Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: Алгоритм Прима и Крускала

Студентка гр.3331506/00401 Щелканова В.А.

Преподаватель Ананьевский М.С.

Санкт-Петербург

2023

Оглавление

Введение

Алгоритмы Прима и Крускала используются для решения проблемы минимального остовного дерева и находят применение в области транспорта, телекоммуникаций, визуализации данных и других областях. В данной курсовой работе мы рассмотрим оба алгоритма, их применения в геоинформационных системах и сравним их эффективность.

Минимальное остовное дерево — это подграф взвешенного связного неориентированного графа, который является деревом и содержит все вершины из исходного графа, обладая минимальной суммой весов рёбер. Оно используется для оптимизации сетевых структур и связных систем.

Алгоритмы Прима и Крускала являются жадными. Жадный алгоритм - это алгоритм решения задачи, который на каждом шаге выбирает локально оптимальное решение в надежде получения глобально оптимального результата.

Целью данной работы является изучение алгоритмов Прима и Крускала, инструментирование их с помощью языка C++ и реализация алгоритмов. Задачей курсовой работы является сравнение результатов работы алгоритмов, анализ возможной оптимизации алгоритмов и описание основных принципов функционирования.

2. Алгоритм Прима

2.1 Определение алгоритма Прима

Алгоритм Прима - это алгоритм минимального остовного дерева, который используется для поиска минимального связующего дерева во взвешенном связном графе. Алгоритм начинает с произвольной вершины и на каждой итерации добавляет к дереву вершину с наименьшим весовым ребром из множества еще непосещенных вершин, обновляя множество ребер после каждой операции.

2.2 Преимущества алгоритма Прима

- Гарантированная нахождение минимального остовного дерева.

- Работает эффективнее, чем алгоритм Крускала на практике.

- Может использоваться для построения остовных деревьев в графах с большой плотностью.

2.3 Недостатки алгоритма Прима

- Работает медленнее, чем алгоритм Крускала в случае разреженных графов.

- Требуется дополнительная структура данных для поиска ребра с наименьшим весом.

2.4 Оценка сложность алгоритма Прима

Общая сложность алгоритма оценивается с помощью формулы 1.1.

$$O = V^2, \tag{1.1}$$

где V – количество вершин графа.

На рисунке 1 приведен график зависимости сложности алгоритма от количества вершин графа.

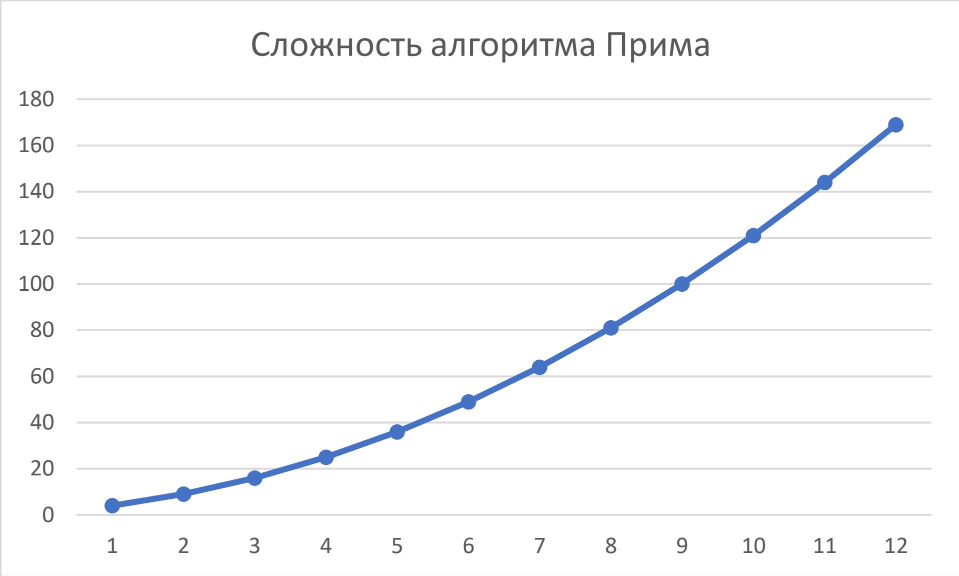

Рисунок 1

2.5 Быстродействие алгоритма Прима

Также были произведены экспериментальные исследования быстродействия работы алгоритма. На вход Программе подавался полный граф с разным количеством вершин от 2 до 1000. Полученный график представлен на рисунке 2.

