

Информатика

Медведев Михаил Геннадиевич

*Кандидат физико-математических наук, доцент
факультета кибернетики Киевского национального
университета имени Тараса Шевченко.*



Поиск в глубину на графе

Многие практические задачи описываются как задачи обхода некоторого лабиринта с целью вычисления определённых его свойств. Математическая абстракция, соответствующая лабиринту, называется *графом*. Граф – это множество точек (комнат), соединённых рёбрами (проходами между комнатами).

В этой статье мы рассмотрим один из основных алгоритмов на графе – поиск в глубину, а вместе с ним технику разметки и раскраски вершин.

Представьте себе, что Вы находитесь у входа в лабиринт. Вам известно, что где-то находится выход, и его надо найти. Лабиринт представляет собой набор тоннелей, расположенных глубоко под землёй. У Вас, увы, нет ни фонарика, ни факела, зато есть смекалка и отсутствие страха перед темнотой.

Описанных условий достаточно, чтобы найти выход. Для этого достаточно воспользоваться правилом «правой руки»: на входе в лабиринт следует взяться правой рукой за стену и постоянно двигаться таким образом, чтобы правая рука непрерывно скользила по стене. Действуя подобным образом, Вы обязательно найдёте выход (если, конечно, к нему существует путь из точки входа; в противном случае Вы снова попадёте на вход).

Попробуем усложнить задачу. Пусть лабиринт представляет собой набор подземных комнат и тон-

нелей, их связывающих. Вход в лабиринт только один (он же является и выходом), а в каждой комнате находится слиток золота. Ваша задача состоит в том, чтобы собрать все (неприменно все!) слитки и вернуться к тому же месту, из которого Вы вошли в лабиринт.

Если не вводить дополнительных условий или Ваших умений (способностей), то задача в такой постановке неразрешима. Во всяком случае, детерминированного алгоритма, который бы на 100% гарантировал успех сбора всех золотых слитков, не существует.

Предположим, что в последней версии задачи в каждой комнате есть одна лампочка, которую можно включить, только попав в эту же комнату. Золотоискатель (то есть Вы) уже вооружён факелом, а также мелом, которым может делать любые пометки на стенах лабиринта. Попав в тёмную комнату,

Вы без труда способны найти выключатель, при помощи которого можно зажечь в ней лампу. Будем считать, что любые две соседние комнаты соединяются прямым недлинным тоннелем. То есть находясь с одной стороны тоннеля и заглянув в него, можно однозначно определить, горит ли свет с другой стороны (любой тоннель всегда соединяет две комнаты).

В такой постановке решение задачи существует. Его можно описать двумя правилами:

1. Если из той или иной текущей комнаты существует тоннель, ведущий в ещё не пройденную комнату (где ещё не горит свет), то идём туда. При этом, попадая в тёмную комнату, мы непременно включаем свет и отмечаем мелом тоннель, по которому сюда попали.

2. Если из текущей комнаты нельзя попасть в ещё неизведанную комнату (условия пункта 1 не выполняются), то возвращаемся по тому тоннелю, по которому мы впервые попали сюда (этот тоннель отмечен у нас мелом).

Описанный метод обхода лабиринта носит название «поиск в глубину».

Поиском в глубину (DFS – depth first search) называется один из методов обхода графа $G=(V, E)$, суть которого состоит в том, чтобы идти «вглубь», пока это возможно. В процессе поиска в глубину вершинам графа присваиваются номера, а рёбра помечаются. Обход вершин

графа происходит согласно принципу: если из текущей вершины есть рёбра, ведущие в непройденные вершины, то идём туда, иначе возвращаемся назад.

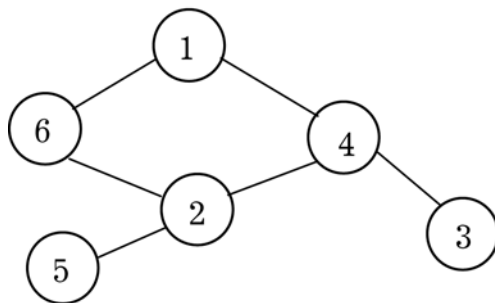
Поиск в глубину начинается с выбора начальной вершины v графа G , которая сразу же помечается как пройденная. Потом для каждой непройденной вершины, смежной с v , рекурсивно вызывается поиск в глубину. Когда все вершины, достижимые из v , будут помечены, поиск заканчивается. Если на некотором (не начальном) шаге обхода поиск закончился, но некоторые вершины остаются непопомеченными (такой случай возможен в случае ориентированного или несвязного графа), то выбирается произвольная из них и поиск повторяется. Процесс поиска продолжается до тех пор, пока все вершины графа G не будут помечены.

