Cecilia R. Aragon в 1989 году.

Его суть заключается в следующем: каждый узел бинарного дерева (обозначим его ключ за x) дополняется случайным числом y . При этом по ключам x образуется бинарное дерево поиска, а по y указанная структура должна обладать свойством обычной двоичной кучи максимумов (то есть каждый элемент больше своих потомков). Поиск в Декартовом дереве осуществляется точно так же, как и в любом другом двоичном дереве, а вставка и удаление реализуется особым образом.

На рисунке показано представление одного и того же дерева в обычном представлении и в декартовых координатах:



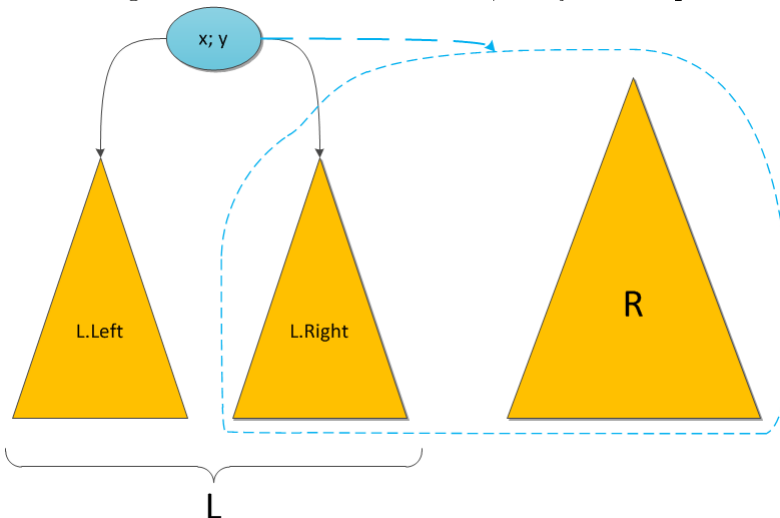
При добавлении второго ключа y представление дерева становится однозначным. Например, правым сыном корня будет самая верхняя вершина, находящаяся справа от корня, а левым — самая верхняя вершина слева от корня.

Из-за того, что координаты y выбираются случайно дерево получается сбалансированным и его высота имеет порядок $O(\log N)$.

8.2 Merge и Split

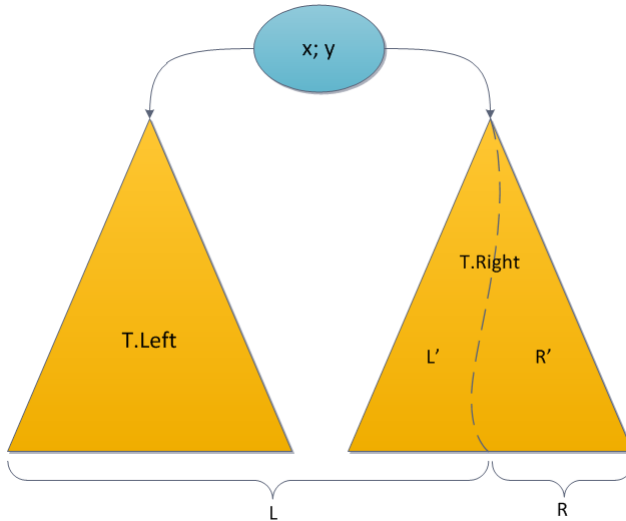
Пусть у нас существует два Декартовых дерева, назовем их «левое» L и «правое» R . Пусть все ключи x в дереве L меньше ключей x дерева R . Мы хотим реализовать операцию Merge для слияния двух этих деревьев в одно Декартово дерево T .

Что будет являться корнем итогового дерева T ? Узел с наибольшим значением ключа y . Кандидатов на это только два: корни деревьев L и R . Допустим, что большее значение ключа y оказалось у корня дерева L . Очевидно, что левым поддеревом корня должно остаться левое поддерево дерева L . В правом поддереве корня дерева T должны оказаться все элементы из правого поддерева корня дерева L , а также все элементы дерева R . Объединить два эти дерева можно с помощью рекурсивного вызова Merge для этих деревьев. Когда одно из объединяемых деревьев становится пустым, результатом работы Merge становится оставшееся, непустое дерево.



Теперь рассмотрим операцию *Split*. Это операция должна разделить Декартово дерево T по ключу x_0 так, чтобы в результате этой операции получилось два Декартовых дерева L и R , таких, что в дереве L содержатся узлы с ключами x меньше x_0 , а в R — с ключами больше x_0 . В случае, если в дереве есть элемент с ключом равным x_0 , его можно отнести к дереву L или R в зависимости от задачи.

Если корень дерева T имеет ключ x меньше чем x_0 , то корень должен оказаться в дереве L , а если больше x_0 — в дереве R . Допустим, корень дерева T оказался в дереве L , он станет его корнем. В дереве L левое поддерево корня останется тем же, что и в дереве T . Вызовем операцию *Split* для правого поддерева корня дерева T и обозначим возвращенный результат за L' и R' . На месте правого поддерева корня дерева T должно оказаться дерево L' . Дерево R при этом будет совпадать с деревом R' .



Обе функции пишутся просто, а время их работы равно высоте дерева, т.е. $O(\log N)$. При этом деревья, полученные в результате применения *Split*, могут быть объединены операцией *Merge*.

8.3 Вставка и удаление элементов

С использованием операций *Split* и *Merge* можно реализовать вставку и удаление элементов из Декартова дерева.

Пусть нам необходимо вставить в Декартово дерево T элемент с ключом x , при этом в нем не было элемента с таким ключом. Дополним ключ x случайным значением y и создадим Декартово дерево с единственным элементом (x, y) , обозначим это дерево за T' .

С помощью операции *Split* разделим дерево T по ключу x на деревья L и R . Затем, с помощью операции *Merge* объединим дерево L и T' , а результат этого объединения снова объединим с помощью *Merge* с деревом R .

Таким образом, нам удалось добавить элемент в дерево с использованием одного *Split* и двух *Merge*. Итоговая сложность добавления составит $O(\log N)$, т.к. каждая операция занимает $O(\log N)$, а операций всего три.

Удаление элемента из дерева осуществляется также с помощью *Split* и *Merge*. Пусть мы хотим удалить элемент x_0 из дерева T . Разделим с помощью *Split* с параметром $x_0 - \varepsilon$ дерево T на деревья L и R . Искомый ключ x_0 окажется при этом в дереве R .

Теперь разделим дерево R по ключу $x_0 + \varepsilon$ на деревья L' и R' . Дерево L' будет состоять из единственного элемента с ключом x_0 и мы просто удалим его. После этого объединим деревья L и R' с помощью Merge.

Удаление элемента реализуется с помощью двух Split и одного Merge за $O(\log N)$.

Неприятным моментом при реализации удаления является использование ε . В случае использования вещественных чисел следует выбирать ε очень аккуратно, а в случае целых чисел произойдет нежелательных переход в вещественную арифметику. В случае строк и других сравнимых структур данных придумать ε достаточно сложно. Красивого решения этой задачи в общем случае нет. Чтобы получить работающее решение, нужно написать функцию Split с параметром, обозначающим в какое из поддеревьев (L или R) необходимо поместить узел, с ключом равным x_0 . Тогда при первом Split необходимо помещать x_0 в правое поддерево, а во втором — в левое. В случае Декартова дерева с целыми числами можно воспользоваться грязным хакем: всегда помещать ключ x_0 , например, в правое поддерево. Тогда первый Split при удалении должен быть вызван с параметром x , а второй — с параметром $x + 1$.

