Основные алгоритмы, комментарии по девятому листку (Графы II. Поиск в глубину)

Шибаев Иннокентий

April 6, 2021

1 По поводу теории

1.1 Поиск в глубину (DFS)

Идея этого алгоритма обхода графов очень простая – переходим по ребрам пока есть куда идти (пока не все вершины перед нами помечены как посещенные), иначе помечаем вершину как обработанную, и возвращаемся. Обычно если какие-то вершины после этого остались не посещены алгоритм запускают и из них.

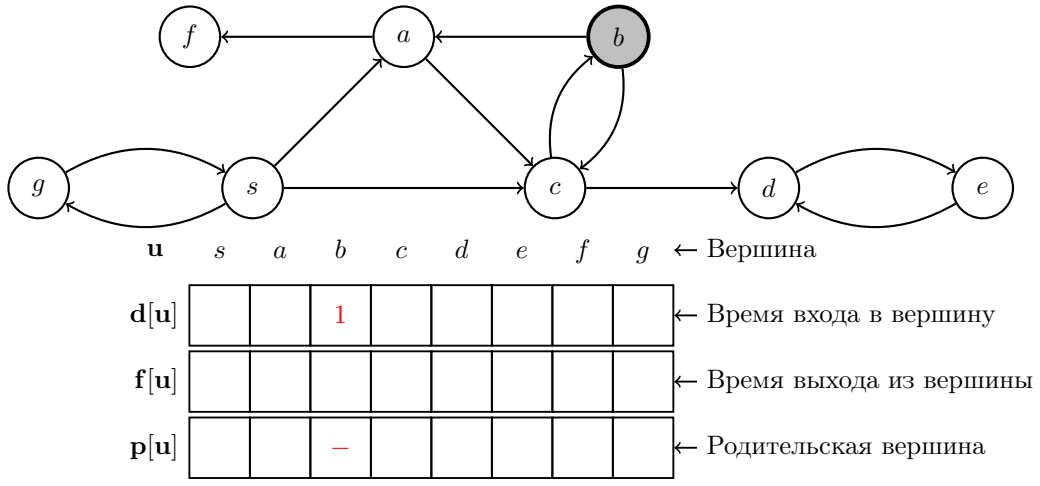
По сути это даже не столько алгоритм, сколько идея, некоторый базовый элемент на котором строятся многие другие алгоритмы, что-то вроде сортировок от мира алгоритмов на графах. Такие примеры мы обсудим в следующих разделах.

Сложность этого алгоритма как и у BFS – $O(|V|+|E|)$ (мы опять же проходим все вершины и просматриваем ребра из каждой).

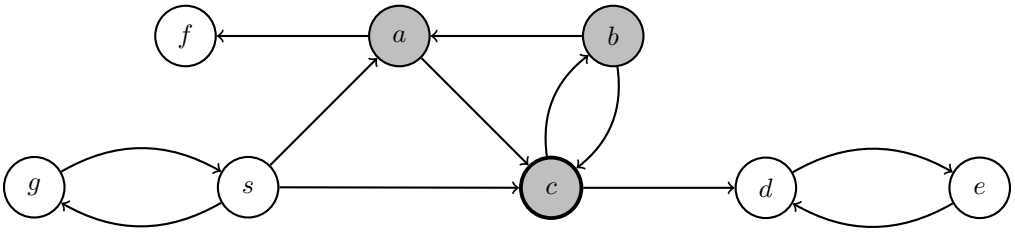
Перейдем к тому что можно делать на основе этого алгоритма.

