|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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 ИКБО-10-21 | Смольников А.Б. |
| Принял преподаватель | Филатов А.С. |

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

Москва 2022

# **Цель работы**

Получение практических навыков по выполнению операций над структурой данных граф.

# **Постановка задачи**

* 1. **Задача 1**

Вариант 22. Вид дерева: АВЛ

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 метод вывода графа и методы, выполняющие задачи, определенные вариантом индивидуального задания.
2. Разработать программу, демонстрирующую работу всех методов класса.
3. Произвести тестирование программы на графе, предложенном в таблице Составить отчет, отобразив в нем описание выполнения всех этапов разработки, тестирования и код всей программы со скриншотами результатов тестирования.

Таблица 1. Задания варианта №22

|  |  |
| --- | --- |
| Задание 1 | 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**

* 1. **Теоретическое введение**

Граф - топологическая модель, состоящая из множества вершин и множества ребер, соединяющих эти вершины. Дерево – связный граф, в котором нет циклов. Это означает, что из любой вершины можно попасть в любую, притом единственным путем. Деревья часто используются для организации иерархической структуры данных, например, при создании двоичных деревьев поиска или кучи, в этом случае одну вершину дерева называют корнем. Лист – вершина, не имеющая потомков. Обход дерева – систематический просмотр всех вершин, при котором каждая вершина встречается один раз.

Обход графа – алгоритм, в котором рассматриваются все вершины графа.

Вершина - точка в графе, для топологической модели не имеет значения координата вершины.

Ребро - неупорядоченная пара двух вершин, которые связаны друг с другом. Эти вершины называются концевыми точками или концами ребра. У ребра может быть вес.

Вершина и ребро называются инцидентными, если вершина является для этого ребра концевой.

Две вершины называются смежными, если они инцидентны одному ребру.

Два ребра называются смежными, если они инцедентны одной вершине.

Циклом в графе называется такая конечная цепь, которая начинается и заканчивается в одной вершине.

Связный граф — граф, содержащий ровно одну компоненту связности.

Мост – такое ребро, при удалении которого количество компонент связности увеличивается.

Остовное дерево — ациклический связный подграф данного связного неориентированного графа, в который входят все его вершины.

Список связных вершин – такой список длины V, который в каждой ячейке хранит номера вершин, смежных с данной вершиной.

* 1. **Функции задания №1**

Для организации хранения графа в памяти используется класс Graph, который содержит в себе информацию о графе в виде списка связных вершин и содержит необходимые методы для выполнения задачи. Списком смежных вершин является контейнер структур adjacent. Хранит в себе номер вершины и список граней, выходящих из данной вершины. Элемент списка граней – структура edge, хранящая номер конечной вершины и вес ребра:

|  |
| --- |
| class Graph {  struct edge{  int vertexEdge;  int weight;  };  struct adjacent{  int vertex;  vector<edge> edges;  };  struct edge2way{  int vertex1;  int vertex2;  int weight;  };  int size=0;  vector<adjacent> graph = vector<adjacent>(0);  vector<bool> PrintVisited;  public:  Graph(int size);  void addEdge(int vertex1, int vertex2, int weight, bool doRepeat = true);  void removeEdge(int vertex1, int vertex2);  void printGraphList();  void findBridges();  void dfs(int v, vector<bool> &visited, vector<int> &timeIn, vector<int> &fup, vector<int> &parent, int &timer);  Graph Kruksal();  bool isCycled(int v, vector<bool> &visited, int parent=-1);  void toGraphviz(std::string filename);  void printTree(std::string prefix="", bool root=true, int vertex=0);  }; |

Конструктор Graph принимает на вход количество вершин графа и инициализирует список смежных вершин данной топологической модели. В конце выполнения в памяти хранится граф без ребер.

|  |
| --- |
| Graph::Graph(int size) {  this->size = size;  for (int i = 0; i < size; ++i) {  adjacent adjacent1;  adjacent1.vertex = i;  graph.push\_back(adjacent1);  }  } |

Метод addEdge() принимает на вход пару номеров вершин, вес ребра и параметр по умолчанию doRepeat, позволяющий строить ориентированные графы. В процессе выполнения проверяет данные на корректность и добавляет ребро в список смежных вершин.