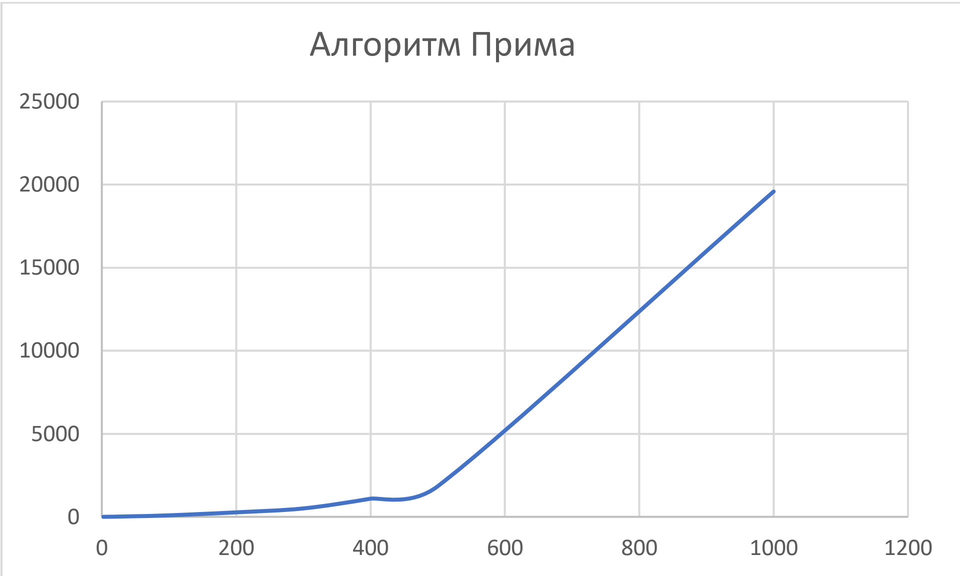

Рисунок 2

2.6. Принципы работы алгоритма Прима

На примере графа, изображенного на рисунке 3 будет произведено построение минимального остовного дерева с помощью алгоритма Прима.

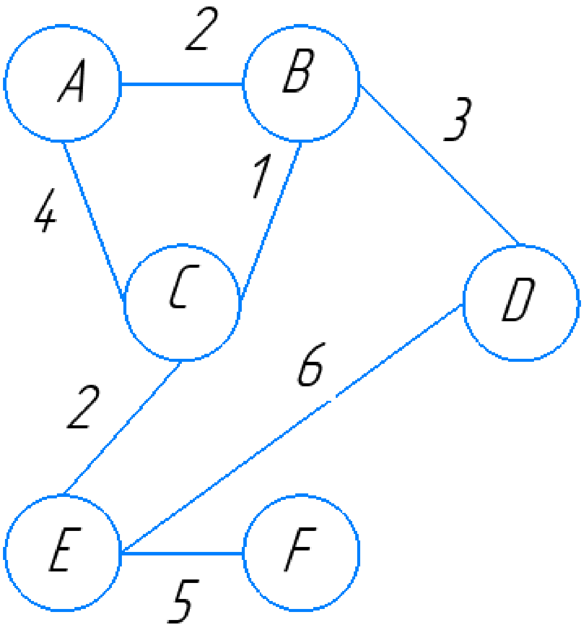

Рисунок 3

Для применения алгоритма Прима нам необходимо выбрать начальную вершину. Для примера выберем вершину A.

