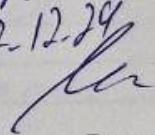


Министерство образования Российской Федерации
Пензенский государственный университет
Кафедра «Вычислительная техника»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К курсовому проектированию
по курсу «Логика и основы алгоритмизации в инженерных задачах»
на тему «Реализация алгоритма поиска Гамильтоновых циклов»

отп.
27.12.24


Выполнила:

Студент группы 23BBB3 Кузнецов К.И.

Принял:

д. т. н. Митрохин М.А.

Пенза 2024

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет Вычислительной техники
Кафедра "Вычислительная техника"

"УТВЕРЖДАЮ"

Зав. кафедрой ВТ

« 17 » 09 2024

ЗАДАНИЕ

на курсовое проектирование по курсу

Логика и основы алгоритмизации вычислительных задач
Студенту Кузнецову Кириллу Игоревичу Группа 23ВВВЗ
Тема проекта Реализация алгоритма нахождения
Гамильтоновых циклов

Исходные данные (технические требования) на проектирование

Разработка алгоритмов и программного обеспечения в соответствии с данными заданием курсового проекта.

Представительная записка должна содержать:

1. Постановку задачи;
2. Теоретическую часть задания;
3. Описание алгоритма поставленной задачи;
4. Пример ручного расчета задачи и вычислений (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 расчет работы алгоритма.

2. Графическая часть

Схема алгоритма в формате блок-схем.

3. Экспериментальная часть

Тестирование программы:
Результаты работы программы
на тестовых данных.

Срок выполнения проекта по разделам

- 1 Исследование теоретической части курсового
- 2 Разработка алгоритмов программы
- 3 Разработка программы
- 4 Тестирование и завершение разработки программы
- 5 Оформление конструкторской записки
- 6
- 7
- 8

Дата выдачи задания "17" 09.24

Дата защиты проекта " " "

Руководитель

Задание получил

"29" 10

2024 г.

Студент

Кур

Содержание

Введение.....	6
Постановка задачи	8
Описание алгоритма программы	9
Описание программы	13
Тестирование	18
Ручной расчёт задачи.....	22
Заключение	24
Список литературы.....	25
Приложение А.....	26

Введение

Данная курсовая работа посвящена разработке программы, реализующей алгоритм поиска Гамильтоновых циклов в графе. Гамильтонов цикл — это цикл в графе, который проходит через каждую вершину ровно один раз и возвращается в начальную вершину. Программа позволяет задавать граф в виде списка смежности, осуществлять поиск всех Гамильтоновых циклов и выводить результаты на экран или сохранять их в текстовый файл.

Работа основана на использовании рекурсивного алгоритма поиска с возвратом (backtracking), оптимизированного для уменьшения числа проверок и исключения заведомо некорректных путей. Реализация также включает возможность проверки заданных ограничений на граф, таких как наличие циклов в графах определенного типа (например, в ориентированных или неориентированных графах).

Программа, реализующая поиск Гамильтоновых циклов, может найти применение в различных областях:

- a) Комбинаторная оптимизация: Решение задач оптимального проектирования сетей, таких как маршрутизация, планирование и логистика, где необходимо найти путь, покрывающий все ключевые точки.
- b) Теория графов: Анализ структур графов и изучение их свойств. Программа может использоваться для проверки гипотез и построения новых теоретических моделей.
- c) Тестирование и отладка: Проверка и тестирование алгоритмов для работы с графами. Программа может быть использована для генерации данных для проверки корректности работы других алгоритмов.
- d) Образовательные цели: Программа предоставляет возможность изучать свойства графов, алгоритмы поиска и оптимизации на практике.

Для реализации проекта выбран язык программирования C++. Этот выбор обусловлен следующими факторами:

1. Высокая производительность: C++ обеспечивает эффективное использование памяти и ресурсов, что особенно важно для работы с алгоритмами на графах, где сложность может расти экспоненциально.
2. Гибкость в реализации алгоритмов: Стандартная библиотека C++ предоставляет широкий набор инструментов для работы с данными, таких как контейнеры STL, которые используются для хранения графа и организации поиска.
3. Модульность и структурированность кода: Использование объектно-ориентированного подхода (ООП) позволяет выделить отдельные модули для работы с графами, оптимизации алгоритмов и ввода-вывода, что делает код удобным для сопровождения.
4. Поддержка платформ и доступность ресурсов: Большое сообщество разработчиков и наличие обширной документации по языку C++ позволяют быстро находить ответы на возникающие вопросы и решать задачи разработки.

Программа разработана с акцентом на простоту использования и надежность, что делает её полезным инструментом для решения как практических, так и теоретических задач, связанных с Гамильтоновыми циклами.

Постановка задачи

Необходимо разработать консольное приложение на языке C++, которое выполняет следующие действия:

- a) Позволяет выбрать способ задачи графа: генерировать случайно или ввести вручную.
- b) Представляет пользователю выбрать вручную размер графа(nG).
- c) Генерирует случайный граф размера nG , используя матрицу смежности. Веса рёбер определяются случайным образом.
- d) Осуществляет поиск всех Гамильтоновых циклов в графе с использованием рекурсивного алгоритма с возвратом, проверяющего возможность добавления каждой вершины в путь.
- e) Позволяет пользователю выбрать начальную вершину для поиска цикла.
- f) Рассчитывает вес каждого найденного цикла и выделяет цикл с минимальным весом.
- g) Отображает найденные циклы и их веса в консоль.
- h) Визуализирует граф с помощью графического интерфейса.

Описание алгоритма программы

Для реализации программы по поиску Гамильтоновых циклов в графе использованы следующие 5 функций: *createG* – функция, которая создает случайный граф в виде матрицы смежности, *isSafe* – проверяет, можно ли добавить вершину на текущую позицию пути, *hamiltonianCycleUtil* – формирует список всех возможных Гамильтоновых циклов, *calculateCycleWeight* - вычисляет общий вес цикла, включая возврат к стартовой вершине, *findAllHamiltonianCycles* – функция, которая объединяет весь процесс: от поиска циклов до вывода результатов.

