САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе № 3
по курсу «Алгоритмы и структуры данных»
Тема: Графы
Вариант №4

Выполнил:
Волжева М.И.
К3141

Проверила:
Артамонова В.Е.

Санкт-Петербург
2023 г.

Содержание отчета

Задачи по варианту

Задача №4. Порядок курсов [10 s, 512 Mb, 1 балл]

- Текст задачи:
  Теперь, когда вы уверены, что в данном учебном плане нет циклических зависимостей, вам нужно найти порядок всех курсов, соответствующий всем зависимостям. Для этого нужно сделать топологическую сортировку соответствующего ориентированного графа. Дан ориентированный ациклический граф (DAG) с n вершинами и m ребрами. Выполните топологическую сортировку.
- Формат входного файла (input.txt).
  Ориентированный ациклический граф с n вершинами и m ребрами по формату 1.
- Ограничения на входные данные.
  $1 \leq n \leq 10^5$, $0 \leq m \leq 10^5$. Графы во входных файлах гарантированно ациклические.
- Формат выходного файла (output.txt).
  Выведите любое линейное упорядочение данного графа (Многие ациклические графы имеют более одного варианта упорядочения, вы можете вывести любой из них).
- Пример:

Input.txt	4 1 3 1	4 3 1 2 4 1 3 1	5 7 2 1 3 2 3 1 4 3 4 1 5 2 5 3
Output.txt	2 3 1 4	4 3 1 2	5 4 3 2 1

Листинг кода:

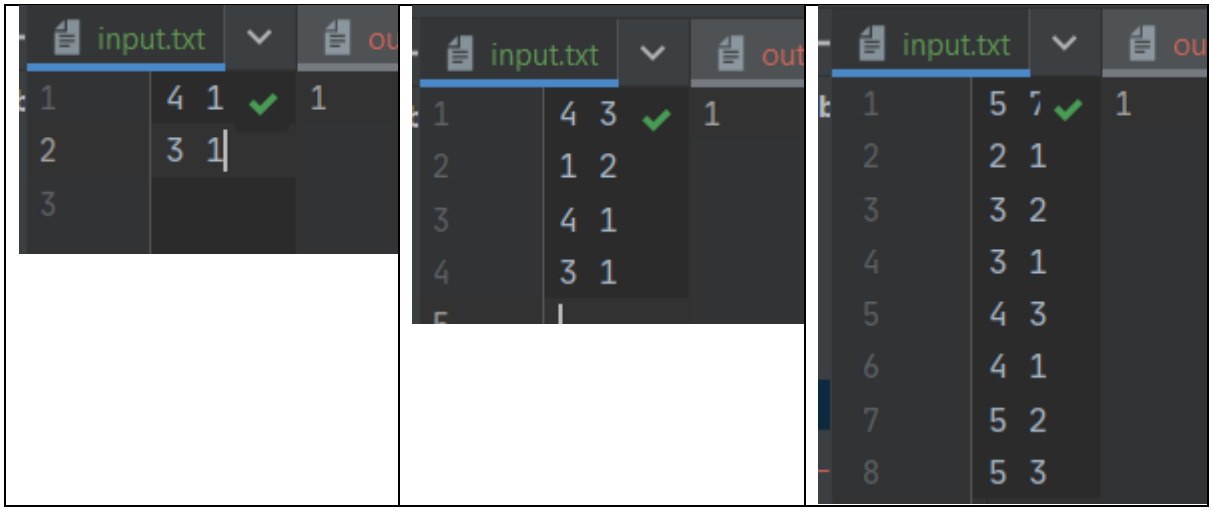
```
import time
import os, psutil

t_start = time.perf_counter()
process = psutil.Process(os.getpid())

f = open('input.txt', 'r')
num_vertices, num_edges = map(int, f.readline().split())
graph = [[] for _ in range(num_vertices)]
```

```
in_degrees = [0] * num_vertices
for _ in range(num_edges):
    start, end = map(int, f.readline().split())
    graph[start - 1].append(end - 1)
    in_degrees[end - 1] += 1

no_incoming_vertices = []
for i in range(num_vertices):
    if in_degrees[i] == 0:
        no_incoming_vertices.append(i)

result = []
while no_incoming_vertices:
    vertex = no_incoming_vertices.pop()
    result.append(vertex + 1)
    for neighbor in graph[vertex]:
        in_degrees[neighbor] -= 1
        if in_degrees[neighbor] == 0:
            no_incoming_vertices.append(neighbor)

t = open('output.txt', 'w+')
for vertex in result:
    t.write(str(vertex) + ' ')


print("Time of working: %s second" % (time.perf_counter() - t_start))
print("Memory", process.memory_info().rss/(1024*1024), "mb")
```

Текстовое объяснение решения:

Нам нужно указать такой линейный порядок на его вершинах, чтобы любое ребро вело от вершины с меньшим номером к вершине с большим номером. Сначала мы читаем данные из файла. Топологическая сортировка выполняется с использованием алгоритма Кана, который включает в себя итеративный выбор вершин без входящих ребер (т.е. вершин с нулевой степенью), добавление их к результату и удаление их и их исходящих ребер из графика. Этот процесс продолжается до тех пор, пока все вершины не будут добавлены к результату или не будет определено, что граф имеет цикл (т.е. нет вершин с нулевой степенью и не все вершины были добавлены крезультату).

Результат работы кода на примерах из текста задачи:



Тестирование алгоритма:

	Время выполнения (seconds)	Затраты памяти (mbs)
Пример	0.0008377999999999997	14.1953125
Пример	0.005640300000000015	14.09375
Пример	0.005374099999999993	4.1015625

Вывод по задаче: Мы научились делать топологическую сортировку соответствующего ориентированного графа.

