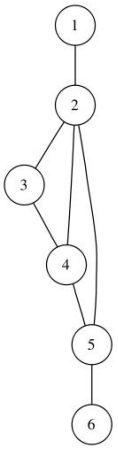
Вступление

Все задачи этого тура были на теорию графов, поэтому введем необходимые определения.

Теория графов

Основные понятия



Граф — это совокупность непустого множества вершин и множества пар вершин.

Вершина, **Узел** — точка, где могут сходиться/выходить рёбра и/или дуги. Множество вершин графа G обозначается V(G).

Ребро соединяет две вершины графа.

Инцидентность — понятие, используемое только в отношении ребра и вершины: если v_1,v_2 — вершины, а e = (v_1,v_2) — соединяющее их ребро, тогда вершина v_1 и ребро e инцидентны, вершина v_2 и ребро e тоже инцидентны. Две вершины (или два ребра) инцидентными быть не могут. Для обозначения ближайших вершин (рёбер) используется понятие смежности.

Смежность — понятие, используемое в отношении только двух рёбер либо только двух вершин: Два ребра, инцидентные одной вершине, называются **смежными**; две вершины, инцидентные одному ребру, также называются **смежными**

Неориентированный граф с 6 вершинами и 7 ребрами.

Ориентированный граф - это пара (V, E), где V - конечное множество вершин (узлов, точек) графа, а E - некоторое множество пар вершин, т.е. подмножество множества V x V или бинарное отношение на V. Элементы E называют ребрами (дугами, стрелками, связями). Для ребра $e = (u, v) \in E$ вершина u называется началом e, а вершина v - концом e, говорят, что ребро e ведет из u в v.

Неориентированный граф G=(V, E) - это ориентированный граф, у которого для каждого ребра $(u, v) \in E$ имеется противоположное ребро $(v, u) \in E$, т.е. отношение E симметрично. Такая пара (u,v), (v,u) называется неориентированным ребром. Для его задания можно использовать обозначение для множества концов: {u, v}, но чаще используется указание одной из пар в круглых скобках. Если $e = (u, v) \in E$, то вершины u и v называются смежными в G, а ребро e и эти вершины называются инцидентными. Степенью вершины в неориентированном графе называется число смежных с ней вершин. Вершина степени 0 называется изолированной.

Компонента связности графа — некоторое множество вершин графа такое, что для любых двух вершин из этого множества существует путь из одной в другую, и не существует пути из вершины этого множества в вершину не из этого множества.

Связный граф — граф, содержащий ровно одну компоненту связности. Это означает, что между любой парой вершин этого графа существует как минимум один путь.

Двудо́льный граф или **бигра́ф** — это математический термин теории графов, обозначающий граф, множество вершин которого можно разбить на две части таким образом, что каждое ребро графа соединяет какую-то вершину из одной части с какой-то вершиной другой части, то есть не существует ребра, соединяющего две вершины из одной и той же части.

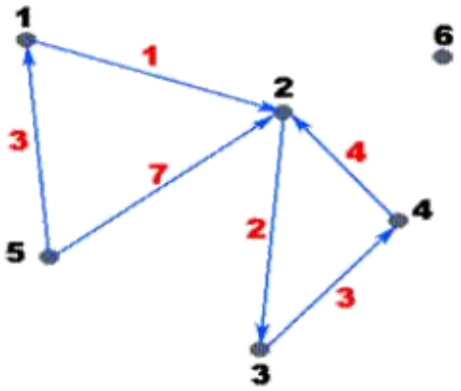
Способы представления графов

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

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

Матрица смежности

Один из самых распространённых способов хранения графа - матрица смежности. Она представляет собой двумерный массив. Если в клетке i, j (i – строка, j - столбец) установлено значение пусто (как правило, это очень большая величина или величина, которой заведомо не может равняться вес ребра), то дуги, начинающейся в вершине i и кончающейся в вершине j, нет. Иначе дуга есть. Если она есть, то в соответствующую ячейку записывают ее вес. Если граф не взвешенный, то вес дуги считается равным единице. Составим матрицу смежности для нашего графа:



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

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

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

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

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

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

Приведем расчеты временной сложности хранения графа матрицей смежности:

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

Список дуг

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

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

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

Приведем расчеты временной сложности хранения графа списком дуг:

Операция	**Временная сложность**
Проверка смежности вершин x и y	O(E)
Перечисление всех вершин смежных с x	O(E)
Определение веса ребра (x,y)	O(E)
Перечисление всех ребер (x, y)	O(E)
Поиск i-ой дуги	O(1)

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

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

Списки смежных вершин

Данный метод хранения графа наиболее часто используется при решении задач с большими ограничениями по памяти и по времени (например, 2 сек и 64 Мбайт памяти на ограничения $V < 10000$, $E < 1000$ – случай разреженного графа). Его суть заключается в том, чтобы для каждой вершины T хранить номера вершин, в которые можно попасть из T, и вес соответствующих ребер (дуг). Для этого необходимо реализовать такую структуру данных, как список. В языках высокого уровня, таких, как C++ и Java, они уже реализованы. Далее нужно реализовать структуру данных (struct в C++ и record в Pascal) – запись с полями «вершина» и «вес». После этого заводим массив списков с V ячейками (количество вершин) и к каждой ячейке «привязываем» список созданного нами типа (структуры).

Эта схема поможет воспринять метод:

mass[0] : . . . . . .

mass[1] : . . . . . .

…..

mass[v] : . . . . .

где «….» – это длина списка, количество вершин, исходящих из i-ой вершины графа.

Для данного примера массив списков будет выглядеть так (в скобках указаны элементы структуры):

Номер ячейки : элементы списка.

1 : (2, 1), (5, 3)
2 : (3, 2)
3 : (4, 3)
4 : (2, 4)
5 : (1, 3), (2, 7)
6 :

Приведем расчеты временной сложности хранения графа списками смежных вершин:

Операция	**Временная сложность**
Проверка смежности вершин x и y	O(E)
Перечисление всех вершин смежных с x	O(E)
Определение веса ребра (x,y)	O(E)
Перечисление всех ребер (x, y)	O(E)
Поиск i-ой дуги	O(1)

Времена у всех операций, вроде бы, такие же, как и у списка рёбер. Однако, большинство из них реально намного меньше. Например, проверка смежности вершин x и y и перечисление всех вершин смежных с x в списке рёбер гарантированно будет выполнятся O(E) операций, т.к. нам обязательно нужно пробежать весь список, а в списке смежности мы бежим только по вершинам, смежным с x.

Кроме того, мы экономим колоссальное количество памяти. Подсчитаем примерный объем для максимального ограничения: $2 * (10000 * 1000) / 1024^2 \approx 19$ Мбайт.

Как видно, это в 10 раз меньше, чем при хранении матрицей смежности, при одинаковых ограничениях. Стоит ещё раз отметить, что, если граф неразреженный, то особой разницы хранения графа нет. Главное, чтобы способ был удобен для выполнения поставленной задачи.