Шаг 1: Из вершины A выходят два ребра в вершины B и C с весами 2 и 4 соответственно. Мы выбираем ребро с наименьшим весом, то есть AB. Помечаем вершины A и B как посещенные и добавляем ребро AB в дерево остова.

Шаг 2: Теперь выбираем ребро с минимальным весом из не посещенных вершин, которые соединены с уже посещенными вершинами. Таким ребром является BC с весом 1. Мы помечаем вершину C как посещенную и добавляем ребро BC в дерево остова.

Шаг 3: Переходим к ребру CE с весом 2.

Шаг 4: После добавления ребра CE мы можем выбрать следующее ребро, которым является ребро BD с весом 3.

Шаг 5: Наконец, мы добавляем последнее ребро EF с весом 5.

Последовательность шагов и конечное остовое дерево изображено на рисунке 4. Вес этого дерева составляет 13.

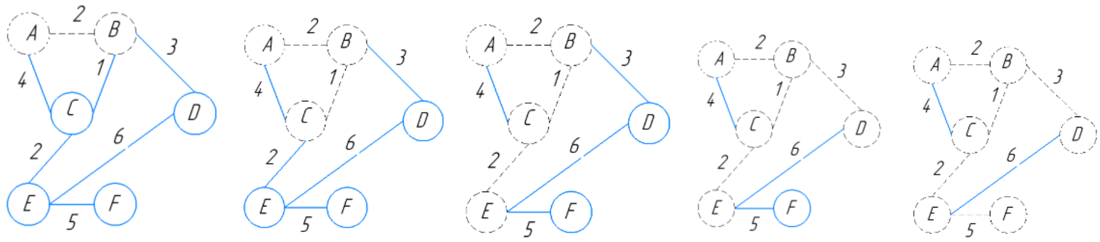

Рисунок 4.

3. Алгоритм Крускала

3.1 Определение алгоритма Крускала

Алгоритм Крускала — это алгоритм нахождения минимального остовного дерева в связном неориентированном графе. Он основан на добавлении в остовное дерево ребер в порядке их возрастания веса до тех пор, пока все вершины не станут связными.

3.2 Преимущества алгоритма Крускала

- Гарантированное нахождение минимального остовного дерева.

- Обладает лучшей асимптотикой временной сложности, чем алгоритм Прима, на разреженных графах.

- Не требует дополнительных структур данных для поиска ребер с наименьшим весом.

3.3 Недостатки алгоритма Крускала

- Может приводить к построению громоздких деревьев в плотных графах, из-за большого числа ребер.

- Медленнее работает на плотных графах по сравнению с алгоритмом Прима.

3.4 Оценка сложности алгоритма Крускала

Общая временная сложность алгоритма оценивается по формуле 1.2

$$(E \cdot logE + E \cdot logV)\,, \qquad (1.2)$$

где $E -$ количесвто ребер, $V -$ количество вершин.

На рисунке 5 приведен график зависимости сложности алгоритма от количества вершин (при условии, что граф - полный).

Рисунок 5

3.5 Быстродействие алгоритма Крускала

Также были произведены экспериментальные исследования быстродействия работы алгоритма. На вход Программе подавался полный граф с разным количеством вершин от 2 до 1000. Полученный график представлен на рисунке 6.

