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):

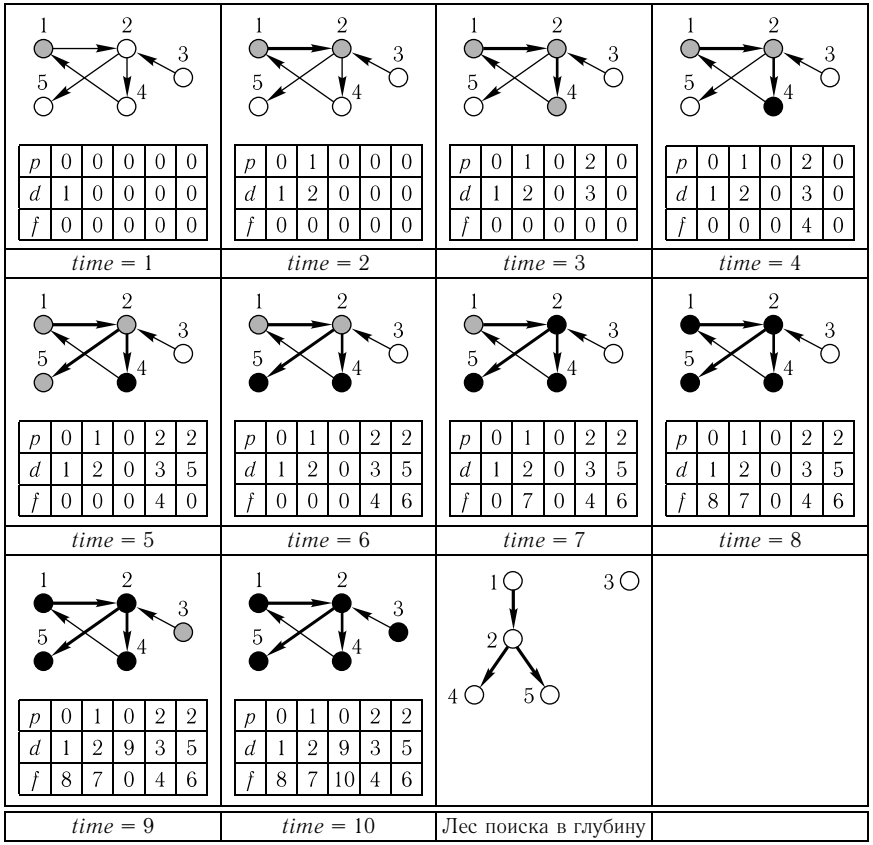

Рис. 1

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.

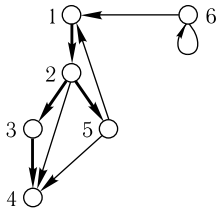

Рис. 2

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).

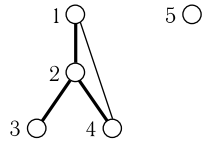

Рис. 3

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).