8.4 Декартово дерево по неявному ключу

Вместо ключа x в Декартовом дереве можно использовать порядковый номер элемента. Нам уже известно, как искать элемент по его порядковому номеру в бинарном дереве поиска (с использованием размеров поддеревьев), этот способ работает и в Декартовом дереве. А от ключа x можно просто избавиться, или, если это слишком сложно представить, можно считать, что ключ x равен порядковому номеру элемента, но вычисляется он динамически и x в явном виде в узлах не хранится. При таком подходе к пониманию ключа x Декартово дерево по неявному ключу остается корректным бинарным деревом поиска.

По-английски Декартово дерево по неявному ключу называется *implicit tree*, первое задокументированное применение этой структуры данных принадлежит Николаю Дурову на финале соревнований ACM ICPC в 2000 году.

Для реализации этой идеи для каждого узла Декартова дерева необходимо хранить поле с размером поддерева с корнем в этом узле. Этот параметр будет меняться при выполнении операций Split и Merge, но он легко пересчитывается как сумма размеров поддеревьев с корнями в детях узла и этот пересчет удобно делать при выходе из рекурсивной функции Split или Merge. На асимптотическую сложность алгоритма пересчет не повлияет.

Чтобы происходящее имело смысл, к каждому узлу дерева необходимо приписать некоторое дополнительное содержательное поле, не являющееся ключом и не участвующему ни в каких сравнениях. Рассмотрим на примере текстового редактора. Если мы редактируем большой документ, например, размером в несколько миллионов символов, как «Война и мир» и хотим вставить одну букву где-нибудь в начале документа, то, при хранении текста в виде массива, нам потребуется $O(N)$ операций чтобы сдвинуть все элементы массива, освободить место под новый символ и записать его. Это довольно медленно и может вызвать задержки даже на современном компьютере.

Попробуем хранить текст в виде Декартова дерева по неявному ключу. Каждый узел дерева хранит случайный параметр y , а также размер поддерева, что позволяет находить узел с нужным нам номером (неявным ключом) за $O(\log N)$. Дополнительно

в каждом узле хранится буква. То есть для каждой позиции мы можем за $O(\log N)$ определить, какая буква там находится. Если мы хотим вставить букву на позицию k , то процесс ее вставки очень похож на добавление элемента с ключом $x = k$ в Декартово дерево. А именно, мы применим операцию Split для разрезания по неявному ключу k , получим деревья L и R , создадим дерево, состоящее из одного узла со случайным y , размером поддерева 1 и дополнительным параметром — буквой, которую мы хотим вставить. Назовем это дерево T' . Затем объединим деревья L и T' с помощью операции Merge и еще одним Merge объединим получившееся дерево с деревом R . Таким образом новая буква окажется на своем месте, а сложность составит $O(\log N)$.

Также можно реализовать и удаление из Декартова дерева по неявному ключу аналогично удалению элемента по ключу в обычном Декартовом дереве, сложность этой операции составит $O(\log N)$.

Таким образом, мы научились реализовывать массив, в котором есть операции доступа к элементу по номеру, вставки элемента на заданную позицию и удаления элемента с заданной позиции, все эти операции имеют сложность $O(\log N)$. Если в каждом узле дополнительно хранить ссылку на предка, то можно реализовать доступ к следующему элементу в среднем за $O(1)$ точно таким же образом, как это делалось в обычном бинарном дереве поиска. То есть, в терминах текстового редактора, мы можем отобразить k последовательных символов начиная с некоторого за $O(\log N + k)$

8.5 Групповые операции

Продолжая аналогию с текстовым редактором, мы реализуем еще две довольно естественные операции: вырезания части текста и вставки текста в определенное место. То есть Ctrl+X и Ctrl+V.

Естественно, эти операции можно реализовать, удаляя или вставляя текст по буквам, но тогда сложность составит $O(k \log N)$, где k — количество букв во вставляемом фрагменте. Попробуем реализовать эти операции таким образом, чтобы их сложность не зависела от k .

Начнем с операции удаления. Обозначим за p позицию, начиная с которой мы производим удаление, а за $q = p + k$ — номер первого символа за пределами удаляемого фрагмента, то есть должны быть удалены все символы из полуоткрытого интервала $[p, q)$, а количество удаляемых элементов равно k . Пусть текст хранится в Декартовом дереве по неявному ключу T . Применим операцию Split к этому дереву с параметром p (нужна реализация Split, в которой элемент с неявным ключом оказывается в правом поддереве). В результате у нас получится деревья L и R . Разрежем с помощью Split с ключом k дерево R на деревья L' и R' . Затем объединим с помощью Merge деревья L и R' и получим текст с удаленным фрагментом, начиная с позиции p и длиной k . В L' при этом останется Декартово дерево по неявному ключу, содержащее вырезанный фрагмент. Если он не нужен, то память можно почистить.

Теперь рассмотрим процесс вставки фрагмента в текст, причем и фрагмент и текст являются Декартовыми деревьями по неявному ключу. Если мы хотим вставить в дерево T фрагмент T' начиная с позиции p , то нам достаточно разрезать дерево T с помощью Split с параметром p на деревья L и R , затем с помощью Merge объединить деревья L и T' и еще одним Merge объединить получившееся дерево с R .

Таким образом, и для операции вырезания и для операции вставки фрагмента ис-

пользуется всего три операции Merge и Split и сложность этих операций составляет $O(\log N)$ независимо от длины фрагмента.

8.6 Проталкивание обещаний

Рассмотрим некоторое поддереву Декартова дерева по неявному ключу. Очевидно, что в этом поддереве содержатся элементы с последовательными номерами, то есть отрезок, начало которого вычисляется как сумма размеров левых поддеревьев и длины пути при поиске от корня, а длина — как размер поддерева с этим корнем.

Нам уже известна структура данных, где каждый элемент соответствует некоторому отрезку — это дерево отрезков. Дерево отрезков позволяет выполнять ряд интересных операций, например, прибавление значения ко всем элементам на отрезке или присваивание всем элементам на отрезке одного и того же значения. Нам хотелось бы научиться совершать подобные операции над Декартовым деревом по неявному ключу.

В дереве отрезков подобные операции выполняются достаточно легко: узлы этого дерева хорошо структурированы и легко доказать сложность $O(\log N)$ при выполнении операций над отрезком. В Декартовом дереве по неявному ключу отрезки, соответствующие узлам поддерева, могут иметь разную длину и вообще плохо структурированы, так что реализация этой операции обещает быть сложной, а сложность — не самой лучшей.

Для реализации таких операций нужно отвлечься от дерева отрезков и вернуться к стандартному подходу при работе с Декартовым деревом: вырезать интересующий нас отрезок, наложить на его корень нужную нам модификацию, а затем слить два дерева.

Рассмотрим реализацию Split и Merge для корректной работы присваивания на отрезке. Договоримся поддерживать следующий инвариант: последняя по времени операция присваивания для элемента находится выше всего в дереве (ближе всего к его корню). Для поддержки этого инварианта необходимо при любом проходе по дереву при реализации операций Merge и Split при входе в функцию сразу же проталкивать обещание изменения вниз, в каждого из детей этого узла. В принципе, можно модифицировать и функцию поиска, которая также при входе в какой-либо узел будет проталкивать обещание применить операцию модификации в своих детей — это не изменит ее асимптотическую сложность, но позволит положить обещанное значение ровно в тот элемент, для которого осуществляется поиск.

Реализация обещаний и их проталкивания довольно проста. Для обещания модификации нужно в каждом узле завести поле для модификации, а для их проталкивания достаточно присвоить значение этого поля для текущего в узла в то же поле для его существующих детей.

Эти операции никак не изменяют асимптотическую сложность ни одной из операций.

Кроме присваивания можно реализовать любую операцию моноида, то есть множества с нейтральным элементом и с единственной бинарной операцией, обладающей свойством ассоциативности.