Итак, мы рассмотрели лишь некоторые методы представления графа в памяти компьютера. Существует ещё много способов представления графа в памяти, как статических, так и динамических. Однако не стоит

распыляться на все сразу, так как они мало чем могут помочь на олимпиаде по информатике. Все методы, рассмотренные выше, хороши для решения школьных олимпиадных задач, не требуют долгой и сложной реализации, поэтому нужно иметь их на вооружении. Можно сделать выводы, что наиболее оптимальной структурой является массив списков, так как он экономит память и не тратит драгоценное время. Но чтобы им смело пользоваться, надо много потренироваться в его реализации.

Методы обхода графа

Поиск в глубину(DFS)

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

Общая идея

Общая идея алгоритма состоит в следующем: для каждой *не пройденной* вершины необходимо найти все *не пройденные* смежные вершины и повторить поиск для них.

Пошаговое представление

1. Выбираем любую вершину из еще *не пройденных*, обозначим ее как u.
2. Запускаем процедуру $dfs(u)$
   - Помечаем вершину u как *пройденную*
   - Для каждой *не пройденной* смежной с u вершиной (назовем ее v) запускаем $dfs(v)$
3. Повторяем шаги 1 и 2, пока все вершины не окажутся *пройденными*.

Цвета вершин

Зачастую, простой информации "были/не были в вершине" не хватает для конкретных целей. Поэтому в процессе алгоритма вершинам задают некоторые цвета:

- если вершина *белая*, значит, мы в ней еще не были, вершина *не пройдена*;
- *серая* - вершина *проходится* в текущей процедуре dfs;
- *черная* - вершина *пройдена*, все итерации dfs от нее завершены.

Такие "метки" в основном используются при поиске цикла.

Время работы

Оценим время работы обхода в глубину. Процедура dfs вызывается от каждой вершины не более одного раза, а внутри процедуры рассматриваются все такие ребра $\{e \mid begin(e) = u\}$. Всего таких ребер для всех вершин в графе $O(E)$, следовательно, время работы алгоритма оценивается как $O(V + E)$.

Применения алгоритма

- Поиск любого пути в графе.
- Поиск лексикографически первого пути в графе.
- Проверка, является ли одна вершина дерева предком другой: В начале и конце итерации поиска в глубину будет запоминать "время" захода и выхода в каждой вершине. Теперь за O(1) можно найти ответ: вершина i является предком вершины j тогда и только тогда, когда $start_i < start_j$ и $end_i > end_j$.
- Задача LCA (наименьший общий предок).
- Топологическая сортировка: Запускаем серию поисков в глубину, чтобы обойти все вершины графа. Отсортируем вершины по времени выхода по убыванию - это и будет ответом.
- Проверка графа на ацикличность и нахождение цикла
- Поиск компонент сильной связности: Сначала делаем топологическую сортировку, потом транспонируем граф и проводим снова серию поисков в глубину в порядке, определяемом топологической сортировкой. Каждое дерево поиска - сильносвязная компонента.

- Поиск мостов: Сначала превращаем граф в ориентированный, делая серию поисков в глубину, и ориентируя каждое ребро так, как мы пытались по нему пройти. Затем находим сильносвязные компоненты. Мостами являются те рёбра, концы которых принадлежат разным сильносвязным компонентам.

Поиск в ширину (BFS)

BFS является одним из самых основных алгоритмов, составляющих основу многих других.

Общая идея

Простое описание (рисунок будет ниже). Мы сейчас стоим в некоторой стартовой (завод) вершине [s], из которой по ребрам видны только соседние вершины. И нам очень нужно как можно скорее попасть в вершину [t], которая находится где-то в этом графе. Далее мы поступаем так. Просматриваем по ребрам (а именно свободным дорогам) нашей вершины соседей: есть ли среди них [t]. Если нет, то записываем всех (впервые обнаруженных) соседей в очередь «нужно там побывать». Когда просмотрели всех соседей — отмечаем свою вершину – «тут уже побывали». Достаем первую непосещенную вершину из очереди и идем в нее.

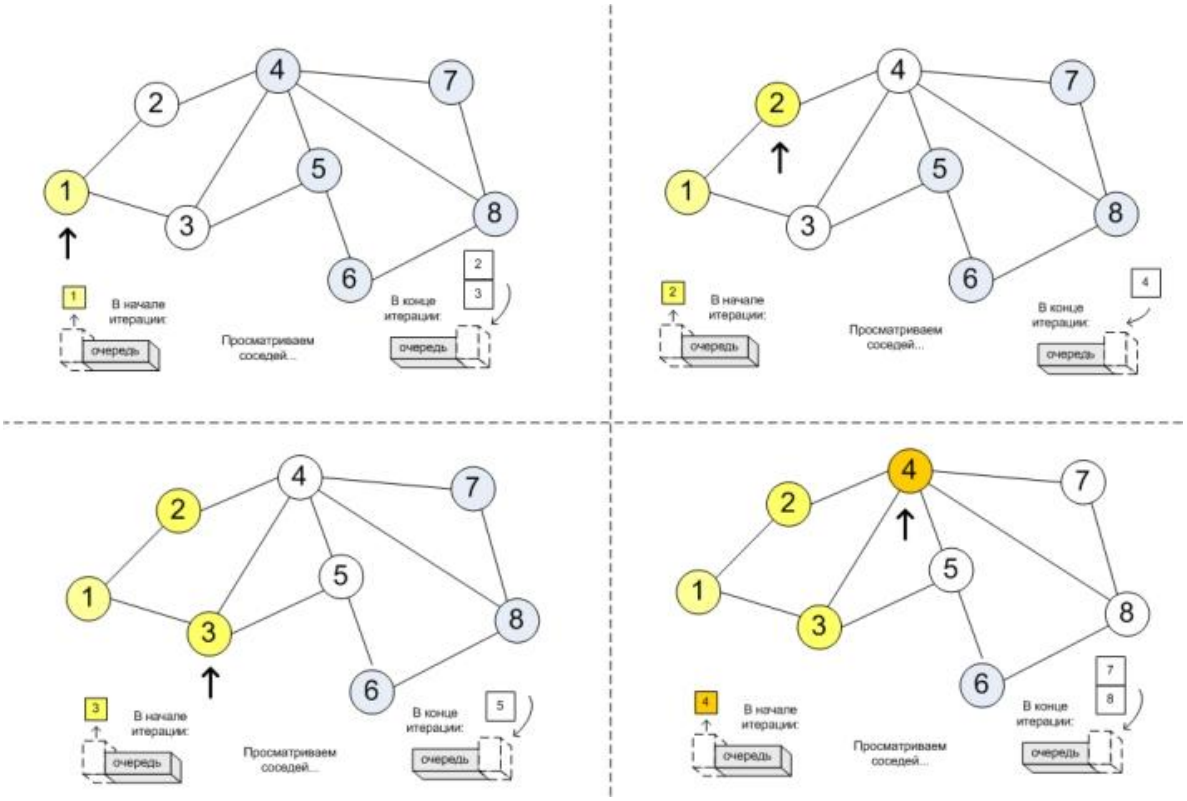
Продолжаем поиски таким же образом. При этом те вершины, в которых однажды побывали — игнорируем (ни шагу назад). Если по дороге встретили [t] – отлично, цель достигнута!

