МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ И СИСТЕМ СВЯЗИ

КУРСОВАЯ РАБОТА (ПРОЕКТ)
ЗАЩИЩЕНА С ОЦЕНКОЙ

РУКОВОДИТЕЛЬ

ассистент		А. Н. Головенков
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОЙ РАБОТЕ

ТОПОЛОГИЧЕСКАЯ СОРТИРОВКА

по дисциплине: Основы программирования

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ гр. № 2451		М. А. Никитин
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2025

ОГЛАВЛЕНИЕ

1. ВВЕДЕНИЕ

Классической моделью для решения подобных задач является ориентированный ациклический граф, вершины которого соответствуют отдельным действиям, а дуги — отношениям зависимости между ними. В таких графах существует по крайней один линейный порядок вершин, при котором все зависимости соблюдаются; нахождение такого порядка называется топологической сортировкой.

В качестве наглядной аналогии можно рассмотреть процесс получения справок у нескольких чиновников, каждый из которых требует предоставления определённых документов от других. Если в системе зависимостей нет циклических ссылок (что сделало бы выполнение задачи невозможным), то всегда можно построить последовательный план обращения, гарантирующий, что все необходимые справки будут собраны вовремя. Именно эту ситуацию формально описывает задача топологической сортировки.

Целью данной курсовой работы является разработка программы, реализующей алгоритм топологической сортировки для ориентированного ациклического графа, заданного матрицей смежности. Основное внимание уделяется алгоритму на основе поиска в глубину (DFS), который эффективно строит порядок вершин за время O(|V| + |E|). Результатом работы программы является не только текстовый список вершин в топологическом порядке, но и файл в формате Graphviz, позволяющий наглядно визуализировать структуру графа и полученную последовательность.

Актуальность работы обусловлена широкой применимостью топологической сортировки в компьютерных науках и смежных областях, а также важностью понимания базовых алгоритмов работы с графами для будущих специалистов в области программирования и анализа данных. Практическая ценность заключается в создании инструмента, который может быть использован для анализа зависимостей в реальных системах, учебных

демонстраций и решения задач, связанных с упорядочиванием действий в условиях ограничений.

2. ПОСТАНОВКА ЗАДАЧИ

Задачей данной курсовой работы является разработка программы, реализующей алгоритм топологической сортировки ориентированного ациклического графа (DAG).

Алгоритм, лежащий в основе решения, основан на методе поиска в глубину (DFS). Топологический порядок вершин формируется в порядке убывания времени выхода из вершин при обходе в глубину: вершина добавляется в результирующий список в тот момент, когда обработка всех её потомков завершена. Этот подход позволяет за один обход графа построить корректный линейный порядок вершин для ациклического графа. Процесс включает в себя следующие этапы: представление графа в памяти на основе входной матрицы смежности, рекурсивный обход в глубину, формирование результата сортировки и генерация визуального представления графа с использованием языка DOT.

Требования к решению: Программа должна корректно считывать ориентированный граф из файла в виде матрицы смежности. Для ациклического графа программа должна формировать один из возможных топологических порядков его вершин и записывать результат в выходной файл. Необходимо провести теоретический анализ алгоритма, включая оценку его вычислительной сложности O(|V| + |E|) и сравнение с альтернативным алгоритмом Кана. Реализация должна автоматически генерировать файл в формате Graphviz для визуализации структуры исходного графа.

3. АЛГОРИТМ

3.1. Основные идеи алгоритма

Идеи, положенные в основу алгоритма:

1. Сведение задачи упорядочивания к обходу графа — задача нахождения корректной последовательности вершин сводится к систематическому обходу всех вершин графа, при котором каждая вершина обрабатывается после всех её зависимостей.
2. Использование времени выхода из вершин — ключевое наблюдение: если рассматривать момент завершения обработки всех потомков вершины (время выхода из неё при DFS), то для любой дуги (u, v) время выхода из u будет больше времени выхода из v. Таким образом, вершины, отсортированные в порядке убывания времени выхода, образуют топологический порядок.
3. Эффективность поиска в глубину — алгоритм за один обход графа строит итоговый порядок вершин, что обеспечивает высокую эффективность.
4. Структуры данных и математический аппарат:

- V — множество вершин (чиновников/справок), $|V| = n$
- E — множество дуг (зависимостей), $|E| = m$
- Дуга $(u, v) \in E$ означает, что справка u требуется для получения справки v.

