Основные алгоритмы, комментарии по восьмому листку (Графы I. Поиск в ширину и кратчайшие пути)

Шибаев Иннокентий

April 1, 2021

1 По поводу теории

1.1 Поиск кратчайших путей в графах

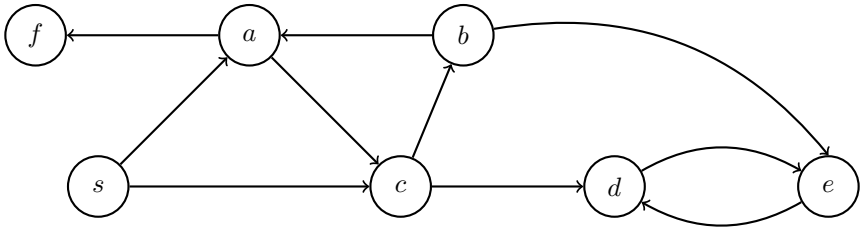
Для графа $G = (V, E)$ эта задача обычно ставится в виде «Найти кратчайший путь из вершины s в вершину t», при этом E это множество ребер с весами (троек вида (u, v, w)). Графы мы считаем ориентированными (неориентированные трафы мы сводим к ориентированным раздваивая ребра).

На прошлом семинаре мы разобрали три алгоритма решающих задачу выше.

1.1.1 Поиск в ширину (BFS)

Этот алгоритм ищет кратчайшие пути от вершины v до всех остальных при условии, что веса на ребрах равны 1. Случай с $w \in \mathbb{N}$ можно свести к этому вводя дополнительные вершины (хотя эффективность такого подхода не очень высока, сложность в этом случае мы обсудим в конце).

Рассмотрим работу алгоритма на примере графа с семинара



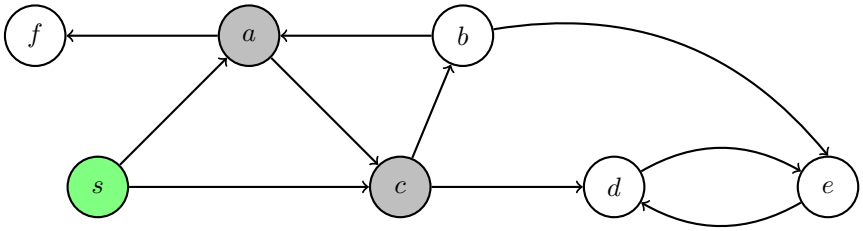
На каждом шаге алгоритм берет очередную вершину u из очереди, и добавляет в очередь вершины в которые ведут ребра из вершины u (пропуская уже посещенные, для этого помечаем вершины которые уже были в очереди серым, а вершины которые были извлечены из очереди и обработаны). Изначально в очереди находится вершина для которой мы ищем расстояния до остальных (в нашем случае это вершина s).

Для того чтобы получать расстояния надо поддерживать массив $d[u]$ расстояний от вершины s до остальных вершин, и при обработке вершины u и ребра $u \to v$ обновлять $d[v] = d[u] + 1$ (опять же, если вершина u еще не была помечена).

Чтобы восстанавливать пути достаточно хранить $p[u]$ – массив вершин-предков (который можно обновлять в тот же момент когда и $d[v]$). Тогда путь от вершины s до вершины t восстанавливается как

$$t \leftarrow p[t] \leftarrow p[p[t]] \leftarrow \ldots \leftarrow s.$$

Изначально в очереди есть только вершина s. После ее обработки получаем следующую картину:

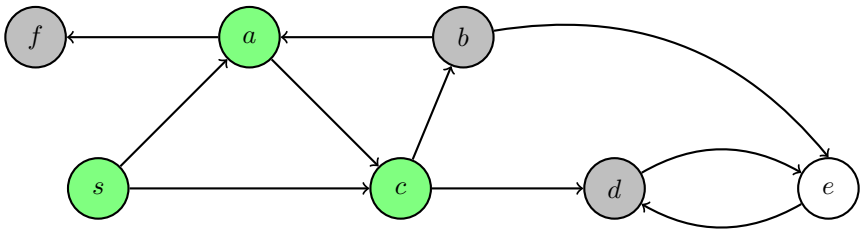


u	s	a	b	c	d	e	f	← Вершина
d[**u**]	0	1		1				← Расстояние от s
p[**u**]	–	s		s				← Родительская вершина