Для того, чтобы не заезжать в одни и те же перекрестки по нескольку раз, мы будем их отмечать в массиве mark[..]. После осмотра соседей из [u] вершины, ставим отметку mark[u] = 1 – значит, что на [u]-ом перекрестке мы «уже побывали».

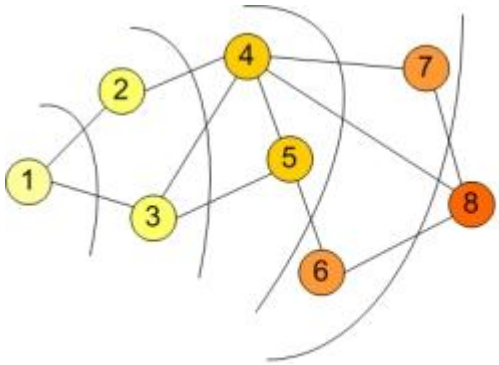
Сам алгоритм можно понимать как процесс "поджигания" графа: на нулевом шаге поджигаем только вершину s. На каждом следующем шаге огонь с каждой уже горящей вершины перекидывается на всех её соседей; т.е. за одну итерацию алгоритма происходит расширение "кольца огня" в ширину на единицу (отсюда и название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

Пример работы

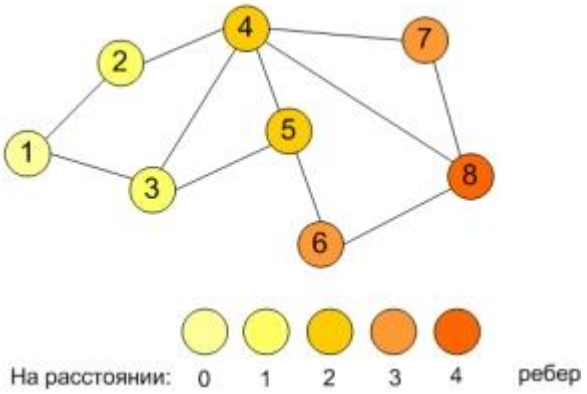
На рисунке: в вершинах – написаны порядковые номера



После завершения алгоритма получим следующую картину:



В конечном итоге:



Предки вершин и расстояния

Также можно посчитать длины кратчайших путей (для чего просто надо завести массив длин путей $d[]$), и компактно сохранить информацию, достаточную для восстановления всех этих кратчайших путей (для этого надо завести массив "предков" $p[]$, в котором для каждой вершины хранить номер вершины, из которой мы попали в эту вершину). Расстояние до текущей вершины будет на 1 больше, чем расстояние до ее предка(*d[u] = d[p[u]] + 1, d[start] = 0,* где *start* – вершина, из которой запущен обход).

Время работы

Оценим время работы для входного графа $G = (V, E)$. В очередь добавляются только непосещенные вершины, поэтому каждая вершина посещается не более одного раза. Операции внесения в очередь и удаления из нее требуют $O(1)$ времени, так что общее время работы с очередью составляет $O(|V|)$ операций. Для каждой вершины v рассматривается не более $deg\ v$ ребер, инцидентных ей. Так как $\sum_{v \in V} deg\ v = 2|E|$, то время, используемое на работу с ребрами, составляет $O(|E|)$. Поэтому общее время работы алгоритма поиска в ширину — $O(|V| + |E|)$.

Применения алгоритма

- Поиск **кратчайшего пути** в невзвешенном графе.
- Поиск **компонент связности** в графе за $O(n + m)$.

  Для этого мы просто запускаем обход в ширину от каждой вершины, за исключением вершин, оставшихся посещёнными ($\text{used} = \text{true}$) после предыдущих запусков. Таким образом, мы выполняем обычный запуск в ширину от каждой вершины, но не обнуляем каждый раз массив $\text{used}[]$, за счёт чего мы каждый раз будем обходить новую компоненту связности, а суммарное время работы алгоритма составит по-прежнему $O(n + m)$ (такие несколько запусков обхода на графе без обнуления массива used называются серией обходов в ширину).

- Нахождения решения какой-либо задачи (игры) **с наименьшим числом ходов**, если каждое состояние системы можно представить вершиной графа, а переходы из одного состояния в другое — рёбрами графа.

  Классический пример — игра, где робот двигается по полю, при этом он может передвигать ящики, находящиеся на этом же поле, и требуется за наименьшее число ходов передвинуть ящики в требуемые позиции. Решается это обходом в ширину по графу, где состоянием (вершиной) является набор координат: координаты робота, и координаты всех коробок.

- Нахождение кратчайшего пути в 0-1-**графе** (т.е. графе взвешенном, но с весами равными только 0 либо 1): достаточно немного модифицировать поиск в ширину: если текущее ребро нулевого веса, и происходит улучшение расстояния до какой-то вершины, то эту вершину добавляем не в конец, а в начало очереди.
- Нахождение **кратчайшего цикла** в неориентированном невзвешенном графе: производим поиск в ширину из каждой вершины; как только в процессе обхода мы пытаемся пойти из текущей вершины по какому-то ребру в уже посещённую вершину, то это означает, что мы нашли кратчайший цикл, и останавливаем обход в ширину; среди всех таких найденных циклов (по одному от каждого запуска обхода) выбираем кратчайший.
- Найти все рёбра, лежащие **на каком-либо кратчайшем пути** между заданной парой вершин (a, b). Для этого надо запустить 2 поиска в ширину: из a, и из b. Обозначим через $d_a[]$ массив кратчайших расстояний, полученный в результате первого обхода, а через $d_b[]$ — в результате второго обхода. Теперь для любого ребра (u, v) легко проверить, лежит ли он на каком-либо кратчайшем пути: критерием будет условие $d_a[u] + 1 + d_b[v] = d_a[b]$.
- Найти все вершины, лежащие **на каком-либо кратчайшем пути** между заданной парой вершин (a, b). Для этого надо запустить 2 поиска в ширину: из a, и из b. Обозначим через $d_a[]$ массив кратчайших расстояний, полученный в результате первого обхода, а через $d_b[]$ — в результате второго обхода. Теперь для любой вершины v легко проверить, лежит ли он на каком-либо кратчайшем пути: критерием будет условие $d_a[v] + d_b[v] = d_a[b]$.
- Найти **кратчайший чётный путь** в графе (т.е. путь чётной длины). Для этого надо построить вспомогательный граф, вершинами которого будут состояния (v, c), где v — номер текущей вершины, $c = 0 \dots 1$ — текущая чётность. Любое ребро (a, b) исходного графа в этом новом графе превратится в два ребра $((u, 0), (v, 1))$ и $((u, 1), (v, 0))$. После этого на этом графе надо обходом в ширину найти кратчайший путь из стартовой вершины в конечную, с чётностью, равной 0.

Алгоритм Флойда-Уоршелла нахождения кратчайших путей между всеми парами вершин

Дан ориентированный или неориентированный взвешенный граф G с n вершинами. Требуется найти значения всех величин d_{ij} — длины кратчайшего пути из вершины i в вершину j.

