Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»
(Университет ИТМО)

Факультет **Инфокоммуникационных технологий**

Образовательная программа **Интеллектуальные системы в гуманитарной сфере**

Направление подготовки **45.03.04 Интеллектуальные системы в гуманитарной сфере**

О Т Ч Е Т

лабораторной работе 3

на тему: “Графы”

Обучающийся Королева Екатерина
К3143

Работа выполнена с оценкой ____

Преподаватель:
_____
(подпись)

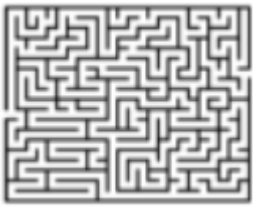
Дата 22.06.2022

Санкт-Петербург, 2021

Задачи из варианта 7:

1 Задача. Лабиринт [5 s, 512 Mb, 1 балл]

Лабиринт представляет собой прямоугольную сетку ячеек со стенками между некоторыми соседними ячейками.
Вы хотите проверить, существует ли путь от данной ячейки к данному выходу из лабиринта, где выходом также является ячейка, лежащая на границе лабиринта (в примере, показанном на рисунке, есть два выхода: один на левой границе и один на правой границе). Для этого вы представляете лабиринт в виде неориентированного графа: вершины графа являются ячейками лабиринта, две вершины соединены неориентированным ребром, если они смежные и между
ними нет стены. Тогда, чтобы проверить, существует ли путь между двумя заданными ячейками лабиринта, достаточно проверить, что существует путь между соответствующими двумя вершинами в графе.



Вам дан неориентированный граф и две различные вершины u и v. Проверьте, есть ли путь между u и v.
• Формат ввода / входного файла (input.txt). Неориентированный граф с n вершинами и m ребрами по формату 1. Следующая строка после ввода всго графа содержит две вершины u и v.
• Ограничения на входные данные. $2 \leq n \leq 103$,
$1 \leq m \leq 103$, $1 \leq u, v \leq n$, $u \neq v$.
• Формат вывода / выходного файла (output.txt). Выведите 1, если есть путь между вершинами u и v; выведите 0, если пути нет.
• Ограничение по времени. 5 сек.
• Ограничение по памяти. 512 мб.

```
import tracemalloc
import time
t_start = time.perf_counter()
tracemalloc.start()


def explore(x):
   visited[x] = True
   for apex in graph[x]:
```

```
        if not visited[apex]:
            explore(apex)


if __name__ == '__main__':
    graph = dict()
    with open('input.txt') as file:
        n, m = map(int, file.readline().split())
        for i in range(n):
            graph[i + 1] = list()
        for i in range(m):
            u, v = map(int, file.readline().split())
            graph[u].append(v)
            graph[v].append(u)
        s, file = map(int, file.readline().split())

    visited = [False]
    for _ in range(n):
        visited.append(False)
    explore(s)

    with open('output.txt', 'w') as file:
        file.write(f'{"1" if visited[file] else "0"}')


print("Время работы (в секундах):",
time.perf_counter()-t_start)
print("Память %d, и пик %d" % tracemalloc.get_traced_memory())

'''
Считываем данные и записываем в словарь связи между вершинами
графа. Запускаем функцию explore, которая заменяет
в каждом элементе массива visited False на True, если она
может пройти из начальной точки в текущую. Если значение
конечной точки равно True, то выводим 1, противном случае – 0.
'''

Время работы (в секундах): 0.0006937000000000054
Память 2976, и пик 18355
```

11 Задача. Алхимия [1 s, 16 Mb, 3 балла]*Замена задачи 10

Алхимики средневековья владели знаниями о превращении различных химических веществ друг в друга. Это подтверждают и недавние исследования археологов. В ходе археологических раскопок было обнаружено m глиняных табличек, каждая из которых была покрыта непонятными на первый взгляд символами. В результате расшифровки выяснилось, что каждая из табличек описывает одну алхимическую реакцию, которую умели проводить алхимики. Результатом алхимической реакции является превращение одного вещества в другое. Задан набор алхимических реакций, описанных на найденных глиняных табличках, исходное вещество и требуемое вещество. Необходимо выяснить: возможно ли преобразовать исходное вещество в требуемое с помощью этого набора реакций, а в случае положительного ответа на этот вопрос – найти минимальное количество реакций, необходимое для осуществления такого преобразования.

