СТАТЬИ

Поиск в глубину и его применение

А. П. Лахно

Поиск в глубину (или обход в глубину) является одним из основных и наиболее часто употребляемых алгоритмов анализа графов.

Согласно этому алгоритму обход вершин графа осуществляется по следующему правилу. Начиная с некоторой вершины, мы идем по ребрам графа, пока не упремся в *тупик*. Вершина называется *тупиком*, если в ней нет исходящих ребер, ведущих в непосещенные вершины. После попадания в тупик мы возвращаемся назад вдоль пройденного пути, пока не обнаружим вершину, у которой есть исходящие ребра, ведущие в непосещенные вершины, и из нее идем по одному из таких ребер. Процесс кончается, когда мы возвращаемся в начальную вершину, а все соседние вершины уже оказались посещенными. Если после этого остаются непосещенные вершины, то повторяем поиск из одной из них в соответствии с вышеописанным алгоритмом. Так делаем до тех пор, пока не обнаружим все вершины графа.

Наиболее подходящим способом для реализации данного алгоритма является рекурсия. В этом случае все возвраты вдоль пройденного пути будут осуществляться автоматически, в результате работы механизма реализации рекурсии в языке программирования (заметим, что не все языки позволяют записывать в явном виде рекурсивные алгоритмы).

Исходный граф $G = (V, E)$, где V — множество вершин графа, а E — множество его ребер, может быть как ориентированным, так и неориентированным.

Переходя в вершину u из вершины v по ребру (v, u), мы запоминаем предшественника u при обходе в глубину: $p[u] = v$. Для вершин, у которых предшественников нет, положим $p[u] = -1$. Таким образом, получается *дерево поиска в глубину*. Если поиск повторяется из нескольких вершин, то получается несколько деревьев, образующих *лес поиска в глубину*. Лес поиска в глубину состоит из всех вершин исходного графа и ребер, по которым эти вершины впервые достигнуты.

Для наглядности будем считать, что в процессе работы алгоритма вершины графа могут быть белыми, серыми и черными. Изначально все вершины помечены белым цветом: `color`$[v] =$ `white`. Впервые обнаружив вершину v, мы красим ее серым: цветом `color`$[v] =$ `grey`. По окончании обработки всех исходящих ребер красим вершину v в черный цвет: `color`$[v] =$ `black`.

Таким образом, белый цвет соответствует тому, что вершина еще не обнаружена, серый — тому, что вершина уже обнаружена, но обработаны еще не все исходящие из нее ребра, черный — тому, что вершина уже обнаружена и все исходящие из нее ребра обработаны.

Помимо того, для каждой вершины в процессе поиска в глубину полезно запоминать еще два параметра: в $d[v]$ будем записывать «время» первого попадания в вершину, а в $f[v]$ — «время» окончания обработки всех исходящих из v ребер. При этом $d[v]$ и $f[v]$ представляют собой целые числа из диапазона от 1 до $2|V|$, где $|V|$ — число вершин графа.

Вершина v будет белой до момента $d[v]$, серой между $d[v]$ и $f[v]$, черной после $f[v]$.

Цвета вершин и пометки времени представляют собой удобный инструмент для анализа свойств графа и, как будет показано в дальнейшем, широко используются в различных алгоритмах на графах, в основе которых лежит поиск в глубину.

Ниже приводится схема возможной реализации поиска в глубину *Depth-first search* (*Dfs*):

```
1  Procedure Dfs(v);
2  begin
3    color[v] := grey;
4    time := time + 1; d[v] := time;
//цикл по всем ребрам, исходящим из v
5    for {u: (v, u) ∈ E} do
6      if color[u] = white then begin
7        p[u] := v;
8        Dfs(u)
9      end;
10   color[v] := black;
11   time := time + 1; f[v] := time
12 end;
// основная программа
13 for {v ∈ V} do begin
// инициализация значений
14   color[v] := white;
15   p[v] := -1;
16   d[v] := 0; f[v] := 0
17 end;
18 time := 0;
//цикл по всем вершинам
19 for {v ∈ V} do
20   if color[v] = white then Dfs(v);
```

Рассмотрим пошаговое исполнение алгоритма на примере конкретного ориентированного графа (жирным помечаются ребра, вошедшие в лес поиска в глубину):