|  |
| --- |
| void Graph::addEdge(int vertex1, int vertex2, int weight, bool doRepeat) {  if(vertex1>=size || vertex2>=size){  cerr<<"Error: vertex out of range"<<endl;  return;  }  edge newEdge;  newEdge.vertexEdge = vertex2;  newEdge.weight = weight;  if(vertex1!=vertex2 and doRepeat){  edge newEdge2;  newEdge2.vertexEdge = vertex1;  newEdge2.weight = weight;  graph[vertex2].edges.push\_back(newEdge2);  }  graph[vertex1].edges.push\_back(newEdge);  } |

Метод printGraphList() выводит в консоль список смежных вершин графа и вес соответствующих ребер, итеративно получая значения из структуры данных.

|  |
| --- |
| void Graph::printGraphList() {  for (int i = 0; i < graph.size(); ++i) {  cout << graph[i].vertex << ": ";  for (int j = 0; j < graph[i].edges.size(); ++j) {  cout << graph[i].edges[j].vertexEdge << "(" << graph[i].edges[j].weight << ") ";  }  cout << endl;  }  } |

Метод findBridges() находит мосты графа, обходя его в глубину, начиная с каждой вершины, при помощи вспомогательного метода dfs()

|  |
| --- |
| void Graph::findBridges() {  vector<int> timeIn(size, -1);  vector<int> fup(size, -1);  vector<bool> visited(size, false);  vector<int> parent(size, -1);  int timer = 0;  for (int i = 0; i < size; ++i) {  if (!visited[i]) {  dfs(i, visited, timeIn, fup, parent, timer);  }  }  } |

Метод dfs() принимает на вход номер вершины, массив посещенных вершин, массив времен захода, массив времен fup, равных минимуму из времени захода в саму вершину, времён захода в каждую вершину p, являющуюся концом некоторого обратного ребра, массив родительских вершин и сам таймер. В процессе выполнения рекурсивно обходит вершины, проверяя граф на компоненты связности. Метод является вспомогательным и реализует обход в глубину, выводит в консоль найденные мосты графа.

|  |
| --- |
| void Graph::dfs(int v, vector<bool> &visited, vector<int> &timeIn, vector<int> &fup, vector<int> &parent, int &timer){  visited[v] = true;  timeIn[v] = fup[v] = timer++;  for (int i = 0; i < graph[v].edges.size(); ++i) {  int to = graph[v].edges[i].vertexEdge;  if (to == parent[v]) {  continue;  }  if (visited[to]) {  fup[v] = min(fup[v], timeIn[to]);  } else {  parent[to] = v;  dfs(to, visited, timeIn, fup, parent, timer);  fup[v] = min(fup[v], fup[to]);  if (fup[to] > timeIn[v]) {  cout<<"BRIDGE "<< v << " " << to << endl;  }  }  }  } |

Метод Kruskal() реализует алгоритм Крускала по поиску минимального остовного дерева данного графа. В процессе выполнения создается новый граф, заполняется данными из исходного, удаляются все дубликаты ребер (необходимо для корректной работы). Далее ребра сортируются по возрастанию их веса и добавляются в новый граф, если их появление не создает цикла.

|  |
| --- |
| Graph Graph::Kruksal() {  vector<edge2way> edgesKruskal;  for (int i = 0; i < graph.size(); ++i) {  for (int j = 0; j < graph[i].edges.size(); ++j) {  edge2way edge;  edge.vertex1 = graph[i].vertex;  edge.vertex2 = graph[i].edges[j].vertexEdge;  edge.weight = graph[i].edges[j].weight;  edgesKruskal.push\_back(edge);  }  }  //delete duplicate edges  for (int i = 0; i < edgesKruskal.size(); ++i) {  for (int j = i+1; j < edgesKruskal.size(); ++j) {  if(edgesKruskal[i].vertex1 == edgesKruskal[j].vertex2 && edgesKruskal[i].vertex2 == edgesKruskal[j].vertex1){  edgesKruskal.erase(edgesKruskal.begin()+j);  j--;  }  }  }  //sort edges by weight  sort(edgesKruskal.begin(), edgesKruskal.end(), [](edge2way a, edge2way b){  return a.weight < b.weight;  });  for(auto e: edgesKruskal){  cout<<e.vertex1<<" "<<e.vertex2<<" "<<e.weight<<endl;  }  Graph graphKruskal(size);  //add edges to graph if they don't create cycle  for (int i = 0; i < edgesKruskal.size(); ++i) {  vector<bool> visited(size, false);  graphKruskal.addEdge(edgesKruskal[i].vertex1, edgesKruskal[i].vertex2, edgesKruskal[i].weight, true);  cout<<"Added edge "<<edgesKruskal[i].vertex1<<" "<<edgesKruskal[i].vertex2<<endl;  if(graphKruskal.isCycled(edgesKruskal[i].vertex1, visited)){  cout<<"CYCLE "<<edgesKruskal[i].vertex1<<" "<<edgesKruskal[i].vertex2<<endl;  graphKruskal.removeEdge(edgesKruskal[i].vertex1, edgesKruskal[i].vertex2);  }  }  return graphKruskal;  } |