• Формат входных данных (input.txt) и ограничения. Первая строка входного файла INPUT.TXT содержит целое число m (0 ≤ m ≤ 1000) – количество записей в книге. Каждая из последующих m строк описывает одну алхимическую реакцию и имеет формат «вещество1 -> вещество2», где «вещество1» – название исходного вещества, «вещество2» – название продукта алхимической реакции. m + 2-ая строка входного файла содержит название вещества, которое имеется исходно, m + 3-ая – название вещества, которое требуется получить. Во входном файле упоминается не более 100 различных веществ. Название каждого из веществ состоит из строчных и заглавных английских букв и имеет длину не более 20 символов. Строчные и заглавные буквы различаются.

• Формат выходных данных (output.txt). В выходной файл OUTPUT.TXT выведите минимальное количество алхимических реакций, которое требуется для получения требуемого вещества из исходного, или -1, если требуемое вещество невозможно получить.

• Ограничение по времени. 1 сек.

• Ограничение по памяти. 16 мб.

```
import tracemalloc
import time
t_start = time.perf_counter()
tracemalloc.start()
```

```
import collections


def find_min_path(elements_graph, start: str, end: str):
    len_map = dict()
    queue_name = collections.deque()
    len_map[start] = 0
    queue_name.append(start)
    while len(queue_name) != 0:
        curr_v = queue_name.popleft()
        if curr_v == end:
            return len_map[curr_v]
        if curr_v in elements_graph:
            for next_elem in elements_graph[curr_v]:
                if next_elem not in len_map:
                    len_map[next_elem] = len_map[curr_v] + 1
                    queue_name.append(next_elem)
    return -1


if __name__ == '__main__':
    n = int(input())
    elements_graph = dict()
    for i in range(n):
        inp_str = list(map(str, input().split(" -> ")))
        if inp_str[0] not in elements_graph:
            elements_graph[inp_str[0]] = [inp_str[1]]
        else:
            elements_graph[inp_str[0]].append(inp_str[1])
    fr_element = input()
    to_elem = input()
    print(find_min_path(elements_graph, fr_element, to_elem))

print("Время работы (в секундах):", time.perf_counter() -
t_start)
print("Память %d, и пик %d" % tracemalloc.get_traced_memory())

'''
Задача аналогична задаче 6, только вместо чисел, ключами
вершин графа являются названия веществ.
Поэтому делаем поправку на ввод ориентированного графа, и
задача решена.
```

P.s.6 задача: Для решения данной задачи нам понадобится обход графа в ширину. Для этого используется очередь. В очередь добавляются
все еще не посещенные вершины, доступные из текущей, вместе с количеством ребер, которое надо пройти, чтобы достичь
данной вершины из исходной. Считываем неориентированный граф и кладем результат функции “shortPath” в переменную
“result”, значение которой и есть ответ.
'''

Время работы (в секундах): 2.3703427
Память 1903, и пик 9989

Проверка задачи на acmp:

Екатерина Максимовна Королева	0743	Python	Accepted		0,046	802 Кб

14 Задача. Автобусы [1 s, 16 Mb, 3 балла]

Между некоторыми деревнями края Власюки ходят автобусы. Поскольку пассажиропотоки здесь не очень большие, то автобусы ходят всего несколько раз в день. Марии Ивановне требуется добраться из деревни d в деревню v как можно быстрее (считается, что в момент времени 0 она находится в деревне d).

• Формат входных данных (input.txt) и ограничения. Во входном файле INPUT.TXT записано число N - общее число деревень ($1 \le N \le 100$), номера деревень d и v, затем количество автобусных рейсов R ($0 \le R \le 10000$). Затем идут описания автобусных рейсов. Каждый рейс задается номером деревни отправления, временем отправления, деревней назначения и временем прибытия (все времена - целые от 0 до 10000). Если в момент t пассажир приезжает в деревню, то уехать из нее он может в любой момент времени, начиная с t.

• Формат выходных данных (output.txt). В выходной файл OUTPUT.TXT вывести минимальное время, когда Мария Ивановна может оказаться в деревне v. Если она не сможет с помощью указанных автобусных рейсов добраться из d в v, вывести -1.

• Ограничение по времени. 1 сек.