Программа запрашивает у пользователя способ задачи графа и размер графа(*nG*) от 1 до 20. Программа просит ввести матрицу смежности в ручную или автоматически генерирует граф размером *nG* x *nG* с помощью функции *createG*. Пользователю предлагается ввести стартовую вершину, после чего начинается поиск Гамильтоновых циклов.

Описание функций:

1. *createG*: Создает случайный граф, представленный матрицей смежности. Вес ребер – случайные числа от 1 до 7, отсутствующие рёбра представлены нулями. Граф генерируется таким образом, чтобы ребра между разными вершинами добавлялись случайно.
2. *isSafe*: Проверяет, можно ли добавить вершину в текущий путь, опираясь на два критерия: Между текущей вершиной пути и новой вершиной существует ребро. Новая вершина ещё не добавлена в путь.
3. *hamiltonianCycleUtil*: Рекурсивно формирует список всех возможных Гамильтоновых циклов. Если текущий путь покрывает все вершины и возвращается к стартовой, он добавляется в список циклов.

4. *calculateCycleWeight*: Суммирует веса рёбер для заданного цикла, включая возврат к стартовой вершине. Используется для оценки оптимального цикла.
5. *findAllHamiltonianCycles*: Управляет процессом поиска циклов. Она запускает вспомогательные функции, анализирует найденные циклы и определяет цикл с минимальным весом.

Ниже представлен псевдокод всех функций.

Функция *createG*(целое число *size*, двумерный массив *G*):

Для каждого целого числа *i* от 0 до *size*-1:

Для каждого целого числа *j* от 0 до *size*-1:

ЕСЛИ *i* не равно *j* И случайное значение % 2 == 0:

Установить *G*[*i*][*j*] равным случайному числу от 1 до 7

ИНАЧЕ:

Установить *G*[*i*][*j*] равным 0

Конец цикла

Вернуть *G*

Конец функции

Функция *isSafe*(целое число *v*, двумерный массив *G*, список *path*, целое число *pos*):

ЕСЛИ *G*[*path*[*pos*-1]][*v*] == 0 ИЛИ *v* уже содержится в *path*:

Вернуть false

Вернуть true

Конец функции

Функция *hamiltonianCycleUtil*(двумерный массив *G*, список *path*, целое число *pos*, целое число *size*, список *allPaths*):

ЕСЛИ *pos* == *size*:

ЕСЛИ *G*[*path*[*pos*-1]][*path*[0]] > 0:

Добавить *path* в *allPaths*

Вернуться

ИНАЧЕ:

Для каждого целого числа v от 1 до $size-1$:

ЕСЛИ $isSafe(v, G, path, pos)$:

Установить $path[pos]$ равным v

Вызвать $hamiltonianCycleUtil(G, path, pos+1, size, allPaths)$ //

Рекурсивный вызов

Установить $path[pos]$ равным -1 // Вернуть в исходное состояние

Конец цикла

Конец ЕСЛИ

Конец функции

Функция $calculateCycleWeight$ (двумерный массив G , список $path$):

Инициализировать целое число $weight$ равным 0

Для каждого целого числа i от 0 до $size-2$:

Добавить $G[path[i]][path[i+1]]$ к $weight$

Добавить $G[path[size-1]][path[0]]$ к $weight$

Вернуть $weight$

Конец функции

Функция $findAllHamiltonianCycles$ (двумерный массив G , целое число $size$, целое число $start$):

Создать список $path$ размером $size$, заполненный значениями -1

Установить $path[0]$ равным $start$

Создать пустой список $allPaths$

Вызвать $hamiltonianCycleUtil(G, path, 1, size, allPaths)$

ЕСЛИ $allPaths$ пуст:

Вывести "Гамильтонов циклов не существует"

ИНАЧЕ:

Инициализировать целое число minWeight равным INT_MAX

Для каждого цикла в allPaths:

Вывести цикл

Вычислить вес цикла с помощью calculateCycleWeight

ЕСЛИ вес цикла меньше minWeight:

Обновить minWeight и лучший цикл bestCycle

Вывести "Самый короткий цикл с минимальным весом: ", bestCycle и minWeight

Конец функции

Описание программы

Программа начинается с создания случайного графа с помощью функции `createG(int size)`, которая генерирует матрицу смежности для графа с заданным числом вершин. Для каждой пары вершин в графе создается ребро с случайным весом от 1 до 7. Ребра между одинаковыми вершинами не создаются, и в соответствующих ячейках матрицы будет стоять 0.

```
int** createG(int size) {
    int** G = new int* [size];
    for (int i = 0; i < size; i++) {
        G[i] = new int[size];
        for (int j = 0; j < size; j++) {
            G[i][j] = (i != j && rand() % 2 == 0) ? rand() % 7 + 1 : 0;
        }
    }
    return G;
}
```

После того как граф создан, его структура выводится на экран с помощью функции `printG`. Эта функция выводит двумерную матрицу смежности, где значения от 1 до 7 показывают веса ребер, а нули обозначают отсутствие ребра.

```
void printG(int** G, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            cout << G[i][j] << " ";
        }
        cout << endl;
    }
}
```

Функция `hamiltonianCycleUtil` играет ключевую роль в поиске гамильтоновых циклов в графе. Она использует рекурсивный алгоритм для построения всех возможных путей, которые могут стать гамильтоновыми циклами, начиная с определенной вершины.

Когда функция вызывается, она пытается найти путь длиной `pos` из всех возможных вершин графа, начиная с вершины 0 и двигаясь по графу поочередно, добавляя новые вершины к текущему пути. Аргумент `pos`

указывает на текущую позицию в пути, то есть количество вершин, которые уже были добавлены в текущую цепочку.

Для каждой вершины на текущем шаге алгоритм проверяет два условия. Во-первых, необходимо удостовериться, что вершина не входит в текущий путь, чтобы не нарушать условие гамильтонова цикла (каждая вершина может встречаться в пути только один раз). Для этого используется проверка с помощью вектора `path`, который хранит список уже посещенных вершин. Во-вторых, проверяется наличие ребра между текущей вершиной и предыдущей (если ребро отсутствует, этот путь отклоняется).