Задача №9. Аномалии курсов валют [10 s, 512 Mb, 2 балла]

- Текст задачи:
  Вам дан список валют c1, c2, . . . , cn вместе со списком обменных курсов:rij – количество единиц валюты cj , котороеможно получить за одну единицу ci. Вы хотите проверить, можно линачать делать обмен с одной единицы какой-либо валюты, выполнить последовательность обменов и получить более одной единицы той же валюты, с которой вы начали бмен. Тогда достаточно проверить, есть ли в этом графе отрицательныйцикл. Пусть цикл ci → cj → ck → ci имеет отрицательный вес. Это означает, что −(log cij + log cjk + log cki) < 0 и, следовательно, log cij + log cjk + log cki > 0. Для заданного ориентированного графа с возможными отрицательнымивесами ребер, у которого n вершин и m ребер, проверьте, содержит ли он цикл с отрицательным суммарным весом.

m ребер, проверьте, содержит ли он цикл с отрицательным суммарным весом.

- Формат ввода / входного файла (input.txt). Ориентированный взвешенный граф задан по формату 1.
- Ограничения на входные данные. 1 ≤ n ≤ 10^3, 0 ≤ m ≤ 10^4, вес каждого ребра – целое число, не превосходящее по модулю 10^4.
- Формат выходного файла (output.txt).
  Выведите 1, если граф содержит цикл с отрицательным суммарным весом. Выведите 0 в противном случае.
- Пример:

Input.txt	4 4 1 2 -5 4 1 2 2 3 2 3 1 1
Output.txt	1

Листинг кода:

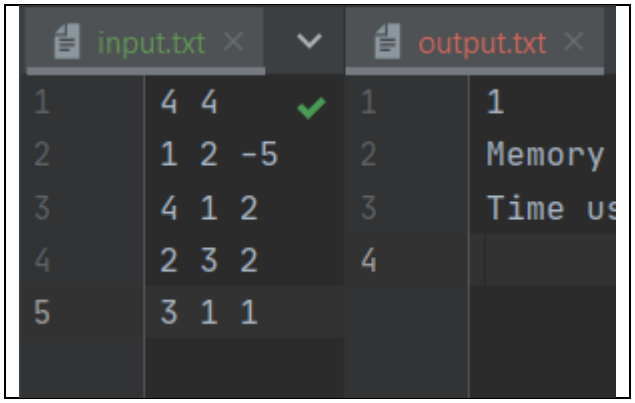
```
def create_graph(n, m, edges, directed=False):
    graph = {i + 1: [] for i in range(n)}
    for edge in edges:
        u, v = map(int, edge.split()[:2])
        if len(edge.split()) == 3:
            w = int(edge.split()[2])
            graph[u] = [*graph[u], (v, w)] if u in graph.keys() else [(v,
w)]
        else:
            graph[u] = [*graph[u], v] if u in graph.keys() else [v]
            if not directed:
                graph[v] = [*graph[v], u] if v in graph.keys() else [u]
    return graph


def has_negative_cycle(graph):
    dist = {v: 0 for v in graph}
    for _ in range(len(graph) - 1):
        for u in graph:
            for v, weight in graph[u]:
                if dist[v] > dist[u] + weight:
                    dist[v] = dist[u] + weight
    for u in graph:
        for v, weight in graph[u]:
            if dist[v] > dist[u] + weight:
                return True
    return False

def measure_memory_and_time(func):
    import time
    import psutil
    import os
```

```
    def _wrapped(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        memory = psutil.Process(os.getpid()).memory_info().rss
        print(f"Memory usage: {memory // 1024 ** 2} megabytes")
        print(f"Time usage: {end_time - start_time} seconds")
        return result
    return _wrapped

def set_stdout(file: str = "output.txt"):
    try:
        import sys
        sys.stdout = open(file, 'wt', encoding='utf-8')
    except Exception as e:
        raise IOError(e)


@measure_memory_and_time
def main():
    set_stdout()
    with open('input.txt', 'rt', encoding='utf-8') as f:
        input = f.readline
        n, m = map(int, input().split())
        graph = create_graph(n, m, [input() for _ in range(m)],
directed=True)
    print(1 if has_negative_cycle(graph) else 0)

if __name__ == '__main__':
    main()
```

Текстовое объяснение решения:

Считываем числа и создаем словарь граф, если их 3 и граф взвешенный, то ключ это вершина, а значение это список вершин, к которым идет путь и Вес, иначе просто вершины к которым идет путь. directed флаг на ориентированный граф. Проходимся по всем вершинам графа и их смежным вершинам, обновляем расстояние до смежной вершины, если новый маршрут короче, чем текущее расстояние до этой вершины. Если в графе есть отрицательный цикл, то мы можем бесконечно уменьшатьрасстояние до вершины, входящей в цикл. Затем еще раз проходимся по всем вершинам и их смежным вершинам и проверяем, можно ли уменьшитьрасстояние до смежной вершины.

Результат работы кода на примерах из текста задачи:



Тестирование алгоритма:

	Время выполнения (seconds)	Затраты памяти (mbs)
Пример	0.014278411865234375	14. 23828125

Вывод по задаче: Мы проверели граф на содержание цикла с отрицательным суммарным весом.

Задача №14. Автобусы [1 s, 16 Mb, 3 балла]

- Текст задачи:
  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 входного файла (input.txt).
  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 выходного файла (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.

