ЛЕКЦИЯ 18
Обход графа в глубину

1. Множества и словари

Задание множеств:

```
A = None # Пустое множество
A = {1, 2 'hello'} # Явное перечисление элементов
A = set('hello') # Множество букв в строке
B = [1, 2, 1, 2]; A = set(B) # Множество из списка или любого итерируемого объекта
```

Работа с элементами множеств:

```
C = {1, 2, 'hello'}
for elem in range(C): # Перебираем все элементы множества
    print(elem)
sorted(C) # Список из отсортированных элементов множества
1 in C # Проверка принадлежности
2 not in C
A.add(3) # Добавление элемента
A.remove(3) # Удаление элемента, который есть в множестве
A.discard(4) # Удаление элемента, которого может и не быть в множестве
A.pop() # Извлечение случайного элемента из множества с удалением его
```

Задание словарей

```
D = {} # Пустой словарь
D = {1: 'a', 2: 'b'} # Явное перечисление
D = dict([(1, 'a'), (2, 'b')]) # Словарь из списка пар элементов
D = dict(zip([1, 2], ['a', 'b'])) # Словарь из итерируемого объекта, возвращающего пары
значений
D = {i: chr(i + ord('a')) for i in range(1, 3)} # Генератор словарей
```

Работа с элементами словаря

```
len(D) # Количество элементов в словаре
D[key] # Поиск по ключу, который есть в словаре
key in D # Проверка принадлежности словарю
D[key] = value # Установка или изменение значения
del D[key] # Удаление ключа, который есть в словаре
value = D.pop(key) # Удаление ключа вместе с возвращением значения
value = D.pop(key, no_key_value)
key, value = D.popitem() # Извлечение из словаря пары (ключ, значение) с удалением ключа
D.get(key, no_key_value) # Значение по ключу, no_key_value, если ключа нет
D[key] = D.get(key, 0) + 1 # Самая простая реализация счетчика
for key in D: # Перебираем все ключи
    print(key, D[key])
for key, value in D.items(): # Перебираем все пары (ключ, значение)
```

```
    print(kay, value)
for value in D.values(): # Перебор всех значений
    print(value)
sorted(D) # Отсортированный список ключей
sorted(D.values()) # Отсортированный список значений
sorted(D.items()) # Отсортированный по ключу список пар (ключ, значение
sorted(D.items(), key = lambda x: x[1])
```

2. Реализация записи графа

Будем задавать граф как список ребер. Количество вершин, потом количество ребер.

5 6

После зададим ребра:

0 1
0 2
1 2
1 3
2 3
0 4

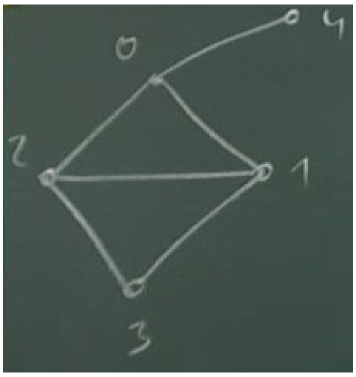

Рис. 1. Пример графа

Программа №2.1. Считывание графа как матрицы и как списка смежностей

```
def read_graph_as_matrix():
    N, M = [int(x) for x in input().split()]
    graph = [[0]*N for i in range(N)] # матрица смежностей
    for edge in range(M):
        a, b = [int(x) for x in input().split()]
        graph[a][b] = 1
        graph[b][a] = 1
    return graph

def print2d(A):
    for line in A:
        print(*line)
    print()

def read_graph_as_lists():
    N, M = [int(x) for x in input().split()]
    graph = [[] for i in range(N)]
    for edge in range(M):
        a, b = [int(x) for x in input().split()]
        graph[a].append(b)
        graph[b].append(a) # Для ориентированного графа строка не нужна
    return graph

graph = read_graph_as_lists()
print2d(graph)
```

В итоге `read_graph_as_matrix()` даст нам такой результат:

$$\begin{matrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{matrix}$$

а `read_graph_as_lists()`:

$$\begin{matrix} 1 & 2 & 4 \\ 0 & 2 & 3 \\ 0 & 1 & 3 \\ 1 & 2 & \\ 0 & & \end{matrix}$$

3. Алгоритм обхода графа в глубину

3.1. Алгоритм

Перебираем соседей по часовой стрелке. Граф считаем неориентированным.