Когда находит подходящую вершину для добавления в путь, функция рекурсивно вызывает себя с увеличением позиции в пути, переходя к следующей вершине. Этот процесс продолжается до тех пор, пока не будет найдено решение (путь длиной `size` вершин). Если путь включает все вершины, проверяется наличие ребра между последней и первой вершинами, чтобы замкнуть цикл. Если такое ребро существует, найден гамильтонов цикл, и путь добавляется в список всех найденных циклов `allPaths`.

Если на некотором шаге пути не удастся найти подходящую вершину (то есть нет доступных вершин, которые удовлетворяют условиям), функция "откатывается" назад (это называется "бэктрекинг"), возвращаясь на предыдущую вершину и пытаясь выбрать другую возможность для продолжения пути. Этот процесс повторяется до тех пор, пока не будут исследованы все возможные варианты путей.

```
void hamiltonianCycleUtil(int** G, vector<int>& path, int pos, int size,
vector<vector<int>>& allPaths) {
    if (pos == size) {
        if (G[path[pos - 1]][path[0]] > 0) {
            allPaths.push_back(path);
        }
        return;
    }

    for (int v = 1; v < size; v++) {
        if (isSafe(v, G, path, pos)) {
            path[pos] = v;
            hamiltonianCycleUtil(G, path, pos + 1, size, allPaths);
            path[pos] = -1; }
    }
}
```

Каждый найденный гамильтонов цикл затем анализируется с помощью функции `calculateCycleWeight`, которая вычисляет суммарный вес цикла. Для этого она поочередно добавляет веса всех ребер между соседними вершинами в пути и суммирует их.

```
int calculateCycleWeight(int** G, const vector<int>& path) {
    int weight = 0;
    for (int i = 0; i < path.size() - 1; i++) {
        weight += G[path[i]][path[i + 1]];
    }
    weight += G[path[path.size() - 1]][path[0]];
    return weight;
}
```

После нахождения всех циклов и вычисления их весов программа выводит результаты. Функция `findAllHamiltonianCycles` организует процесс поиска циклов, начиная с указанной вершины, и выводит информацию о каждом найденном цикле и его весе.

```
void findAllHamiltonianCycles(int** G, int size, int start) {
    vector<int> path(size, -1);
    path[0] = start;

    vector<vector<int>> allPaths;
    hamiltonianCycleUtil(G, path, 1, size, allPaths);

    if (allPaths.empty()) {
        cout << "Гамильтонов циклов не существует." << endl;
    }
    else {
        cout << "Найдено Гамильтонов циклов: " << allPaths.size() << endl;

        int minWeight = INT_MAX;
        vector<int> shortestCycle;

        for (size_t i = 0; i < allPaths.size(); i++) {
            vector<int>& cycle = allPaths[i];
            for (size_t j = 0; j < cycle.size(); j++) {
                cout << cycle[j] + 1 << " ";
            }
            cout << cycle[0] + 1 << endl;
            int weight = calculateCycleWeight(G, cycle);
            cout << "Вес цикла: " << weight << endl;

            if (weight < minWeight) {
                minWeight = weight;
                shortestCycle = cycle;
            }
        }

        cout << "\nСамый короткий Гамильтонов цикл (с минимальным весом " <<
minWeight << "): ";
        for (size_t k = 0; k < shortestCycle.size(); ++k) {
            cout << shortestCycle[k] + 1 << " ";
        }
        cout << shortestCycle[0] + 1 << endl;
    }
}
```

```
}  
}
```

Для визуализации графа используется несколько функций: `getEdgeColor`, `drawArrow`, `createNodes`, `drawMenu`, `runGraphicsWindow`.

Функция `getEdgeColor` определяет цвет рёбер графа в зависимости от их веса. Она использует конструкцию `switch`, чтобы выбрать соответствующий цвет для каждого веса от 1 до 7. Например, для веса 1 возвращается красный цвет, для 2 — оранжевый, и так далее. Если вес ребра выходит за пределы указанного диапазона, возвращается белый цвет по умолчанию. Эта функция помогает визуально различать рёбра с разными весами, улучшая восприятие графа.

Функция `drawArrow` рисует стрелку между двумя вершинами графа. Она получает координаты начала и конца стрелки, цвет и смещения для корректировки позиции. Сначала вычисляется направление и длина стрелки, а затем рисуется линия между двумя точками. Для того чтобы создать вид стрелки, используется объект `ConvexShape`, представляющий собой треугольник, который размещается в нужном месте и указывает на конец стрелки. Угол и положение стрелки вычисляются в зависимости от координат двух вершин, что позволяет точно отобразить связь между ними.

Функция `createNodes` отвечает за создание и расположение вершин графа. Вершины отображаются как маленькие круги, которые равномерно размещаются по окружности вокруг заданного центра. Для каждой вершины создаётся текстовая метка, которая отображает её номер. Эти круги и метки сохраняются в два вектора, `smallCircles` и `labels`, которые затем используются для отрисовки на экране. Такая организация позволяет легко управлять расположением и визуализацией вершин.

Функция `drawMenu` рисует меню с информацией о цветах рёбер и их весах. В меню для каждого веса от 1 до 7 создаётся маленький цветной круг, который визуально указывает на цвет рёбер с данным весом. Рядом с каждым цветным кругом отображается текстовая метка, которая указывает на вес

ребра. Это меню позволяет пользователю понимать, что означают разные цвета рёбер, и помогает интерпретировать граф в целом.

Основная функция программы, `runGraphicsWindow`, инициализирует окно для отрисовки графа и шрифт для текстовых меток. После этого она вызывает функцию для создания вершин графа. В основном цикле отрисовки окна происходит следующее: для каждого ребра, если его вес больше нуля, рисуется стрелка между соответствующими вершинами, цвет которой определяется функцией `getEdgeColor`. Вершины и их метки также отрисовываются с использованием ранее созданных объектов. В дополнение, отображается меню, в котором объясняется значение каждого цвета рёбер. Этот цикл продолжается до тех пор, пока пользователь не закроет окно. Весь процесс обеспечивает наглядную и удобную визуализацию графа, позволяя пользователю понимать его структуру и связи.

Тестирование

Среда разработки Microsoft Visual Studio 2022 предоставляет все средства, необходимые при разработке и отладке многомодульной программы.

