Алгоритмы на графах. Семинар 11. 18 апреля 2019 г.

Подготовил: Горбунов Э.

Источники: [Кормен 1, §7.5, 22, 25, 26.2], [Кормен 3, §21, 23.2, 24, 25], [ДПВ, Глава 4, Глава 5, §5.1]

Ключевые слова: поиск кратчайших путей в графе, алгоритм Дейкстры, алгоритм Беллмана-Форда, алгоритм Флойда-Уоршолла, структура UNION-FIND, алгоритмы Краскала и Прима поиска минимального остовного дерева

Задача поиска кратчайших путей в графе

На прошлом семинаре мы большое внимание уделили алгоритму поиска в ширину (BFS — breadth-first search). Как мы уже заметили, данный алгоритм позволяет находить наименьшие по количеству рёбер пути из заданной вершины во все вершины, в которые из неё можно попасть. Иными словами, поиск в ширину решает задачу о поиске кратчайших путей из данной вершины во все остальные вершины для графов, у которых все рёбра равноценны (имеют одинаковый вес). Но если граф моделирует, например, карту автомобильных дорог (вершины соответствуют населённым пунктам, рёбра — дорогам), то для нахождения кратчайшего (в житейском смысле) пути между двумя пунктами имеет смысл различать рёбра между собой. Обычно в таких случаях вводят *весовую функцию* $w : E \to \mathbb{R}$, которая каждому ребру графа ставит в соответствие некоторое *вещественное* число. В случае примера с картой автомобильных дорог разумно выбрать весовую функцию как длину соответствующей дороги в некоторых единицах измерения длины — в таком случае весовая функция будет положительной. Однако мы допускаем, что она может быть и отрицательной для некоторых графов. Такая общая постановка рассматривается не случайно: граф может задавать состояния некоторой фирмы; при переходе от одного состояния к другому фирма, скажем, заключает сделку; от сделки она может нести убытки или, наоборот, получать прибыль. В такой постановке становится понятно, что если мы собираемся ввести весовую функцию, то она должна иметь возможность принимать значения обоих знаков.

Весом пути $p = (v_0, v_1, \ldots, v_k)$ называется следующая сумма:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Весом кратчайшего пути из вершины u в вершину v будем называть следующую величину

$$\delta(u, v) = \begin{cases} \min\{w(p) \mid u \overset{p}{\rightsquigarrow} v\}, & \text{если существует путь из } u \text{ в } v, \\ \infty & \text{иначе.} \end{cases}$$

Важно понимать, что если мы допускаем наличие рёбер отрицательного веса, то задача имеет смысл только в том случае, если нет циклов отрицательного веса (или их нет на любом пути между двумя интересующими нас вершинами). Действительно, по циклу отрицательного веса можно пройти сколько угодно раз, тем самым каждый раз уменьшая вес пути на некоторую положительную константу, то есть вес кратчайшего пути будет неопределён (в таком случае говорят, что он равен $(-\infty)$).

Кроме того, мы будем рассматривать задачи, в которых нам нужно найти какой-то кратчайший путь (кратчайших путей из одной вершины в другую может быть несколько, но для нас они все будут равноценны). Поэтому в рассматриваемых нами процедурах мы будем запоминать некоторого предшественника для каждой вершины (как это мы делали в алгоритмах поиска в ширину и в глубину). В течении работы рассматриваемые нами алгоритмы будут строить так называемые подграфы предшествования. Допустим, мы хотим найти кратчайшие пути из вершины s во все остальные вершины. *Подграф предшествования* $G_\pi = (V_\pi, E_\pi)$ определяется следующим образом: V_π — это те вершины, у которых предшественник отличен от NIL, и стартовая вершина, то есть

$$G_\pi = \{v \in V \mid \pi[v] \neq \text{NIL}\} \cup \{s\};$$

рёбра G_π — это рёбра из $\pi[v] \neq \text{NIL}$ в v:

$$E_\pi = \{(\pi[v], v) \in E \mid v \in V_\pi \setminus \{s\}\}.$$

Деревья кратчайших путей и релаксация

Во-первых, зафиксируем простое, но важное свойство: любая часть кратчайшего пути есть кратчайший путь. Формально это можно записать в виде следующей теоремы.

Теорема 1. Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$. Если $p = (v_1, v_2, \ldots, v_k)$ — кратчайший путь из v_1 в v_k и $1 \leqslant i \leqslant j \leqslant k$, то $p_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ — кратчайший путь из v_i в v_j.

Из него следуют ещё два полезных для нас факта.

Следствие 1. Пусть $G = (V, E)$ — взвешенный ориентированнный граф с весовой функцией $w : E \to \mathbb{R}$. Рассмотрим кратчайший путь p из s в v. Пусть $u \to v$ — последнее ребро в этом пути (p есть $s \overset{p'}{\rightsquigarrow} u \to v$). Тогда $\delta(s, v) = \delta(s, u) + w(u, v)$.

Следствие 2. Пусть $G = (V, E)$ — взвешенный ориентированнный граф с весовой функцией $w : E \to \mathbb{R}$; пусть $s \in V$. Тогда для всякого ребра $(u, v) \in E$ имеем $\delta(s, v) \leqslant \delta(s, u) + w(u, v)$.