- Пример:

Input.txt	3 1 3 4 1 0 2 5 1 1 2 3 2 3 3 5 1 1 3 10
Output.txt	5

Листинг кода:

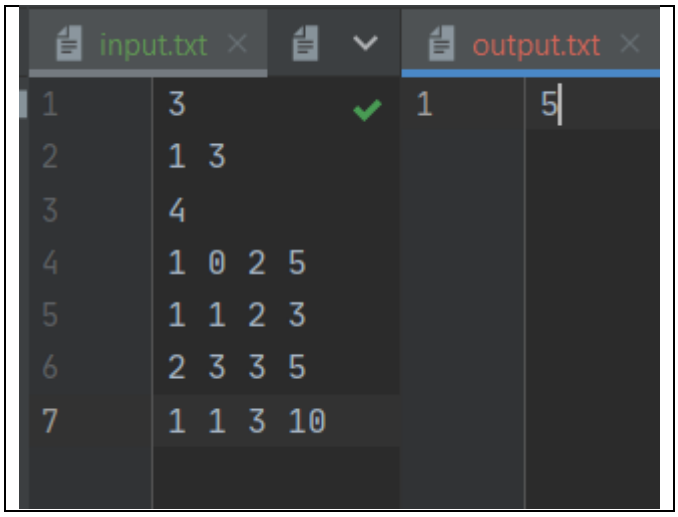
```
import tracemalloc
import time

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

f = open('input.txt')
num_villages = int(f.readline())
buses = [[] for _ in range(num_villages+1)]
start_village, end_village = map(int, f.readline().split())
num_routes = int(f.readline())
for i in range(num_routes):
    source_village, departure_time, dest_village, arrival_time = map(int,
f.readline().split())
    buses[source_village].append((departure_time, dest_village,
arrival_time))

INF = float('inf')
arrival_times = [INF] * (num_villages+1)
arrival_times[start_village] = 0
visited = [False] * (num_villages+1)

while True:
    min_time = INF
    for i in range(1, num_villages+1):
        if not visited[i] and arrival_times[i] < min_time:
            min_time = arrival_times[i]
            min_village = i
    if min_time == INF:
        break
    current_village = min_village
    visited[current_village] = True
    for departure_time, next_village, arrival_time in
buses[current_village]:
        if arrival_times[current_village] <= departure_time and
arrival_time <= arrival_times[next_village]:
            arrival_times[next_village] = arrival_time
t = open('output.txt', 'w')
if arrival_times[end_village] == INF:
    t.write('-1')
```

```
else:
    t.write(str((arrival_times[end_village])))
print("Время работы (в секундах):", time.perf_counter() - t_start)
print("Память %d, и пик %d" % tracemalloc.get_traced_memory())
```

Текстовое объяснение решения:

Это код на Python, который решает проблему нахождения минимального времени прибытия из начальной деревни в конечную, учитывая список маршрутов между деревнями. Код сначала считывает входные данные из файла с именем "input.txt ". Затем код инициализирует пустой список автобусов для каждой деревни и заполняет их маршрутами, считанными из входного файла. После этого код инициализирует список времени прибытия, при этом все значения устанавливаются равными бесконечности, за исключением начальной деревни, время прибытия которой равно 0. Код также инициализирует список посещенных деревень, для всех
значений которого установлено значение False. Затем код переходит в цикл, который повторяется до тех пор, пока не будут посещены все деревни. На каждой итерации он находит не посещенную деревню с наименьшим временем прибытия и помечает ее как посещенную. Затем он проверяет каждый автобус, отправляющийся из текущей деревни, чтобы узнать, может ли он увеличить время прибытия в деревню назначения. Если автобус может увеличить время прибытия, время прибытия в пункт назначения обновляется. Наконец, код записывает минимальное время прибытия конечной деревни в файл с именем "output.txt ". Если маршрута до конечной деревни нет, код записывает "-1" в выходной файл.

Результат работы кода на примерах из текста задачи:



19542322	14.06.2023 22:49:06	Волжева Мария Ильинична	0134	Python	Accepted		0,046	1674 Кб

Тестирование алгоритма:

	Время выполнения (seconds)	Затраты памяти (mbs)
Пример	0.0008377999999999997	14.1953125

Вывод по задаче: Написана программа для нахождения нужного автобуса для Марии Ивановны, на котором можно добраться из деревни d в деревню v как можно быстрее.

Задачи по выбору

Задача №1. Лабиринт [5 s, 512 Mb, 1 балл]

- Текст задачи:
  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.
  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.
  $2 \le n \le 10^3$, $1 \le m \le 10^3$, $1 \le u, v \le n$, $u \neq v$.
- Формат выходного файла (output.txt).
  Выведите 1, если есть путь между вершинами u и v; выведите 0, если пути нет.
- Пример:

Input.txt	4 4 1 2 3 2 4 3 1 4 1 4	4 2 1 2 3 2 1 4
Output.txt	1	0

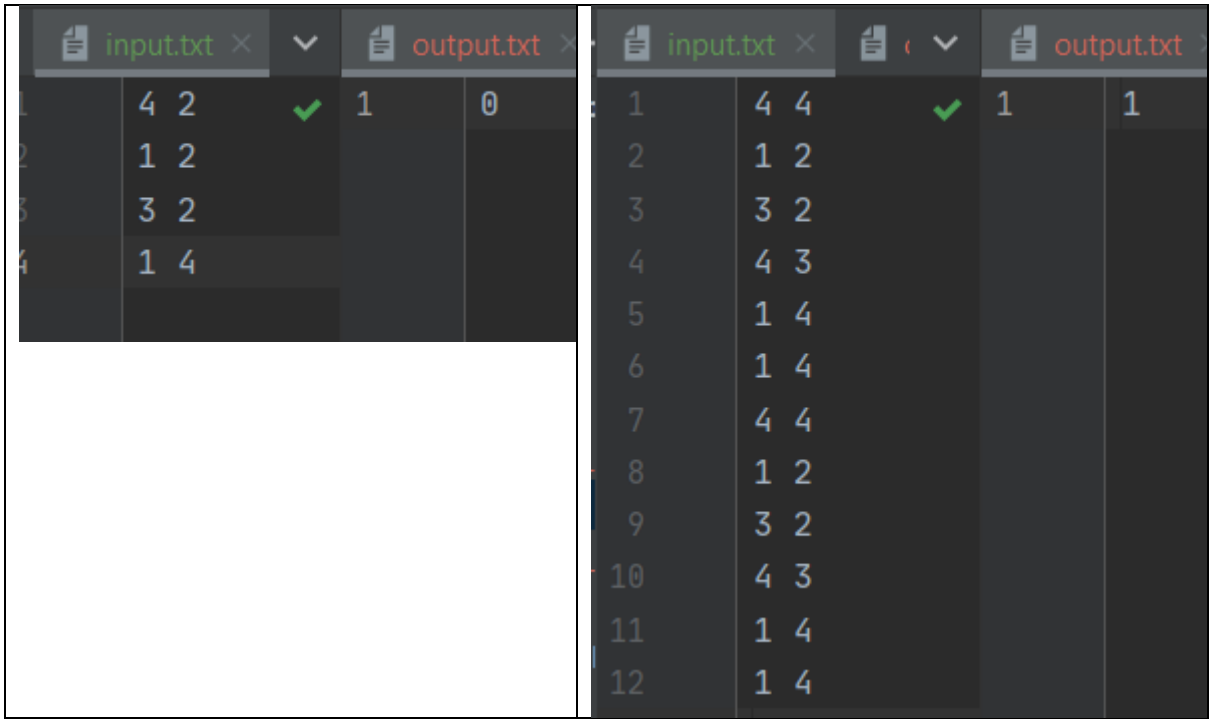
Листинг кода:

```
import time
import os, psutil


def existPath(count_edges, count_peaks, edges, u, v):
    mat = [[False for i in range(count_peaks)]
           for j in range(count_peaks)]

    for i in range(count_edges):
        mat[edges[i][0]][edges[i][1]] = True

    for k in range(count_peaks):
        for i in range(count_peaks):
            for j in range(count_peaks):
                mat[i][j] = (mat[i][j] or mat[i][k] and mat[k][j])

    if (u >= count_peaks or v >= count_peaks):
        return False

    if (mat[u][v]):
        return True

    return False


t_start = time.perf_counter()
process = psutil.Process(os.getpid())
f = open("input.txt")
m = open("output.txt", "w")
count_peaks, count_edges = f.readline().split()
edges = []
for each in range(int(count_edges)):
    edg1, edg2 = f.readline().split()
    edges.append([int(edg1) - 1, int(edg2) - 1])
    edges.append([int(edg2) - 1, int(edg1) - 1])
u, v = f.readline().split()
u, v = int(u) - 1, int(v) - 1


if existPath(int(count_edges) * 2, int(count_peaks), edges, u, v) or
existPath(int(count_edges) * 2, int(count_peaks), edges, v, u):
    m.write("1")
else:
    m.write("0")

f.close()
m.close()
print("Time of working: %s second" % (time.perf_counter() - t_start))
print("Memory", process.memory_info().rss/(1024*1024), "mb")
```

Текстовое объяснение решения:

Сначала мы записываем все данные в матрицу, где обозначено между какими вершинами есть ребра. Далее мы проходимся по всем вершинам и смотрим, есть ли среди них непосредственная связь или связь через какую-то вершину.

Результат работы кода на примерах из текста задачи:



Тестирование алгоритма:

	Время выполнения (seconds)	Затраты памяти (mbs)
Пример	0. 004554500000000017	14. 20703125
Пример	0.006016300000000002	

Вывод по задаче: Мы научились читать граф из файла и определять есть ли путь между собой любые две вершины.

Задача № 2. Компоненты [5 s, 512 Mb, 1 балл]

- Текст задачи:
  Теперь вы решаете сделать так, чтобы в лабиринте не было мертвых зон, то есть чтобы из каждой клетки был доступен хотя бы один выход. Для этого вы находите связные компоненты соответствующего неориентированного графа и следите за тем, чтобы каждый компонент содержал выходную ячейку. Дан неориентированный граф с n вершинами и m ребрами. Нужно посчитать количество компонент связности в нем.
- Формат ввода.
  Неориентированный граф с n вершинами и m ребрами по формату 1.

- Ограничения на входные данные.
  $1 \leq n \leq 10^3$, $0 \leq m \leq 10^3$
- Формат выходного файла (output.txt).
  Выведите количество компонент связности.
- Пример:

Input.txt	4 2 1 2 3 2
Output.txt	2

Листинг кода:

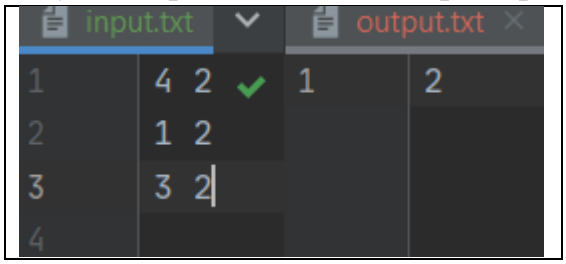
```
import time
import os, psutil


def ToDict(count_peaks, ribs):
    ribsDict = dict()
    for i in range(count_peaks):
        currRibs = []
        for j in range(len(ribs)):
            if ribs[j][0] == i + 1:
                currRibs.append(ribs[j][1])
            elif ribs[j][1] == i + 1:
                currRibs.append(ribs[j][0])
        ribsDict[i + 1] = currRibs
    return ribsDict


def DFS(start, verts):
    Visited[start] = True
    verts.append(start)
    for u in ribsDict[start]:
        if not Visited[u]:
            DFS(u, verts)
    return verts


t_start = time.perf_counter()
process = psutil.Process(os.getpid())
f = open("input.txt")
m = open("output.txt", "w")
count_peaks, count_ribs = f.readline().split()
ribs = []
for each in range(int(count_ribs)):
    rib1, rib2 = f.readline().split()
    ribs.append([int(rib1), int(rib2)])

ribsDict = ToDict(int(count_peaks), ribs)
Visited = [False] * (int(count_peaks) + 1)
comps = list()

for i in range(1, int(count_peaks) + 1):
    if not Visited[i]:
        comps.append(DFS(i, list()))
```

```
m.write(str(len(comps)))

f.close()
m.close()
print("Time of working: %s second" % (time.perf_counter() - t_start))
print("Memory", process.memory_info().rss/(1024*1024), "mb")
```

Текстовое объяснение решения:

Сначала мы заводим словарь, в котором записано из какой вершины в какую есть ребро. После этого мы запускаем обход в глубину из какой-то вершины, а далее из всех, которые мы не обошли в предыдущих обходах. Результат обхода сохраняем в массив – его длина и есть компонент связности.