1.2 Классификация ребер

При обходе графа с помощью DFS можно поддерживать счетчик времени, и записывать когда мы первый раз входим в вершину и выходим (т.е. полностью заканчиваем обрабатывать) из нее. Это соответственно дает массивы $d[u]$ и $f[u]$. Помимо этого можно так же как мы это делали для алгоритмов поиска минимального пути сохранять массив предков $p[u]$. Рассмотрим это все на примере (немного модифицированный граф с семинара). Начнем мы в вершине b:

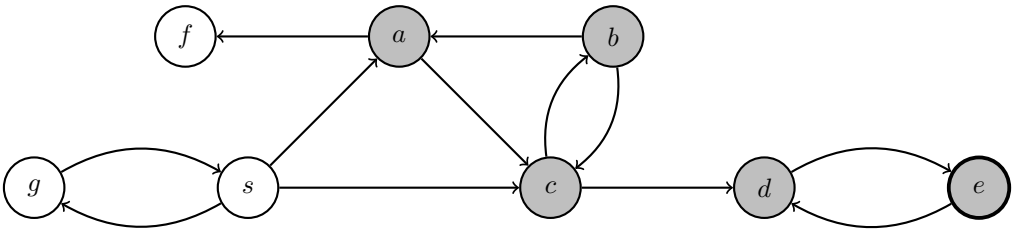


Из вершины в которой мы сейчас находимся (это вершина b, выделена жирным кружком, серый цвет означает что мы уже заходили в вершину но еще не закончили ее обрабатывать) есть 2 ребра: ребро (b, a) и ребро (b, c). Договоримся, что здесь и далее в таких случаях мы будем просматривать их в лексикографическом порядке, т.е. в начале мы пойдем по ребру (b, a). Там (в вершине a) будет та же ситуация, поэтому мы пойдем по ребру (a, c):



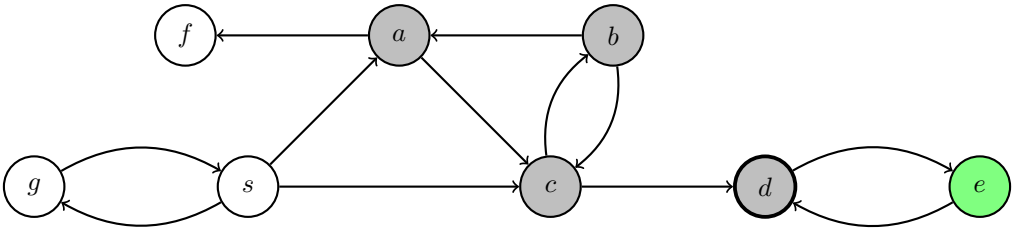
u	s	a	b	c	d	e	f	g
d[**u**]		2	1	3				
f[**u**]								
p[**u**]		b	–	a				

В вершине c мы видим ребро (c, b), по которому мы не будем переходить так как вершина b уже помечена как обрабатываемая, поэтому мы перейдем вместо этого по ребру (c, d) и, далее, по ребру (d, e):



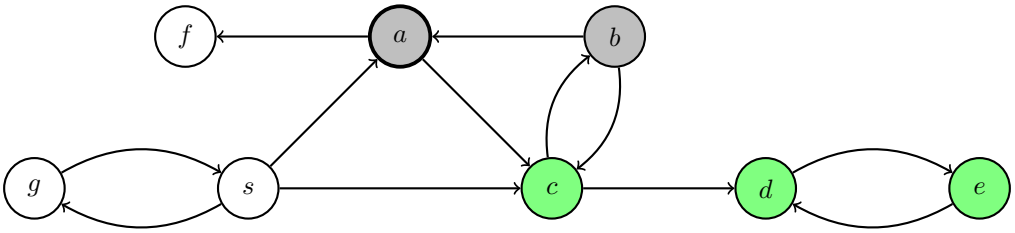
u	s	a	b	c	d	e	f	g
d[**u**]		2	1	3	4	5		
f[**u**]								
p[**u**]		b	–	a	c	d		

и для вершины e уже нет ребер по которым можно перейти, поэтому мы заканчиваем ее обрабатывать, помечаем ее, записываем время выхода и возвращаемся в вершину d:



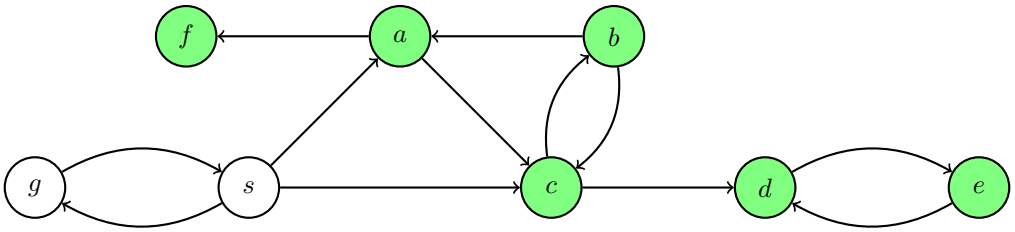
u	s	a	b	c	d	e	f	g
d[**u**]		2	1	3	4	5		
f[**u**]						6		
p[**u**]		b	–	a	c	d		

и то же самое повторяется с вершинами d и c:



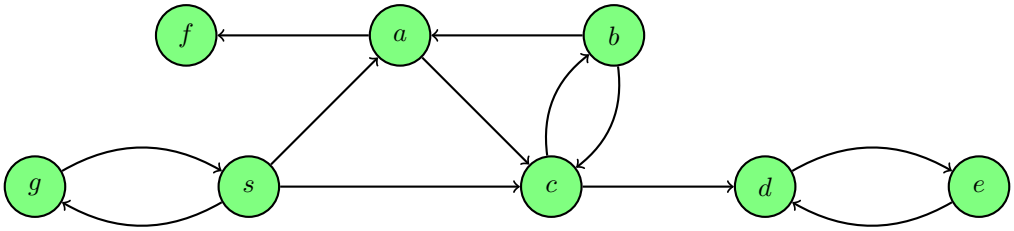
u	s	a	b	c	d	e	f	g
d[**u**]		2	1	3	4	5		
f[**u**]				8	7	6		
p[**u**]		b	–	a	c	d		

Для вершины a есть еще одно ребро которое ведет в необработанную вершину f, и, после ее обработки, мы, наконец, получаем



u	s	a	b	c	d	e	f	g
d[**u**]		2	1	3	4	5	9	
f[**u**]		11	12	8	7	6	10	
p[**u**]		b	–	a	c	d	a	

Теперь мы видим несколько не посещенных вершин, запустим DFS из вершины s (продолжая тот же счетчик времени) и получим



u	s	a	b	c	d	e	f	g
d[**u**]	13	2	1	3	4	5	9	14
f[**u**]	16	11	12	8	7	6	10	15
p[**u**]	–	b	–	a	c	d	a	s

А теперь перейдем собственно к классификации ребер. Их четыре вида:

1. Ребра дерева (леса) – те ребра графа по котором реально происходили переходы при обходе графа, фактически мы их записывали при заполнении массива $p[u]$.

2. Прямые ребра – предположим при обходе начатом из некоторой вершины s мы в начале посетили вершину u, а затем вершину v, и при этом существует ребро (u, v). Тогда это ребро мы назовем *прямым*. С точки зрения $d[u]$ и $f[u]$ это означает что выполняется следующая цепочка неравенств

$$d[u] < d[v] < f[v] < f[u]$$

   (хотя центральное верно всегда). Т.е. фактически «v потомок u» это то же что и «интервал времени в течение которого вершина v обрабатывалась вложен в соответствующий интервал u».