Метод removeEdge() принимает на вход два номера вершин и удаляет ребро из графа. Метод является вспомогательным и необходим для реализации алгоритма Крускала.

|  |
| --- |
| void Graph::removeEdge(int vertex1, int vertex2) {  for (int i = 0; i < graph[vertex1].edges.size(); ++i) {  if(graph[vertex1].edges[i].vertexEdge == vertex2){  graph[vertex1].edges.erase(graph[vertex1].edges.begin()+i);  break;  }  }  for (int i = 0; i < graph[vertex2].edges.size(); ++i) {  if(graph[vertex2].edges[i].vertexEdge == vertex1){  graph[vertex2].edges.erase(graph[vertex2].edges.begin()+i);  break;  }  }  } |

Метод isCycled() принимает на вход номер вершины, массив посещенных вершин, номер родительского узла. Выполняет рекурсивную проверку графа на ацикличность при помощи упрощенного обхода в глубину. Является вспомогательным для имплементации алгоритма Крускала.

|  |
| --- |
| //check acyclic using dfs  bool Graph::isCycled(int v, vector<bool> &visited, int parent) {  visited[v] = true;  for (int i = 0; i < graph[v].edges.size(); ++i) {  int to = graph[v].edges[i].vertexEdge;  if (to == parent) {  continue;  }  if (visited[to]) {  return true;  } else {  if(isCycled(to, visited, v)){  return true;  }  }  }  return false;  } |

Метод printTree() получает на вход строку префикса, логическое значение корня и номер вершины. Рекурсивно выводит в консоль полученное остовное дерево, изменяя префикс для наглядного вывода:

|  |
| --- |
| void Graph::printTree(std::string prefix, bool root, int vertex) {  if(root){  PrintVisited = vector<bool>(size, false);  cout<<vertex<<endl;  }  PrintVisited[vertex] = true;  adjacent element = graph[vertex];  int adj\_start = element.vertex;  for(edge ed : element.edges){  int adj\_end = ed.vertexEdge;  if(!PrintVisited[adj\_end]){  cout << prefix;  if(root){  cout << "└──";  }else{  cout << "├──";  }  cout << adj\_end << endl;  printTree(prefix + (root ? "│ " : "│ "), false, adj\_end);  }  // PrintVisited[adj\_end] = true;  }  } |

* 1. **Интерфейс**

При запуске программы пользователю необходимо ввести количество вершин, количество граней и тройки значений начало-конец-вес. После этого запустится основной алгоритм, выполняющий все задачи.