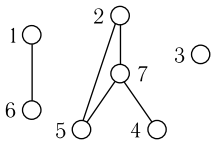

Рис. 4

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

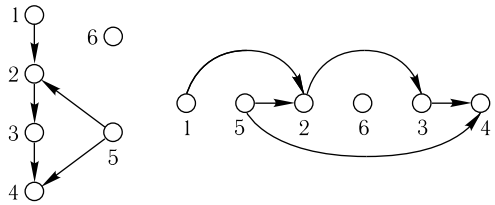

Рис. 5

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
```

```
11 end;
//основная программа
12 for v := 1 to n do begin
13   color[v] := white; list[v] := 0
14 end;
15 num := 0;
16 for v := 1 to n do
17   if color[v] = white then Dfs(v);
//печатаем вариант топологической сортировки
18 for v := 1 to n печать(list[v]);
```

По окончании работы алгоритма массив `list` будет содержать топологическую сортировку данного графа. Условие в строке 6 осуществляет проверку на наличие циклов (ориентированный граф не имеет циклов тогда и только тогда, когда поиск в глубину не находит в нем обратных ребер). При нахождении обратного ребра (строка 7) следует выдать соответствующее сообщение и завершить исполнение алгоритма. Последнее для рекурсивного алгоритма не совсем тривиально.

Мосты

Мостом неориентированного графа G называется ребро, при удалении которого увеличивается количество компонент связности графа. Соответственно, для связного графа мостом называется ребро, при удалении которого граф перестает быть связным.

При удалении всех мостов граф распадается на компоненты связности, которые называются *компонентами реберной двусвязности*.

Между любыми двумя вершинами одной компоненты реберной двусвязности существуют, по крайней мере, два пути, не пересекающиеся по ребрам. Верно и обратное утверждение: любые две вершины, между которыми существуют два пути, не пересекающиеся по ребрам, принадлежат одной компоненте реберной двусвязности.

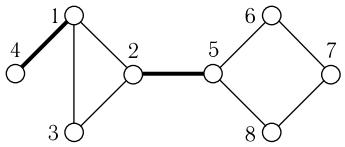

Рис. 6

Между двумя компонентами реберной двусвязности не может быть более одного ребра — в противном случае они образовывали бы одну компоненту реберной двусвязности. Если две различные компоненты реберной двусвязности соединены ребром, то это ребро — мост.

Так, например, для графа на рис. 6 мостами будут ребра: (1, 4) и (2, 5), а компонентами реберной двусвязности, соответственно, наборы вершин: (1, 2, 3), (4), (5, 6, 7, 8).

Рассмотрим задачу нахождения мостов для заданного неориентированного графа G.

Эта задача решается с помощью двух поисков в глубину.

При первом поиске, проходя по ребру (v, u) в направлении от вершины v к вершине u, «ориентируем» его против направления движения, т. е. запрещаем прохождение по ребру (v, u) в направлении от v к u. При обнаружении новой вершины заносим ее в конец списка `list`.

Применим описанный алгоритм к графу на рис. 6. Начиная из вершины 1, проходим по ребрам (1, 2) и (2, 3), ориентируя их против направления движения. Из вершины 3 не выходят ребра, ведущие в непосещенные вершины. Возвращаемся в вершину 2, проходим по ребрам (2, 5), (5, 6), (6, 7) и (7, 8). Возвращаемся в вершину 1, проходим по ребру (1, 4). Возвращаемся в вершину 1 и завершаем поиск в глубину.

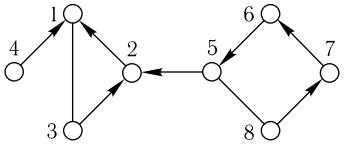

Рис. 7

На рис. 7 представлено как будет выглядеть граф с рис. 6 по окончании первого поиска в глубину. При этом по «ориентированным» ребрам можно ходить только в направлении, указанном стрелками, а по «неориентированным» ребрам — в обоих направлениях. Список `list` выглядит следующим образом: 1, 2, 3, 5, 6, 7, 8, 4.

Второй поиск в глубину осуществляется с учетом «ориентированных» ребер. Внешний цикл по вершинам должен идти в том порядке, в каком они записаны в списке `list`. Получающиеся деревья поиска красим каждое в свой цвет. Эти деревья являются компонентами реберной двусвязности нашего графа. Для графа на рис. 7 получим следующие три дерева поиска: (1, 3, 2), (5, 8, 7, 6) и (4).