8.7 Избавление от y

Введение параметра «размер поддерева» для каждого узла позволило нам избавиться от параметра x и получить Декартово дерево по неявному ключу. Оно также

позволяет избавиться и от случайного значения y .

Значение y используется только в операции Merge для выбора корня из двух поддеревьев. Казалось бы, можно просто выбирать в качестве корня объединенного поддерева любой из корней объединяемых деревьев равновероятно, но это не так. Если в одном из деревьев элементов больше, чем в другом, то больше вероятность того, что корень большего поддерева имеет параметр y больше, чем у меньшего по размеру поддерева.

Мы знаем, что y — равномерно распределенная случайная величина. Пусть два объединяемых дерева имеют размеры a и b , то есть всего у нас $a + b$ случайных чисел. Вероятность того, что наибольшее число находится в первом дереве равна $\frac{a}{a+b}$, а во втором — $\frac{b}{a+b}$. Таким образом, зная размеры объединяемых деревьев мы можем выбирать корень, например, генерируя случайное целое число из интервала от 0 до $a + b$ и если число оказалось меньше a , то выбирается корень левого поддерева (с размером a), а иначе — правого (с размером b). Такой подход позволяет неасимптотически уменьшить расход памяти и делает код более понятным и коротким.

Лекция 9

Префиксные суммы, разреженные таблицы и другое

9.1 Префиксные суммы

Для решения задачи RSQ (Range Sum Query) без модификации существует очень простой и эффективный способ — префиксные суммы. Пусть нам дана последовательность из N чисел и необходимо отвечать только на запросы типа «сумма чисел на отрезке от L до R », без запросов модификации. Нам потребуется дополнительный массив `pref` длиной N , k -ый элемент которого будет хранить сумму всех элементов, до k -го включительно. Наивный подсчет такого массива займет $O(N^2)$, но это время легко сократить до $O(N)$, пользуясь соотношениями: $\text{pref}[0] = a[0]$ и $\text{pref}[k] = \text{pref}[k - 1] + a[k]$, где a — исходный массив.

a	5	3	8	1	4
pref	5	8	16	17	21

Пользуясь массивом `pref` легко ответить на запрос суммы на отрезке от L до R : $\text{pref}[R] - \text{pref}[L - 1]$.

Однако, в этой простой идее и ее реализации есть несколько технических моментов, осложняющих жизнь. Во-первых, это возможное переполнение. Например, если все числа исходного массива укладываются в тип `int`, то их сумма может превышать максимально допустимое значение. Во-вторых, это проблема с запросами, в которых $L = 0$ — в них мы будем обращаться к нулевому элементу. Можно обработать такие запросы отдельно, просто проверяя L на равенство нулю и возвращая $\text{pref}[R]$. В случае, если задача подразумевает нумерацию элементов с единицы, то удобно сделать $\text{pref}[0] = 0$ и тогда не будет возникать никаких особых случаев. Иногда используют и сдвиг нумерации, чтобы $\text{pref}[k]$ содержал в себе сумму элементов исходного массива с номерами от 0 до $k - 1$ — благодаря этому $\text{pref}[0]$ не будет использоваться и его также можно сделать нулем. Это потребует изменения индексов в функции ответа на запрос префиксной суммы.

Вместо сложения можно использовать любую операцию, для которой существует обратная операция. Для сложения это вычитание, а, например, для умножения — деление (однако здесь следует быть очень внимательными). Существуют и операции, которые обратны сами себе, например, XOR («исключающее или» или, что то же самое, побитовое сложение по модулю два). Префиксные суммы подходят и для задач о подсчете количества элементов на отрезке, обладающих каким-либо свойством. В таком случае

при подсчете массива для префиксной суммы на k -ом шаге прибавляется единица, если k -ый элемент искомого массива обладал необходимым свойством, и ноль в противном случае.

Также, вместо подсчета префиксной суммы можно использовать суффиксную сумму, которая считается похожим образом и может дать тот же результат. Иногда бывают полезны префиксные и суффиксные минимумы и максимумы, хотя для них нельзя определить минимум или максимум на отрезке.

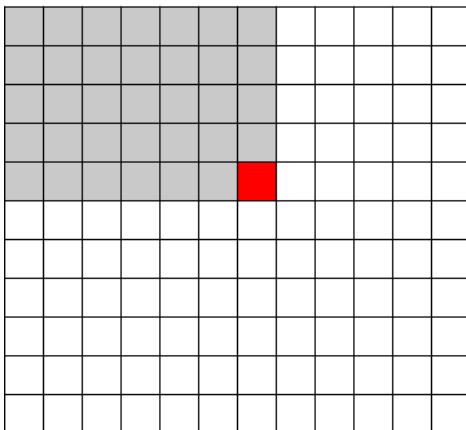
Предподсчет массива для хранения префиксных сумм требует $O(N)$ времени и дополнительной памяти, а каждый ответ на запрос занимает $O(1)$.

9.2 Двумерные префиксные суммы

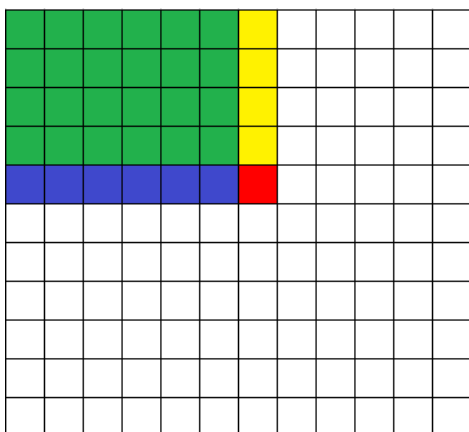
Аналогично одномерному случаю можно сформулировать задачу RSQ для двумерного случая: необходимо научиться отвечать на запрос суммы в прямоугольнике с левым верхним углом в клетке с координатами (i_1, j_1) и правым нижним углом в клетке с координатами (i_2, j_2) .

По аналогии с одномерным массивом префиксных сумм будем считать двумерный массив сумм на верхнем левом уголке. То есть $\text{pref}[i][j]$ должна хранить сумму всех элементов находящихся выше и левее, т.е. тех, у которых номер строки не превосходит i , а номер столбца — j .

На рисунке серым помечены клетки, сумму для которых нужно подсчитать, а красным — текущая клетка:



Чтобы быстро подсчитать ячейку массива $\text{pref}[i][j]$ необходимо знать значения массива pref для всех чисел выше и левее. Подсчет осуществляется несколько сложнее, чем в одномерном случае. На рисунке для подсчета значения $\text{pref}[i][j]$ необходимо просуммировать все значения в цветных ячейках:



Сумма желтых и зеленых клеток уже подсчитана и хранится в ячейке $\text{pref}[i-1][j]$, сумма синих и зеленых — в $\text{pref}[i][j-1]$. Если просуммировать их, то зеленая область будет посчитана дважды, однако, мы можем один раз вычесть сумму в этой области, она хранится в $\text{pref}[i-1][j-1]$. Также надо не забыть прибавить значение, которое хранится в красной ячейке. Итоговое значение будет вычисляться по формуле $\text{pref}[i][j] = \text{pref}[i-1][j] + \text{pref}[i][j-1] - \text{pref}[i-1][j-1] + a[i][j]$.

Зная массив префиксных сумм мы можем вычислить сумму в прямоугольнике с координатами $(i_1, j_1), (i_2, j_2)$, по формуле

$$\text{pref}[i_2][j_2] - \text{pref}[i_1 - 1][j_2] - \text{pref}[i_2][j_1 - 1] + \text{pref}[i_1 - 1][j_1 - 1]$$

Эта формула выводится из рассуждений, аналогичных использованным для подсчета массива префиксных сумм. На рисунке значками $+$ и $-$ помечены клетки, для которых берутся слагаемые в этом рассуждении.

			j_1			j_2			
			$+$			$-$			
i_1									
i_2			$-$				$+$		

Также как и в одномерном случае, возникает проблема с нулевыми индексами, которая решается аналогичными же методами. Вместо суммирования также можно использовать другие операции, имеющие обратные.

