Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

УТВЕРЖДАЮ
Зав.кафедрой,
к. ф.-м. н.
_____ С. В. Миронов

ОТЧЕТ О ПРАКТИКЕ

Студентки 2 курса 251 группы факультета КНиИТ
Кожиной Ольги Олеговны

вид практики: педагогическая
кафедра: математической кибернетики и компьютерных наук
курс: 2
семестр: 4
продолжительность: 2 нед., с 12.07.2017 г. по 21.07.2017 г.

Руководитель практики от университета,

доцент, к. ф.-м. н. _____ Ю. Н. Кондратова

Руководитель практики от организации (учреждения, предприятия),

руководитель ЦОПП _____ М. Р. Мирзаянов

Тема практики:«Обход графа в глубину»

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью практики является освоение алгоритма поиска в глубину и его приложений. В результате прохождения практики были получены навыки решения типовых задач:

- Поиск компонент связанности;
- Поиск точек сочленения и мостов;
- Определение отношения вложенности между вершинами;
- Топологическая сортировка;
- Ориентация графа;
- Проверка графа на ацикличность.

1 Теоретические сведения

Обход — алгоритм, который посещает каждую вершину по одному разу в некотором порядке.

Обход графа в ширину можно сравнить с распространением огня, а обход в глубину — с человеком, который ходит по лабиринту, пытаясь найти выход.

Поиск в глубину (DFS — *Depth First Search*) оперирует тремя цветами: белым, серым и черным. Изначально вершины являются белыми (непосещенными), а в процессе обхода они темнеют. Вершина становится серой, когда обход впервые обнаружил эту вершину. Вершина становится черной, когда обход завершил окончательно обработку этой вершины и покинул ее навсегда.

1.1 Стратегия поиска в глубину

Стратегия поиска в глубину, как следует из её названия состоит в том, чтобы идти «в глубь» графа, насколько это возможно, т. е. находясь в текущей (серой) вершине рассматриваем список «соседей» и рекурсивно переходим в белую вершину, делая её серой. Когда просмотр списка «соседей» завершен, следовательно, нет соседних белых вершин, делаем текущую вершину черной и покидаем её, возвращаясь по стеку рекурсии вверх. Наглядно эта стратегия продемонстрирована на рисунке 1.

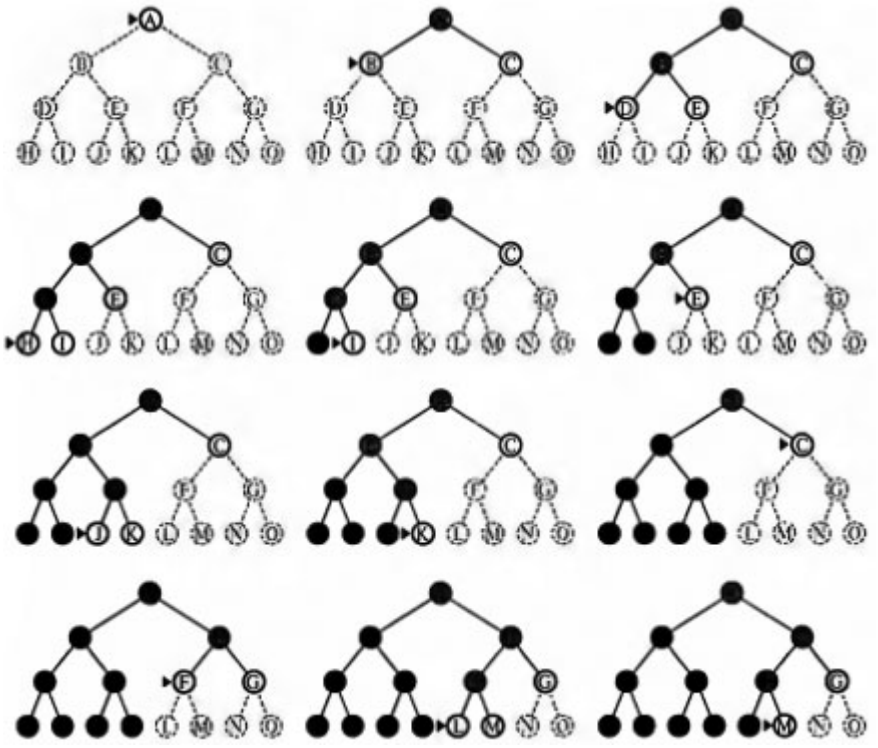

Рисунок 1 – Пример обхода графа в глубину

При хранении графа списками смежности обход в глубину, запущенный из вершины графа, посещает все вершины, достижимые из данной, и строит дерево поиска в глубину. В этом случае временная сложность алгоритма составит $O(x + y)$, где x — количество вершин, достижимых из заданной, а y — количество ребер (дуг), исходящих из x посещенных вершин, или $O(n+m)$, если посещен весь граф, n — общее количество вершин, а m — общее количество ребер.

