

Министерство науки и высшего образования Российской Федерации

ФГБОУ ВО «Пензенский государственный университет»

Факультет вычислительной техники

Кафедра «Вычислительная техника»

Пояснительная записка к курсовой работе
по курсу «Логика и основы алгоритмизации в инженерных задачах»
на тему «Реализация алгоритма выделения компонент связности орграфа,
используя
поиск в глубину»

28.12.25
отлично
оценка

Выполнил:

студент группы 24BVB1

Тусков А.А.

Принял:

к.т.н., доцент Юрова О.В.

Пенза, 2025

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет Вычислительной техники
Кафедра "Вычислительная техника"

"УТВЕРЖДАЮ"

Зав. кафедрой ВТ

« » 20

ЗАДАНИЕ

на курсовое проектирование по курсу

Логика и основы алгоритмизации Линейного задания "
Студенту Тухов Арсений Андреевич Группа 24ВВВ1
Тема проекта Визуализация алгоритма выделения компонент
связности графа, используя поиск в глубину

Исходные данные (технические требования) на проектирование

- Разработка алгоритмов и программного обеспечения в соответствии
с данным заданием курсового проекта.
Пояснительная записка должна содержать:
1. Исходную задачу;
 2. Краткое описание задачи;
 3. Описание алгоритма поставленной задачи;
 4. Пример ручного расчета задачи и объяснений (на небольшом
участке работы алгоритма);
 5. Описание самой программы;
 6. Тесты;
 7. Список литературы;
 8. Листинг программы;
 9. Результаты работы программы.

Объем работы по курсу
1. Расчетная часть

Ручной расчёт работы алгоритма

2. Графическая часть

Схема алгоритма / формате блок-схем.

3. Экспериментальная часть

Тестирование программы;

Результаты работы программы на тестовых данных

Срок выполнения проекта по разделам

1 Исследование теоретической части курсового проекта

2 Разработка алгоритмов программы

3 Разработка программы

4 Тестирование и завершение разработки программы

5 Оформление пояснительной записки

6

7

8

Дата выдачи задания "12" сентября 2025г.

Дата защиты проекта "29" декабря 2025г.

Руководитель Прова О.В.

Задание получил "25" сентября 2025г.

Студент Тусков Арсений Андреевич

Содержание

Реферат.....	5
Введение	6
1 Постановка задачи.....	9
2 Теоретическая часть задания	12
3 Описание программы.....	15
3.1. Описание алгоритмов	15
3.3 Функции для работы с графом	20
3.4 Функции реализации алгоритмов.....	22
3.5 Вспомогательные функции	24
3.6 Главное меню и управление программой	25
4 Тестирование	27
4.1. Генерация графа	29
4.2. Сохранение и загрузка результатов действий алгоритмов	32
4.3 Результаты.....	33
5 Ручной расчет алгоритма Косарайю	34
Заключение.....	35
Список литературы.....	37
Приложение А. Листинг программы	38

Реферат

Отчет стр. 57 , рисунков 18.

ГРАФ, ТЕОРИЯ ГРАФОВ, ОРГРАФ, ДОСТИЖИМОСТЬ, ПОИСК В ГЛУБИНУ, КОМПОНЕНТА.

Цель исследования – разработка программы, способная выделять компоненты сильной связности орграфа, используя алгоритм поиска в ширину.

В работе рассмотрены правила поиска в глубину, на основе которых находятся компонент связности орграфа.

Введение

Ориентированные графы, представляющие собой математические модели систем с направленными связями, занимают центральное место в современной информатике и прикладной математике. Эти структуры данных находят применение в самых разнообразных областях: от анализа социальных сетей, где направленные рёбра могут означать отношения "подписчик-автор", до проектирования цифровых схем, где они моделируют потоки сигналов между компонентами. Особую важность в контексте анализа ориентированных графов приобретает задача выделения компонент сильной связности — групп вершин, в пределах которых каждая вершина достижима из любой другой. Решение этой задачи позволяет выявлять тесно связанные сообщества в социальных графах, находить циклические зависимости в программном коде, определять кластеры в биологических сетях и решать множество других прикладных проблем.

Алгоритм Косарайю, предложенный в 1978 году индийским ученым Саитом Рао Косарайю, является одним из наиболее элегантных и понятных методов решения задачи поиска компонент сильной связности. Его красота заключается в простоте концепции: двукратное применение поиска в глубину (Depth-First Search, DFS) с промежуточным транспонированием графа. Несмотря на кажущуюся прямолинейность подхода, алгоритм обладает строгим математическим обоснованием и оптимальной асимптотической сложностью $O(V + E)$, где V — количество вершин, а E — количество рёбер. Эта эффективность в сочетании с относительной простотой реализации делает алгоритм Косарайю идеальным объектом для изучения в рамках учебного курса по алгоритмам и структурам данных.

Целью настоящей курсовой работы является создание комплексного программного обеспечения на языке C, реализующего не только классический

алгоритм Косарайю, но и предоставляющего инструменты для всестороннего анализа ориентированных графов.

Методологической основой реализации послужил язык программирования C, выбранный за свою эффективность, близость к аппаратным ресурсам и широкое использование в системном программировании. Разработка велась в среде Microsoft Visual Studio Community 2022, что обеспечило современные инструменты отладки, контроля версий и тестирования. Консольный интерфейс, несмотря на кажущуюся простоту, был спроектирован с учетом принципов юзабилити, обеспечивая интуитивно понятное взаимодействие даже для пользователей, не обладающих специальной подготовкой в области теории графов.

Актуальность данной работы определяется возрастающей потребностью в инструментах анализа сложных сетевых структур в эпоху больших данных. Умение эффективно выявлять сильносвязанные компоненты становится критически важным навыком для специалистов в области анализа данных, компьютерных наук и прикладной математики. Представленная программа не только решает конкретную алгоритмическую задачу, но и служит наглядной иллюстрацией того, как теоретические концепции из области дискретной математики могут быть воплощены в практические вычислительные инструменты.

В ходе работы над проектом были рассмотрены альтернативные подходы к решению задачи (в частности, алгоритм Тарьяна), однако выбор в пользу алгоритма Косарайю был обусловлен его дидактической ценностью: последовательность шагов алгоритма легко визуализируется и понимается, что делает его идеальным для образовательных целей. Разработанное программное обеспечение может быть использовано не только как практическая реализация алгоритма, но и как демонстрационный материал при изучении темы "Компоненты связности в графах" в рамках учебных курсов по дискретной математике и алгоритмам.

Таким образом, данная курсовая работа представляет собой законченное программное решение, сочетающее в себе строгую алгоритмическую основу, практическую направленность и образовательную ценность, внося вклад как в развитие вычислительных методов анализа графов, так и в методику преподавания соответствующих дисциплин.

1 Постановка задачи

В рамках данной курсовой работы требуется разработать программную реализацию алгоритма выделения компонент связности ориентированного графа с использованием поиска в глубину. Основная задача заключается в создании консольного приложения на языке C, которое позволяет находить все компоненты сильной связности в произвольном ориентированном графе с применением классического алгоритма Косарайю. Программа должна обеспечивать полный цикл работы: от формирования или загрузки графа до визуализации результатов и сохранения истории вычислений.

Ориентированный граф (орграф) должен представляться в памяти программы в виде матрицы смежности размером $N \times N$, где N — количество вершин. Каждый элемент матрицы $a[i][j]$ принимает значение 1, если существует направленное ребро из вершины i в вершину j , и 0 в противном случае. Для неориентированных графов (рассматриваемых в качестве дополнительного функционала) матрица смежности должна быть симметричной. Граф может быть сформирован несколькими способами: автоматическая генерация со случайными рёбрами с заданной вероятностью, ручной ввод матрицы смежности пользователем или загрузка из текстового файла predetermined формата. При генерации случайного графа необходимо исключить возможность создания петель (рёбер из вершины в саму себя) и обеспечить, чтобы каждая вершина имела хотя бы одно исходящее ребро для нетривиальности результата.

Ключевым алгоритмическим компонентом программы является реализация метода Косарайю для нахождения компонент сильной связности. Алгоритм должен выполняться в три последовательных этапа. На первом этапе осуществляется обход в глубину (DFS) исходного графа, в ходе которого вершины по мере завершения их обработки помещаются в стек. Этот порядок важен, так как он отражает топологическую сортировку конденсации графа.

На втором этапе выполняется транспонирование исходного графа — создание новой матрицы смежности, в которой все рёбра изменяют направление на противоположное. На третьем этапе производится второй обход в глубину, но уже по транспонированному графу, причём вершины выбираются из стека в порядке, обратном их добавлению (LIFO). Каждый запуск DFS на этом этапе из ещё не посещённой вершины находит все вершины, принадлежащие одной компоненте сильной связности. Алгоритм завершается, когда все вершины обработаны, а найденные компоненты представляются пользователю в удобочитаемом виде с указанием номеров вершин, входящих в каждую компоненту.

Помимо основного алгоритма для ориентированных графов, программа должна предоставлять дополнительный функционал для работы с неориентированными графами. В этом случае компоненты связности находятся с помощью стандартного поиска в глубину без учёта направления рёбер. Также необходимо реализовать алгоритм поиска компонент слабой связности для ориентированных графов, который игнорирует направление рёбер и рассматривает граф как неориентированный. Это позволяет продемонстрировать различия между типами связности и углубить понимание структуры графа.

Программа должна обладать интуитивно понятным консольным интерфейсом, организованным в виде многоуровневого меню. Пользователь может выбирать количество вершин в графе (от 2 до 100), тип графа (ориентированный или неориентированный), режим отображения подробных шагов работы алгоритма. Для ориентированных графов предоставляется выбор между поиском сильной и слабой связности. Все операции сопровождаются пояснительными сообщениями и визуализацией промежуточных результатов: выводом матрицы смежности, списка рёбер, процесса транспонирования графа, содержимого стека вершин и последовательности обхода. Особое внимание уделяется образовательной

составляющей — при включённом режиме пошагового отображения программа подробно комментирует каждый этап алгоритма, что способствует лучшему пониманию его работы.