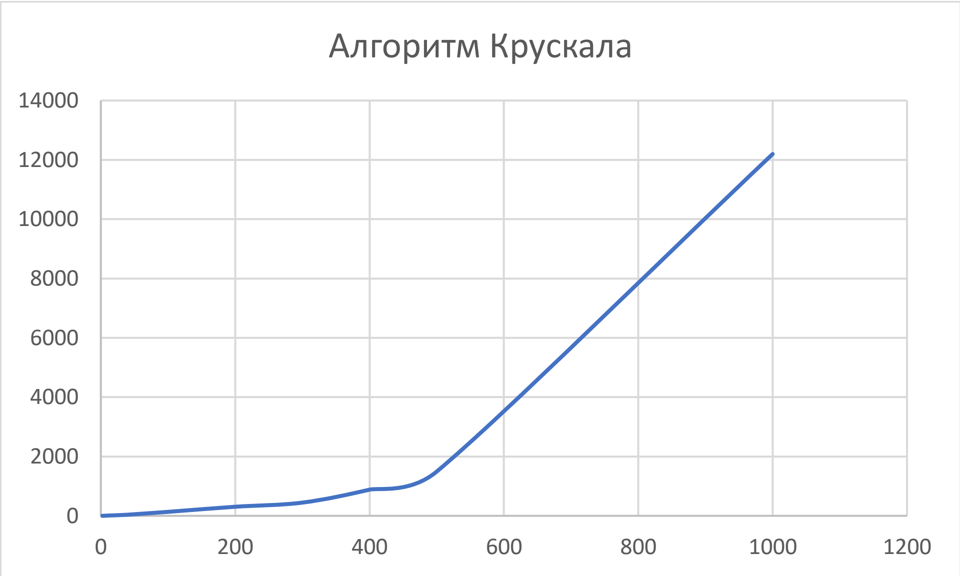

Рисунок 6

3.6 Принципы работы алгоритма Крускала

На примере графа, изображенного на рисунке 7 будет произведено построение минимального остовного дерева с помощью алгоритма Крускала.

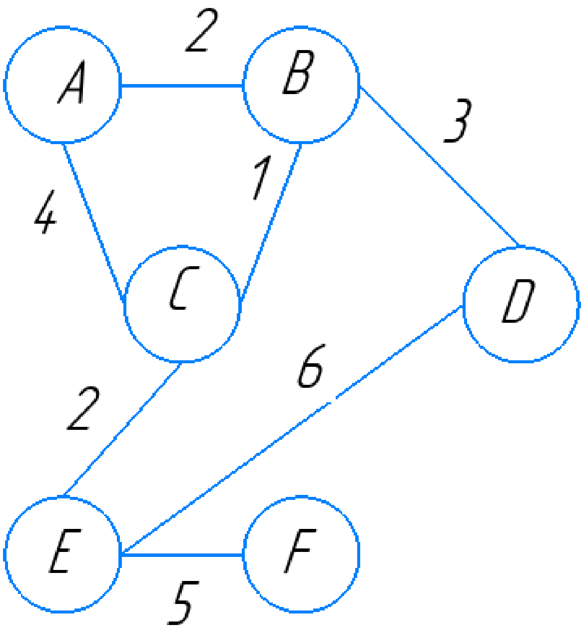

Рисунок 7

Шаг 1: Создаем лес из тривиальных деревьев (графов, состоящих из одной вершины).

Шаг 2: Расположим ребра в порядке возрастания веса:

BC – 1, AB – 2, CE – 2, BD – 3, AC – 4, EF – 5, DE – 6.

Шаг 3: Берем первое ребро из списка и проверяем, принадлежат ли его вершины одному дереву. Если да, то отбрасываем это ребро. Если нет, то соединяем два дерева в одно, используя это ребро.

Шаг 4: Берем второе ребро из списка, и соединяем вершины A и B.

Проделываем шаг 3 до тех пор пока дерево не будет построено.

Последовательность выполнения алгоритма Крускала, а также итоговое дерево изображены на рисунке 8. Вес получившегося дерева равен 16, что больше, чем получилось при выполнении алгоритма Прима.

