|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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 № 6** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Основные алгоритмы работы с графами»** | |
|  | |
| Выполнил студент группы ИНБО-03-22 | Алтухов А.В. |
| Принял преподаватель | Филатов А.С. |

|  |  |  |
| --- | --- | --- |
| Лабораторная работа выполнена | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись студента)* |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись руководителя)* |

Москва 2023

# **Цель работы**

Получение практических навыков по выполнению операций над структурой данных граф

# **Постановка 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, добавления узла, добавления ребра, вывода графа и методы, выполняющие задачи, определенные вариантом индивидуального задания.
2. Разработать программу, демонстрирующую работу всех методов класса.
3. Произвести тестирование программы на графах, предложенных в таблице 2.
4. Составить отчет, отобразив в нем описание выполнения всех этапов разработки, тестирования и код всей программы со скриншотами результатов тестирования. Скриншоты результатов тестирования сопровождать изображениями графов.

**Вариант №3. Условия задания:**

|  |  |
| --- | --- |
| Представление графа в памяти | Задачи |
| Список смежных вершин | Определить мосты графа.  Составить программу реализации алгоритма Крускала построения остовного дерева минимального веса. Разработать доступный способ (форму) вывода результирующего дерева на экран монитора. |

# **Решение задачи**

Графы в программировании представляют собой абстрактный способ представления различных типов отношений, например, дорог, соединяющих города, и других видов сетей. Графы состоят из вершин и рёбер. Вершина — это точка на графе, а ребро — это то, что соединяет две точки на графе.

Сфера применения графов в программировании очень обширна. Они часто используются в алгоритмах поиска решений, например, для нахождения кратчайшего пути по маршруту, эффективного расположения дорожек на схеме, победной игровой стратегии и т.п.

Чтобы описать граф, нам потребуется где-то хранить ребра, которые исходят из вершины. Для этого я написал структуру данных Edge

|  |
| --- |
| struct Edge {  int from, to, weight;  Edge(int f, int t, int w) : from(f), to(t), weight(w) {}  bool operator<(const Edge& other) const {  return weight < other.weight;  }  }; |

Сам граф описан в классе Graph, у которого есть ряд методов, о которых дальше пойдет речь.

|  |
| --- |
| class Graph {  private:  unordered\_map<int, vector<int>> adjacencyList;  vector<Edge> edges;  public:  void addNode(int node);  void addEdge(int from, int to, int weight);  void printGraph();  void findBridges();  void runKruskal();  } |

Чтобы добавить ребро в граф был написан открытый метод addEdge, который в качестве параметров принимает номер начальной вершины, конечной и вес ребра. Метод ничего не возвращает.

|  |
| --- |
| void addEdge(int from, int to, int weight) {  if (adjacencyList.find(from) != adjacencyList.end() && adjacencyList.find(to) != adjacencyList.end()) {  adjacencyList[from].push\_back(to);  adjacencyList[to].push\_back(from);  edges.emplace\_back(from, to, weight);  cout << "Ребро между " << from << " и " << to << " с весом " << weight << " добавлено.\n";  }  else {  cout << "Узлы " << from << " и/или " << to << " не существуют.\n";  }  } |

Для вывода графа на экран в виде списка смежных вершин, как указано моим вариантом, был написан метод printGraph, который ничего не получает и не возвращает.

|  |
| --- |
| void printGraph() {  cout << "Граф:\n";  for (const auto& entry : adjacencyList) {  cout << entry.first << " -> ";  for (int neighbor : entry.second) {  cout << neighbor << " ";  }  cout << "\n";  }  } |

Чтобы найти мосты в графе, был написан метод findBridges, который ничего не получает на вход и ничего не возвращает.

|  |
| --- |
| void findBridges() {  unordered\_map<int, int> entryTime;  unordered\_map<int, int> lowestTime;  int time = 0;  for (const auto& entry : adjacencyList) {  int node = entry.first;  if (entryTime.find(node) == entryTime.end()) {  findBridgesHelper(node, -1, entryTime, lowestTime, time);  }  }  } |

Для поиска остовного дерева минимального веса в графе был реализован метод runKruskal алгоритмом Крускала. Метод ничего не возвращает.

|  |
| --- |
| void runKruskal() {  sort(edges.begin(), edges.end());  vector<int> parent(edges.size(), -1);  vector<Edge> result;  for (const Edge& edge : edges) {  int parentFrom = find(parent, edge.from);  int parentTo = find(parent, edge.to);  if (parentFrom != parentTo) {  result.push\_back(edge);  unionSets(parent, parentFrom, parentTo);  }  }  cout << "Остовное дерево минимального веса (алгоритм Крускала):\n";  for (const Edge& edge : result) {  cout << edge.from << " -- " << edge.to << " : " << edge.weight << "\n";  }  } |