Предполагается, что граф не содержит циклов отрицательного веса (тогда ответа между некоторыми парами вершин может просто не существовать — он будет бесконечно маленьким).

Этот алгоритм был одновременно опубликован в статьях Роберта Флойда (Robert Floyd) и Стивена Уоршелла (Варшалла) (Stephen Warshall) в 1962 г., по имени которых этот алгоритм и называется в настоящее время. Впрочем, в 1959 г. Бернард Рой (Bernard Roy) опубликовал практически такой же алгоритм, но его публикация осталась незамеченной.

Описание алгоритма

Ключевая идея алгоритма — разбиение процесса поиска кратчайших путей на **фазы**.

Перед k-ой фазой ($k = 1 \dots n$) считается, что в матрице расстояний $d[][]$ сохранены длины таких кратчайших путей, которые содержат в качестве внутренних вершин только вершины из множества $\{1, 2, \dots, k-1\}$ (вершины графа мы нумеруем, начиная с единицы).

Иными словами, перед k-ой фазой величина $d[i][j]$ равна длине кратчайшего пути из вершины i в вершину j, если этому пути разрешается заходить только в вершины с номерами, меньшими k (начало и конец пути не считаются).

Легко убедиться, что чтобы это свойство выполнилось для первой фазы, достаточно в матрицу расстояний $d[][]$ записать матрицу смежности графа: $d[i][j] = g[i][j]$ — стоимости ребра из вершины i в вершину j. При этом, если между какими-то вершинами ребра нет, то записать следует величину

"бесконечность" ∞. Из вершины в саму себя всегда следует записывать величину 0, это критично для алгоритма.

Пусть теперь мы находимся на k-ой фазе, и хотим **пересчитать** матрицу $d[][]$ таким образом, чтобы она соответствовала требованиям уже для $k+1$-ой фазы. Зафиксируем какие-то вершины i и j. У нас возникает два принципиально разных случая:

- Кратчайший путь из вершины i в вершину j, которому разрешено дополнительно проходить через вершины $\{1, 2, \ldots, k\}$, **совпадает** с кратчайшим путём, которому разрешено проходить через вершины множества $\{1, 2, \ldots, k-1\}$.

  В этом случае величина $d[i][j]$ не изменится при переходе с k-ой на $k+1$-ую фазу.

- "Новый" кратчайший путь стал **лучше** "старого" пути.

  Это означает, что "новый" кратчайший путь проходит через вершину k. Сразу отметим, что мы не потеряем общности, рассматривая далее только простые пути (т.е. пути, не проходящие по какой-то вершине дважды).

  Тогда заметим, что если мы разобьём этот "новый" путь вершиной k на две половинки (одна идущая $i \Rightarrow k$, а другая — $k \Rightarrow j$), то каждая из этих половинок уже не заходит в вершину k. Но тогда получается, что длина каждой из этих половинок была посчитана ещё на $k-1$-ой фазе или ещё раньше, и нам достаточно взять просто сумму $d[i][k] + d[k][j]$, она и даст длину "нового" кратчайшего пути.

Объединяя эти два случая, получаем, что на k-ой фазе требуется пересчитать длины кратчайших путей между всеми парами вершин i и j следующим образом:

```
new_d[i][j] = min (d[i][j], d[i][k] + d[k][j]);
```

Таким образом, вся работа, которую требуется произвести на k-ой фазе — это перебрать все пары вершин и пересчитать длину кратчайшего пути между ними. В результате после выполнения n-ой фазы в матрице расстояний $d[i][j]$ будет записана длина кратчайшего пути между i и j, либо ∞, если пути между этими вершинами не существует.

Последнее замечание, которое следует сделать, — то, что можно **не создавать отдельную матрицу** $\text{new\_d}[][]$ для временной матрицы кратчайших путей на k-ой фазе: все изменения можно делать сразу в матрице $d[][]$. В самом деле, если мы улучшили (уменьшили) какое-то значение в матрице расстояний, мы не могли ухудшить тем самым длину кратчайшего пути для каких-то других пар вершин, обработанных позднее.

Асимптотика алгоритма, очевидно, составляет $O(n^3)$.

Восстановление самих путей

Легко поддерживать дополнительную информацию — так называемых "предков", по которым можно будет восстанавливать сам кратчайший путь между любыми двумя заданными вершинами **в виде последовательности вершин**.

Для этого достаточно кроме матрицы расстояний $d[][]$ поддерживать также **матрицу предков** $p[][]$, которая для каждой пары вершин будет содержать номер фазы, на которой было получено кратчайшее расстояние между ними. Понятно, что этот номер фазы является не чем иным, как "средней" вершиной искомого кратчайшего пути, и теперь нам просто надо найти кратчайший путь между вершинами i и $p[i][j]$, а также между $p[i][j]$ и j. Отсюда получается простой рекурсивный алгоритм восстановления кратчайшего пути.

Система непересекающихся множеств

Условие

Поставим перед собой следующую задачу. Пускай мы оперируем элементами N видов (для простоты, здесь и далее — числами от 0 до N-1). Некоторые группы чисел объединены в множества. Также мы можем добавить в структуру новый элемент, он тем самым образует множество размера 1 из самого себя. И наконец, периодически некоторые два множества нам потребуется сливать в одно.

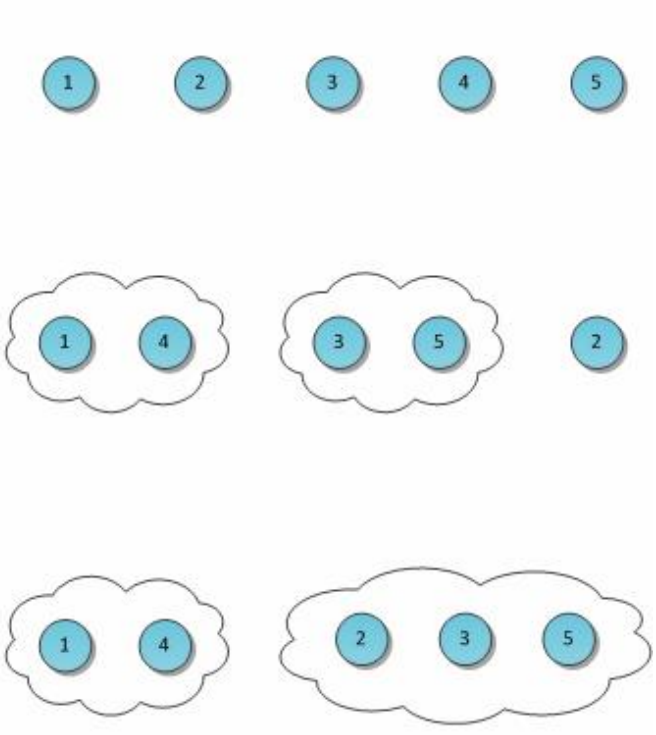
Формализируем задачу: создать *быструю* структуру, которая поддерживает следующие операции:

MakeSet(X) — внести в структуру новый элемент X, создать для него множество размера 1 из самого себя. Find(X) — возвратить *идентификатор* множества, которому принадлежит элемент X. В качестве идентификатора мы будем выбирать один элемент из этого множества — *представителя* множества. Гарантируется, что для одного и того же множества представитель будет возвращаться один и тот же, иначе невозможно будет работать со структурой: не будет корректной даже проверка принадлежности двух элементов одному множеству if (Find(X) == Find(Y)).