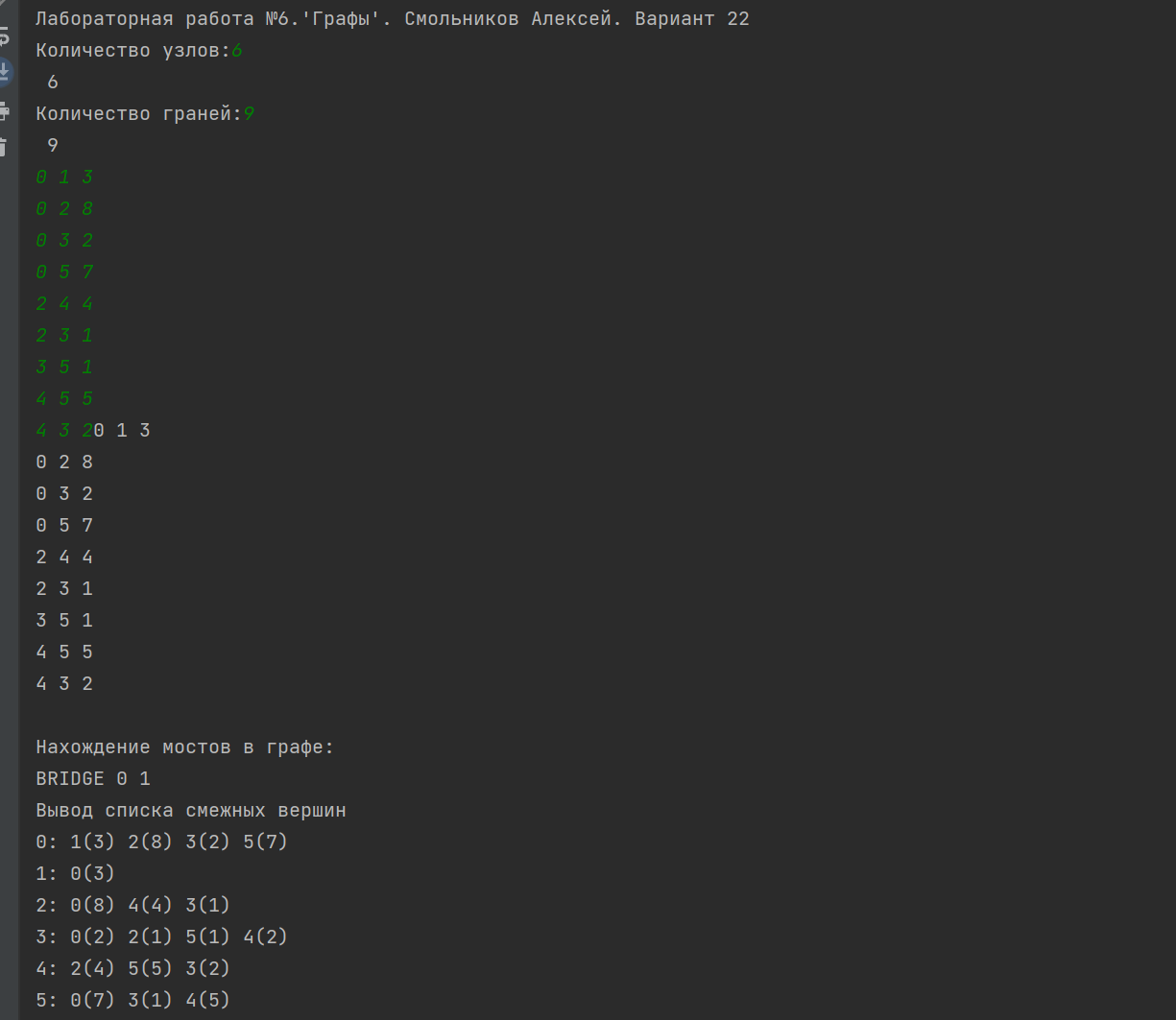


Рисунок 1. Интерфейс программы

# **Тестирование**

Тестирование проведем на графе 1 (риснок 2). Введем данные в нужном виде, рассмотрим список смежных вершин. Видно, что граф создается корректно, список соответствует действительности (рисунок 3).

