РОЗРАХУНКОВО -ГРАФІЧНА РОБОТА

Козуб К.О АІ-212

Завдання та мета роботи

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.

Алгоритм Пріма

Суть самого алгоритму Прима теж зводиться до жадібного перебору ребер, але вже з певної множини. На вході так само є порожній підграф, який і будемо добудовувати до потенційного мінімального кістяка.

- Спочатку наш підграф складається з однієї будь-якої вершини вихідного графа.
- Потім з ребер інцидентних цій вершині, вибирається таке мінімальне ребро, яке б пов'язало дві абсолютно різні компоненти зв'язності, однією з яких і є наш подграф. Тобто, як тільки у нас з'являється можливість додати нову вершину до нашого підграфа, ми тут же включаємо її за мінімально можливою вагою.
- Продовжуємо виконувати попередній крок доти, доки не знайдемо шукане MST

Алгоритм Крускала

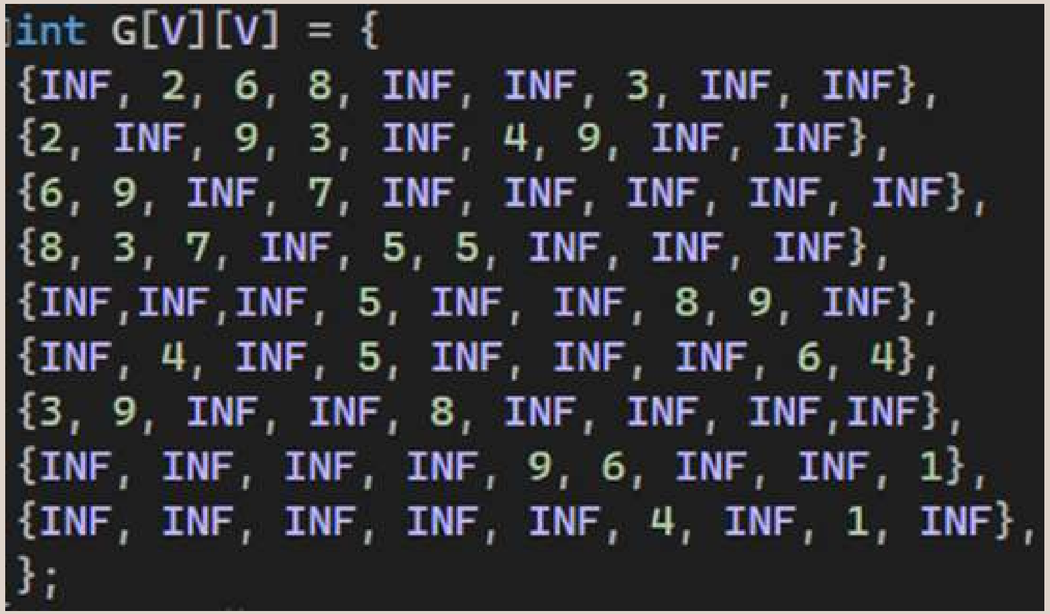
Механізм, яким працює даний алгоритм, дуже простий. На вході є порожній підграф, який і будемо добудовувати до потенційного мінімального кістяка. Будемо розглядати тільки зв'язкові графи, в іншому випадку при застосуванні алгоритму Крускала ми будемо отримувати не мінімальне остовне дерево, а просто остовний ліс.

- Спочатку ми робимо сортування ребер по невпаданню за їх вагами
- Додаємо i-е ребро до нашого підграфа лише в тому випадку, якщо дане ребро з'єднує дві різні компоненти зв'язності, одним з яких є наш підграф. Тобто, на кожному кроці додається мінімальне за вагою ребро, один кінець якого міститься в нашій підграфі, а інший ще немає.
- Алгоритм завершить свою роботу після того, як безліч вершин нашого підграфа збігатиметься з безліччю вершин вихідного графа.

