МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКОГО ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра АПУ

ОТЧЕТ

по лабораторной работе № 4

по дисциплине «Алгоритмы и структуры данных»

Тема: Построение минимального остовного дерева

Студентка гр. 1361	________________	Горбунова Д. А.
Студентка гр. 1361	________________	Токарева У. В.
Преподаватель	________________	Беляев А. В.

Санкт-Петербург

2022

Цель работы: ознакомление с вариантами реализации алгоритмов на графах на примере задачи построения минимального остовного дерева.

Теоретическая часть

Пусть дан взвешенный (ребрам присвоены веса) связный неориентированный граф, содержащий циклы, т.е. замкнутые маршруты. Остовным (покрывающим) деревом называется его подграф, не содержащий циклов, и при этом включающий все вершины исходного графа. Минимальным остовным деревом называется такое остовное дерево, для которого сумма весов ребер минимальна.

Два наиболее распространенных алгоритма построения минимального остовного дерева:

- Алгоритм Прима (Роберт Прим, 1957 г.)
- Алгоритм Краскала (Джозеф Краскал, 1956 г.)

Алгоритм Прима

Алгоритм начинается с выбора произвольной вершины. Она принимается за часть построенного минимального остовного дерева.

Далее в цикле в каждой итерации рассматриваются только те ребра исходного графа, одна из вершин, которых строго принадлежит уже построенной части, а другая строго не принадлежит (если второе условие не проверять, то в графе возникнут циклы). Из списка всех ребер, удовлетворяющих этому условию, выбирается ребро с наименьшим весом и добавляется к построенной части.

Итерации повторяются до тех пор, пока все вершины не окажутся включенными в остовное дерево.

Алгоритм Краскала

Алгоритм начинается с того, что каждая вершина исходного графа помещается в свое множество (состоящее из одной вершины) – компоненту связности.

Далее в цикле в каждой итерации из всех ребер, которые соединяют разные компоненты связности, выбирается ребро с наименьшим весом, и с помощью него две отдельные компоненты связности объединяются в одну.

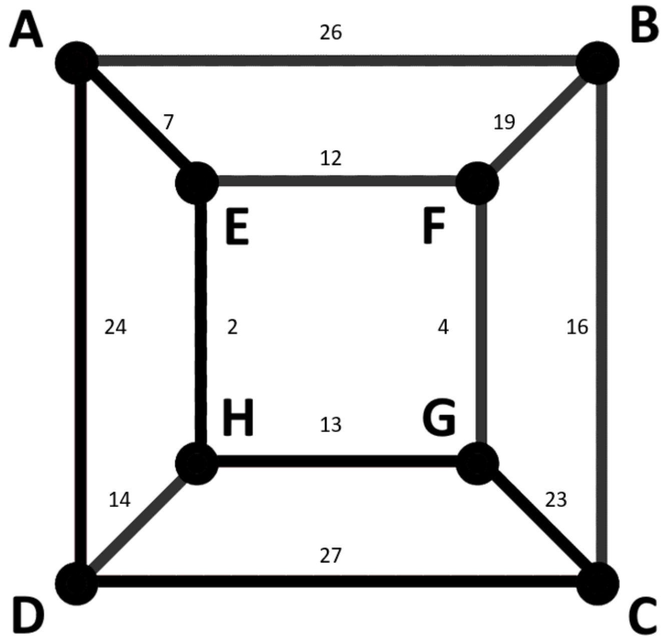
Итерации повторяются до тех пор, пока количество компонент связности не уменьшится до одной – она и будет представлять собой остовное дерево.

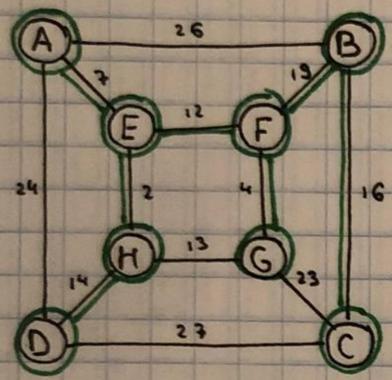
Оценка быстродействия алгоритмов

В случае «наивной» реализации (хранение информации о наличии и стоимости ребер в двумерном массиве) асимптотика обоих алгоритмов – O(n2). В случае применения более сложных структур для хранения сведений об упорядоченном списке ребер ¬– O(n·log2n).

