|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |   Институт Информационных технологий | |
|  | |
| Кафедра Математического обеспечения и стандартизации информационных технологий | |
|  |  |
|  |  |

|  |  |
| --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 7.2** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  Тема. Графы: создание, алгоритмы обхода, важные задачи теории графов | |
|  | |
| Выполнил студент группы ИНБО-12-23 | Албахтин И.В. |
| Принял ассистент | Муравьёва Е.А. |

Москва 2024

СОДЕРЖАНИЕ

[Цель работы 3](#_Toc182941257)

[ЗАДАНИЕ 1 3](#_Toc182941258)

[1.1 Формулировка задачи 3](#_Toc182941259)

[1.2 Решение задачи 3](#_Toc182941260)

[ЗАДАНИЕ 2 8](#_Toc182941261)

[2.1 Формулировка задачи 8](#_Toc182941262)

[2.2 Математическая модель решения (описание алгоритма). 9](#_Toc182941263)

[2.3 Код программы 9](#_Toc182941264)

[2.4 Результаты тестирования 9](#_Toc182941265)

[ВЫВОДЫ 10](#_Toc182941266)

[СПИСОК ЛИТЕРАТУРЫ 11](#_Toc182941267)

# Цель работы

Получение практических навыков по выполнению операций над

структурой данных граф.

# ЗАДАНИЕ 1

## 1.1 Формулировка задачи

Ответьте на вопросы и выполните упражнения:

## Решение задачи

1. Дайте определения понятиям: ориентированный граф, неориентированный граф, взвешенный граф, связный граф, центр графа, диаметр графа, матрица смежности.

**Ориентированный граф** – это граф, у которого каждое ребро имеет направление (от одной вершины к другой).

**Неориентированный граф** – это граф, в котором каждое ребро соединяет две вершины без указания направления.

**Взвешенный граф** – это граф, у которого каждому ребру присвоен вес (числовое значение).

**Связный граф** – это граф, в котором существует путь между любой парой вершин.

**Центр графа** – это вершина (или множество вершин), минимизирующая максимальное расстояние до всех остальных вершин графа.

**Диаметр графа** – это максимальное расстояние (по числу ребер или сумме весов) между двумя вершинами графа.

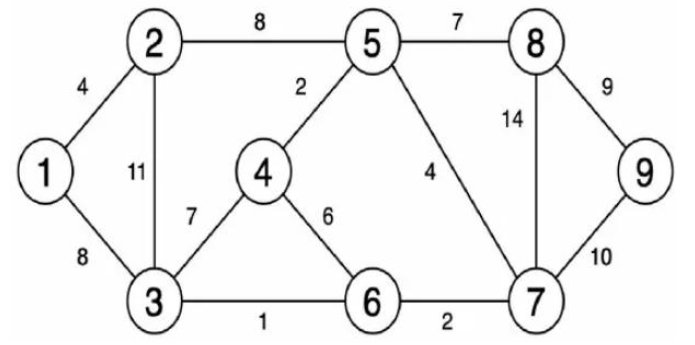
**Матрица смежности** – это квадратная матрица, в которой строки и столбцы соответствуют вершинам графа, а элементы показывают наличие и вес ребра между вершинами (0, если ребра нет).

1. Что такое остовное дерево графа?

**Остовное дерево графа** – это подграф, включающий все вершины исходного графа и минимальное число ребер, чтобы граф оставался связным. При этом в остовном дереве не должно быть циклов.

1. Какое количество ребер в остовном графе?

Если граф имеет n вершин, то в остовном дереве всегда n−1 ребро. Например, если граф содержит 9 вершин, в остовном дереве будет 9−1=8 ребер.

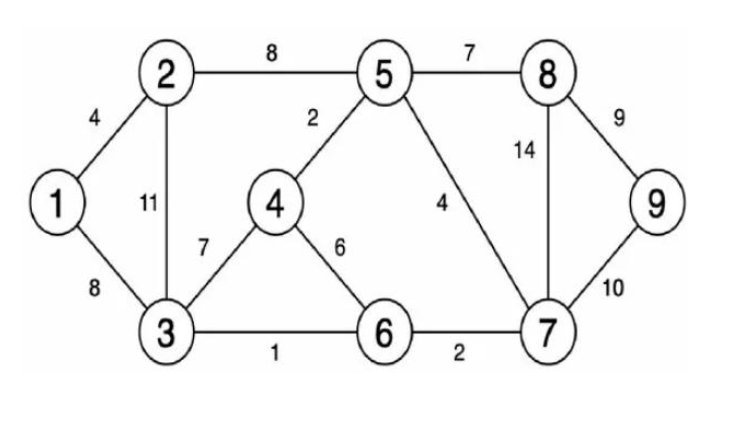


1. Что такое кратчайший путь в графе?