Возможно расширение идеи префиксных сумм и на большее количество измерений.

9.3 Разреженные таблицы

Для решения задачи RMQ (Range Minimum/Maximum Query) без модификаций также существует эффективная структура, позволяющая отвечать на запросы за $O(1)$ — разреженные таблицы (Sparse table).

Разреженная таблица — это двумерный массив, в каждой ячейке которого хранится значение минимума на некотором отрезке исходного массива. Длины отрезков, для которых хранится минимум, будут степенями двойки. В нулевой строке разреженной

таблице хранятся минимумы для отрезков длиной $2^0 = 1$, в первой — для отрезков длиной $2^1 = 2$ и т.д. Номер столбца в разреженной таблице задает позицию, с которой начинается отрезок. Несложно заметить, что первая строка разреженной таблицы совпадает с исходным массивом, а количество строк составляет $\lfloor \log N \rfloor$. Формально, каждая ячейка вычисляется по формуле:

$$st[i][j] = \min(a[j], a[j+1], \dots, a[j+2^i-1]), i \in [0.. \log N]$$

Пример вычисленной таблицы:

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

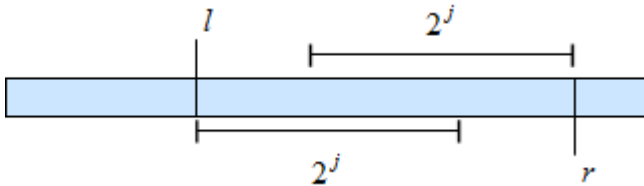
Вычислить таблицы за $O(N \log N)$ можно воспользовавшись тем фактом, что минимум на отрезке длиной 2^i является минимумом из минимумов для двух отрезков длиной 2^{i-1} , полностью покрывающих отрезок длиной 2^i . Т.е. для всех строк разреженной таблицы, кроме нулевой, вычисление происходит по формуле:

$$st[i][j] = \min(st[i-1][j], st[i-1][j+2^{i-1}])$$

Таким образом, сложность подсчета всей разреженной таблицы составит $O(N \log N)$.

После подсчета таблицы мы можем за $O(1)$ найти минимум на любом отрезке, который начинается с позиции j и длина которого равна 2^i — для этого достаточно просто взять значение $st[i][j]$.

Ответить на запрос минимума на произвольном отрезке можно разбив этот отрезок на два отрезка, длина которых является степенью двойки и которые полностью накрывают весь запрос (возможно, они будут пересекаться).



Формально минимум на отрезке $[l..r]$ вычисляется по следующим формулам:

$$\min(a[l], a[l+1], \dots, a[r]) = \min(st[i][l], st[i][r-2^i+1])$$

$$i = \max(k | 2^k \leq r - l + 1)$$

Чтобы такой запрос работал эффективно, необходимо научиться быстро определять максимальную степень двойки, не превосходящую длины отрезка. Стандартной операции, реализующей такой поиск, нет. Перебор степени двойки и поиск последней, не превышающей длины отрезка, займет $O(\log N)$. Чтобы реализовать ответ на зпрос за $O(1)$ необходимо для каждого числа, не превосходящего N предподсчитать максимальную степень двойки (N — длина всего массива и, соответственно, максимальная длина запроса). Предподсчет будет занимать $O(N \log N)$ или $O(N)$ в зависимости от реализации. Т.к. построение разрежной таблицы занимает $(O(N \log N))$, то с точки зрения асимптотики неважно, какой способ предподсчета выбрать.

Для использования в разреженных таблицах пригодна не только операция минимума или максимума, но и любая другая операция, обладающая свойством дистрибутивности, ассоциативности и идемпотентности. Операция \times называется идемпотентной, если $a \times a = a$.

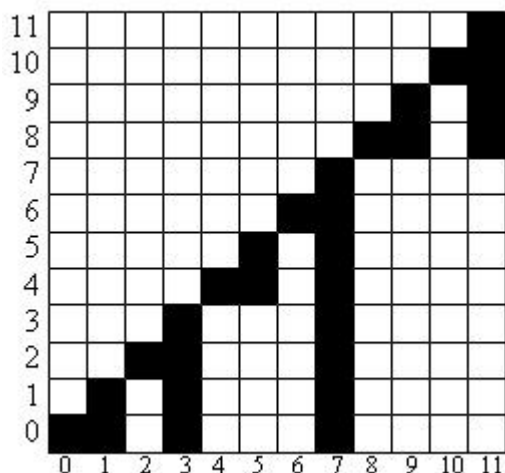
9.4 Дерево Фенвика

Дерево Фенвика позволяет решить задачу RSQ с запросом и модификацией отдельного элемента за $O(\log N)$ с очень маленькой константой и простой реализацией. Эта структура была предложена Питером Фенвиком в 1994 году.

Пусть дан массив a , дерево Фенвика описывается массивом такого же размера t , каждый элемент которого вычисляется по формуле: $t[i] = \sum_{k=F(i)}^i a_k$. Функция $F(i)$ может быть различной, мы рассмотрим вариант, позволяющий реализовать операции за $\log N$.

Определим $F(i)$ следующим образом: если запись числа i в двоичной системе счисления оканчивается нулем, то $F(i) = i$. Если же запись числа i оканчивается на одну или несколько единиц, то превратим все эти единицы в нули и это и будет значением функции $F(i)$. Такая функция записывается очень легко: $F(i) = i \& (i + 1)$, где $\&$ — то побитовое «и».

На рисунке по горизонтали находятся индексы массива t , а по вертикали — массива a . Черные клетки показывают, что соответствующий элемент массива a входит в сумму для элемента массива t .



Для построения дерева Фенвика мы воспользуемся функцией подсчета суммы на префиксе для уже построенной части дерева Фенвика.

Пусть нам необходимо вычислить значение суммы на префиксе до некоторого индекса r . Первым слагаемым будет сумма на отрезке $a[F(r)..r]$, она хранится в ячейке $t[r]$. Следующее слагаемое будет суммой отрезка $a[F(F(r) - 1)..F(r) - 1]$ и т.д.

Количество шагов не будет превосходить $O(\log N)$. Действительно, если в конце числа стояла группа единиц, то за одну операцию она превратится в группу нулей, причем количество нулей в конце числа станет больше, чем было единиц. На следующем шаге из числа вычитается единица и все нули в конце числа превращаются в единицы, причем их стало больше, чем единиц на предыдущем шаге. Таким образом, количество шагов не превосходит удвоенного количества цифр в двоичной записи числа и составля-

ет $O(\log N)$. Для константы можно доказать и более строгую оценку, но асимптотическая сложность останется прежней.

Умея находить сумму на префиксе мы можем находить и сумму на отрезке и, таким образом, построить дерево Фенвика по определению. Однако использование операции модификации позволит сэкономить некоторое количество строк кода.

Реализуем операцию прибавления к текущему значению массива $a[i]$ некоторого числа d , назовем ее $inc(i, d)$. Из этой операции легко сделать и присваивание $a[i] = x$, для этого достаточно сделать $inc(i, x - a[i])$. Все эти изменения нужно отразить и в дереве Фенвика t .

Для этого нам нужно быстро понимать, для каких j наш индекс i попадет в отрезки $[F(j)..j]$. Естественно, наше число попадает только в отрезки, где $j \geq i$ и точно попадает в отрезок для $j = i$ (он и будет первым отрезком, в которое попадает i). Чтобы получить следующее число, в отрезок для которого попадает i , достаточно поменять в текущем числе последний ноль на единицу. Такую замену можно сделать функцией $H(j) = j|(j+1)$, где $|$ — побитовое «или». Заменять значения нужно до тех пор, пока текущая граница отрезка меньше N — размера массива. Сложность этой операции равна $O(\log N)$ и доказывается аналогично сложности операции подсчета суммы.