Основное правило: для того, чтобы пойти на праздник надо вначале позвать всех своих друзей на праздник, но только тех, кто еще не позван.

Рис. 2. Граф с выбранным началом

Далее по часовой стрелке от 12 часов выбираем следующую вершину и зовём ее на праздник. Будем отмечать порядок обхода (номера показывают не индексы в графе, а просто порядок вызова).

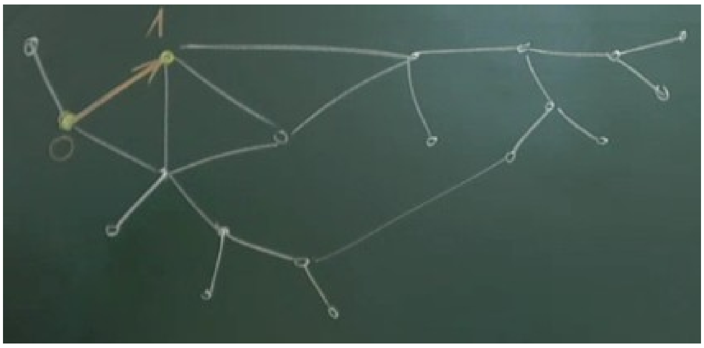

Рис. 3. Точка 1 перекрашена в "серый" цвет

Теперь вершину 1 красим в "серый" цвет. 0-ой ждёт ответа от 1-го.

1-ый начинает перебирать всех своих не позванных соседей по часовой стрелке от 12 часов.

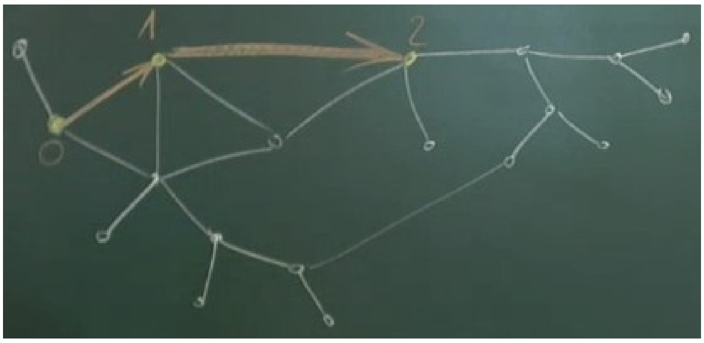

Рис. 4. Точка 2 перекрашена в "серый" цвет

И так процесс продолжается.

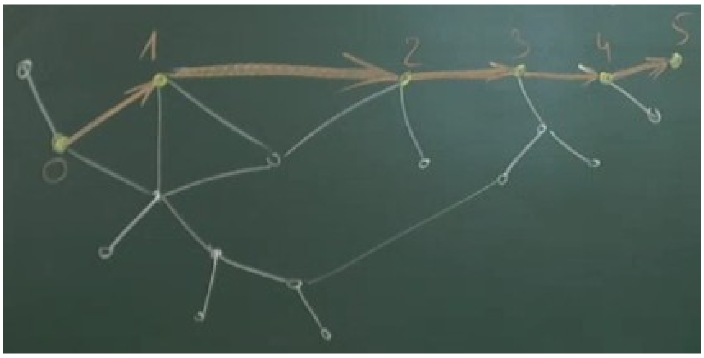

Рис. 5. Точки 0-5 перекрашены в "серый" цвет

Теперь 5-ый перебирает всех своих соседей. Но у него только один сосед — это 4-ый, и он уже позван. Т.о. 5-ый уже всех позвал. 5-я точка перекрашивается в "чёрный" цвет и идёт на праздник. 4-му возвращается от 5-го команда, что 5-ый всех позвал. Дальше 4-ый продолжает звать друзей и зовёт 6-го. 6-ая точка перекрашивается в "серый" цвет.

6-ой перебирает всех друзей и убеждается, что он всех позвал. Точка перекрашивается в "чёрный" цвет и идёт на праздник. 4-му возвращается команда, что 6-ой позвал всех друзей.

Т.о. 4-ый тоже позвал всех друзей. Точка перекрашивается в "чёрный" и дает команду 3-ему.