**Кратчайший путь в графе** — это путь между двумя вершинами, который имеет минимальную длину. Длина пути может определяться количеством ребер (в невзвешенных графах) или суммой весов ребер (в взвешенных графах).

6. Найдите кратчайший путь от вершины 1 до вершины 9, используя

алгоритм Дейкстры.

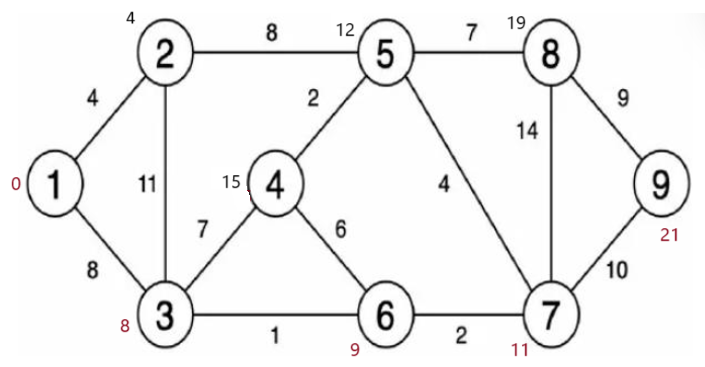


Алгоритм Дейкстры:

**Алгоритм Дейкстры** используется для поиска кратчайших путей от одной заданной вершины (источника) до всех остальных вершин в графе с неотрицательными весами ребер.

Шаги алгоритма:

1. Назначить начальной вершине метку 0, а всем остальным вершинам — метку бесконечность.
2. Пометить начальную вершину как посещенную.
3. Для всех смежных с текущей вершиной вершин обновить метки: Новая метка = минимальное значение между текущей меткой и метка и текущей вершины + веса ребра.
4. Выбрать непосещенную вершину с наименьшей меткой и сделать её текущей.
5. Повторять процесс, пока все вершины не будут посещены или пока не найдена цель.



Итоговый путь:

1→3→6→7→91.

**Длина пути:** 8+1+2+10=21

7. В чем отличие алгоритма Дейкстры от алгоритма Флойда-Уоршала.

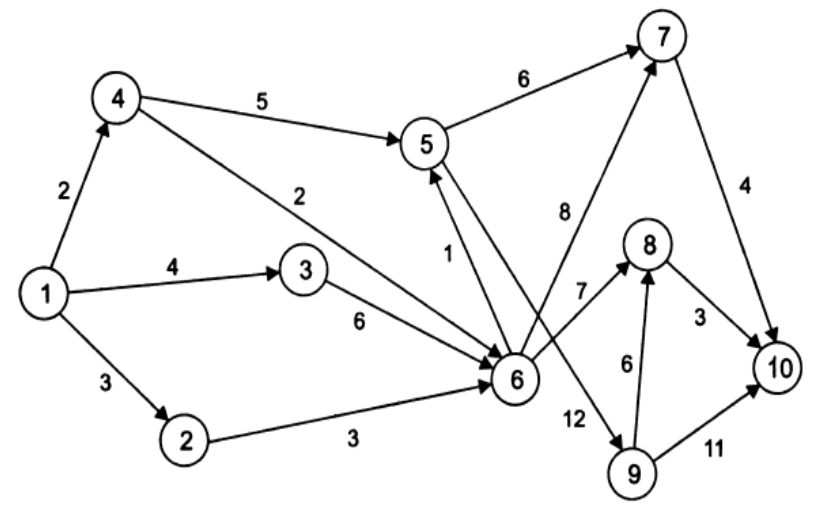
Какова вычислительная сложность каждого алгоритма по времени и

памяти.

|  |  |  |
| --- | --- | --- |
| **Характеристика** | **Алгоритм Дейкстры** | **Алгоритм Флойда-Уоршала** |
| **Задача** | Поиск кратчайшего пути от одной вершины ко всем другим | Поиск кратчайших путей между всеми парами вершин |
| **Тип графа** | Граф должен иметь положительные веса рёбер | Работает с графами с положительными и отрицательными весами (но не с отрицательными циклами) |
| **Вычислительная сложность** | O(V2) при реализации с матрицей смежности; O(V⋅logV+E) при использовании очереди с приоритетом | O(V3) |
| **Потребление памяти** | O(V+E) для графа | O(V2) для матрицы кратчайших путей |
| **Подход** | Жадный алгоритм | Динамическое программирование |

8. Обойдите граф, используя метод поиска а) в ширину; б) в глубину.