• Ограничение по памяти. 16 мб.

```
import tracemalloc
import time
t_start = time.perf_counter()
tracemalloc.start()


with open('input.txt') as f:
   N = int(f.readline())
   buses = [[] for _ in range(N+1)]
   d, v = map(int, f.readline().split())
   R = int(f.readline())
   for i in range(R):
       n1, t1, n2, t2 = map(int, f.readline().split())
       buses[n1].append((t1, n2, t2))

INF = float('inf')
Time  = [INF] * (N+1)
Time[d] = 0
visited = [False] * (N+1)
while True:
   min_time = INF
   for i in range(1, N+1):
```

```
        if not visited[i] and Time[i] < min_time:
            min_time = Time[i]
            min_village = i
    if min_time == INF:
        break
    n1 = min_village
    visited[n1] = True
    for t1, n2, t2 in buses[n1]:
        if Time[n1] <= t1 and t2 <= Time[n2]:
            Time[n2] = t2

with open('output.txt', 'w') as f:
    if Time[v] == INF:
        f.write('-1')
    else:
        f.write(str((Time[v])))


print("Время работы (в секундах):", time.perf_counter() -
t_start)
print("Память %d, и пик %d" % tracemalloc.get_traced_memory())

'''
Считываем описания рейсов в массив “buses”, пункты отправления
и прибытия. Далее формируем массив “Time”, который мы
заполняем бесконечностями, а ячейку пункта отправления
обнуляем. Массив будет показывать минимальное время прибытия в
соответствующую деревню. Также понадобится массив “visited”
для обозначения того, проверили ли мы все рейсы из данной
деревни или нет. В цикле мы ищем деревню с минимальным
временем прибытия и отмечаем ее, как проверенную на все рейсы.
Затем проверяем рейсы из данной деревни. Если время его
отправления из нее меньше, чем время прибытия в нее, а время
прибытия в другую деревню меньше той, что была, то заменяем
время прибытия в деревню по соответствующему рейсу.
Таким образом, мы получим массив прибытий в деревни за
минимально возможное время. Если же значение времени осталось
бесконечным, то в данную деревню невозможно попасть с данными
рейсами.
'''

Время работы (в секундах): 0.0007604999999999973
Память 2480, и пик 19066
```

Проверка задачи на acmp:

Екатерина Максимовна Королева	0134	Python	Accepted		0,046	154

Дополнительные задачи:

3 Задача. Циклы [5 s, 512 Mb, 1 балл]

Учебная программа по инфокоммуникационным технологиям определяет пререквизиты для каждого курса в виде списка курсов, которые необходимо пройти перед тем, как начать этот курс. Вы хотите выполнить проверку согласованности учебного плана, то есть проверить отсутствие циклических зависимостей. Для этого строится следующий ориентированный граф: вершины соответствуют курсам, есть направленное ребро (u, v) – курс u следует пройти перед курсом v. Затем достаточно проверить, содержит ли полученный граф цикл. Проверьте, содержит ли данный граф циклы.

• Формат ввода / входного файла (input.txt). Ориентированный граф с n вершинами и m ребрами по формату 1.

• Ограничения на входные данные. $1 \le n \le 103$, $0 \le m \le 103$.

• Формат вывода / выходного файла (output.txt). Выведите 1, если данный граф содержит цикл; выведите 0, если не содержит.

• Ограничение по времени. 5 сек.

• Ограничение по памяти. 512 мб.

```
import tracemalloc
import time
t_start = time.perf_counter()
tracemalloc.start()


with open('input.txt') as f:
    n, m = map(int, f.readline().split())
    sides = {}
    for i in range(1, n+1):
        sides[i] = []
    for i in range(m):
        v1, v2 = map(int, f.readline().split())
        sides[v1].append(v2)

for u in sides:
    visited = []
    parent = {}
    cur_node = u
    node_found = 1
    node_completed = 0
    while True:
        visited.append(cur_node)
```

```
        flag = False
        for i in sides[cur_node]:
            if i == u:
                print(1)
                exit()
            if i not in visited:
                parent[i] = cur_node
                cur_node = i
                node_found += 1
                flag = True
                break
        if not flag:
            node_completed += 1
            if node_found == node_completed:
                break
            cur_node = parent[cur_node]


print("Время работы (в секундах):", time.perf_counter() -
t_start)
print("Память %d, и пик %d" % tracemalloc.get_traced_memory())

'''
Считываем граф, учитывая, что он ориентированный. По сути,
чтобы обнаружить цикл, нужно найти вершину, которая
достижима сама из себя. Для пользуемся обходом в глубину и
пробегаемся по всем вершинам графа. Если таковая вершина
находится, значит в графе есть цикл и выводится ‘1’. В
обратном случае выводится ‘0’.
'''

Время работы (в секундах): 0.0008580000000000115
Память 2376, и пик 18486
```

6 Задача. Количество пересадок [10 s, 512 Mb, 1 балл]

Вы хотите вычислить минимальное количество сегментов полета, чтобы добраться из одного города в другой. Для этого вы строите следующий неориентированный граф: вершины представляют города, между двумя вершинами есть ребро всякий раз, когда между соответствующими двумя городами есть перелет. Тогда достаточно найти кратчайший путь из одного из заданных городов в другой.