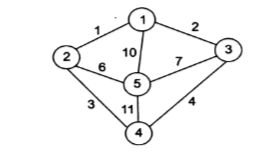


Рисунок 2. Исходный граф

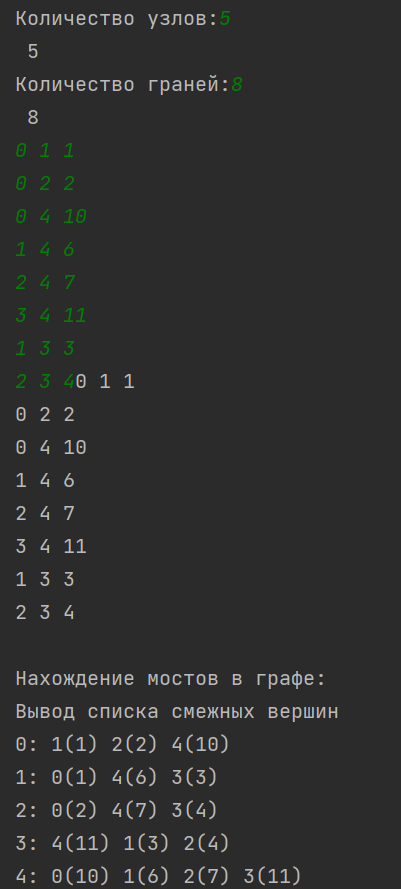


Рисунок 3. Результат заполнения графа

Нахождение мостов проверим на графе 12 (рисунок 4). Из рисунка 5 видно, что алгоритм работает корректно.

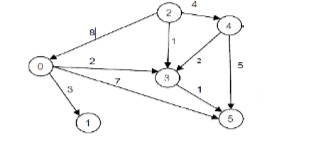


Рисунок 4. Линейный поиск

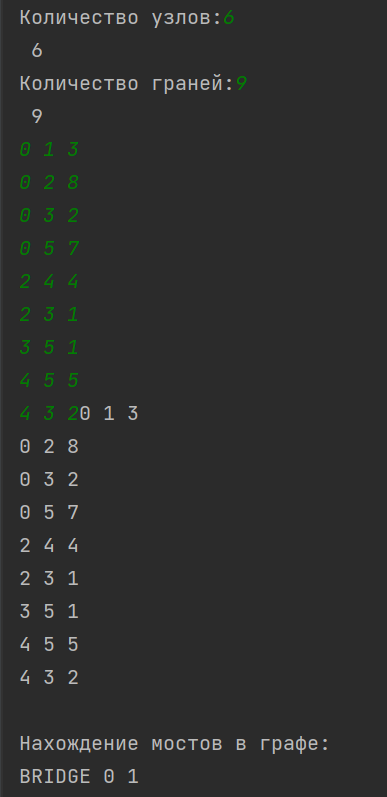


Рисунок 5. Поиск моста в графе

Для этого же графа (рисунок 4) выведем отсортированные ребра и циклы. Из рисунка 6 видно, что алгоритм работает корректно. Также из рисунка 7 заметно, что алгоритм Крускала работает корректно.

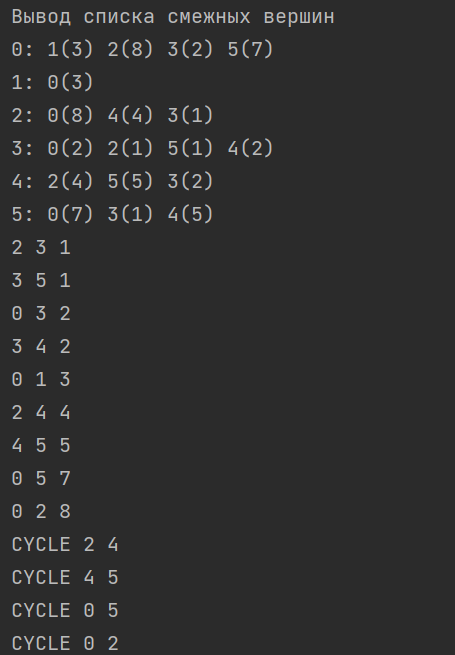


Рисунок 6. Добавление записи

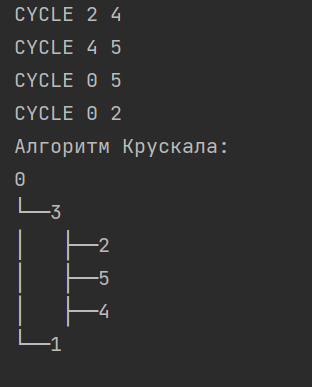


Рисунок 7. Результат работы алгоритма Крускала

При помощи средств языка Graphviz реализован графический вывод графов. Сравним граф 12 (рисунок 4) до и после применения алгоритма Крускала (рисунки 8 и 9), программа работает корректно:

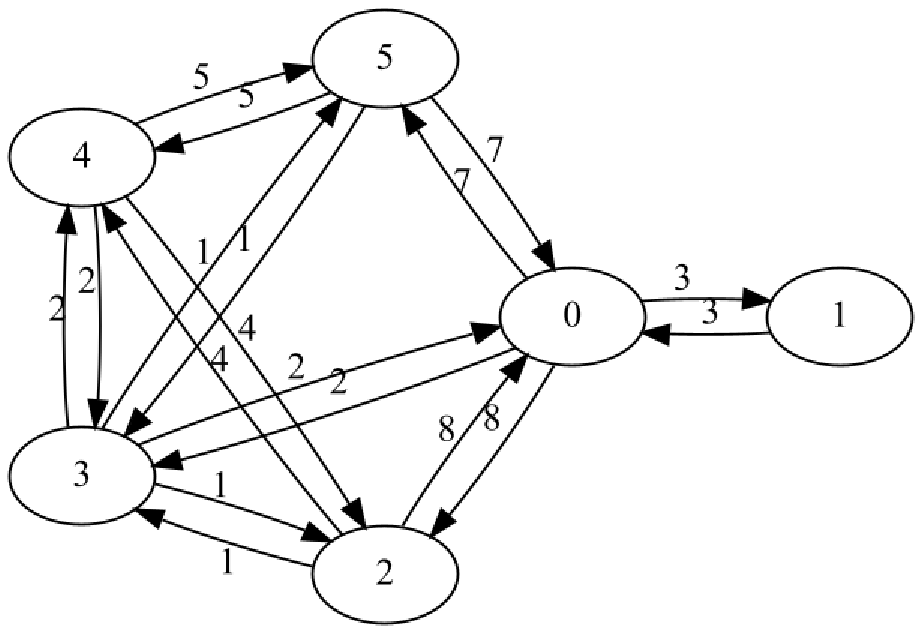


Рисунок 8. Введенное дерево

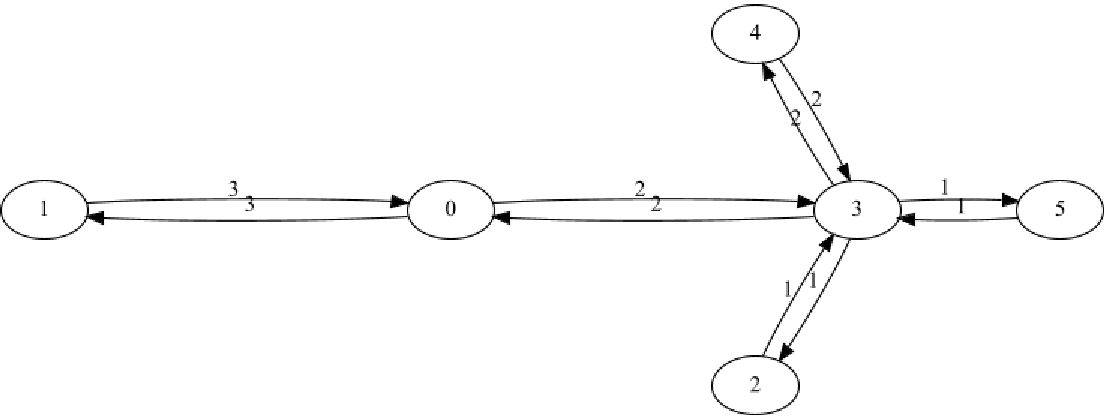


Рисунок 9. Полученное минимальное остовное дерево

# **Выводы**

В результате выполнения работы были:

* получены навыки в разработки и реализации алгоритмов управления бинарным деревом поиска и сбалансированными бинарными деревьями поиска (АВЛ – деревьями);
* получены навыки в применении файловых потоков прямого доступа к данным файла;
* получены навыки в применении сбалансированного дерева поиска для прямого доступа к записям файла.

