МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ ПОЛІТЕХНІЧНИЙ УНІВЕРСИТЕТ
ІНСТИТУТ КОМП'ЮТЕРНИХ СИСТЕМ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ

Розрахунково Графічна Робота

З дисципліни: «Теорія алгоритмів»

На тему: «Алгоритм аналізу обчислювальної складності алгоритмів Прима та Крускала на випадкових графах з різними розмірами і густиною»

Виконала:
Студентка групи АІ – 212
Козуб К.О
Перевірила:
Арсірій О.О.

Одеса 2022

Завдання: Алгоритм аналізу обчислювальної складності алгоритмів Пріма і Крускала на випадкових графах з різними розмірами та щільністю.

Мета: Розробка алгоритму аналізу обчислювальної складності алгоритмів Прима та Крускала на випадкових графах з різними розмірами та щільностями для обґрунтованого вибору найкращого за заданих вхідних даних.

Розробка загальної схеми алгоритму:

Алгоритм Пріма:

Алгоритм Прима — жадібний алгоритм побудови мінімального кістякового дерева зваженого зв'язного неорієнтованого графу. На вхід алгоритму подається зв'язковий неорієнтований граф. Для кожного ребра задається його вартість. Спочатку береться довільна вершина і знаходиться ребро, інцидентне цій вершині і що має найменшу вартість. Знайдене ребро і дві вершини, що з'єднуються ним, утворюють дерево. Потім розглядаються ребра графа, один кінець яких — вершина, що вже належить дереву, а інший — ні; з цих ребер вибирається ребро найменшої вартості. Ребро, що вибирається на кожному кроці, приєднується до дерева. Зростання дерева відбувається доти, доки вичерпані всі вершини вихідного графа. Алгоритм завершує роботу після того, як всі вершини опиняються включені в дерево, що будується. Оскільки алгоритм розширює дерево по одній вершині за ітерацію, загальна кількість таких ітерацій дорівнює n-1, де n-кількість вершин графа. Жадібний алгоритм-алгоритм, що полягає у прийнятті локально оптимальних рішень на кожному етапі, припускаючи, що кінцеве рішення також виявиться оптимальним.

Алгоритм Prim (G)
// Входные данные: Взвешенный связный граф G= (V, E)

// Выходные данные: ET, множество ребер, составляющих минимальное

// остовное дерево G

VT←{v0}

ET←∅

fori←1 to|V|-1 do

Поиск ребра с минимальным весом e* = (v*,u*) среди всех

ребер (v, u) таких, что v∈VTи u∈V-VT

VT←VT∪{u*}

ET←ET∪{e*}

Return ET

Алгоритм Крускала:

Алгоритм Краскала – ефективний алгоритм побудови мінімального острівного дерева зваженого зв'язкового неорієнтованого графа. Алгоритм було вперше описано Джозефом Крускалом 1956 року. Візьмемо зважений зв'язний граф G=(V, E), де V — множина вершин, E — множина ребер, для кожного з яких задано вагу. Тоді ациклічна множина ребер, що поєднують усі вершини графу і чия загальна вага мінімальна, називається мінімальним кістяковим деревом. Алгоритм Крускала починається з побудови виродженого лісу, що містить V дерев, кожне з яких складається з однієї вершини. Далі виконуються операції об'єднання двох дерев, для чого використовуються найкоротші можливі ребра, поки не утвориться єдине дерево. Це дерево і буде мінімальним кістяковим деревом.