Теперь обсудим технику *релаксации*. Идея состоит в том, чтобы для каждой вершины $v \in V$ хранить некоторое число $d[v]$, являющееся верхней оценкой веса кратчайшего пути из вершины s в вершину v (*оценка кратчайшего пути*). В начале работы алгоритмов поиска кратчайших путей мы будем вызывать функцию Initialize-Single-Source(G, s), которая для каждой вершины $v \in V$ делает два присваивания: $d[v] \leftarrow \infty, \pi[v] \leftarrow \text{NIL}$; затем для вершины s мы кладём $d[s] \leftarrow 0$. Релаксация ребра (u, v) состоит в следующем: если $d[v] > d[u] + w(u, v)$, то $d[v] \leftarrow d[u] + w(u, v)$, так как мы можем попасть в вершину v через вершину u. Эту незатейливую процедуру мы будем обозначать Relax(u, v, w). Отметим, что сразу после вызова Relax(u, v, w) будет выполнено неравенство $d[v] \leqslant d[u] + w(u, v)$.

Кроме того, если мы сначала вызовем процедуру Initialize-Single-Source(G, s), а затем в некотором порядке будем релаксировать рёбра, то всегда будет выполнено неравенство $d[u] \geqslant \delta(s, u)$ для всех $u \in V$. Если в какой-то момент для некоторой вершины $v \in V$ будет выполнено равенство $d[v] = \delta(s, v)$, то оно останется верным и при последующих релаксациях рёбер. Отсюда следует, что если вершина $v \in V$ недостижима из s, то при произвольной последовательности релаксации рёбер значение $d[v]$ будет оставаться бесконечным.

Теперь посмотрим, что происходит с графом предшествования G_π при произвольной последовательности релаксаций рёбер графа G. Будем рассматривать графы без циклов отрицательного веса, достижимых из начальной вершины. Оказывается, что в таком случае граф G_π будет всегда являться деревом с корнем в вершине s. Зафиксируем этот факт в виде следующей теоремы.

Теорема 2. Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией w и стартовой вершиной s, причём в графе G нет циклов отрицательного веса, достижимых из s. Тогда после операции Initialize-Single-Source(G, s), за которой следует произвольная последовательность релаксаций рёбер графа G, подграф предшественников G_π является деревом с корнем s.

Доказательство. В графе G_π после вызова Initialize-Single-Source(G, s) находится только вершина s, то есть в начальный момент этот подграф является деревом с корнем s. Покажем индукцией по количеству релаксаций рёбер, что это свойство сохраняется при произвольной последовательности релаксаций рёбер.

В графе G_π появляются новые вершины только в том случае, если происходит релаксация некоторого ребра (u, v), причём до этого $d[v]$ было бесконечным, а после релаксации стало конечным (это происходит только в том случае, если $d[u]$ на момент релаксации конечно). Тогда $\pi[v]$ становится равным u, то есть к дереву G_π добавляется лист v, то есть G_π остаётся деревом.

Проверим теперь ситуацию, когда происходит релаксация некоторого ребра (u, v), для которого и $d[u]$, и $d[v]$ являются конечными величинами. Это означает, что вершины u и v уже находятся в графе предшествования

G_π. В случае $d[v] > d[u] + w(u,v)$ при релаксации ребра (u,v) поддерево с корнем v графа G_π отрезается от $\pi[v]$ и присоединяется к вершине u, то есть $\pi[v] \leftarrow u$. Возникает резонный вопрос: а не появлось ли после такой операции в графе G_π цикла? Если цикл возник после релаксации ребра (u,v), то из рассуждений выше следует, что вершина u обязана быть потомком v. Покажем, что тогда этот цикл обязан иметь отрицательный вес, откуда и получим противоречие.

Итак, перед реклаксацией ребра (u,v) выполнялось неравенство $d[v] > d[u]+w(u,v)$, то есть $d[u]-d[v]+w(u,v) < 0$. Покажем, что путь в графе G_π от вершины v до вершины u не превосходит $d[u]-d[v]$ (отсюда будет следовать, что вместе с ребром (u,v) он образует цикл отрицательного веса). Достаточно это показать для одного ребра, то есть показать, что если в какой-то момент ребро (x,y) принадлежит графу G_π, то в этот момент

$$w(x,y) \leqslant d[y] - d[x].$$

Непосредственно после релаксации (x,y) это неравенство обращается в равенство. Далее могут уменьшаться значения $d[y]$ и $d[x]$. Если уменьшается $d[x]$, то неравенство остаётся верным. Если же уменьшаетс $d[y]$, то это означает, что произошла релаксация некоторого ребра (z,y), а значит, теперь $\pi[y] = z$ и выполняется неравенство $w(z,y) \leqslant d[y] - d[z]$, а ребра (x,y) в графе G_π больше нет. □

Из доказанной теоремы следует, что если в какой-то момент $\forall v \in V \hookrightarrow d[v] = \delta(s,v)$, то граф G_π является *деревом кратчайших путей.*