Ниже представлен псевдокод DFS, где u — вершина, из которой осуществляется обход в глубину, $pred$ — вершина-предок для u (если DFS первоначально запускается из неё, то она является предком для самой себя), p — массив предков, $color$ — массив цветов.

```
dfs(u, pred)
        color[u] = GRAY
        p[u] = pred
        for (v : g[u]) //v - сосед u
                if (color[v] == WHITE)
                        dfs(v, u)
        color[u] = BLACK
```

Удобно инициализировать массив предков $p[i] = -1$, т. е. если $p[i] = -1$, то предок не определен.

Таким образом, если корень дерева располагается в узле 1, то вызов будет представлять собой команду $dfs(1, 1)$.

1.2 Различия между BFS и DFS

Оба эти обхода работают за $O(n + m)$ и просты в реализации, однако:

1. В отличие от обхода в ширину, дерево обхода в глубину не кодирует информацию о кратчайших путях, а содержит информацию о произвольных путях из корня.
2. Обход в глубину в большой степени использует и анализирует циклическую структуру графа, что делает его фундаментом для более сложных алгоритмов анализа циклов, мостов, точек сочленения, компонент двусвязности и т. д.

1.3 Вариации написания DFS

Минимальная версия DFS записывается довольно коротко, что является причиной её частого использования. Такая реализация достаточна для выде-

ления компонент связности, поэтому нет необходимости для использования массива предков и трех цветов. Ниже представлен псевдокод минимальной версии DFS.

```
dfs(u)
        used[u] = true
        for (v : g[u])
                if (!used[v])
                        dfs(v)
```

Полная версия DFS для каждой вершины записывает две метки времени: момент входа в вершину (вершина меняет цвет с белого на серый) и момент выхода из неё (вершина меняет цвет с серого на черный). Все моменты времени — различные последовательные числа, поэтому для поддержания таймера времени потребуется глобальная переменная $T = 0$. Здесь tin — массив времен входов, $tout$ — массив времен выходов. Ниже представлен псевдокод полной версии DFS, а также иллюстрации дерева с временными метками (рис. 2) и временных отрезков этого дерева (рис. 3).

```
T = 0
dfs(u, pred)
        color[u] = GRAY
        p[u] = pred
        tin[u] = T++
        for (v : g[u])
                if (color[v] == WHITE)
                        dfs(v, u)
        color[u] = BLACK
        tout[u] = T++
```

Временные метки кодируют информацию об иерархии для дерева поиска в глубину. Любая пара отрезков времен ($[tin[i], tout[i]]$) вершин либо не пересекается (отрезки идут один за другим), либо один отрезок вложен в другой. Если отрезок времен вершины v вложен в отрезок времен вершины u, то u — предок v в дереве поиска в глубину. Если отрезки не пересекаются, то никакая из двух вершин не является предком другой и эти вершины лежат в разных поддеревьях.

