Санкт-Петербургский государственный университет телекоммуникаций

им. проф. М.А. Бонч-Бруевича

**Алгоритмы и структуры данных**

Отчёт по лабораторной работе №4  
«Алгоритмы поиска минимального остовного дерева**»**

Группа: ИКПИ-33

Студенты: Семенихин А. Р.

Коньков М. Д.

Санкт-Петербург

2025

# Цель работы

Целью работы является ознакомление с алгоритмами поиска минимального остовного дерева и методикой оценки эффективности алгоритмов.

# Описание алгоритмов

Алгоритм Прима в идее и реализации очень похож на алгоритм Дейкстры. Как и в алгоритме Дейкстры, мы поддерживаем уже обработанную часть графа (минимального остовного дерева), и постепенно её расширяем за счёт ближайших вершин.

Утверждается, что если разделить вершины графа на два множества (обработанные и необработанные), первое из которых составляет связную часть минимального остовного дерева, то ребро минимальной длины, связывающее эти два множества гарантированно будет входить в минимальное остовное дерево.

Таким образом, для нахождения минимального остовного дерева начнём с произвольной вершины и будем постепенно добавлять ближайшие к уже имеющимся.

Сложность:

Алгоритм Краскала также достаточно прост в своей идее и реализации. Он заключается в сортировке всех рёбер в порядке возрастания длины, и поочерёдному добавлению их в минимальный остов, если они соединяют различные компоненты связности.

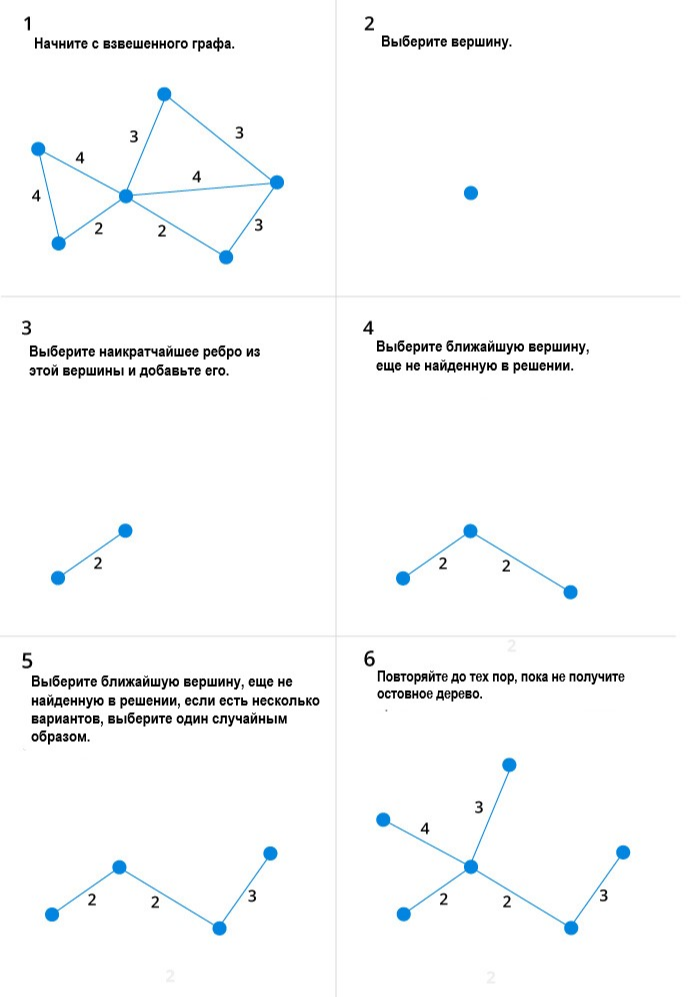
Более формально: пусть мы уже нашли некоторые рёбра, входящие в минимальный остов. Утверждается, что среди всех рёбер, соединяющих различные компоненты связности, в минимальный остов будет входить ребро с минимальной длиной.

Для реализации алгоритма Краскала необходимо уметь сортировать рёбра по возрастанию длины (для этого воспользуемся собственным типом данных) и проверять, соединяет ли ребро две различных компоненты связности.