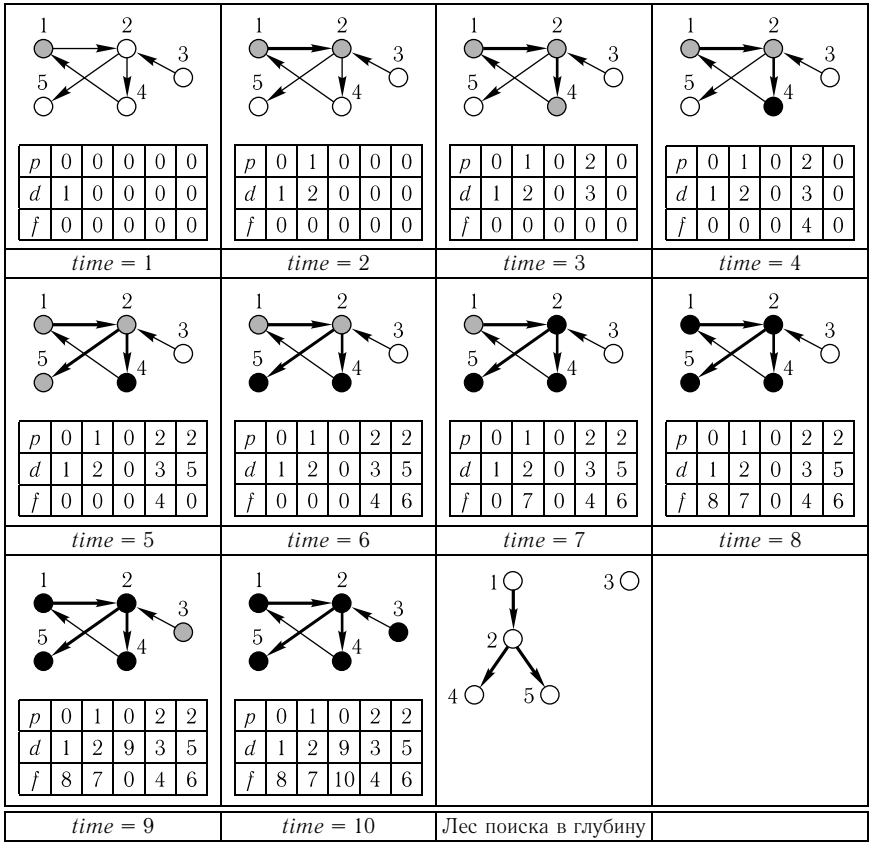

Рис. 1

Подсчитаем общее число операций при выполнении поиска в глубину на графе $G = (V, E)$. Изначальная инициализация данных (строки 13–18 алгоритма) и внешний цикл по вершинам (строки 19–20) требуют $O(V)$ времени.

Для каждой вершины процедура *Dfs* вызывается ровно один раз, причем время работы каждого вызова определяется временем, необходимым на просмотр всех исходящих из вершины ребер (5–9). Это время зависит от способа хранения графа.

Если хранить граф в виде матрицы смежности, то цикл в процедуре *Dfs* (строки 5–9) занимает $O(V)$ времени. Тогда суммарное время работы всех

вызовов *Dfs* равно $O(V^2)$. Таким образом, время работы поиска в глубину при использовании матрицы смежности равно $O(V^2)$.

Если же хранить граф списками смежности или списком ребер, то цикл (5–9) занимает O (число исходящих из вершины ребер) времени. Суммарное время работы всех вызовов *Dfs* равно $O(E)$, так как каждое ребро просматривается лишь однажды. Таким образом, время работы поиска в глубину при использовании списков смежности или списка ребер равно $O(V + E)$.

В графах, где число ребер E не велико (порядка числа вершин V), второй способ хранения, несмотря на несколько более сложную реализацию, дает ощутимый выигрыш во времени.

Свойства пометок времени

Очень красивое и важное свойство пометок времени состоит в том, что время обнаружения и время окончания обработки образуют правильную скобочную структуру. Действительно, будем обозначать обнаружение вершины открывающей скобкой с индексом номера вершины, а окончание обработки — закрывающей скобкой с индексом номера вершины. Тогда последовательность событий, выстроенная в порядке возрастания времени, будет правильно построенным скобочным выражением.

Так, например, для графа с рис. 1 мы получим следующее скобочное выражение:

time	1	2	3	4	5	6	7	8	9	10
	$({}^1$	$({}^2$	$({}^4$	${}^4)$	$({}^5$	${}^5)$	${}^2)$	${}^1)$	$({}^3$	${}^3)$

Поясним, из каких соображений следует это свойство. Для белой вершины v обозначим через $W(v)$ множество всех белых вершин, доступных из вершины v по путям, в которых все промежуточные вершины также являются белыми.

Вызов процедуры *Dfs* для белой вершины v делает серой эту вершину, затем полностью обрабатывает все вершины из $W(v)$, оставляя серые и черные вершины без изменений, после чего делает вершину v черной.

Таким образом, для любой вершины u из $W(v)$ верно: $d[v] < d[u] < f[u] < f[v]$, что соответствует выражению $({}^u\ldots({}^v\ldots{}^v)\ldots{}^u)$.