Алгоритм Kruskal(G)

// Входные данные: Взвешенный связный граф G= (V, E)

// Выходные данные: ET, множество ребер,составляющих минимальное

// остовное дерево G

Сортировка множества Ев неубывающем порядке весов ребер

$ei0*\leq\cdots ei|E|-1*$

ET←∅

ecounter←0

k←0

whileecounter< |V| -1do

k=k+1

ifET∪{eik}–ациклический граф

ET← ET∪{eik};

ecounter=ecounter+1

return ET

Опис програмної реалізації алгоритму:

Алгоритм Прима:

```
#include <iostream>
#include <cstring>
#include <ctime>
using namespace std;
#define INF 9999999
#define V //Количество вершин в графе
int G[V][V] = {

    //Матрица смежности в виде двумерного массива

};
int main()
{
    unsigned int start_time = clock();//начало выполнение алгоритма
    int t = 0;
    while (t <= 1000)
    {
        int no_edge;
        int selected[V];
        memset(selected, false, sizeof(selected));
        no_edge = 0;
        selected[0] = true;
        int x;
        int y;
        cout << "Edge" << "  : " << "Weight" << endl;
        while (no_edge < V - 1) {
            int min = INF;
            x = 0;
            y = 0;
            for (int i = 0; i < V; i++) {
                if (selected[i]) {
                    for (int j = 0; j < V; j++) {
                        if (!selected[j] && G[i][j]) {
                            if (min > G[i][j]) {
                                min = G[i][j];
                                x = i;
                                y = j;
                            }
                        }
                    }
                }
            }
            cout << x << " - " << y << " :  " << G[x][y] << endl;
            selected[y] = true;
            no_edge++;
        }
        t++;
    }
```

```
    unsigned int end_time = clock(); ();//конец выполнение алгоритма
    cout << "runtime = " << (end_time - start_time) / 1000.0 << endl;//вывод времени
работы алгоритма
    return 0;
}
```

Алгоритм Крускала:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>
using namespace std;
#define edge pair<int,int>
class Graph {
private:
	vector<pair<int, edge>> G; // graph
	vector<pair<int, edge>> T; // mst
#define edge pair<int,int>
	int* parent;
	int V; // number of vertices/nodes in graph
public:
	Graph(int V);
	void AddWeightedEdge(int u, int v, int w);
	int find_set(int i);
	void union_set(int u, int v);
	void kruskal();
	void print();
};
Graph::Graph(int V) {
	parent = new int[V];
	for (int i = 0; i < V; i++)
		parent[i] = i;
	G.clear();
	T.clear();
}
void Graph::AddWeightedEdge(int u, int v, int w) {
	G.push_back(make_pair(w, edge(u, v)));
}

int Graph::find_set(int i) {
	// If i is the parent of itself
	if (i == parent[i])
		return i;
	else
		// Else if i is not the parent of itself
		// Then i is not the representative of his set,
		// so we recursively call Find on its parent
		return find_set(parent[i]);
}
void Graph::union_set(int u, int v) {
	parent[u] = parent[v];
}

void Graph::kruskal() {
	int i, uRep, vRep;
	sort(G.begin(), G.end()); // increasing weight
	for (i = 0; i < G.size(); i++) {
		uRep = find_set(G[i].second.first);
		vRep = find_set(G[i].second.second);
		if (uRep != vRep) {
			T.push_back(G[i]); // add to tree
			union_set(uRep, vRep);
		}
```

```
	}
}
void Graph::print() {
	cout << "Edge  :" << " Weight" << endl;
	for (int i = 0; i < T.size(); i++) {
		cout << T[i].second.first << " - " << T[i].second.second << " : " <<
T[i].first << endl;
	}
}
int main() {
	unsigned int start_time = clock();//начало выполнение алгоритма
	int t = 0;
	while (t <= 1000)
	{
		Graph g();//Количество вершин в графе

