● 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 3

Выполнила:
Еремеева А.В.
К3244(13.2)

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета **2**

Задачи по варианту **3**
Задача №3. Циклы (1 балл) 3
Задача №12. Цветной лабиринт (2 балла) 6
Задача №13. Грядки (3 балла) 9

Дополнительные задачи **12**
Задача №5. Проверка сбалансированности(1.5 балла) 12
Задача №7. Двудольный граф(1.5 балла) 14

Вывод **17**

Задачи по варианту

Задача №3. Циклы(1 балл)

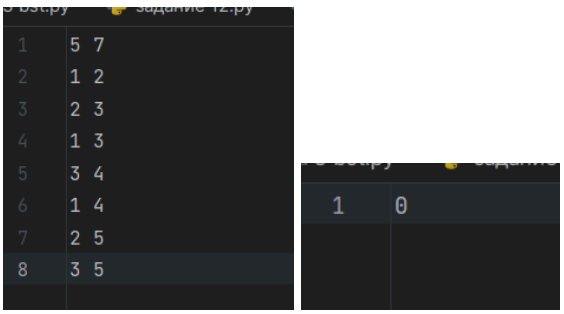
Учебная программа по инфокоммуникационным технологиям определяет пререквизиты для каждого курса в виде списка курсов, которые необходимо пройти перед тем, как начать этот курс. Вы хотите выполнить проверку согласованности учебного плана, то есть проверить отсутствие циклических зависимостей. Для этого 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.
Листинг кода:

```
def dfs_cycle(graph, u, visited, visiting):
   visited[u] = True  # Помечаем узел как посещенный
   visiting[u] = True  # Помечаем узел как находящийся в процессе обхода

   for neighbor in graph[u]:
       if not visited[neighbor]:  # Если соседний узел не был посещен
             if dfs_cycle(graph, neighbor, visited, visiting):  # Рекурсивно
проверяем его
               return True
         elif visiting[neighbor]:  # Если соседний узел находится в процессе
обхода
           return True  # Найден цикл

   visiting[u] = False  # Убираем узел из списка текущего обхода
   return False  # Циклов не найдено



with open("input.txt", "r") as input_file, open("output.txt", "w") as
output_file:
        n, m = map(int, input_file.readline().split())  # Читаем количество
курсов и зависимостей
       graph = [[] for _ in range(n)]  # Создаем граф

       for _ in range(m):
           u, v = map(int, input_file.readline().split())
           graph[u - 1].append(v - 1)  # Добавляем ребро в граф (u перед v)

       visited = [False] * n  # Список посещенных узлов
       visiting = [False] * n  # Список узлов в текущем обходе
```

```
    has_cycle = False  # Флаг для проверки наличия цикла
    for node in range(n):
        if not visited[node]:  # Если узел еще не посещен
            if dfs_cycle(graph, node, visited, visiting):
                has_cycle = True
                break  # Цикл найден

    # Записываем результат в файл
    output_file.write('1' if has_cycle else '0')
```

Текстовое объяснение решения: dfs_cycle выполняет обход в глубину (DFS) графа, начиная с узла u, чтобы проверить наличие цикла.Аргументы которые передаются в функцию: graph–список смежности, представляющий граф,u–текущий узел, который мы рассматриваем,visited–список, который указывает, был ли узел уже посещен,visiting–список, который указывает, находится ли узел в процессе обхода.visited[u] = True Устанавливаем значение True для visited[u], что означает, что узел u теперь считается посещённым.Устанавливаем значение True для visiting[u], что означает, что мы находимся в процессе обхода узла u.Начинаем цикл по всем соседям текущего узла u в графе.Проверяем, был ли соседний узел neighbor посещён.Если сосед не был посещён, рекурсивно вызываем dfs_cycle для этого соседа. Если обнаружен цикл, функция возвращает True.Если соседний узел уже находится в процессе обхода, это означает, что мы нашли цикл.Когда все соседи узла u обработаны, мы убираем u из visiting, так как он больше не находится в текущем пути обхода.
python

Результат работы кода на значениях из задачи:



	Время выполнения	Затраты памяти
Пример из задачи	0.00 сек.	23.3 Mib
Верхняя граница диапазона значений входных данных из текста задачи(1000000)	2.1 сек.	77 Mib

Вывод по задаче: программа проверяет отсутствие циклических зависимостей

Задача №12.Цветной лабиринт (2 балла)

В одном из парков одного большого города недавно был организован новый аттракцион Цветной лабиринт. Он состоит 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 описание 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 нельзя пойти по коридору соответствующего цвета.