Для строгого доказательства утверждения о правильной скобочной структуре пометок времени можно рассуждать по индукции. При этом, доказывая требуемое свойство рекурсивной процедуры *Dfs*, предполагаем, что для всех внутренних рекурсивных вызовов это свойство уже выполнено.

При поиске в глубину как в ориентированном, так и в неориентированном графе для любых двух вершин u и v выполняется ровно одно из трех утверждений:

1) отрезки $[d[u], f[u]]$ и $[d[v], f[v]]$ не пересекаются;

2) отрезок $[d[u], f[u]]$ целиком содержится внутри отрезка $[d[v], f[v]]$, и u — потомок v в дереве поиска в глубину;
3) отрезок $[d[v], f[v]]$ целиком содержится внутри отрезка $[d[u], f[u]]$, и v — потомок u в дереве поиска в глубину.

Эти утверждения позволяют быстро определять взаимное расположение вершин в лесу поиска в глубину.

Классификация ребер

Ребра ориентированного графа делятся на несколько категорий в зависимости от их роли при поиске в глубину.

1. *Ребра деревьев* — это ребра, входящие в лес поиска в глубину. На рис. 2 это ребра (1, 2), (2, 3), (2, 5), (3, 4).
2. *Обратные ребра* — это ребра, соединяющие вершину с ее предком в дереве поиска в глубину (ребра-циклы, возможные в ориентированных графах, считаются обратными ребрами). На рис. 2 это ребра (5, 1), (6, 6).
3. *Прямые ребра* — это ребра, соединяющие вершину с ее потомком, но не входящие в лес поиска в глубину: (2, 4) на рис. 2.
4. *Перекрестные ребра* — все остальные ребра графа. Они могут соединять две вершины из одного дерева поиска в глубину, если ни одна их этих вершин не является предком другой, или же вершины из разных деревьев: (5, 4), (6, 1) на рис. 2.

Тип ребра (v, u) можно определить по цвету вершины u в момент, когда ребро исследуется в первый раз: белый цвет означает ребро дерева ((v, u) войдет в лес поиска в глубину), серый (u является предком v) — обратное ребро, черный (ни одна из них не является предком другой) — прямое или перекрестное ребро.

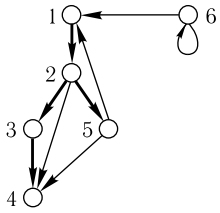

Рис. 2

Эта классификация оказывается полезной в различных задачах, решаемых с использованием поиска в глубину. Так например, ориентированный граф не имеет циклов тогда и только тогда, когда поиск в глубину не находит в нем обратных ребер.

В неориентированном графе одно и то же ребро можно рассматривать с разных концов, и в зависимости от этого оно может попасть в разные категории. Будем относить ребро к той категории, которая стоит раньше в классификации для ориентированных графов.

Например, для графа с рис. 3 ребро (1, 4) можно считать как прямым, так и обратным. В соответствии с принятым утверждением, мы отнесем его к категории обратных.

Оказывается, что при принятых соглашениях прямых и перекрестных ребер в неориентированных графах не будет. Действительно, пусть (v, u) — произвольное ребро неориентированного графа. Без ограничения общности можно считать, что при поиске в глубину $d[v] < d[u]$. Тогда вершина v должна быть обнаружена и обработана раньше, чем закончится обработка вершины u, так как v содержится и в списке вершин, смежных с u. Если ребро (v, u) в первый раз обрабатывается в направлении от v к u, то (v, u) становится ребром дерева. Если же оно впервые обрабатывается в направлении от u к v, то оно становится обратным ребром (в этот момент вершина v серая, так как она обнаружена, но ее обработка еще не завершена).

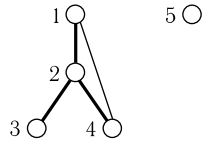

Рис. 3

Таким образом, для графа на рис. 3 ребра (1, 2), (2, 3) и (2, 4) будут ребрами дерева, а (1, 4) — обратным ребром.

Перейдем к рассмотрению стандартных задач, решаемых с помощью поиска в глубину. Напомним, что в предыдущем разделе мы уже фактически показали, как с помощью поиска в глубину проверить ацикличность ориентированного графа или, наоборот, убедиться в наличии циклов.

Компоненты связности

Неориентированный граф G называется *связным*, если любые две его вершины достижимы друг из друга по ребрам графа. Несвязный граф распадается на несколько связных частей, никакие две из которых не соединены ребрами. Эти части и называются *компонентами связности* графа.

Так например, граф на рис. 4 распадается на три компоненты связности: (1, 6), (2, 7, 5, 4), (3).

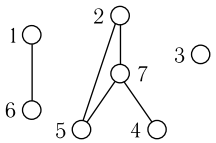

Рис. 4

Возникает задача разбиения неориентированного графа $G = (V, E)$ на компоненты связности.