Важным требованием является реализация системы сохранения и загрузки истории поиска. Все результаты работы алгоритма (дата и время выполнения, количество вершин, тип связности, количество найденных компонент и их состав) должны записываться в текстовый файл. Пользователь должен иметь возможность просматривать историю прямо в программе, что позволяет анализировать предыдущие запуски, сравнивать результаты для разных графов и типов связности. Файловый ввод-вывод должен быть защищён от ошибок: проверяться корректность открытия файлов, обрабатываться нестандартные ситуации, обеспечиваться сохранение данных даже при аварийном завершении программы.

Программа должна корректно работать с графами различной структуры: от небольших учебных примеров до графов со 100 вершинами. Необходимо обеспечить обработку граничных случаев: полностью связных графов (одна компонента), графов без рёбер (каждая вершина — отдельная компонента), графов с циклами и сильносвязными компонентами различного размера. Производительность алгоритма должна быть приемлемой для заданного диапазона вершин, что достигается оптимальным выбором структур данных и алгоритмических подходов.

В итоге программа представляет собой законченное учебное приложение, демонстрирующее не только работу алгоритма Косарайю, но и общие принципы обработки графов, реализации поиска в глубину, работы с файлами и построения пользовательских интерфейсов в консольных приложениях на языке C. Разработка такой программы требует глубокого понимания теории графов, алгоритмических основ и принципов программной инженерии, что соответствует целям курсовой работы по дисциплине "Логика и основы алгоритмизации".

2 Теоретическая часть задания

Алгоритм Косарайю (англ. *Kosaraju's Algorithm*) позволяет находить все компоненты сильной связности (SCC) в ориентированном графе. Этот алгоритм основан на двойном применении поиска в глубину (DFS) и является одним из классических методов решения данной задачи. Алгоритм опирается на ключевое наблюдение: компоненты сильной связности инвариантны относительно транспонирования графа.

Алгоритм Косарайю обеспечивает полное и корректное выделение всех компонент сильной связности в графе, что позволяет анализировать структуру ориентированных сетей, выявлять циклы и взаимозависимости. Отличие алгоритма Косарайю от других методов (например, алгоритма Тарьяна или алгоритма поиска компонент связности в неориентированных графах) заключается в его простоте реализации и наглядности: он явно разделяет процесс на две независимые фазы обхода графа.

Принцип работы алгоритма:

- Первый обход в глубину. Выполняется стандартный DFS по исходному графу. По мере завершения обработки каждой вершины (когда все её потомки обработаны), вершина помещается в стек. Этот порядок гарантирует, что вершины, принадлежащие одной компоненте сильной связности, будут расположены в стеке компактно.

- Транспонирование графа. Создаётся новый граф, в котором все рёбра имеют направление, противоположное исходному. Если в исходном графе было ребро из вершины u в вершину v , то в транспонированном графе будет ребро из v в u . Эта операция сохраняет компоненты сильной связности, но изменяет структуру достижимости между ними.

– Второй обход в глубину. Выполняется DFS по транспонированному графу, причём вершины обрабатываются в порядке, обратном их положению в стеке (от верхушки стека к его основанию). Каждый запуск DFS из ещё не посещённой вершины находит все вершины одной компоненты сильной связности.

– Формирование результата. Вершины, посещённые в каждом запуске DFS на втором этапе, образуют отдельную компоненту сильной связности. Процесс продолжается до тех пор, пока все вершины не будут обработаны.

Математическое

обоснование:

Пусть $G = (V, E)$ — ориентированный граф, а $G^T = (V, E^T)$ — его транспонированная версия, где $E^T = \{(v, u) \mid (u, v) \in E\}$. Алгоритм Косарайю использует следующее свойство: компоненты сильной связности в G и G^T совпадают. Первый DFS определяет такой порядок вершин, что при обратном обходе транспонированного графа мы гарантированно находим целиком каждую компоненту, прежде чем перейти к вершинам из других компонент.

Отличия от других алгоритмов:

В отличие от алгоритма для неориентированных графов, который находит компоненты связности, игнорируя направление рёбер, алгоритм Косарайю учитывает ориентацию рёбер и находит именно компоненты *сильной* связности, где требуется взаимная достижимость. Алгоритм Косарайю особенно эффективен в образовательных целях и в случаях, когда простота реализации важнее предельной оптимизации производительности.

Асимптотическая

сложность:

Алгоритм Косарайю имеет временную сложность $O(V + E)$, где V — количество вершин, а E — количество рёбер в графе. Это оптимальная

сложность для решения данной задачи. Пространственная сложность также составляет $O(V + E)$ для хранения графа и вспомогательных структур данных.

Области

применения:

- Анализ структуры веб-графа (поиск сильно связанных групп страниц)
- Оптимизация компиляторов (анализ графа вызовов функций)
- Социальные сети (выявление тесно связанных сообществ)
- Анализ зависимостей в программных системах
- Нахождение конденсации графа (представление SCC в виде одной вершины)

3 Описание программы

Программа была написана в среде разработки VisualStudioCommunity 2022 на языке программирования C.

Далее будут приведены описания функций, структур и действия элементов управления, используемых в программе.

3.1. Описание алгоритмов

ОСНОВНОЙ АЛГОРИТМ КОСАРАЙЮ

ФУНКЦИЯ kosaraju(граф: Граф, результат: ПоискРезультат, показать_шаги: Логическое):

```
// Инициализация
```

```
стек = пустой стек
```

```
индекс_стека = -1
```

```
посещенные = массив[граф.размер] заполненный 0
```

```
// ШАГ 1: Первый обход в глубину (DFS)
```

```
ЕСЛИ показать_шаги = ИСТИНА:
```

```
    ВЫВЕСТИ "ШАГ 1: ПЕРВЫЙ ОБХОД В ГЛУБИНУ"
```

```
    ВЫВЕСТИ "Цель: Определить порядок завершения обработки  
    вершин"
```

```
ДЛЯ каждой вершины i от 0 до граф.размер-1:
```

```
    ЕСЛИ не посещенные[i]:
```

```
        ВЫВЕСТИ "Начинаем обход из вершины" i+1
```

```
        fill_order(i, посещенные, стек, &индекс_стека, граф)
```

```
// ШАГ 2: Транспонирование графа
```

```
ВЫВЕСТИ "ШАГ 2: ТРАНСПОНИРОВАНИЕ ГРАФА"
```

```

transpose_graph(граф)

// Сброс массива посещенных для второго обхода
ДЛЯ i от 0 до граф.размер-1:
    посещенные[i] = 0

// ШАГ 3: Второй обход в глубину
ВЫВЕСТИ "ШАГ 3: ВТОРОЙ ОБХОД В ГЛУБИНУ"
результат.компонент_счетчик = 0

ПОКА индекс_стека >= 0:
    v = стек[индекс_стека]
    индекс_стека = индекс_стека - 1

    ЕСЛИ не посещенные[v]:
        ВЫВЕСТИ "Начинаем обход из вершины" v+1 "(корень
компоненты)"

        новая_компонента = СОЗДАТЬ_КОМПОНЕНТУ()
        DFSUtil(v, посещенные, &новая_компонента, граф, 1)

        // Сохраняем найденную компоненту
        результат.компоненты[результат.компонент_счетчик] =
новая_компонента
        результат.компонент_счетчик = результат.компонент_счетчик +
1

        ВЫВЕСТИ "Найдена компонента сильной связности"
результат.компонент_счетчик

```

```
// Заполняем информацию о результате  
результат.вершин_количество = граф.размер  
результат.время = ТЕКУЩЕЕ_ВРЕМЯ()
```

```
ВЫВЕСТИ "Поиск завершен. Найдено"  
результат.компонент_счетчик "компонент"  
КОНЕЦ_ФУНКЦИИ
```

ФУНКЦИЯ fill_order(вершина: Целое, посещенные: Массив, стек:
Массив,

```
    указатель_индекса: Указатель, граф: Граф):  
    посещенные[вершина] = 1
```

ДЛЯ каждой вершины i от 0 до граф.размер-1:

ЕСЛИ граф.матрица[вершина][i] = 1 И посещенные[i] = 0:

ВЫВЕСТИ "Переход из вершины" вершина+1 "в вершину" i+1

fill_order(i, посещенные, стек, указатель_индекса, граф)

// Добавляем вершину в стек после обработки всех соседей

стек[указатель_индекса + 1] = вершина

указатель_индекса = указатель_индекса + 1

ВЫВЕСТИ "Вершина" вершина+1 "полностью обработана и
 добавлена в стек"

КОНЕЦ_ФУНКЦИИ

ФУНКЦИЯ transpose_graph(граф: Граф):

// Создаем транспонированную матрицу

ДЛЯ i от 0 до граф.размер-1:

ДЛЯ j от 0 до граф.размер-1:

граф.транспонированная[j][i] = граф.матрица[i][j]

// Визуализация процесса (для небольших графов)

ЕСЛИ граф.размер <= 10:

ВЫВЕСТИ "Исходная матрица смежности:"

ВЫВЕСТИ_МАТРИЦУ(граф.матрица)

ВЫВЕСТИ "Транспонированная матрица:"

ВЫВЕСТИ_МАТРИЦУ(граф.транспонированная)

ВЫВЕСТИ "Каждое ребро меняет направление на
противоположное"

КОНЕЦ_ФУНКЦИИ

Алгоритм слабой связности (для орграфа)

ФУНКЦИЯ find_weak_connectivity(граф: Граф, результат:
ПоискРезультат,

показать_шаги: Логическое):

// Инициализация

посещенные = массив[граф.размер] заполненный 0

результат.компонент_счетчик = 0

результат.вершин_количество = граф.размер

ЕСЛИ показать_шаги = ИСТИНА:

ВЫВЕСТИ "АЛГОРИТМ ПОИСКА СЛАБОЙ СВЯЗНОСТИ"

ВЫВЕСТИ "Принцип: игнорируем направление рёбер"

```

// Обход всех вершин
ДЛЯ каждой вершины i от 0 до граф.размер-1:
    ЕСЛИ не посещенные[i]:
        // Создаем новую компоненту
        компонента = СОЗДАТЬ_КОМПОНЕНТУ()

        // Используем стек для итеративного DFS
        стек = пустой стек
        стек.положить(i)
        посещенные[i] = 1

        ВЫВЕСТИ "Начинаем обход из вершины" i+1 "(корень
компоненты)"

        ПОКА стек не пуст:
            текущая = стек.взять()
            компонента.добавить(текущая)

            ВЫВЕСТИ "Обрабатываем вершину" текущая+1

            // Проверяем связи в ЛЮБОМ направлении
            ДЛЯ j от 0 до граф.размер-1:
                // Если есть ребро в любом направлении
                ЕСЛИ (граф.матрица[текущая][j] = 1 ИЛИ
граф.матрица[j][текущая] = 1)
                    И посещенные[j] = 0:
                        посещенные[j] = 1
                        стек.положить(j)

```

ЕСЛИ показать_шаги = ИСТИНА:

ВЫВЕСТИ "Переход к вершине" j+1

// Сохраняем компоненту

результат.компоненты[результат.компонент_счетчик] =
компонента

результат.компонент_счетчик = результат.компонент_счетчик +
1

ВЫВЕСТИ "Найдена компонента слабой связности"
результат.компонент_счетчик

результат.время = ТЕКУЩЕЕ_ВРЕМЯ()

ВЫВЕСТИ "Найдено" результат.компонент_счетчик "компонент
слабой связности"

КОНЕЦ_ФУНКЦИИ

ФУНКЦИЯ ТРАНСПОНИРОВАНИЯ ГРАФА

void transpose_graph(Graph* g) - Транспонирует матрицу смежности графа.

3.3 Функции для работы с графом

void generate_directed_graph(Graph* g) - Генерирует случайный
ориентированный граф с заданными параметрами

void generate_undirected_graph(Graph* g) - Генерирует случайный
неориентированный граф.

void print_matrix(Graph* g) - Выводит матрицу смежности графа в форматированном виде.

void print_connections(Graph* g) - Выводит список исходящих связей для каждой вершины ориентированного графа.

```
Создано ребро: 5 -- 7
Создано двунаправленное ребро: 6 -- 7
Всего создано ребер: 14

=== МАТРИЦА СМЕЖНОСТИ ===
(1 - есть ребро, 0 - нет ребра)
```

	1	2	3	4	5	6	7
1	0	0	0	0	0	0	1
2	1	0	0	0	0	1	1
3	0	0	0	0	1	0	0
4	0	0	0	0	1	0	0
5	0	1	1	0	0	0	1
6	0	1	0	0	0	0	1
7	0	1	1	0	0	1	0

```
Интерпретация для ориентированного графа:
- Элемент [i][j] = 1 означает ребро ИЗ вершины i В вершину j
- Матрица НЕ обязательно симметрична
- Диагональ всегда нулевая (петли не допускаются)
```

Рисунок 1 – Вывод графа

```
Интерпретация для ориентированного графа:
- Элемент [i][j] = 1 означает ребро ИЗ вершины i В вершину j
- Матрица НЕ обязательно симметрична
- Диагональ всегда нулевая (петли не допускаются)

=== СПИСОК ИСХОДЯЩИХ СВЯЗЕЙ ===
(Для ориентированного графа)
```

Вершина 1	-- { 7 }
Вершина 2	-- { 1 6 7 }
Вершина 3	-- { 5 }
Вершина 4	-- { 5 }
Вершина 5	-- { 2 3 7 }
Вершина 6	-- { 2 7 }
Вершина 7	-- { 2 3 6 }

Рисунок 2 – Вывод связей в графе

3.4 Функции реализации алгоритмов

Функции алгоритма Косарайю

void kosaraju(Graph* g, SearchResult* result, int show_steps)

```
Показывать подробные шаги алгоритма Косарайю? (1 – да, 0 – нет): 1

=====
ШАГ 1: ПЕРВЫЙ ОБХОД В ГЛУБИНУ (DFS)
Цель: Определить порядок завершения обработки вершин
Стек заполняется в порядке убывания времени завершения
=====

Начинаем обход из вершины 1:
  Переход из вершины 1 в вершину 7
  Переход из вершины 7 в вершину 2
  Переход из вершины 2 в вершину 6
  Вершина 6 полностью обработана и добавлена в стек
  Вершина 2 полностью обработана и добавлена в стек
  Переход из вершины 7 в вершину 3
  Переход из вершины 3 в вершину 5
  Вершина 5 полностью обработана и добавлена в стек
  Вершина 3 полностью обработана и добавлена в стек
  Вершина 7 полностью обработана и добавлена в стек
  Вершина 1 полностью обработана и добавлена в стек

Начинаем обход из вершины 4:
  Вершина 4 полностью обработана и добавлена в стек

Содержимое стека (вершины в порядке завершения обработки):
Вершины: 6 2 5 3 7 1 4
(Вершина вверху стека: 4, внизу: 6)

Нажмите любую клавишу для продолжения...
```

Рисунок 3 – Вывод алгоритма Косарайю

```

=== ПРОЦЕСС ТРАНСПОНИРОВАНИЯ ГРАФА ===
Исходная матрица смежности:
      1  2  3  4  5  6  7
1  0  0  0  0  0  0  1
2  1  0  0  0  0  1  1
3  0  0  0  0  1  0  0
4  0  0  0  0  1  0  0
5  0  1  1  0  0  0  1
6  0  1  0  0  0  0  1
7  0  1  1  0  0  1  0

Транспонированная матрица смежности:
      1  2  3  4  5  6  7
1  0  0  0  0  0  0  0
2  0  0  0  0  1  1  1
3  0  0  0  0  1  0  1
4  0  0  0  0  0  0  0
5  0  0  1  1  0  0  0
6  0  1  0  0  0  0  1
7  1  1  0  0  1  1  0

Наглядное представление транспонирования:
Каждое ребро меняет направление на противоположное:
Ребро 1 -- 7 становится 7 -- 1
Ребро 2 -- 1 становится 1 -- 2
Ребро 2 -- 6 становится 6 -- 2
Ребро 2 -- 7 становится 7 -- 2
Ребро 3 -- 5 становится 5 -- 3
Ребро 4 -- 5 становится 5 -- 4
Ребро 5 -- 2 становится 2 -- 5
Ребро 5 -- 3 становится 3 -- 5
Ребро 5 -- 7 становится 7 -- 5
Ребро 6 -- 2 становится 2 -- 6
Ребро 6 -- 7 становится 7 -- 6
Ребро 7 -- 2 становится 2 -- 7
Ребро 7 -- 3 становится 3 -- 7
Ребро 7 -- 6 становится 6 -- 7
Нажмите любую клавишу для продолжения...

```

Рисунок 4 – Транспонирование графа

```

=====
ШАГ 3: ВТОРОЙ ОБХОД В ГЛУБИНУ
Цель: Найти компоненты сильной связности
Обход идет по транспонированному графу в порядке из стека
Каждое дерево поиска – одна компонента сильной связности
=====

--- Начинаем обход из вершины 4 (корень компоненты) ---
Найдена компонента сильной связности 1: 4

--- Начинаем обход из вершины 1 (корень компоненты) ---
Переход из вершины 1 в вершину 2
Переход из вершины 2 в вершину 5
Переход из вершины 5 в вершину 3
Переход из вершины 3 в вершину 7
Переход из вершины 7 в вершину 6
Найдена компонента сильной связности 2: 1 2 5 3 7 6

```

Рисунок 5 – Второй обход в глубину

```

Сохранить результаты в файл? (y/n): y
Результаты сохранены в файл search_history.txt

Нажмите любую клавишу для продолжения...|

```

Рисунок 6 – Сохранение в файл

```

Запись 3:
Дата и время: 24.12.2025 12:48:32
Количество вершин: 7
Количество компонент: 2
Компонента 1: 4
Компонента 2: 1 2 5 3 7 6
-----

```

```

Нажмите любую клавишу для продолжения...

```

Рисунок 7 – Вывод содержимого файла

3.5 Вспомогательные функции

`void clearInputBuffer()` – очищает буфер ввода (рисунок 7).

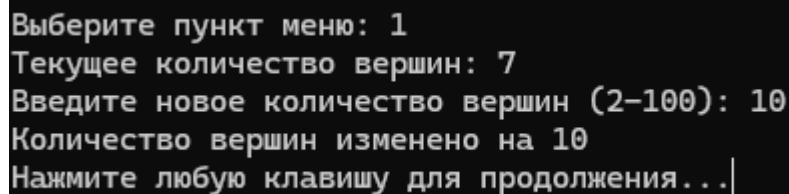
`bool isPositiveInteger(const char* str)` – проверяет, является ли строка положительным целым числом (рисунок 6).

`bool isIntegerInRange(const char* str, int min, int max)` – проверяет, находится ли число в заданном диапазоне (рисунок 6).

`int inputPositiveInteger(const char* prompt)` – запрашивает у пользователя ввод положительного целого числа с проверкой (рисунок 6).

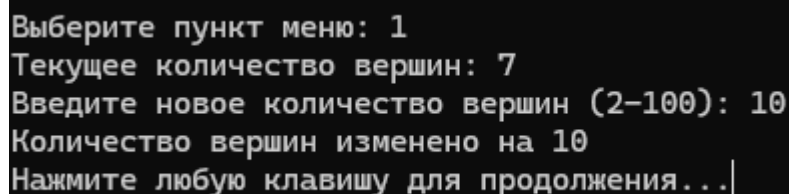
`int inputIntegerInRange(const char* prompt, int min, int max)` – запрашивает ввод числа в заданном диапазоне (рисунок 6).