Дан неориентироанный граф с n вершинами и m ребрами, а также две вершины u и v, нужно посчитать длину кратчайшего пути между u и v (то есть, минимальное количество ребер в пути из u в v).

• Формат ввода / входного файла (input.txt). Нериентированный граф задан по формату 1. Следующая строка содержит две вершины u и v.

• Ограничения на входные данные. $2 \leq n \leq 10^5$, $0 \leq m \leq 10^5$, $1 \leq u, v \leq n$, $u \neq v$.

• Формат вывода / выходного файла (outputt.txt). Выведите минимальное количество ребер в пути из u в v. Выведите -1, если пути нет.

• Ограничение по времени. 10 сек.

• Ограничение по памяти. 512 мб.

```
import tracemalloc
import time
t_start = time.perf_counter()
tracemalloc.start()


from collections import deque

def shortPath(u, v):
   global sides
   search_queue = deque()
   search_queue.append((u, 0))
   visited = []
   while search_queue:
       cur_node, path = search_queue.popleft()
       if cur_node == v:
           return path
       path += 1
       if cur_node not in visited:
           visited.append(cur_node)
           for node in sides[cur_node]:
```

```
                search_queue.append((node, path))
    return -1

with open('input.txt') as f:
    n, m = map(int, f.readline().split())
    sides = {}
    for i in range(n+1):
        sides[i] = []
    for i in range(m):
        v1, v2 = map(int, f.readline().split())
        sides[v1].append(v2)
        sides[v2].append(v1)
    u, v = map(int, f.readline().split())

result = shortPath(u, v)
print(result)


print("Время работы (в секундах):", time.perf_counter() -
t_start)
print("Память %d, и пик %d" % tracemalloc.get_traced_memory())

'''
Для решения данной задачи нам понадобится обход графа в
ширину. Для этого используется очередь. В очередь добавляются
все еще не посещенные вершины, доступные из текущей, вместе с
количеством ребер, которое надо пройти, чтобы достичь
данной вершины из исходной. Считываем неориентированный граф и
кладем результат функции “shortPath” в переменную
“result”, значение которой и есть ответ.
'''
```

```
Время работы (в секундах): 0.00026670000000000166
Память 3329, и пик 18067
```

7 Задача. Двудольный граф [10 s, 512 Mb, 1.5 балла]

Неориентированный граф называется двудольным, если его вершины можно разбить на две части так, что каждое ребро графа соединяет вершины из разных частей, то есть не существует рёбер между вершинами одной и той же части графа. Двудольные графы естественным образом возникают в задачах, где граф используется для моделирования связей между объектами двух разных типов (например, мальчиками и девочками, или студентами и общежитиями). Альтернативное определение таково: граф двудольный, если его вершины можно раскрасить двумя цветами (например, черным и белым) так, что концы каждого ребра окрашены в разные цвета.

Дан неориентированный граф с n вершинами и m ребрами, проверьте, является ли он двудольным.

• Формат ввода / входного файла (input.txt). Неориентированный граф задан по формату 1.

• Ограничения на входные данные. $1 \le n \le 105$, $0 \le m \le 105$.

• Формат вывода / выходного файла (output.txt). Выведите 1, если граф двудольный; и 0 в противном случае.

• Ограничение по времени. 10 сек.

• Ограничение по памяти. 512 мб.

```
import tracemalloc
import time
t_start = time.perf_counter()
tracemalloc.start()


def explore(x, temp):
   visited[x] = True
   for apex in graph[x]:
       if not visited[apex]:
           if temp:
               for t in graph[apex]:
                   if t in white:
                       return False
               white.add(apex)
               if not explore(apex, False):
                   return False
           else:
               for t in graph[apex]:
                   if t in black:
                       return False
               black.add(apex)
```

```
                if not explore(apex, True):
                    return False
    return True


def DFS(graph):
    for v in graph:
        if not visited[v]:
            black.add(v)
            if not explore(v, True):
                return False
    return True


if __name__ == '__main__':
    graph = dict()
    white, black = set(), set()
    with open('input.txt') as file:
        n, m = map(int, file.readline().split())
        for i in range(n):
            graph[i + 1] = list()
        for i in range(m):
            u, v = map(int, file.readline().split())
            graph[u].append(v)
            graph[v].append(u)
    visited = [False]
    for _ in range(n):
        visited.append(False)

    with open('output.txt', 'w') as file:
        file.write(f'{DFS(graph)}')

print("Время работы (в секундах):", time.perf_counter() -
t_start)
print("Память %d, и пик %d" % tracemalloc.get_traced_memory())

'''
Считываем неориентированный граф и запускаем функцию DFS
(поиск в глубину), которая проходит по каждой вершине графа.
Если вершина еще не посещалась, то функция присваивает ей
черный цвет. Затем проверяется, исследована функция или нет.
В функции explore параметрами являются вершина и True. Затем
для каждой вершины, связанной с текущей, идет проверка.
```