		//добавление ребер графа, их веса и вершин, инцидентными ребру

		g.kruskal();
		g.print();
		t++;
	}
	unsigned int end_time = clock();//конец выполнение алгоритма
	cout << "runtime = " << (end_time - start_time) / 1000.0 << endl;//вывод времени
работы алгоритма
	return 0;
}
```

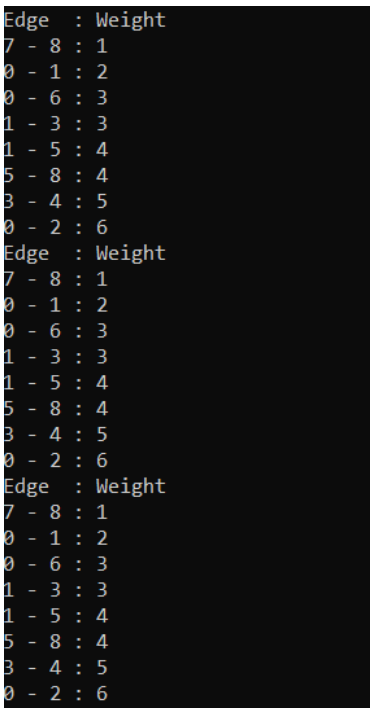
Візуалізація покрокового виконання завдання для підтвердження коректності роботи програми:

Алгоритм Прима:

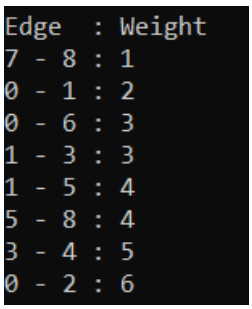
1.Ми вказуємо кількість вершин і заповнюємо таблицю суміжності (двовимірний масив). При заповненні масиву там, де таблиці суміжності були 0 ми замінюємо на INF.

```
int G[V][V] = {
{INF, 2, 6, 8, INF, INF, 3, INF, INF},
{2, INF, 9, 3, INF, 4, 9, INF, INF},
{6, 9, INF, 7, INF, INF, INF, INF, INF},
{8, 3, 7, INF, 5, 5, INF, INF, INF},
{INF,INF,INF, 5, INF, INF, 8, 9, INF},
{INF, 4, INF, 5, INF, INF, INF, 6, 4},
{3, 9, INF, INF, 8, INF, INF, INF,INF},
{INF, INF, INF, INF, 9, 6, INF, INF, 1},
{INF, INF, INF, INF, INF, 4, INF, 1, INF},
};
```

2.Запускаємо програму, вона робить даний алгоритм 1000 разів з виведенням мінімального кістякового дерева при кожній ітерації.



3.Після всіх 1000 повторень алгоритму Пріма, отримуємо час виконання всіх повторень.



Алгоритм Крускала:

1. Ми вказуємо кількість вершин і додаємо ребра графа, їх ваги та вершини, інцидентні цим ребрам.

```
g.AddWeightedEdge(u: 0, v: 1, w: 2);
g.AddWeightedEdge(u: 0, v: 2, w: 6);
g.AddWeightedEdge(u: 0, v: 6, w: 3);
g.AddWeightedEdge(u: 0, v: 3, w: 8);
g.AddWeightedEdge(u: 1, v: 0, w: 2);
g.AddWeightedEdge(u: 1, v: 2, w: 9);
g.AddWeightedEdge(u: 1, v: 3, w: 3);
g.AddWeightedEdge(u: 1, v: 6, w: 9);
g.AddWeightedEdge(u: 1, v: 5, w: 4);
g.AddWeightedEdge(u: 2, v: 0, w: 6);
g.AddWeightedEdge(u: 2, v: 1, w: 9);
g.AddWeightedEdge(u: 2, v: 3, w: 7);
g.AddWeightedEdge(u: 3, v: 0, w: 8);
g.AddWeightedEdge(u: 3, v: 1, w: 3);
g.AddWeightedEdge(u: 3, v: 4, w: 5);
g.AddWeightedEdge(u: 3, v: 5, w: 5);
g.AddWeightedEdge(u: 4, v: 3, w: 5);
g.AddWeightedEdge(u: 4, v: 6, w: 8);
g.AddWeightedEdge(u: 4, v: 7, w: 9);
g.AddWeightedEdge(u: 5, v: 1, w: 4);
g.AddWeightedEdge(u: 5, v: 3, w: 5);
g.AddWeightedEdge(u: 5, v: 7, w: 6);
g.AddWeightedEdge(u: 5, v: 8, w: 4);
g.AddWeightedEdge(u: 6, v: 0, w: 3);
g.AddWeightedEdge(u: 6, v: 1, w: 9);
g.AddWeightedEdge(u: 6, v: 4, w: 8);
g.AddWeightedEdge(u: 7, v: 4, w: 9);
g.AddWeightedEdge(u: 7, v: 5, w: 6);
g.AddWeightedEdge(u: 7, v: 8, w: 1);
g.AddWeightedEdge(u: 8, v: 7, w: 1);
g.AddWeightedEdge(u: 8, v: 5, w: 4);
g.kruskal();
g.print();
```

2. Запускаємо програму, вона робить цей алгоритм 1000 разів з виведенням мінімального кістякового дерева при кожній ітерації.