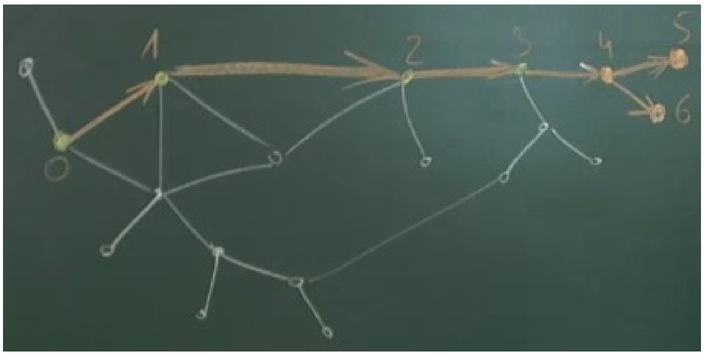

Рис. 6. Точка 4 перекрашена в "чёрный" цвет

3-ий зовёт оставшихся.

Процесс повторяется...

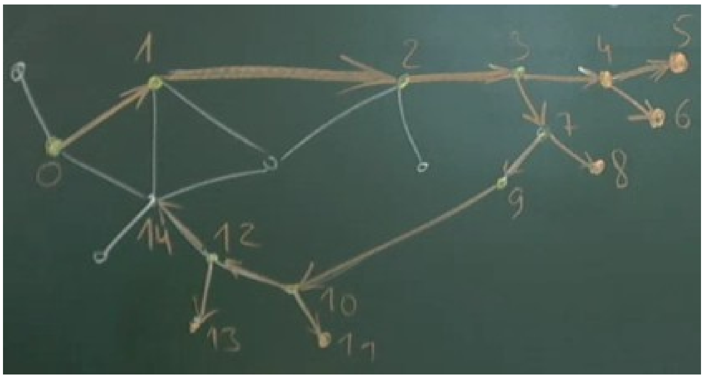

Рис. 7. Дошли до 14

14-ый позвал бы 1-го, но он уже позван. Заметим, что из серой вершины мы пытаемся позвать серую вершину, что значит, что в графе есть цикл.

Т.о. 14-ый зовёт следующего по часовой стрелке. 15 точка перекрашивается в "серый". 15-му звать некого, поэтому точка перекрашивается в "чёрный" и дает сигнал 14-му.

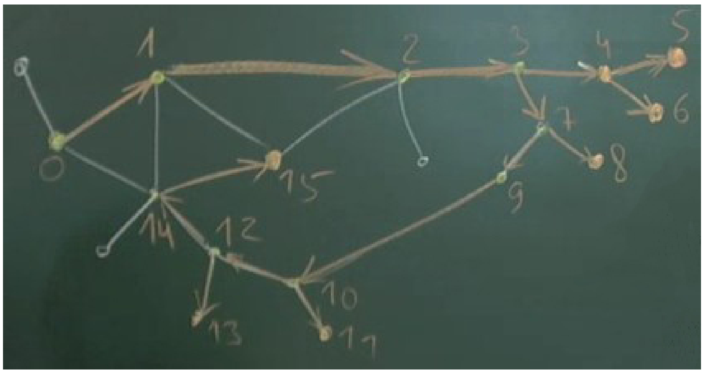

Рис. 8. Прошли 15-го

14-ый продолжает звать друзей, зовёт 16-го, 16-ый возвращает сигнал 14-му.

14-му становится некого звать. Он возвращает сигнал 13-му и перекрашивается в "чёрный". 13 возвращает сигнал 12-му и т.д. до 2-го и все они идут на праздник.

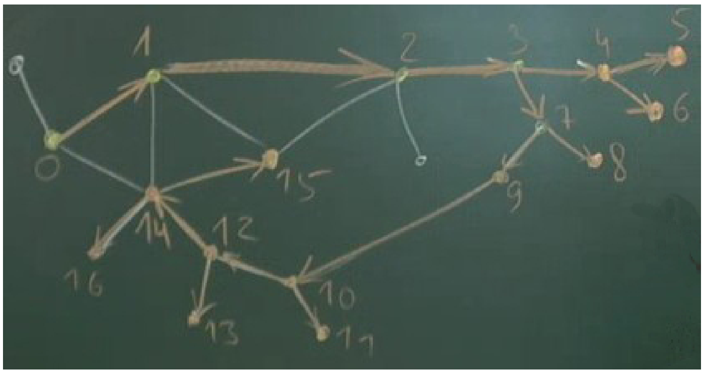

Рис. 9. Длинный возврат

2-ой зовёт 17-го, ему в свою очередь звать некого, он идёт на праздник и возвращает сигнал 2-му. 2-ой тоже всех позвал. В итоге сигнал возвращается 0-му, который зовёт 18-го. В итоге, все точки перекрашены в "чёрный" цвет.