Пример 1. Вывести последовательность вершин графа при поиске в глубину.

Вход. Первая строка содержит количество вершин n в неориентированном связном графе. Каждая из следующих строк содержит пару вершин, соединённых ребром графа. Вершины нумеруются с 1 до n .

Выход. Последовательность вершин, посещаемых при поиске в глубину. Первой посещается вершина с номером 1, как показано в примере.

Пример входа	Пример выхода
6	Vertex 1 is visited
1 4	Vertex 4 is visited
1 6	Vertex 2 is visited
2 4	Vertex 5 is visited
2 5	Vertex 6 is visited
2 6	Vertex 3 is visited
3 4	



Пусть m – матрица смежности графа ($m[i][j] = 1$, если существует ребро между вершинами i и j , и $m[i][j] = 0$, если иначе), $used$ –

булевый массив, в котором $used[i] = 1$, если вершина i помечена (пройдена) и $used[i] = 0$, если иначе.

```
#include <stdio.h>
#include <memory.h>
#define MAX 100

int m[MAX][MAX], used[MAX];
int i, n, a, b, ptr;

void dfs(int v)
{
    int i;
    // включаем в комнату (вершине v графа) лампочку
    used[v] = 1;
    printf("Vertex %d is visited\n", v);
    // ищем тоннель, через который можно попасть в ещё
    // непройденную комнату
    for (i = 1; i <= n; i++)
        if (m[v][i] && !used[i]) dfs(i);
}

void main(void)
{
    memset(m, 0, sizeof(m));
    memset(used, 0, sizeof(used));
    // читаем входные данные
    scanf("%d", &n);
    while (scanf("%d %d", &a, &b) == 2)
        m[a][b] = m[b][a] = 1;
    // запускаем поиск в глубину с вершины 1
    dfs(1);
}
```

При поиске в глубину можно использовать *цвет вершины*. Изначально цвета всех вершин *белые*. Когда вершина проходится первый раз, она становится *серой*. Вершина является *обработанной*, если полностью просмотрены все её смежные вершины. Когда вершина обработана, она становится *чёрной*.

Каждой вершине можно приписать две *метки времени*:

$d[v]$ – время, когда вершина обнаружена и стала *серой*;

$f[v]$ – время, когда вершина

обработана и стала *чёрной*.

Эти метки используются во многих алгоритмах на графах и являются полезными для анализа свойств поиска в глубину.

Метки времени $d[v]$ и $f[v]$ являются целыми числами от 1 до $2|V|$. Для каждой вершины v имеет место неравенство $d[v] < f[v]$. Вершина v будет белой до момента времени $d[v]$, серой с $d[v]$ до $f[v]$ и чёрной после $f[v]$.

Поиск в глубину на связном неориентированном графе можно описать следующим образом:

DFS(*u*)

1. Отмечаем вершину *u* пройденной и окрашиваем её в серый цвет.

2. Для каждой вершины *v*, смежной с *u* и окрашенной в белый цвет, вызываем dfs(*v*).

3. Окрашиваем вершину *u* в чёрный цвет.

Пример 2. Расстановка меток вершин графа при поиске в глубину.

Вход. Первая строка содержит количество вершин *n* в связном неориентированном графе. Каждая из следующих строк содержит пару вершин, соединённых ребром графа. Вершины графа нумеруются с 1 до *n*.

Выход. В *i*-ой строке вывести информацию об *i*-ой вершине в формате, приведённом в примере. Вместе с номером вершины *i* сле-

дует вывести значения *d*[*i*] и *f*[*i*].

Цвет вершины храним в ячейках массива *used*: 0 – вершина белая, 1 – серая, 2 – чёрная.