# **Исходный код программы**

main.cpp

|  |
| --- |
| #include <iostream>  #include <string>  #include "Graph.h"  using namespace std;  int main()  {  system("chcp 65001");  int n,a,b,w;  cout<<"Лабораторная работа №6.'Графы'. Смольников Алексей. Вариант 22"<<endl;  cout<<"Количество узлов: ";  cin>>n;  cout<<"Количество граней: ";  int e;  cin>>e;  Graph graph(n);  for (int i=0;i<e;i++)  {  cin>>a;  cin>>b;  cin>>w;  graph.addEdge(a,b,w);  }  cout<<"Нахождение мостов в графе: "<<endl;  graph.findBridges();  cout<<"Вывод списка смежных вершин"<<endl;  graph.printGraphList();  graph.toGraphviz("graph.gv");  Graph graph2 = graph.Kruksal();  graph2.toGraphviz("graphKruksal.gv");  cout<<"Алгоритм Крускала: "<<endl;  graph2.printTree();  return 0;  } |

Graph.h

|  |
| --- |
| //  // Created by lexa2k on 15.11.2022.  //  #ifndef SIAOD2SEM\_GRAPH\_H  #define SIAOD2SEM\_GRAPH\_H  #include <iostream>  #include <vector>  #include <queue>  #include <string>  #include <fstream>  #include <algorithm>  using namespace std;  // По заданию гарантируется, что вершины нащзываются от 0 до n-1,  // а при создании графа мы создаем n вершин, так что оператор индексирования работает корректно  class Graph {  struct edge{  int vertexEdge;  int weight;  };  struct adjacent{  int vertex;  vector<edge> edges;  };  struct edge2way{  int vertex1;  int vertex2;  int weight;  };  int size=0;  vector<adjacent> graph = vector<adjacent>(0);  vector<bool> PrintVisited;  public:  Graph(int size);  void addEdge(int vertex1, int vertex2, int weight, bool doRepeat = true);  void removeEdge(int vertex1, int vertex2);  void printGraphList();  void findBridges();  void dfs(int v, vector<bool> &visited, vector<int> &timeIn, vector<int> &fup, vector<int> &parent, int &timer);  Graph Kruksal();  bool isCycled(int v, vector<bool> &visited, int parent=-1);  void toGraphviz(std::string filename);  void printTree(std::string prefix="", bool root=true, int vertex=0);  };  #endif //SIAOD2SEM\_GRAPH\_H |