`int minKey(const int* key, const bool* mstSet, int V)` – вспомогательная функция для алгоритма Прима: находит вершину с минимальным ключом.



```
Выберите пункт меню: 1
Текущее количество вершин: 7
Введите новое количество вершин (2-100): 10
Количество вершин изменено на 10
Нажмите любую клавишу для продолжения...|
```

Рисунок 8 –Проверка на ввод 1



```
Выберите пункт меню: 1
Текущее количество вершин: 7
Введите новое количество вершин (2-100): 10
Количество вершин изменено на 10
Нажмите любую клавишу для продолжения...|
```

Рисунок 9 –Проверка на ввод 2

3.6 Главное меню и управление программой

Функция `int main()` реализует консольное меню с следующими пунктами (рисунок 10):

Задать параметры для графа – запрашивает количество вершин

Запустить алгоритм поиска сильной компонент связности орграфа – отображает матрицу смежности сгенерированного графа и выводит список исходящих связей. Далее предлагается запустить алгоритм Косарайю. Сохраняет результаты в файл `search_history.txt` (опционально).

Запустить алгоритм поиска слабой связности орграфа – выводит матрицу смежности и запускает алгоритм поиска слабой связности. Сохраняет результаты в файл *search_history.txt* (опционально).

Просмотреть историю поиска – выводит полную сохраненную (опционально) историю поиска, которая сохраняется в файле *search_history.txt* (опционально).

Выйти из программы – корректно освобождает память и завершает работу программы.

```
=====
Курсовая работа
по дисциплине "Логика и основы алгоритмизации"
Тема: Реализация алгоритма поиска компонент связности
в орграфе, используя поиск в глубину

Студент: Тусков Арсений Андреевич
Группа: 24-BBB1
Преподаватель: Юрова О.В.
=====

МЕНЮ:
1. Установить количество вершин (текущее: 7)
2. Сильная связность ориентированного графа (алгоритм Косарайю)
3. Слабая связность ориентированного графа
4. Неориентированный граф
5. Просмотреть историю поиска
6. Выход

Выберите пункт меню: |
```

Рисунок 10 – Пример проверки на ввод

4 Тестирование

В качестве среды разработки была выбрана программа Microsoft Visual Studio версии Community 2022. Программа обладает всеми средствами необходимыми при разработке и отладке программы. Для отладки использовались несколько возможностей VisualStudio: режим отладки, трассировка, анализ содержимого переменных, мониторинг оперативной памяти. Такие возможности, как точка останова, шаг с заходом, шаг без захода были использованы для тестирования отдельных функций.

Тестирование проводилось во время разработки и также после завершения разработки. После внедрения каждой новой функции/элемента управления программа запускалась в режиме отладки для поиска возможных ошибок и преждевременного устранения этих ошибок. В ходе нее было выявлено некоторое количество проблем, связанных с обработкой данных, работой с памятью, работой элементов управления, работой форм.

Таблица 1 – Описание поведения программы при тестировании

№	Описание теста	Предусловие	Тестирование	Ожидаемый результат
1	Запуск программы	Нет	Запуск программы с помощью Visual studio	Открытие формы main
2	Установить количество вершин	Программа запущена	Выбор количества вершин. Ввод неправильных значений (отрицательные числа, 0 вершин)	Сообщение об ошибке для некорректного ввода. При корректном вводе - создание графа с выбранными характеристиками и вывод сообщения об принятых параметрах.

№	Описание теста	Предусловие	Тестирование	Ожидаемый результат
3	Сильная связность орграфа	Граф сгенерирован и начат процесс поиска	Выбор пункта “Сильная связность графа”, затем “Запустить алгоритм Косарайю”	Успешный вывод графа в консоль и запуск поиска, сохранение результатов
4	Слабая связность графа	Граф сгенерирован и начат процесс поиска	Выбор пункта “Слабая связность графа”, затем “Запустить алгоритм поиска”	Успешный вывод графа в консоль и запуск поиска, сохранение результатов
5	Сохранение и загрузка Файла	Файл с результатами поиска, а также дата/время, когда был воспроизведен алгоритм	Сохраняется в файл при выполнении программы /Выбор пункта "Загрузить граф из файла" с указанием имени файла	Успешная работа графа с файлами (сохранение /загрузка), При отсутствии файла или неверном формате - сообщение об ошибке

4.1. Генерация графа

В форме main выбираю 1 пункт меню «Установить количество вершин» (рисунок 11).

```
=====
Текущий граф: создан (5 вершин), сохранен в current_graph.txt
=====

МЕНЮ:
1. Установить количество вершин и создать граф
2. Показать текущий граф
3. Сильная связность (алгоритм Косарайю)
4. Слабая связность
5. Просмотреть историю поиска
6. Выход
```

Рисунок 11 – Выбор действий

Далее вписываем параметры в «Введите новое количество верши» (Рисунок 12).

```
Выберите пункт меню: 1
Текущее количество вершин: 7
Введите новое количество вершин (2-100): |
```

Рисунок 12 – Ввод данных

Затем запускаем один из алгоритмов поиска (Рисунок 13).

```

=== МАТРИЦА СМЕЖНОСТИ ===
(1 - есть ребро, 0 - нет ребра)

      1   2   3   4   5   6   7   8   9  10
-----
1 | 0   0   0   0   0   1   0   0   0   1
2 | 0   0   1   0   0   0   1   0   0   0
3 | 0   0   0   0   1   0   0   0   0   0
4 | 0   1   0   0   0   1   1   0   0   1
5 | 0   0   1   1   0   0   0   0   0   0
6 | 1   0   1   0   0   0   0   1   1   0
7 | 0   0   0   1   0   0   0   0   0   0
8 | 0   0   0   1   0   0   0   0   0   1
9 | 0   0   1   0   0   1   0   0   0   0
10 | 0   0   0   0   0   0   0   1   0   0

=== СПИСОК ИСХОДЯЩИХ СВЯЗЕЙ ===
(Для ориентированного графа)
Вершина 1 --> { 6 10 }
Вершина 2 --> { 3 7 }
Вершина 3 --> { 5 }
Вершина 4 --> { 2 6 7 10 }
Вершина 5 --> { 3 4 }
Вершина 6 --> { 1 3 8 9 }
Вершина 7 --> { 4 }
Вершина 8 --> { 4 10 }
Вершина 9 --> { 3 6 }
Вершина 10 --> { 8 }

```

Рисунок 13 – Алгоритм Косарайю

Далее – «Процесс поиска» (Рисунок 14).

Показывать подробные шаги алгоритма Косарайю? (1 – да, 0 – нет): 1

=====

ШАГ 1: ПЕРВЫЙ ОБХОД В ГЛУБИНУ (DFS)

Цель: Определить порядок завершения обработки вершин
Стек заполняется в порядке убывания времени завершения

=====

Начинаем обход из вершины 1:

Переход из вершины 1 в вершину 6
Переход из вершины 6 в вершину 3
Переход из вершины 3 в вершину 5
Переход из вершины 5 в вершину 4
Переход из вершины 4 в вершину 2
Переход из вершины 2 в вершину 7
Вершина 7 полностью обработана и добавлена в стек
Вершина 2 полностью обработана и добавлена в стек
Переход из вершины 4 в вершину 10
Переход из вершины 10 в вершину 8
Вершина 8 полностью обработана и добавлена в стек
Вершина 10 полностью обработана и добавлена в стек
Вершина 4 полностью обработана и добавлена в стек
Вершина 5 полностью обработана и добавлена в стек
Вершина 3 полностью обработана и добавлена в стек
Переход из вершины 6 в вершину 9
Вершина 9 полностью обработана и добавлена в стек
Вершина 6 полностью обработана и добавлена в стек
Вершина 1 полностью обработана и добавлена в стек

Содержимое стека (вершины в порядке завершения обработки):

Вершины: 7 2 8 10 4 5 3 9 6 1

(Вершина вверху стека: 1, внизу: 7)

Рисунок 14 – Обход в глубину

4.2. Сохранение и загрузка результатов действий алгоритмов

Для начала нужно запустить алгоритм через главное меню и дождаться результатов его работы (рисунок 15).

```
--- Начинаем обход из вершины 1 (корень компоненты) ---
Переход из вершины 1 в вершину 6
Переход из вершины 6 в вершину 4
Переход из вершины 4 в вершину 5
Переход из вершины 5 в вершину 3
Переход из вершины 3 в вершину 2
Переход из вершины 3 в вершину 9
Переход из вершины 4 в вершину 7
Переход из вершины 4 в вершину 8
Переход из вершины 8 в вершину 10
Найдена компонента сильной связности 1: 1 6 4 5 3 2 9 7 8 10

=====
ФИНАЛЬНЫЕ РЕЗУЛЬТАТЫ (СИЛЬНАЯ СВЯЗНОСТЬ)
=====
Количество компонент сильной связности: 1
Всего вершин в графе: 10

Граф является СИЛЬНО СВЯЗНЫМ!
Любая вершина достижима из любой другой.

Компоненты сильной связности:
Компонента 1 (размер: 10): { 1 6 4 5 3 2 9 7 8 10 }
```

Рисунок 15 – Вывод результатов работы алгоритма

Для того чтобы удостовериться в сохранении результатов перешел в директорию проекта и открыл файл "search_history.txt". Судя по содержимому файла, результат сохранен успешно (Рисунок 16).

```
Сохранить результаты в файл? (y/n): y
Результаты сохранены в файл search_history.txt

Нажмите любую клавишу для продолжения...|
```

Рисунок 16 – Подтверждение сохранения в файл

После этого выбрал пункт "Просмотреть историю поиска" (рисунок 17).