Граф может быть представлен матрицей A размером $n \times n$, где:

A[i][j] = 1, если существует дуга из вершины i в вершину j

A[i][j] = 0, в противном случае

3.1.Псевдокод

```
 Input: adjacencyList – array of arrays (adjacency list
representation of graph), numVertices – int
Output: out – int array (topological order of vertices)
   if numVertices = 0:
              return []
      end if
used ← bool array of size numVertices, all false  /* массив для
отслеживания посещенных вершин */
topologicalAnswer ← int array (empty)       /* результат
топологической сортировки */
      for i = 0 … numVertices - 1:    /* обходим все вершины графа */
          if not used[i]:             /* если вершина еще не посещена */
             dfs(i, topologicalAnswer, used, adjacencyList)  /*
запускаем DFS от этой вершины */
          end if
      end for
      reverse topologicalAnswer          /* разворачиваем порядок */
out ← int array of size length(topologicalAnswer)
      for i = 0 … length(topologicalAnswer) - 1:
            out[i] ← topologicalAnswer[i]
      end for
      if length(out) = 0:
          return []                /* пустой граф */
      end if
return out
Function: dfs (helper function for TopologicalSort)
Input: v – int (current vertex), topologicalAnswer – int array
(reference), used – bool array (reference), adjacencyList –
array of arrays (reference)

      used[v] ← true                                    /* помечаем
вершину как посещенную */
```

```
      for each u in adjacencyList[v]:        /* обходим всех соседей
текущей вершины */
    if not used[u]:              /* если сосед еще не посещен */
                dfs(u, topologicalAnswer, used, adjacencyList)  /*
рекурсивно обрабатываем соседа */
           end if
      end for
      append v to topologicalAnswer          /* добавляем вершину в
результат после обработки всех потомков */
```

3.2. Шаги алгоритма

1. Инициализация:

- Создать массив used размера n, инициализированный значениями false.
- Создать пустой массив topologicalAnswer.

2. Построение топологического порядка:

- Для каждой непосещенной вершины v запустить dfs(v):
- Пометить v как посещенную (used[v] = true).
- Рекурсивно обработать всех непосещенных соседей v.
- После обработки всех потомков добавить v в topologicalAnswer.

3. Формирование результата:

- Развернуть массив topologicalAnswer (получить обратный порядок).
- Результат содержит вершины в топологическом порядке.

4. Генерация визуализации:

Записать граф и порядок в файл формата Graphviz (DOT).

Обоснование корректности:

Вершина добавляется в topologicalAnswer после обработки всех её потомков. Для любой дуги (u, v) вершина v добавляется раньше u. После разворота получаем порядок, где u идет раньше v — это и есть топологический порядок.

Вычислительная сложность:

- Время: O(|V| + |E|)
- Память: O(|V| + |E|)

3.3. Пошаговое выполнение на примере

Получим справки у 4 чиновников (0, 1, 2, 3)

Ограничения:

- Чиновник 0: справок не требуется

- Чиновник 1: требуется справка от чиновника 0

- Чиновник 2: требуется справка от чиновника 1

- Чиновник 3: требуется справка от чиновника 2

Граф зависимостей:

Чиновник 0 → Чиновник 1 → Чиновник 2 → Чиновник 3

Матрица смежности (1 = нужна справка):

  0 1 2 3

0 [0 1 0 0] (для чиновника 1 нужна справка 0)

1 [0 0 1 0] (для чиновника 2 нужна справка 1)

2 [0 0 0 1] (для чиновника 3 нужна справка 2)

3 [0 0 0 0] (для чиновника 3 ничего не нужно)

Шаг 1: Инициализация

Создаем список посещенных чиновников:

visited = [нет, нет, нет, нет]

Создаем список порядка завершения (когда все справки получены):

order = []

Шаг 2: Начинаем с чиновника 0

Проверяем чиновника 0: еще не посещен

Записываем в журнал: "Посетили чиновника 0"

visited = [да, нет, нет, нет]

Проверяем зависимости чиновника 0:

Для получения справки от чиновника 0 нужны справки от: [чиновник 1]

(Это означает, что есть дуга 0 → 1, т.е. справка 0 нужна для справки 1)

Проверяем чиновника 1: еще не посещен

Записываем: "Переходим к чиновнику 1"

Шаг 3: Обрабатываем чиновника 1

Записываем в журнал: "Посетили чиновника 1"

visited = [да, да, нет, нет]

Проверяем зависимости чиновника 1:

Для получения справки от чиновника 1 нужны справки от: [чиновник 2]

(Дуга 1 → 2: справка 1 нужна для справки 2)

Проверяем чиновника 2: еще не посещен

Записываем: "Переходим к чиновнику 2"

Шаг 4: Обрабатываем чиновника 2

Записываем в журнал: "Посетили чиновника 2"

visited = [да, да, да, нет]

Проверяем зависимости чиновника 2:

Для получения справки от чиновника 2 нужны справки от: [чиновник 3]

(Дуга 2 → 3: справка 2 нужна для справки 3)

Проверяем чиновника 3: еще не посещен

Записываем: "Переходим к чиновнику 3"

Шаг 5: Обрабатываем чиновника 3

Записываем в журнал: "Посетили чиновника 3"

visited = [да, да, да, да]