Unite(X, Y) — объединить два множества, в которых лежат элементы X и Y, в одно новое.

На рисунке я продемонстрирую работу такой гипотетической структуры.

```
MakeSet(1);
MakeSet(2);
MakeSet(3);
MakeSet(4);
MakeSet(5);

Find(4) = 4


Unite(1, 4);
Unite(3, 5);


Find(4) = 4
Find(1) = 4
Find(2) = 2


Unite(5, 2);


Find(5) = 2
Find(3) = 2
```



Реализация

Классическая реализация DSU была предложена Bernard Galler и Michael Fischer в 1964 году, однако исследована (включая временную оценку сложности) Робертом Тарьяном уже в 1975. Тарьяну же принадлежат некоторые результаты, улучшения и применения, о которых мы сегодня ещё поговорим.

Хранить структуру данных будем в виде леса, то есть превратим DSU в систему непересекающихся деревьев. Все элементы одного множества лежат в одном соответствующем дереве, представитель дерева — его корень, слияние множеств суть просто объединение двух деревьев в одно. Как мы увидим, такая идея вкупе с двумя небольшими эвристиками ведет к поразительно высокому быстродействию получившейся структуры.

Для начала нам потребуется массив p, хранящий для каждой вершины дерева её непосредственного предка (а для корня дерева X — его самого). С помощью одного только этого массива можно эффективно реализовать две первые операции DSU:

MakeSet(X)

Чтобы создать новое дерево из элемента X, достаточно указать, что он является корнем собственного дерева, и предка не имеет.

```
public void MakeSet(int x)
{
   p[x] = x;
}
```

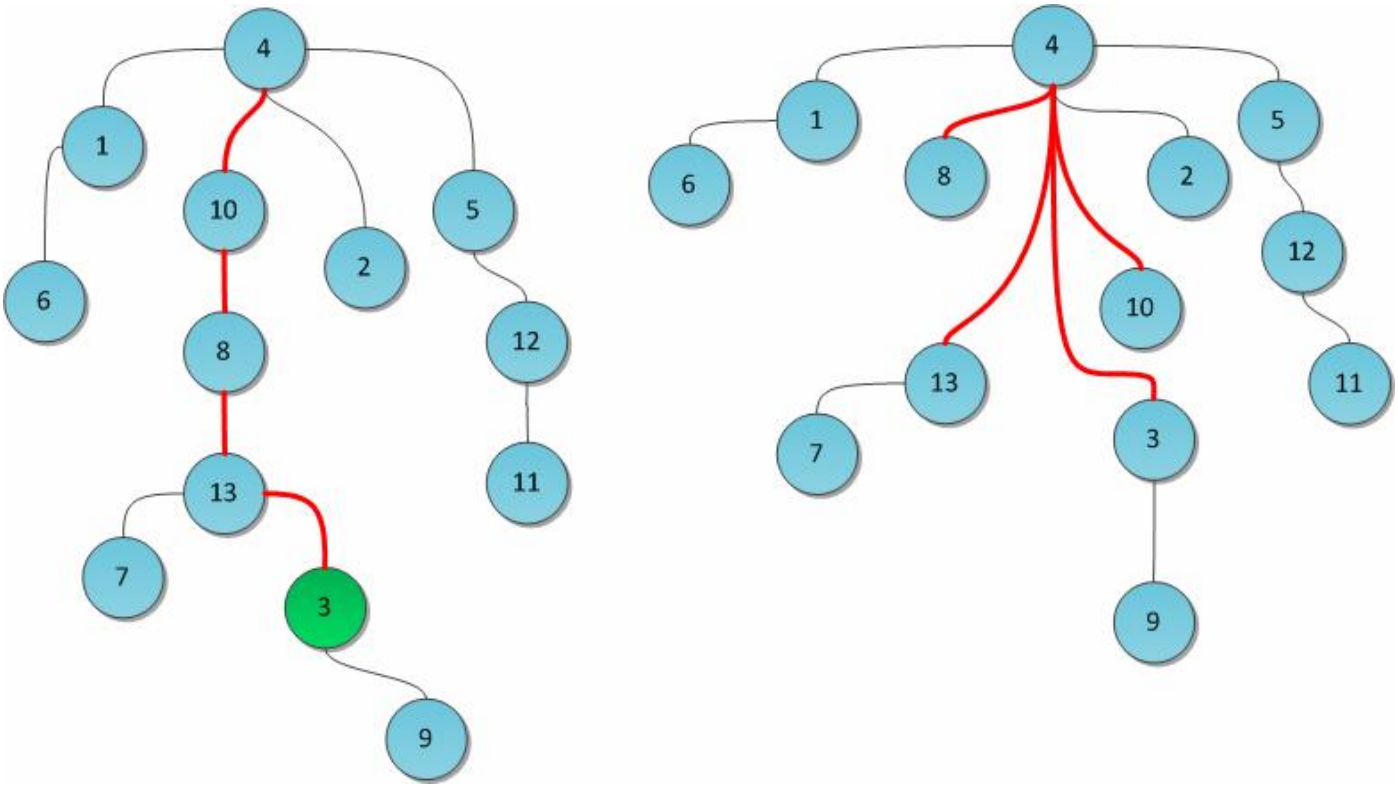
Find(X)

Представителем дерева будем считать его корень. Тогда для нахождения этого представителя достаточно подняться вверх по родительским ссылкам до тех пор, пока не наткнемся на корень.

Но это еще не все: такая наивная реализация в случае вырожденного (вытянутого в линию) дерева может работать за O(N), что недопустимо. Можно было бы попытаться ускорить поиск. Например, хранить не только непосредственного предка, а большие таблицы логарифмического подъема вверх, но это требует много памяти. Или хранить вместо ссылки на предка ссылку на собственно корень — однако тогда при слиянии деревьев (Unite) придется менять эти ссылки всем элементам одного из деревьев, а это опять-таки временные затраты порядка O(N).

Мы пойдем другим путём: вместо ускорения реализации будем просто пытаться не допускать чрезмерно длинных веток в дереве. Это первая эвристика DSU, она называется *сжатие путей* (path compression). Суть эвристики: после того, как представитель таки будет найден, мы для каждой вершины по пути от X к корню изменим предка на этого самого представителя. То есть фактически переподвесим все эти вершины вместо длинной ветви непосредственно к корню. Таким образом, реализация операции Find становится двухпроходной.

На рисунке показано дерево до и после выполнения операции Find(3). Красные ребра — те, по которым мы прошлись по пути к корню. Теперь они перенаправлены. Заметьте, как после этого кардинально уменьшилась высота дерева.