• Формат входных данных (input.txt) и ограничения. 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.

Листинг кода:

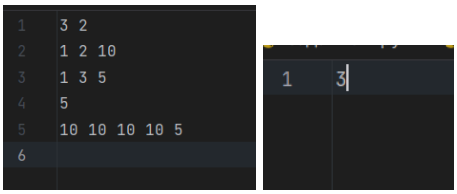
```
def labyrinth(graph, colors):
    # Начинаем с вершины 1
    node = 1
    for color in colors:
        # Проверяем, есть ли ребро текущего цвета из текущей вершины
        if color in graph[node]:
            node = graph[node][color]  # Переходим в следующую вершину
```

```
        else:
                return 'INCORRECT'   # Если переход невозможен, возвращаем
INCORRECT
    return node  # Возвращаем конечный узел после всех переходов


# Чтение входных данных
with open("input.txt", "r", encoding='utf-8') as input_file,
open("output.txt", "w", encoding='utf-8') as output_file:
    # Считываем количество вершин (n) и рёбер (m)
    n, m = map(int, input_file.readline().split())

     # Инициализируем граф как словарь, где ключ — вершина, а значение —
словарь соседей с цветами рёбер
    graph = {i: {} for i in range(1, n + 1)}

    # Считываем m рёбер
    for _ in range(m):
        u, v, c = map(int, input_file.readline().split())
            # Добавляем ребро с цветом c между вершинами u и v в обоих
направлениях
        graph[u][c] = v
        graph[v][c] = u

    # Считываем последовательность цветов
    colors = list(map(int, input_file.readline().split()))

    # Вычисляем результат работы функции и записываем его в файл
    result = labyrinth(graph, colors)
    output_file.write(str(result))
```

Текстовое объяснение решения:
Программа сначала открывает файл input.txt, который содержит описание графа и последовательность цветов рёбер, по которым нужно пройти.
Первая строка файла содержит два числа n и m:
n — количество вершин (или узлов) в графе.
m — количество рёбер (связей между узлами).
Следующие m строк содержат описание рёбер, где:
u и v — это номера вершин, которые соединяются ребром.
c — это цвет ребра, соединяющего вершины u и v. Для каждой строки u v c мы обновляем граф, добавляя связь от u к v и от v к u с цветом c, так как рёбра неориентированные (двунаправленные).Последняя строка файла содержит последовательность цветов, по которым нужно двигаться по графу.Построение графа строим граф в виде словаря, где ключ — это

номер вершины, а значение — это другой словарь, представляющий соседние вершины и соответствующие цвета рёбер между ними
Функция поиска пути (labyrinth)начинает движение из вершины 1.
Идем по графу, проверяя, есть ли ребро с текущим цветом из текущей вершины:Если такое ребро существует, мы переходим к следующей вершине.Если такого ребра нет (то есть ребро нужного цвета отсутствует), функция возвращает "INCORRECT".
Если после обработки всей последовательности цветов мы прошли весь путь, функция возвращает номер конечной вершины, в которой мы оказались.
После выполнения функции программа записывает результат в файл output.txt. Результатом будет либо номер конечной вершины, если путь существует, либо "INCORRECT", если пройти по заданной последовательности цветов невозможно.

Результат работы кода на примерах из задачи:



	Время выполнения	Затраты памяти
Примеры из задачи()	0.00сек.	22.8 Mib
Верхняя граница диапазона значений входных данных из текста задачи(1000000)	1.91 сек.	55.2Mib

Вывод по задаче: программа осуществляет поиск пути в лабиринте, представленном в виде графа, где вершины — это узлы (например, перекрестки), а рёбра (связи между узлами) имеют определённый цвет.

Задача №13. Задача. Грядки (3 балла)

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.

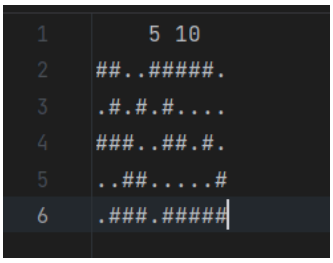
Формат входных данных (input.txt) и ограничения. В первой строке входного файла INPUT.TXT находятся числа N и M через пробел, далее идут N строк по M символов. Символ # обозначает территорию грядки, точка соответствует незанятой территории. Других символов в исходном файле нет ($1 \le N, M \le 200$)