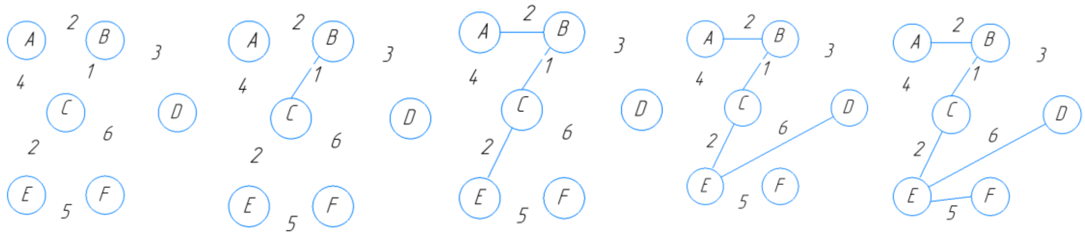

Рисунок 8

4. Заключение

Как алгоритм Прима, так и алгоритм Крускала используются для построения минимального остовного дерева в графе. Однако, алгоритм Прима начинает с одной вершины и постепенно добавляет к ней новые вершины, пока не будет построено минимальное остовное дерево. Алгоритм Крускала же начинает с отдельных ребер и объединяет группы вершин по мере добавления новых ребер.

Оба алгоритма дают одинаковый результат, если граф не содержит циклов. Однако, если в графе есть циклы, то алгоритм Прима может зациклиться, тогда как алгоритм Крускала продолжит работу, просто игнорируя циклические ребра.

Таким образом, выбор между алгоритмом Прима и Крускала зависит от конкретной задачи и свойств графа.

Алгоритм Прима лучше использовать, если граф плотный, то есть имеет много ребер. Это связано с тем, что алгоритм Прима работает быстрее на плотных графах, так как он добавляет вершины постепенно и не рассматривает все ребра.

Алгоритм Крускала лучше использовать, если граф разреженный, то есть имеет мало ребер. Это связано с тем, что алгоритм Крускала рассматривает все ребра и объединяет группы вершин, что может быть эффективнее на разреженных графах.

5.Приложение

Код программы

1) Заголовочный файл list_h

```
#ifndef list_h
#define list_h

#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

template<class T>
class Node {
public:
    T data;
    Node* next;
};

// Определяем класс списка
template<class T>
class List {
private:
    Node<T>* head;
    Node<T>* tail;

public:
    List();
    List(const List& other);
    List& operator=(const List& other);
    ~List();


    void push_tail(T value);
    void delete_head();
    void clear_list();
    Node<T>* get_head() const;
    void change_element(int pos, T value);
    T get_element(int pos) const;
    int get_size();
    void create_list(int size, T value);
};


template<class T>
List<T>::List() {
    head = nullptr;
    tail = nullptr;
}


template<class T>
List<T>::List(const List& other) {
    head = nullptr;
    tail = nullptr;
    Node<T>* current = other.head;
    while (current != nullptr) {
        push_tail(current->data);
        current = current->next;
```

```
    }
}


template<class T>
void List<T>::clear_list() {
    Node<T>* current = head;
    while (current != nullptr) {
        Node<T>* next = current->next;
        delete current;
        current = next;
    }
    head = nullptr;
    tail = nullptr;
}


template<class T>
List<T>& List<T>::operator=(const List& other) {
    if (this != &other) {
        clear_list();
        Node<T>* current = other.head;
        while (current != nullptr) {
            push_tail(current->data);
            current = current->next;
        }
    }
    return *this;
}


template<class T>
List<T>::~List() {
    clear_list();
}


template<class T>
void List<T>::push_tail(T value) {
    Node<T>* temp = new Node<T>;
    temp->data = value;
    temp->next = nullptr;

    if (head == nullptr) {
        head = temp;
        tail = temp;
    }
    else {
        tail->next = temp;
        tail = temp;
    }
}


template<class T>
void List<T>::delete_head() {
    if (head == nullptr) {
        return;
    }
    Node<T>* temp = head;
    head = head->next;
    if (head == nullptr) {
        tail = nullptr;
```

```
    }
    delete temp;
}


template<class T>
void List<T>::change_element(int pos, T value) {
    if (pos < 1) {
        return;
    }
    Node<T>* current = head;
    for (int i = 1; i < pos; i++) {
        if (current == nullptr) {
            return;
        }
        current = current->next;
    }
    current->data = value;
}


template<class T>
T List<T>::get_element(int pos) const {
    if (pos < 1) {
        return T();
    }
    Node<T>* current = head;
    for (int i = 1; i < pos; i++) {
        if (current == nullptr) {
            return T();
        }
        current = current->next;
    }
    return current->data;
}


template<class T>
int List<T>::get_size() {
    int size = 0;
    Node<T>* current = head;
    while (current != nullptr) {
        size++;
        current = current->next;
    }
    return size ;
}


template<class T>
void List<T>::create_list(int size, T value) {
    clear_list();
    for (int i = 0; i < size; i++) {
        push_tail(value);
    }
}
#endif  // list_h
```