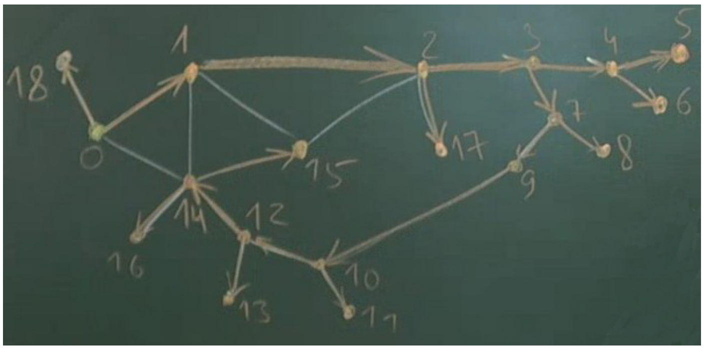

Рис. 10. Все позваны

Применение обхода в глубину:

1. Проверка связности графа (будем добавлять вершины в множество пройденых вершин)
2. Выделение компонентов связности
3. Поиск циклов (проверка ацикличности графов)
4. Поиск компонент сильной связности орграфов (алгоритм Косарайю)

3.2. Реализация на Python

Программа №3.1. Построение алгоритма

```
def read_graph_as_lists():
    N, M = [int(x) for x in input().split()]
    graph = [[] for i in range(N)]
    for edge in range(M):
        a, b = [int(x) for x in input().split()]
        graph[a].append(b)
        graph[b].append(a)
    return graph

def call_all_friends(me, friends, already_called = None):
    if already_called is None:
        already_called = set()
    """ Правило: тебя позвали на праздник, но пойти можно только тогда, когда
        позовешь всех своих еще не позванных друзей
    """
    already_called.add(me)
    for friend in friends[me]:
        if friend not in already_called:
            print(friend, 'был позван на праздник')
            call_all_friends(friend, friends, already_called)
            print(friend, 'пошел на праздник')

graph = read_graph_as_lists()
call_all_frineds(0, graph)
```

Запишем теперь более строго:

Программа №3.2. Реализация алгоритма обхода графа в глубину

```python
def dfs(vertex, graph, used = None): # Depth-first search
    if used is None:
        used = set()
    used.add(vertex)
    for neighbour in graph[vertex]:
        if neighbour not in used:
            dfs(neighbour, graph, used)

graph = read_graph_as_lists()
used = set()
number_of_components = 0
for vertex in range(len(graph)): # Подсчет компонент связности
    if vertex not in used:
        dfs(vertex, graph, used)
        number_of_components += 1

print('Количество компонент связности:', number_of_components)
```

4. Дерево

Дерево — связный граф, в котором

1. Нет простых циклов
2. От a к b только один путь
3. $N_{\text{вершин}} = M_{\text{ребер}} + 1$

Ориентированное дерево — ациклический орграф, в котором только одна вершина имеет нулевую степень захода (корень). Вершины с нулевой степенью исхода называются "листья". Остальные — узлы ветвления. Для корневого дерева можно ввести уровень узла — длина пути от корня до вершины (уровень иерархии).

У любого связного графа есть подграф, являющийся деревом и содержащий все исходные вершины.

Вершина сама по себе тоже является деревом.

Остовное дерево — пограф исходного графа, в котором выброшено максимальное количество ребер так, чтобы связность еще сохранилась. Обход графа в глубину позволяет построить одно из остовных деревьев.

Свойства:

- Дерево не имеет кратных ребер и петель
- Граф является деревом $\Leftrightarrow$ когда любые две различные вершины можно соединить единственным простым путем (простой цепью).
- Любое дерево однозначно определяется расстояниями между его концевыми вершинами со степенью 1 (длиной наименьшей цепи).
- Любое дерево, множество вершин которого более чем счетное является планарным графом.

Шарнир — вершина, при удалении которой количество компонент связности увеличивается.

Мост — ребро, при удалении которого количество компонент связности увеличивается. Такие ребра еще известны как разрезающие ребра или перешейки.