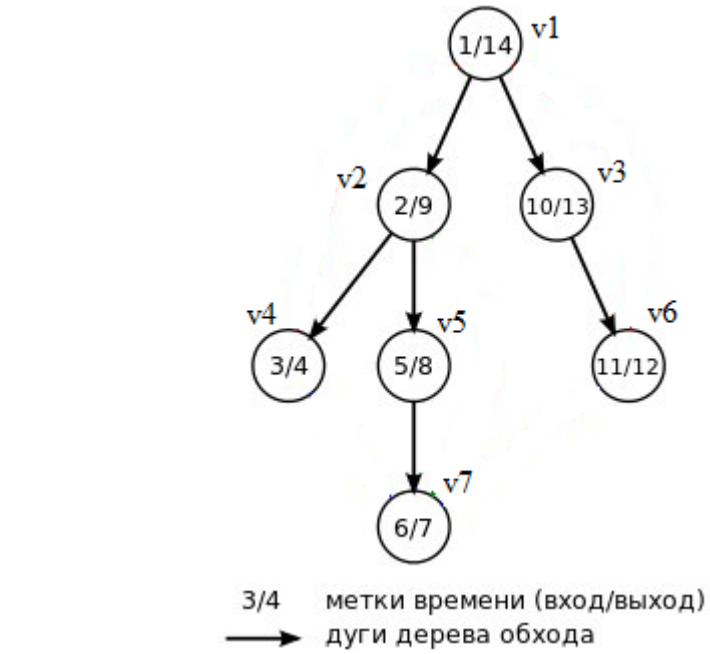

Рисунок 2 – Дерево поиска в глубину с временными метками

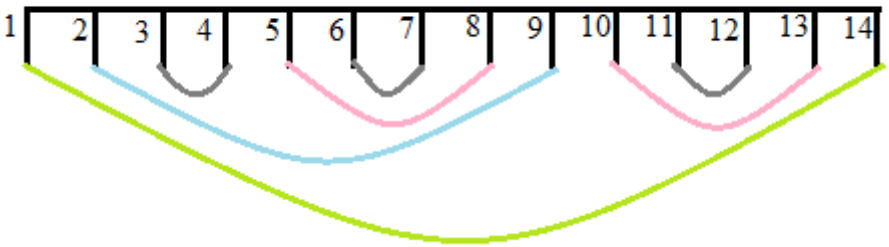

Рисунок 3 – Временные отрезки дерева поиска

1.4 Классификация ребер при DFS

Поиск в глубину классифицирует ребра на основании их участия в DFS при первом просмотре этого ребра. Ребра бывают:

1. Древесные — ребра, образующие дерево поиска в глубину (ведут из серой вершины в белую). Пример: ребро $0 \to 1$.
2. Обратные — ребра, которые ведут вверх по дереву, замыкают цикл (ведут из серой вершины в серую). Пример: ребро $3 \to 1$.
3. Прямые — ребра, которые ведут вниз по дереву (ведут из серой вершины в черную). Пример: ребро $0 \to 2$.
4. Перекрестные — ребра, которые ведут от одного поддерева к другому (ведут из серой вершины в черную). Пример: ребро $4 \to 3$ (рисунок 4).

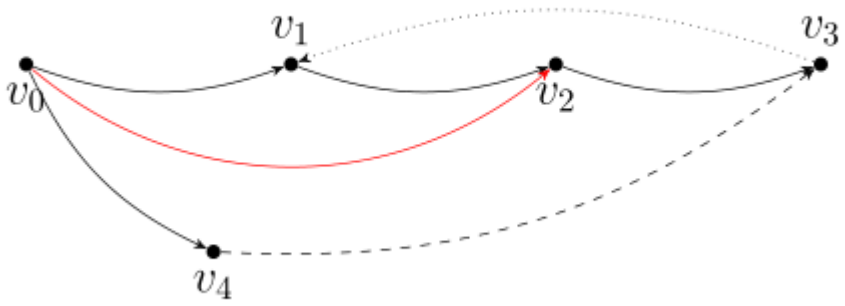

Рисунок 4 – Граф со всеми видами ребер

Различить два последних класса ребер можно легко на основании временных меток: если отрезки времен не вложены, значит, ребро перекрестное.

Замечание: в ориентированном графе есть цикл тогда и только тогда, когда существует обратное ребро.

DFS для неориентированного графа делит все ребра на 2 группы: древесные и обратные.

1.5 Серия DFS

Не сбрасывая цвета вершины, пройдем по всем вершинам вызывая DFS из каждой белой в текущий момент времени вершины. Ниже представлена реализация серии DFS, где V — список всех вершин.

```
1 color[] <- WHITE
2 for (u : V)
3         if (color[u] == WHITE)
4                 dfs(u, u)
```

Так все вершины графа будут посещены, а также будет построен набор деревьев поиска в глубину, покрывающих все вершины графа. Суммарное время работы — $O(n + m)$. Серия поисков в глубину применяется для выделения компонент связанности графа.