```
Edge  : Weight
7 - 8 : 1
0 - 1 : 2
0 - 6 : 3
1 - 3 : 3
1 - 5 : 4
5 - 8 : 4
3 - 4 : 5
0 - 2 : 6
Edge  : Weight
7 - 8 : 1
0 - 1 : 2
0 - 6 : 3
1 - 3 : 3
1 - 5 : 4
5 - 8 : 4
3 - 4 : 5
0 - 2 : 6
Edge  : Weight
7 - 8 : 1
0 - 1 : 2
0 - 6 : 3
1 - 3 : 3
1 - 5 : 4
5 - 8 : 4
3 - 4 : 5
0 - 2 : 6
```

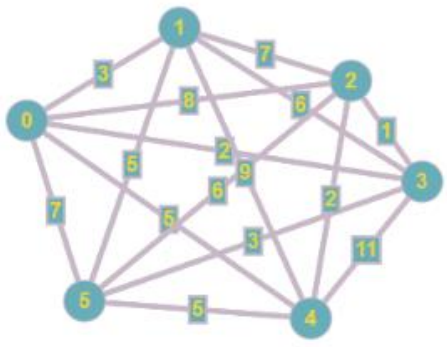
3. Після всіх 1000 повторень алгоритму Крускала отримуємо час виконання всіх повторень.

```
Edge  : Weight
7 - 8 : 1
0 - 1 : 2
0 - 6 : 3
1 - 3 : 3
1 - 5 : 4
5 - 8 : 4
3 - 4 : 5
0 - 2 : 6
```

Приклади та візуалізація вхідних даних:

Приклад 1:

Граф:



Таблиця суміжності цього графа:

	1	2	3	4	5	6
1	0	3	8	2	5	7
2	3	0	7	6	9	5
3	8	7	0	1	2	6
4	2	6	1	0	11	3
5	5	9	2	11	0	5
6	7	5	6	3	5	0

Вхідні дані алгоритм Прима:

```
#define V 6 //задать константу равную количеству вершин
int G[V][V] = {
{INF, 3, 8, 2, 5, 7},
{3, INF, 7, 6, 9, 5},
{8, 7, INF, 1, 2, 6},
{2, 6, 1, INF, 11, 3},
{5, 9, 2, 11, INF, 5},
{7, 5, 6, 3, 5, INF},
}; //матрица смежности в виде двумерного массива
```

Вхідні дані алгоритм Крускала:

```
Graph g(6); //количество вершин в графе
g.AddWeightedEdge(0, 1, 3);
g.AddWeightedEdge(0, 2, 8);
g.AddWeightedEdge(0, 3, 2);
g.AddWeightedEdge(0, 4, 5);
g.AddWeightedEdge(0, 5, 7);
g.AddWeightedEdge(1, 0, 3);
g.AddWeightedEdge(1, 2, 7);
g.AddWeightedEdge(1, 3, 6);
g.AddWeightedEdge(1, 4, 9);
g.AddWeightedEdge(1, 5, 5);
g.AddWeightedEdge(2, 0, 8);
g.AddWeightedEdge(2, 1, 7);
g.AddWeightedEdge(2, 3, 1);
g.AddWeightedEdge(2, 4, 2);
g.AddWeightedEdge(2, 5, 6);
g.AddWeightedEdge(3, 0, 2);
g.AddWeightedEdge(3, 1, 6);
g.AddWeightedEdge(3, 2, 1);
g.AddWeightedEdge(3, 4, 11);
g.AddWeightedEdge(3, 5, 3);
g.AddWeightedEdge(4, 0, 5);
g.AddWeightedEdge(4, 1, 9);
```