Лекция 10

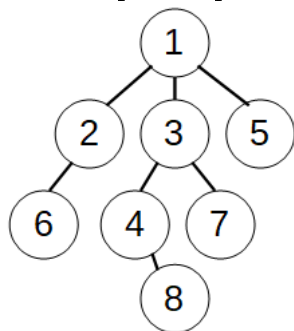
Наименьший общий предок

10.1 Взаимное расположение вершин в дереве

Пусть дано дерево T с выделенным корнем (не бинарное и не дерево поиска, просто дерево как связный неориентированный граф без циклов). Нам необходимо для двух вершин $V1$ и $V2$ научиться определять их взаимное расположение. Будем называть вершину V предком вершины U , если U располагается в поддереве, корнем которого является вершина V .

Две вершины в дереве могут быть расположены так, что $V1$ является предком $V2$ или $V2$ является предком $V1$. Также возможен вариант, что ни одна из вершин не является предком другой — эти вершины расположены в разных поддеревьях.

Рассмотрим дерево на рисунке:



Запустим обход в глубину от корня (вершины 1) и будем выписывать номера вершин в массив `order` каждый раз, когда функция обхода в глубину оказывается в этой вершине (при первом входе или при возвращении из сына). Получится следующая последовательность:

Индекс	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
order	1	2	6	2	1	3	4	8	4	3	7	3	1	5	1

Если в дереве N вершин, то в этой последовательности будет $2N - 1$ число — каждое ребро порождает два числа (когда проходим по ребру вниз и когда возвращаемся по нему вверх), а также на первой позиции дополнительно выписывается корень.

Для каждой вершины мы можем определить позицию первого и последнего вхождения в эту последовательность:

Вершина	1	2	3	4	5	6	7	8
first	0	1	5	6	13	2	10	7
last	14	3	11	8	13	2	10	7

Такую таблицу легко построить за $O(N)$. Пользуясь этой таблицей и свойствами обхода в глубину (мы выходим из вершины только после того как обработаны все ее потомки), легко сформулировать того, что вершина $V1$ является потомком вершины $V2$: отрезок $[first[V1], last[V1]]$ полностью содержится в отрезке $[first[V2], last[V2]]$. Вообще говоря, достаточно проверить принадлежность отрезку $[first[V2], last[V2]]$ любого из чисел $[first[V1]$ или $last[V1]]$, т.к. отрезки не могут пересекаться, а могут быть только вложены друг в друга. Также отрезки могут не иметь общих точек, тогда вершины $V1$ и $V2$ лежат в разных поддеревьях.

После подсчета за $O(N)$ мы можем определить взаимное расположение вершин за $O(1)$.

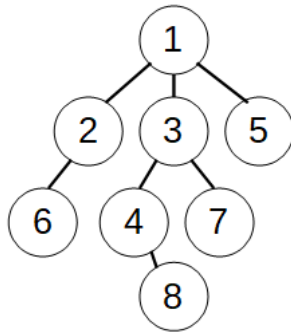
10.2 Наименьший общий предок

Общим предком вершин $V1$ и $V2$ называется такая вершина V которая лежит и на пути от корня до вершины $V1$ и на пути от корня до вершины $V2$. Таких вершин может быть несколько и наименьшим общим предком называется самая нижняя вершина (наиболее удаленная от корня). По-английски эта вершина называется Least Common Ancestor (LCA).

Рассмотрим массив `order`, в котором хранится порядок обхода вершин. Найдем первое вхождение вершины $V1$ (оно хранится в `first[V1]`) и первое вхождение вершины $V2$. Допустим, что `first[V1]` меньше `first[V2]`, в противном случае поменяем $V1$ и $V2$ местами. Последовательность номеров вершин из массива `order` от позиции `first[V1]` до позиции `first[V2]` содержит в себе некоторый путь от вершины $V1$ до вершины $V2$. Этот путь не является кратчайшим и включает в себя посещение всех промежуточных поддеревьев, но главное, что в нем содержится и наименьший общий предок этих вершин. Путь в дереве между двумя вершинами обязательно проходит через их наименьшего общего предка.

При этом по построению массива `order` (с помощью обхода в глубину), мы можем быть уверены, что на пути от вершины $V1$ до вершины $V2$ в том порядке, как перечислены вершины в массиве `order`, не содержится никакой вершины, находящейся выше их наименьшего общего предка. Действительно, если мы поднялись выше наименьшего общего предка, то мы никогда не зайдём в него повторно, а вершины $V1$ и $V2$ обязательно должны содержаться в поддереве, корнем которого является их наименьший общий предок.

Таким образом, для нахождения наименьшего общего предка двух вершин достаточно найти номер вершины в массиве `order` начиная с позиции `first[V1]` и заканчивая позицией `first[V2]`, расстояние от корня до которого минимально. Для этого нам необходимо подсчитать расстояние от корня до каждой из вершин дерева (массив `h`). Это можно сделать заранее, вместе с построением массива `order` не увеличивая асимптотическую сложность.



Индекс	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
order	1	2	6	2	1	3	4	8	4	3	7	3	1	5	1
h	0	1	2	1	0	1	2	3	2	1	2	1	0	1	0

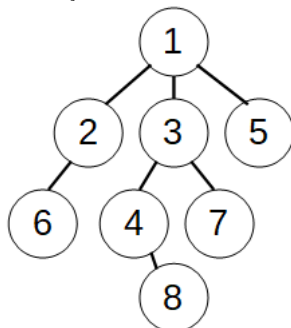
Таким образом, задача поиска наименьшего общего предка вершин $V1$ и $V2$ сводится к поиску индекса минимального элемента в массиве h на отрезке от $\text{first}[V1]$ до $\text{first}[V2]$. Номер вершины находится в массиве order в ячейке с найденным индексом.

Для реализации запроса минимума на отрезке (RMQ) подходит либо разреженная таблица ($O(N \log N)$ предподсчет, $O(1)$ на запрос) или дерево отрезков ($O(N)$ предподсчет, $O(\log N)$ на запрос). Выбирать подходящую структуру следует исходя из соотношения количества элементов в дереве и количества запросов, но чаще более подходящим выбором оказывается разреженная таблица.

10.3 Двоичные подъемы

Научимся определять для каждой вершины какая вершина находится выше нее в дереве на k уровней. Чаще всего дерево задается с помощью указания предка для каждой вершины, так что мы уже знаем для всех вершин, какая вершина находится на один уровень выше. Если дерево задано как граф, можно определить предков для каждой вершины с помощью обхода в глубину.

Зная непосредственного предка для каждой вершины легко определить, какая вершина находится на два уровня выше текущей — для этого достаточно узнать предка нашего предка. Следующим шагом можно узнать, какая вершина находится на четыре уровня выше: для этого достаточно взять номер вершины, находящейся на два уровня выше, для вершины, которая находится на два уровня выше текущей. Продолжая такие рассуждения, мы сможем определить вершину, находящуюся на 2^i уровней выше. Максимальная глубина дерева из N вершин составляет $N - 1$, поэтому необходимо выполнять эти действия до тех пор, пока $2^i < N$. Данные удобно хранить в таблице up размером N на $\log N$, где номер столбца задает номер вершины, а номер строки — величину k . Это во многом похоже на вычисление разреженной таблицы.



k	1	2	3	4	5	6	7	8
0	-1	1	1	3	1	2	3	4
1	-1	-1	-1	1	-1	1	1	3
2	-1	-1	-1	-1	-1	-1	-1	-1

Пользуясь таблицей `up` мы можем легко определить предка вершины, находящегося ровно на расстоянии k . Для этого необходимо перебирать степени двойки, начиная со старшей и если $2^i \geq k$, то осуществлять переход (т.е. изменять текущую вершину v на `up[i][v]`) и уменьшать k на 2^i . Такой метод позволяет подняться на произвольное количество уровней за $O(\log N)$.