1	2
a	c

Очередь вершин

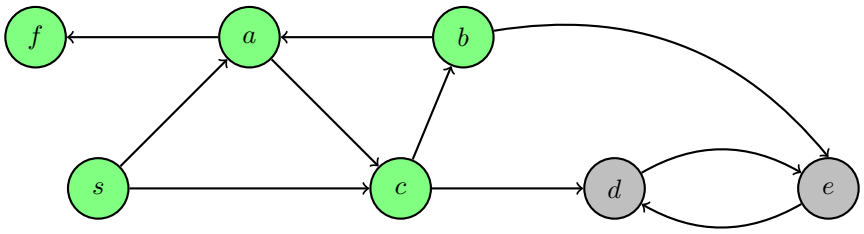
затем, после обработки вершины a (для которой есть только одна непомеченная вершина f) и вершины c мы получаем



u	s	a	b	c	d	e	f
d[**u**]	0	1	2	1	2		2
p[**u**]	–	s	c	s	c		a

1	2	3
f	b	d

и после обработки f (которая ничего не меняет) и b (во время которой добавляется e) мы приходим к



u	s	a	b	c	d	e	f
d[**u**]	0	1	2	1	2	3	2
p[**u**]	–	s	c	s	c	b	a

1	2
d	e

при этом больше непомеченных вершин не осталось, таким образом больше изменений не произойдет.

Работу алгоритма можно описать по-разному, к примеру можно представить что во входную вершину мы наливаем воду – тогда вода в начале достигнет вершин удаленных

на расстояние 1 (a, c), затем вершин на расстоянии 2 (f, b, d) и, наконец, дойдет до вершины e.

В процессе работы мы посещаем каждую (достижимую из s) вершину графа – не более $|V|$ вершин, и просматриваем все ребра исходящие из каждой такой вершины, таким образом алгоритм работает за $O(|V| + |E|)$.

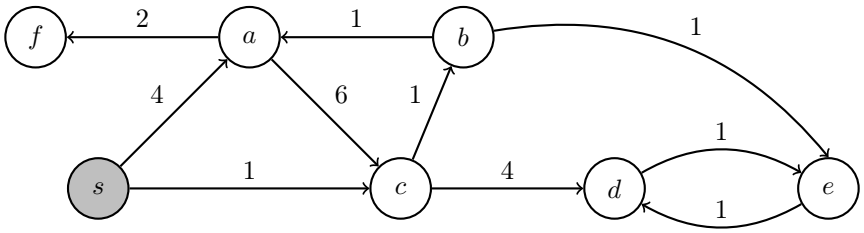
Для случая $w \in \mathbb{N}$ мы можем представить каждое ребро как цепь вершин соединенных ребрами веса 1. Обозначив $W = \max_i w_i$ получаем что число вершин вырастет максимум до $W|V|$, а число ребер увеличится на $(W - 1)|V|$, таким образом сложность будет $O(W|V| + |E|)$.

1.1.2 Алгоритм Дейкстры

Этот алгоритм решает ту же задачу поиска кратчайших расстояний от одной вершины до остальных для случая неотрицательных весов.

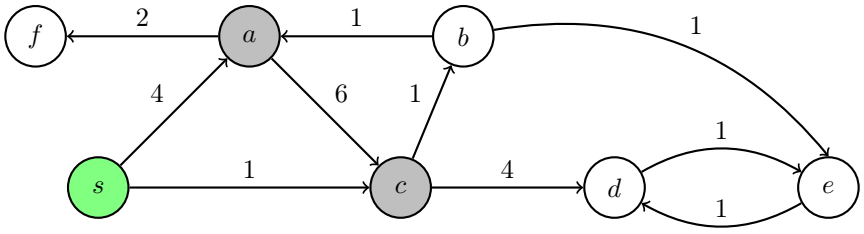
Алгоритм похож на предыдущий, только теперь мы на каждом шаге для обновления выбираем еще не обработанную вершину u (серую) с наименьшим до нее расстоянием от вершины s, и обновления происходят при условии $(u, v, w) \in E$ и $d[v] > d[u] + w$. Если обновление произошло то надо поменять и родителя: $p[v] = u$.