Тестирование проводилось в рабочем порядке, в процессе разработки, после завершения написания программы. В ходе тестирования было выявлено и исправлено множество проблем, связанных с вводом данных, изменением дизайна выводимых данных, алгоритмом программы, взаимодействием функций.

Тесты:

1. Запуск программы

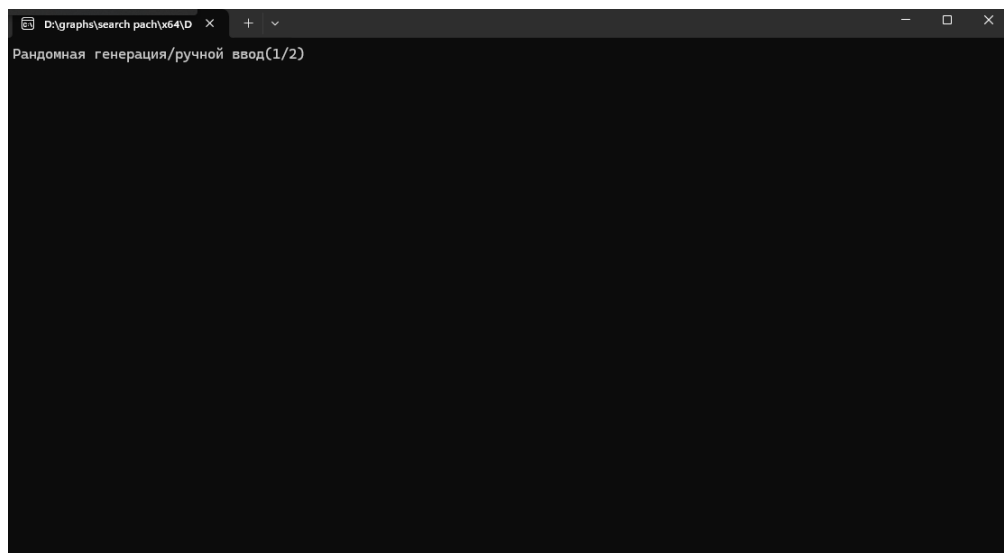
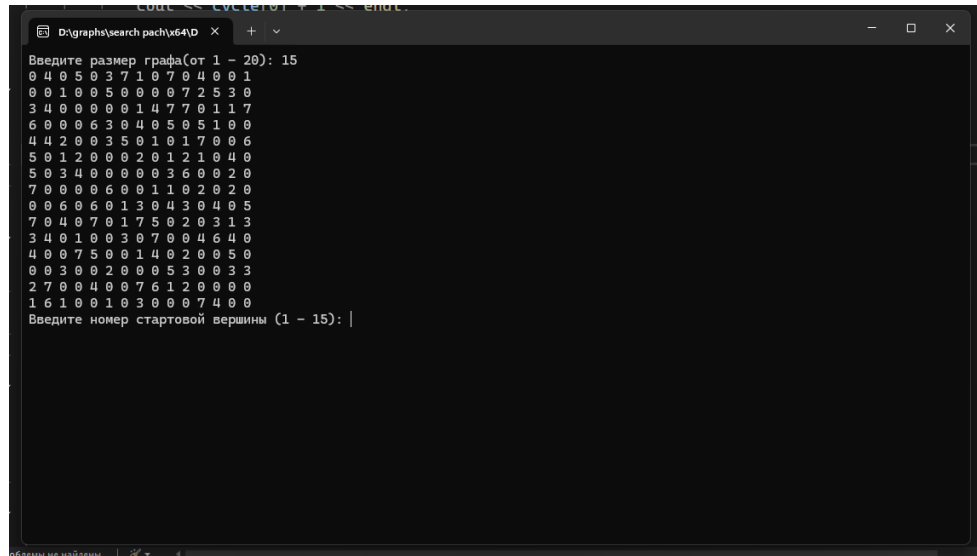


Рисунок 1 – Запуск окна, вывод сообщения о выборе размера графа

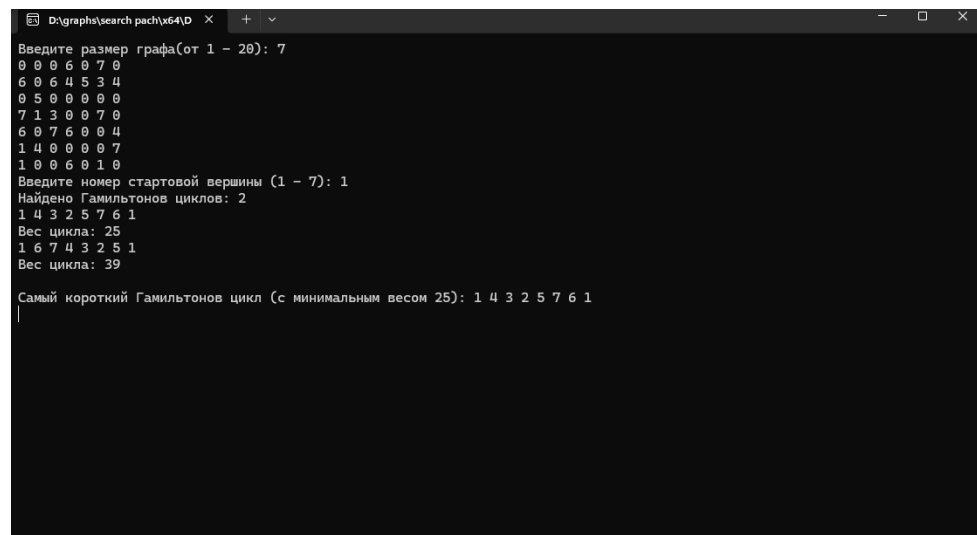
2. Выбор размера графа



```
Введите размер графа(от 1 - 20): 15
0 4 0 5 0 3 7 1 0 7 0 4 0 0 1
0 0 1 0 0 5 0 0 0 0 7 2 5 3 0
3 4 0 0 0 0 0 1 4 7 7 0 1 1 7
6 0 0 0 6 3 0 4 0 5 0 5 1 0 0
4 4 2 0 0 3 5 0 1 0 1 7 0 0 6
5 0 1 2 0 0 0 2 0 1 2 1 0 4 0
5 0 3 4 0 0 0 0 0 3 6 0 0 2 0
7 0 0 0 6 0 0 1 1 0 2 0 2 0
0 0 6 0 6 0 1 3 0 4 3 0 4 0 5
7 0 4 0 7 0 1 7 5 0 2 0 3 1 3
3 4 0 1 0 0 3 0 7 0 0 4 6 4 0
4 0 0 7 5 0 0 1 4 0 2 0 0 5 0
0 0 3 0 0 2 0 0 0 5 3 0 0 3 3
2 7 0 0 4 0 0 7 6 1 2 0 0 0 0
1 6 1 0 0 1 0 3 0 0 0 7 4 0 0
Введите номер стартовой вершины (1 - 15): |
```