Сложность:

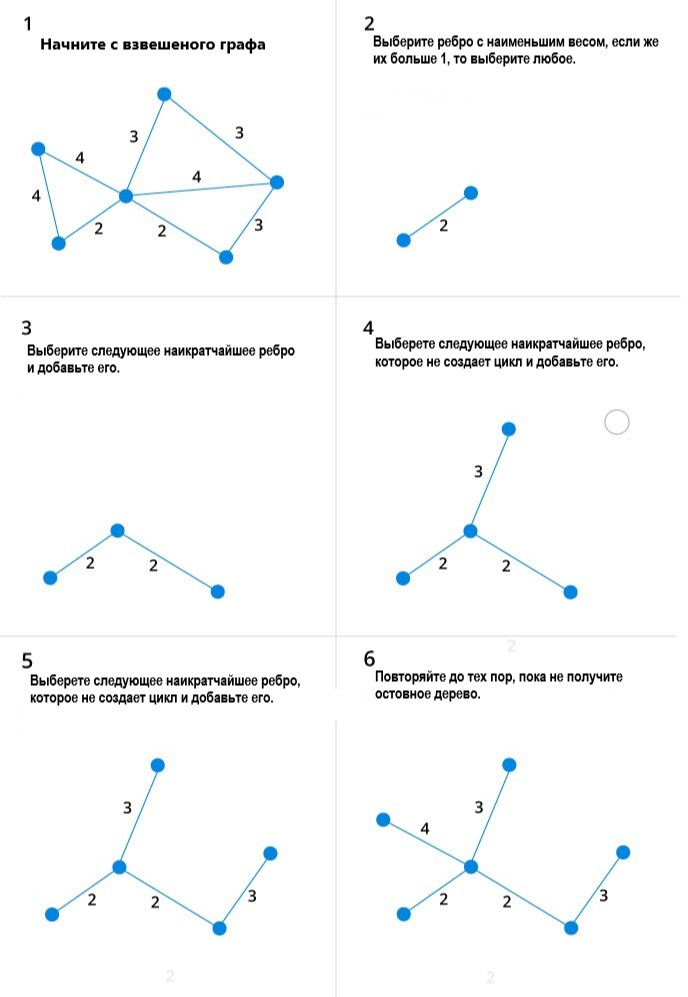
# Пошаговое сравнение

В таблице 1 отображена пошаговая работа алгоритма Прима.

*Таблица 1. Пошаговая работа алгоритма Прима.*

В таблице 2 отображена пошаговая работа алгоритма Краскала.

*Таблица 2. Пошаговая работа алгоритма Краскала.*



По результатам пошагового сравнения, алгоритм Прима и алгоритм Краскала на графе из 6 вершин выполняются за одинаковое число шагов.

# Описание программы

Программа для выполнения настоящей работы обеспечивает выполнение процедур поиска подстроки в фоновом режиме и подсчёт временных затрат на работу каждого из алгоритмов. Имеется возможность указать число итераций, в течение которых будут сняты показания времени.

При запуске программы следует ввести длину строки и количество итераций согласно инструкциям. По завершении работы процедур построения минимального остовного дерева на экран будет выведено среднее время работы обоих алгоритмов. Для завершения работы программы следует ввести символ q, для продолжения работы — любой другой символ.

Программа создана с использованием языка программирования C++ в среде программирования Visual Studio Code для работы в системе под управлением ОС Windows.

# Полученные результаты

Результаты оценки временной сложности алгоритмов (в микросекундах) представлены в таблице 3 и отображены на рисунке 1. Оценки произведены на разреженных графах.

*Таблица 3. Зависимость времени работы алгоритмов одномерной оптимизации от заданной точности вычислений (в микросекундах).*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Точность / Алгоритм | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| Алгоритм Прима | 13,70 | 24,44 | 32,10 | 45,83 | 53,00 | 62,40 | 66,38 | 77,60 | 84,30 | 92,50 |
| Алгоритм Краскала | 1,73 | 2,75 | 5,50 | 5,36 | 6,00 | 9,20 | 10,30 | 10,60 | 12,10 | 14,70 |

*Рис. 1. Зависимость времени работы алгоритмов построения минимального остовного дерева от количества вершин в графе.*

Данные, отображенные в таблице и на рисунке, показывают, что на разреженных графах алгоритм Краскала работает в среднем быстрее, чем алгоритм Прима.

# Выводы

В ходе проведённой лабораторной работы получены следующие результаты:

1. Изучены алгоритмы построения минимального остовного дерева Прима и Краскала.
2. Создана программа, позволяющая выполнить оценку временной сложности алгоритмов.
3. Проведена оценка временной сложности алгоритмов Прима и Краскала на разреженных графах.
4. По результатам оценки, на разреженных графах алгоритм Краскала работает в среднем быстрее, чем алгоритм Прима.