1.6 Топологическая сортировка

Топологическая сортировка ориентированного ациклического графа представляет собой такое линейное упорядочивание всех его вершин, что если граф G содержит ребро (u, v), то u при таком упорядочивании располагается до v (если граф содержит цикл, такая сортировка невозможна). Топологическую

сортировку графа можно рассматривать как такое упорядочивание его вершин вдоль горизонтальной линии, что все ребра направлены слева направо.

Ориентированные ациклические графы используются во многих приложениях для указания последовательности событий. На рисунке 5 приведен пример графа, построенного профессором Рассеянным для утреннего одевания. Ребро (u, v) показывает, что ведь u должна быть одета раньше вещи v. Топологическая сортировка (рисунок 6) этого графа дает нам порядок одевания [1].

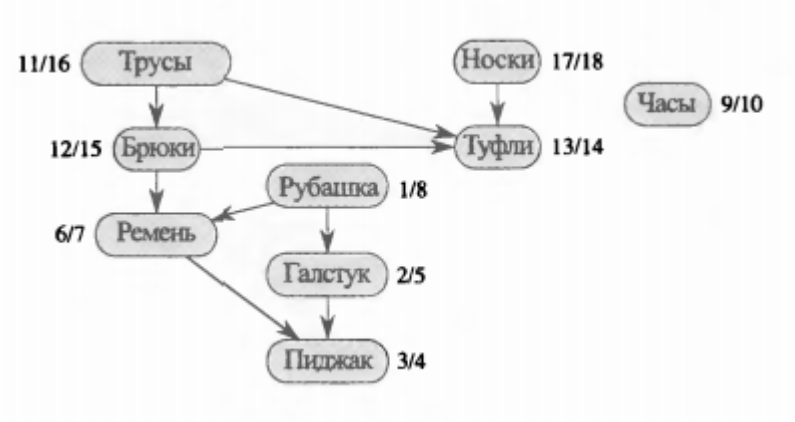

Рисунок 5 – Граф утреннего одевания

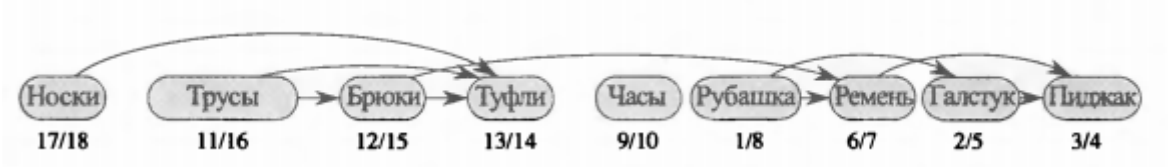

Рисунок 6 – Топологическая сортировка графа утреннего одевания

Алгоритм топологической сортировки будет следующий:

1. Произведем серию DFS. Если в процессе DFS была попытка пойти из серой вершины в серую, то топологической сортировки не существует.
2. Отсортируем все вершины по убыванию времени окончания обработки. Найденный порядок и есть топологическая сортировка.

Замечание: добавляя вершину в некоторый вектор в момент времени окончания её обработки и перевернув в конце этот вектор, можно легко получить искомую последовательность.

2 Примеры задач и их решения

2.1 Компоненты связности

Условие.

ограничение по времени на тест — 0.5 секунд

ограничение по памяти на тест — 256 мегабайт

ввод — стандартный ввод

вывод — стандартный вывод

Вам задан неориентированный граф с n вершинами и m ребрами. В графе отсутствуют петли и кратные ребра. Определите компоненты связности заданного графа.

Входные данные. Граф задан во входном файле следующим образом: первая строка содержит числа n и $m\,(1 \leqslant n \leqslant 1000)$. Каждая из следующих m строк содержит описание ребра — два целых числа из диапазона от 1 до n — номера концов ребра.

Выходные данные. На первой строке выходного файла выведите число l — количество компонент связности заданного графа. На следующей строке выведите n чисел из диапазона от 1 до l — номера компонент связности, которым принадлежат соответствующие вершины. Компоненты связности следует занумеровать от 1 до l произвольным образом.

Решение. Заинициализируем переменную $nt = 0$. Во вспомогательном векторе `used` будем хранить номер компоненты связности, к которой она принадлежит. Произведем серию обходов в глубину из каждой непомеченной вершины, увеличивая каждый раз в цикле серии значение счетчика cnt и передавая его в функцию $dfs(i, cnt)$, чтобы все узлы компоненты с корнем в i принадлежали компоненте с номером cnt. Выведем на экран вектор `used` — это и будет ответом.