Исходный код в рекурсивной форме написать достаточно просто:

```
public int Find(int x)
{
   if (p[x] == x) return x;
   return p[x] = Find(p[x]);
}
```

Unite(X, Y)

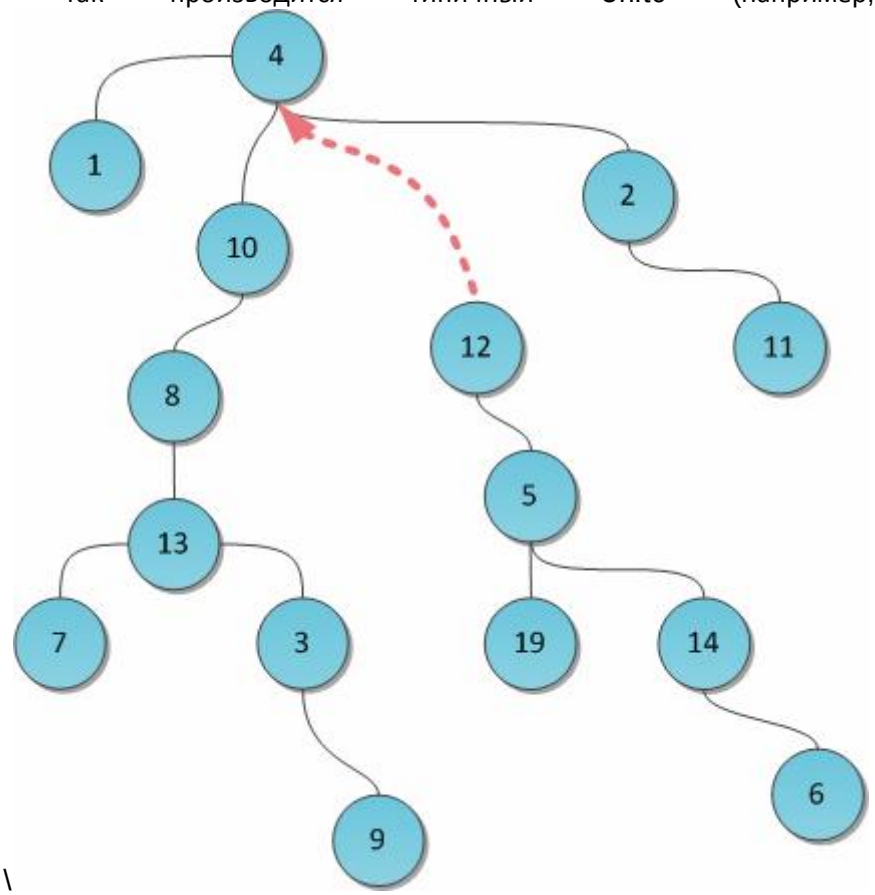
Со слиянием двух деревьев придется чуть повозиться. Найдем для начала корни обоих сливаемых деревьев с помощью уже написанной функции Find. Теперь, помня, что наша реализация хранит только ссылки на непосредственных родителей, для слияния деревьев достаточно было бы просто подвесить один из корней (а с ним и все дерево) сыном к другому. Таким образом все элементы этого дерева автоматически станут принадлежать другому — и процедура поиска представителя будет возвращать корень нового дерева.

Встает вопрос: какое дерево к какому подвешивать? Всегда выбирать какое-то одно, скажем, дерево X, не годится: легко подобрать пример, на котором после N объединений мы получим вырожденное дерево — одну ветку из N элементов. И тут в ход вступает вторая эвристика DSU, направленная на уменьшение высоты деревьев.

Будем хранить помимо предков еще один массив Rank. В нем для каждого дерева будет храниться верхняя граница его высоты — то есть длиннейшей ветви в нем. Заметьте, не сама высота — в процессе выполнения Find длиннейшая ветвь может самоуничтожиться, а тратить еще итерации на нахождение новой длиннейшей ветви слишком дорого. Поэтому для каждого корня в массиве Rank будет записано число, гарантированно больше или равное высоте его дерева.

Теперь легко принять решении о слиянии: чтобы не допустить слишком длинных ветвей в DSU, будем подвешивать более низкое дерево к более высокому. Если их высоты равны — не играет роли, кого подвешивать к кому. Но в последнем случае новоиспеченному корню надо не забыть увеличить Rank.

Вот так производится типичный Unite (например, с параметрами 8 и 19):\



```
public void Unite(int x, int y)
{
   x = Find(x);
   y = Find(y);
   if (rank[x] < rank[y])
      p[x] = y;
   else
   {
      p[y] = x;
      if (rank[x] == rank[y])
         ++rank[x];
   }
}
```

Однако на практике оказывается, что можно и не тратить дополнительные O(N) памяти на махинации с рангами. Достаточно выбирать корень для переподвешивания **случайным образом** — как ни удивительно, но такое решение дает на практике скорость, вполне сравнимую с оригинальной ранговой реализацией. Автор данной статьи всю жизнь пользуется именно рандомизированным DSU, и еще не было случая, когда тот бы подвёл.

Реализация на C++:

```
public void Unite(int x, int y)
{
   x = Find(x);
   y = Find(y);
   if (rand.Next() % 2 == 0)
      swap(x, y);
   p[x] = y;
}
```

Быстродействие

В силу применения двух эвристик скорость работы каждой операции сильно зависит от структуры дерева, а структура дерева — от списка выполненных до того операций. Исключение составляет только MakeSet — её время работы очевидно O(1). Для остальных двух скорость очень неочевидна.

Роберт Тарьян доказал в 1975 г. замечательный факт: время работы как Find, так и Unite на лесе размера N есть $O(\alpha(N))$.

Под $\alpha(N)$ в математике обозначается обратная функция Аккермана, то есть, функция, обратная для $f(N) = A(N, N)$. Функция Аккермана $A(N, M)$ известна тем, что у нее колоссальная скорость роста. К примеру, $A(4, 4) = 2^{2^{2265536}}-3$, это число поистине огромно. Вообще, для всех мыслимых практических значений N обратная функция Аккермана от него не превысит 5. Поэтому её можно принять за константу и считать $O(\alpha(N)) \cong O(1)$.

Итак, имеем:

MakeSet(X)	O(1)
Find(X)	O(1)- амортизированно
Unite(X, Y)	O(1)- амортизированно
Расход памяти	O(N).