Если вершина еще не посещалась, то проверяется, имеет ли она отличный цвет от текущей. В итоге, если все имеют различный цвет, то выводим True, в противном случае – False.
'''

Время работы (в секундах): 0.0005886999999999976
Память 3544, и пик 20062

8 Задача. Стоимость полета [10 s, 512 Mb, 1.5 балла]

Теперь вас интересует минимизация не количества пересадок, а общей стоимости полета. Для этого строится взвешенный граф: вес ребра из одного города в другой – это стоимость соответствующего перелета.

Дан ориентированный граф с положительными весами ребер, n - количество вершин и m - количество ребер, а также даны две вершины u и v. Вычислить вес кратчайшего пути между u и v (то есть минимальный общий вес пути из u в v).

• Формат ввода / входного файла (input.txt). Ориентированный взвешенный граф задан по формату 1. Следующая строка содержит две вершины u и v.

• Ограничения на входные данные. 1 ≤ n ≤ 104, 0 ≤ m ≤ 105, 1 ≤ u, v ≤ n, u ≠ v, вес каждого ребра – неотрицательное целое число, не превосходящее 108.

• Формат вывода / выходного файла (output.txt). Выведите минимальный вес пути из u в v. Веведите -1, если пути нет.

• Ограничение по времени. 10 сек.

• Ограничение по памяти. 512 мб.

```
import tracemalloc
import time
t_start = time.perf_counter()
tracemalloc.start()


def extractMin():
    m = float('+inf')
    im = 0
    for i in graph:
        if distance[i] < m:
            m = distance[i]
            im = i
    if im == 0:
        return im, []
    subj = graph[im]
    graph.pop(im)
    return im, subj


def dijkstra(start):
    distance[start] = 0
    while len(graph) != 0:
```

```
        u, subj = extractMin()
        if u == 0:
            break
        for v in subj:
            if distance[v] > distance[u] + weights[(u, v)]:
                distance[v] = distance[u] + weights[(u, v)]


if __name__ == '__main__':
    graph = dict()
    weights = dict()
    distance = [float('+inf')]
    with open('input.txt') as file:
        n, m = map(int, file.readline().split())
        for i in range(n):
            graph[i + 1] = list()
        for i in range(m):
            u, v, m = map(int, file.readline().split())
            graph[u].append(v)
            weights[(u, v)] = m
        s, f = map(int, file.readline().split())

    for _ in range(n):
        distance.append(float('+inf'))

    dijkstra(s)

    with open('output.txt', 'w') as file:
        file.write(f'{"-1" if distance[f] == float("+inf") else
str(distance[f])}')

print("Время работы (в секундах):", time.perf_counter() -
t_start)
print("Память %d, и пик %d" % tracemalloc.get_traced_memory())

'''
Считываем входные данные, записываем веса в значения
отдельного словаря, в котором ключи - ребра. Затем запускаем
алгоритм Дейкстры. В этом алгоритме мы проходим по каждому
значению графа, запуская функцию extractMin, которая находит
минимальный вес из всех возможных, записывая его индекс. Затем
из основного графа удаляется вершина с этим весом и
значения индекса и подграфа возвращаются в алгоритм Дейкстры.
Затем проверяется, если хоть одна из вершин подграфа имеет
```

```
большую дистанцию, чем дистанция полученного индекса и вес
ребра, то заменяем. В итоге получаем минимальный вес,
в противном случае – -1.
'''
```

Время работы (в секундах): 0.0006314000000000042
Память 3296, и пик 19819

16 Задача. Рекурсия [1 s, 16 Mb, 3 балла]

Одним из важных понятий, используемых в теории алгоритмов, является рекурсия. Неформально ее можно определить как использование в описании объекта самого себя. Если речь идет о процедуре, то в процессе исполнении эта процедура напрямую или косвенно (через другие процедуры) вызывает сама себя.