Вместе с номером вершины-предка на заданном расстоянии в таблице `up` можно хранить какую-либо дополнительную информацию про путь до этого предка. Например, это может быть минимальный номер вершины на пути. С помощью этой информации мы сможем отвечать на запросы вида «определите минимальный номер вершины на пути от вершины v к ее предку, находящемуся на k уровней выше». Подсчет такой информации не изменит асимптотику алгоритма. Кроме того, к вершинам дерева может быть приписана какая-то дополнительная метка и можно также сохранять информацию об этих метках в таблице `up`. Например, в задаче может быть дано генеалогическое дерево, где номер вершины символизирует человека, а дополнительная метка — величину состояния этого человека. Тогда задача может быть поставлено таким образом: для запроса v, k определите самого богатого предка человека v , который жил не более k поколений назад.

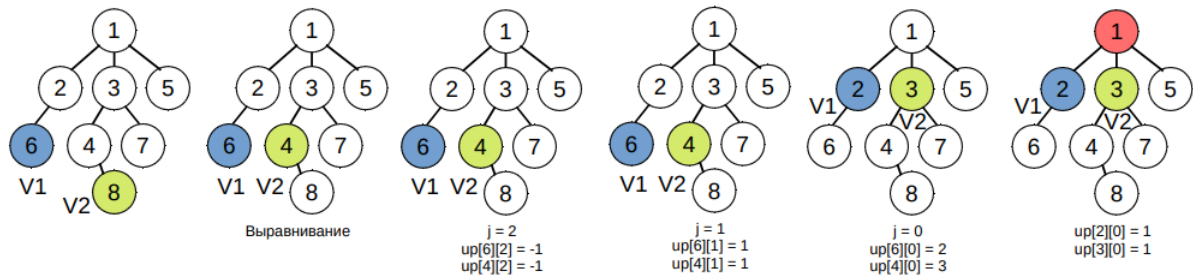
10.4 LCA через двоичные подъемы

Информацию, накопленную в массиве `up` можно использовать и для нахождения наименьшего общего предка для вершин $V1$ и $V2$. Сначала необходимо выровнять высоту вершин. Допустим, вершина $V1$ оказалась глубже вершины $V2$ на j слоев. Тогда поднимемся от вершины $V1$ на j и заменим число $V1$ на найденную вершину. В этот момент могла оказаться так, что мы попали в вершину $V2$ (то есть вершина $V2$ является предком вершины $V1$), тогда $V2$ и является наименьшим общим предком.

Если же оказалось, что после подъема $V1$ и $V2$ различны мы можем попытаться найти минимальную величину k , на которую нужно подняться из обеих вершин (их высоты теперь одинаковы), чтобы попасть в одну и ту же вершину и из $V1$ и из $V2$. Эта вершина и будет их наименьшим общим предком. Найти величину подъема можно с помощью бинарного поиска, ведь если поднявшись на k от каждой из вершин мы оказываемся в одной и той же вершине, то и при подъеме на более число мы будем оказываться в одной и той же вершине. Сложность бинарного поиска составляет $(\log N)$, каждый подъем занимает $O(\log N)$, общая сложность такого метода составит $O(\log^2 N)$. При использовании бинарного поиска можно было бы и не выравнивать высоту вершин предварительно, а просто аккуратно реализовать подъем от более глубокой вершины не на m уровней вверх, а дополнительно прибавив к этому числу разницу высот в вершинах.

Однако, предварительное выравнивание высот вершин позволяет найти LCA за $O(\log N)$, а не за $O(\log^2 N)$. Будем перебирать степени двойки в порядке убывания и пытаться подняться от обеих вершин вверх на 2^j . Если в результате этого подъема мы попадем в одну и ту же вершину, то не будем осуществлять его. Если же в результате

подъема мы попадаем в разные вершины, то заменим значения $V1$ и $V2$ на эти вершины. После того, как мы переберем все степени двойки, будет достаточно подняться из любой из вершин $V1$ или $V2$ на один уровень вверх и мы попадем в наименьшего общего предка.



10.5 Кратчайшие пути в дереве

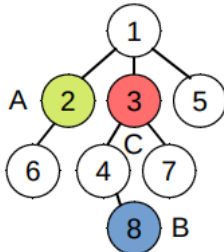
Между двумя вершинами дерева существует единственный, простой путь и он же является кратчайшим. Очевидно, что путь от вершины $V1$ до вершины $V2$ состоит из двух участков: подъема от вершины $V1$ до $LCA(V1, V2)$ и спуска от $LCA(V1, V2)$ до вершины $V2$. Если одна вершина является предком другой, то одна из частей пути окажется пустой.

Длину кратчайшего пути легко вычислить, для этого достаточно сложить глубины вершин $V1$ и $V2$ и вычесть удвоенную глубину $LCA(V1, V2)$.

Рассмотрим такую задачу: найти вершину, лежащую на середине пути от вершины A до вершины B в дереве. Очевидно, что вершина, лежащая на середине пути существует только в случае, если длина пути по ребрам четная.

Вычислим длину пути от A до B за $O(\log N)$ с помощью LCA, проверим четность и поднимемся от более глубокой вершины с помощью двоичных подъемов на половину длины пути.

Рассмотрим еще одну задачу: необходимо подсчитать количество вершин в дереве, равноудаленных от вершин A и B . Очевидно, что вершина, лежащая на середине пути, равноудалена от A и B , обозначим ее за C . Кроме нее также равноудалены от A и от B все вершины, лежащие в поддеревьях вершины C , которые не содержат ни A , ни B .



Нам необходимо научиться проверять, что одна вершина не лежит в поддереве другой. Это было бы легко сделать, если бы мы записывали порядок обхода вершин, но в этой задаче нам нужно не только проверить этот факт, но и найти сына вершины C , который является корнем поддерева, содержащего A или B . Достаточно подняться от вершины A и от вершины B на высоту, равную половине длины пути и проверить, оказались ли мы в вершине C , таким образом, мы сможем определить, лежит ли вершина A или B в поддереве с корнем в C . После того, как мы определим это, необходимо найти

корень поддерева, вершиной которого является сын вершины C и которому принадлежит вершина A или B (та или те, которые содержатся в поддереве с корнем в C). Для этого достаточно подняться от A и B на величину, равную половине длины пути минус один. Таким образом, мы окажемся в сыне вершины C и определим интересное нас поддерево.

Теперь нам хотелось бы просуммировать размеры поддеревьев сыновей вершины C , кроме тех, в которых содержатся вершины A или B — все вершины в них будут равноудалены от A и от B . Размеры поддеревьев можно легко подсчитать с помощью обхода в глубину. Но перебор всех поддеревьев может занять $O(N)$ (например, если вершина C будет корнем и будет лежать в центре «солнышка» и иметь $O(N)$ потомков). Нас интересует только суммарный размер всех поддеревьев, кроме поддеревьев содержащих A или B , поэтому нам достаточно вычесть из размера поддерева, корнем которого является C , размеры поддеревьев, содержащие A или B .

В этой задаче есть еще один особый случай, когда C является $LCA(A, B)$. В этом случае равноудаленными от A и B будут также все вершины «наддеревья» C . Опять же, нас интересует только их количества, которое легко определить, вычтя из размера всего дерева размер поддерева с корнем в вершине C .