Алгоритм Дейкстры

Алгоритм Дейкстры позволяет найти кратчайшие пути из вершины s во все достижимые из неё вершины для взвешенного ориентированного графа $G = (V,E)$, в котором *веса всех рёбер неотрицательные.*

Идея алгоритма состоит в следующем: поддерживается множество S, состоящее из вершин, для которых мы уже нашли кратчайшие пути (т.е. $d[u] = \delta(s,u)$). Далее алгоритм добавляет к S вершину $v \in V \setminus S$ с наименьшим $d[v]$, а затем производит релаксацию всех рёбер, выходящих из v, после чего цикл повторяется. Вершины, не лежащие в S, хранятся в очереди с приоритетами Q (приоритеты определяются значениями d; чем меньше d, тем больше приоритет). Кроме того, считаем, что граф задан списком смежных вершин.

```
 1: procedure DIJKSTRA(G, w, s)
 2:     INITIALIZE-SINGLE-SOURCE(G, s)
 3:     S ← ∅
 4:     Q ← V[G]
 5:     while Q ≠ ∅ do
 6:         u ← EXTRACT-MIN(Q)
 7:         S ← S ∪ {u}
 8:         for (для) всех вершин v ∈ Adj[u] do
 9:             RELAX(u, v, w)
10:         end for
11:     end while
12: end procedure
```

В строке 6 происходит вызов процедуры EXTRACT-MIN(Q), которая извлекает из очереди элемент с наименьшим d. Далее мы обсудим детали реализации очереди с приоритетами, а пока что разберёмся, почему алгоритм Дейкстры корректно вычисляет кратчайшие расстояния.

Теорема 3. Пусть $G = (V,E)$ — взвешенный ориентированный граф с неотрицательной весовой функцией $w : E \to \mathbb{R}$ и исходной вершиной s. Тогда после применения алгоритма Дейкстры к этому графу для всех вершин $u \in V$ будут выполняться равенства $d[u] = \delta(s,u)$.

Доказательство. Покажем, что после любого числа итераций цикла **while** выполнены следующие свойства:

а) для вершин $v \in S$ значение $d[v]$ равно $\delta(s,v)$, причём существует кратчайший путь из s в v, целиком лежащий во множестве S, кроме последней вершины;

б) для вершин $v \in Q = V \setminus S$ значение $d[v]$ равно наименьшему весу пути из s в v среди тех путей, все вершины которых, кроме последней, лежат в S (если же таких путей нет, то $d[v] = \infty$).

После первой итерации цикла **while** в S лежит только вершина s и проведены релаксации всех рёбер, выходящих из s. Тогда свойства а) и б) будут выполнены. Проверим, что они не нарушатся и на следующих итерациях.

Пусть u — вершина из Q с наименьшим d. Если $d[u] = \infty$, то для всех вершин из Q значение d равно ∞, а значит, из S нельзя попасть во множество $V \setminus S$. Поэтому свойства а) и б) уже выполнены (по предположению индукции).

Если же $d[u]$ конечно, то существует кратчайший путь из s в u, целиком лежащий во множестве S, кроме последней вершины u. Действительно, если есть какой-то другой путь из s в u, не удовлетворяющий описанному свойству, то пусть первая вершина на этом пути, которая не лежит в S есть $y \neq u$. Но в силу выбора u мы имеем, что $d[y] \geqslant d[u]$. По индуктивному предположению а) вес этого пути не меньше $d[y]$ (веса рёбер неотрицательны). Но тогда и вес всего пути не меньше $d[u]$. Следовательно, условие а) останется верным после добавления u ко множеству S. Осталось проверить условие б). Но оно будет выполнено после релаксации рёбер, исходящих из u, так как все пути, которые мы добавили в рассмотрение, проходят через вершину u (остальные уже были рассмотрены на предыдущих шагах). Поэтому, проведя релаксацию рёбер, выходящих из u, значения d для всех вершин из $V \setminus S'$, где $S' = S \cup \{u\}$, будут равняться весам наименьших путей, которые целиком лежат в S', кроме последней вершины. □

Из Теоремы 2 следует, что алгоритм Дейкстры позволяет построить дерево кратчайших путей с корнем в вершине s.

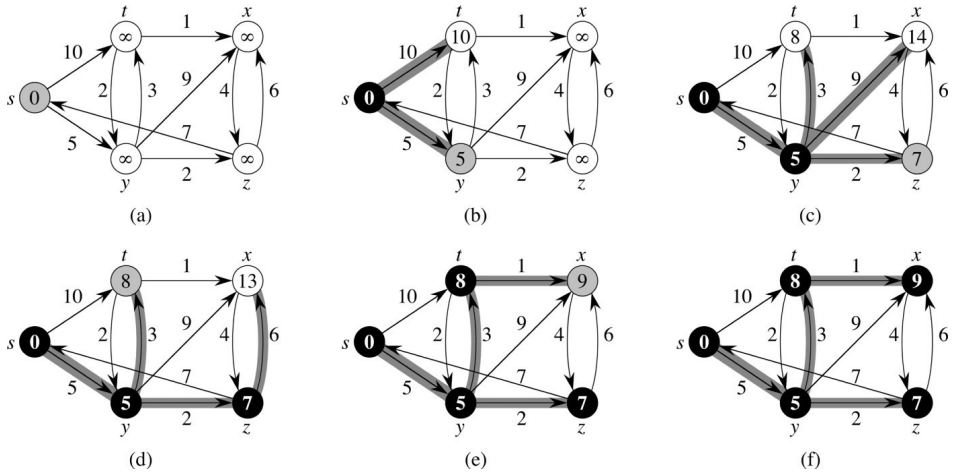

Рис. 1: Пример работы алгоритма Дейкстры.