2) Заголовочный файл graph_h

```
#ifndef graph_h
#define graph_h

#include <iostream>
#include <vector>
#include <map>
#include <algorithm>
#include "list_h.h"

class Graph {
private:
    struct Vertex {
        int data;
        List<std::pair<int, int>>* edges; // пара (dest, weight)
    };
    std::map<int, Vertex*> graph;


public:
    Graph();
    void add_vertex(int value);
    void add_edge(int v1, int v2, int weight);
    void display();
    int get_size();
    List<std::pair<int, int>>* get_list(int value); // пара (dest, weight)
    void prim_algorithm();
    void kruskal_algorithm();
};

Graph::Graph() {}

void Graph::add_vertex(int value) {
    if (graph.find(value) != graph.end()) {
        return;
    }
    Vertex* vertex = new Vertex;
    vertex->data = value;
    vertex->edges = new List<std::pair<int, int>>;
    graph[value] = vertex;
}


void Graph::add_edge(int v1, int v2, int weight) {
    if (graph.find(v1) == graph.end() || graph.find(v2) == graph.end()) {
        return;
    }
    graph[v1]->edges->push_tail({v2, weight}); // пара (dest, weight)
    graph[v2]->edges->push_tail({v1, weight}); // пара (dest, weight)
}


void Graph::display() {
    for (const auto& element : graph) {
        std::cout << element.first << " -> ";
        for (int i = 1; i <= element.second->edges->get_size(); i++) {
            std::cout << "(" << element.second->edges->get_element(i).first
<< ", " << element.second->edges->get_element(i).second << ") ";
        }
        std::cout << std::endl;
    }
}
```

```
int Graph::get_size() {
    return graph.size();
}

List<std::pair<int, int>>* Graph::get_list(int value) {
    if (graph.find(value) == graph.end()) {
        return nullptr;
    }
    return graph[value]->edges;
}


void Graph::prim_algorithm() {
    if (graph.empty()) {
        return;
    }
    int startVertex = graph.begin()->first;
    List<int> parent;
    parent.create_list(get_size(), -1);
    List<bool> visited;
    visited.create_list(get_size(), false);
    List<int> distance;
    distance.create_list(get_size(), INT_MAX);

    parent.change_element(startVertex, -1);
    distance.change_element(startVertex, 0);

    for (int i = 0; i < get_size(); i++) {
        int currentVertex = -1;
        int minDistance = INT_MAX;
        for (const auto& element : graph) {
            int v = element.first;
            if (!visited.get_element(v) && distance.get_element(v) <
minDistance) {
                currentVertex = v;
                minDistance = distance.get_element(v);
            }
        }

        if (currentVertex == -1) {
            break;
        }

        visited.change_element(currentVertex, true);

        List<std::pair<int, int>>* edges = get_list(currentVertex);
        for (int j = 1; j <= edges->get_size(); j++) {
            int v = edges->get_element(j).first;
            int weight = edges->get_element(j).second;
            if (!visited.get_element(v) && weight < distance.get_element(v))
{
                parent.change_element(v, currentVertex);
                distance.change_element(v, weight);
            }
        }
    }


    for (const auto& element : graph) {
        int v = element.first;
        int p = parent.get_element(v);
        if (p != -1) {
```

```
            std::cout << v << " - " << p << "\tweight: " <<
distance.get_element(v) << std::endl;
        }
    }
}


void Graph::kruskal_algorithm() {
    if (graph.empty()) {
        return;
    }

    std::map<int, int> disjoint;
    for (const auto& element : graph) {
        disjoint[element.first] = element.first;
    }

    // create sorted edges list
    std::vector<std::pair<int, std::pair<int, int>>> edges; // пара (weight,
(source, dest))
    for (const auto& element : graph) {
        int source = element.first;
        for (int i = 1; i <= element.second->edges->get_size(); i++) {
            int dest = element.second->edges->get_element(i).first;
            int weight = element.second->edges->get_element(i).second;
            if (source < dest) {
                edges.push_back({ weight, { source, dest } });
            }
        }
    }
    std::sort(edges.begin(), edges.end());

    // find MST using heap
    Graph mst;
    std::make_heap(edges.begin(), edges.end(), [](std::pair<int,
std::pair<int, int>> a, std::pair<int, std::pair<int, int>> b) {
        return a.first > b.first;
    });
    while (mst.get_size() < graph.size() - 1 && !edges.empty()) {
        std::pop_heap(edges.begin(), edges.end(), [](std::pair<int,
std::pair<int, int>> a, std::pair<int, std::pair<int, int>> b) {
            return a.first > b.first;
        });
        std::pair<int, std::pair<int, int>> edge = edges.back();
        int weight = edge.first;
        int source = edge.second.first;
        int dest = edge.second.second;
        edges.pop_back();

        if (disjoint[source] != disjoint[dest]) {

            mst.add_vertex(source);
            mst.add_vertex(dest);
            mst.add_edge(source, dest, weight);


            int destination = disjoint[dest];
            int sourc = disjoint[source];
            for (auto& s : disjoint) {
                if (s.second == destination) {
                    s.second = sourc;
                }
            }
```

```
        }
    }

    // display MST
    mst.display();
}

#endif  // graph_h
```