Полный код программы приведен в приложении A.

2.2 Обход в глубину

Условие.

ограничение по времени на тест — 0.5 секунд

ограничение по памяти на тест — 256 мегабайт

ввод — стандартный ввод

вывод — стандартный вывод

Задан неориентированный граф из n вершин. Требуется найти путь из вершины v_1 в вершину v_2, при этом обязательно использовать обход в глубину. Обход в глубину надо реализовать таким образом, что для очередной вершины графа в обходе надо просматривать все смежные с ней вершины в порядке увеличения их номера. То есть требуется реализовать «обычный» обход в глубину.

Входные данные. В первой строке через пробел записано три натуральных числа n, v_1, v_2 $(1 \leqslant n \leqslant 100, 1 \leqslant v_1, v_2 \leqslant n)$.

В каждой из следующих n строк записано по n чисел — матрица смежности графа. «1» обозначает, что между соответствующими вершинами есть ребро, «0» — что нет.

Выходные данные. В первую строку выведите длину пути (количество ребер графа в пути). Во вторую строку выведите сам путь начиная с вершины v_1 и заканчивая вершиной v_2.

Выведите единственную строку -1, если пути нет.

Решение. Считаем матрицу смежности, заведем вектор предков `p` и вектор `used` для помечивания вершин. Запустим рекурсивную функцию `dfs`, принимающую два аргумента: стартовую вершину u и вершину-предок pr. Пройдемся в цикле по смежным с u вершинам, если они не были помечены, вызовем `dfs` от них. Выведем ответ, воспользовавшись вектором предков `p`. Если последний добавленный в вектор ответа `ans` элемент равен -1, то выводим -1, иначе выводим размер вектора `ans` и его самого в обратном порядке.

Полный код программы приведен в приложении Б.

2.3 Точки сочленения

Условие.

ограничение по времени на тест — 0.5 секунд

ограничение по памяти на тест — 256 мегабайт

ввод — стандартный ввод

вывод — стандартный вывод

Вам задан неориентированный граф с n вершинами и m ребрами. В графе отсутствуют петли и кратные ребра. Известно, что граф связный.

Найдите все точки сочленения в заданном графе, то есть такие вершины, удаление которых ведет к увеличению числа компонент связности.

Входные данные. Граф задан во входном файле следующим образом: первая строка содержит числа n и $m\,(1 \leqslant n \leqslant 200)$. Каждая из следующих m строк содержит описание ребра — два целых числа из диапазона от 1 до n — номера концов ребра.

Выходные данные. В первую строку выведите число c — количество точек сочленения в заданном графе.

В следующую строку выведите c целых чисел — номера вершин, которые являются точками сочленения, в возрастающем порядке.

Решение. Воспользуемся следующим наивным способом нахождения точек сочленения:

1. Посчитаем общее количество компонент связности серией DFS;
2. Запретим последовательно использовать каждую из вершин (например, пометив её черным цветом), а затем также для каждого из n вариантов запрета серией DFS найдем точки сочленения (если число компонент связности увеличилось при запрете использования вершины i, то вершина i есть точка сочленения).

Однако в данной задаче можно опустить первый пункт (граф связный по условию).

Полный код программы приведен в приложении B.

2.4 Проверка на двудольность

Условие.

ограничение по времени на тест — 2 секунды
ограничение по памяти на тест — 256 мегабайт
ввод — стандартный ввод
вывод — стандартный вывод

Дан неориентированный граф. Необходимо проверить, является ли он двудольным, т. е. можно ли разбить его вершины на два множества так, чтобы никакие две вершины внутри каждого множества не были смежны.

Входные данные. Граф задан во входном файле следующим образом: первая строка содержит числа n и $m\,(1 \leqslant n \leqslant 200)$. Каждая из следующих m строк содержит описание ребра — два целых числа из диапазона от 1 до n — номера концов ребра.

В первой строке содержится натуральное число $T\,(1 \leqslant T \leqslant 10000)$ — количество наборов тестовых данных. Во первой строке каждого набора за-

писано два целых числа N и M ($1 \leqslant N \leqslant 100000, 0 \leqslant M \leqslant 100000$) — число вершин и ребер в графе соответственно.