Стартовая вершина – 1.



**а) Поиск в ширину**

**Алгоритм:**

1. Создать очередь, добавить в неё стартовую вершину (1).
2. Установить для всех вершин статус *не посещена*.
3. Пока очередь не пуста:
   * Извлечь вершину из очереди.
   * Отметить вершину как *посещённую*.
   * Добавить в очередь всех её соседей, которые ещё не посещены.

1 2 3 4 6 5 7 8 9 10

**б) Поиск в глубину (DFS)**

**Алгоритм:**

1. Использовать стек или рекурсию.
2. Начать с вершины 1 и пометить её как *посещённую*.
3. Для каждого соседа текущей вершины, если он не посещён:
   * Перейти к соседу (рекурсивно или через стек).
   * Отметить его как *посещённого*.

1 2 6 5 7 10 9 8 3 4

# ЗАДАНИЕ 2

## 2.1 Формулировка задачи

1. Разработать класс «Граф», обеспечивающий хранение и работу со

структурой данных «граф», в соответствии с вариантом индивидуального задания:

* Реализовать метод ввода графа с клавиатуры, наполнение графа

осуществлять с помощью метода добавления одного ребра.

* Реализовать методы, выполняющие задачи, определенные вариантом индивидуального задания.
* Разработать доступный способ (форму) вывода результирующего дерева на экран монитора.

2. Разработать программу, демонстрирующую работу всех методов класса.

3. Произвести тестирование программы на одном из графов, предложенных в таблице.

4. Составить отчет, отобразив в нем описание выполнения всех этапов

разработки, тестирования и код всей программы со скриншотами результатов тестирования.

Таблица 1. Вариант заданий

|  |  |  |
| --- | --- | --- |
| Номер варианта | Представление графа в памяти | Задачи |
| 3 | Матрица  смежности | Найти и вывести Эйлеров цикл в графе.  Составить программу реализации алгоритма Прима построения остовного дерева минимального веса. |

Таблица 2. Примеры графов для тестирования алгоритмов

|  |  |
| --- | --- |
| **№** | **Граф** |
| 3 |  |

## 2.2 Математическая модель решения (описание алгоритма).

**1. Метод Graph::Graph (конструктор)**

Алгоритм:

* **Шаг 1**: Установить флаг ориентированности графа is\_oriented в значение mark\_is\_oriented.
* **Шаг 2**: Инициализировать вектор graph\_ размером count\_vertex + 1, где каждый элемент — это множество.
* **Шаг 3**: Для каждого ребра в edges добавить связь from -> to в граф.
* **Шаг 4**: Если граф не ориентированный, добавить обратную связь to -> from.

**2. Метод Graph::GetAdjacent**

Алгоритм:

* **Шаг 1**: Вернуть множество смежных вершин для указанной вершины vertex.

**3. Метод Graph::Size**

Алгоритм:

* **Шаг 1**: Вернуть размер графа (количество вершин), вычитая 1 из размера graph\_.

**4. Метод Graph::AddEdge**

Алгоритм:

* **Шаг 1**: Добавить вершину edge.too в множество смежных вершин для вершины edge.from.
* **Шаг 2**: Если граф не ориентированный, добавить edge.from в множество смежных вершин для edge.too.

**5. Метод Graph::RemoveEdge**

Алгоритм:

* **Шаг 1**: Удалить вершину edge.too из множества смежных вершин для вершины edge.from.
* **Шаг 2**: Если граф не ориентированный, удалить edge.from из множества смежных вершин для edge.too.

**6. Функция FindEulerCycleRecursion**

Алгоритм:

* **Шаг 1**: Пока у текущей вершины from есть смежные вершины:
  + **Шаг 2**: Получить первую смежную вершину too.
  + **Шаг 3**: Удалить ребро from -> too из графа.
  + **Шаг 4**: Рекурсивно вызвать функцию для вершины too.
* **Шаг 5**: Добавить вершину from в конец цикла.