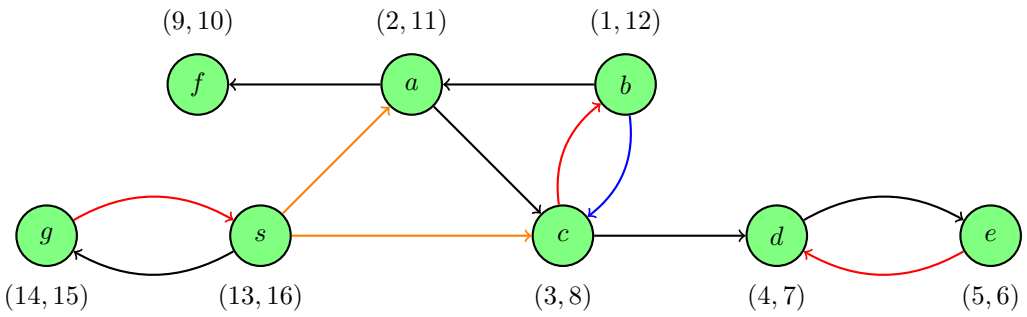
3. Обратные ребра – то же что и в прошлом пункте, только ребро (v, u). Неравенства остаются теми же – вершина v все еще потомок вершины u.

4. Перекрестные ребра – их проще всего описать сразу через неравенство. Пусть для вершин u, v и ребра (v, u) выполнено

$$d[u] < f[u] < d[v] < f[v]$$

   (заметим что существенным здесь является только центральное), тогда такое ребро называется *перекрестным*. Т.е. фактически условие в том что интервалы времени не пересекаются.

   Заметим, что ребра (u, v) тогда существовать не могло – иначе мы бы посетили вершину v раньше чем закрыли бы вершину u.



На графе выше числа рядом с вершинами это пары $(d[u], f[u])$, черным отмечены ребра дерева (в данном случае – леса), синим – прямые ребра (из тех что не являются ребрами дерева), красным отмечены обратные ребра, и оранжевым – перекрестные.

С перекрестными ребрами тут получилось не очень информативно, т.к. они появились только для вершин которые мы обработали следующей итерацией DFS (начав из другой вершины). К примеру если бы было ребро (f, c) то оно было бы перекрестным, т.к. их временные интервалы не пересекаются (добавление этого ребра очевидно бы ничего не изменило при работе алгоритма – когда мы дошли до вершины f вершина c уже была обработана).

1.3 Применение алгоритма DFS

Ниже приведены некоторые задачи для которых есть алгоритмы использующие DFS или его модификации.

1.3.1 Поиск Эйлерова цикла

Алгоритм обсуждавшийся на семинаре в основе своей использует как раз обход с помощью модификации DFS – мы идем вглубь пока не наткнемся на посещенную вершину, из которой уже нельзя будет перейти по какому-либо ребру, после чего выводим ее и возвращаемся, и делаем так пока не найдем вершину с еще не использованным ребром.

Или как на лекции – при нахождении посещенной вершины вырезаем цикл и еще раз запускаем DFS, а потом склеиваем циклы.

1.3.2 Определение компонент сильной связности (алгоритм Косараджу)

Опять же, этот алгоритм обсуждался на семинаре – мы применяем DFS, инвертируем ребра, и применяем его по порядку к вершинам отсортированным по убыванию $f[u]$.

1.3.3 Топологическая сортировка

Достаточно сделать DFS и выписать вершины в порядке обратном времени выхода. Зачем это нужно? К примеру чтобы определять как разрешать зависимости при сборке (линковке) исходных файлов в языках вроде $C++$.

1.3.4 Поиск точек сочленения/мостов

Точкой сочленения в связном неориентированном графе G называется вершина v такая что при ее удалении граф теряет связность. Мостом называется то же самое только для ребер.

И здесь довольно понятным образом возникает DFS – фактически вершина является точкой сочленения если из ее потомков нет обратных ребер в ее предков (в неориентированном графе можно говорить только об одном из видов прямых/обратных ребер, по понятным причинам). Примерно то же и для мостов.

Сам алгоритм мы здесь разбирать не будем, важно лишь отметить что он опять же основан на DFS.

1.3.5 Проверка графа на двудольность

Здесь на самом деле подходит как DFS так и BFS – идем и помещаем вершины по очереди в свои доли, попутно проверяя что коллизий (т.е. случаев когда есть ребро между вершинами одной доли) нет.