В последующих M строках записано по паре номеров вершин, соединенных ребром. Номера вершин — это целые числа от 1 до N. Гарантируется, что в графе не содержится петель и мультиребер. Известно, что сумма значений N по всем T тестовым наборам не превосходит 100000, а сумма значений M не превосходит 200000.

Выходные данные. Выведите T строк, по одной для каждого набора входных данных. Если граф двудольный, выводите «YES», иначе «NO» (без кавычек).

Решение. Произведем серию поисков в глубину, используя вектор цветов `color` и вспомогательный параметр c, принимающий значение 0 или 1. При заходе в функцию инициализируем $color[u] = c$ и сменим значение, сложив c по модулю 2 с 1. Рассмотрим всех соседей вершины: если хотя бы один из них имеет такой же цвет, как у данной вершины, то ответ «NO». Заметим, что граф двудольный тогда и только тогда, когда все циклы в нем четной длины.

Полный код программы приведен в приложении Г.

ЗАКЛЮЧЕНИЕ

В ходе практики были изучены свойства алгоритма поиска в глубину, а также его преимущества по сравнению с алгоритмом обхода в ширину. Практические задачи были решены при помощи различных модификаций алгоритма DFS, причем основными являются:

- использование булевого массива `used` (поиск компонент связанности графа);
- использование цветов для пометки вершин (проверка графа на ацикличность);
- использование массивов для хранения меток времени (определение вложенности вершин).

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Кормен, Т. Х.* Алгоритмы: построение и анализ, 3-е издание / Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, К. Штайн. — Москва: Вильямс, 2013.

ПРИЛОЖЕНИЕ А

Программный код задачи «Компоненты связности»

```
#include <iostream>
#include <map>
#include <string>
#include <fstream>
#include <stdio.h>
#include <set>
#include <vector>
#include <algorithm>
#include <queue>

using namespace std;

#define pb push_back

vector<int> used;
vector<vector<int>> g;
void dfs(int u, int cnt)
{
        used[u] = cnt;
        for(int i = 0; i < g[u].size(); i++)
        {
                if(used[g[u][i]] == 0)
                dfs(g[u][i], cnt);
        }
}

int main()
{
#ifdef _DEBUG
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
#endif

        int n, m; cin >> n >> m;
        g = vector<vector<int>> (n);
        used = vector<int> (n, 0);
        for(int i = 0; i < m; i++)
        {
                int x, y; cin >> x >> y;
                x--, y--;
                g[x].pb(y);
                g[y].pb(x);
        }

        int cnt = 0;
```

```
46          for(int i = 0; i < n; i++)
47          {
48                  if(used[i] == 0)
49                  {
50                          cnt++;
51                          dfs(i, cnt);
52                  }
53          }
54
55          cout << cnt << endl;
56          for(int i = 0; i < n; i++)
57                  cout << used[i] << " " ;
58          return 0;
59  }
```

ПРИЛОЖЕНИЕ Б

Программный код задачи «Обход в глубину»

```
1   #include <iostream>
2   #include <map>
3   #include <string>
4   #include <fstream>
5   #include <stdio.h>
6   #include <set>
7   #include <vector>
8   #include <algorithm>
9   #include <queue>
10
11  using namespace std;
12
13  #define ll long long
14  #define pb push_back
15  #define mp make_pair
16
17  int n, v1, v2;
18  vector<int> used;
19  vector<vector<int>> g;
20  vector<int> p;
21  void dfs(int u, int pr)
22  {
23          used[u] = 1;
24          p[u] = pr;
25          for(int i = 0; i < n; i++)
26          {
27                  if(used[i] == 0 && g[u][i] == 1)
28                          dfs(i, u);
29          }
30          used[u] = 2;
31  }
```

```
32
33  int main()
34  {
35  #ifdef _DEBUG
36      freopen("in.txt", "r", stdin);
37      freopen("out.txt", "w", stdout);
38  #endif
39
40          cin >> n >> v1 >> v2;
41          g = vector<vector<int>> (n, vector<int>(n));
42          used = vector<int> (n, 0);
43          p = vector<int> (n, -1);
44          for(int i = 0; i < n; i++)
45          {
46                  for(int j = 0; j < n; j++)
47                  {
48                          cin >> g[i][j];
49                  }
50          }
51          v1--, v2--;
52          dfs(v1, v1);
53          vector<int> ans;
54          while(true)
55          {
56                  ans.pb(v2);
57                  if(v2 == v1 || v2 == -1)
58                          break;
59                  v2 = p[v2];
60          }
61
62          if(ans.size() > 0 && ans.back() == -1)
63                  cout << -1 << endl;
64          else
65          {
66                  cout << ans.size() - 1 << endl;
67                  for(int i = ans.size() - 1; i >= 0; i--)
68                          cout << ans[i] +1 << " ";
69          }
70          return 0;
71  }
```