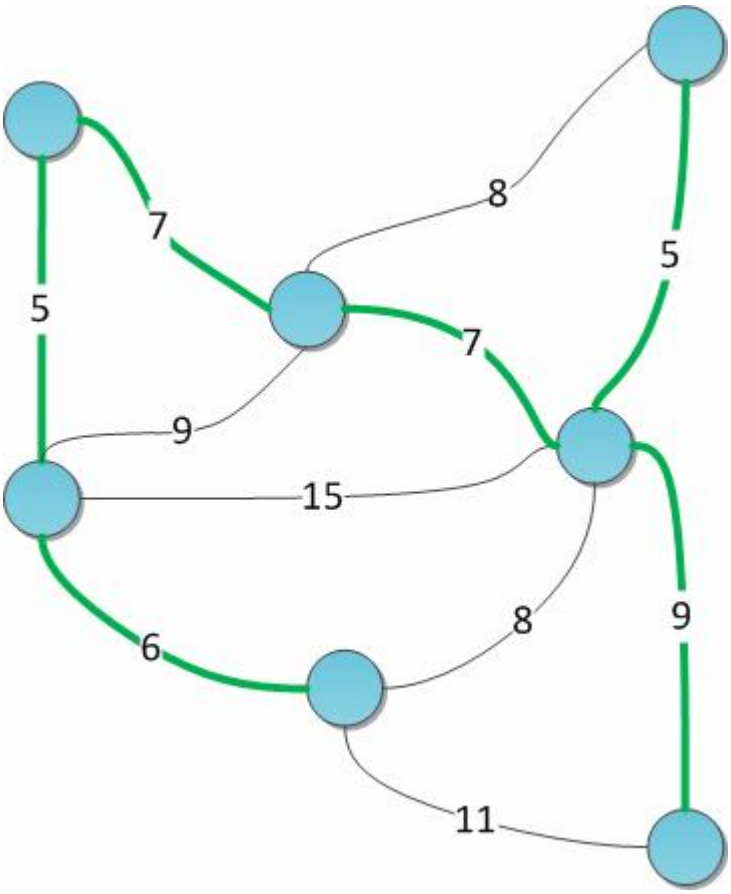
Практические применения

Для DSU известно большое число различных использований. Большинство связано с решением некоторой задачи **в режиме оффлайн** — то есть когда список запросов касательно структуры, которые поступают программе, известен заранее. Я приведу здесь несколько таких задач вместе с краткими идеями решений.

Остов минимального веса

Дан неориентированный связный граф со взвешенными ребрами. Выкинуть из него некоторые ребра так, чтобы в итоге получилось дерево, причем суммарный вес ребер этого дерева должен быть наименьшим.

На рисунке показан взвешенный граф с выделенным минимальным остовом



Алгоритм Краскала для решения этой задачи: отсортируем все ребра по возрастанию веса и будем поддерживать текущий лес минимального веса с помощью DSU. Изначально DSU состоит из N деревьев, по одной вершине в каждом. Идем по ребрам в порядке возрастания; если текущее ребро объединяет разные компоненты — сливаем их в одну и запоминаем ребро как элемент остова. Как только количество компонент достигнет единицы — мы построили искомое дерево.

Компоненты связности в мультиграфе

Дан мультиграф (граф, в котором пара вершин может быть соединена более чем одним непосредственным ребром), к которому поступают запросы вида «удалить некоторое ребро» и «а сколько сейчас в графе компонент связности?» Весь список запросов известен заранее.

Решение банально. Выполним сначала все запросы на удаление, посчитаем количество компонент в итоговом графе, запомним его. Получившийся граф запихнем в DSU. Теперь будем идти по запросам удаления в обратном порядке: каждое удаление ребра из старого графа означает *возможное* слияние двух компонент в нашем «flashback-графе», хранящемся в DSU; в таком случае текущее количество компонент связности уменьшается на единичку.

Генерация лабиринтов

Задача: сгенерировать лабиринт с одним входом и одним выходом.

Алгоритм решения:

Начнем с состояния, когда установлены все стены, за исключением входа и выхода. На каждом шаге алгоритма выберем случайную стену. Если ячейки, между которыми она стоит, еще никак не соединены (лежат в разных компонентах DSU), то уничтожаем её (сливаем компоненты). Продолжаем процесс до некоторого состояния сходимости: например, когда вход и выход соединены; либо, когда осталась одна компонента.

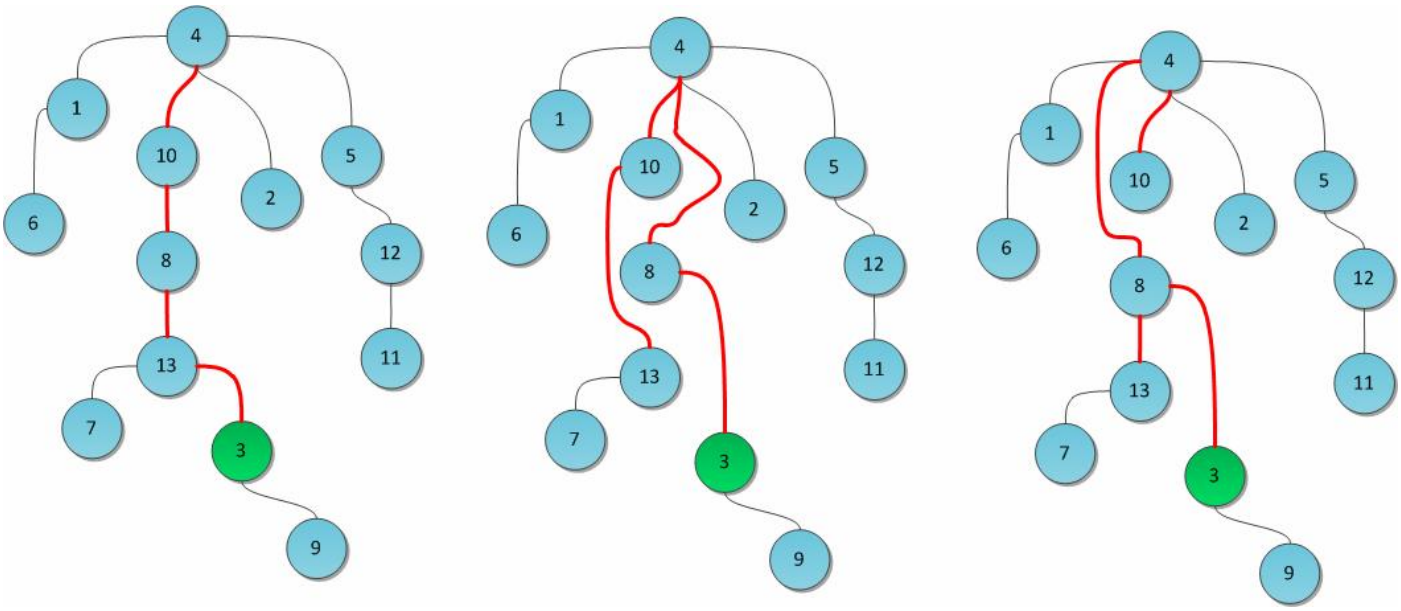
Однопроходные алгоритмы

Существуют варианты реализации Find(X), которые требуют одного прохода до корня, а не двух, однако сохраняют ту же или почти ту же степень быстродействия. Они реализуют другие стратегии сокращения высоты дерева, в отличие от path compression.

Вариант №1: *path splitting*. По пути от вершины X до корня перенаправить родительскую связь каждой вершины с её предка на предка предка (дедушку).

Вариант №2: *path halving*. Взять идею path splitting, однако перенаправлять связи не всех вершин по пути, а только тех, на которых делаем перенаправления — то есть «дедушек».

На рисунке взято все то же дерево, в нем выполняется запрос Find(3). По центру показан результат с применением path splitting, справа — path halving.



Функциональная реализация

У системы непересекающихся множеств есть один большой недостаток: она не поддерживает ни в какой форме операцию Undo, потому что реализована насквозь в императивном стиле. Гораздо удобнее была бы реализация DSU в функциональном стиле, когда каждая операция не изменяет структуру на месте, а возвращает слегка модифицированную новую структуру, в которой произведены требуемые изменения (при этом большая часть памяти у старой и новой структур общая). Такие структуры данных в английской терминологии носят название persistent, они широко применяются в чистом функциональном программировании, где доминирует идея неизменяемости данных.