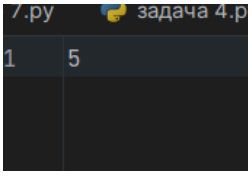
Листинг кода:

```
def garden_count(graph, n, m):
    counter = 0

    def dfs(x, y):
        """Рекурсивный поиск в глубину (DFS) для обработки текущей грядки."""
        stack = [(x, y)]
        while stack:
            x, y = stack.pop()
            if graph[x][y] == '#':
                graph[x][y] = '.'  # Помечаем клетку как посещённую
                # Добавляем соседние клетки в стек, если они внутри границ
                if x > 0:
                    stack.append((x - 1, y))
                if y > 0:
                    stack.append((x, y - 1))
                if x < n - 1:
                    stack.append((x + 1, y))
                if y < m - 1:
                    stack.append((x, y + 1))

    for i in range(n):
        for j in range(m):
            if graph[i][j] == '#':  # Начинаем поиск с новой грядки
                counter += 1
                dfs(i, j)  # Обрабатываем всю грядку с помощью DFS

    return counter



with open('input.txt', "r") as file, open("output.txt", "w") as output_file:
        n, m = map(int, file.readline().split())  # Читаем размеры сада
        garden = [list(file.readline().strip()) for _ in range(n)]  # Читаем садовую карту
        result = garden_count(garden, n, m)
        output_file.write(str(result))
```

Текстовое объяснение решения:
Проходим по каждой клетке сада и проверяем, является ли она частью новой грядки. Если клетка содержит символ #, увеличиваем счётчик грядок на 1 и запускаем поиск DFS (функция dfs), чтобы пометить все клетки этой грядки как посещённые, заменив их на.. Когда находим клетку с символом #, используем стек для хранения клеток, которые нужно проверить.Затем извлекаем клетку из стека и проверяем её соседей. Если соседняя клетка также содержит #, мы добавляем её в стек для дальнейшего исследования.Так мы помечаем все клетки этой грядки как посещённые.
Таким образом, перебираем каждую клетку в саду. Если находим новую грядку (клетку с #), то запускаем DFS для этой клетки и увеличиваем счётчик грядок.

Результат работы кода на значениях из задачи:





Результат проверки на acmp

[Вернуться к задаче] [Редактировать решение]

```python
def garden_count(graph, n, m):
    counter = 0

    def dfs(x, y):
        """Рекурсивный поиск в глубину (DFS) для обработки текущей грядки."""
        stack = [(x, y)]
        while stack:
            x, y = stack.pop()
            if graph[x][y] == '#':
                graph[x][y] = '.'  # Помечаем клетку как посещённую
                # Добавляем соседние клетки в стек, если они внутри границ
                if x > 0:
                    stack.append((x - 1, y))
                if y > 0:
                    stack.append((x, y - 1))
                if x < n - 1:
                    stack.append((x + 1, y))
                if y < m - 1:
                    stack.append((x, y + 1))

    for i in range(n):
        for j in range(m):
            if graph[i][j] == '#':  # Начинаем поиск с новой грядки
                counter += 1
                dfs(i, j)  # Обрабатываем всю грядку с помощью DFS

    return counter


with open('input.txt', "r") as file, open("output.txt", "w") as output_file:
        n, m = map(int, file.readline().split())  # Читаем размеры сада
        garden = [list(file.readline().strip()) for _ in range(n)]  # Читаем садовую карту
        result = garden_count(garden, n, m)
        output_file.write(str(result))
```

Тест	Результат	Время	Память
1	Accepted	0,031	498 Кб
2	Accepted	0,015	494 Кб
3	Accepted	0,031	506 Кб
4	Accepted	0,062	922 Кб
5	Accepted	0,046	1878 Кб
6	Accepted	0,046	1878 Кб
7	Accepted	0,062	3638 Кб
8	Accepted	0,093	2142 Кб
9	Accepted	0,046	2242 Кб
10	Accepted	0,062	914 Кб
11	Accepted	0,046	914 Кб
12	Accepted	0,015	498 Кб

Размер кода: 764

Посылки решений:

ID	Дата	Язык	Результат	Тест	Время	Память
22005384	28.09.2024 9:18:10	Python	Accepted		0,093	3638 Кб
22005181	28.09.2024 8:45:18	Python	Runtime error	1		474 Кб

	Время выполнения	Затраты памяти
Пример из задачи	0.093сек.	3638 Кб
Верхняя граница диапазона значений входных данных из текста задачи(1000)	1.99сек.	67.4Mib

Вывод по задаче: программа обходит все клетки сада и используем DFS для поиска всех клеток, связанных с каждой грядкой.