# Исходный код программы

#include <iostream>

#include <vector>

#include <time.h>

#include <queue>

#include <algorithm>

#include <cstdlib>

#define MAX\_W 100

#define C 100000

using namespace std;

struct edge

{

int src, dest, w;

};

bool cmp(edge &a, edge &b)

{

return a.w < b.w;

}

vector<vector<int>> genGraph(int n, int v)

{

vector <edge> graph;

vector <vector<int>> matrixGraph(n, vector<int>(n, -1));

vector <bool> used(n, false);

queue<int> next;

next.push(0);

int unused\_n = n - 1;

int unused\_v = v;

while (unused\_n > 0)

{

int amountOfV = rand() % (v / n) + 1;

int cur\_src = next.front();

next.pop();

for (int i = 0; i < amountOfV && unused\_n > 0; i++)

{

int w = rand() % MAX\_W + 1;

matrixGraph[cur\_src][unused\_n] = w;

matrixGraph[unused\_n][cur\_src] = w;

next.push(unused\_n);

unused\_n--;

unused\_v--;

}

}

while (unused\_v > 0)

{

int src = rand() % n;

int dest = rand() % n;

if (src == dest || matrixGraph[src][dest] != -1)

{

continue;

}

int w = rand() % MAX\_W + 1;

matrixGraph[src][dest] = w;

matrixGraph[dest][src] = w;

unused\_v--;

}

return matrixGraph;

}

vector<edge> matGraphConv(vector<vector<int>> matGraph, int n)

{

vector<edge> graph;

for (int i = 0; i < n; i++)

{

for (int j = i+1; j < n; j++)

{

edge cur;

if (matGraph[i][j] != -1)

{

cur.src = i;

cur.dest = j;

cur.w = matGraph[i][j];

graph.push\_back(cur);

}

}

}

return graph;

}

vector <int> leader;

int getLeader(int x)

{

if (x == leader[x])

{

return x;

}

return leader[x] = getLeader(leader[x]);

}

bool unite(int x, int y)

{

x = getLeader(x);

y = getLeader(y);

if (x == y)

{

return false;

}

if (rand() % 2 == 0)

{

swap(x, y);

}

leader[x] = y;

return true;

}

void kruskal(vector<edge> graph, int n, int m)

{

sort(graph.begin(), graph.end(), cmp);

leader.resize(n);

for (int i = 0; i < n; i++)

{

leader[i] = i;

}

for (int i = 0; i < m; i++)

{

int x = graph[i].dest, y = graph[i].src;

}

return;

}

void prim(vector<vector<int>> graph, int n)

{

int no\_edge;

std::vector<bool> selected(n);

selected[0] = true;

int x, y;

while (no\_edge < n - 1)

{

int min = INT\_MAX;

x = 0;

y = 0;

for (int i = 0; i < n; i++)

{

if (selected[i])

{

for (int j = 0; j < n; j++)

{

if (!selected[j] && graph[i][j])

{

if (min > graph[i][j] && graph[i][j] != 0)

{

min = graph[i][j];

x = i;

y = j;

}

}

}

}

}

selected[y] = true;

no\_edge++;

}

}

int main()

{

srand(static\_cast<unsigned int>(time(NULL)));

char c = ' ';

while (c != 'q')

{

int r = 0;

std::cout << "Iterations:" << std::endl;

std::cin >> r;

int n = 0;

std::cout << "Vertex count:" << std::endl;

std::cin >> n;

int m = 0;

std::cout << "Edges count:" << std::endl;

std::cin >> m;

double time\_f1\_sum = 0, time\_f2\_sum = 0;

for (int k = 0; k < r; k++)

{

vector<vector<int>> matGraph = genGraph(n, m);

std::cout << "Graph gen 1" << std::endl;

clock\_t start = clock();

prim(matGraph, n);

clock\_t end = clock();

double seconds = (double)(end - start) / CLOCKS\_PER\_SEC;

time\_f1\_sum += seconds;

matGraph = genGraph(n, m);

vector<edge> graph = matGraphConv(matGraph, n);

std::cout << "Graph gen 2" << std::endl;

start = clock();

kruskal(graph, n, m);

end = clock();

seconds = (double)(end - start) / CLOCKS\_PER\_SEC;

time\_f2\_sum += seconds;

}

double time\_1 = time\_f1\_sum / (double)r;

std::cout << "The time (prim): " << time\_1 << " seconds" << std::endl;

double time\_2 = time\_f2\_sum / (double)r;

std::cout << "The time (kruskal): " << time\_2 << " seconds" << std::endl;

std::cout << "Enter any char to continue; enter 'q' to exit." << std::endl;

std::cin >> c;

}

return 0;

}