Опять же посмотрим на примере того же графа (с добавленными весами на ребрах):



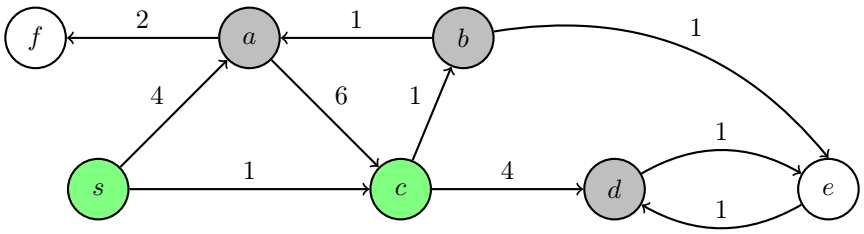
u	s	a	b	c	d	e	f	← Вершина
d[**u**]	0	∞	∞	∞	∞	∞	∞	← Расстояние от s
p[**u**]	–							← Родительская вершина

На первом шаге расстояния до всех вершин кроме вершины s равны ∞, поэтому мы берем ее, обновляем расстояния до вершин a и c и помечаем вершину s как обработанную (зеленым)



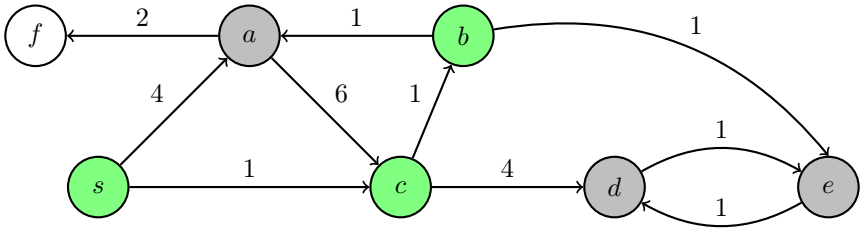
u	s	a	b	c	d	e	f
d[**u**]	0	4	∞	1	∞	∞	∞
p[**u**]	–	s		s			

Теперь среди необработанных вершин наименьшее расстояние до вершины c, поэтому мы обрабатываем ее, обновляем расстояния до b и d и получаем



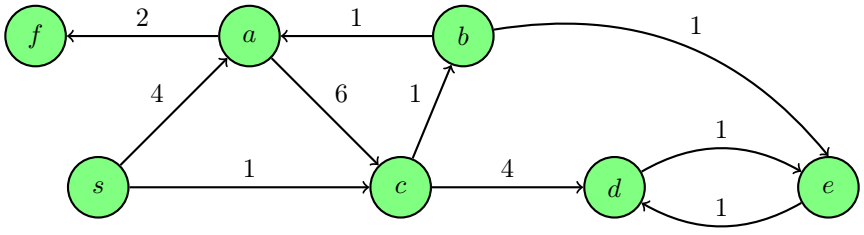
u	s	a	b	c	d	e	f
d[**u**]	0	4	2	1	5	∞	∞
p[**u**]	–	s	c	s	c		

Наконец, при обработке вершины b мы первый раз сталкиваемся с нетривиальным случаем – при обновлении вершины a мы получаем $d[b] + w = 2 + 1 < d[a] = 4$, поэтому мы обновляем расстояние до вершины a, а также ее предка: $p[a] = b$. Помимо этого происходит обновление e:



u	s	a	b	c	d	e	f
d[**u**]	0	3	2	1	5	3	∞
p[**u**]	–	b	c	s	c	b	

Продолжая этот процесс мы, наконец, приходим к конечному состоянию



u	s	a	b	c	d	e	f
d[**u**]	0	3	2	1	4	3	5
p[**u**]	–	b	c	s	e	b	a

Красным здесь все так же отмечены изменения. Помимо очевидного обновления f произошло так же обновление расстояния до d – добраться до него из c через последовательность $c \to b \to e \to d$ быстрее $(1 + 1 + 1 = 3)$ чем напрямую (4).