Проверяем зависимости чиновника 3:

Для получения справки от чиновника 3 нужны справки от: [] (список пуст)

Все необходимые справки уже получены (их нет)

Записываем: "Все справки для чиновника 3 получены"

Записываем в журнал завершения: "Чиновник 3 завершен"

order = [3]

Шаг 6: Возвращаемся к чиновнику 2

Возвращаемся в журнал чиновника 2:

Все зависимости обработаны

Справка от чиновника 3 уже получена (чиновник 3 уже обработан)

Записываем: "Все справки для чиновника 2 получены"

Записываем в журнал завершения: "Чиновник 2 завершен"

order = [3, 2

Шаг 7: Возвращаемся к чиновнику 1

Возвращаемся в журнал чиновника 1:

Все зависимости обработаны

Справка от чиновника 2 уже получена (чиновник 2 уже обработан)

Записываем: "Все справки для чиновника 1 получены"

Записываем в журнал завершения: "Чиновник 1 завершен"

order = [3, 2, 1]

Шаг 8: Возвращаемся к чиновнику 0

Возвращаемся в журнал чиновника 0:

Все зависимости обработаны

Справка от чиновника 1 уже получена (чиновник 1 уже обработан)

Записываем: "Все справки для чиновника 0 получены"

Записываем в журнал завершения: "Чиновник 0 завершен"

order = [3, 2, 1, 0]

Шаг 9: Проверяем всех чиновников

Проверяем, не осталось ли непосещенных чиновников:

Чиновник 0: visited[0] = да → уже обработан

Чиновник 1: visited[1] = да → уже обработан

Чиновник 2: visited[2] = да → уже обработан

Чиновник 3: visited[3] = да → уже обработан

Все чиновники обработаны.

Шаг 10: Формируем план обращения

Журнал завершения (в обратном порядке): [3, 2, 1, 0]

Разворачиваем порядок: [0, 1, 2, 3]

ПЛАН ОБРАЩЕНИЯ К ЧИНОВНИКАМ:

1. Обратиться к чиновнику 0

2. Обратиться к чиновнику 1

3. Обратиться к чиновнику 2

4. Обратиться к чиновнику 3

Шаг 11: Верификация плана

Проверяем, что все зависимости соблюдены:

Для чиновника 1 нужна справка 0:

В плане: чиновник 0 идет перед чиновником 1

Для чиновника 2 нужна справка 1:

В плане: чиновник 1 идет перед чиновником 2

Для чиновника 3 нужна справка 2:

В плане: чиновник 2 идет перед чиновником 3

3.4.Блок-схема алгоритма

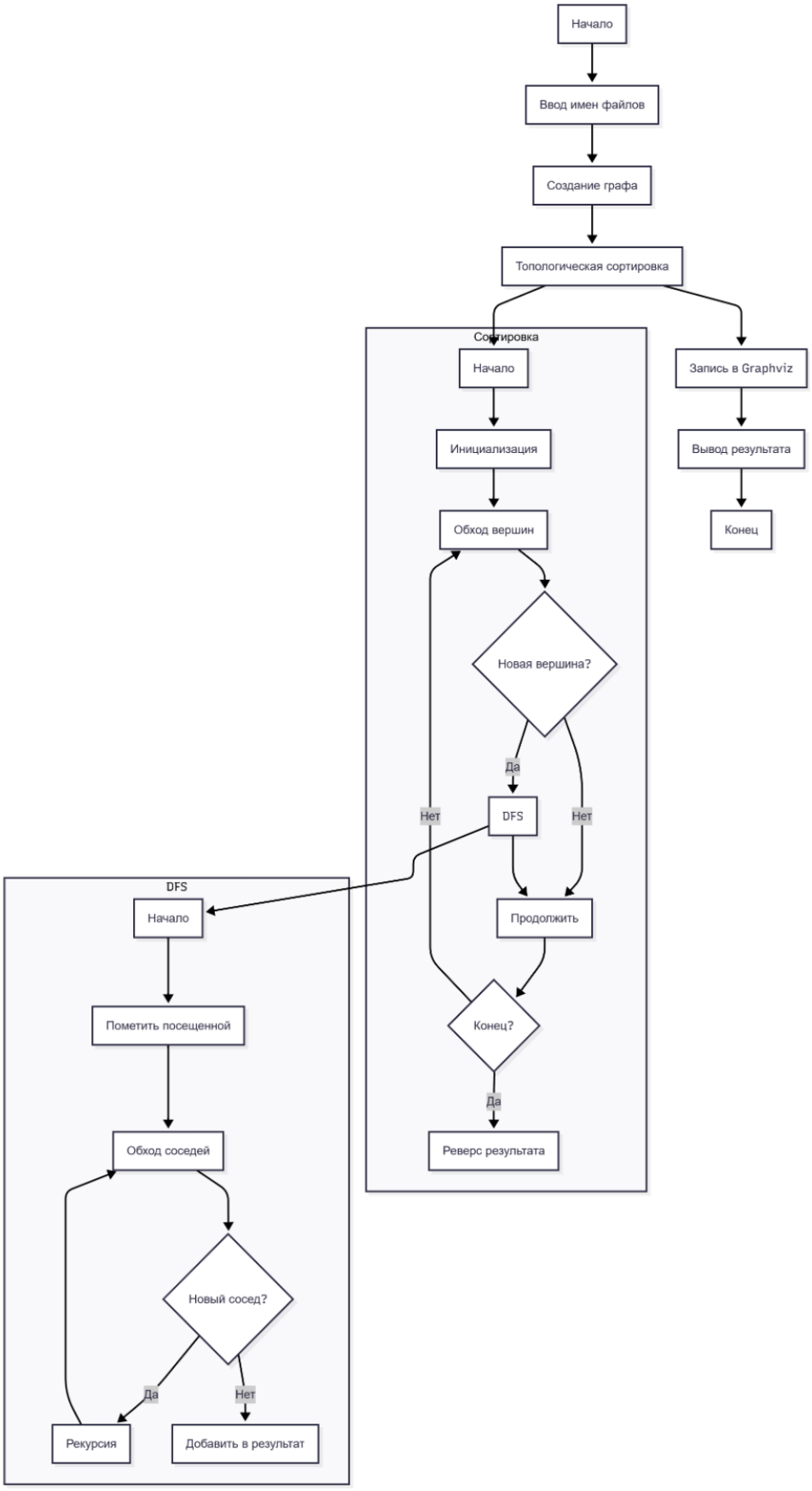

Рисунок 1 - Блок-схема алгоритма

3.5. Структуры данных

В реализации используются следующие структуры данных: список смежности adjacencyList_ в виде вектора векторов целых чисел для представления ориентированного графа, где для каждой вершины хранится список вершин, в которые ведут направленные рёбра. Целочисленная переменная numVertices_ хранит общее количество вершин графа. При выполнении обхода в глубину используется булевый вектор used для отметки посещённых вершин. Вектор целых чисел topologicalAnswer накапливает вершины в порядке, обратном топологическому, до финального разворота. Для работы с файлами применяются строки для хранения путей к входным и выходным файлам.

3.6. Анализ сложности

Время инициализации структур данных, включая создание массивов посещенных вершин и результата, составляет O(V), где V - количество вершин. Каждая вершина обрабатывается один раз при инициализации, что обеспечивает линейную зависимость от размера графа.

Основные вычислительные затраты приходятся на обход графа с помощью поиска в глубину. Каждая вершина посещается ровно один раз, а каждое ребро проверяется один раз при обходе смежных вершин. Это дает сложность O(V + E), где E - количество ребер в графе. Данная оценка является оптимальной для алгоритмов обхода графов. Операция разворота итогового массива имеет сложность O(V), так как требует однократного прохода по всем элементам массива размером V. Эта операция выполняется после завершения обхода графа. Запись результата в формате Graphviz требует O(V + E) операций, поскольку необходимо записать информацию о всех вершинах и всех ребрах графа. Каждая вершина и каждое ребро обрабатываются константное количество раз.

Таким образом, общая вычислительная сложность алгоритма составляет O(V + E), что является оптимальной для задачи топологической сортировки.

Данная оценка значительно эффективнее $O(V^2)$ простых подходов, особенно для разреженных графов, где E значительно меньше V^2. Пространственная сложность алгоритма составляет $O(V + E)$ для хранения графа в виде списка смежности, плюс $O(V)$ для вспомогательных массивов посещенных вершин и стека вызовов поиска в глубину. В худшем случае глубина рекурсии может достигать $O(V)$, что следует учитывать при работе с большими графами. Преимущество алгоритма на основе поиска в глубину проявляется для графов любой плотности, делая его универсальным решением для задач топологической сортировки.

4. ИНСТРУКЦИЯ ПОЛЬЗОВАТЕЛЯ

Программа запускается пользователем из командной строки. Командная строка принимает два параметра: имя файла с исходным текстом (input.txt), и имя файла для сохранения результатов (output.dot). Входные файлы должны быть в формате .txt.