Дополнительные задачи

Задача №5. Проверка сбалансированности(1.5 балла)

Департамент полиции города сделал все улицы односторонними. Вы хотели бы проверить, можно ли законно проехать с любого перекрестка на какой-либо другой перекресток. Для этого строится ориентированный граф: вершины – это перекрестки, существует ребро (u, v) всякий раз, когда в городе есть улица (с односторонним движением) из u в v. Тогда достаточно проверить, все ли вершины графа лежат в одном компоненте сильной связности. Нужно вычислить количество компонентов сильной связности заданного ориентированного графа с n вершинами и m ребрами.
• Формат ввода / входного файла (input.txt). Ориентированный граф с n вершинами и m ребрами по формату 1

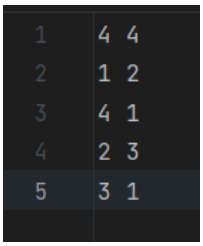
Листинг кода:

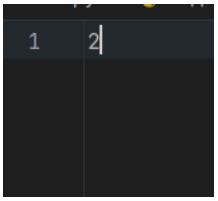
```
# Чтение входных данных
with open('input.txt', 'r') as file_input:
      n, m = map(int, file_input.readline().split())  # Читаем количество вершин и рёбер
      order = []  # Порядок завершения вершин
      visited = [False for _ in range(n)]  # Массив для отслеживания посещённых вершин
      g = [[] for _ in range(n)]  # Граф
      g_reverse = [[] for _ in range(n)]  # Обратный граф

      # Чтение рёбер
      for _ in range(m):
          a, b = map(int, file_input.readline().split())
          g[a - 1].append(b - 1)  # Обычный граф
          g_reverse[b - 1].append(a - 1)  # Обратный граф

      # Первый проход DFS: добавляем вершины в порядок по завершению их обработки
      def dfs1(u: int) -> None:
          visited[u] = True
          for neighbor in g[u]:
              if not visited[neighbor]:
                  dfs1(neighbor)
          order.append(u)

      # Второй проход DFS по обратному графу
      def dfs2(v: int) -> None:
          visited[v] = True
          for neighbor in g_reverse[v]:
              if not visited[neighbor]:
                  dfs2(neighbor)

      # Первый проход: выполняем DFS для каждой непосещенной вершины
      for i in range(n):
          if not visited[i]:
              dfs1(i)

      # Сброс visited перед вторым проходом
      visited = [False for _ in range(n)]
      k = 0  # Счётчик компонентов сильной связности

      # Второй проход: обработка вершин в порядке убывания их завершения
      for i in range(n):
          v = order[n - 1 - i]  # Берем вершины в обратном порядке завершения DFS
          if not visited[v]:
              k += 1  # Увеличиваем счётчик компонент сильной связности
              dfs2(v)
```

```
    # Запись результата в выходной файл
    with open('output.txt', 'w') as file_output:
        file_output.write(str(k))
```

Текстовое объяснение решения:
Читаются числа n (количество вершин) и m (количество рёбер).Создается order — список, в который будем сохранять порядок завершения вершин при первом DFS.visited — массив, чтобы отслеживать посещённые вершины.g — список смежности для исходного графа.g_reverse — список смежности для обратного графа (где все рёбра поменяны местами).
Заполняем графы каждое ребро (a, b) добавляется в список смежности исходного графа g как ребро из a в b, и в обратный граф g_reverse как ребро из b в a.
Далее мы начинаем обход графа с вершины u и помечаем её как посещённую.Для каждой соседней вершины, если она ещё не была посещена, запускается рекурсивный вызов dfs1.Когда все соседи посещены и текущая вершина обработана, её номер добавляется в список order (список завершения). Этот список поможет нам при втором проходе DFS по обратному графу.
Для каждой вершины, если она ещё не была посещена, мы запускаем первый DFS. В результате все вершины будут посещены, и в список order добавятся вершины в порядке их завершения.
После первого DFS все вершины помечены как посещённые, поэтому перед вторым DFS мы сбрасываем массив visited, чтобы начать с чистого листа для обратного графа.
Это аналогичная функция DFS2, но теперь она запускается по обратному графу g_reverse. Она позволяет найти компоненты сильной связности, так как обрабатывает вершины в порядке, обратном их завершению в первом DFS.В этой части мы обходим вершины в порядке, обратном их завершению в первом DFS (по списку order).
Если вершина ещё не посещена, мы запускаем второй DFS по обратному графу. Каждый запуск второго DFS на новую вершину означает, что мы нашли новую компоненту сильной связности, поэтому увеличиваем счётчик k .В этой части мы обходим вершины в порядке, обратном их завершению в первом DFS (по списку order).
Если вершина ещё не посещена, мы запускаем второй DFS по обратному графу. Каждый запуск второго DFS на новую вершину означает, что мы