Рекурсия является очень «мощным» методом построения алгоритмов, но таит в себе некоторые опасности. Например, неаккуратно написанная рекурсивная процедура может войти в бесконечную рекурсию, то есть, никогда не закончить свое выполнение (на самом деле, выполнение закончится с переполнением стека). Поскольку рекурсия может быть косвенной (процедура вызывает сама себя через другие процедуры), то задача определения того факта, является ли данная процедура рекурсивной, достаточно сложна. Попробуем решить более простую задачу.

Рассмотрим программу, состоящую из n процедур P1, P2, . . . , Pn. Пусть для каждой процедуры известны процедуры, которые она может вызывать. Процедура P называется потенциально рекурсивной, если существует такая последовательность процедур Q0, Q1, . . . , Qk, что Q0 = Qk = P и для i = 1...k процедура Qi−1 может вызвать процедуру Qi. В этом случае задача будет заключаться в определении для каждой из заданных процедур, является ли она потенциально рекурсивной.

Требуется написать программу, которая позволит решить названную задачу.

• Формат входных данных (input.txt) и ограничения. Первая строка входного файла INPUT.TXT содержит целое число n – количество процедур в программе (1 ≤ n ≤ 100). Далее следуют n блоков, описывающих процедуры. После каждого блока следует строка, которая содержит 5 символов «*».

Описание процедуры начинается со строки, содержащий ее идентификатор, состоящий только из маленьких букв английского алфавита и цифр. Идентификатор непуст, и его длина не превосходит 100 символов. Далее идет строка, содержащая число k (k ≤ n) – количество процедур, которые могут быть вызваны описываемой процедурой. Последующие k строк содержат идентификаторы этих процедур – по одному идентификатору на строке.

Различные процедуры имеют различные идентификаторы. При этом ни одна процедура не может вызвать процедуру, которая не описана во входном файле.

• Формат выходных данных (output.txt). В выходной файл OUTPUT.TXT для каждой процедуры, присутствующей во входных

данных, необходимо вывести слово YES, если она является потенциально рекурсивной, и слово NO – в противном случае, в том же порядке, в каком они перечислены во входных данных.

- Ограничение по времени. 1 сек.
- Ограничение по памяти. 16 мб.

```
import tracemalloc
import time
t_start = time.perf_counter()
tracemalloc.start()

file1 = open('input.txt')
file2 = open('output.txt', 'w')
n = int(file1.readline())
sides = {}
for i in range(n):
    v1 = file1.readline()
    sides[v1] = []
    k = int(file1.readline())
    for j in range(k):
        v2 = file1.readline()
        sides[v1].append(v2)
    edge = file1.readline()
for u in sides:
    visited = []
    parent = {}
    cur_node = u
    node_found = 1
    node_completed = 0
    while True:
        visited.append(cur_node)
        flag = False
        found = False
        for i in sides[cur_node]:
            if i == u:
                file2.write('YES\n')
                found = True
                break
            if i not in visited:
                parent[i] = cur_node
                cur_node = i
                node_found += 1
                flag = True
                break
```

```
        if found:
            break
        if not flag:
            node_completed += 1
            if node_found == node_completed:
                break
            cur_node = parent[cur_node]
    if not found:
        file2.write('NO\n')


print("Время работы (в секундах):", time.perf_counter() -
t_start)
print("Память %d, и пик %d" % tracemalloc.get_traced_memory())

'''
Для чтобы функция была потенциально рекурсивной, нужно, чтобы
в графе существовал цикл, включающий данную функцию.
Для этого с помощью обхода в глубину ищем из вершины графа в
саму себя, как и в задаче 3.
P.s.3 задача: Считываем граф, учитывая, что он
ориентированный. По сути, чтобы обнаружить цикл, нужно найти
вершину,
которая достижима сама из себя. Для пользуемся обходом в
глубину и пробегаемся по всем вершинам графа. Если таковая
вершина  находится, значит в графе есть цикл и выводится '1'.
В обратном случае выводится '0'.
'''
```

```
Время работы (в секундах): 0.0006892000000000009
Память 4155, и пик 20080
```

Екатерина Максимовна Королева	0345	Python	Accepted		0,078	2022

Вывод:

В ходе данной лабораторной работы была изучена и применена на практике такая структура данных, как графы. Рассмотрены различные виды графов, типы их обходов и другие алгоритмы, связанные с графами.