У дерева любая вершина является шарниром, а любое ребро мостом.

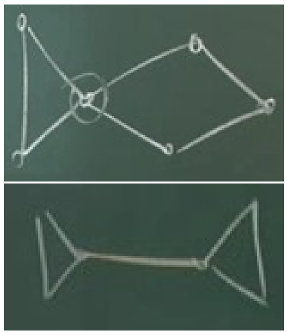

Рис. 11. Шарнир и мост

5. Алгоритм Косарайю

Задача: поиск сильно связной компоненты орграфа. Возьмем такой орграф. В нем будет 1 слабая компонента.

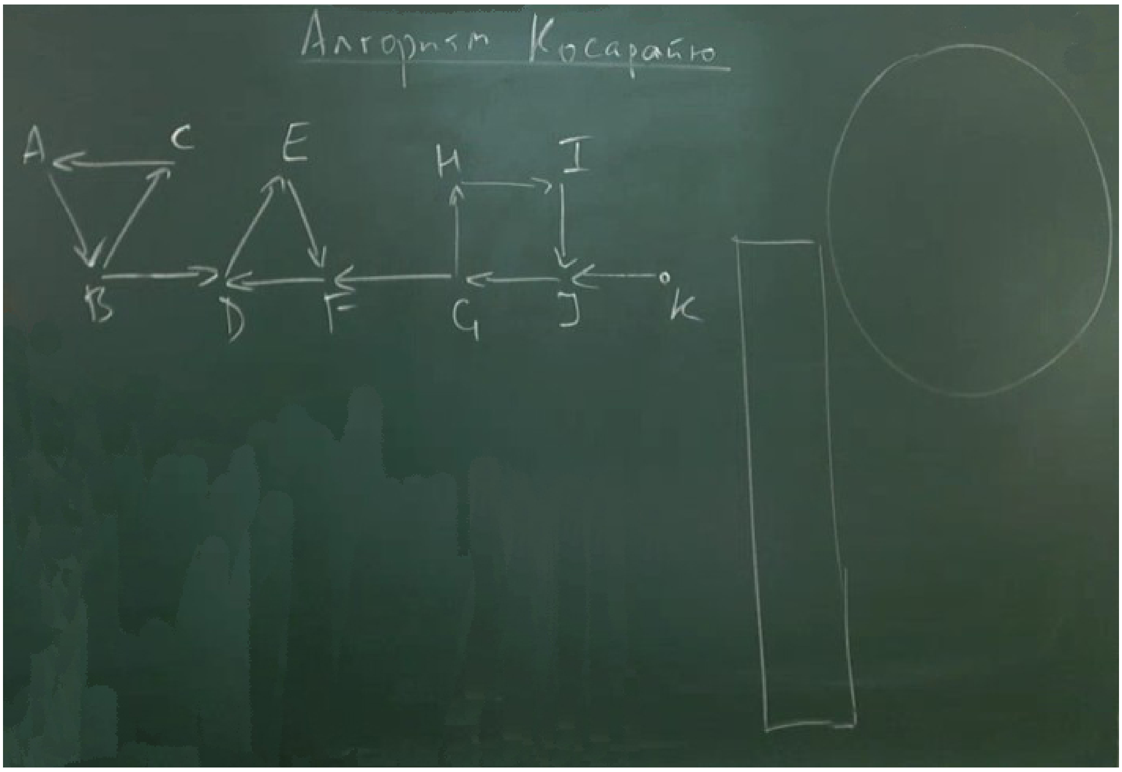

Рис. 12. Орграф

Запустим обход в глубину. Пусть будет множество вершин, которых мы уже использовали, и список вершин в порядке обхода (типа как стек).

Начнем с вершины A. Из A вызываем B. Они оказываются в множестве использованных вершин, также заполняется стек вызванных вершин. Далее вызовем вершину D (не принципиально D или C), добавим ее в множество и стек аналогично. Далее вызовем E, потом F. После происходит откат, и мы вызываем вершину C. Дальше выбираем любую вершину, которую не использовали. Пусть это будет G. Потом вызываем H, I, J. Остается вершина K, она становится последней.