Пример запуска: input.txt output.dot

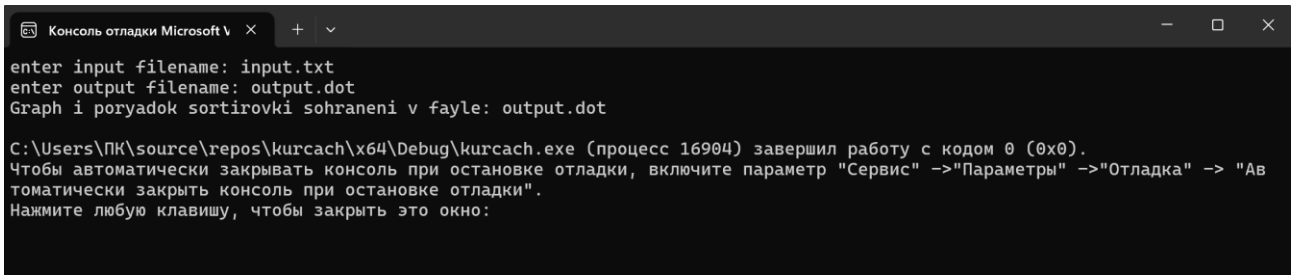

Рисунок 2 - Запуск программы из командной строки

Входной файл: текстовый файл (.txt), содержащий матрицу смежности ориентированного графа в формате, понятном для программы топологической сортировки.

Выходной файл: текстовый файл специального формата (.dot) для описания графов, который содержит инструкции для визуализации графа и найденной топологической сортировки.

5. ТЕСТОВЫЕ ПРИМЕРЫ

5.1 Результаты тестов

Тест 1

Входной файл:

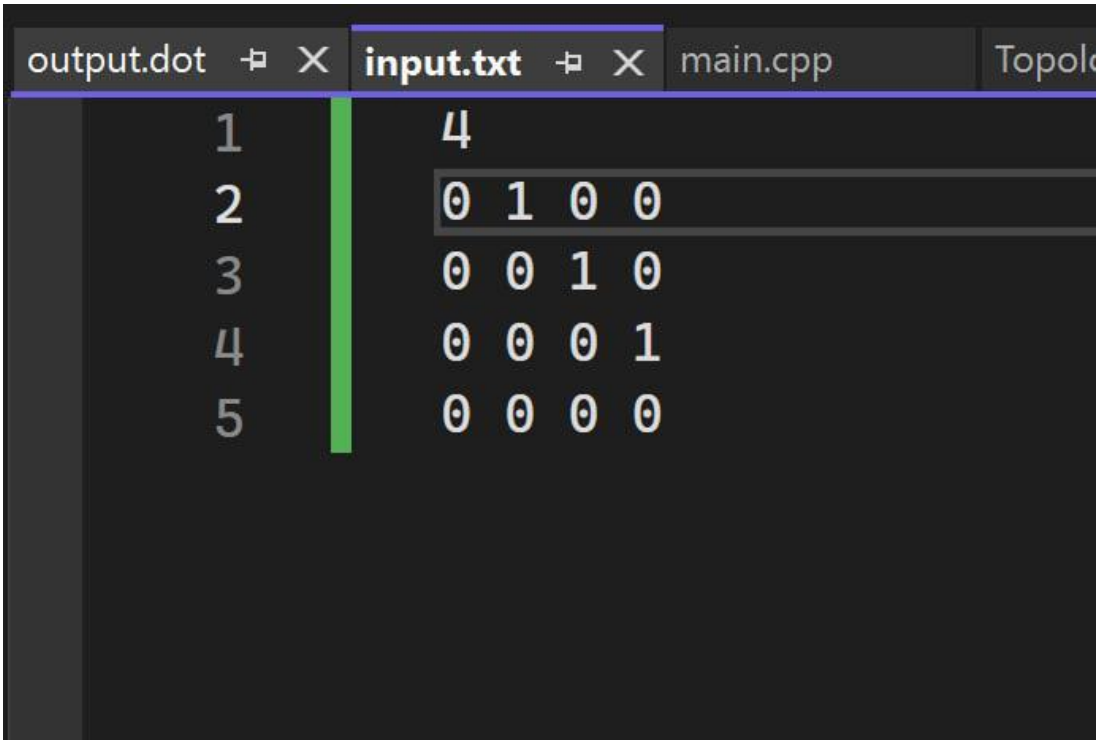

Рисунок 3 - Содержимое входного файла для теста 1

Выходной файл:

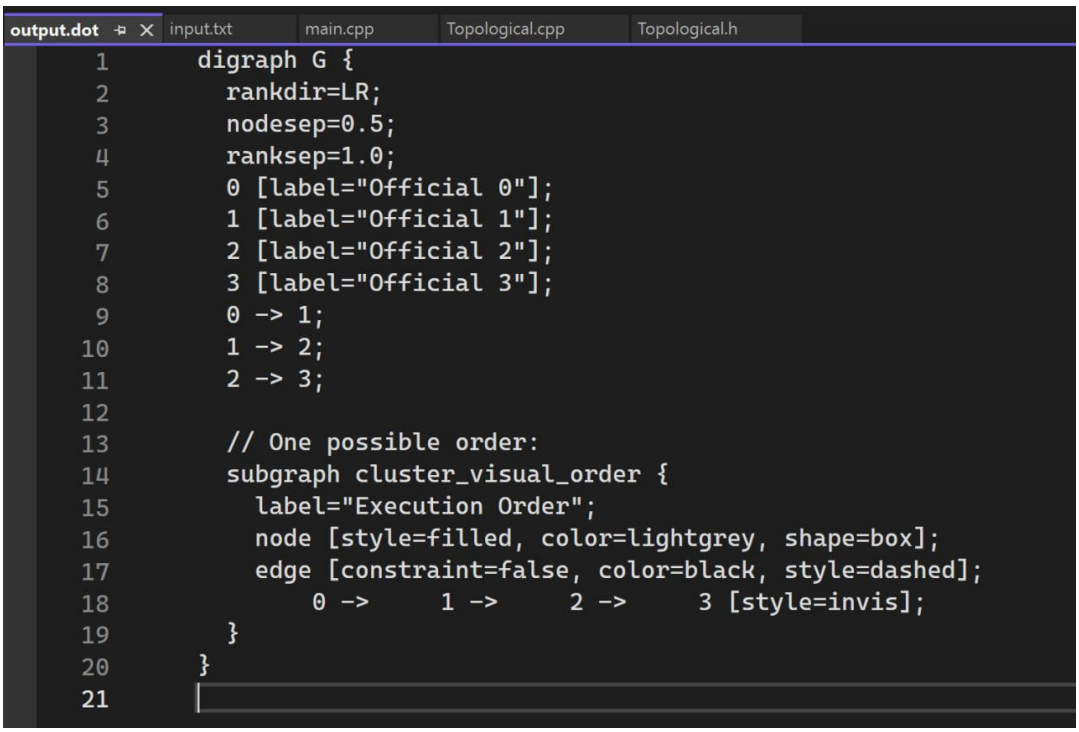

Рисунок 4 – Содержимое выходного файла для теста 1

Тест 2

Входной файл:

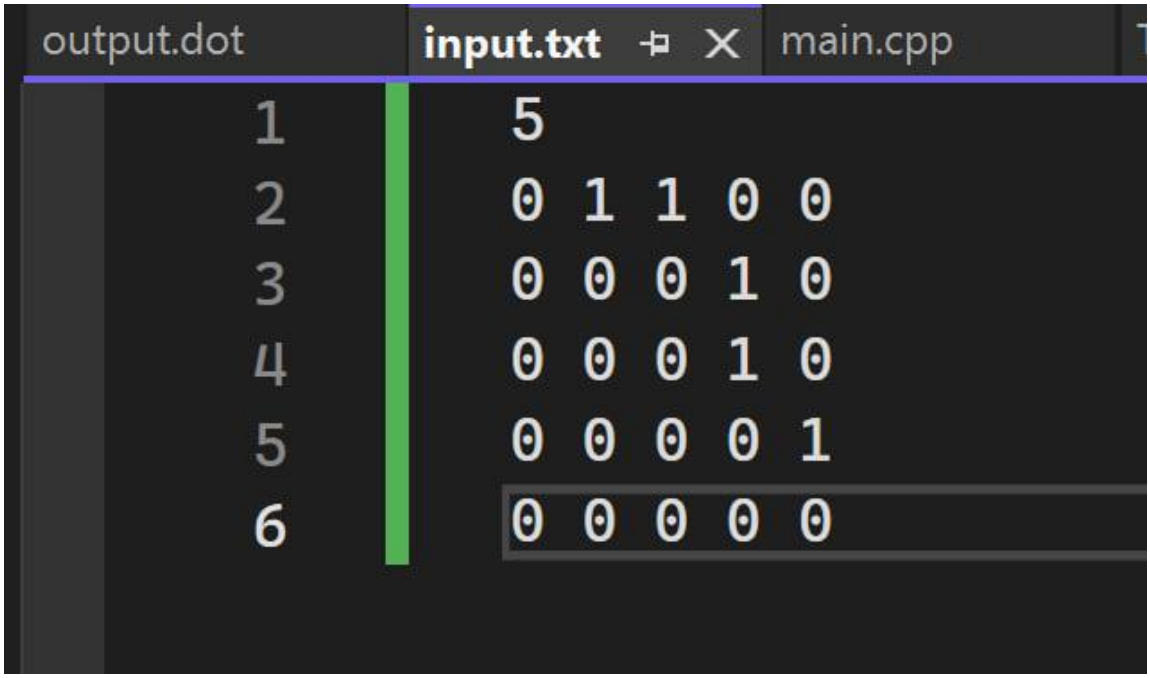

Рисунок 5 - Содержимое входного файла для теста 2

Выходной файл:

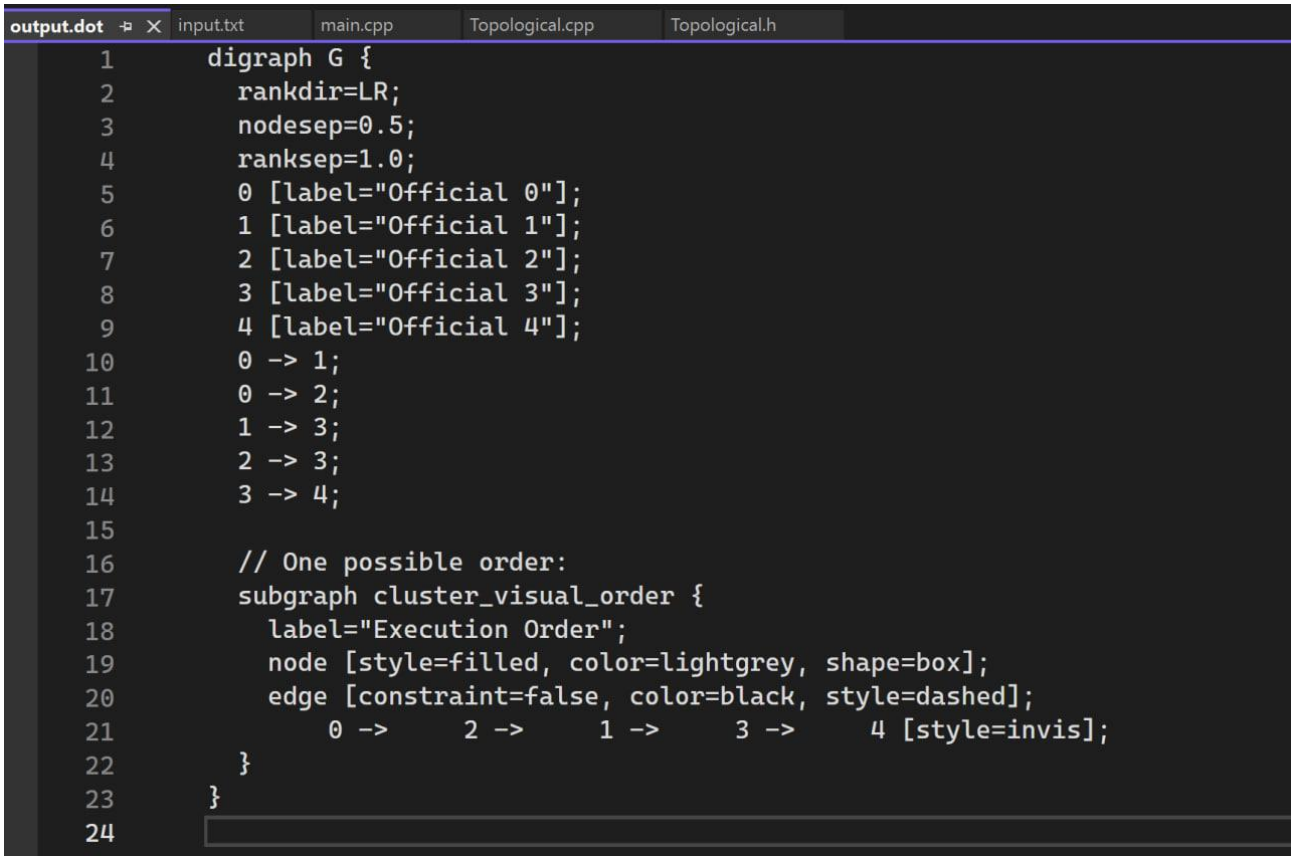

Рисунок 6 - Содержимое выходного файла для теста 2

Тест 3

Входной файл:

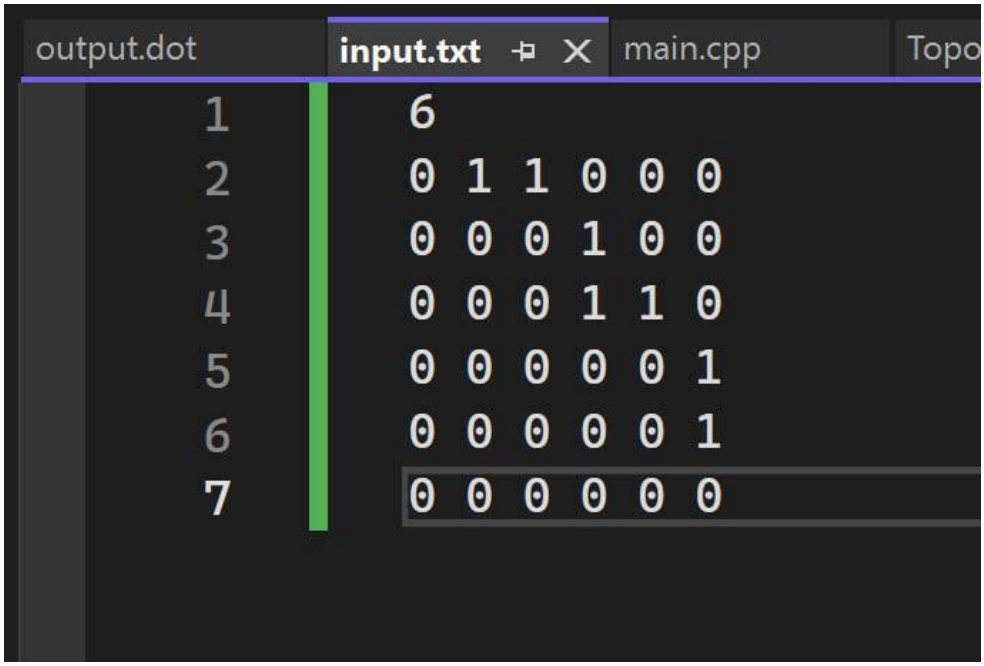

Рисунок 7 - Содержимое входного файла для теста 3

Выходной файл:

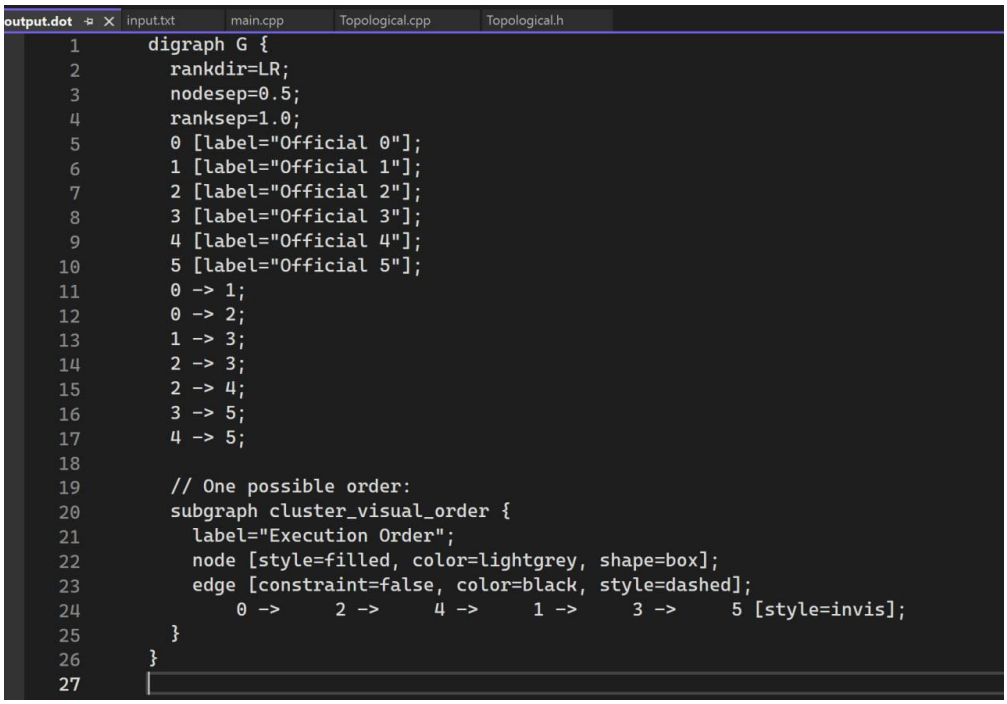

Рисунок 8 - Содержимое выходного файла для теста 3

5.2 Анализ полученных результатов

Алгоритм корректно обрабатывает все ациклические ориентированные графы (DAG). Алгоритм обрабатывает графы любой размерности, включая граничные случаи: одну вершину, пустой граф, полный порядок.

6. ЗАКЛЮЧЕНИЕ

В ходе работы был успешно реализован алгоритм топологической сортировки на основе поиска в глубину. Алгоритм продемонстрировал корректную работу на различных тестовых графах и подтвердил свою вычислительную эффективность. Теоретическая оценка сложности O(V + E) нашла практическое подтверждение в реализации.

Основные принципы алгоритма - использование времени выхода из вершин при обходе в глубину и проверка на ацикличность графа - были успешно применены на практике. Особое внимание уделено обработке возможных циклов в графе и генерации наглядного представления результатов.

Разработанная программа корректно определяет возможность топологической сортировки и строит допустимый порядок вершин для ациклических ориентированных графов. Интеграция с системой визуализации Graphviz обеспечивает удобное представление структуры графа и полученного порядка обхода.

Алгоритм представляет собой эффективное решение для задач упорядочивания, возникающих в планировании выполнения задач, анализе зависимостей и других прикладных областях. Реализация может быть использована как самостоятельное приложение так и в качестве основы для более сложных систем анализа графов.

7. СПИСОК ЛИТЕРАТУРЫ

[1] К. Т., Алгоритмы. Построение и анализ, 2013.

[2] А. А., Дж. Х., Дж. У., Построение и анализ вычислительных алгоритмов, 2013.