Рисунок 2 – Вывод графа и сообщения о выборе стартовой вершины

3. Выбор стартовой вершины



```
Введите размер графа(от 1 - 20): 7
0 0 0 6 0 7 0
6 0 6 4 5 3 4
0 5 0 0 0 0 0
7 1 3 0 0 7 0
6 0 7 6 0 0 4
1 4 0 0 0 0 7
1 0 0 6 0 1 0
Введите номер стартовой вершины (1 - 7): 1
Найдено Гамильтонов циклов: 2
1 4 3 2 5 7 6 1
Вес цикла: 25
1 6 7 4 3 2 5 1
Вес цикла: 39

Самый короткий Гамильтонов цикл (с минимальным весом 25): 1 4 3 2 5 7 6 1
|
```

Рисунок 3 – Вывод количества циклов, циклов и короткого цикла

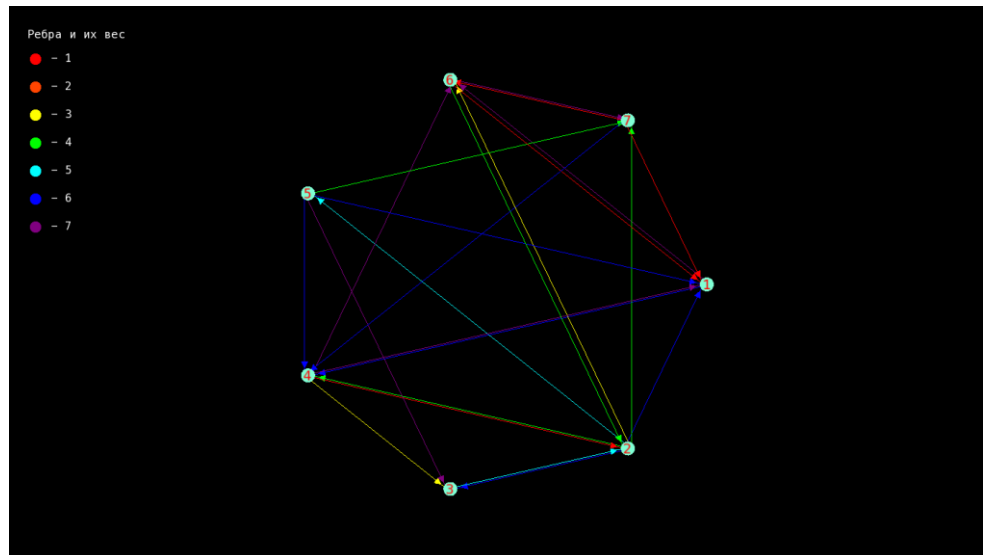


Рисунок 4 – Окно с отображением графа

```

Консоль отладки Microsoft V  x  +  v
Введите размер графа(от 1 - 20): 5
0 0 0 0 0
0 0 7 3 0
2 0 0 2 0
3 4 2 0 0
6 4 0 0 0
Введите номер стартовой вершины (1 - 5): 5
Гамильтонов циклов не существует.

D:\graphs\search rach\x64\Debug\search rach.exe (процесс 14390) завершил работу с кодом 0 (0x0).
Нажмите любую клавишу, чтобы закрыть это окно:

```

Рисунок 5 – Отображение сообщения отсутствия циклов

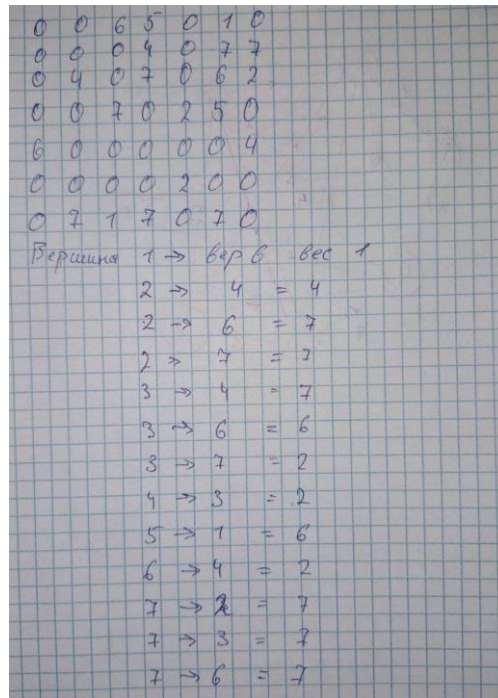
Таблица 1 - Описание поведения программы при тестировании

Описание теста	Ожидаемый результат	Полученный результат
Запуск программы	Запуск окна, вывод сообщения о выборе размера графа	Верно
Выбор размера графа	Вывод графа и сообщения о выборе стартовой вершины	Верно
Выбор стартовой вершины	Вывод количества циклов, циклы, самый короткий цикл или отсутствие циклов Открытие окна с отображением графа	Верно

Ручной расчёт задачи

Проведем проверку программы посредством ручных вычислений на примере графа размером 7 на 7 исходную матрицу смежности сгенерируем с помощью фиксированного значения(`srand(42);`).

Найдем ребра графа по матрице смежности.



Handwritten adjacency matrix and edge list for a 7x7 graph. The matrix is shown as a grid of numbers. Below it, the edges are listed as vertex-to-vertex connections with their weights.