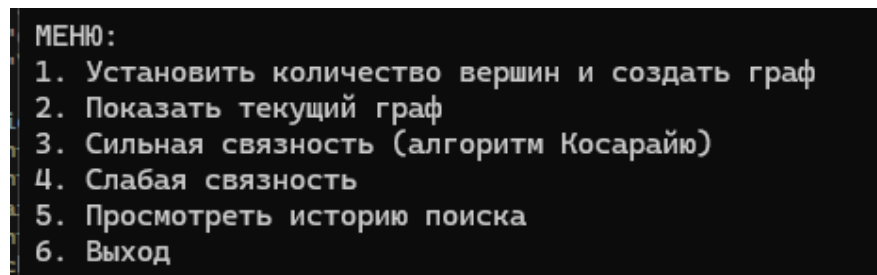


Рисунок 17 – Пункт в меню для выбора

Файл с результатом поиска (рисунок 18)

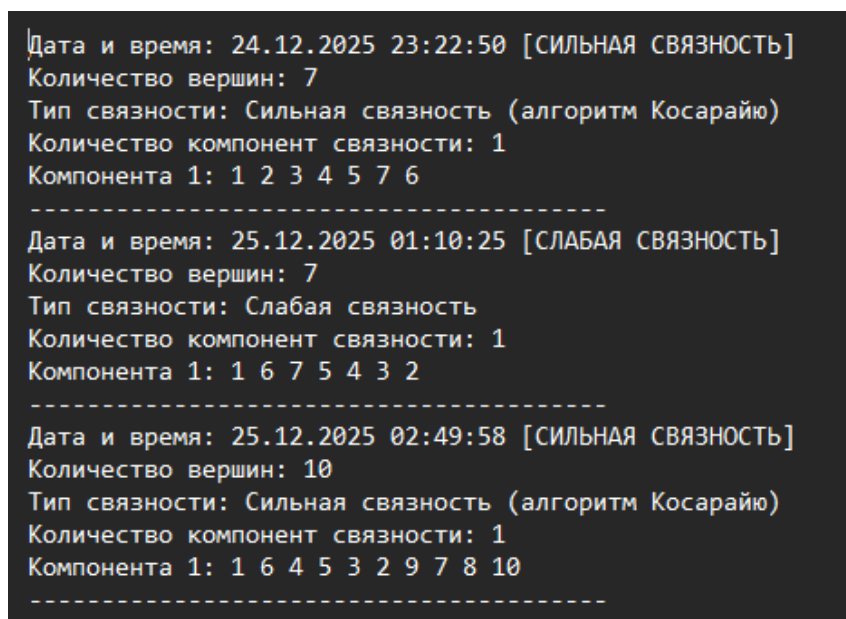


Рисунок 18 – Отображение в самом файле

4.3 Результаты

Результаты таблицы находятся в табл.2

Таблица 2 – Результаты поведения программы при тестировании

№	Описание теста	Полученный результат
1	Запуск программы	Верно
2	Введение параметров	Верно
3	Сильная связность графа	Верно
4	Слабая связность графа	Верно
5	Сохранение и загрузка Файлов	Верно

5 Ручной расчет алгоритма Косарайю

Рассмотрим ориентированный граф с 5 вершинами, заданный матрицей смежности вручную пользователем:

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	0
3	0	0	0	1	0
4	0	0	0	0	1
5	0	0	0	0	0

Расчет:

Сильная связность (алгоритм Косарайю):

1. DFS: Стек: [5, 4, 3, 2, 1]
2. Рёбра: $2 \rightarrow 1$, $1 \rightarrow 2$, $3 \rightarrow 2$, $4 \rightarrow 1$, $4 \rightarrow 3$, $5 \rightarrow 4$
3. DFS по стеку:
 - Из 5: $C_1 = \{5\}$
 - Из 4: $C_2 = \{4\}$
 - Из 3: $C_3 = \{3\}$
 - Из 2: $2 \rightarrow 1 \rightarrow 2$, $C_4 = \{1, 2\}$

Результат: 4 компоненты: $\{5\}$, $\{4\}$, $\{3\}$, $\{1,2\}$

Слабая связность:

- Неориентированные рёбра: 1–2, 1–4, 2–3, 3–4, 4–5
- Все вершины связаны: одна компонента $\{1,2,3,4,5\}$

Заключение

В процессе выполнения данной курсовой работы был проведен глубокий анализ и практическая реализация алгоритма Косарайю для выделения компонент сильной связности в ориентированных графах. В ходе разработки были закреплены и расширены навыки работы с фундаментальными алгоритмами теории графов, в частности, освоены методы двукратного применения поиска в глубину (DFS) и операции транспонирования графа, которые составляют основу исследуемого алгоритма.

В рамках работы была создана комплексная программа на языке C, реализующая не только классический алгоритм Косарайю для поиска компонент сильной связности, но и дополнительные алгоритмы для анализа различных типов связности в графах. Особое внимание уделялось образовательной составляющей — программа демонстрирует пошаговое выполнение алгоритма с подробными комментариями, что позволяет наглядно изучать его работу даже пользователям, не имеющим специальной подготовки в области теории графов.

Разработанное программное обеспечение успешно решает поставленную задачу, предоставляя следующие возможности:

1. Генерацию ориентированных и неориентированных графов с контролируемыми параметрами
2. Анализ трех типов связности: сильной (алгоритм Косарайю), слабой и обычной (для неориентированных графов)
3. Визуализацию структуры графа через матрицу смежности и списки связей
4. Сохранение истории вычислений с возможностью последующего анализа
5. Образовательный режим с подробным объяснением каждого шага алгоритма

Алгоритм Косарайю был тщательно протестирован на графах различной структуры: от небольших учебных примеров до графов со 100 вершинами. Тестирование подтвердило корректность реализации, устойчивость работы при различных входных данных и эффективность алгоритма для анализа графов умеренного размера. Особое внимание при тестировании уделялось граничным случаям: полностью связным графам (одна компонента), графам без рёбер (каждая вершина — отдельная компонента), графам с циклами и сложными взаимосвязями.

Практическая значимость работы заключается в том, что разработанная программа может быть использована:

- Как учебное пособие для изучения алгоритмов на графах в рамках курсов дискретной математики и алгоритмизации
- Как инструмент анализа реальных сетевых структур в научных исследованиях
- Как основа для более сложных систем анализа графов в прикладных областях

В ходе работы были также рассмотрены альтернативные подходы к решению задачи (алгоритм Тарьяна), что позволило провести сравнительный анализ методов и обосновать выбор алгоритма Косарайю для данной реализации.

Приобретенные в ходе выполнения курсовой работы навыки включают:

1. Умение эффективно реализовывать алгоритмы поиска в глубину
2. Навыки работы с матричными представлениями графов
3. Опыт создания интуитивно понятных консольных интерфейсов
4. Умение проектировать системы сохранения и восстановления данных
5. Навыки тестирования и отладки алгоритмически сложных программ

Таким образом, поставленные в курсовой работе задачи были успешно выполнены. Разработанная программа представляет собой законченное решение, сочетающее в себе алгоритмическую корректность, практическую полезность и образовательную ценность, что соответствует целям изучения дисциплины "Логика и основы алгоритмизации". Полученные результаты и опыт могут быть использованы в дальнейшем для решения более сложных задач анализа графов и разработки алгоритмического программного обеспечения.

Список литературы

1. <https://habr.com/ru/articles/847836/> (даты обращения 20.09.2025-20.12.2025)
2. <https://www.geeksforgeeks.org/dsa/prims-minimum-spanning-tree-mst-greedy-algo-5/> (даты обращения 20.09.2025-20.12.2025)
3. <https://github.com/Algorithms-and-Data-Structures-2021/semester-work-prim>(даты обращения 20.09.2025-20.12.2025)
4. <https://www.geeksforgeeks.org/c/prims-algorithm-in-c/> (даты обращения 20.09.2025-20.12.2025)
5. <https://studfile.net/preview/2919668/page:10/>(даты обращения 20.09.2025-20.12.2025)
6. https://neerc.ifmo.ru/wiki/index.php?title=Отношение_связности,_компоненты_связности(даты обращения 20.09.2025-20.12.2025)
7. <https://habr.com/ru/articles/537290/>(даты обращения 20.09.2025-20.12.2025)
- 8.

Приложение А. Листинг программы

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <windows.h>

#define MAX_VERTICES 100
#define MAX_HISTORY 100
#define FILENAME "search_history.txt"
#define GRAPH_FILENAME "current_graph.txt"

// Структуры данных
typedef struct {
    int size;
    int matrix[MAX_VERTICES][MAX_VERTICES];
    int transposed[MAX_VERTICES][MAX_VERTICES];
    int is_generated;
} Graph;

typedef struct {
    int vertices[MAX_VERTICES];
    int count;
} Component;

typedef struct {
    char timestamp[50];
    int vertex_count;
    int component_count;
    Component components[MAX_VERTICES];
} SearchResult;

// Глобальные переменные
SearchResult history[MAX_HISTORY];
int history_count = 0;
int N = 7;
Graph current_graph;

// Прототипы функций
void save_to_file(SearchResult result);
void save_current_graph_to_file();
void load_history();
void view_history();
void transpose_graph(Graph* g);
void fill_order(int v, int visited[], int stack[], int* index,
Graph* g);
```

```

void DFSUtil(int v, int visited[], Component* comp, Graph* g,
int use_transposed);
void kosaraju(Graph* g, SearchResult* result, int show_steps);
void find_weak_connectivity(Graph* g, SearchResult* result, int
show_steps);
void generate_directed_graph(Graph* g);
void manual_input_graph(Graph* g);
void set_vertex_count_and_generate();
void show_current_graph();
void print_matrix(Graph* g);
void print_connections(Graph* g);
void print_header();
void process_strong_connectivity();
void process_weak_connectivity();

// Вспомогательная функция для очистки буфера ввода
void clear_input_buffer() {
    int c;
    while ((c = getchar()) != '\n' && c != EOF);
}