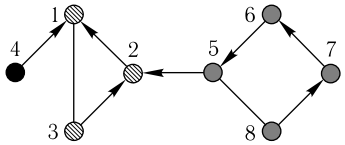

Рис. 8

Теперь остается только выбрать ребра исходного графа G, соединяющие вершины разного цвета (рис. 8): (1, 4) и (2, 5) — именно они и будут мостами.

Поясним, почему деревья поиска в глубину, получающиеся во время второго поиска, являются компонентами реберной двусвязности нашего графа.

Пусть очередное дерево поиска получилось в результате запуска из некоторой вершины x компоненты реберной двусвязности A. Докажем, что полученное дерево совпадает с A, в предположении, что все деревья поиска, полученные ранее, действительно образуют компоненты реберной двусвязности.

Во-первых, покажем, что дерево поиска в глубину не может содержать вершин из какой-то другой компоненты реберной двусвязности B (рис. 9).

Если между компонентами A и B нет ребер, то утверждение очевидно. Пусть между A и B есть некоторое ребро (v, u) (более одного ребра меж-

ду A и B быть не может, поскольку в противном случае они образовывали бы одну компоненту реберной двусвязности).

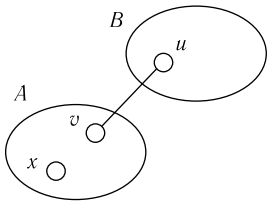

Рис. 9

Если вершина u уже обработана, то пройти по ребру (v, u), а значит, и попасть в компоненту B нельзя.

Если же вершина u еще не обработана, то она стоит в списке `list` заведомо позже вершины v. Это значит, что при первом поиске в глубину мы попали в v, когда u была еще белой. Не пройдя по ребру (v, u), из вершины v в вершину u попасть невозможно (иначе A и B образовывали бы одну компоненту реберной двусвязности). Поэтому при первом поиске в глубину «сориентируем» ребро (v, u) в направлении от u к v, а значит, при втором поиске в глубину попасть из A в B будет уже невозможно.

Покажем теперь, почему дерево поиска, начатого из некоторой вершины x компоненты реберной двусвязности A, будет содержать все вершины из A (рис. 10). Будем рассуждать от противного. Предположим, что некоторая вершина y, лежащая в этой же компоненте, не войдет в дерево поиска. Поскольку по предположению все деревья поиска, полученные ранее, действительно образуют компоненты реберной двусвязности, то y может не попасть в дерево второго поиска в глубину только в том случае, когда она недостижима из x в графе с учетом ориентации ребер. Обозначим через X множество вершин компоненты A, достижимых из вершины x при втором поиске в глубину, а через Y — множество вершин компоненты A, из которых при втором поиске в глубину достижима вершина y. Поскольку X и Y лежат внутри одной компоненты реберной двусвязности, то между ними есть, по крайней мере, два ребра. А так как вершина y недостижима из вершины x, то все ребра между X и Y ориентированы от Y к X. Все ориентированные ребра принадлежат деревьям первого поиска в глубину, причем ребра между X и Y принадлежат одному дереву (X и Y — подмножества компоненты реберной двусвязности A, а компонента реберной двусвязности всегда связна). Но тогда получается, что при построении одного дерева поиска в глубину мы из X в Y попадали минимум два раза, а из Y в X ни одного, чего быть не может — противоречие. Следовательно, все вершины A достижимы из x, а значит, войдут в дерево поиска.

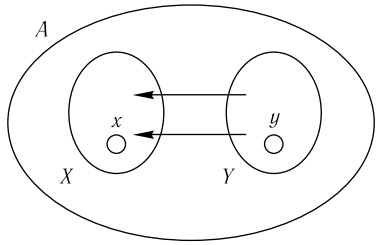

Рис. 10

Таким образом, деревья поиска в глубину, получающиеся при втором поиске в глубину, являются компонентами реберной двусвязности исходного графа G.

Ниже приводится схема возможной реализации алгоритма поиска мостов в неориентированном графе $G = (V, E)$:

```
//первый поиск в глубину
1   Procedure Dfs1(v);
2   begin
3     color[x] := 1;
4     inc(num); list[num] := v;
5     for {u:(v, u)   E} do
6       if color[u] = 0 then begin
7         Dfs1(u);
8       {запретить прохождение по ребру (v, u) в направлении от v к u}
9       end;
10   end;
//второй поиск в глубину
11  Procedure Dfs2(v);
12  begin
13    color[x] := num;
14    for {u:(v, u)   E} do
15      if (color[u] = 0) and {можно идти от v к u} then Dfs2(u);
16  end;
//основная программа
//вызов первого поиска
17  for v := 1 to n do begin
18    color[v] := 0; list[v] := 0;
19  end;
20  num:=0;
21  for v:=1 to n do
22    if color[v] = 0 then Dfs1(v);
//вызов второго поиска
23  for v := 1 to n do color[v] := 0;
24  num:=0;
25  for v:=1 to n do
26    if color[list[v]] = 0 then begin
27      inc(num);
28      Dfs2(list[v]);
29    end;
//печатаем ребра, являющиеся мостами
30  for {v, u:(v, u)   E} do
31    if (color[v]<>color[u]) then печать((v, u))
```

Для того чтобы запрещать прохождение по ребру в заданном направлении (строка 8), а также проверять возможность прохождения (строка 15),

полезно вместе с каждым ребром хранить два флажка, отвечающие за возможность прохождения по ребру в каждом из направлений.

По окончании работы программы переменная `num` будет содержать количество компонент реберной двусвязности заданного графа, а в массиве `color` будут храниться номера компонент реберной двусвязности, к которым принадлежат соответствующие вершины.

Точки сочленения

Точкой сочленения неориентированного графа G называется вершина, при удалении которой вместе со всеми смежными ребрами увеличивается количество компонент связности графа. Соответственно, для связного графа точкой сочленения называется вершина, при удалении которой граф перестает быть связным.

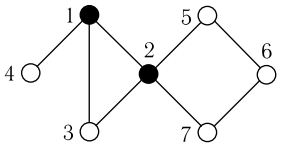

Рис. 11

Так, например, для графа на рис. 11 точками сочленения будут вершины 1 и 2.

Рассмотрим задачу нахождения точек сочленения для заданного неориентированного графа G.

Эта задача решается с помощью поиска в глубину следующим образом.

Применив к графу, изображенному на рис. 11, алгоритм поиска в глубину, получим некоторый лес поиска в глубину (рис. 12).

Корень дерева (начальная вершина) поиска в глубину является точкой сочленения тогда и только тогда, когда у него более одного сына в дереве поиска в глубину. Сыновьями вершины v являются вершины, впервые обнаруженные из v при поиске в глубину. Для графа, изображенного на рис. 12, вершина 1 имеет двух сыновей: 4 и 2, и поэтому она является точкой сочленения.

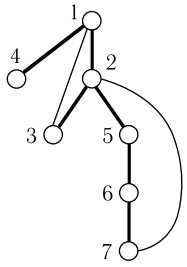

Рис. 12

Действительно, если корень дерева имеет только одного сына или же вообще не имеет сыновей, то он, очевидно, не является точкой сочленения. Если же у корня более одного сына, то к нему «подвешены» несколько поддеревьев, между которыми нет ребер (при поиске в глубину в неориентированном графе перекрестных ребер быть не может). Таким образом, при удалении корня количество компонент связности увеличится, а значит, он является точкой сочленения.

Так для графа, изображенного на рис. 12, к вершине 1 «подвешены» два поддерева. Первое состоит из одной вершины 4, второе — из вершин 2, 3, 5, 6, 7. При удалении вершины 1 граф распадается на две компоненты связности: (4) и (2, 3, 5, 6, 7).

Для того чтобы понять, является ли промежуточная вершина v дерева поиска в глубину точкой сочленения, надо узнать, существует ли поддерево с корнем в одном из сыновей u вершины v, которое «отсоединится» от дерева поиска при удалении самой вершины v. Поддерево может «отсоединиться» от дерева поиска в том и только том случае, когда из вершин этого поддерева нет обратных ребер, ведущих в вершины, обнаруженные до вершины v.

Пусть $up[u]$ — минимальное время обнаружения среди всех вершин тех, которые могут быть достигнуты из поддерева с корнем в вершине u при прохождении ровно по одному обратному ребру.

Промежуточная вершина дерева поиска в глубину v является точкой сочленения тогда и только тогда, когда у нее существует сын u такой, что $up[u] \geqslant d[v]$, где $d[v]$ — время обнаружения вершины v. В соответствии с этим признаком, вершина 2 графа (рис. 12) является точкой сочленения, поскольку у вершины 2 есть сын 5 такой, что $up[5] = d[2]$ (в поддереве с корнем в вершине 5 есть только одно обратное ребро (7, 2)).

Действительно, если существует такой сын u вершины v, что $up[u] \geqslant d[v]$, то при удалении вершины v поддерево с корнем u отделится от дерева, образовав новую компоненту связности, а значит, v — точка сочленения. Если же такого сына не существует, то при удалении v количество компонент связности не увеличится, так как все поддеревья с корнями в сыновьях вершины v соединены с «верхней» частью исходного дерева обратными ребрами.

Таким образом, для нахождения точек сочленения в процессе поиска в глубину необходимо отслеживать количество сыновей $ch[v]$ у корней деревьев поиска в глубину, а также вычислять величину $up[v]$ для всех вершин графа.

Величина $up[v]$ вычисляется как минимум из величин $up[u]$ для всех вершин u, являющихся сыновьями v в дереве поиска в глубину, и времен обнаружения $d[w]$ всех вершин w, достижимых непосредственно из v по обратному ребру. Изначально $up[v]$ присваивается значение $n + 1$, заведомо большее времени обнаружения любой вершины.

Ниже приводится схема возможной реализации алгоритма поиска точек сочленения в неориентированном графе $G = (V, E)$:

```
1   Procedure Dfs(v);
2   begin
3     color[v] := 1; inc(time);
4     d[v] := time; up[v] := n + 1;
5     for {u:(v, u)   E} do
6       if color[u] = 0 then begin
7         inc(ch[v]); Dfs(u);
          //u является сыном v в дереве поиска
```

```
8         up[v] := min(up[v], up[u]);
          //проверка внутренних вершин
9         if up[u] >= d[v] then
10          r[v] := true
11      end else
          //u достижима из v по обратному ребру
12        up[v] := min(up[v], d[u]);
13  end;
//основная программа
14  for v := 1 to n do begin
      //инициализация данных
15    color[v] := 0; d[v] := 0;
16    up[v] := 0; ch[v] := 0;
17    r[v] := false;
18  end;
19  time:=0;
20  for v := 1 to n do
21    if color[v] = 0 then begin
22      Dfs(v);
        //проверка количества детей у корней деревьев
23      if ch[v] > 1 then r[v] := true
24                   else r[v] := false;
25    end;
```

По окончании работы программы массив r содержит информацию о том, какие вершины являются точками сочленения: $r[v] =$ `true` тогда и только тогда, когда v — точка сочленения данного графа.

Поиск мостов и точек сочленения представляет интерес, например, для анализа надежности компьютерных сетей.

Задача о поиске мостов соответствует вопросу нахождения соединительных линий, при поломке одной из которых сеть перестает быть связной, а задача о поиске точек сочленения позволяет решить вопрос нахождения компьютеров, при поломке одного из которых сеть перестает быть связной.

Компоненты сильной связности

Компонентой сильной связности ориентированного графа называется максимальное множество вершин, в котором существуют пути из любой вершины в любую другую. Так например, граф на рис. 13 состоит из четырех компонент сильной связности: (1, 2, 4), (3, 5), (6) и (7).

Каждая вершина ориентированного графа G принадлежит какой-либо компоненте сильной связности, но некоторые ребра могут не принадлежать никакой компоненте сильной связности. Такие ребра соединяют вершины из разных компонент сильной связности.

Связи между компонентами сильной связности можно представить путем создания *конденсации* графа G. Конденсацией графа G называется граф, построенный следующим образом. Наборы вершин, образующие одну компоненту сильной связности C исходного графа, сливаются в одну вершину конденсации C^*. Из вершины C^* в вершину D^* конденсации есть ребро тогда и только тогда, когда в графе G есть ребро, ведущее из некоторой вершины x компоненты сильной связности C в некоторую вершину y компоненты сильной связности D. Конденсация графа, изображенного на рис. 13, представлена на рис. 14. Вершины 1, 2, 4 исходного графа, образующие одну компоненту сильной связности, сливаются в одну вершину, вершины 3, 5 — во вторую, вершина 6 — в третью, а вершина 7 — в четвертую. Ребра (4, 3), (2, 3) и (6, 7) отражены в конденсации, поскольку соединяют вершины разных компонент сильной связности.

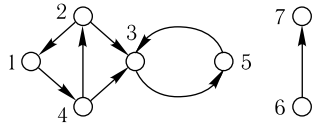

Рис. 13

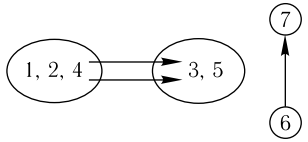

Рис. 14

В конденсации не может быть циклов, поскольку если бы существовал цикл, то все компоненты сильной связности, входящие в этот цикл, образовывали бы одну компоненту сильной связности.

Рассмотрим задачу нахождения компонент сильной связности ориентированного графа $G = (V, E)$.

Эта задача решается с помощью двух поисков в глубину по следующему алгоритму.

1. Сначала выполняется поиск в глубину на исходном графе G. По окончании обработки очередной вершины заносим ее в начало списка `list`.
2. Строим транспонированный граф $G^T = (V, E^T)$, полученный из исходного графа G заменой направления всех ребер на противоположное.