Оценим время работы алгоритма Дейкстры. Для начала рассмотрим случай, когда очередь с приоритетами задаётся как обычный массив. Тогда стоимость операции Extract-Min есть $O(V)$ (эквивалентно задаче поиска наименьшего элемента в массиве длины V). Так нужно сделать для каждой вершины ровно V раз, а значит, суммарная стоимость всех удалений вершин из очереди равна $O(V^2)$. Кроме того, каждая вершина добавляется ко множеству S ровно один раз, и каждое ребро из $\mathrm{Adj}[v]$ обрабатывается только один раз. Стоимость каждой релаксации есть $O(1)$, поэтому общая стоимость алгоритма есть $O(V^2 + E) = O(V^2)$.

Однако если граф является разреженным, то выгоднее реализовать очередь с приоритетами с помощью двоичной кучи. Как это делать? Во-первых, мы будем поддерживать основное свойство кучи с обратным знаком, то есть будем поддерживать $A[i] \geqslant A[\mathrm{Parent}[i]]$ (сравниваются значения d). Процедура Extract-Min в таком случае реализуется простым способом: нужно вернуть корневую вершину двоичной кучи (там будет минимум), а затем поменять в массиве первый элемент с последним, уменьшить размер кучи на единицу и вызвать процедуру $\textsc{Heapify}(A, 1)$. Время работы процедуры Extract-Min составляет $O(\log V)$. Стоимость построения кучи (строка 4) составляет $O(V)$. Присваивание $d[v] \leftarrow d[u] + w(u, v)$ можно так же реализовать за $O(\log V)$

(уменьшаем ключ у вершины v до значения $d[u] + w(u, v)$, а затем «поднимаем» её на нужный уровень вверх: если значение ключа у родителя больше, чем у данной вершины, то меняем их местами, и так далее). Всего таких присваиваний будет произведено не более E штук, так что общая стоимость алгоритма Дейкстры, основанного на очередях с приоритетами, реализованными при помощи двоичных куч, составляет $O((V+E)\log V)$. Если граф связен, то $E \geqslant V - 1$ и предыдущая оценка записывается в виде $O(E \log V)$.

Алгоритм Беллмана-Форда

Алгоритм Беллмана-Форда решает задачу о поиске кратчайших путях из одной вершины для случая, когда веса рёбер могут быть отрицательными, но в графе нет циклов отрицательного веса, достижимых из s. В отличие от алгоритма Дейкстры данный алгоритм умеет решать задачу и для графов с отрицательными весами рёбер. Более того, он может определить, есть ли в графе цикл отрицательного веса (если нет, то алгоритм выдаст TRUE, а иначе — FALSE).

```
 1: procedure BELLMAN-FORD(G, w, s)
 2:     INITIALIZE-SINGLE-SOURCE(G, s)
 3:     for i ← 1 to |V(G)| − 1 do
 4:         for (для) каждого ребра (u, v) ∈ E(G) do
 5:             RELAX(u,v,w)
 6:         end for
 7:     end for
 8:     for (для) каждого ребра (u, v) ∈ E(G) do
 9:         if d[v] > d[u] + w(u, v) then
10:             return FALSE
11:         end if
12:     end for
13:     return TRUE
14: end procedure
```

Иными словами, $V - 1$ раз производится релаксация всех рёбер графа G. Как мы увидим далее, после такой операции для всех $v \in V$ должно выполняться равенство $d[v] = \delta(s, v)$, если в графе нет циклов отрицательной длины. Цикл в строках 8-12 проверяет граф G на отсутствие циклов отрицательной длины (см. Следствие 2).

Время работы алгоритма Беллмана-Форда есть $O(VE)$, поскольку в строках 3-7 выполняется $O(VE)$ релаксаций рёбер, время работы релаксации одного ребра равняется $O(1)$, строка 1 выполняется за время $O(V)$, а цикл в строках 8-12 — за время $O(E)$.

Теорема 4. Пусть $G = (V, E)$ — взвешенный ориентированный граф с весовой функцией $w : E \to \mathbb{R}$ и исходной вершиной s, не содержащий циклов отрицательного веса, достижимых из s. Тогда после применения алгоритма Беллмана-Форда к этому графу для всех вершин $u \in V$ будут выполняться равенства $d[u] = \delta(s, u)$.