Пример входа	Пример выхода
6	Vertex: 1, Gray: 1, Black 12
1 4	Vertex: 2, Gray: 3, Black 8
1 6	Vertex: 3, Gray: 9, Black 10
2 4	Vertex: 4, Gray: 2, Black 11
2 5	Vertex: 5, Gray: 4, Black 5
2 6	Vertex: 6, Gray: 6, Black 7
3 4	

```
#include <stdio.h>
#include <memory.h>
#define MAX 100

int m[MAX][MAX], d[MAX], f[MAX], used[MAX];
int i, n, a, b, time;

void dfs(int v)
{
    int i;
    used[v] = 1;
    d[v] = time++;
    for (i = 1; i <= n; i++)
        if (m[v][i] && !used[i]) dfs(i);
    used[v] = 2;
    f[v] = time++;
}

void main(void)
{
    memset(m, 0, sizeof(m)); memset(used, 0, sizeof(used));
    scanf("%d", &n);
    while (scanf("%d%d", &a, &b) == 2)
        m[a][b] = m[b][a] = 1;
    time = 1; dfs(1);

    for (i = 1; i <= n; i++)
        printf("Vertex: %d, Gray: %d, Black %d\n", i, d[i], f[i]);
}
```

Реализуем процедуру поиска в глубину на случай несвязного или ориентированного графа. В этом

```
for (i = 1; i <= n; i++)
    if (!used[i]) dfs(i);
```

Пример 3. Рассмотрим ориентированный граф, изображённый на рисунке 1. На рисунке 2 показан обход графа в глубину с расстановкой меток, начиная с вершины 4. Около

каждой вершины v запишем её номер в последовательности обхода в глубину, а в фигурных скобках – метки времени $d[v]$ и $f[v]$.

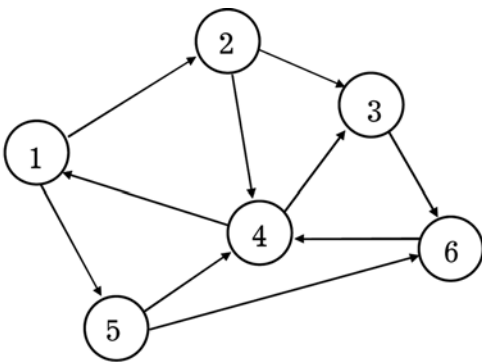


Рис. 1. Ориентированный граф

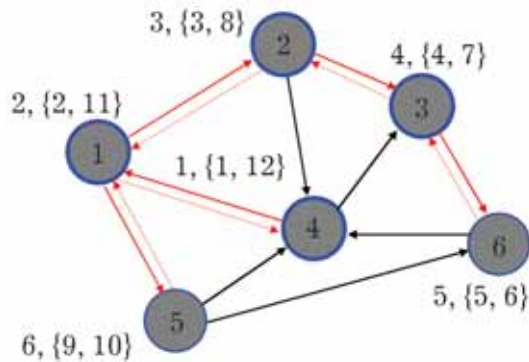


Рис. 2. Поиск в глубину

В результате обхода графа в глубину получаем *подграф предшествования*, который представляет собой лес поиска в глубину (состоит из деревьев поиска в глубину). На рис. 3 представлен соответствующий граф предшествования.

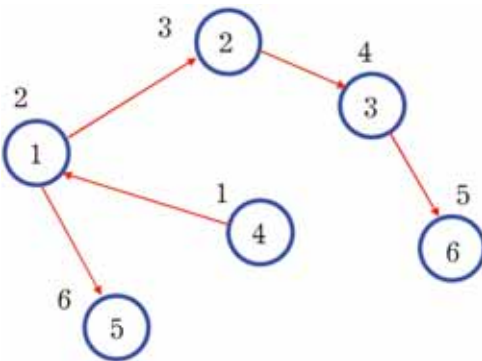


Рис. 3. Подграф предшествования



Классификация рёбер

Поиск в глубину используется для классификации рёбер графа. Имеются 4 типа рёбер.



1. *Дугами дерева* являются рёбра, ведущие к вершинам, которые раньше не посещались. Они формируют для заданного графа *глубинный остоновый лес*.

2. *Обратными* называются дуги, идущие в остовном лесу от потомков к предкам. Дуга, идущая из вершины в саму себя, считается *обратной*.

3. *Прямыми* называются дуги, идущие от предков к собственным потомкам, но которые не являются дугами дерева.

4. *Поперечными* дугами являются рёбра, соединяющие вершины, не являющиеся ни предками, ни потомками.

На рисунке 4 изображены:

- дуги дерева – красным цветом;
- обратные дуги – зелёным цветом;
- прямые дуги – розовым цветом;
- поперечные дуги – коричневым цветом.