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

Вершина 1 → вер 6 вес 1
2 → 4 = 4
2 → 6 = 7
2 → 7 = 7
3 → 4 = 7
3 → 6 = 6
3 → 7 = 2
4 → 3 = 2
5 → 1 = 6
6 → 4 = 2
7 → 1 = 7
7 → 3 = 7
7 → 6 = 7

Рисунок 6 – Ручная проверка ребер

Нарисуем визуальную модель графа и отобразим все возможные Гамильтоновы циклы и их вес.

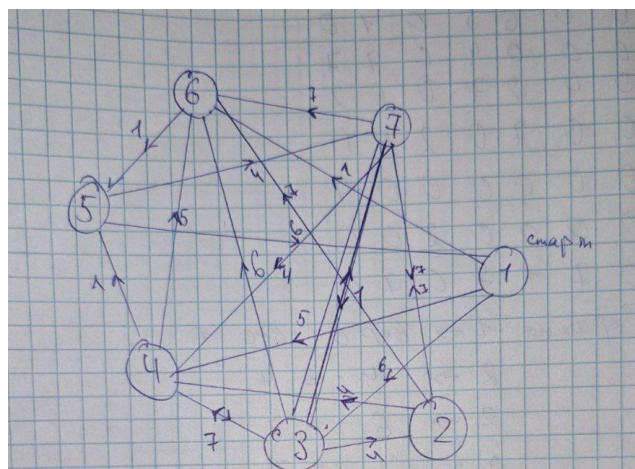


Рисунок 7 – Отрисовка графа вручную

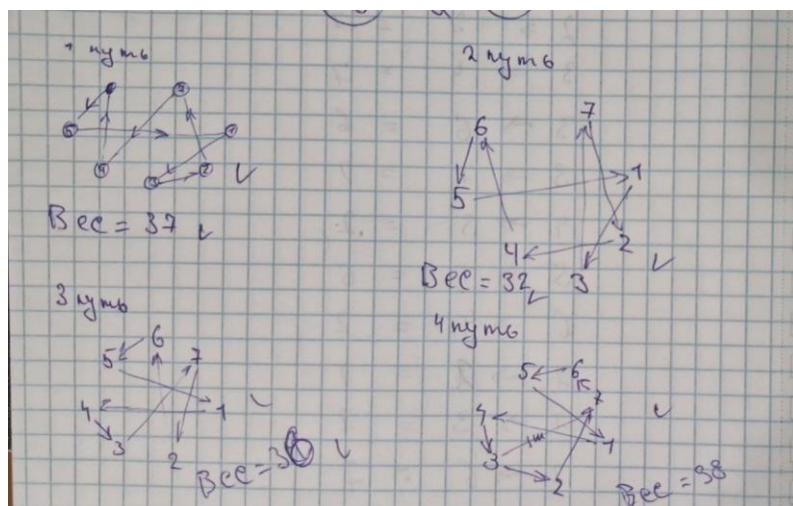


Рисунок 8 – Проверка циклов

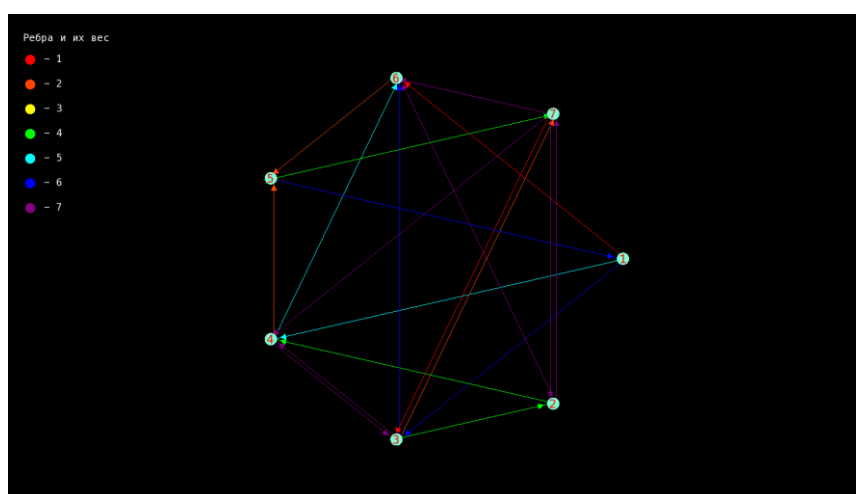


Рисунок 9 – Отрисовка графа программой

```

Элемент графа[7][4]: 7
Элемент графа[7][5]: 0
Элемент графа[7][6]: 7
Элемент графа[7][7]: 0
0 0 6 5 0 1 0
0 0 0 4 0 7 7
0 4 0 7 0 6 2
0 0 7 0 2 5 0
6 0 0 0 0 0 4
0 0 0 0 2 0 0
0 7 1 7 0 7 0
Введите номер стартовой вершины (1 - 7): 1
Найдено Гамильтонов циклов: 4
1 3 2 7 4 6 5 1
Вес цикла: 37
1 3 7 2 4 6 5 1
Вес цикла: 32
1 4 3 2 7 6 5 1
Вес цикла: 38
1 4 3 7 2 6 5 1
Вес цикла: 36
Самый короткий Гамильтонов цикл (с минимальным весом 32): 1 3 7 2 4 6 5 1

```

Рисунок 10 – Отрисовка графа программой

Ручная проверка с результатами программы совпали, следовательно программа работает верно.

Заключение

В ходе работы было реализовано создание программного продукта, позволяющего визуализировать граф и искать гамильтоновы циклы в ориентированных графах с учетом веса рёбер. Одной из главных особенностей данного программного продукта является интеграция алгоритма поиска гамильтоновых циклов с минимальным весом, а также возможность визуального представления графа с использованием библиотеки SFML.

Программный продукт включает консольный интерфейс для взаимодействия с пользователем и графический интерфейс для наглядного отображения структуры графа. Ключевая особенность визуализации — цветовая индикация рёбер в зависимости от их веса, что повышает удобство восприятия.

Поставленная задача была полностью реализована. Недостатком программы является отсутствие современного пользовательского интерфейса с развитой функциональностью (например, оконных меню, кнопок или ввода данных в графическом режиме). Тем не менее, данный подход упрощает работу с программой и минимизирует её зависимость от сторонних библиотек.