**7. Функция DFS**

Алгоритм:

* **Шаг 1**: Пометить текущую вершину from как посещённую.
* **Шаг 2**: Для каждой смежной вершины too:
  + Если вершина не посещена и не равна родительской вершине parent, рекурсивно вызвать DFS для too.

**8. Функция CheckEvenDegrees**

Алгоритм:

* **Шаг 1**: Если граф не ориентированный:
  + **Шаг 2**: Создать вектор степеней вершин.
  + **Шаг 3**: Для каждой вершины проверить, является ли её степень чётной. Если нет, вернуть false.
* **Шаг 4**: Если граф ориентированный:
  + **Шаг 5**: Создать векторы входящих и исходящих степеней для вершин.
  + **Шаг 6**: Вычислить входящие и исходящие степени для каждой вершины.
  + **Шаг 7**: Проверить, совпадают ли входящие и исходящие степени для каждой вершины. Если нет, вернуть false.
* **Шаг 8**: Вернуть true, если все проверки пройдены.

**9. Функция FindEulerCycle**

Алгоритм:

* **Шаг 1**: Создать вектор посещённых вершин used размером graph.Size() + 1.
* **Шаг 2**: Запустить DFS из вершины 1.
* **Шаг 3**: Проверить, остались ли непосещённые вершины с рёбрами. Если такие есть, вернуть nullopt.
* **Шаг 4**: Проверить чётность степеней всех вершин. Если проверка не прошла, вернуть nullopt.
* **Шаг 5**: Инициализировать пустой вектор cycle.
* **Шаг 6**: Запустить рекурсивную функцию FindEulerCycleRecursion для вершины 1.
* **Шаг 7**: Перевернуть цикл и вернуть его.

**10. Метод Graph::Graph (конструктор)**

Алгоритм:

* **Шаг 1**: Установить флаг ориентированности графа is\_oriented в значение mark\_is\_oriented.
* **Шаг 2**: Инициализировать вектор graph\_ размером count\_vertex + 1, где каждый элемент — это вектор пар {вершина, вес}.
* **Шаг 3**: Для каждого ребра from -> too с весом weight добавить связь в граф.
* **Шаг 4**: Если граф не ориентированный, добавить обратную связь too -> from с тем же весом.

**11. Метод Graph::GetAdjacent**

Алгоритм:

* **Шаг 1**: Вернуть вектор смежных вершин (пар {вершина, вес}) для указанной вершины vertex.

**12. Метод Graph::Size**

Алгоритм:

* **Шаг 1**: Вернуть размер графа (количество вершин), равное graph\_.size().

**13. Функция FindMST**

Алгоритм:

* **Шаг 1**: Инициализировать вектор минимальных расстояний dist длиной graph.Size(), где каждая вершина имеет значение {cMaxDist, 0}. Для вершины 1 установить {0, 0}.
* **Шаг 2**: Инициализировать множество best\_vertices с вершиной 1 и её расстоянием {0, 1}.
* **Шаг 3**: Инициализировать вектор used для отслеживания посещённых вершин.
* **Шаг 4**: Пока множество best\_vertices не пусто:
  + **Шаг 5**: Извлечь вершину с минимальным расстоянием best\_v.
  + **Шаг 6**: Пометить вершину best\_v как посещённую.
  + **Шаг 7**: Для каждой смежной вершины too и веса weight:
    - Если вес weight меньше текущего минимального расстояния до too и вершина ещё не посещена:
      * Удалить старое значение вершины too из множества best\_vertices.
      * Обновить минимальное расстояние до too в векторе dist.
      * Добавить новое значение вершины too в множество best\_vertices.
* **Шаг 8**: Инициализировать стоимость MST mst\_cost равной 0 и пустой вектор рёбер MST mst.
* **Шаг 9**: Для каждой вершины от 2 до graph.Size():
  + Добавить стоимость ребра к mst\_cost.
  + Добавить ребро {dist[too].second, too, dist[too].first} в mst.
* **Шаг 10**: Вернуть пару {mst\_cost, mst}.