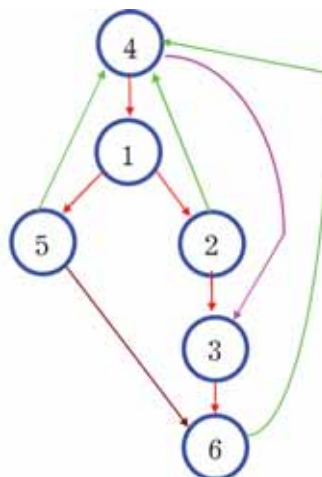


Рис. 4. Глубинный остоновый лес

Свойства поиска в глубину

Время начала и окончания обработки вершины при поиске в глубину образуют правильную скобочную структуру, если появление вершины u обозначать через « (u) », а окончание обработки вершины u обозначать через « $)$ ». Поиск в глубину на графе, изображённом на рисунке 2, образует следующую скобочную структуру:

(4 (1 (2 (3 (6 6) 3) 2) (5 5) 1) 4)

Теорема о скобочной структуре.
При поиске в глубину для про-

извольных двух вершин u и v выполняется одно и только одно из следующих свойств:

1. Отрезки $(d[u], f[u])$ и $(d[v], f[v])$ не пересекаются.

2. Отрезок $(d[u], f[u])$ полностью содержится в отрезке $(d[v], f[v])$. При этом u является потомком v при поиске в глубину.

3. Отрезок $(d[v], f[v])$ полностью содержится в отрезке $(d[u], f[u])$. При этом v является потомком u при поиске в глубину.

Следствие. Вложение интервалов для потомков.

Вершина v является потомком вершины u при поиске в глубину тогда и только тогда, когда

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

Теорема про белый путь. Вершина v является потомком вершины u в лесу поиска в глубину тогда и только тогда, когда в момент $d[u]$ обнаружения вершины u существует путь из u в v , состоящий только из белых вершин.

Теорема про свойства рёбер. Ребро (u, v) является:

а) ребром дерева или прямым

ребром тогда и только тогда, когда

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

б) обратным ребром тогда и только тогда, когда

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

в) поперечным ребром тогда и только тогда, когда

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

Теорема о классификации рёбер в неориентированном графе. При поиске в глубину в неориентированном графе произвольное ребро является или ребром дерева, или обратным. Прямых и перекрёстных рёбер не существует.

Задачи

В качестве приложения поиска в глубину рассмотрим три эквивалентные между собой задачи на неориентированном связном графе.

1. Можно ли покрасить вершины графа в два цвета так, чтобы любое его ребро соединяло вершины разных цветов?

2. Проверить, является ли граф двудольным. Напомним, что граф $G(V, E)$ является двудольным, если множество его вершин можно разбить на два подмножества $V = V_1 \cup V_2$ так, что любое ребро графа соединяет вершины из разных подмножеств.

3. Проверить, существует ли в графе цикл нечётной длины, то есть цикл, состоящий из нечётного количества вершин.

Эквивалентность задач 1 и 2. Пусть вершины графа можно покрасить двумя цветами. Тогда все вершины с одним цветом отнесём к одной доле, а вершины с другим цветом – к другой. Если ребро соединяет две вершины разного цвета, то это же ребро соединяет вершины из разных долей. Двудольный граф построен.

Эквивалентность задач 2 и 3.

Если граф двудольный, то любой его цикл содержит чётное количество вершин. Если граф не двудольный, то он обязательно содержит цикл нечётной длины.

Рассмотрим первую задачу более детально, приведя полностью её условие, описание ввода-вывода и детальное решение на языке Си. Эта задача взята с WEB страницы <http://uva.onlinejudge.org> университета Вальядолид, посвящённой олимпиадному программированию, её номер 10004 в списке задач.

Пример 4. Раскраска вершин графа двумя цветами [Вальядолид, 10004]. Имеется связный неориентированный граф. Определить, можно ли раскрасить его вершины двумя цветами так, чтобы никакие две соседние вершины не были покрашены одним цветом.

Вход. Состоит из нескольких тестов. Первая строка каждого теста содержит количество вершин n ($1 < n < 200$) и количество рёбер l в графе. Каждая из следующих l строк содержит два числа – номера вершин графа, соединённых ребром. Вершины графа нумеруются числами от 0 до $n-1$. Последняя строка содержит $n=l=0$ и не обрабатывается.