Обсудим сложность этого алгоритма. Опять же, мы последовательно обрабатываем вершины. Каждый раз мы *ищем вершину с минимальным расстоянием до нее*, после чего начинаем *обрабатывать ребра исходящие из нее*. Вопрос в том как организовать эти процессы.

1. Мы можем сделать все довольно напрямую – будем для каждой вершины хранить флаг того обработана ли она, после чего на каждом шаге проходить по списку всех вершин и искать ту до которой расстояние минимально за $O(|V|)$.

   При этом для каждого ребра обновление происходит за $O(1)$ – мы смотрим на ребро (u, v, w) и обновляем $d[v]$ если $d[v] > d[u] + w$.

   Таким образом получаем $O(|V|^2 + |E|)$.

2. Мы можем поступить по-другому – поддерживать какую-нибудь структуру в которой можно быстро искать элемент с минимальным значением, и при этом быстро искать элемент по ключу (чтобы быстро обновлять расстояния до вершин). Такая структура нам известна, это очередь с приоритетами, и она позволяет делать эти операции за $O(\log n)$.

   В нашем случае в структуре лежат вершины (и расстояния до них), таким образом операции будут производится за $O(\log |V|)$. В результате получаем сложность алгоритма $O(|V| \log |V| + |E| \log |V|)$. В случае разреженных графов ($|E| \sim |V|$) это сильно лучше прошлого варианта. Наоборот, если граф плотный ($|E| \sim |V|^2$) предыдущий вариант даст лучшую асимптотику.

Еще одно замечание здесь связано с BFS. По сути своей в нем делается ровно то же самое – каждый раз мы обрабатываем вершину с минимальным до нее расстоянием. Просто в виду того что веса всех ребер равны простая очередь становится достаточной для поиска минимума, и мы получаем что обе операции (поиск минимума и обновление) можно делать за $O(1)$, что и приводит к $O(|V| + |E|)$.

1.1.3 Алгоритм Форда-Беллмана

В отличие от предыдущего, этот алгоритм работает и в случае наличия ребер отрицательного веса (однако наличие цикла отрицательного веса, понятно, недопустимо – мы всегда можем «накрутить» на нем сколь угодно большое отрицательное число). Расплачиваемся за это мы сложностью $O(|V||E|)$.

Сам алгоритм очень простой – $|V| - 1$ раз мы проходим по всем ребрам графа и пытаемся обновить значения, т.е. для ребра (u, v, w) сделать $d[v] = \min\{d[v], d[u] + w\}$. Пути, опять же, можно восстанавливать если хранить массив предков вершин.

Корректность его основывается на следующем утверждении:

Лемма 1.1. На i-м шаге работы, алгоритм Форда-Беллмана находит минимальные расстояния от вершины s до тех вершин, до которых есть оптимальный путь состоящий из не более чем i ребер.

Схема доказательства. Рассматриваем оптимальный путь из вершины s до вершины t, он имеет вид $s, u_2, \ldots, u_{i-1}, t$, при этом для каждой из вершин в этом пути префикс тоже является оптимальным (иначе его можно было бы улучшить). Далее по индукции, для исходной вершины условие выполнено, пусть выполнено для всех путей длины $i-1$ тогда когда мы просмотрим ребро (u_{i-1}, t) мы делаем обновление. □

Одно из свойств алгоритма - с помощью него можно детектировать наличие отрицательного цикла – достаточно попытаться обновить расстояния еще раз – изменение произойдет если и только если есть цикл отрицательного веса.