## 2.3 Код программы

**Листинг 1.** Код программы

#include <algorithm>

#include <iostream>

#include <optional>

#include <unordered\_set>

#include <vector>

using namespace std;

struct Edge {

  size\_t from;

  size\_t too;

};

class Graph {

 private:

  vector<unordered\_set<size\_t>> graph\_;

 public:

  bool is\_oriented;

  explicit Graph(const size\_t count\_vertex, const vector<Edge>& edges,

                 const bool mark\_is\_oriented = true)

      : is\_oriented(mark\_is\_oriented) {

    graph\_.resize(count\_vertex + 1);

    for (const auto& [from, too] : edges) {

      graph\_[from].insert(too);

      if (!is\_oriented) {

        graph\_[too].insert(from);

      }

    }

  }

  unordered\_set<size\_t> GetAdjacent(const size\_t vertex) const {

    return graph\_[vertex];

  }

  size\_t Size() const { return graph\_.size() - 1; }

  void AddEdge(const Edge& edge) {

    graph\_[edge.from].insert(edge.too);

    if (!is\_oriented) {

      graph\_[edge.too].insert(edge.from);

    }

  }

  void RemoveEdge(const Edge& edge) {

    graph\_[edge.from].erase(edge.too);

    if (!is\_oriented) {

      graph\_[edge.too].erase(edge.from);

    }

  }

};

void FindEulerCycleRecursion(const size\_t from, Graph& graph,

                             vector<size\_t>& cycle) {

  while (!graph.GetAdjacent(from).empty()) {

    size\_t too = \*graph.GetAdjacent(from).begin();

    graph.RemoveEdge({from, too});

    FindEulerCycleRecursion(too, graph, cycle);

  }

  cycle.push\_back(from);

}

void DFS(const size\_t from, const Graph& graph, vector<bool>& used,

         const int parent = -1) {

  used[from] = true;

  for (const auto& too : graph.GetAdjacent(from)) {

    if (!used[too] && too != parent) {

      DFS(too, graph, used, from);

    }

  }

}

bool CheckEvenDegrees(const Graph& graph) {

  if (!graph.is\_oriented) {

    vector<size\_t> degrees(graph.Size() + 1);

    for (size\_t from = 1; from < graph.Size() + 1; ++from) {

      if (graph.GetAdjacent(from).size() % 2 == 1) {

        return false;

      }

    }

  } else {

    vector<size\_t> in\_degree(graph.Size() + 1);

    vector<size\_t> out\_degree(graph.Size() + 1);

    for (size\_t from = 1; from < graph.Size() + 1; ++from) {

      out\_degree[from] = graph.GetAdjacent(from).size();

      for (const auto& too : graph.GetAdjacent(from)) {

        ++in\_degree[too];

      }

    }

    for (size\_t vertex = 1; vertex < graph.Size() + 1; ++vertex) {

      if (in\_degree[vertex] != out\_degree[vertex]) {

        return false;

      }

    }

  }

  return true;

}

optional<vector<size\_t>> FindEulerCycle(Graph& graph) {

  vector<bool> used(graph.Size() + 1);

  DFS(1, graph, used);

  for (size\_t vertex = 1; vertex < graph.Size() + 1; ++vertex) {

    if (!used[vertex] && graph.GetAdjacent(vertex).size() > 0) {

      return nullopt;

    }

  }

  if (!CheckEvenDegrees(graph)) {

    return nullopt;

  }

  vector<size\_t> cycle;

  FindEulerCycleRecursion(1, graph, cycle);

  reverse(cycle.begin(), cycle.end());

  return cycle;

}

int main() {

  size\_t count\_vertices;

  size\_t count\_edges;

  cout << "введите количество вершин: ";

  cin >> count\_vertices;

  cout << "введите количество ребер: ";

  cin >> count\_edges;

  Graph graph(count\_vertices, {}, false);  // true если граф ориентированный

  for (size\_t i = 0; i < count\_edges; ++i) {

    Edge edge;

    cin >> edge.from >> edge.too;

    graph.AddEdge(edge);

  }

  auto cycle = FindEulerCycle(graph);

  if (!cycle) {

    cout << "No Euler cycle\n";

    return 0;

  }

  for (const auto& vertex : cycle.value()) {

    cout << vertex << ' ';

  }

  return 0;

}

**Листинг 2.** Код программы

#include <algorithm>

#include <climits>

#include <iostream>

#include <numeric>

#include <set>

#include <vector>

using namespace std;

const long long cMaxDist = LLONG\_MAX;

struct Edge {

  size\_t from;

  size\_t too;

  long long weight;