3) main.cpp

```
#include "graph.h"


int main() {
    Graph g;
    for (int i = 1; i <= 10; i++) {
        g.add_vertex(i);
    }
    for (int i = 1; i <= 10; i++) {
        for (int j = i + 1; j <= 10; j++) {
            g.add_edge(i, j, rand());
        }
    }

    std::cout << "Graph:" << std::endl;
    g.display();

    std::cout << "Minimum spanning tree using Prim's algorithm:" <<
std::endl;
    clock_t begin = clock();
    g.prim_algorithm();
    std::cout << "Minimum spanning tree using Kruskal's algorithm:" <<
std::endl;
    g.kruskal_algorithm();
    clock_t end = clock();
    double time_spent = (end - begin);
    printf("The elapsed time is %f seconds", time_spent);
    cout <<endl;

    return 0;
}
```

Список литературы

1. Наймарк А.Д. Алгоритмы и структуры данных: учебник. М.

2. Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн, "Алгоритмы: построение и анализ". 2-е издание. ООО "Издательский дом "Вильямс"", 2010.

3. Кормен, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд. "Алгоритмы: построение и анализ". 2005. С. 554-558.

4. "Prim's Minimum Spanning Tree (MST) - Greedy Algorithm”. GeeksforGeeks, https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/.

5. "Kruskal's Minimum Spanning Tree Algorithm - Greedy Algo-2”. GeeksforGeeks, https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/.

6. Романенко А. А., Камышанский В. Б. Алгоритмы вычислительной математики: Курс лекций. — Ярославль: ЯрГУ, 2015. — С. 58.