ИДЗ

Горбунова Дарья
Вариант 5

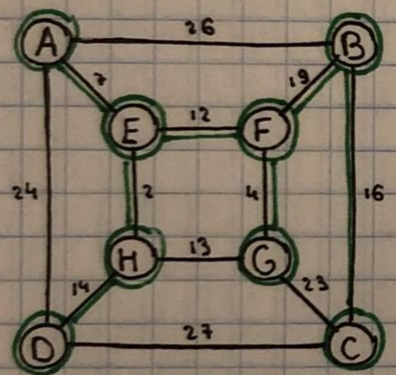




Алгоритм Прима

start: A

v	e	$\sum w(e)$
A	-	0
E	AE	7
H	EH	9
F	EF	21
G	FG	25
D	HD	39
B	FB	58
C	BC	74



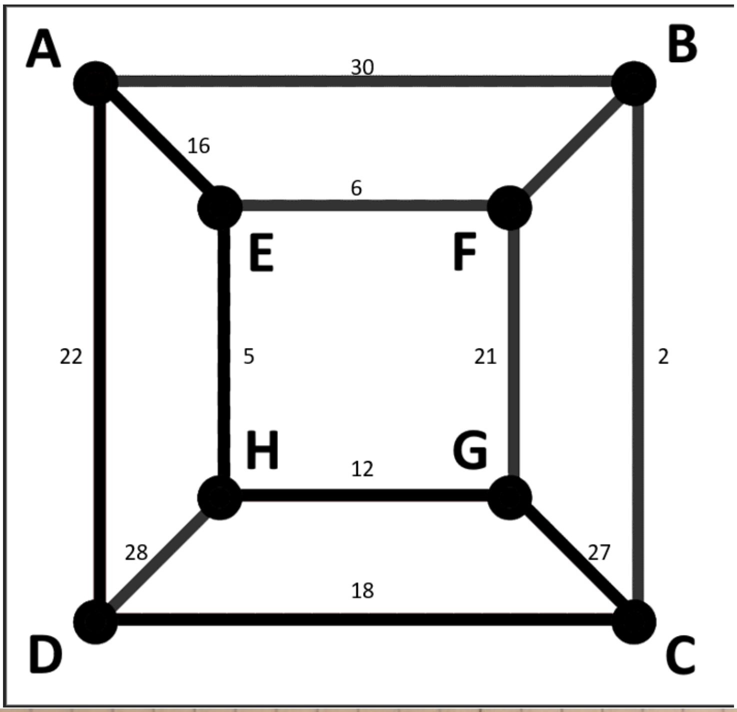
Алгоритм Краскала

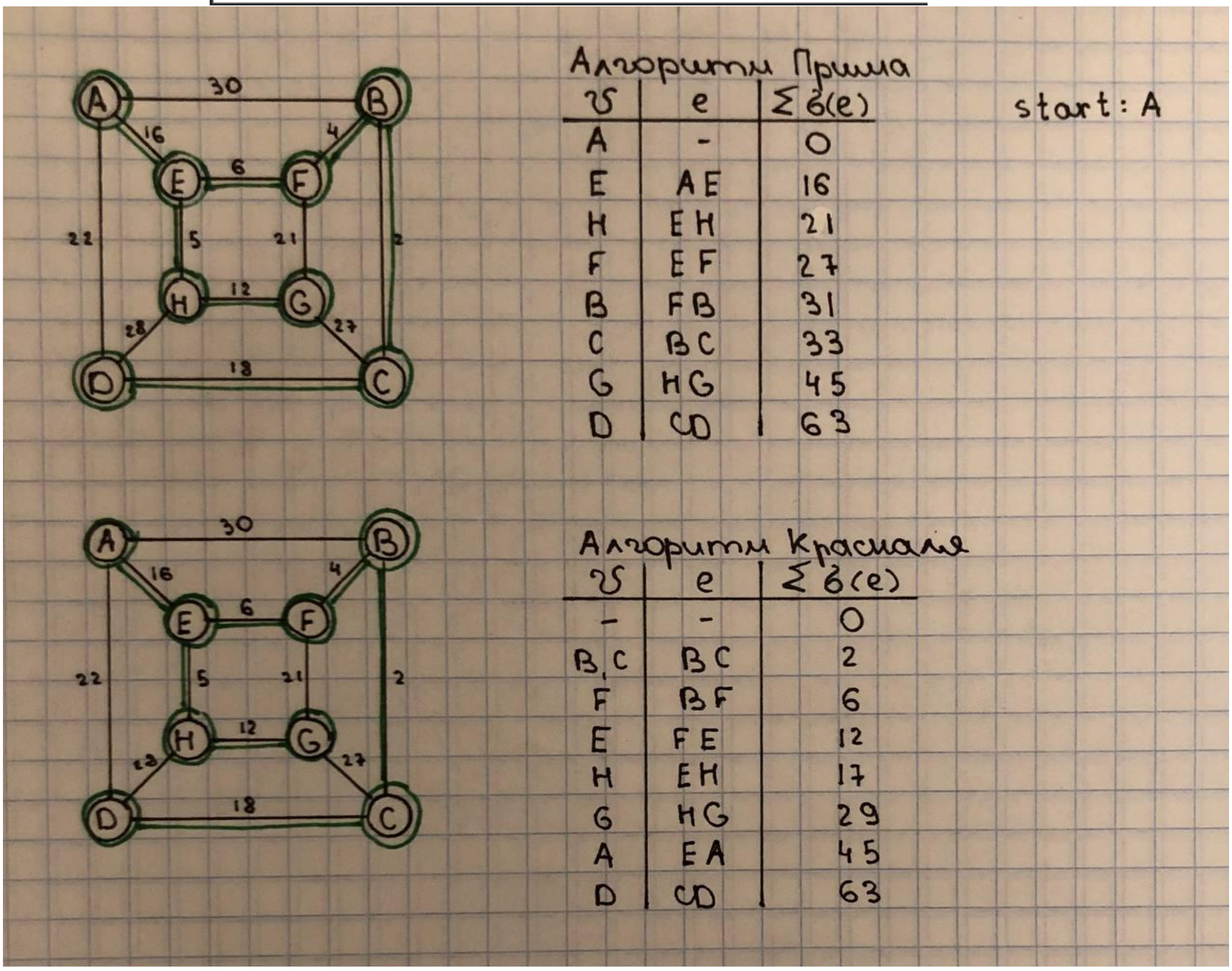
v	e	$\sum w(e)$
-	-	0
E, H	EH	2
A	EA	9
F	EF	21
G	FG	25
D	HD	39
B	FB	58
C	BC	74

Вывод: Сумма ребер и полученные графы полностью идентичны.

Токарева Ульяна

Вариант 17





Вывод: Сумма ребер и полученные графы полностью идентичны.

Результаты работы программы.

Edges	Weight	Number
0 -- 1:	44	1
0 -- 4:	36	2
0 -- 5:	40	3
0 -- 8:	19	4
0 -- 11:	25	5
0 -- 12:	17	6
0 -- 14:	46	7
1 -- 2:	200	8
1 -- 6:	16	9
1 -- 9:	21	10
1 -- 10:	11	11
2 -- 3:	200	12
2 -- 6:	31	13
2 -- 7:	21	14
2 -- 9:	50	15
2 -- 12:	23	16
3 -- 4:	200	17
3 -- 10:	32	18
3 -- 11:	15	19
4 -- 5:	200	20
4 -- 6:	25	21
4 -- 9:	33	22
5 -- 6:	5	23
5 -- 12:	22	24
5 -- 13:	16	25
6 -- 7:	200	26
6 -- 10:	48	27
6 -- 11:	20	28
6 -- 12:	27	29
6 -- 13:	8	30
7 -- 8:	200	31
7 -- 9:	43	32
7 -- 13:	35	33
8 -- 9:	200	34
9 -- 10:	45	35
9 -- 13:	40	36
9 -- 14:	12	37
10 -- 11:	30	38
10 -- 12:	40	39
11 -- 12:	200	40
12 -- 13:	200	41
13 -- 14:	200	42

Рисунок 3 – Вывод исходного графа.