Эта задача решается с помощью поиска в глубину следующим образом. Запускаем поиск в глубину из первой вершины. Все вершины, обнаруженные в ходе этого алгоритма, принадлежат одной компоненте связности. Если остались необнаруженные вершины, то запускаем поиск в глубину из любой из них. Вновь обнаруженные вершины принадлежат другой компоненте связности. Повторяем этот процесс до тех пор, пока не останется необнаруженных вершин, каждый раз относя вновь обнаруженные вершины к очередной компоненте связности.

Ниже приводится схема возможной реализации алгоритма разбиения графа на компоненты связности:

```
1  Procedure Dfs(v);
2  begin
```

```
3    comp[v] := num;
4    for {u:(v, u) ∈ E} do
5      if comp[u] = 0 then Dfs(u)
6  end;
//основная программа
7  for {v ∈ V} do comp[v] := 0;
8  num := 0;
9  for {v ∈ V} do
10  if comp[v] = 0 then begin
//найдена очередная компонента связности
11    inc(num);
12    Dfs(v)
13  end;
```

По окончании работы программы переменная `num` будет содержать количество компонент связности графа G, а в массиве `comp` будут храниться номера компонент, к которым принадлежат соответствующие вершины.

Граф связен в том и только том случае, когда все его вершины обнаружены после первого же запуска поиска в глубину (`num = 1`, т. е. граф состоит из одной компоненты связности).

Топологическая сортировка

Пусть имеется ориентированный граф G без циклов. Задача о топологической сортировке этого графа состоит в том, чтобы указать такой порядок вершин, при котором ребра графа ведут только из вершин с меньшим номером к вершинам с бóльшим номером. Если в графе есть циклы, то такого порядка не существует. Можно переформулировать задачу о топологической сортировке следующим образом: расположить вершины графа на горизонтальной прямой так, чтобы все ребра графа шли слева направо. В жизни это соответствует, например, следующим проблемам: в каком порядке следует располагать темы в школьном курсе математики, если известно для каждой темы, знания каких других тем необходимы для ее изучения; в каком порядке следует надевать на себя комплект одежды, начиная с нижнего белья и заканчивая верхней одеждой. Очевидно, что зачастую задача топологической сортировки имеет не единственное решение.

На рис. 5 представлен граф и один из вариантов его топологической сортировки: 1, 5, 2, 6, 3, 4. Заметим, что, например, вершину с номером 6 можно поставить в любое место топологической сортировки этого графа. Вершины 1 и 5 также могут располагаться в другом порядке (сначала 5, а затем 1). В остальном порядок топологической сортировки вершин данного графа фиксирован.

Алгоритм нахождения топологической сортировки также основан на поиске в глубину. Применим поиск в глубину к нашему графу G. Завершая

обработку каждой вершины (делая ее черной), заносим ее в начало списка. По окончании обработки всех вершин полученный список будет содержать

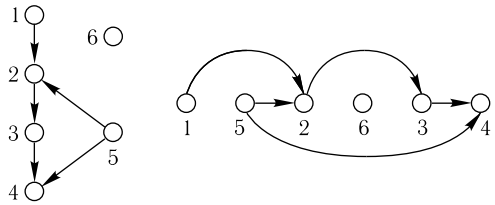

Рис. 5

топологическую сортировку данного графа. Обозначим количество вершин в графе n. Так как в результирующий список должны попасть все n вершин нашего графа, то реализовать его можно на обычном массиве, записывая в него элементы списка начиная с n-го места в массиве и заканчивая первым.

Покажем, что полученный список действительно удовлетворяет основному свойству топологической сортировки: все ребра ведут от вершин с меньшим номером к вершинам с бóльшим номером в нашем списке. Предположим противное: пусть существует ребро (v, u) такое, что вершина u встречается в нашем списке раньше вершины v. Но тогда обработка вершины v была завершена раньше, чем обработка вершины u. Это означает, что в момент окончания обработки вершины v вершина u могла быть либо белой, либо серой. В первом случае мы должны были бы пройти по ребру (v, u), но не сделали этого. Во втором случае поиск в глубину нашел бы обратное ребро, т. е. граф G содержал бы циклы. Оба случая приводят к противоречию, а значит, наше предположение неверно и указанный список действительно является топологической сортировкой.

Ниже приводится схема возможной реализации алгоритма построения топологической сортировки. Причем, если в графе есть циклы, что означает невозможность его топологической сортировки, это также будет обнаружено в процессе работы алгоритма.

```
1  Procedure Dfs(v);
2  begin
3    color[v] := grey;
4    for {u:(v, u) ∈ E} do begin
5      if color[u] = white then Dfs(v)
6      else if color[u] = grey then
7           {граф имеет циклы, конец}
8    end;
9    inc(num); list[n-num+1] := v;
10   color[v] := black
```