Метод find используется в алгоритме Крускала для нахождения корня поддерева, к которому принадлежит вершина. Это вспомогательная операция для определения принадлежности вершин к различным поддеревьям. Метод получает на вход ссылку на вектор и вершину. Возвращает корень поддерева, к которому принадлежит вершина.

|  |
| --- |
| int find(vector<int>& parent, int vertex) {  if (parent[vertex] == -1)  return vertex;  return find(parent, parent[vertex]);  } |

Метод unionSets также используется в алгоритме Крускала. Он объединяет два поддерева, делая одно из вершин другим поддеревом. Метод получает на вход ссылку на вектор и вершины из первого и второго поддерева. Метод ничего не возвращает.

|  |
| --- |
| void unionSets(vector<int>& parent, int x, int y) {  int rootX = find(parent, x);  int rootY = find(parent, y);  parent[rootX] = rootY;  } |

Метод findBridgesHelper используется в алгоритме поиска мостов в графе. Рекурсивно обходя граф, он выявляет мосты и выводит информацию о них. Метод получает на вход текущую вершину, ее родитель, Мапа для отслеживания времени входа в вершины, мапа для отслеживания самого раннего времени и ссылка на переменную, представляющую текущее время. Метод ничего не возвращает.

|  |
| --- |
| void findBridgesHelper(int current, int parent, unordered\_map<int, int>& entryTime, unordered\_map<int, int>& lowestTime, int& time) {  entryTime[current] = lowestTime[current] = ++time;  for (int neighbor : adjacencyList[current]) {  if (neighbor == parent) continue;  if (entryTime.find(neighbor) == entryTime.end()) {  findBridgesHelper(neighbor, current, entryTime, lowestTime, time);  lowestTime[current] = min(lowestTime[current], lowestTime[neighbor]);  if (lowestTime[neighbor] > entryTime[current]) {  cout << "Мост: " << current << " - " << neighbor << "\n";  }  }  else {  lowestTime[current] = min(lowestTime[current], entryTime[neighbor]);  }  }  } |

При запуске программы пользователю предоставляются данные о графе, который создается автоматически (рис. 1), после чего выводится список всех доступных пользователю операций (рис. 2).

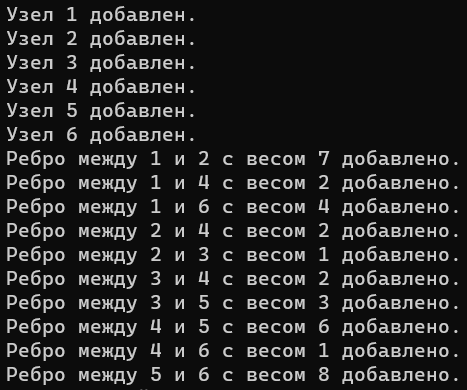


Рисунок 1. Выбор графа при запуске программы

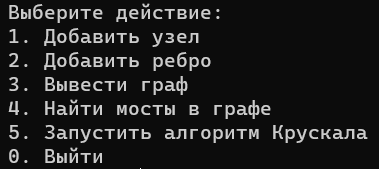


Рисунок 2. Список операций при запуске программы

# **Тестирование программы**

Протестируем все методы, которые нам доступны. Исходный граф представлен на рисунке 3.