Доказательство. Рассмотрим произвольный кратчайший путь $p = (s = v_0, v_1, \ldots, v_k = v)$. Так как в графе нет циклов отрицательной длины, то можно считать, что в пути p нет циклов (цикл не уменьшает длину пути). Поэтому можно считать, что $k \leqslant V - 1$. Покажем индукцией по i, что после i-й итерации цикла в строках 3-7 будет выполнено равенство $d[v_i] = \delta(s, v_i)$ (если это доказать, то получим, что после $V - 1$ итерации в $d[v_k] = d[v]$ будет записано $\delta(s, v)$, а в силу произвольности выбора v это будет выполнено и для всех вершин в графе G).

При $i = 0$ утверждение верно ($d[s] = \delta(s, s) = 0$). Пусть при $i > 0$ после $(i - 1)$-й итерации выполнялось равенство $d[v_{i-1}] = \delta(s, v_{i-1})$. Тогда из Следствия 1 получаем, что после i-й итерации выполняется $d[v_i] = \delta(s, v_i)$, поскольку на i-й итерации произойдёт релаксация ребра (v_{i-1}, v_i). □

Из Теоремы 2 следует, что алгоритм Беллмана-Форда позволяет построить дерево кратчайших путей с корнем в вершине s.

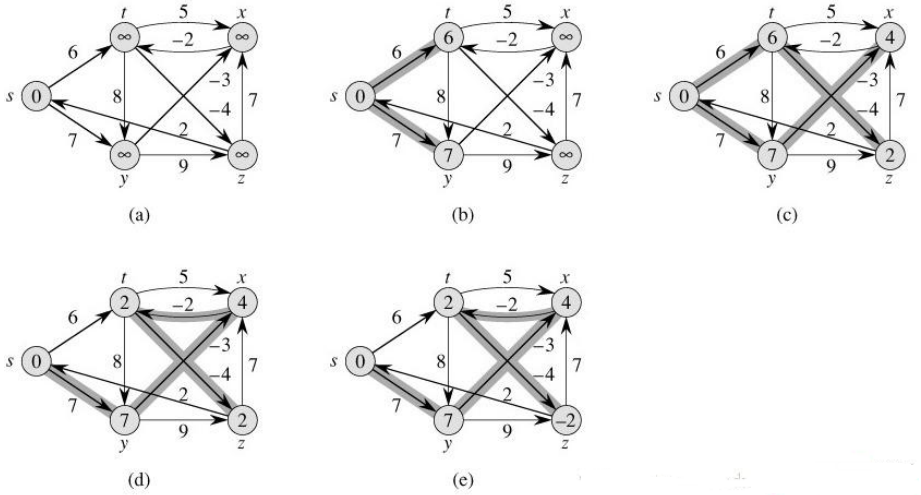

Рис. 2: Пример работы алгоритма Беллмана-Форда.

Алгоритм Флойда-Уоршолла

Алгоритм Флойда-Уоршолла решает задачу поиска кратчайших путей между любыми двумя парами вершин ориентированного взвешенного графа, у которого нет циклов отрицательного веса. Мы покажем, что этот метод работает за время $\Theta(V^3)$, то есть для плотных графов (когда $V^2 = O(E)$) алгоритмы Флойда-Уоршолла и Беллмана-Форда имеют одинаковую сложность $O(V^3)$.

Перейдём к описанию алгоритма. *Промежуточной* вершиной простого пути $p = (v_1, v_2, \ldots, v_l)$ буде называть любую из вершин $v_2, v_3, \ldots, v_{l-1}$ (напомню, что раз мы работаем с графами, в которых нет циклов отрицательного веса, то можем считать, что кратчайший путь между любыми двумя вершинами не содержит циклов, а значит, кратчайший путь состоит не более чем из $V - 1$ ребра).

Будем считать, что вершинами графа G являются числа $1, 2, 3, \ldots, n$. Рассмотрим произвольное $k \leqslant n$. Для данной пары вершин $i, j \in V$ рассмотрим все пути из i в j, у которых все промежуточные вершины лежат во множестве $\{1, 2, \ldots, k\}$. Пусть p — путь минимального веса среди всех таких путей (он будет простым, как мы уже обсудили). Есть две возможности. Если k не является промежуточной в пути p, то все промежуточные вершины принадлежат $\{1, 2, \ldots, k-1\}$, то есть p будет и кратчайшим путём между i и j среди путей, у которых промежуточные вершины лежат во множестве $\{1, 2, \ldots, k-1\}$. Если же k является промежуточной вершиной в пути p, то она разбивает путь p на две части p_1 и p_2: $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ (вершина k встречается лишь однажды, ибо путь p является простым). Но тогда путь p_1 является кратчайшим путём из i в k среди путей с промежуточными вершинами из $\{1, 2, \ldots, k-1\}$, а путь p_2 — кратчайшим из k в j с промежуточными вершинами из множества $\{1, 2, \ldots, k-1\}$.

Зафиксируем эти рассуждения в виде рекуррентной формулы. Пусть $d_{ij}^{(k)}$ — вес кратчайшего пути из i в j с промежуточными вершинами из $\{1, 2, \ldots, k\}$. Тогда из написанного выше следует, что

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & \text{если } k = 0, \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}, & \text{если } k \geqslant 1. \end{cases}$$

Матрица $D^{(n)} = (d_{ij}^{(n)})$ содержит искомое решение, поскольку разрешаются любые промежуточные вершины.