Список литературы

1. Новиков Ф.А. «Дискретная математика для программистов» 2-е изд. – СПб.: Питер, 2000.
2. Кристофидес Н. «Теория графов. Алгоритмический подход» - Мир, 1978
3. Герберт Шилдт «Полный справочник по C++» - Вильямс, 2006

Приложение А.

Листинг программы

main.cpp

```
#include "GraphOperations.h"
#include "Rendering.h"
#include <SFML/Graphics.hpp>
#include <locale>
#include <ctime>
#include <iostream>

using namespace std;

int main() {
    srand(time(NULL));
    setlocale(LC_ALL, "Russian");

    int flag = 0;
    int nG = 0;
    int** G = 0;

    cout << "Рандомная генерация/ручной ввод(1/2): ";
    cin >> flag;

    if (flag < 1 || flag > 2) {
        cout << "Неверный выбор!" << endl;
        exit(-1);
    }

    if (flag == 1) {
        nG = handleUserInputSize();
        G = initGraph(nG);
    }
    else {
        nG = handleUserInputSize();
        G = new int* [nG];
        cout << "Введите граф размером " << nG << ", числа от 0 до 7: " << endl;
        for (int i = 0; i < nG; i++) {
            G[i] = new int[nG];
            for (int j = 0; j < nG; j++) {
                cout << "Элемент графа[" << i + 1 << "][" << j + 1 << "]: ";
                cin >> G[i][j];
                if (G[i][j] < 0 || G[i][j] > 7) {
                    cout << "Неверный выбор!" << endl;
                    exit(-1);
                }
            }
        }
        printG(G, nG);
    }

    int start = handleUserInputStart(nG);
    findAllHamiltonianCycles(G, nG, start);
    runGraphicsWindow(G, nG);

    return 0;
}
```

GraphOperations.h

```
#ifndef GRAPH_OPERATIONS_H
#define GRAPH_OPERATIONS_H
```

```

#include <vector>

int** createG(int size);
void printG(int** G, int size);
bool isSafe(int v, int** G, std::vector<int>& path, int pos);
void hamiltonianCycleUtil(int** G, std::vector<int>& path, int pos, int size,
std::vector<std::vector<int>>& allPaths);
int calculateCycleWeight(int** G, const std::vector<int>& path);
void findAllHamiltonianCycles(int** G, int size, int start);
int** initGraph(int size);
int handleUserInputStart(int size);
int handleUserInputSize();

#endif

```

Rendering.h

```

#ifndef RENDERING_H
#define RENDERING_H

#include <SFML/Graphics.hpp>
#include <vector>

sf::Color getEdgeColor(int weight);
void drawArrow(sf::RenderWindow& window, float x1, float y1, float x2, float y2,
sf::Color color, float offsetX = 0, float offsetY = 0);
void createNodes(std::vector<sf::CircleShape>& smallCircles,
std::vector<sf::Text>& labels, const sf::Vector2f& center, int count, float
radius, float smallCircleRadius, const sf::Font& font);
void drawMenu(sf::RenderWindow& window, const sf::Font& font);
void runGraphicsWindow(int** G, int nG);

#endif

```

GraphOperations.cpp

```

#include "GraphOperations.h"
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <climits>

using namespace std;

int** createG(int size) {
    int** G = new int* [size];
    for (int i = 0; i < size; i++) {
        G[i] = new int[size];
        for (int j = 0; j < size; j++) {
            G[i][j] = (i != j && rand() % 2 == 0) ? rand() % 7 + 1 : 0;
        }
    }
    return G;
}

void printG(int** G, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            cout << G[i][j] << " ";
        }
        cout << endl;
    }
}

```



```

bool isSafe(int v, int** G, vector<int>& path, int pos) {
    if (G[path[pos - 1]][v] == 0) return false;
    for (int i = 0; i < pos; i++) {
        if (path[i] == v) return false;
    }
    return true;
}

void hamiltonianCycleUtil(int** G, vector<int>& path, int pos, int size,
vector<vector<int>>& allPaths) {
    if (pos == size) {
        if (G[path[pos - 1]][path[0]] > 0) {
            allPaths.push_back(path);
        }
        return;
    }

    for (int v = 1; v < size; v++) {
        if (isSafe(v, G, path, pos)) {
            path[pos] = v;
            hamiltonianCycleUtil(G, path, pos + 1, size, allPaths);
            path[pos] = -1;
        }
    }
}

int calculateCycleWeight(int** G, const vector<int>& path) {
    int weight = 0;
    for (int i = 0; i < path.size() - 1; i++) {
        weight += G[path[i]][path[i + 1]];
    }
    weight += G[path[path.size() - 1]][path[0]];
    return weight;
}

void findAllHamiltonianCycles(int** G, int size, int start) {
    vector<int> path(size, -1);
    path[0] = start;

    vector<vector<int>> allPaths;
    hamiltonianCycleUtil(G, path, 1, size, allPaths);

    if (allPaths.empty()) {
        cout << "Гамильтонов циклов не существует." << endl;
    }
    else {
        cout << "Найдено Гамильтонов циклов: " << allPaths.size() << endl;

        int minWeight = INT_MAX;
        vector<int> shortestCycle;

        for (size_t i = 0; i < allPaths.size(); ++i) {
            vector<int>& cycle = allPaths[i];
            for (size_t j = 0; j < cycle.size(); ++j) {
                cout << cycle[j] + 1 << " ";
            }
            cout << cycle[0] + 1 << endl;
            int weight = calculateCycleWeight(G, cycle);
            cout << "Вес цикла: " << weight << endl;

            if (weight < minWeight) {
                minWeight = weight;
                shortestCycle = cycle;
            }
        }
    }
}

```

```

        cout << "\nСамый короткий Гамильтонов цикл (с минимальным весом " <<
minWeight << "): ";
        for (size_t k = 0; k < shortestCycle.size(); ++k) {
            cout << shortestCycle[k] + 1 << " ";
        }
        cout << shortestCycle[0] + 1 << endl;
    }
}

int** initGraph(int size) {
    int** G = createG(size);
    printG(G, size);
    return G;
}

int handleUserInputStart(int size) {
    int start;
    cout << "Введите номер стартовой вершины (1 - " << size << "): ";
    cin >> start;
    start -= 1;

    if (start < 0 || start >= size) {
        cout << "Неверный номер вершины!" << endl;
        exit(-1);
    }
    return start;
}

int handleUserInputSize() {
    int nG;
    cout << "Введите размер графа (от 1 - 20): ";
    cin >> nG;

    if (nG < 1 || nG > 20) {
        cout << "Неверный размер графа!" << endl;
        exit(-1);
    }
    return nG;
}