};

class Graph {

 private:

  vector<vector<pair<size\_t, long long>>> graph\_;

 public:

  bool is\_oriented;

  explicit Graph(const size\_t count\_vertex, const vector<Edge>& edges,

                 const bool mark\_is\_oriented = true)

      : is\_oriented(mark\_is\_oriented) {

    graph\_.resize(count\_vertex + 1);

    for (auto [from, too, weight] : edges) {

      graph\_[from].push\_back({too, weight});

      if (!is\_oriented) {

        graph\_[too].push\_back({from, weight});

      }

    }

  }

  auto GetAdjacent(const size\_t vertex) const { return graph\_[vertex]; }

  size\_t Size() const { return graph\_.size(); }

};

pair<long long, vector<Edge>> FindMST(const Graph& graph) {

  vector<pair<long long, size\_t>> dist(graph.Size(),

                                       {cMaxDist, 0});  // {dist, from}

  dist[1] = {0, 0};

  set<pair<long long, size\_t>> best\_vertices = {{0, 1}};  // {dist, vertex}

  vector<bool> used(graph.Size());

  while (!best\_vertices.empty()) {

    size\_t best\_v = best\_vertices.begin()->second;

    best\_vertices.erase(best\_vertices.begin());

    used[best\_v] = true;

    for (auto [too, weight] : graph.GetAdjacent(best\_v)) {

      if (weight < dist[too].first && !used[too]) {

        best\_vertices.erase({dist[too].first, too});

        dist[too] = {weight, best\_v};

        best\_vertices.insert({dist[too].first, too});

      }

    }

  }

  long long mst\_cost = 0;

  vector<Edge> mst;

  for (size\_t too = 2; too < graph.Size(); ++too) {

    mst\_cost += dist[too].first;

    mst.push\_back({dist[too].second, too, dist[too].first});

  }

  return {mst\_cost, mst};

}

int main() {

  size\_t count\_vertices;

  size\_t count\_edges;

  cout << "введите количество вершин: ";

  cin >> count\_vertices;

  cout << "введите количество ребер: ";

  cin >> count\_edges;

  vector<Edge> edges(count\_edges);

  for (auto& [from, too, weight] : edges) {

    cin >> from >> too >> weight;

  }

  cout << endl;

  Graph graph(count\_vertices, edges, false);

  auto [mst\_cost, mst] = FindMST(graph);

  cout << mst\_cost << '\n';

  for (const auto& [from, too, weight] : mst) {

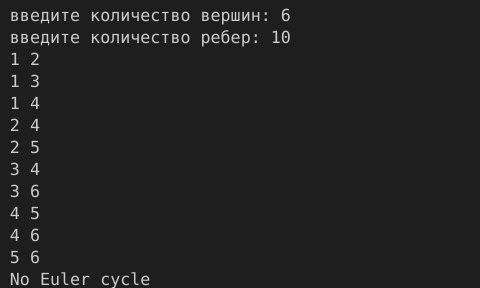
    cout << from << ' ' << too << ' ' << weight << '\n';

  }

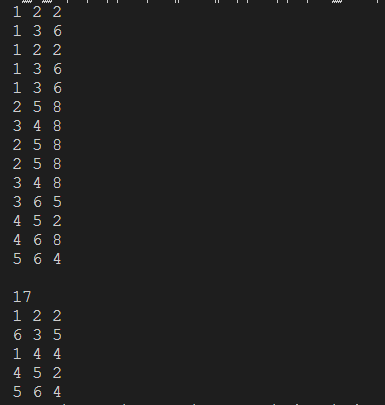
}

## 2.4 Результаты тестирования

Результаты тестирования



**Рисунок 1. Тестирование всех методов задания №2 по графу варианта**



**Рисунок 2. Тестирование всех методов задания №2 по графу варианта**

# ВЫВОДЫ

В процессе выполнения практической работы были получены знания и практические навыки работы с графами в языке C++

# СПИСОК ЛИТЕРАТУРЫ

1. Портал дистанционного обучения РТУ МИРЭА [электронный ресурс]: официальный сайт. – Москва, 2023. – URL: <https://online-edu.mirea.ru/course/view.php?id=4020> (дата обращения 19.11.2024).

2. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.

3. Документация по языку С++ [Электронный ресурс]. URL: <https://docs.microsoft.com/ru-ru/cpp/cpp/> (дата обращения 19.11.2024).

4. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: <https://online-edu.mirea.ru/course/view.php?id=4020> (дата обращения 19.11.2024).