Кроме того, нам хотелось бы уметь находить не только веса кратчайших путей, но и сами пути. Поэтому нужен способ вычисления предшественников. К счастью, это тоже можно задать рекуррентной формулой. Пусть $\pi_{ij}^{(k)}$ — это вершина, предшествующая вершине j на кратчайшем пути из i в j с промежуточными вершинами из множества $\{1, 2, \ldots, k\}$. Тогда

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL}, & \text{если } i = j \text{ или } w_{ij} = \infty, \\ i, & \text{если } i \neq j \text{ и } w_{ij} < \infty, \end{cases}$$

и для $k \geqslant 1$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)}, & \text{если } d_{ij}^{(k-1)} \leqslant d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)}, & \text{если } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Напишем процедуру, которая по матрице весов рёбер W вычисляет матрицу кратчайших расстояний $D^{(n)}$.

```
1:  procedure FLOYD-WARSHALL(W)
2:      n ← rows[W]
3:      Initialize Π^(0)
4:      D^(0) ← W
5:      for k ← 1 to n do
6:          for i ← 1 to n do
7:              for j ← 1 to n do
8:                  if d_ij^(k-1) ⩽ d_ik^(k-1) + d_kj^(k-1) then
9:                      d_ij^(k) ← d_ij^(k-1)
10:                     π_ij^(k) ← π_ij^(k-1)
11:                 else
12:                     d_ij^(k) ← d_ik^(k-1) + d_kj^(k-1)
13:                     π_ij^(k) ← π_kj^(k-1)
14:                 end if
15:             end for
16:         end for
17:     end for
18:     return D^(n)
19: end procedure
```

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Рис. 3: Пример работы алгоритма Флойда-Уоршолла.

Структура данных UNION-FIND

В этом разделе мы рассмотрим структуру данных под названием *система непересекающихся множеств* (в нашем курсе мы её ещё будем называть структурой данных UNION-FIND). Данная структура данных представляет из себя набор непересекающихся непустых множеств, в каждом из которых зафиксирован один представитель. При этом должны поддерживаться следующие 3 операции.

1. Make-Set(x). Данная процедура создаёт из элемента x множество $\{x\}$ и возвращает указатель на представителя этого множества (то есть на x).
2. Find-Set(x). Процедура по элементу x определяет, в каком множестве этот элемент лежит, то есть возвращает указатель на представителя множества, в котором лежит x.
3. Union(x, y). Эта процедура применима, если элементы x и y содержатся в различных множествах. По заданным x и y процедура объединяет множества, в которых лежат данные элементы, и возвращает указатель на представителя объединения.

Системы непересекающихся множеств могут применяться, например, для поиска связных компонент графа. Причём такой подход разумно применять, если в графе появляются новые рёбра со временем (в таком случае классический подход с использованием поиска в глубину привёл бы к повторному его запуску; при работе с непересекающимися множествами это делается разумнее).

Перейдём теперь к вопросу о реализации структуры $UNION - FIND$.

Реализация при помощи списков

Самый простой способ реализации непересекающихся множеств хранит каждое множество в виде списка. Первый элемент списка является представителем множества, а каждый элемент списка имеет указатель на последующий элемент и на первый элемент. Кроме того, будем хранить для каждого списка указатели на его первый и последний элементы. При такой реализации процедуры Make-Set и Find-Set требуют времени $O(1)$: Make-Set создаёт список из одного элемента, а Find-Set возвращает указатель на первый элемент списка.

Однако при реализации объединения в таком подходе могут возникнуть неприятности. Предположим, что объединяем мы два списка x и y следующим образом: один из списков, скажем, x, подсоединяем к концу другого, а затем изменяем указатели, отвечающие за начало списка, в списке x так, чтобы они указывали на начало списка y. Время работы операции объединения при таком подходе будет равно $O(|x|)$, где $|x|$ — длина списка x. Легко привести пример, когда время выполнения m операций Union квадратично по числу операций. Пусть даны n элементов $x_1, x_2, \ldots, x_n$. Выполним операции Make-Set(x_i) для всех $i = 1, 2, \ldots, n$, а затем $n-1$ операций Union(x_1, x_2), Union(x_2, x_3), $\ldots$, Union(x_{n-1}, x_n) и пусть каждый раз список, соответствующий первой вершине, подсоединяется к концу списка, соответствующего второй вершине. В силу построения стоимость операции Union(x_i, x_{i+1}) равна $O(i)$, поэтому суммарная стоимость выполнения всех операций Union составляет

$$O\left(\sum_{i=1}^{n-1} i\right) = O(n^2).$$

Исправить ситуацию может *весовая эвристика*: будем дополнительно хранить длину списка и при операции Union более короткий список добавлять в конец более длинного (в случае совпадения длин, будем добавлять первый список в конец второго). Нетрудно показать, что в таком случае стоимость последовательности из m операций Make-Set, Find-Set и Union, среди которых n операций Make-Set, есть $O(m + n \log n)$. Действительно, стоимость каждой из операций Make-Set и Find-Set, а также стоимость сравнения размеров списков, соединения списков и обновления записи о размере множества, есть $O(1)$, поэтому суммарная стоимость указанных операций равняется $O(m)$. Оценим стоимость обновления указателей при объединении списков. Каждый элемент x меняет указатель на начальный элемент не более $\lceil \log n \rceil$ раз, поскольку смена указателя на начало происходит только при объединении, причём объединении со множеством, размер которого хотя бы такой же (то есть полученное множество имеет размер хотя бы вдвое больший, чем у множества, содержащего x до объединения). Всего элементов n, поэтому стоимость обновления указателей есть $O(n \log n)$, что и требовалось доказать.