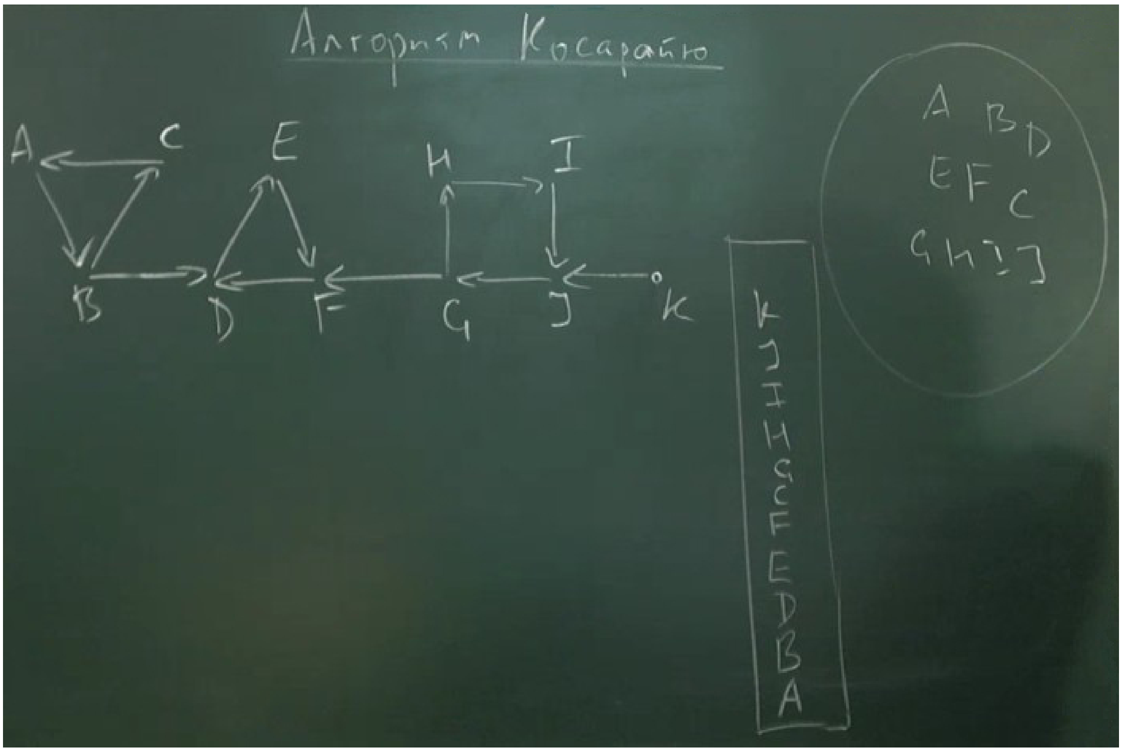

Рис. 13. После прохождения по графу

Далее мы разворачиваем наш граф, т.е. меняем все направления. А теперь от верхней вершины в стеке запустим обход в глубину на обращенном графе.

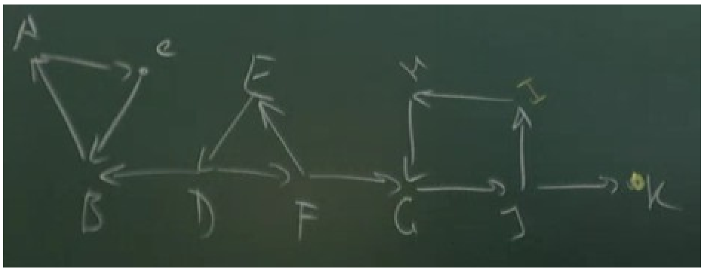

Рис. 14. Обращенный граф

Но от K дойти никуда нельзя. Это и есть сильная компонента. K добавляем в новое множество использованных вершин.

Дальше начинаем обход с вершины I, т.е. I—>H—>G—>J. Их добавляем в множество использованных и стираем из стека. Дальше идти опять некуда. Значит, это вторая сильная компонента.

Переходим к вершине C. Выполняем обход: C—>B—>A. Их добавляем в множество использованных и стираем из стека. Дальше идти некуда. Т.о. это еще одна компонента сильной связности.

Переходим к вершине F. В G уже не идем т.к. она использована. Выполняем обход F—>E—>D. Их добавляем в множество использованных и стираем из стека. Дальше идти некуда. Т.о. это еще одна компонента сильной связности.

В итоге получилось 4 компонент сильной связности.

Примечание: не принципиально, оборачивать ли граф сначала или потом (по википедии сначала нужно развернуть граф).