Выход. Для каждого теста вывести строку «BICOLORABLE.», если вершины графа можно раскрасить требуемым образом, и «NOT BICOLORABLE.», если иначе.

Решение. Для решения задачи воспользуемся поиском в глубину. Изначально вершины не просмотрены, пометим их все цветом 0. По мере прохождения по вершинам будем красить их в два цвета: 1 и 2. Если текущая вершина имеет цвет i ($i=1, 2$), то следующую вершину, в которую мы попадаем при поиске в глубину, красим в цвет $3-i$. При этом проверяем, чтобы две соседние вершины не были покрашены в одинаковый цвет.

Реализация. Матрицу смежности

графа храним в массиве g размера $MAX = 200$. Массив $mark$ содержит информацию о цвете вершины. Глобальная переменная $Error$ отвечает

Пример входа	Пример выхода
3 3	NOT BICOLORABLE.
0 1	BICOLORABLE.
1 2	
2 0	
4 3	
0 1	
1 2	
2 3	
0 0	

за правильность раскраски: как только найдутся две соседние вершины, покрашенные одинаково, переменная примет значение 1.

```
#define MAX 200
int g[MAX][MAX], mark[MAX], Error;
```

Процедура dfs выполняет поиск в глубину. Ей передаются два параметра: текущая вершина v и её цвет $color$. Если уже встретился случай невозможности требуемой раскраски ($Error = 1$), то выйти из процедуры. Помечаем вершину v цветом $color$. Ищем непросмотрен-

ную вершину i , в которую можно пойти, продолжив поиск в глубину. При этом если соседняя вершина i уже была просмотрена, проверяем условие окрашенности вершин v и i в разные цвета. Если указанные вершины покрашены в одинаковый цвет, устанавливаем $Error = 1$.

```
void dfs(int v, int color)
{
    int i;
    if (Error) return;
    mark[v] = color;
    for (i = 0; i < n; i++)
        if (g[v][i])
            if (!mark[i]) dfs(i, 3-color);
            else if (mark[v] == mark[i]) Error = 1;
}
```

Основной цикл программы. Читаем число вершин графа n и

количество рёбер l . Обнуляем матрицу смежности и массив $mark$.

```
while (scanf("%d %d", &n, &l), n)
{
    memset(g, 0, sizeof(g));
    memset(mark, 0, sizeof(mark));
```

Читаем данные графа.

```
for (i = 0; i < 1; i++)  
{  
    scanf("%d %d", &a, &b);  
    g[a][b] = g[b][a] = 1;  
}
```

Обнуляем значение переменной *Error*, запускаем процедуру поиска

в глубину с нулевой вершины, которую красим цветом 1.

```
Error = 0; dfs(0,1);
```

В зависимости от значения переменной *Error* выводим результат.

```
if (Error) printf("NOT BICOLORABLE.\n");  
else printf("BICOLORABLE.\n");  
}
```

Заключение

Алгоритм обхода графа в глубину является базовой техникой решения многих сложных прикладных задач. Проверить своё умение применять эту технику можно на сайте олимпиадных задач <http://acm.mipt.ru/judge> (см. задачи 012 «Корректный словарь», 013 «Коробки», 053 «Любовный многоугольник», 092 «Пожарные станции», 098 «Минимальное дополнение до сильно связного графа»). Вот пример одной задачи с этого сайта.

Задача «Корректный словарь». Корректный словарь – это такой словарь, в котором сложные понятия определяются через простые, те в свою очередь через более простые и так далее вплоть до неопределяемых аксиоматических понятий. В корректном словаре не должно быть

«циклических определений» и для одного термина должно быть только одно определение. Нужно написать программу, которая умеет определять, корректный словарь или нет.

Вход. Первая строка содержит натуральное число $D < 100$ – количество словарных статей (определений). Затем вводятся D строчек с описанием этих определений. Каждое описание начинается с определяемого термина (слова без пробелов), затем идёт число терминов (меньше 20), которые используются в определении этого термина, и затем идут используемые в определении термины, разделённые пробелами. Все термины состоят менее чем из 20 букв.

Выход. Выход должен содержать либо слово «CORRECT», либо «NOT CORRECT».

Список литературы

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. – Москва, Санкт-Петербург, Киев, 2005. – 1292 с.
2. Винокуров Н.А., Ворожцов А.В. Практика и теория программирования, Кн. 2. – М.: Физматкнига, 2008. – 288 с.