10.6 Минимальное остовное дерево с заданным ребром

Рассмотрим такую задачу: для каждого ребра графа необходимо найти вес минимального остовного дерева, обязательно содержащего заданное ребро. Мы можем воспользоваться любым алгоритмом поиска минимального остовного дерева за $O(E \log V)$ и для ребер, вошедших в минимальный осто, ответ станет очевиден. Осталось научиться решать задачу для ребер, не входящих в минимальное остовное дерево.

После того, как определено минимальное остовное дерево, подвесим его за произвольную вершину и подсчитаем таблицу up для двоичных подъемов, дополнительно храня вес максимального ребра на пути подъема. Для подъема на один это просто вес ребра из вершины к ее предку, а для подъемов большей длины выбирается максимум из двух ребер на половинках этого подъема.

Будем перебирать все ребра графа, не входящие в минимальный осто, за $O(E)$ и для каждого ребра графа (u, v) находить $LCA(u, v)$. После нахождения LCA мы можем найти максимальное ребро на пути из u в $LCA(u, v)$ и на пути из v в $LCA(u, v)$. Максимум из двух этих чисел даст нам вес максимального ребра на пути из u в v по ребрам минимального остовного дерева. После добавления ребра (u, v) в минимальном остове образуется цикл и необходимо удалить максимальное ребро на этом цикле, исключая ребро (u, v) . Вес этого ребра нам удалось определить с помощью двоичных подъемов, а разница веса ребра (u, v) и найденного даст величину отличия дерева с ребром (u, v) от минимального остовного дерева, что позволит определить суммарный вес ребер.

Таким образом, вся задача будет решена за $O(E \log V)$.

10.7 Дополнительная информация в вершинах

Задача: в игре «Герои 3» карта оказалась сгенерирована в виде дерева, в узлах которого находятся замки, в которых можно нанять существ, а ребра — это дороги между замками. У героя есть 7 слотов под существ, в каждом замке есть несколько существ, которых можно нанять (количество существ для найма в замке может быть значительно больше 7). Каждое существо обладает одним параметром — силой, и представлено в замке в одном экземпляре. Герой перемещается из замка A в замок B по кратчайшему пути и хочет собрать самую сильную армию на своем пути. Запросов с парами вершин будет много.

Для решения этой задачи нам нужно построить таблицу up для двоичных подъемов, дополнительно храня для каждого подъема 7 самых сильных существ на пути. Чтобы сделать это, нужно предварительно выбрать 7 самых сильных существ в каждом замке (более слабые существа точно не попадут в армию героя), это можно сделать за $O(K)$ (где K — общее количество существ в замке) с помощью поиска K -ой порядковой статистики с использованием идеи быстрой сортировки. Когда лучшие существа в замках будут определены, необходимо аккуратно посчитать лучших существ на пути вверх для таблицы двоичных подъемов. Следует избегать дублирования существ, которое может произойти при склеивании двух путей в один, этого можно достичь, например, не учитывая в пути начального замка и добавлять существ из него при склеивании путей отдельно. Для пути также достаточно хранить только 7 самых сильных существ на этом пути, отбрасывая остальных при каждом склеивании.

После того, как таблица для двоичных подъемов посчитана, дальнейшее решение не представляет особенной сложности: достаточно найти $LCA(A, B)$ и найти 7 самых сильных существ на пути от A до $LCA(A, B)$ и от B до $LCA(A, B)$, объединить их и выбрать 7 самых сильных из них. Общая сложность решения будет состоять из выбора существ в каждом из замков (эта часть будет равно суммарному количеству существ во всех замках), построении таблицы двоичных подъемов за $O(N \log N)$ и ответ на каждый запрос за $O(\log N)$.

Лекция 11

Арифметика и теория чисел

11.1 Простые числа

Простым числом называется натуральное число, большее 1, которое делится нацело только на себя и 1 (имеет два натуральных делителя). Составными числами, соответственно, называются все остальные натуральные числа, большие единицы. Является ли 1 простым числом — вопрос дискуссионный (в российской математической традиции скорее нет).

В первую очередь нам необходимо уметь проверять, является ли число N простым. Т.е. нам необходимо узнать, существуют ли такие натуральные числа x, y ($1 < x, y < N$), что $x \times y = N$. Кроме того, примем, что $x \leq y$, тогда можно сказать, что x меньше либо равен квадратному корню из числа N (если это условие не выполнено, то произведение будет заведомо больше N).

Таким образом, можно написать функцию, которая будет довольно эффективно проверять число на простоту:

```
bool isprime(int n)
{
    for (int i = 2; i * i <= n; ++i)
        if (n % i == 0)
            return false;
    return true;
}
```

В этой функции используется умножение вместо извлечения квадратного корня — это позволяет избежать проблем с точностью, кроме того, извлечение корня очень медленная операция.

Пользуясь теми же соображениями, очень просто написать функцию, находящую все разложения числа на два множителя. Для этого достаточно убрать проверку на равенство N двум и в заменить `return 0` на обработку двух найденных делителей i и n/i .

Оба этих алгоритма имеют сложность $O(\sqrt{N})$.

11.2 Факторизация числа

Факторизацией числа называется его разложение на простые множители с учетом их степени. Факторизация чисел широко применяется в народном хозяйстве, например, повсеместно используется для шифрования (алгоритм RSA).

Простейший алгоритм факторизации числа перебирает все возможные числа до \sqrt{N} и пытается разделить число N на каждое из этих чисел столько раз, сколько это возможно. Каждый раз число N уменьшается. Если после выполнения этого алгоритма N не стало равным единице, то имеющееся значение также является простым делителем числа N , просто этот делитель больше \sqrt{N} . Очевидно, что такой делитель может быть только один. Например, число 26 представляется как произведение чисел 2 и 13, где 13 больше \sqrt{N} .

Следует понимать, что в случае больших N проверка делимости происходит не за $O(1)$, а, вообще говоря, сложность такой проверки составляет $O(\log^2 N)$ при делении в столбик. Использование различных ухищрений, основанных на быстро преобразовании Фурье позволяет ускорить эту операцию, однако сложность все равно будет больше $O(\log N)$.

Также понятно, что перебирать в качестве потенциального множителя нужно не все числа, а только простые. Иногда заранее создают предподсчитанный массив с простыми числами до K , что позволяет довольно эффективно использовать метод перебора всех делителей для чисел до K^2 . С учетом того, что количество простых чисел от 2 до N примерно равно $\frac{N}{\ln N}$, то совместив предподсчет и деление в столбик можно получить сложность факторизации $O(\sqrt{N} \times \log N)$.

11.3 Решето Эратосфена

Решето Эратосфена позволяет найти все простые числа до N за $O(N \log N \log N)$. Идея алгоритма очень простая: выписываются все числа от 2 до N , затем вычеркиваются все числа, кратные двум (кроме двойки), затем числа кратные трем (кроме тройки) и т.д. В итоге невычеркнутыми остаются только простые числа.

В программе это можно реализовать в виде массива bool. Если очередное k -ое число не вычеркнуто, то можно начинать вычеркивать числа начиная с k^2 , т.к. меньшие составные числа, кратные k , имеют простой делитель меньший чем k и уже были вычеркнуты на прошлых шагах алгоритма.

Докажем асимптотику алгоритма $O(N \log N \log N)$. Для каждого простого числа p будет выполнено $\frac{n}{p}$ действий по вычеркиванию. То есть общее количество действий будет равно:

$$\sum_{p \leq N, p \in \text{primes}} \frac{N}{p} = N \times \sum_{1 \leq N, p \in \text{primes}} \frac{1}{p}$$

Поскольку количество простых чисел до N примерно равно $\frac{N}{\ln N}$, то k -ок по счету простое число примерно равно $k \ln k$. Отдельно выделив простое число 2 получим:

$$N \times \sum_{1 \leq N, p \in \text{primes}} \frac{1}{p} \approx \frac{1}{2} + \sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k}$$

Эта формула является приближением интеграла непрерывной монотонной функции методом прямоугольников, поэтому можно считать, что:

$$\sum_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k} \approx \int_{k=2}^{\frac{n}{\ln n}} \frac{1}{k \ln k} dk = \ln \ln \frac{n}{\ln n} - \ln \ln 2 = \ln(\ln n - \ln \ln n) - \ln \ln 2 \approx \ln \ln n$$

Подставляя в начальную формулу, получим:

$$\sum_{p \leq N, p \in \text{primes}} \frac{N}{p} \approx n \ln \ln n$$

Что и требовалось доказать.