// Функции для работы с файлами
void save_to_file(SearchResult result) {
    FILE* file = fopen(FILENAME, "a");
    if (file == NULL) {
        printf("Ошибка открытия файла!\n");
        return;
    }

    fprintf(file, "Дата и время: %s\n", result.timestamp);
    fprintf(file, "Количество вершин: %d\n",
result.vertex_count);
    fprintf(file, "Тип связности: ");

    // Определяем тип связности из названий функций
    if (strstr(result.timestamp, "СИЛЬНАЯ")) {
        fprintf(file, "Сильная связность (алгоритм
Косарайю)\n");
    }
    else if (strstr(result.timestamp, "СЛАБАЯ")) {
        fprintf(file, "Слабая связность\n");
    }
    else if (strstr(result.timestamp, "НЕОРИЕНТИРОВАННЫЙ")) {
        fprintf(file, "Обычная связность (неориентированный
граф)\n");
    }
    else {
        fprintf(file, "Не определен\n");
    }
}

```

```

        fprintf(file, "Количество компонент связности: %d\n",
result.component_count);

        for (int i = 0; i < result.component_count; i++) {
            fprintf(file, "Компонента %d: ", i + 1);
            for (int j = 0; j < result.components[i].count; j++) {
                fprintf(file, "%d ",
result.components[i].vertices[j] + 1);
            }
            fprintf(file, "\n");
        }
        fprintf(file, "-----\n");
        fclose(file);

        printf("Результаты сохранены в файл %s\n", FILENAME);
    }

void save_current_graph_to_file() {
    FILE* file = fopen(GRAPH_FILENAME, "w");
    if (file == NULL) {
        printf("Ошибка открытия файла для сохранения графа!\n");
        return;
    }

    fprintf(file, "Текущий граф (сохранен: ");
    time_t now = time(NULL);
    struct tm* t = localtime(&now);
    char timestamp[50];
    strftime(timestamp, sizeof(timestamp), "%d.%m.%Y %H:%M:%S",
t);
    fprintf(file, "%s)\n", timestamp);

    fprintf(file, "Количество вершин: %d\n",
current_graph.size);
    fprintf(file, "Матрица смежности:\n");

    for (int i = 0; i < current_graph.size; i++) {
        for (int j = 0; j < current_graph.size; j++) {
            fprintf(file, "%d ", current_graph.matrix[i][j]);
        }
        fprintf(file, "\n");
    }

    fprintf(file, "\nСписок исходящих связей:\n");
    for (int i = 0; i < current_graph.size; i++) {
        fprintf(file, "Вершина %d --> { ", i + 1);
        int has_connections = 0;
        for (int j = 0; j < current_graph.size; j++) {

```

```

        if (current_graph.matrix[i][j]) {
            fprintf(file, "%d ", j + 1);
            has_connections = 1;
        }
    }
    if (!has_connections) {
        fprintf(file, "нет исходящих связей");
    }
    fprintf(file, " }\n");
}

fclose(file);
printf("Текущий граф сохранен в файл %s\n", GRAPH_FILENAME);
}

void load_history() {
    FILE* file = fopen(FILENAME, "r");
    if (file == NULL) {
        history_count = 0;
        return;
    }

    char line[256];
    history_count = 0;

    while (fgets(line, sizeof(line), file) && history_count <
MAX_HISTORY) {
        if (strstr(line, "Дата и время:")) {
            sscanf(line, "Дата и время: %49[^\n]",
history[history_count].timestamp);
        }
        else if (strstr(line, "Количество вершин:")) {
            sscanf(line, "Количество вершин: %d",
&history[history_count].vertex_count);
        }
        else if (strstr(line, "Количество компонент
связности:")) {
            sscanf(line, "Количество компонент связности: %d",
&history[history_count].component_count);
        }
        else if (strstr(line, "Компонента")) {
            int comp_index;
            char comp_data[256];
            sscanf(line, "Компонента %d: %[^\n]", &comp_index,
comp_data);

            // Парсинг вершин компоненты
            char* token = strtok(comp_data, " ");
            int vertex_count = 0;

```

```

        while (token != NULL && vertex_count < MAX_VERTICES)
        {
            history[history_count].components[comp_index -
1].vertices[vertex_count] =
                atoi(token) - 1;
            vertex_count++;
            token = strtok(NULL, " ");
        }
        history[history_count].components[comp_index -
1].count = vertex_count;

        // Если это последняя компонента, увеличиваем
счетчик
        if (comp_index ==
history[history_count].component_count) {
            history_count++;
        }
    }

    fclose(file);
}

void view_history() {
    system("cls");
    printf("=== ИСТОРИЯ ПОИСКА ===\n\n");

    if (history_count == 0) {
        printf("История поиска пуста.\n");
        printf("\nНажмите любую клавишу для продолжения...");
        clear_input_buffer();
        getchar();
        return;
    }

    for (int i = 0; i < history_count; i++) {
        printf("Запись %d:\n", i + 1);
        printf("    Дата и время: %s\n", history[i].timestamp);
        printf("    Количество вершин: %d\n",
history[i].vertex_count);
        printf("    Количество компонент: %d\n",
history[i].component_count);

        for (int j = 0; j < history[i].component_count; j++) {
            printf("    Компонента %d: ", j + 1);
            for (int k = 0; k < history[i].components[j].count;
k++) {
                printf("%d ",
history[i].components[j].vertices[k] + 1);

```

```

        }
        printf("\n");
    }
    printf("-----\n");
}

printf("\nНажмите любую клавишу для продолжения...");
clear_input_buffer();
getchar();
}

// Функции алгоритма Косарайю
void transpose_graph(Graph* g) {
    if (g->size <= 10) {
        printf("\n=== ПРОЦЕСС ТРАНСПОНИРОВАНИЯ ГРАФА ===\n");
        printf("Исходная матрица смежности:\n");
        printf("    ");
        for (int j = 0; j < g->size; j++) printf("%3d", j + 1);
        printf("\n");
        for (int i = 0; i < g->size; i++) {
            printf("%3d", i + 1);
            for (int j = 0; j < g->size; j++) {
                printf("%3d", g->matrix[i][j]);
            }
            printf("\n");
        }
    }

    // Транспонирование: меняем строки и столбцы местами
    for (int i = 0; i < g->size; i++) {
        for (int j = 0; j < g->size; j++) {
            g->transposed[j][i] = g->matrix[i][j];
        }
    }

    if (g->size <= 10) {
        printf("\nТранспонированная матрица смежности:\n");
        printf("    ");
        for (int j = 0; j < g->size; j++) printf("%3d", j + 1);
        printf("\n");
        for (int i = 0; i < g->size; i++) {
            printf("%3d", i + 1);
            for (int j = 0; j < g->size; j++) {
                printf("%3d", g->transposed[i][j]);
            }
            printf("\n");
        }

        // Наглядное объяснение

```

```

        printf("\nНаглядное представление транспонирования:\n");
        printf("Каждое ребро меняет направление на
противоположное:\n");
        for (int i = 0; i < g->size; i++) {
            for (int j = 0; j < g->size; j++) {
                if (g->matrix[i][j] == 1) {
                    printf("    Ребро %d --> %d становится %d -->
%d\n",
                                i + 1, j + 1, j + 1, i + 1);
                }
            }
        }
        printf("Нажмите любую клавишу для продолжения...\n");
        clear_input_buffer();
        getchar();
    }
}

void fill_order(int v, int visited[], int stack[], int* index,
Graph* g) {
    visited[v] = 1;

    for (int i = 0; i < g->size; i++) {
        if (g->matrix[v][i] && !visited[i]) {
            if (g->size <= 10) {
                printf("    Переход из вершины %d в вершину %d\n",
v + 1, i + 1);
            }
            fill_order(i, visited, stack, index, g);
        }
    }

    // Добавляем вершину в стек после обработки всех её соседей
    stack[++(*index)] = v;
    if (g->size <= 10) {
        printf("    Вершина %d полностью обработана и добавлена в
стек\n", v + 1);
    }
}

void DFSUtil(int v, int visited[], Component* comp, Graph* g,
int use_transposed) {
    visited[v] = 1;
    comp->vertices[comp->count++] = v;

    for (int i = 0; i < g->size; i++) {
        int edge_exists = use_transposed ? g->transposed[v][i] :
g->matrix[v][i];
        if (edge_exists && !visited[i]) {

```

```

        if (g->size <= 10) {
            printf("  Переход из вершины %d в вершину %d\n",
v + 1, i + 1);
        }
        DFSUtil(i, visited, comp, g, use_transposed);
    }
}

void kosaraju(Graph* g, SearchResult* result, int show_steps) {
    int stack[MAX_VERTICES];
    int stack_index = -1;
    int visited[MAX_VERTICES];

    // Инициализация массива посещенных вершин
    for (int i = 0; i < g->size; i++) {
        visited[i] = 0;
    }

    if (show_steps && g->size <= 10) {
        printf("\n=====\\n");
        printf("ШАГ 1: ПЕРВЫЙ ОБХОД В ГЛУБИНУ (DFS)\\n");
        printf("Цель: Определить порядок завершения обработки
вершин\\n");
        printf("Стек заполняется в порядке убывания времени
завершения\\n");
        printf("=====\\n");
    }

    // ШАГ 1: Первый обход в глубину для заполнения стека
    for (int i = 0; i < g->size; i++) {
        if (!visited[i]) {
            if (show_steps && g->size <= 10) {
                printf("\\nНачинаем обход из вершины %d:\\n", i +
1);
            }
            fill_order(i, visited, stack, &stack_index, g);
        }
    }

    if (show_steps && g->size <= 10) {
        printf("\\nСодержимое стека (вершины в порядке завершения
обработки):\\n");
        printf("Вершины: ");
        for (int i = 0; i <= stack_index; i++) {
            printf("%d ", stack[i] + 1);
        }
        printf("\\n");
        printf("(Вершина вверху стека: %d, внизу: %d)\\n",

```