```
g.AddWeightedEdge(4, 2, 2);
g.AddWeightedEdge(4, 3, 11);
g.AddWeightedEdge(4, 5, 5);
g.AddWeightedEdge(5, 0, 7);
g.AddWeightedEdge(5, 1, 5);
g.AddWeightedEdge(5, 2, 6);
g.AddWeightedEdge(5, 3, 3);
g.AddWeightedEdge(5, 4, 5); //первая вершина, вторая вершина, вес ребра
```

Прима

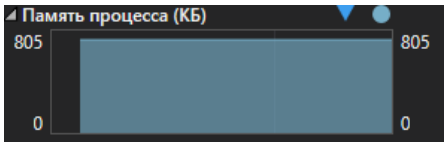
```
Edge   : Weight
0 - 3 :  2
3 - 2 :  1
2 - 4 :  2
0 - 1 :  3
3 - 5 :  3
runtime = 3.088
```

Крускала

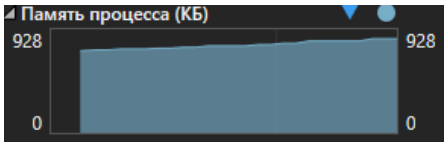
```
Edge   : Weight
2 - 3 : 1
0 - 3 : 2
2 - 4 : 2
0 - 1 : 3
3 - 5 : 3
runtime = 3.096
```

Використання пам'яті під час роботи алгоритму:

Прима:

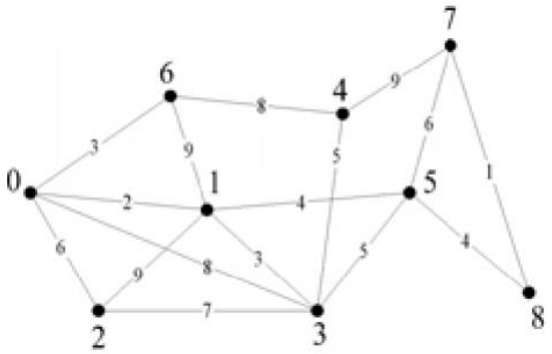


Крускала:



Приклад 2:

Граф:



Таблиця суміжності цього графа:

	0	1	2	3	4	5	6	7	8
0	0	2	6	8	0	0	3	0	0
1	2	0	9	3	0	4	9	0	0

2	6	9	0	7	0	0	0	0	0
3	8	3	7	0	5	5	0	0	0
4	0	0	0	5	0	0	8	9	0
5	0	4	0	5	0	0	0	6	4
6	3	9	0	0	8	0	0	0	0
7	0	0	0	0	9	6	0	0	1
8	0	0	0	0	0	4	0	1	0

Вхідні дані алгоритм Прима:

```
#define V 9 //Количество вершин в графе
int G[V][V] = {
{INF, 2, 6, 8, INF, INF, 3, INF, INF},
{2, INF, 9, 3, INF, 4, 9, INF, INF},
{6, 9, INF, 7, INF, INF, INF, INF, INF},
{8, 3, 7, INF, 5, 5, INF, INF, INF},
{INF,INF,INF, 5, INF, INF, 8, 9, INF},
{INF, 4, INF, 5, INF, INF, INF, 6, 4},
{3, 9, INF, INF, 8, INF, INF, INF,INF},
{INF, INF, INF, INF, 9, 6, INF, INF, 1},
{INF, INF, INF, INF, INF, 4, INF, 1, INF},
};
```

Вхідні дані алгоритм Крускала:

```
Graph g(9); //количество вершин в графе
g.AddWeightedEdge(0, 1, 2);
g.AddWeightedEdge(0, 2, 6);
g.AddWeightedEdge(0, 6, 3);
g.AddWeightedEdge(0, 3, 8);
g.AddWeightedEdge(1, 0, 2);
g.AddWeightedEdge(1, 2, 9);
g.AddWeightedEdge(1, 3, 3);
g.AddWeightedEdge(1, 6, 9);
g.AddWeightedEdge(1, 5, 4);
g.AddWeightedEdge(2, 0, 6);
g.AddWeightedEdge(2, 1, 9);
g.AddWeightedEdge(2, 3, 7);
g.AddWeightedEdge(3, 0, 8);
g.AddWeightedEdge(3, 1, 3);
g.AddWeightedEdge(3, 4, 5);
g.AddWeightedEdge(3, 5, 5);
g.AddWeightedEdge(4, 3, 5);
g.AddWeightedEdge(4, 6, 8);
g.AddWeightedEdge(4, 7, 9);
g.AddWeightedEdge(5, 1, 4);
g.AddWeightedEdge(5, 3, 5);
g.AddWeightedEdge(5, 7, 6);
g.AddWeightedEdge(5, 8, 4);
```

```
g.AddWeightedEdge(6, 0, 3);
g.AddWeightedEdge(6, 1, 9);
g.AddWeightedEdge(6, 4, 8);
g.AddWeightedEdge(7, 4, 9);
g.AddWeightedEdge(7, 5, 6);
g.AddWeightedEdge(7, 8, 1);
g.AddWeightedEdge(8, 7, 1);
g.AddWeightedEdge(8, 5, 4);
```

Прима

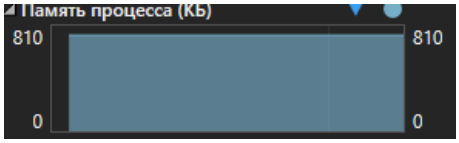
```
Edge   : Weight
7 - 8 : 1
0 - 1 : 2
0 - 6 : 3
1 - 3 : 3
1 - 5 : 4
5 - 8 : 4
3 - 4 : 5
0 - 2 : 6
```

Крускала

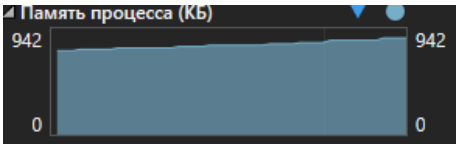
```
Edge   : Weight
7 - 8 : 1
0 - 1 : 2
0 - 6 : 3
1 - 3 : 3
1 - 5 : 4
5 - 8 : 4
3 - 4 : 5
0 - 2 : 6
```

Використання пам'яті під час роботи алгоритму:

Прима:

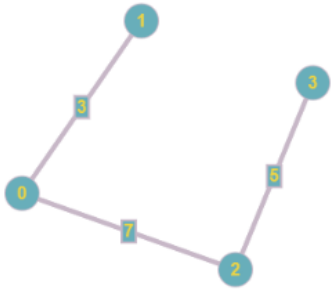


Крускала:



Приклад 3:

Граф:



Таблиця суміжності цього графа:

	1	2	3	4
1	0	3	7	0
2	3	0	0	0
3	7	0	0	5
4	0	0	5	0

Вхідні дані алгоритм Прима:

```
#define V 4 //задать константу равную количеству вершин
int G[V][V] = {
{INF, 3, 7, INF},
{3, INF, INF, INF},
{7, INF, INF, 5},
{INF, INF, 5, INF},
};  //матрица смежности в виде двумерного массива
```

Вхідні дані алгоритм Крускала:

```
Graph g(4); //количество вершин в графе
g.AddWeightedEdge(0, 1, 3);
g.AddWeightedEdge(0, 2, 7);
g.AddWeightedEdge(1, 0, 3);
g.AddWeightedEdge(2, 0, 7);
g.AddWeightedEdge(2, 3, 5);
```

Прима

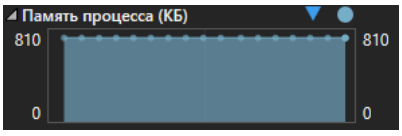
```
Edge   : Weight
0 - 1 :  3
0 - 2 :  7
2 - 3 :  5
runtime = 2.063
```

Крускала

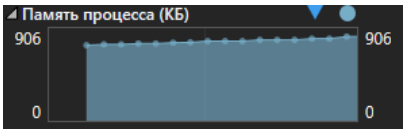
```
Edge   : Weight
0 - 1 : 3
2 - 3 : 5
0 - 2 : 7
runtime = 2.064
```

Використання пам'яті під час роботи алгоритму:

Прима:

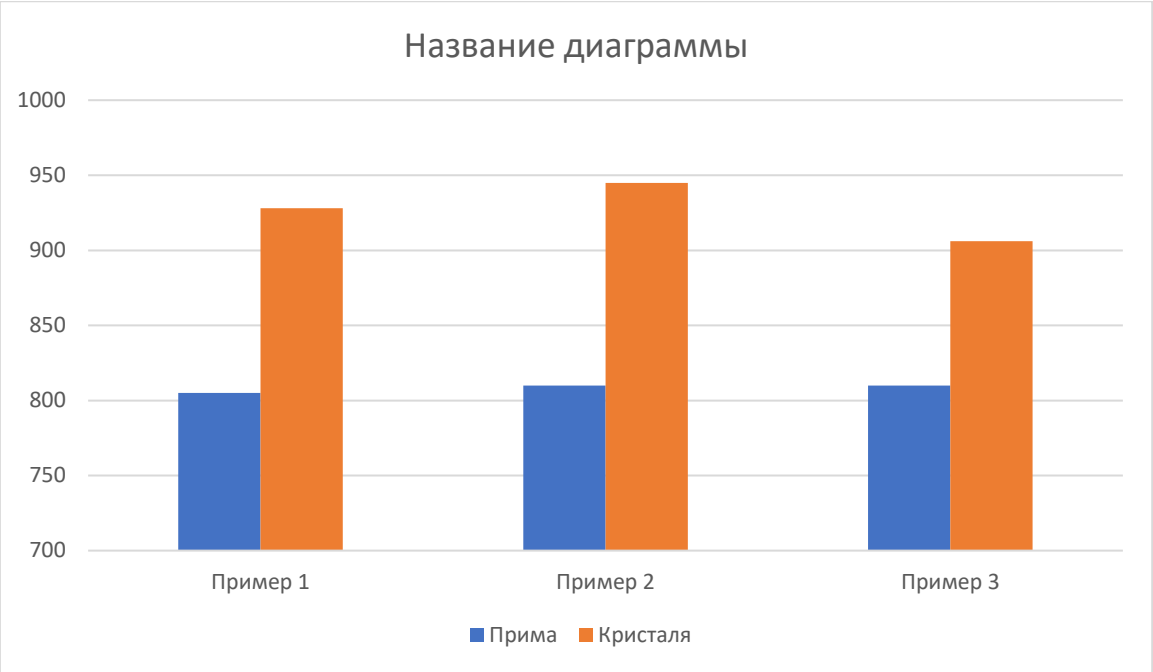


Крускала:

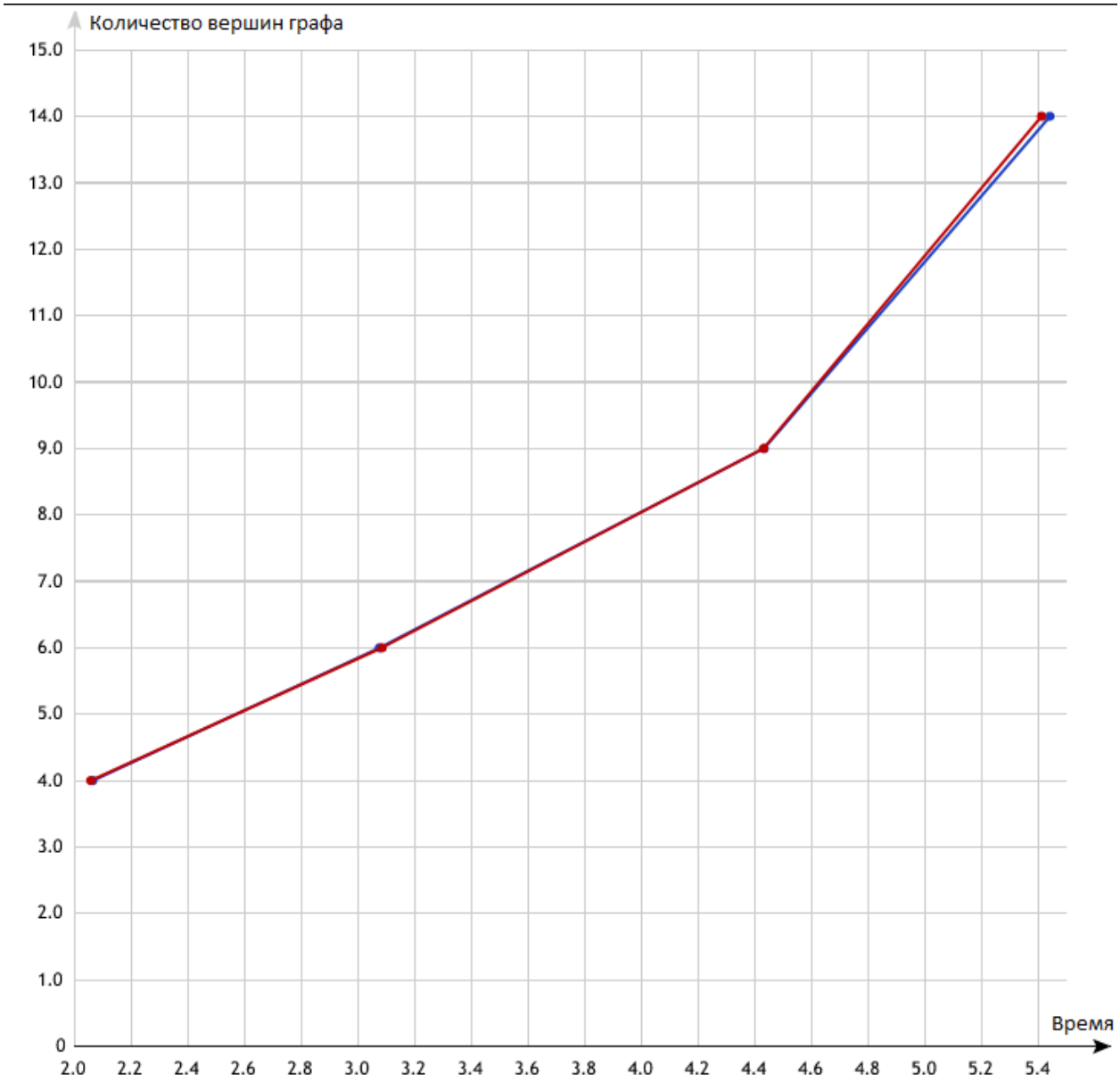


Аналіз отриманих результатів та висновки щодо досягнення мети РГР:

График по памяти:



Графік за часом і кількістю вершин графа (Синій алгоритм Прима, червоний алгоритм Крускала):



Висновок: якщо кількість вершин достатньо мала, то доцільніше використовувати алгоритм Прима, в іншому випадку доцільно користуватися алгоритмом Крускала. Відмінності у швидкості роботи хоча обидва алгоритми працюють за O(MlogN), існують константні відмінності у швидкості роботи. На розріджених графах (кількість ребер приблизно дорівнює кількості вершин) швидше працює алгоритм Крускала, а на насичених (кількість ребер приблизно дорівнює квадрату кількості вершин) - алгоритм Прима (при використанні матриці суміжності).Насправді частіше використовується алгоритм Крускала.