ПРИЛОЖЕНИЕ В

Программный код задачи «Точки сочленения»

```
1  #include <iostream>
2  #include <map>
3  #include <string>
4  #include <fstream>
5  #include <stdio.h>
```

```cpp
#include <set>
#include <vector>
#include <algorithm>
#include <queue>

using namespace std;

#define ll long long
#define pb push_back
#define mp make_pair

vector<int> used;
vector<vector<int>> g;
void dfs(int u, int j)
{

	used[u] = 1;

	for(int i = 0; i < g[u].size(); i++)
	{
		if(used[g[u][i]] == 0 && g[u][i] != j)
		{
			dfs(g[u][i], j);
		}

	}
	used[u] = 2;
}

int main()
{
#ifdef _DEBUG
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
#endif

	int t, n, m;

	cin >> n >> m;
	g = vector<vector<int>> (n);
	used = vector<int> (n, 0);
	for(int i = 0; i < m; i++)
	{
		int x, y;
		cin >> x >> y;
		x--, y--;
		g[x].pb(y);
		g[y].pb(x);
```

```
54          }
55
56          vector<int> ans;
57          for(int j = 0; j < n; j++)
58          {
59                  used = vector<int> (n, 0);
60                  int c = 0;
61                  for(int i = 0; i < n; i++)
62                  {
63                          if(i != j)
64                          {
65
66                                  if(used[i] == 0)
67                                  {
68                                          if(c > 0)
69                                          {
70                                                  ans.push_back(j + 1);
71                                                  break;
72                                          }
73                                          dfs(i, j);
74                                          c++;
75                                  }
76                          }
77                  }
78          }
79
80          cout << ans.size() << endl;
81          for(int i = 0; i < ans.size(); i++)
82                  cout << ans[i] << " ";
83
84          return 0;
85  }
```

ПРИЛОЖЕНИЕ Г

Программный код задачи «Проверка на двудольность»

```
1  #include <iostream>
2  #include <map>
3  #include <string>
4  #include <fstream>
5  #include <stdio.h>
6  #include <set>
7  #include <vector>
8  #include <algorithm>
9  #include <queue>
10
11 using namespace std;
12
13 #define ll long long
```

```
14 #define pb push_back
15 #define mp make_pair
16
17 vector<int> used, color;
18 vector<vector<int>> g;
19 void dfs(int u, bool c, int &flag)
20 {
21         if(!flag)
22                 return;
23         used[u] = 1;
24         c ^= 1;
25         color[u] = c;
26         for(int i = 0; i < g[u].size(); i++)
27         {
28                 if(used[g[u][i]] == 0)
29                 {
30                         dfs(g[u][i], c, flag);
31                 }
32                 if(color[u] == color[g[u][i]])
33                 {
34                         flag = 0;
35                         return;
36                 }
37         }
38         used[u] = 2;
39 }
40
41 int main()
42 {
43 #ifdef _DEBUG
44     freopen("in.txt", "r", stdin);
45     freopen("out.txt", "w", stdout);
46 #endif
47
48         int t, n, m; cin >> t;
49         for(int q = 0; q < t; q++)
50         {
51                 cin >> n >> m;
52                 g = vector<vector<int>> (n);
53                 used = vector<int> (n, 0);
54                 color = vector<int> (n, -1);
55                 for(int i = 0; i < m; i++)
56                 {
57                         int x, y; cin >> x >> y;
58                         x--, y--;
59                         g[x].pb(y);
60                         g[y].pb(x);
61                 }
```

```
62
63                      int flag = 1;
64                      bool c = 0;
65                      for(int i = 0; i < n; i++)
66                      {
67                              if(used[i] == 0)
68                                      dfs(i, c, flag);
69                      }
70
71
72                      if(flag)
73                              cout << "YES\n";
74                      else
75                              cout << "NO\n";
76              }
77              return 0;
78      }
```