```

        stack[stack_index] + 1, stack[0] + 1);
printf("\nНажмите любую клавишу для продолжения...\n");
clear_input_buffer();
getchar();
}

// ШАГ 2: Транспонирование графа
transpose_graph(g);

// Сброс массива посещенных вершин для второго обхода
for (int i = 0; i < g->size; i++) {
    visited[i] = 0;
}

if (show_steps && g->size <= 10) {
    printf("\n=====\\n");
    printf("ШАГ 3: ВТОРОЙ ОБХОД В ГЛУБИНУ\\n");
    printf("Цель: Найти компоненты сильной связности\\n");
    printf("Обход идет по транспонированному графу в порядке
из стека\\n");
    printf("Каждое дерево поиска - одна компонента сильной
связности\\n");
    printf("=====\\n");
}

// ШАГ 3: Второй обход в глубину по порядку из стека
result->component_count = 0;
while (stack_index >= 0) {
    int v = stack[stack_index--];

    if (!visited[v]) {
        if (show_steps && g->size <= 10) {
            printf("\n--- Начинаем обход из вершины %d
(корень компоненты) ---\\n", v + 1);
        }

        Component new_comp;
        new_comp.count = 0;
        DFSUtil(v, visited, &new_comp, g, 1);

        // Сохраняем найденную компоненту
        result->components[result->component_count] =
new_comp;

        if (show_steps && g->size <= 10) {
            printf("Найдена компонента сильной связности %d:
", result->component_count + 1);
            for (int j = 0; j < new_comp.count; j++) {
                printf("%d ", new_comp.vertices[j] + 1);
            }
        }
    }
}

```

```

        }
        printf("\n");
    }

    result->component_count++;
}

// Заполняем информацию о результате поиска
result->vertex_count = g->size;
time_t now = time(NULL);
struct tm* t = localtime(&now);
strftime(result->timestamp, sizeof(result->timestamp),
    "%d.%m.%Y %H:%M:%S [СИЛЬНАЯ СВЯЗНОСТЬ]", t);
}

// Функция для поиска слабой связности в орграфе
void find_weak_connectivity(Graph* g, SearchResult* result, int
show_steps) {
    int visited[MAX_VERTICES] = { 0 };
    result->component_count = 0;
    result->vertex_count = g->size;

    time_t now = time(NULL);
    struct tm* t = localtime(&now);
    strftime(result->timestamp, sizeof(result->timestamp),
        "%d.%m.%Y %H:%M:%S [СЛАБАЯ СВЯЗНОСТЬ]", t);

    if (show_steps && g->size <= 10) {
        printf("\n=====\\n");
        printf("АЛГОРИТМ ПОИСКА СЛАБОЙ СВЯЗНОСТИ\\n");
        printf("=====\\n");
        printf("Принцип: игнорируем направление рёбер\\n");
        printf("Считаем граф неориентированным:\\n");
        printf("- Если есть ребро i->j, считаем связь i<->j\\n");
        printf("- Если есть ребро j->i, считаем связь i<->j\\n");
        printf("=====\\n");
    }

    for (int i = 0; i < g->size; i++) {
        if (!visited[i]) {
            Component comp;
            comp.count = 0;

            // Используем стек для DFS
            int stack[MAX_VERTICES];
            int top = -1;

            stack[++top] = i;

```

```

        visited[i] = 1;

        if (show_steps && g->size <= 10) {
            printf("\nНачинаем обход из вершины %d (корень
компоненты %d):\n",
                i + 1, result->component_count + 1);
        }

        while (top >= 0) {
            int v = stack[top--];
            comp.vertices[comp.count++] = v;

            if (show_steps && g->size <= 10) {
                printf("    Обрабатываем вершину %d\n", v +
1);
            }

            // Проверяем все вершины, связанные в ЛЮБОМ
направлении
            for (int j = 0; j < g->size; j++) {
                // Если есть ребро в ЛЮБОМ направлении
                if ((g->matrix[v][j] == 1 || g->matrix[j][v]
== 1) && !visited[j]) {
                    visited[j] = 1;
                    stack[++top] = j;

                    if (show_steps && g->size <= 10) {
                        if (g->matrix[v][j] == 1 && g-
>matrix[j][v] == 1) {
                            printf("        Переход к вершине %d
(двунаправленная связь)\n", j + 1);
                        }
                        else if (g->matrix[v][j] == 1) {
                            printf("        Переход к вершине %d
(ребро %d->%d)\n", j + 1, v + 1, j + 1);
                        }
                        else {
                            printf("        Переход к вершине %d
(обратное ребро %d->%d)\n",
                                j + 1, j + 1, v + 1);
                        }
                    }
                }
            }
        }

        // Сохраняем найденную компоненту
        result->components[result->component_count] = comp;

```

```

        if (show_steps && g->size <= 10) {
            printf("  Найдена компонента слабой связности
%d: { ",
                result->component_count + 1);
            for (int j = 0; j < comp.count; j++) {
                printf("%d ", comp.vertices[j] + 1);
            }
            printf("} (размер: %d)\n", comp.count);
        }

        result->component_count++;
    }
}

// Функции для работы с графами
void generate_directed_graph(Graph* g) {
    srand(time(NULL));

    // ШАГ 1: Инициализация всей матрицы нулями
    for (int i = 0; i < g->size; i++) {
        for (int j = 0; j < g->size; j++) {
            g->matrix[i][j] = 0;
        }
    }

    // ШАГ 2: Генерация рёбер
    printf("\nГенерация ориентированного графа...\n");
    printf("Правила генерации:\n");
    printf("1. Петли (ребра из вершины в себя) не
генерируются\n");
    printf("2. Вероятность создания ребра между двумя разными
вершинами: 40%%\n");
    printf("3. Ребра могут быть однонаправленными или
двунаправленными\n");

    int edge_count = 0;
    for (int i = 0; i < g->size; i++) {
        for (int j = 0; j < g->size; j++) {
            if (i != j) { // Исключаем петли
                if (i < j) { // Обработываем только верхний
треугольник для избежания дублирования
                    int probability = rand() % 100;

                    if (probability < 40) {
                        // Создаем однонаправленное или
двунаправленное ребро
                        int direction = rand() % 3;

```

```

        switch (direction) {
        case 0: // i → j
            g->matrix[i][j] = 1;
            printf("    Создано ребро: %d ->
%d\n", i + 1, j + 1);
            edge_count++;
            break;
        case 1: // j → i
            g->matrix[j][i] = 1;
            printf("    Создано ребро: %d ->
%d\n", j + 1, i + 1);
            edge_count++;
            break;
        case 2: // i ↔ j (двунаправленное)
            g->matrix[i][j] = 1;
            g->matrix[j][i] = 1;
            printf("    Создано двунаправленное
ребро: %d <-> %d\n", i + 1, j + 1);
            edge_count += 2;
            break;
        }
    }
}

printf("Всего создано ребер: %d\n", edge_count);

// ШАГ 3: Гарантируем, что у каждой вершины есть хотя бы
одно исходящее ребро
for (int i = 0; i < g->size; i++) {
    int has_outgoing = 0;
    for (int j = 0; j < g->size; j++) {
        if (g->matrix[i][j] == 1) {
            has_outgoing = 1;
            break;
        }
    }

    if (!has_outgoing && g->size > 1) {
        // Добавляем случайное исходящее ребро
        int target;
        do {
            target = rand() % g->size;
        } while (target == i);

        g->matrix[i][target] = 1;
    }
}

```

```

        printf("    Добавлено обязательное ребро: %d → %d\n",
i + 1, target + 1);
        edge_count++;
    }
}

g->is_generated = 1;
// Автоматически сохраняем граф в файл при генерации
save_current_graph_to_file();
}

void manual_input_graph(Graph* g) {
    printf("\n=== РУЧНОЙ ВВОД ГРАФА ===\n");
    printf("Введите матрицу смежности (%dx%d):\n", g->size, g-
>size);
    printf("(0 - нет ребра, 1 - есть ребро)\n\n");

    for (int i = 0; i < g->size; i++) {
        printf("Строка %d (вершина %d): ", i + 1, i + 1);
        for (int j = 0; j < g->size; j++) {
            int value;
            scanf("%d", &value);
            if (value != 0 && value != 1) {
                printf("Ошибка: введите 0 или 1!\n");
                j--;
                continue;
            }
            g->matrix[i][j] = value;
        }
    }

    g->is_generated = 1;
    printf("\nГраф успешно введен вручную!\n");
    // Автоматически сохраняем граф в файл при ручном вводе
    save_current_graph_to_file();
}

void set_vertex_count_and_generate() {
    printf("Текущее количество вершин: %d\n", N);
    printf("Введите новое количество вершин (2-%d): ",
MAX_VERTICES);

    int new_count;
    if (scanf("%d", &new_count) != 1) {
        printf("Некорректный ввод!\n");
        clear_input_buffer();
        printf("Нажмите любую клавишу для продолжения...");
        getchar();
        return;
    }
}

```

```

    }

    if (new_count >= 2 && new_count <= MAX_VERTICES) {
        N = new_count;
        current_graph.size = N;
        current_graph.is_generated = 0;

        printf("Количество вершин изменено на %d\n", N);

        int choice;
        printf("\nВыберите способ создания графа:\n");
        printf("1. Сгенерировать случайный граф\n");
        printf("2. Ввести граф вручную\n");
        printf("Выберите (1 или 2): ");
        scanf("%d", &choice);

        if (choice == 1) {
            generate_directed_graph(&current_graph);
            printf("\nГраф успешно сгенерирован и сохранен в
файл!\n");
        }
        else if (choice == 2) {
            manual_input_graph(&current_graph);
        }
        else {
            printf("Неверный выбор. Граф не создан.\n");
        }
    }
    else {
        printf("Некорректное значение! Допустимый диапазон: 2-
%d\n", MAX_VERTICES);
    }

    printf("Нажмите любую клавишу для продолжения...");
    clear_input_buffer();
    getchar();
}

void show_current_graph() {
    system("cls");
    printf("=== ТЕКУЩИЙ ГРАФ ===\n\n");

    if (!current_graph.is_generated) {
        printf("Граф еще не создан!\n");
        printf("Используйте пункт меню 'Установить количество
вершин и создать граф'\n");
    }
    else {
        printf("Количество вершин: %d\n", current_graph.size);
    }
}

```