Graph.cpp

|  |
| --- |
| //  // Created by lexa2k on 15.11.2022.  //  #include "Graph.h"  Graph::Graph(int size) {  this->size = size;  for (int i = 0; i < size; ++i) {  adjacent adjacent1;  adjacent1.vertex = i;  graph.push\_back(adjacent1);  }  }  void Graph::addEdge(int vertex1, int vertex2, int weight, bool doRepeat) {  if(vertex1>=size || vertex2>=size){  cerr<<"Error: vertex out of range"<<endl;  return;  }  edge newEdge;  newEdge.vertexEdge = vertex2;  newEdge.weight = weight;  if(vertex1!=vertex2 and doRepeat){  edge newEdge2;  newEdge2.vertexEdge = vertex1;  newEdge2.weight = weight;  graph[vertex2].edges.push\_back(newEdge2);  }  graph[vertex1].edges.push\_back(newEdge);  }  void Graph::printGraphList() {  for (int i = 0; i < graph.size(); ++i) {  cout << graph[i].vertex << ": ";  for (int j = 0; j < graph[i].edges.size(); ++j) {  cout << graph[i].edges[j].vertexEdge << "(" << graph[i].edges[j].weight << ") ";  }  cout << endl;  }  }  //https://e-maxx.ru/algo/bridge\_searching  //https://e-maxx.ru/algo/dfs  void Graph::findBridges() {  vector<int> timeIn(size, -1);  vector<int> fup(size, -1);  vector<bool> visited(size, false);  vector<int> parent(size, -1);  int timer = 0;  for (int i = 0; i < size; ++i) {  if (!visited[i]) {  dfs(i, visited, timeIn, fup, parent, timer);  }  }  }  void Graph::dfs(int v, vector<bool> &visited, vector<int> &timeIn, vector<int> &fup, vector<int> &parent, int &timer){  visited[v] = true;  timeIn[v] = fup[v] = timer++;  for (int i = 0; i < graph[v].edges.size(); ++i) {  int to = graph[v].edges[i].vertexEdge;  if (to == parent[v]) {  continue;  }  if (visited[to]) {  fup[v] = min(fup[v], timeIn[to]);  } else {  parent[to] = v;  dfs(to, visited, timeIn, fup, parent, timer);  fup[v] = min(fup[v], fup[to]);  if (fup[to] > timeIn[v]) {  cout<<"BRIDGE "<< v << " " << to << endl;  }  }  }  }  Graph Graph::Kruksal() {  vector<edge2way> edgesKruskal;  for (int i = 0; i < graph.size(); ++i) {  for (int j = 0; j < graph[i].edges.size(); ++j) {  edge2way edge;  edge.vertex1 = graph[i].vertex;  edge.vertex2 = graph[i].edges[j].vertexEdge;  edge.weight = graph[i].edges[j].weight;  edgesKruskal.push\_back(edge);  }  }  //delete duplicate edges  for (int i = 0; i < edgesKruskal.size(); ++i) {  for (int j = i+1; j < edgesKruskal.size(); ++j) {  if(edgesKruskal[i].vertex1 == edgesKruskal[j].vertex2 && edgesKruskal[i].vertex2 == edgesKruskal[j].vertex1){  edgesKruskal.erase(edgesKruskal.begin()+j);  j--;  }  }  }  //sort edges by weight  sort(edgesKruskal.begin(), edgesKruskal.end(), [](edge2way a, edge2way b){  return a.weight < b.weight;  });  for(auto e: edgesKruskal){  cout<<e.vertex1<<" "<<e.vertex2<<" "<<e.weight<<endl;  }  Graph graphKruskal(size);  //add edges to graph if they don't create cycle  for (int i = 0; i < edgesKruskal.size(); ++i) {  vector<bool> visited(size, false);  graphKruskal.addEdge(edgesKruskal[i].vertex1, edgesKruskal[i].vertex2, edgesKruskal[i].weight, true);  // cout<<"Added edge "<<edgesKruskal[i].vertex1<<" "<<edgesKruskal[i].vertex2<<endl;  if(graphKruskal.isCycled(edgesKruskal[i].vertex1, visited)){  cout<<"CYCLE "<<edgesKruskal[i].vertex1<<" "<<edgesKruskal[i].vertex2<<endl;  graphKruskal.removeEdge(edgesKruskal[i].vertex1, edgesKruskal[i].vertex2);  }  }  return graphKruskal;  }  void Graph::removeEdge(int vertex1, int vertex2) {  for (int i = 0; i < graph[vertex1].edges.size(); ++i) {  if(graph[vertex1].edges[i].vertexEdge == vertex2){  graph[vertex1].edges.erase(graph[vertex1].edges.begin()+i);  break;  }  }  for (int i = 0; i < graph[vertex2].edges.size(); ++i) {  if(graph[vertex2].edges[i].vertexEdge == vertex1){  graph[vertex2].edges.erase(graph[vertex2].edges.begin()+i);  break;  }  }  }  //check acyclic using dfs  bool Graph::isCycled(int v, vector<bool> &visited, int parent) {  visited[v] = true;  for (int i = 0; i < graph[v].edges.size(); ++i) {  int to = graph[v].edges[i].vertexEdge;  if (to == parent) {  continue;  }  if (visited[to]) {  return true;  } else {  if(isCycled(to, visited, v)){  return true;  }  }  }  return false;  }  void Graph::toGraphviz(std::string filename) {  ofstream file;  file.open(filename);  file << "digraph G {" << endl;  file << "layout=circo;"<<endl;  for (int i = 0; i < size; ++i) {  for (int j = 0; j < graph[i].edges.size(); ++j) {  file << graph[i].vertex << " -> " << graph[i].edges[j].vertexEdge << " [label=" << graph[i].edges[j].weight << "]" << endl;  }  }  file << "}";  file.close();  }  //pretty print tree in console  void Graph::printTree(std::string prefix, bool root, int vertex) {  if(root){  PrintVisited = vector<bool>(size, false);  cout<<vertex<<endl;  }  PrintVisited[vertex] = true;  adjacent element = graph[vertex];  int adj\_start = element.vertex;  for(edge ed : element.edges){  int adj\_end = ed.vertexEdge;  if(!PrintVisited[adj\_end]){  cout << prefix;  if(root){  cout << "└──";  }else{  cout << "├──";  }  cout << adj\_end << endl;  printTree(prefix + (root ? "│ " : "│ "), false, adj\_end);  }  // PrintVisited[adj\_end] = true;  }  } |