Реализация при помощи деревьев

Рассмотрим теперь более эффективную реализацию структуры UNION-FIND. Представим каждое множество в виде корневого дерева, вершины которого являются элементами, а корень является представителем, причём каждая вершина указывает на своего родителя, а корень указывает сам на себя. Наивные реализации операций выглядят так: MAKE-SET создаёт дерево с единственной вершиной; FIND-SET(x) состоит в том, что мы идём от вершины x по стрелкам, указывающим на родителя, пока не дойдём до корня; UNION состоит в том, что мы заставляем корень одного из деревьев указывать не на себя, а на корень другого дерева. Однако при таком подходе больших преимуществ по сравнению со списочной реализацией не видно: например, в результате $n-1$ операции UNION может получится дерево, являющееся цепочкой из n вершин.

Существуют две эвристики, которые помогают добиться почти линейной скорости выполнения m операций с непересекающимися множествами. Первая эвристика — *объединение по рангу*. Она напоминает весовую эвристику в списочной реализации: мы приписываем каждой вершине некоторое число, называемое *рангом*, и два поддерева объединяем так, чтобы дерево, у которого ранг корня меньше, было подсоединено к корню второго дерева, причём ранги в таком случае мы менять не будем. Если же объединяются два дерева, ранги корней которых равны, то мы подсоединим первое дерево к корню второго и увеличим ранг корня второго дерева на единицу. Кроме того, будем считать, что ранг корня дерева, состоящего из одной вершины, равен нулю. Если больше ничего не предполагать, то введённое определение ранга соответствует высоте поддерева с корнем в данной вершине.

Однако это верно до тех пор, пока мы не станем использовать вторую эвристику — *сжатие путей*. Она заключается в следующем: после того, как путь от вершины к корню пройден, дерево перестраивается: в каждой из вершин пути указатель устанавливается непосредственно на корень. Ранги при это остаются прежними. Тогда в следующий раз при вызове процедуры FIND-SET из любой вершины на обработанном пути мы получим ответ за время $O(1)$, если больше данное дерево не перестраивалось. При использовании сжатия путей ранг теряет свою интерпретацию, как высота поддерева с корнем в данной вершине. Однако, он остаётся верхней границей на высоту поддерева в данной вершине всегда.

Зафиксируем реализации процедур для работы с системой непересекающихся множеств, заданных в виде деревьев с двумя описанными эвристиками. Ниже $p[x]$ обозначает родителя вершины x (указатель на родителя), а $\text{rank}[x]$ — её ранг.

```
 1: procedure MAKE-SET(x)
 2:     p[x] ← x
 3:     rank[x] ← 0
 4: end procedure
 5: procedure UNION(x, y)
 6:     LINK(FIND-SET(x), FIND-SET(y))
 7: end procedure
 8: procedure LINK(x, y)
 9:     if rank[x] > rank[y] then
10:         p[y] ← p[x]
11:     else
12:         p[x] ← y
13:         if rank[x] = rank[y] then
14:             rank[y] ← rank[y] + 1
15:         end if
16:     end if
17: end procedure
18: procedure FIND-SET(x)
19:     if x ≠ p[x] then
20:         p[y] ← FIND-SET(p[x])
21:     end if
22:     return p[x]
23: end procedure
```

Покажем, что при использовании данных двух эвристик время выполнения m операций MAKE-SET, FIND-SET

и Union, среди которых n операций Make-Set, есть $O(m \log^* n)$. Величину $\log^* n$ называют *итерированным логарифмом*. Она определяется следующим образом: $\log^* 1 = 0$ и $\log^* n = \min\{i \mid \underbrace{\log\log\ldots\log}_{i \text{ раз}} \leqslant 1\}$. Отметим, что итерированный логарифм — это очень медленно растущая функция: $\log^* 2^{65536} = 5$, то есть во всех мыслимых приложениях можно считать, что $\log^* n \leqslant 5$. Таким образом, оценка $O(m \log^* n)$ практически линейна по m.

Для начала перечислим нужные нам свойства рангов.

1. Для всякой вершины x, кроме корня, $\text{rank}[x] < \text{rank}[p[x]]$. Это следует из правил построения деревьев. Кроме того, ранг перестаёт меняться, когда вершина перестаёт быть корнем дерева.