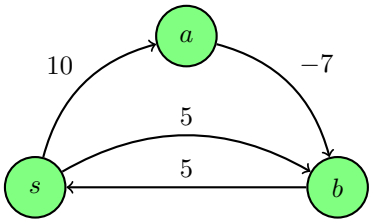
При этом можно так же реализовать и поиск этого цикла – достаточно найти вершину для которой произошло изменение, и перейти по массиву предков – в какой-то момент мы войдем в цикл (который будет все в том же массиве предков), и дальше его можно будет вывести.

2 По поводу задач

Задача 2.1 (Задача 6 из листка 8)**.** Приведите пример взвешенного ориентированного графа, на котором алгоритм Дейкстры находит кратчайшие пути неправильно.

Решение. Мы знаем что этот алгоритм корректно работает на графах с положительными весами ребер. Поэтому надо рассмотреть какой-нибудь граф с отрицательными весами. При этом рассматривать графы с отрицательными циклами – перебор, суть задачи в том чтобы продемонстрировать зазор между возможностями алгоритма Дейкстры и алгоритмом Форда-Беллмана.

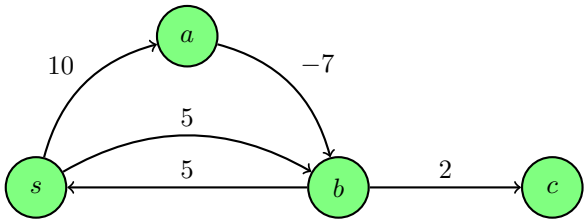
Рассмотрим к примеру вот такой граф:



	s	a	b	**u**
d[u]	0	10	5	
p[u]	–	s	s	

Результат работы алгоритма на нем представлен справа. Почему он такой? Потому что в начале (после обработки вершины s) мы обрабатываем вершину b, и т.к. мы пометили ее как обработанную дальнейшие изменения к ней уже не проходят (этот момент как раз опирается на то что ребра положительны – алгоритм Дейкстры жадный, если мы уже обработали вершину – расстояние до нее улучшиться не может).

Но давайте модифицируем его, будем обновлять расстояние даже если вершина уже была обработана, тогда на графе выше мы получим корректное расстояние до вершины b (а именно 3). Однако этого не достаточно, рассмотрим к примеру вот такой случай:



	s	a	b	c	**u**
d[u]	0	10	3	7	
p[u]	–	s	a	b	

Проблема в том что после обработки b мы уже не можем вытащить c и обработать его опять после того как расстояние до b улучшилось. При этом дерево получилось корректным, а вот сами расстояния – нет. □

Задача 2.2 (Задача 8 из листка 8)**.** Допустим, что нам надо нужно найти кратчайшие расстояния от данной вершины в графе с весами рёбер в интервале $0, 1, ...W$,где W – сравнительно небольшое число.

1. Покажите, что тогда алгоритм Дейкстры можно модифицировать, получив время работы $O(W|V| + |E|)$.

2. Постройте другой вариант алгоритма с оценкой $O((|V| + |E|) \log |W|)$.

Решение. 1. Первый пункт можно за это время решить и с помощью BFS (это обсуждалось в соответствующем разделе). Но тогда будет не очень понятно как решать пункт 2, поэтому все же приведем решение через модификацию Дейкстры.

Идея в следующем – можно поддерживать массив списков где j-й список содержит вершины до которых расстояние сейчас равно j – тогда минимальная вершина извлекается очень просто – мы идем по этому массиву списков и ищем

первый непустой, при этом так как новые списки могут добавляться только дальше можно не пробегать этот массив списков каждый раз с начала, а идти по нему в одну сторону. И мы берем вершину, смотрим на ее ребра, обновляем расстояния до вершин (при этом нам даже не надо удалять ее из более плохих списков где она возможно находилась – мы просто пропустим ее когда там ее увидим).

И так как в худшем случае размер этого массива будет $W|V|$ мы и получаем сложность $O(W|V| + |E|)$.

Важно отметить что весь массив поддерживать не надо – достаточно поддерживать его отрезок длины $2W$ – т.к. шаг больше чем на W невозможен.

2. Чтобы решить второй пункт надо модифицировать первый – использовать очередь с приоритетам для хранения этих списков.

□