В силу чисто императивной идеи алгоритмов DSU её функциональная реализация с сохранением быстродействия долгое время казалась немыслимой. Тем не менее, в 2007 году Sylvain Conchon и Jean-Christophe Filliâtre представили в своей работе искомый функциональный вариант, в котором операция Unite возвращает измененную структуру. Если говорить честно, он не совсем функциональный, он использует императивные присваивания, однако они надежно скрыты внутри реализации, а интерфейс persistent DSU — чисто функциональный.

Основная идея реализации — использование структур данных, реализующих «персистентные массивы»: они поддерживают те же операции, что и массивы, однако все так же не модифицируют память, а возвращают измененную структуру. Такой массив можно легко реализовать с помощью какого-нибудь дерева, однако в таком случае операции с ним будут занимать $O(\log_2 N)$ времени, а для DSU такая оценка оказывается уже чрезмерно большой.

Минимальное остовное дерево. Алгоритм Крускала

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

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

Свойства минимального остова

- Минимальный остов **уникален**, **если веса всех рёбер различны**. В противном случае, может существовать несколько минимальных остовов (конкретные алгоритмы обычно получают один из возможных остовов).
- Минимальный остов является также и **остовом с минимальным произведением** весов рёбер. (доказывается это легко, достаточно заменить веса всех рёбер на их логарифмы)
- Минимальный остов является также и **остовом с минимальным весом самого тяжелого ребра**. (это утверждение следует из справедливости алгоритма Крускала)
- **Остов максимального веса** ищется аналогично остову минимального веса, достаточно поменять знаки всех рёбер на противоположные и выполнить любой из алгоритм минимального остова.

Алгоритм Крускала

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

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

Алгоритм Крускала с системой непересекающихся множеств

Реализация с использованием структуры данных "система непересекающихся множеств" (DSU), которая позволит достигнуть асимптотики O (M log N).

Описание

Так же, как и в простой версии алгоритма Крускала, отсортируем все рёбра по неубыванию веса. Затем поместим каждую вершину в своё дерево (т.е. своё множество) с помощью вызова функции DSU MakeSet - на это уйдёт в сумме O (N). Перебираем все рёбра (в порядке сортировки) и для каждого ребра за O (1) определяем, принадлежат ли его концы разным деревьям (с помощью двух вызовов FindSet за O (1)). Наконец, объединение двух деревьев будет осуществляться вызовом Union - также за O (1). Итого мы получаем асимптотику O (M log N + N + M) = O (M log N).

Разбор задач

Задача 11. Охрана империи

В этой задаче необходимо было подсчитать количество ребер в неориентированном графе. Из свойств матрицы смежности неориентированного невзвешеного графа нам известно, что количество единичек в нем равно удвоенному количеству ребер. Исходя из этого мы можем найти ответ, посчитав количество единиц и поделив его пополам.

Задача 12. Патруль

В данной задаче на вход подавался граф в виде матрицы смежности и необходимо было вывести список ребер. Выводя все ребра, где стоят единицы, мы каждое выведем дважды, поэтому один из вариантов как этого избежать – выводить все ребра, которые находятся в описании выше либо ниже главной диагонали матрицы. Матрица будет симметрична относительно главной диагонали, а поэтому рассмотрев только ее часть , мы выведем каждое ребро ровно 1 раз.

Задача 13. Сокращение

В этой задаче необходимо было проверить наличие кратных ребер в графе. Известно, что вершин не больше 100, поэтому мы можем использовать матрицу смежности для представления графа. Очевидно, что граф неориентированный. При считывании отмечаем каждое ребро, при этом в матрице ставим не 1, а увеличиваем предыдущее значение количества ребер между планетами. Таким образом если в графе нет кратных ребер, то матрица будет содержать только 1 и 0, а в противном случае еще какие – либо натуральные числа.

Задача 14. Благополучие

Переводя эту задачу на язык графов, нам необходимо найти количество вершин, которые находятся в одной компоненте связности с данной. Для решения данной задачи можно было воспользоваться любым из известных методов обхода графа. Запустим этот обход из заданной вершины. Все вершины, которые находятся в одной компоненте с заданной будут «покрашены» в черный цвет. Посчитаем количество таких вершин, это и будет ответом на задачу.

Задача 15. Банкет

В этой задаче нам необходимо было проверить, является ли заданный граф двудольный и по возможности разбить на две доли. Произведём серию поисков в ширину. Т.е. будем запускать поиск в ширину из каждой непосещённой вершины. Ту вершину, из которой мы начинаем идти, мы помещаем в первую долю. В процессе поиска в ширину, если мы идём в какую-то новую вершину, то мы помещаем её в долю, отличную от доли текущей вершину. Если же мы пытаемся пройти по ребру в вершину, которая уже посещена, то мы проверяем, чтобы эта вершина и текущая вершина находились в разных долях. В противном случае граф двудольным не является.

По окончании работы алгоритма мы либо обнаружим, что граф не двудолен, либо найдём разбиение вершин графа на две доли.

Задача 16. Династии

Заметим, что если каждого жителя представить в виде вершины, а связь между ними в виде ребра, то получим граф. Он может быть как связным так и несвязным. Очевидно, что наша задача свелась к поиску количества компонент связности в графе. Количество компонент связности можно найти с помощью любого из описанных выше обходов. Для этого находим первую непосещенную вершину, запускаем из нее обход и так до тех пор, пока все вершины не будут посещены. Количество запусков обхода и есть ответом к задаче.

Задача 17. Императорское путешествие

Нам необходимо найти 2 самые удаленные планеты поиском в ширину и добавить к ответу 1(поиск в ширину вернет количество ребер на пути, а количество планет на 1 больше). Одно из возможных решений: запускаем поиск в ширину из каждой вершины, для нее получаем массив расстояний до каждой планеты. Обновляем наш ответ максимумом из этого массива + 1 и нашим текущим ответом.

Задача 18. Строительство дорог

Решение основано на системе непересекающихся множеств (DSU) планет. Две планеты принадлежат одному и тому же множеству тогда и только тогда, когда они достижимы друг из друга по уже построенным путям. Введем счетчик текущего количества множеств, изначально равный n. Изначально в каждом множестве находится только одна планета. При считывании пути (x, y) проверим, находятся ли планеты x и y в одном множестве. Если нет, то объединим содержащие их множества и уменьшим счетчик на 1. Как только счетчик станет равным 1 (то есть все планеты будут принадлежать одному множеству), выведем количество считанных путей.

Задача 19. Торговая сеть

В этой задаче необходимо было найти минимальное остовное дерево. Для его поиска можно было воспользоваться описанным выше алгоритмом Крускала либо алгоритмом Прима. Заметим, что ограничения задачи требуют оптимальной реализации за O (M log N).

Задача 20. Карта империи

Очевидно, что ответом на задачу будет результат работы алгоритма Флойда, описанного выше. Применим его к считанной матрице и выведем полученную матрицу расстояний.