2. Пусть $\text{size}[x]$ — количество вершин в поддереве, корнем которого является вершина x. Тогда $\text{size}[x] \geqslant 2^{\text{rank}[x]}$. Действительно, в начальный момент (после операции Make-Set) для каждого дерева данное неравенство выполнено. Операция Find-Set не меняет ни рангов вершин, ни размеров деревьев, поэтому после её применения неравенство остаётся верным (если оно было верным). Осталось рассмотреть операцию $\text{Link}(x, y)$. Не умаляя общности, считаем, что $\text{rank}[x] \leqslant \text{rank}[y]$. Тогда размер или ранг может измениться только у вершины y. Если $\text{rank}[x] < \text{rank}[y]$, то ранг вершины y не меняется, а размер увеличивается, то есть неравенство $\text{size}[y] \geqslant 2^{\text{rank}[y]}$ остаётся верным. Если же $\text{rank}[y] = \text{rank}[x]$, то новый «размер вершины» y становится равным $\text{size}'[y] = \text{size}[x] + \text{size}[y]$, а ранг вершины y становится равным $\text{rank}'[y] = \text{rank}[y] + 1$. Отсюда

$$\text{size}'[y] = \text{size}[x] + \text{size}[y] \geqslant 2^{\text{rank}[x]} + 2^{\text{rank}[y]} = 2^{\text{rank}[y]+1} = 2^{\text{rank}'[y]}.$$

3. Число вершин ранга r не превосходит $\frac{n}{2^r}$. Это утверждение напрямую следует из предыдущего свойства. Отсюда следует, что ранг любой вершины не превосходит $\lfloor \log n \rfloor$.

Нам будет удобно считать, что m — это число операций Make-Set, Find-Set и Link, среди которых n операций Make-Set. Это не меняет временной сложности, потому что одна операция Union соответствует двум операциям Find-Set и одной операции Link.

Стоимость каждой операции Make-Set и Link есть $O(1)$, а их суммарная стоимость $O(m)$. Осталось оценить суммарную стоимость выполнения операций Find-Set.

При выполнении операции Find-Set мы двигаемся вверх по одному из деревьев и заканчиваем поиск в корне. При этом при переходе от одной вершины к другой ранг увеличивается. После сжатия путей ранги родителей, встреченных на пути, увеличатся (если не считать корневую вершину) — это простое, но важное наблюдение.

Отдельно оценим количество шагов, когда при переходе от вершины к родителю ранг рос «быстро» и когда он рос «медленно» (посещений корневых вершин суммарно $O(m)$, поэтому мы их учитывать не будем). Рассмотрим функцию $\beta(k) = \lceil (1,9)^k \rceil$ (такой выбор функции будет пояснён позднее; отметим только, что $\beta(k) > k$ для всех $k > 0$ и $\beta(k)$ возрастает). Пусть $\text{rank}[u] = k$ и $v = p[u]$. Будем считать, что ранг растёт быстро при переходе от u к v, если $\text{rank}[v] \geqslant \beta(k)$.

Оценим число шагов, когда ранг рос не быстро. Для данной вершины u ранга k её ранг может меняться не быстро, если ранг её родителя лежит в промежутке от $k+1$ до $\beta(k)-1$. Значит, число шагов, когда ранг рос не быстро для данной вершины, не превосходит $\beta(k) - 1 - k - 1 < \beta(k)$. Вершин ранга k не больше $\frac{n}{2^k}$, поэтому общее число шагов, когда ранг растёт не быстро, не превосходит

$$\sum_{k=0}^{\lfloor \log n \rfloor} n \cdot \frac{\beta(k)}{2^k} \leqslant n \cdot \sum_{k=0}^{\infty} \left(\frac{1,9}{2}\right)^k + n \cdot \sum_{k=0}^{\infty} \frac{1}{2^k} = O(n),$$

поскольку ряды $\sum\limits_{k=0}^{\infty} \left(\frac{1,9}{2}\right)^k$ и $\sum\limits_{k=0}^{\infty} \frac{1}{2^k}$ сходится.

Теперь оценим число шагов, когда ранг растёт сильно, то есть когда $\text{rank}[v] \geqslant \beta(\text{rank}[u])$. Если бы $\beta(k) = 2^k$, то в силу того, что ранг не превосходит $\log n$, мы бы получили, что максимальное число таких переходов на каждом пути примерно равно $\log^* n$. Если $\beta(k) = \lceil (1,9)^k \rceil$, то максимальное число таких переходов возрастёт

не больше, чем вдвое, поскольку $\beta(\beta(k)) \geqslant 2^k$, то есть в любом случае переходов в худшем случае будет $O(\log^* n)$, а значит общее число шагов такого рода для всех операций будет $O(m \log^* n)$.

Рассмотренная структура данных весьма полезна для построения эффективной реализации *алгоритма Краскала* поиска *минимального остовного дерева*, работающей за время $O(E \log E) = O(E \log V)$, поскольку $E \leqslant V^2$. Другой алгоритм поиска минимального остова — *алгоритм Прима* — работает за время $O(E \log V)$, если использовать очередь с приоритетами, реализованную через двоичную кучу. Об обоих алгоритмах можно прочитать в [Кормен 1, §24] и [Кормен 3, §23].