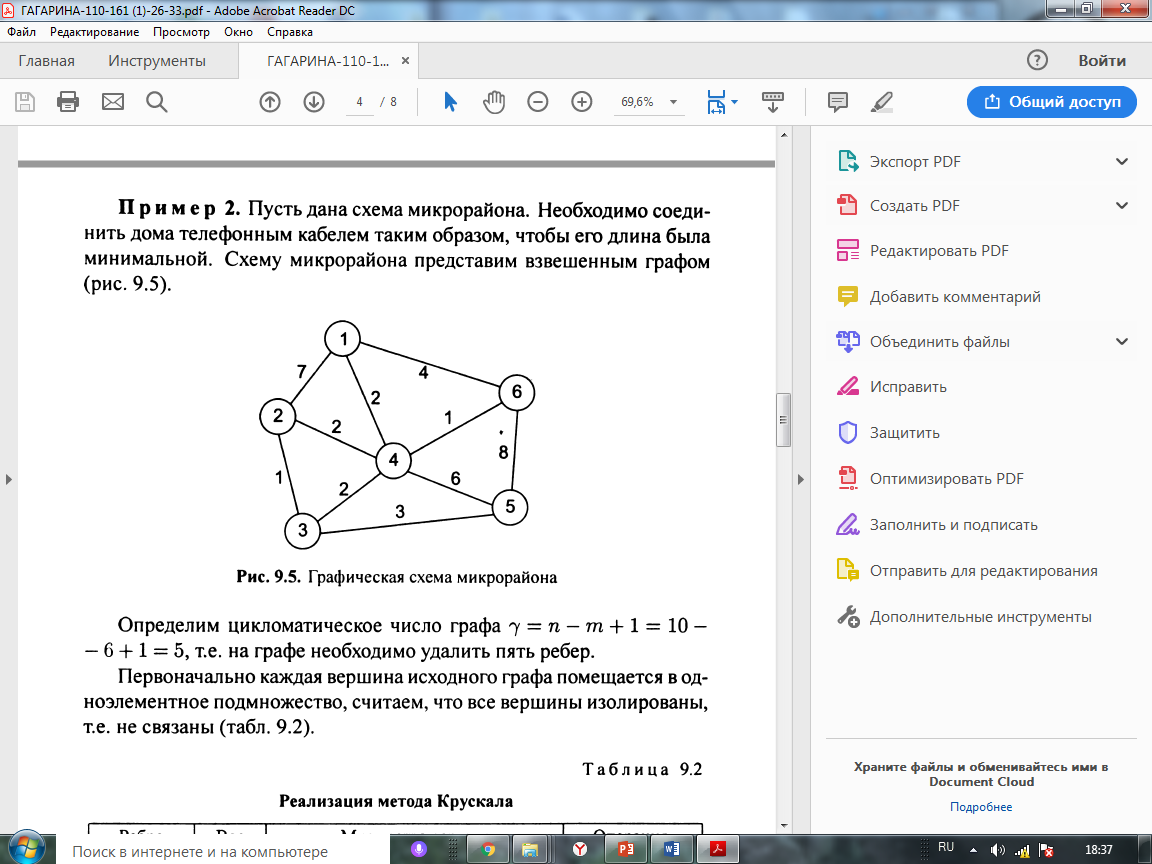


Рисунок 3. Неориентированный граф № 2 из файла с заданием

Приступим к тестированию, и попробуем вывести исходный граф на экран (рис. 4).

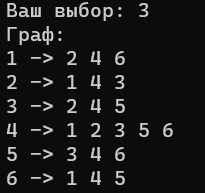


Рисунок 4. Список смежных вершин

Для того чтобы в исходном графе найти мосты, для начала необходимо их сделать. Поэтому добавим новый узел (рис. 5) и новое ребро (рис. 6), а затем попробуем вызвать операцию 4 (рис. 7). Выберем операцию 3, которая должна вывести «рейтинг» вершин по количеству соседей (рис. 7). Глядя на рисунок 7, можно убедиться, что, действительно, теперь у графа есть мосты.

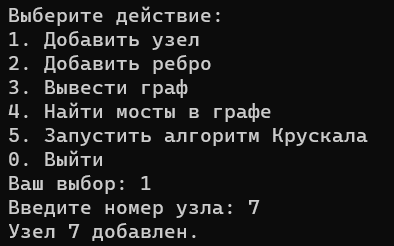


Рисунок 5. Добавление нового узла

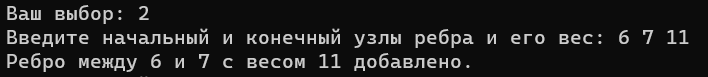


Рисунок 6. Добавление нового ребра

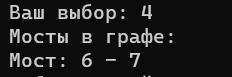


Рисунок 7. Нахождение мостов в графе

Теперь добавим ещё одно ребро (рис. 8), например между 7 и 1, чтобы в графе не оказалось мостов и убедимся в правильности работы метода (рис. 9).

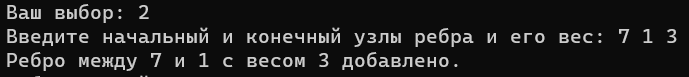


Рисунок 8. Добавление ребра между 7 и 1

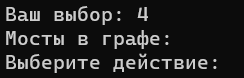


Рисунок 9. Нахождение мостов в графе

Теперь попробуем запустить алгоритм Крускала и проверим его работу (рис. 10).

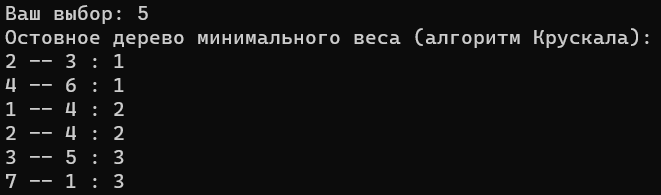


Рисунок 10. Алгоритм Крускала

# **Вывод**

В данной практической работе были получены практические навыки по разработке структуры данных граф и выполнению операций над ней.