```

        printf("(Граф автоматически сохранен в файле %s)\n",
GRAPH_FILENAME);

        print_matrix(&current_graph);
        print_connections(&current_graph);
    }

    printf("\nНажмите любую клавишу для продолжения...");
    clear_input_buffer();
    getchar();
}

void print_matrix(Graph* g) {
    printf("\n=== МАТРИЦА СМЕЖНОСТИ ===\n");
    printf("(1 - есть ребро, 0 - нет ребра)\n\n");

    printf("      ");
    for (int j = 0; j < g->size; j++) {
        printf("%4d  ", j + 1);
    }
    printf("\n");

    printf("      ");
    for (int j = 0; j < g->size; j++) {
        printf("-----");
    }
    printf("\n\n");

    // Строки матрицы
    for (int i = 0; i < g->size; i++) {
        printf("%3d |", i + 1);
        for (int j = 0; j < g->size; j++) {
            printf("%4d  ", g->matrix[i][j]);
        }
        printf("\n\n");
    }
}

void print_connections(Graph* g) {
    printf("\n=== СПИСОК ИСХОДЯЩИХ СВЯЗЕЙ ===\n");
    printf("(Для ориентированного графа)\n\n");

    for (int i = 0; i < g->size; i++) {
        printf("Вершина %d --> { ", i + 1);
        int has_connections = 0;

        for (int j = 0; j < g->size; j++) {
            if (g->matrix[i][j]) {
                printf("%d ", j + 1);
            }
        }
    }
}

```

```

        has_connections = 1;
    }
}

    if (!has_connections) {
        printf("нет исходящих связей");
    }
    printf(" }\n");
}

}

// функции меню
void print_header() {
    system("cls");

    printf("=====\n");
    printf("          КУРСОВАЯ РАБОТА\n");
    printf("          по дисциплине \"Логика и основы\n\n\n");
    printf("          Тема: Реализация алгоритма поиска\n\n\n");
    printf("          компонент связности\n");
    printf("          в орграфе, используя поиск в\n\n\n");
    printf("          Студент: Тусков Арсений Андреевич\n");
    printf("          Группа: 24-BBВ1\n");
    printf("          Преподаватель: Юрова О.В.\n");

    printf("=====\n\n");

    // Информация о текущем графе
    if (current_graph.is_generated) {
        printf("Текущий граф: создан (%d вершин), сохранен в\n\n",
            current_graph.size, GRAPH_FILENAME);
    }
    else {
        printf("Текущий граф: не создан\n");
    }

    printf("=====\n\n");
}

void process_strong_connectivity() {
    if (!current_graph.is_generated) {
        printf("\nОШИБКА: Граф не создан!\n");
        printf("Сначала создайте граф через пункт меню\n\n");
        printf("'Установить количество вершин и создать граф'\n");
    }
}

```

```

        printf("\nНажмите любую клавишу для продолжения...");
        clear_input_buffer();
        getchar();
        return;
    }

    printf("\n=== СИЛЬНАЯ СВЯЗНОСТЬ ОРИЕНТИРОВАННОГО ГРАФА
===\n");
    printf("(Алгоритм Косарайю)\n");
    printf("Используется текущий граф (%d вершин)\n\n",
current_graph.size);

    print_matrix(&current_graph);
    print_connections(&current_graph);

    int show_steps;
    printf("\nПоказывать подробные шаги алгоритма Косарайю? (1 -
да, 0 - нет): ");
    scanf("%d", &show_steps);

    SearchResult result;
    // Инициализация структуры результата
    result.component_count = 0;
    result.vertex_count = current_graph.size;

    kosaraju(&current_graph, &result, show_steps);

    printf("\n===== \n");
    printf("ФИНАЛЬНЫЕ РЕЗУЛЬТАТЫ (СИЛЬНАЯ СВЯЗНОСТЬ)\n");
    printf("===== \n");
    printf("Количество компонент сильной связности: %d\n",
result.component_count);
    printf("Всего вершин в графе: %d\n", result.vertex_count);

    if (result.component_count == 1) {
        printf("\nГраф является СИЛЬНО СВЯЗНЫМ!\n");
        printf("Любая вершина достижима из любой другой.\n");
    }
    else if (result.component_count == result.vertex_count) {
        printf("\nКаждая вершина образует отдельную
компоненту.\n");
        printf("В графе нет сильных связей между вершинами.\n");
    }

    printf("\nКомпоненты сильной связности:\n");
    for (int i = 0; i < result.component_count; i++) {
        printf("Компонента %d (размер: %d): { ", i + 1,
result.components[i].count);
        for (int j = 0; j < result.components[i].count; j++) {

```

```

        printf("%d ", result.components[i].vertices[j] + 1);
    }
    printf("}\n");
}

char save;
printf("\nСохранить результаты в файл? (y/n): ");
clear_input_buffer();
scanf(" %c", &save);

if (save == 'y' || save == 'Y') {
    save_to_file(result);
    if (history_count < MAX_HISTORY) {
        history[history_count++] = result;
    }
    else {
        printf("История переполнена, результат не добавлен в историю.\n");
    }
}

printf("\nНажмите любую клавишу для продолжения...");
clear_input_buffer();
getchar();
}

void process_weak_connectivity() {
    if (!current_graph.is_generated) {
        printf("\nОШИБКА: Граф не создан!\n");
        printf("Сначала создайте граф через пункт меню 'Установить количество вершин и создать граф'\n");
        printf("\nНажмите любую клавишу для продолжения...");
        clear_input_buffer();
        getchar();
        return;
    }

    printf("\n=== СЛАБАЯ СВЯЗНОСТЬ ОРИЕНТИРОВАННОГО ГРАФА\n");
    printf("\n(Игнорируем направление рёбер)\n");
    printf("Используется текущий граф (%d вершин)\n\n", current_graph.size);

    print_matrix(&current_graph);
    print_connections(&current_graph);

    int show_steps;
    printf("\nПоказывать подробные шаги алгоритма? (1 - да, 0 - нет): ");

```

```

scanf("%d", &show_steps);

SearchResult result;
// Инициализация структуры результата
result.component_count = 0;
result.vertex_count = current_graph.size;

find_weak_connectivity(&current_graph, &result, show_steps);

printf("\n=====\\n");
printf("ФИНАЛЬНЫЕ РЕЗУЛЬТАТЫ (СЛАБАЯ СВЯЗНОСТЬ)\\n");
printf("=====\\n");
printf("Количество компонент слабой связности: %d\\n",
result.component_count);
printf("Всего вершин в графе: %d\\n", result.vertex_count);

if (result.component_count == 1) {
    printf("\\nГраф является СЛАБО СВЯЗНЫМ!\\n");
    printf("Если игнорировать направление рёбер, граф
связен.\\n");
}
else if (result.component_count == result.vertex_count) {
    printf("\\nКаждая вершина изолирована.\\n");
    printf("Даже без учета направления рёбер вершины не
связаны.\\n");
}

printf("\\nКомпоненты слабой связности:\\n");
for (int i = 0; i < result.component_count; i++) {
    printf("Компонента %d (размер: %d): { ", i + 1,
result.components[i].count);
    for (int j = 0; j < result.components[i].count; j++) {
        printf("%d ", result.components[i].vertices[j] + 1);
    }
    printf("}\\n");
}

// ДОБАВЛЕНО: Запрос на сохранение результатов
char save;
printf("\\nСохранить результаты в файл? (y/n): ");
clear_input_buffer();
scanf(" %c", &save);

if (save == 'y' || save == 'Y') {
    save_to_file(result);
    if (history_count < MAX_HISTORY) {
        history[history_count++] = result;
    }
    else {

```

```

        printf("История переполнена, результат не добавлен в
историю.\n");
    }
}

// ДОБАВЛЕНО: Ожидание нажатия клавиши
printf("\nНажмите любую клавишу для продолжения...");
clear_input_buffer();
getchar();
}

int main() {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    // Инициализация текущего графа
    current_graph.size = N;
    current_graph.is_generated = 0;

    // Инициализация матриц нулями
    for (int i = 0; i < MAX_VERTICES; i++) {
        for (int j = 0; j < MAX_VERTICES; j++) {
            current_graph.matrix[i][j] = 0;
            current_graph.transposed[i][j] = 0;
        }
    }

    load_history();

    while (1) {
        print_header();

        printf("МЕНЮ:\n");
        printf("1. Установить количество вершин и создать
граф\n");
        printf("2. Показать текущий граф\n");
        printf("3. Сильная связность (алгоритм Косарайю)\n");
        printf("4. Слабая связность\n");
        printf("5. Просмотреть историю поиска\n");
        printf("6. Выход\n");
        printf("\nВыберите пункт меню: ");

        int choice;
        if (scanf("%d", &choice) != 1) {
            printf("Некорректный ввод!\n");
            clear_input_buffer();
            printf("Нажмите любую клавишу для продолжения...");
            getchar();
            continue;
        }
    }
}

```

```

    }

    switch (choice) {
    case 1:
        set_vertex_count_and_generate();
        break;
    case 2:
        show_current_graph();
        break;
    case 3:
        process_strong_connectivity();
        break;
    case 4:
        process_weak_connectivity();
        break;
    case 5:
        view_history();
        break;
    case 6:
        printf("Выход из программы...\n");
        return 0;
    default:
        printf("Некорректный выбор! Выберите пункт от 1 до
6.\n");

        printf("Нажмите любую клавишу для продолжения...");
        clear_input_buffer();
        getchar();
        break;
    }
}

return 0;
}

```