Решето Эратосфена довольно требовательно к памяти, поэтому имеет смысл использовать на переменные типа `bool`, а использовать битовое сжатие. Это не только сократит потребление памяти в 8 раз, но и ускорит работу, т.к. чаще будут происходить обращения к памяти, хранящейся в кэше процессора.

С учетом того, что у всех составных чисел до N существует простой делитель, не превосходящий \sqrt{N} , то просеивание можно продолжать до тех пор, пока квадрат очередного числа не превосходит N .

Также можно ускорить решето Эратосфена полностью избавившись от четных чисел, что позволяет сократить потребление памяти и ускорить работу алгоритма вдвое.

Кроме того, вместо пометки для составного числа можно записывать туда первый найденный простой делитель, это позволит проводить факторизацию любого числа до N за $O(\log N)$ (количество делителей простого числа N , очевидно, не превосходит $\log_2 N$). Если, кроме этого, в отдельном массиве сохранять еще и второй сомножитель простого делителя, то можно реализовать операцию факторизации без использования деления, что также неасимптотически ускоряет этот процесс.

11.4 НОД и НОК

Наибольшим общим делителем (НОД, по-английски Greatest Common Divisor, GCD) двух натуральных чисел называется такое максимальное натуральное число, которое является делителем и первого и второго числа. Наименьшим общим кратным (НОК, по-английски Least Common Multiplier, lcm) двух натуральных чисел называется такое минимальное натуральное число, которое делится нацело на оба этих числа.

Аналогично вводятся понятия НОД и НОК многих чисел. На практике НОД и НОК многих чисел считаются с помощью последовательно попарного подсчета НОД и НОК (т.е. считается НОД уже обработанной части и очередного числа).

В младшей школе изучается алгоритм Евклида, который позволяет найти НОД двух чисел. В нем используются следующие соотношения:

- 1) $\text{НОД}(a, 0) = a$
- 2) $\text{НОД}(a, b) = \text{НОД}(a \% b, b) = \text{НОД}(a, b \% a)$

Эта идея в виде функции выглядит следующим образом:

```
int gcd(int a, int b)
{
```

```

while (b != 0) {
    a %= b;
    swap(a, b);
}
return a;
}

```

Этот алгоритм очень прост, его сложность в худшем случае, равна $O(\log N)$, где N — большее из чисел.

НОК можно искать исходя из соотношения $\text{НОД}(a, b) \times \text{НОК}(a, b) = a \times b$. Чтобы избежать переполнения лучше всего пользоваться формулой $\text{lcm}(a, b) = a / \text{gcd}(a, b) * b$.

Существует еще один способ поиска НОД: бинарный алгоритм Евклида. Этот способ основывается на соотношениях $\text{НОД}(2 \times a, 2 \times b) = 2 \times \text{НОД}(a, b)$, $\text{НОД}(2 \times a, 2 \times b + 1) = \text{НОД}(a, 2 \times b + 1)$, $\text{НОД}(2 \times a + 1, 2 \times b + 1) = \text{НОД}(2 \times (a - b), 2 \times b + 1)$. Хотя при использовании этих соотношений время написания программы и собственно поиска НОД несколько возрастет (хотя по прежнему сложность алгоритма будет составлять $O(\log n)$), этот способ намного удобнее при поиске НОД длинных чисел. Действительно, в обычном алгоритме Евклида необходима операция взятия остатка от деления длинных чисел, а в бинарном — намного более простые операции длинного вычитания и деления на 2.

11.5 Полезные свойства факторизации

Выпишем представление числа в виде степеней его простых делителей. Например, пусть у нас есть массив простых чисел $[2, 3, 5, 7, 11, 13]$, тогда массив с представлением числа 2600 будет выглядеть как $[3, 0, 2, 0, 0, 1]$ из которого можно получить $2^3 \times 5^2 \times 13^1 = 2600$.

Такое представление неоправданно генерировать для одного числа (если это не является необходимым условием для решения задачи). Однако, если чисел много, то приведение их в такое представление и обратно может быть весьма полезным.

Например, с помощью такого представления очень просто реализовать умножение. Рассмотрим наше число 2600 и число 11858, которое представляется массивом $[1, 0, 0, 2, 2, 0]$. Чтобы получить произведение этих чисел, достаточно сложить соответствующие элементы массивов. Результатом будет массив $[4, 0, 2, 2, 2, 1]$, т.е. $2^4 \times 5^2 \times 7^2 \times 11^2 \times 13^1 = 30830800$, что совпадает с результатом умножения.

После некоторых размышлений можно также реализовать деление с остатком, но в рамках лекции мы этого делать не будем, желающие могут сами придумать алгоритм.

Из этого представления также можно получить НОД и НОК этих чисел.

Для нахождения НОД надо выбирать минимум из степеней (это логично, т.к. число X^n кратно X^k , если $n \geq k$). Для подсчета НОК надо брать максимум из степеней. Этот же метод работает для подсчета НОД и НОК произвольного количества чисел.

Для наших чисел 2600 и 11858, НОД, подсчитанный таким образом, представим массивом $[1, 0, 0, 0, 0, 0]$, т.е. равен 2, а НОК — массивом $[3, 0, 2, 2, 2, 1]$ и равен 15415400. С помощью алгоритма Евклида несложно убедиться, что результат верен.

11.6 Быстрое возведение в степень

Довольно часто возникает задача быстрого возведения числа или другого объекта, для которого определена ассоциативная операция умножения, в какую-либо степень. Наивный алгоритм, когда мы просто нужное число раз умножаем число на само себя, имеет сложность $O(N)$, где N — показатель степени (если считать, что сложность умножения $O(1)$). Поскольку при возведении в степень числа растут очень быстро, то в реальных задачах часто ищется значение по модулю какого-либо числа.

Рассматривая степени некоторых чисел, можно догадаться о методе, которым следует пользоваться. Например, для возведения 3 в четвертую степень нам нужно сделать три операции умножения. Однако, если изменить порядок действий на такой: $(3^2)^2$, то потребуется всего два умножения. Следует помнить, что при возведении числа в степени еще в какую-либо степень показатели степеней перемножаются. Именно на сокращении четных степеней основывается идея быстрого возведения в степень.

Каждый раз, когда степень четная, мы возводим вспомогательную переменную в квадрат, а показатель степени делим на 2. Если число нечетное, то умножаем текущее вспомогательное число на результат и уменьшаем показатель степени на 1. Таким образом хотя бы один раз из двух у нас произойдет уменьшение показателя степени вдвое и, исходя из этого, мы получим сложность алгоритма $O(\log N)$.

Текст функции, которая возводит число a в степень n :

```
int binpow(int a, int n)
{
    if (n == 0)
        return 1;
    if (n % 2 == 1)
        return binpow(a, n - 1) * a;
    int b = binpow(a, n / 2);
    return b * b;
}
```

Для повышения эффективности работы этой функции следует избавиться от рекурсии.

Вместо чисел можно использовать такие объекты, как перестановка или матрица. Например, в случае возведения матрицы смежности неориентированного графа в степень $k - 1$ мы получим количество путей между вершинами i и j длиной ровно k .