Результат работы кода на примерах из текста задачи:



Тестирование алгоритма:

	Время выполнения (seconds)	Затраты памяти (mbs)
Пример	0. 0008977000000000013	14. 14453125

Вывод по задаче: Мы научились находить компоненту связности и обходить граф в глубину.

Задача №11. Алхимия [1 s, 16 Mb, 3 балла]

- Текст задачи:
  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 ввода / входного файла (input.txt). Первая строка входного файла INPUT.TXT содержит целое число m ($0 \le m \le 1000$) – 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.
- Формат выходного файла (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.
- Пример:

Input.txt	5 Aqua -> AquaVita AquaVita -> PhilosopherStone AquaVita -> Argentum Argentum -> Aurum AquaVita -> Aurum Aqua Aurum	5 Aqua -> AquaVita AquaVita -> PhilosopherStone AquaVita -> Argentum Argentum -> Aurum AquaVita -> Aurum Aqua Osmium
Output.txt	2	-1

Листинг кода:

```
import os, psutil
import time
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__':
    t_start = time.perf_counter()
    process = psutil.Process(os.getpid())
    f = open('input.txt', 'r')
    n = int(f.readline())
    elements_graph = dict()
    for i in range(n):
        inp_str = f.readline().strip().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 = f.readline().strip()
    to_elem = f.readline().strip()
    t = open('output.txt', 'w+')
    t.write(str(find_min_path(elements_graph, fr_element, to_elem)))


print("Time of working: %s second" % (time.perf_counter() - t_start))
print("Memory", process.memory_info().rss/(1024*1024), "mb")
```

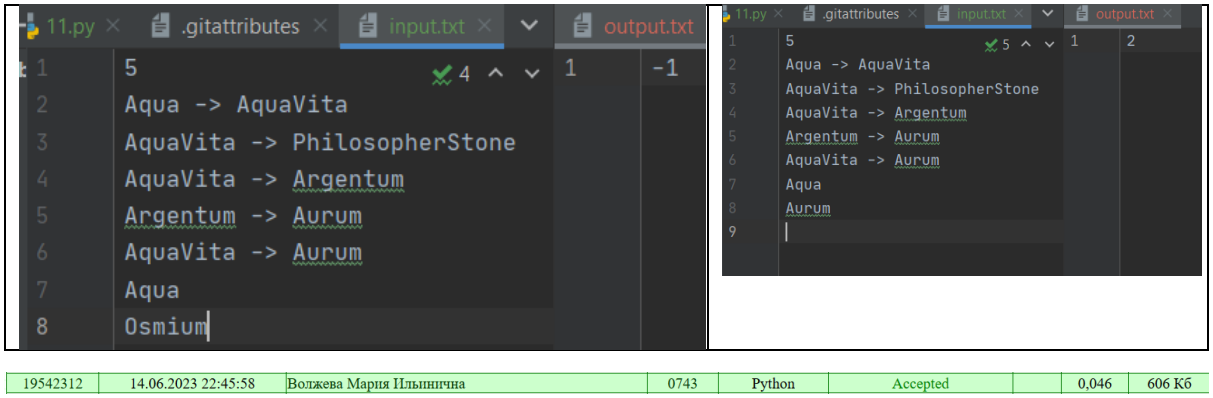
Текстовое объяснение решения:

Этот код считывается на графике, представленном в виде списка направленных ребер из входного файла, причем первая строка файла содержит количество ребер.

Далее код определяет функцию с именем find_min_path, которая принимаеттри аргумента: elements_graph, start и end. elements_graph - это словарь, представляющий граф, где каждый ключ является узлом, а каждое значение представляет собой список узлов, с которыми ключевой узел связан направленным ребром. start и end - это строки, представляющие начальный узел и конечный узел, соответственно. Функция find_min_path инициализирует словарь с именем len_map, который будет использоваться для отслеживания длины кратчайшего пути к каждому узлу. Он также инициализирует deque с именем queue_name, которое будет использоваться для отслеживания узлов, которые будут посещены при поиске в ширину. Он устанавливает длину кратчайшего пути к стартовому узлу равной 0 и

добавляет начальный узел в очередь. Затем функция переходит в цикл while, который продолжается до тех пор, пока очередь не опустеет. Внутри цикла он извлекает крайний левый узел из очереди и проверяет, является ли он конечным узлом. Если это так, то он возвращает длину кратчайшего путик этому узлу. Если текущий узел не является конечным узлом, он проверяет, находится ли этот узел на графике. Если это так, он выполняет итерацию поузлам, к которым подключен текущий узел, и проверяет, были ли они уже добавлены в len_map. Если они этого не сделали, он добавляет их в len_map с длиной, на единицу превышающей длину пути текущего узла. Это также добавляет их в очередь для посещения в будущем. Если конечный узел таки не достигнут, функция возвращает значение -1. После определения функции find_min_path код считывает данные из входного файла, создает словарь графа и выполняет считывание в начальном и конечном узлах. Затем он вызывает функцию find_min_path со словарем графа, начальнымузлом и конечным узлом в качестве аргументов и записывает результат ввыходной файл.

Результат работы кода на примерах из текста задачи:



19542312	14.06.2023 22:45:58	Волжева Мария Ильинична	0743	Python	Accepted		0,046	606 Кб

Тестирование алгоритма:

	Время выполнения (seconds)	Затраты памяти (mbs)
Пример	0. 0008553999999999923	14. 23828125
Пример	0.0013692000000000148	14.125

Вывод по задаче: Выяснено 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

Задача №13. Грядки [1 s, 16 Mb, 3 балла]

- Текст задачи:
  Прямоугольный садовый участок шириной N и длиной M метров разбит на квадраты со стороной 1 метр. На этом участке вскопаны грядки. Грядкой называется совокупность квадратов, удовлетворяющая таким условиям:
  • из любого квадрата этой грядки можно попасть в любой другой квадрат этой же грядки, последовательно переходя по грядке из квадрата в квадрат через их общую сторону;
  • никакие две грядки не пересекаются и не касаются друг друга ни по вертикальной, ни по горизонтальной сторонам квадратов (касание грядок углами квадратов допускается). Подсчитайте количество грядок на садовом участке.
- Формат ввода / входного файла (input.txt). В первой строке входного файла INPUT.TXT находятся числа N и M через пробел, далее идут N строк по M символов. Символ # обозначает территорию грядки, точка соответствует незанятой территории. Других символов в исходном файле нет ($1 \le N, M \le 200$)..
- Формат выходного файла (output.txt).
  В выходной файл OUTPUT.TXT выведите количество грядок на садовом участке.