нашли новую компоненту сильной связности, поэтому увеличиваем счётчик k.
Результат работы кода на значениях из задачи:





	Время выполнения	Затраты памяти
Пример из задачи	0.00сек.	22Mib
Верхняя граница диапазона значений входных данных из текста задачи(100000)	1.67сек.	77.7Mib

Вывод по задаче: программа вычисляет количество компонентов сильной связности заданного ориентированного графа с n вершинами и m ребрами

Задача №7. Двудольный граф (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
Листинг кода:

```
from collections import deque
```

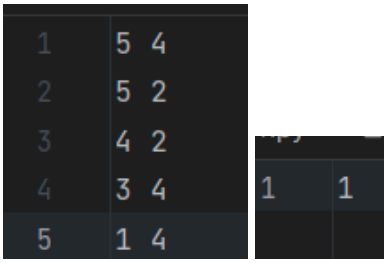
```
def is_bipartite(graph, n):
    color = [-1] * n  # Массив для хранения цветов (-1 означает, что вершина не
окрашена)

    def bfs(start):
        queue = deque([start])
        color[start] = 0  # Красим первую вершину в цвет 0

        while queue:
            u = queue.popleft()
            for v in graph[u]:
                if color[v] == -1:  # Если вершина ещё не окрашена
                    color[v] = 1 - color[u]  # Красим её в противоположный цвет
                    queue.append(v)
                elif color[v] == color[u]:  # Если сосед имеет тот же цвет
                    return False  # Граф не двудольный

        return True

    for i in range(n):
        if color[i] == -1:  # Если вершина не посещена
            if not bfs(i):  # Если не удалось раскрасить граф
                return False  # Граф не двудольный

    return True  # Граф двудольный


# Чтение графа из файла
with open("input.txt", "r") as input_file,open("output.txt", "w") as output_file:
    n, m = map(int, input_file.readline().split())
    graph = [[] for _ in range(n)]

    for _ in range(m):
        u, v = map(int, input_file.readline().split())
        graph[u - 1].append(v - 1)
        graph[v - 1].append(u - 1)  # Граф неориентированный

    # Проверяем, является ли граф двудольным
    if is_bipartite(graph, n):
        output_file.write(str(1))
    else:
        output_file.write(str(0))
```

Текстовое объяснение решения:

Функция is_bipartite принимает два аргумента: graph, представляющий собой граф в виде списка смежности, и n, количество вершин в графе.

Создаётся массив color длиной n, который инициализируется значениями -1, что указывает на то, что ни одна из вершин не была окрашена.Внутри is_bipartite объявляется вложенная функция bfs, которая реализует алгоритм поиска в ширину для окрашивания графа.Создаём очередь queue, в которую помещаем начальную вершину start и окрашиваем её в цвет 0.Запускаем основной цикл, который будет выполняться до тех пор, пока

очередь не станет пустой. В каждой итерации мы извлекаем вершину u из очереди. Для каждой соседней вершины v из списка смежности graph[u]:
Если вершина v не окрашена (== -1), мы окрашиваем её в противоположный цвет относительно u и добавляем в очередь.Если соседняя вершина v уже окрашена и имеет тот же цвет, что и u, это означает, что граф не является двудольным, и поэтому возвращаем False.В противном случае когда все соседи были успешно окрашены и не было конфликтов, функция bfs завершает работу, возвращая True. Далее в другом цикле проходим по всем вершинам графа. Если вершина ещё не была окрашена (color[i] == -1), мы запускаем функцию bfs для этой вершины.Если bfs возвращает False, то и сама функция is_bipartite также возвращает False следовательно граф не является двудольным.Но если все вершины пройдены и не было конфликтов, возвращаем True, что означает, что граф является двудольным.
Результат работы кода на значениях из задачи:



	Время выполнения	Затраты памяти
Пример из задачи	0.00сек.	22Mib
Верхняя граница диапазона значений входных данных из текста задачи(500000)	1.88сек.	98.7Mib

Вывод по задаче: программа делает проверку двудольности графа, используя алгоритм BFS для окрашивания графа двумя цветами.

Вывод: Решая эту лабораторную, я научилась работать со графами