```

Rendering.cpp

```

#include "Rendering.h"
#include "const.h"
#include <cmath>
#include <string>

using namespace sf;
using namespace std;

Color getEdgeColor(int weight) {
    switch (weight) {
        case 1: return Color(255, 0, 0);
        case 2: return Color(255, 69, 0);
        case 3: return Color(255, 255, 0);
        case 4: return Color(0, 255, 0);
        case 5: return Color(0, 255, 255);
        case 6: return Color(0, 0, 255);
        case 7: return Color(128, 0, 128);
        default: return Color::White;
    }
}

void drawArrow(RenderWindow& window, float x1, float y1, float x2, float y2, Color
color, float offsetX, float offsetY) {
    x1 += offsetX;

```

```

y1 += offsetY;
x2 += offsetX;
y2 += offsetY;

float dx = x2 - x1;
float dy = y2 - y1;
float length = sqrt(dx * dx + dy * dy);
float angle = atan2(dy, dx);

Vertex edge[] = {
    Vertex(Vector2f(x1, y1), color),
    Vertex(Vector2f(x2, y2), color)
};
window.draw(edge, 2, Lines);

float arrowSize = 10.0f;
ConvexShape arrow;
arrow.setPointCount(3);
arrow.setPoint(0, Vector2f(0, 0));
arrow.setPoint(1, Vector2f(-arrowSize, -arrowSize / 2));
arrow.setPoint(2, Vector2f(-arrowSize, arrowSize / 2));
arrow.setFill-color(color);
arrow.setPosition(x2 - dx / length * 10, y2 - dy / length * 10);
arrow.setRotation(angle * 180.0f / M_PI);

window.draw(arrow);
}

void createNodes(std::vector<CircleShape>& smallCircles, std::vector<Text>&
labels, const Vector2f& center, int count, float radius, float smallCircleRadius,
const Font& font) {
    for (int i = 0; i < count; ++i) {
        CircleShape smallCircle(smallCircleRadius);
        smallCircle.setFill-color(Color(127, 255, 212));
        float angle = i * (2 * M_PI / count);
        float x = center.x + radius * cos(angle);
        float y = center.y + radius * sin(angle);
        smallCircle.setOrigin(smallCircleRadius, smallCircleRadius);
        smallCircle.setPosition(x, y);
        smallCircles.push_back(smallCircle);

        Text label;
        label.setFont(font);
        label.setString(to_string(i + 1));
        label.setCharacterSize(20);
        label.setFill-color(Color::Red);
        FloatRect textBounds = label.getLocalBounds();
        label.setOrigin(textBounds.width / 2.0f, textBounds.height / 2.0f);
        label.setPosition(x - 2, y - 6);
        labels.push_back(label);
    }
}

void drawMenu(RenderWindow& window, const Font& font) {
    float menuX = 32;
    float menuY = 70;
    float spacing = 40;

    Text colorInf;
    colorInf.setFont(font);
    colorInf.setString(L"Ребра и их вес");
    colorInf.setCharacterSize(16);
    colorInf.setPosition(28, 32);
    colorInf.setFill-color(Color::White);

    window.draw(colorInf);
}

```

```

for (int weight = 1; weight <= 7; ++weight) {
    CircleShape colorCircle(8);
    colorCircle.setFillColor(getEdgeColor(weight));
    colorCircle.setPosition(menuX, menuY + (weight - 1) * spacing);

    Text dashText;
    dashText.setFont(font);
    dashText.setString(" - ");
    dashText.setCharacterSize(16);
    dashText.setFillColor(Color::White);
    dashText.setPosition(menuX + 20, menuY + (weight - 1) * spacing - 4);

    Text weightText;
    weightText.setFont(font);
    weightText.setString(to_string(weight));
    weightText.setCharacterSize(16);
    weightText.setFillColor(Color::White);
    weightText.setPosition(menuX + 50, menuY + (weight - 1) * spacing - 4);

    window.draw(colorCircle);
    window.draw(weightText);
    window.draw(dashText);
}

}

void runGraphicsWindow(int** G, int nG) {
    RenderWindow window(VideoMode(1400, 800), "Graf");
    Font font;
    if (!font.loadFromFile("menlo.ttf")) exit(-1);

    const float mainRadius = 300.0f;
    const float smallCircleRadius = 10.0f;
    Vector2f center(window.getSize().x / 2.0f, window.getSize().y / 2.0f);

    vector<CircleShape> smallCircles;
    vector<Text> labels;
    createNodes(smallCircles, labels, center, nG, mainRadius, smallCircleRadius,
font);

    while (window.isOpen()) {
        Event event;
        while (window.pollEvent(event)) {
            if (event.type == Event::Closed) window.close();
        }

        window.clear();

        for (int i = 0; i < nG; ++i) {
            for (int j = 0; j < nG; ++j) {
                if (G[i][j] > 0) {
                    float angle1 = i * (2 * M_PI / nG);
                    float x1 = center.x + mainRadius * cos(angle1);
                    float y1 = center.y + mainRadius * sin(angle1);
                    float angle2 = j * (2 * M_PI / nG);
                    float x2 = center.x + mainRadius * cos(angle2);
                    float y2 = center.y + mainRadius * sin(angle2);
                    float offsetX = (i < j) ? 5.0f : (i > j ? -5.0f : 0);
                    drawArrow(window, x1, y1, x2, y2, getEdgeColor(G[i][j]),
offsetX, 0);
                }
            }
        }

        for (size_t i = 0; i < smallCircles.size(); ++i) {
            window.draw(smallCircles[i]);
            window.draw(labels[i]);
        }
    }
}

```

```
        drawMenu(window, font);  
        window.display();  
    }  
}
```

const.h

```
#ifndef CONST_H  
#define CONST_H  
  
#ifndef M_PI  
#define M_PI 3.14159265358979323846  
#endif  
  
#endif
```