My Tree

Edges	Weight	Index
1--10	11	0
1--6	16	1
0--12	17	2
0--8	19	3
1--9	21	4
2--7	21	5
0--11	25	6
2--6	31	7
0--4	36	8
0--5	40	9
0--1	44	10
0--14	46	11
2--3	200	13

Summary Weigth:777

Рисунок 4 – Минимальное остоновое дерево методом Краскаля

Исходный код программы

```
#include <iostream>

#include <time.h>

#include <vector>

#include <algorithm>


using namespace std;


struct Edge{

    int vertex_begin;

    int vertex_end;

    int weight;

};

void Print_Edges (int ** graph, int flag){

    cout << "Edges \t Weight \t Number"<<  endl;

    int count = 1;

    for (int v1=0; v1 < flag; v1++)

        for (int v2= v1 + 1; v2 < flag; v2++)

            if (graph[v1][v2] > 0)

                cout << v1 << " -- " << v2 << ": \t" << graph[v1][v2] << "
\t" << count++<< endl;

    return;

}

int Generator_graph(int **graph, int flag){

    int Curr_time = time(NULL);

    if (Curr_time == -1 ){

        cout << "Error time!"<< endl;

        return 1;

    }

    srand (Curr_time);

    int count_edges=0;

    for (int v1=0; v1 < flag-1; v1++){

        for (int v2= v1 + 1; v2 < flag; v2++){

            if ((v1 != v2 && graph[v1][v2] == 0 ) && rand() % 100 < 34 ) {

                graph[v1][v2] = graph[v2][v1]= rand() % 50 + 1 ;//генерация
веса ребра
```

```
                count_edges++;
            }
            else graph[v1][v2] = graph[v2][v1] = -1;
        }
    }
    int critical_weight=200;
    for (int v1=0; v1<flag-1; v1++){
        int v2=v1+1;
        if (graph[v1][v2] == -1){
            graph[v1][v2] = graph[v2][v1]= critical_weight; //создание доп
ребер для гарантии графа
            count_edges++;
        }}
    //cout << "Number of undirected edges:  " << count_edges<< endl<< endl;
    return count_edges;
}
void Sort_Edges (int ** graph, Edge* edges, int count, int flag ) {
    int k = 0;
    for (int v1 = 0; v1 < flag; v1++)
        for (int v2 = v1 + 1; v2 < flag; v2++)
            if (graph[v1][v2] > 0) {
                edges[k].weight = graph[v1][v2];
                edges[k].vertex_begin = v1;
                edges[k++].vertex_end = v2;
            }
    for (int i = 0; i < flag; i++)
        for (int j = 0; j < flag; j++)
            if (edges[i].weight < edges[j].weight)
                swap(edges[i], edges[j]);
}
int Alg_Kruskal( Edge* edges, int flag){
    int* connectivity_component = new int[flag];// компонента связности
    for (int i=0; i<flag; i++) connectivity_component[i]=i;
    int currSumEdges=0;
    cout << "My Tree\nEdges \tWeight \tIndex\n";
    for (int i=0; i< flag; i++){
```

```
        Edge currEdge = edges [i];

        if
(connectivity_component[currEdge.vertex_begin]!=connectivity_component[currEd
ge.vertex_end]){

            int past_connectivity_component =
connectivity_component[currEdge.vertex_begin];

            int new_connectivity_component =
connectivity_component[currEdge.vertex_end];

            cout << currEdge.vertex_begin<<"--"<< currEdge.vertex_end<<"
\t"<< currEdge.weight<< " \t"<< i<<endl;

        for (int j=0;j<flag; j++){

            if (connectivity_component[j] == past_connectivity_component)

                connectivity_component[j]=new_connectivity_component;

        }  }

        currSumEdges+=currEdge.weight;

    }

    return currSumEdges;}


int main(){

    int n;

    cout << "Enter the number of verties:";

    cin >> n;

    int** graph = new int*[n];

    for(int v1 = 0; v1<n; v1++){

        graph[v1] = new int[n];

    }

    for (int v1=0; v1<n; v1++)

        for (int v2=0; v2<n; v2++)

            graph[v1][v2]=0;

    int count_edges;

    count_edges=Generator_graph(graph, n);

    Edge* currEdge = new Edge [count_edges];

    cout << endl<< endl;

    Print_Edges(graph,n);

    cout <<endl<< endl;

    Sort_Edges(graph, currEdge, count_edges, n);

    cout << endl << endl;
```

```
   int sum_weig = Alg_Kruskal(currEdge,n);
   cout << "Summary Weigth:"<<sum_weig<<endl<< endl<< endl;


    delete(graph);
    delete (currEdge);
    return 0;
}
```

ВЫВОД

В ходе выполнения данной лабораторной работы был изучен рекурсивный алгоритм обхода двоичного дерева поиска. Программа, выполняющая построение двоичного дерева поиска, была дополнена алгоритмом обхода дерева.