2 По поводу задач

Задача 2.1 (Задача 8 из листка 9)**.** Турнир с $|V|$ вершинами задан в виде матрицы смежности ($|V|^2$ памяти), предложите алгоритм, который находит общий сток за $O(|V|)$ (или говорит,что его нет). Общим стоком называют вершину, достижимую из любой вершины, такую, что из нее самой ребер не выходит.

Решение. *TL;DR: Поднимаемся в графе по вершинам начиная с первой, просматривая ребра только в вершины с большими номерами, т.к. меньшие уже проиграли (т.е. из них исходило ребро).*

Граф-турнир, это граф у которого для любой пары вершин $u, v \in V, u \neq v$ есть либо ребро (u, v), либо ребро (v, u). Иначе говоря – это ориентированный граф который получен путем определения ориентации у каждого ребра полного графа.

Решение за $O(|V|^2)$ очевидно – пройтись по всем вершинам и проверить что из них нет исходящих ребер. Если такая есть то это сток. Но как это можно сделать за $O(|V|)$?

Идея примерно следующая – т.к. общий сток это вершина их которой ребер не выходит, то давайте переходить из по первым попавшимся ребрам пока идти будет некуда – наверное это и будет кандидат на то чтобы быть искомым общим стоком.

Проблема в том как идти. Здесь нам поможет то что граф у нас задан матрицей смежности. Если мы вышли из какой-то вершины, то она уже, очевидно, стоком быть не может, и, пожалуй, мы бы хотели избегать дальнейшего захода в нее.

Давайте рассмотрим следующий алгоритм. Начнем в вершине с номером 1. Посмотрим, есть ребро из вершины 1 в вершину 2/ Если есть, то вершина 1 уже не может быть общем стоком, переходим к вершине 2. Если же ребра нет, то т.к. это турнир существует ребро $(2, 1)$, значит 2 уже не является общим стоком. В таком случае продолжим, и будем смотреть на ребра $(1, 3)$, $(1, 4)$ и т.д. пока мы либо не выйдем из вершины, либо не дойдем до конца.

Если мы дошли до конца то это значит что это и есть вершина-сток. Предположим мы перешли в какую-то вершину i. Т.к. мы не перешли ни в одну из вершин ранее, то вершины $2, \ldots, i-1$ проиграли вершине 1 – они не являются стоками. Вершина 1 также стоком не является, т.к. мы перешли из нее в вершину i.

Т.е. ни одну из вершин отрезка $1, \ldots, i-1$ проверять уже не надо! Значит мы можем проверять с $i+1$ вершины. В какой-то момент мы дойдем до конца (т.е. посмотрим на ребро $(j, |V|)$, это произойдет т.к. при каждом переходе мы увеличиваем индекс на 1, а вершин как раз $|V|$? т.е. в худшем случае мы дойдем до вершины $|V|$ и в ней уже не останется ребер в вершины с большими номерами). В этот момент надо проверить всю вершину j в которой мы оказались – если нет исходящих из нее ребер то мы нашли сток, иначе стока в графе не существует.

Почему это корректно? ~~По построению~~ Если мы оказались в вершине j то вершины $1, \ldots, j-1$ уже проигрывали (были ребра исходящие из них). После этого мы проверили и не нашли ни одного ребра вида (j, k) где $k \in \{j+1, \ldots, |V|\}$ – значит ни одна из этих вершин также не может быть общим стоком. Таким образом остался один кандидат – вершина j в которой мы сейчас находимся.

Теперь о сложности. Номер вершины в которую мы смотрим (второе значение в ребрах) только возрастает от 2 до $|V|$, поэтому всего мы просмотрим не более $|V| - 1$ ребер в процессе пока поднимаемся до вершины j в которой процесс остановится. После этого мы проверяем саму эту вершину – это тоже не более $|V| - 1$ проверок. Таким образом сложность получилась как раз $O(|V|)$. □