Листинг кода:

```
import time
import tracemalloc


def count_connected_components(n_rows, n_cols, grid):
    visited = set()
    stack = []
    count = 0

    for i in range(n_rows * n_cols):
        if i not in visited and grid[i // n_cols][i % n_cols] == "#":
            count += 1
            stack.append(i)
            while stack:
                x = stack.pop()
                if x not in visited:
                    visited.add(x)
                    row, col = x // n_cols, x % n_cols
                    for dx, dy in ((0, 1),
                                   (0, -1),
                                   (1, 0),
                                   (-1, 0)):
                        nx, ny = row + dx, col + dy
                        if 0 <= nx < n_rows and 0 <= ny < n_cols and
```

```
grid[nx][ny] == "#":
                            stack.append(nx * n_cols + ny)

    return count


def main():
    t_start = time.perf_counter()
    tracemalloc.start()

    f = open("input.txt", "r")
    n_rows, n_cols = map(int, f.readline().split())
    grid = [f.readline().strip() for _ in range(n_rows)]


    count = count_connected_components(n_rows, n_cols, grid)
    t = open("output.txt", "w")
    t.write(str(count))


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


if __name__ == "__main__":
    main()
```

Текстовое объяснение решения:

Код считывается во входном файле, содержащем сетку из символов "#" и ".", представляющих стены и пустые пространства соответственно. Затем он подсчитывает количество подключенных компонентов в сетке, где подключенный компонент представляет собой группу символов "#", которые расположены рядом друг с другом либо по горизонтали, либо по вертикали. Функция count_connected_components принимает размеры сеткии саму сетку и возвращает количество подключенных компонентов. Он использует алгоритм поиска по глубине для обхода сетки и поиска связанных компонентов. Он инициализирует пустой набор посещенных ячеек для отслеживания посещенных ячеек и стек списков для отслеживания посещаемых ячеек. Он также инициализирует переменную count для отслеживания количества подключенных компонентов. Затем функция перебирает все ячейки в сетке, проверяя, является ли каждая ячейка непрочитанным символом "#". Если ячейка является непросмотренной "#", функция увеличивает переменную count, добавляет индекс ячейки в стек и вводит цикл while для посещения всех подключенных ячеек. В цикле while функция извлекает индекс ячейки из стека, проверяет, была ли она посещена, и если нет, помечает ее как посещенную, а затем добавляет ее непрошенных соседей в стек. Основная

функция считывает данные из входного файла, вызывает count_connected_components, чтобы получить количество связанных компонентов, и записывает результат в выходной файл.
Результат работы кода на примерах из текста задачи:

19542193	14.06.2023 22:16:34	Волжева Мария Ильинична	0432	Python	Accepted		0,078	4274 Кб

Вывод по задаче: Подсчитано количество грядок на садовом участке

Задача №16. Рекурсия [1 s, 16 Mb, 3 балла]

- Текст задачи:
  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 ввода / входного файла (input.txt). Первая строка входного файла INPUT.TXT содержит целое число n – количество процедур в программе ($1 \le n \le 100$). 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 ($k \le n$) – 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 выходного файла (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.

Листинг кода:

```
f = open('input.txt')
t = open('output.txt', 'w')
n = int(f.readline())
sides = {}
for i in range(n):
    v1 = f.readline()
    sides[v1] = []
    k = int(f.readline())
    for j in range(k):
        v2 = f.readline()
        sides[v1].append(v2)
    edge = f.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:
                t.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:
        t.write('NO\n')
```

Текстовое объяснение решения:

Этот код открывает файл "input.txt", считывает первую строку файла как число n. Затем в цикле for считывает n пар соединенных вершин и записывает их в словарь "sides". Каждая пара начинается с вершины v1, а затем содержит k соединенных с ней вершин. Затем происходит обход графа в цикле for, начиная с каждой вершины. Внутри цикла while происходит обход графа в ширину. Если граф обходится и находится начальная вершина графа u, записывается 'YES' в файл 'output.txt'. В

противном случае записывается 'NO'. Файл 'output.txt' открывается в режиме записи (необходимо создать файл с таким же именем), чтобы результат работы программы был сохранен в этот файл.
Результат работы кода на примерах из текста задачи:

19542349	14.06.2023 22:59:02	Волжева Мария Ильинична	0345	Python	Accepted		0,062	1658 Кб

Тестирование алгоритма:

	Время выполнения (seconds)	Затраты памяти (mbs)
Пример	0. 0008553999999999923	14. 23828125

Вывод по задаче: Написана программа, которая позволяет решить названную задачу

Задача №12. Цветной лабиринт [1 s, 16 Mb, 2 балла]

- Текст задачи:
  В одном из парков одного большого города недавно был организован новый аттракцион Цветной лабиринт. Он остоит из n комнат, соединенных m двунаправленными коридорами. Каждый из коридоров покрашен в один из ста цветов, при этом от каждой комнаты отходит не более одного коридора каждого цвета. При этом две комнаты могут быть соединены любым количеством коридоров. Человек, купивший билет на аттракцион, оказывается в комнате номер один. Кроме билета, он также получает описание пути, по которому он может выбраться из лабиринта. Это описани представляет собой последовательность цветов c1 . . . ck. Пользоваться ей надо так: находясь в комнате, надо посмотреть на очередной цвет в этой последовательности, выбрать коридор такого цвета и пойти по нему. При этом если из комнаты нельзя пойти по коридору соответствующего цвета, то человеку приходится дальше самому выбирать, куда идти. В последнее время в администрацию парка стали часто поступать жалобы от заблудившихся в лабиринте людей. В связи с этим, возникла необходимость написания программы, проверяющей корректность описания и пути, и, в случае ее корректности, сообщающей номер комнаты, в которую ведет путь Описание пути некорректно, если на пути, который оно описывает,

возникает ситуация, когда из комнаты нельзяпойти по коридору соответствующего цвета.