# **Исходный код программы**

|  |
| --- |
| #include <iostream>  #include <vector>  #include <unordered\_map>  #include <algorithm>  using namespace std;  struct Edge {  int from, to, weight;  Edge(int f, int t, int w) : from(f), to(t), weight(w) {}  bool operator<(const Edge& other) const {  return weight < other.weight;  }  };  class Graph {  private:  unordered\_map<int, vector<int>> adjacencyList;  vector<Edge> edges;  public:  void addNode(int node) {  if (adjacencyList.find(node) == adjacencyList.end()) {  adjacencyList[node] = vector<int>();  cout << "Узел " << node << " добавлен.\n";  }  else {  cout << "Узел " << node << " уже существует.\n";  }  }  void addEdge(int from, int to, int weight) {  if (adjacencyList.find(from) != adjacencyList.end() && adjacencyList.find(to) != adjacencyList.end()) {  adjacencyList[from].push\_back(to);  adjacencyList[to].push\_back(from);  edges.emplace\_back(from, to, weight);  cout << "Ребро между " << from << " и " << to << " с весом " << weight << " добавлено.\n";  }  else {  cout << "Узлы " << from << " и/или " << to << " не существуют.\n";  }  }  void printGraph() {  cout << "Граф:\n";  for (const auto& entry : adjacencyList) {  cout << entry.first << " -> ";  for (int neighbor : entry.second) {  cout << neighbor << " ";  }  cout << "\n";  }  }  void findBridges() {  unordered\_map<int, int> entryTime;  unordered\_map<int, int> lowestTime;  int time = 0;  for (const auto& entry : adjacencyList) {  int node = entry.first;  if (entryTime.find(node) == entryTime.end()) {  findBridgesHelper(node, -1, entryTime, lowestTime, time);  }  }  }  void runKruskal() {  sort(edges.begin(), edges.end());  vector<int> parent(edges.size(), -1);  vector<Edge> result;  for (const Edge& edge : edges) {  int parentFrom = find(parent, edge.from);  int parentTo = find(parent, edge.to);  if (parentFrom != parentTo) {  result.push\_back(edge);  unionSets(parent, parentFrom, parentTo);  }  }  cout << "Остовное дерево минимального веса (алгоритм Крускала):\n";  for (const Edge& edge : result) {  cout << edge.from << " -- " << edge.to << " : " << edge.weight << "\n";  }  }  private:  int find(vector<int>& parent, int vertex) {  if (parent[vertex] == -1)  return vertex;  return find(parent, parent[vertex]);  }  void unionSets(vector<int>& parent, int x, int y) {  int rootX = find(parent, x);  int rootY = find(parent, y);  parent[rootX] = rootY;  }  void findBridgesHelper(int current, int parent, unordered\_map<int, int>& entryTime, unordered\_map<int, int>& lowestTime, int& time) {  entryTime[current] = lowestTime[current] = ++time;  for (int neighbor : adjacencyList[current]) {  if (neighbor == parent) continue;  if (entryTime.find(neighbor) == entryTime.end()) {  findBridgesHelper(neighbor, current, entryTime, lowestTime, time);  lowestTime[current] = min(lowestTime[current], lowestTime[neighbor]);  if (lowestTime[neighbor] > entryTime[current]) {  cout << "Мост: " << current << " - " << neighbor << "\n";  }  }  else {  lowestTime[current] = min(lowestTime[current], entryTime[neighbor]);  }  }  }  };  int main() {  setlocale(LC\_ALL, "Russian");  Graph graph;  int choice;  // Создаем граф автоматически  graph.addNode(1);  graph.addNode(2);  graph.addNode(3);  graph.addNode(4);  graph.addNode(5);  graph.addNode(6);  graph.addEdge(1, 2, 7);  graph.addEdge(1, 4, 2);  graph.addEdge(1, 6, 4);  graph.addEdge(2, 4, 2);  graph.addEdge(2, 3, 1);  graph.addEdge(3, 4, 2);  graph.addEdge(3, 5, 3);  graph.addEdge(4, 5, 6);  graph.addEdge(4, 6, 1);  graph.addEdge(5, 6, 8);  do {  cout << "Выберите действие:\n";  cout << "1. Добавить узел\n";  cout << "2. Добавить ребро\n";  cout << "3. Вывести граф\n";  cout << "4. Найти мосты в графе\n";  cout << "5. Запустить алгоритм Крускала\n";  cout << "0. Выйти\n";  cout << "Ваш выбор: ";  cin >> choice;  switch (choice) {  case 1: {  int node;  cout << "Введите номер узла: ";  cin >> node;  graph.addNode(node);  break;  }  case 2: {  int from, to, weight;  cout << "Введите начальный и конечный узлы ребра и его вес: ";  cin >> from >> to >> weight;  graph.addEdge(from, to, weight);  break;  }  case 3:  graph.printGraph();  break;  case 4:  cout << "Мосты в графе:\n";  graph.findBridges();  break;  case 5:  graph.runKruskal();  break;  case 0:  cout << "Программа завершена.\n";  break;  default:  cout << "Неверный выбор. Попробуйте еще раз.\n";  }  } while (choice != 0);  return 0;  } |