3. Выполняется поиск в глубину на графе G^T. При этом во внешнем цикле вершины перебираются в том порядке, в каком они записаны в список `list`, т. е. в порядке убывания времени выхода. Каждое дерево поиска в глубину красится в свой цвет.

Деревья поиска в глубину, получающиеся на 3-м шаге, и будут компонентами сильной связности исходного графа G. Докажем это утверждение.

Сначала покажем, что если две вершины v и u принадлежат одной компоненте сильной связности, то они войдут в одно дерево второго поиска

в глубину. Поскольку вершины v и u принадлежат одной компоненте сильной связности, то в исходном графе G существует путь как из v в u, так и из u в v. Поскольку граф G^T получен из графа G транспонированием всех его ребер, то в нем также существует путь как из v в u, так и из u в v, а следовательно, v и u попадут в одно дерево второго поиска в глубину.

Теперь покажем, что если две вершины v и u принадлежат одному и тому же дереву второго поиска в глубину, то они лежат в одной компоненте сильной связности графа G. Пусть x — корень дерева поиска, которому принадлежат вершины v и u. Поскольку v является потомком x, то в графе G^T есть путь от вершины x к вершине v, а в графе G, соответственно, есть путь от вершины v к вершине x.

При поиске в глубину на графе G^T вершина v обнаружена позже, чем вершина x, т. е. x имеет меньший номер в списке `list`, а значит, обработка вершины v заканчивается раньше, чем обработка вершины x при первом поиске в глубину.

Предположим, что при первом поиске в глубину в графе G вершина v обнаружена раньше вершины x. Но тогда из-за наличия пути от вершины v к вершине x мы обнаружим и завершим обработку вершины x до окончания обработки вершины v, что противоречит их взаимному расположению в списке `list`.

Значит, предположение неверно и обнаружение x происходит при первом поиске в глубину раньше, чем обнаружение v. Таким образом, $d[x] < d[v] < f[v] < f[x]$, т. е. v является потомком x в дереве первого поиска в глубину, а значит, в графе G существует путь из x в v. А так как существует и путь из v в x, то x и v принадлежат одной компоненте сильной связности.

Аналогично доказывается, что x и u принадлежат одной компоненте сильной связности. Поэтому v и u тоже принадлежат одной компоненте сильной связности, так как существует путь из вершины v в вершину u через вершину x и путь из вершины u в вершину v через вершину x.

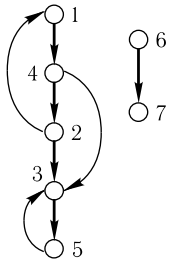

Рис. 15

Таким образом показано, что деревья поиска в глубину, получающиеся на 3-м шаге, являются компонентами сильной связности исходного графа G.

Рассмотрим работу приведенного алгоритма на примере графа, изображенного на рис. 13. Сначала выполняется первый поиск в глубину (рис. 15) на исходном графе G: начиная из вершины 1 проходим по ребрам (1, 4), (4, 2), (2, 3), (2, 5). Дальше идти некуда, возвращаемся в вершину 1, последовательно занося вершины, из которых уходим, в начало списка `list`. Начиная из вершины 6 идем по ребру (6, 7), после чего возвращаемся в вершину 6 и завершаем первый поиск

в глубину. После его окончания список `list` выглядит следующим образом: 6, 7, 1, 4, 2, 3, 5.

Строим по графу G (рис. 13) транспонированный граф G^T (рис. 16).

Запускаем второй поиск в глубину, перебирая во внешнем цикле вершины в соответствии со списком `list`. При этом вершина 6 составляет первое дерево поиска, вершина 7 — второе, вершины 1, 2, 4 — третье, а вершины 3 и 5 — четвертое. Красим вершины в соответствии с номером их дерева поиска (рис. 16). Разбиение вершин по цветам соответствует разбиению исходного графа G на компоненты сильной связности.

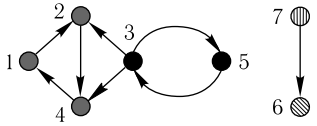

Рис. 16

Ниже приводится схема возможной реализации алгоритма нахождения компонент сильной связности для заданного ориентированного графа G:

```
//первый поиск в глубину
1   Procedure Dfs1(v);
2   begin
3     color[v] := 1;
4     for {u:(v, u) ∈ E} do
5         if color[u] = 0 then Dfs1(u);
6     dec(num); list[num] := v;
7   end;
//второй поиск в глубину
8   Procedure Dfs2(v);
9   begin
10    color[v] := num;
11    for {u:(v, u) ∈ E^T} do
12        if color[u] = 0 then Dfs2(u);
13  end;
//основная программа
//вызов первого поиска в глубину
14  for v := 1 to n do begin
15    color[v] := 0; list[v] := 0;
16  end;
17  num:=n;
18  for v := 1 to n do
19    if color[v] = 0 then Dfs1(v);
//построение транспонированного графа G^T=(V, E^T)
20  E^T:= ∅;
21  for {u:(v, u) ∈ E} do E^T := E^T ∪ (u, v);
//вызов второго поиска в глубину
22  for v := 1 to n do begin
23    color[v] := 0; list[v] := 0;
24  end;
```

```
25  num:=0;
26  for v := 1 to n do
27    if color[list[v]] = 0 then begin
28      inc(num);
29      Dfs2(list[v]);
30    end;
```

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

Построение транспонированного графа G^T занимает $O(V+E)$ времени при хранении графа списками смежности или списком ребер и $O(V^2)$ — при хранении графа матрицей смежности.

В заключение отметим, что все предложенные в статье алгоритмы, основанные на поиске в глубину, имеют оценку сложности $O(V+E)$ при хранении графа списками смежности или списком ребер и $O(V^2)$, соответственно, — при хранении графа матрицей смежности. Оценка времени работы в большинстве из предложенных алгоритмов определяется именно временем работы поиска в глубину.

Литература

1. *Кормен Т.*, *Лейзерсон Ч.*, *Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 2000.
2. *Ахо А.В.*, *Хопкрофт Д.*, *Ульман Дж.Д.* Структуры данных и алгоритмы. М.: Издательский дом «Вильямс», 2003.
3. *Макконнелл Дж.* Анализ алгоритмов. Вводный курс. М.: Техносфера, 2002.
4. *Окулов С.М.* Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004.
5. *Оре О.* Теория графов. М.: Наука, 1968.

Подсчет значения арифметического выражения методом рекурсивного спуска

В. А. Матюхин

На практике часто встречается следующая задача: пользователь вводит арифметическое выражение, например: `(1+2*3)*(4+5)+6*(7+8)+9`, нужно вычислить его значение. Если вы когда-нибудь пробовали писать программу, решающую такую задачу, то, наверное, понимаете, что это не так просто, как кажется с первого взгляда. Если никогда не пробовали, то для того чтобы в полной мере оценить изящность метода, который будет описан в этой статье, попробуйте отложить книгу, сесть за компьютер и попытаться написать такую программу.

Мы рассмотрим красивое решение описанной задачи, использующее *метод рекурсивного спуска*. Фрагменты программы будут приводиться на языке Паскаль, однако думаем, что перевод их на язык Си (равно как и на любой другой) не должен составить большого труда даже для тех, кто с языком Паскаль не знаком.

Сначала рассмотрим более простую задачу: предположим, что все числа у нас — натуральные, а из операций встречаются только сложение и умножение.

Разбиение на лексемы

Для начала давайте напишем процедуру `nextlexem`, которая будет из выражения выделять поочередно все лексемы. *Лексема* — это минимальная единица текста, имеющая самостоятельный смысл. В нашем случае лексемами будут являться знаки арифметических операций, скобки и числа (при этом — обратите внимание! — лексемой является не каждая цифра числа, а число целиком). Эта же процедура будет игнорировать все незначимые символы — пробелы, табуляции и т. д. Еще одна лексема, которая нам понадобится, — конец строки — будет соответствовать тому, что мы хотим выделить из строки следующую лексему, а строка кончилась.

Итак, опишем следующий тип данных:

```
type TLexem = (_Num, _Plus, _Mul, _Open,
               _Close, _End);
```

Можно сделать практически то же самое по-другому:

```
type TLexem = byte;
const _Num = 0;          {Число}
      _Plus = 1;         {Знак сложения}
      _Mul = 2;          {Знак умножения}
```