- Формат ввода / входного файла (input.txt). Первая строка входного файла INPUT.TXT содержит два целых числа n ($1 \le n \le 10000$) и m ($1 \le m \le 100000$) - соответственно количество комнат и коридоров в лабиринте. Следующие m строк содержат описания коридоров. Каждое описание содержит три числа u ($1 \le u \le n$), v ($1 \le v \le n$), c ($1 \le c \le 100$) - соответственно номера комнат, соединенных этим коридором, и цвет коридора. Следующая, (m + 2)-ая строка входного файла содержит длину описания пути - целое число k ($0 \le k \le 100000$). Последняя строка входного файла содержит k целых чисел, разделенных пробелами, - описание пути по лабиринту.
- Формат выходного файла (output.txt).
  В выходной файл OUTPUT.TXT выведите строку INCORRECT, если описание пути некорректно, иначе выведите номер комнаты, в которую ведет описанный путь. Помните, что путь начинается в комнате номер один.

Листинг кода:

```
import time
import tracemalloc

t = time.process_time()
tracemalloc.start()


def labirint(k, path, sides):
    room_cur = 1
    for i in range(k):
        color = path[i]
        flag = False
        for vertex, side in sides[room_cur]:
            if side == color:
                room_cur = vertex
                flag = True
                break
        if not flag:
            return 'INCORRECT'
    return str(room_cur)

f = open('input.txt')
n, m = map(int, f.readline().split())
sides = {}
for i in range(1, n+1):
    sides[i] = []
for _ in range(m):
    u, v, c = map(int, f.readline().split())
    sides[u].append([v, c])
    sides[v].append([u, c])
k = int(f.readline())
```

```
path = list(map(int, f.readline().split()))

d = open('output.txt', 'w')
d.write(labirint(k, path, sides))


print('Время работы: %s секунд' % (time.process_time() - t))
print('Затраты памяти:', float(tracemalloc.get_tracemalloc_memory()) / (2**
20), 'мб')
```

Текстовое объяснение решения:

Первоначально программа читает данные из файла input.txt, который содержит информацию о количестве комнат n, количестве стен m, информацию о стенах между комнатами sides, количество шагов в маршруте k и сам маршрут path. Затем функция labirint() проходит по каждому шагу в маршруте path и проверяет, существует ли стена определенного цвета (color) между текущей комнатой (room_cur) и соседней комнатой. Если такая стена существует, то программа переходит в соседнюю комнату (vertex) и продолжает проверять следующие шаги маршрута. Если же такой стены не существует, то программа завершается с выводом строки "INCORRECT". В конце функция возвращает номер комнаты, в которой находится человек после прохождения маршрута, в виде строки. Затем результат работы функции записывается в файл output.txt с помощью функции d.write().

Результат работы кода на примерах из текста задачи:

19542465	14.06.2023 23:43:04	Волжева Мария Ильинична	0601	Python	Accepted		0,125	12 Мб

Вывод по задаче: Написана программа, которая позволяет решить названную задачу

Задача №8. Стоимость полета [10 s, 512 Mb, 1.5 балла]

- Текст задачи:
  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.
- Формат выходного файла (output.txt).
  Выведите минимальный вес пути из u в v. Веведите -1, если пути нет.

Листинг кода:

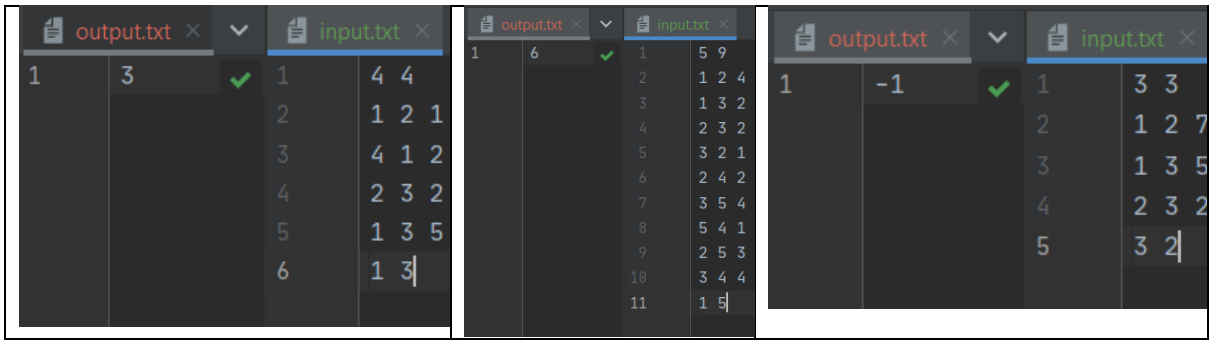
```
from collections import deque
import time
import tracemalloc

t = time.process_time()
tracemalloc.start()


def short_path(u, v):
    global sides
    min_path = float('inf')
    search_queue = deque()
    search_queue.append((u, 0, None))
    while search_queue:
        cur_node, path, parent = search_queue.popleft()
        if cur_node == v:
            if path < min_path:
                min_path = path
        else:
            for node in sides[cur_node]:
                if node[0] != parent:
                    search_queue.append((node[0], path + node[1],
cur_node))
    if min_path == float('inf'):
        return -1
    return min_path

f = open('input.txt')
n, m = map(int, f.readline().split())
sides = {}
for i in range(n+1):
    sides[i] = []
for i in range(m):
    v1, v2, l = map(int, f.readline().split())
    flag = False
    for node in sides[v1]:
        if node[0] == v2 and l < node[1]:
            node[1] = l
            flag = True
    if not flag:
        sides[v1].append([v2, l])
    for node in sides[v1]:
        if node[0] == v2 and l < node[1]:
            node[1] = lflag = False
        if flag:
            sides[v2].append([v1, l])
u, v = map(int, f.readline().split())
```

```
d = open('output.txt', 'w')
d.write(str(short_path(u, v)))


print('Время работы: %s секунд' % (time.process_time() - t))
print('Затраты памяти:', float(tracemalloc.get_tracemalloc_memory()) / (2**
20), 'мб')
```

Текстовое объяснение решения:

Для нахождения кратчайшего пути используется алгоритм поиска в ширину (Breadth-First Search), который запускается функцией "short_path(u, v)" и принимает на вход начальную и конечную вершины. Функция "short_path" ищет кратчайший путь между u и v, используя очередь "search_queue", в которую добавляются вершины, которые нужно проверить на следующей итерации. На каждой итерации из очереди извлекается первый элемент "cur_node", его родитель "parent" и длина пути "path". Если "cur_node" равен v, то проверяется, является ли найденный путь минимальным. Если да, то значение "min_path" обновляется. Если "cur_node" не равен v, то для каждой вершины, соседней с "cur_node" проверяется, была ли она уже посещена, и если нет, то она добавляется в очередь. Вес ребра между "cur_node" и его соседом прибавляется к длине пути "path", а "cur_node" становится родителем соседа. Если минимальный путь между u и v не найден, то функция возвращает -1. Если путь найден, то функция возвращает его длину

Результат работы кода на примерах из текста задачи:



Вывод по задаче: Мы нашли самый дешевый способ добраться из одной точки в другую.

Задача №7. Двудольный граф [10 s, 512 Mb, 1.5 балла]

- Текст задачи:

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

- Формат ввода / входного файла (input.txt). Неориентированный граф задан по формату 1.
- Формат выходного файла (output.txt).
  Выведите 1, если граф двудольный; и 0 в противном случае.
- Листинг кода:

```
from collections import deque
import time
import tracemalloc

t = time.process_time()
tracemalloc.start()


def half_graph(u):
    global sides, total_colors
    search_queue = deque()
    search_queue.append((u, 0))
    visited = []
    while search_queue:
        cur_node, color = search_queue.popleft()
        if cur_node not in visited:
            total_colors[cur_node] = color
            visited.append(cur_node)
            for node in sides[cur_node]:
                if color == 0:
                    search_queue.append((node, 1))
                else:
                    search_queue.append((node, 0))
        elif total_colors[cur_node] != color:
            return 0
    return 1


f = open('input.txt')
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)

total_colors = [None] * (n + 1)
d = open('output.txt', 'w')
d.write(str(half_graph(1)))

print('Время работы: %s секунд' % (time.process_time() - t))
```

```
print('Затраты памяти:', float(tracemalloc.get_tracemalloc_memory())
/ (2** 20), 'мб')
```

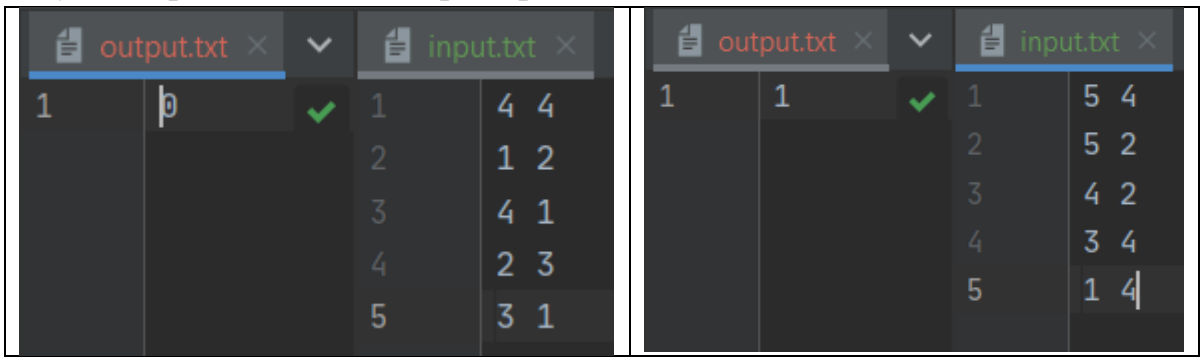
Текстовое объяснение решения:

Далее определен метод half_graph(u), который принимает на вход начальную вершину u и возвращает 1, если граф может быть раскрашен в два цвета, и 0 в противном случае. В данном методе используется структура данных deque из модуля collections, которая представляет собой двустороннюю очередь. В начале метода очередь инициализируется вершиной u и цветом 0.

Затем начинается обход графа в ширину: из очереди извлекается первый элемент, который представляет текущую вершину cur_node и цвет color. Если вершина еще не была посещена, то ей присваивается цвет color, посещенные вершины добавляются в список visited, и для каждой смежной с ней вершины node создается новый элемент очереди с вершиной node и цветом, отличным от цвета текущей вершины.

Если вершина уже была посещена, то проверяется, имеет ли она цвет, отличный от текущего. Если имеет, то граф невозможно раскрасить в два цвета, и метод возвращает 0. Если после обхода очереди граф можно раскрасить в два цвета, то метод возвращает 1. В конце кода вызывается метод half_graph с начальной вершиной 1 и выводится результат его работы

Результат работы кода на примерах из текста задачи:



Вывод по задаче: Мы научиличь определять является ли данный граф двудолным.

Вывод

В данной лабораторной работе были изучены основные алгоритмы графовой теории (поиск в глубину, поиск в ширину). Изучение этих алгоритмов позволяет понять различные методы обработки графов и находить оптимальные пути в них. Это важные инструменты для решения множества задач, связанных с графами, таких как планирование маршрутов, оптимизация транспортных сетей и многое другое