Візуалізація покрокового виконання завдання для підтвердження коректності роботи програми

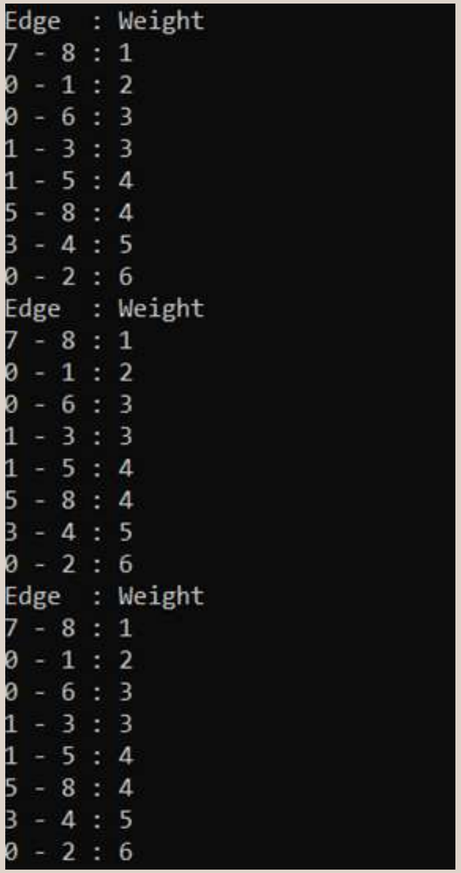
Алгоритм Прима

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

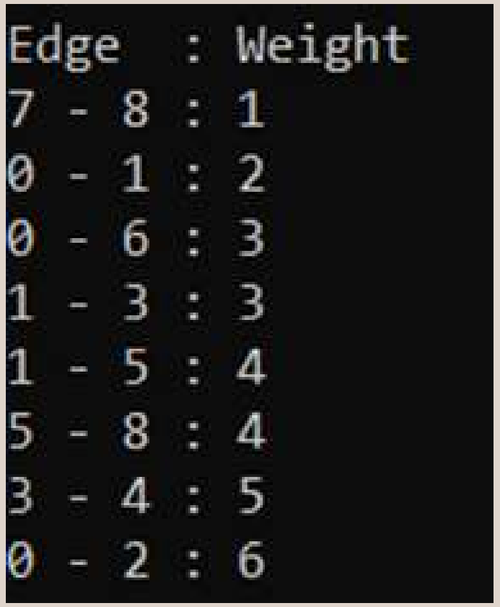


```
int G[V][V] = {
{INF, 2, 6, 8, INF, INF, 3, INF, INF},
{2, INF, 9, 3, INF, 4, 9, INF, INF},
{6, 9, INF, 7, INF, INF, INF, INF, INF},
{8, 3, 7, INF, 5, 5, INF, INF, INF},
{INF,INF,INF, 5, INF, INF, 8, 9, INF},
{INF, 4, INF, 5, INF, INF, INF, 6, 4},
{3, 9, INF, INF, 8, INF, INF, INF,INF},
{INF, INF,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

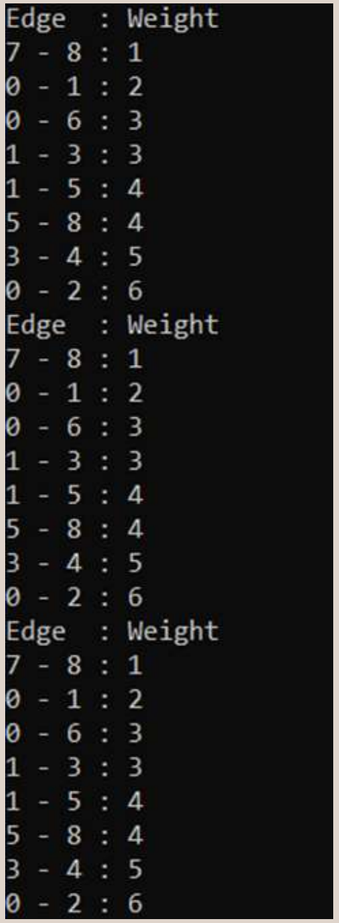


Алгоритм Крускала

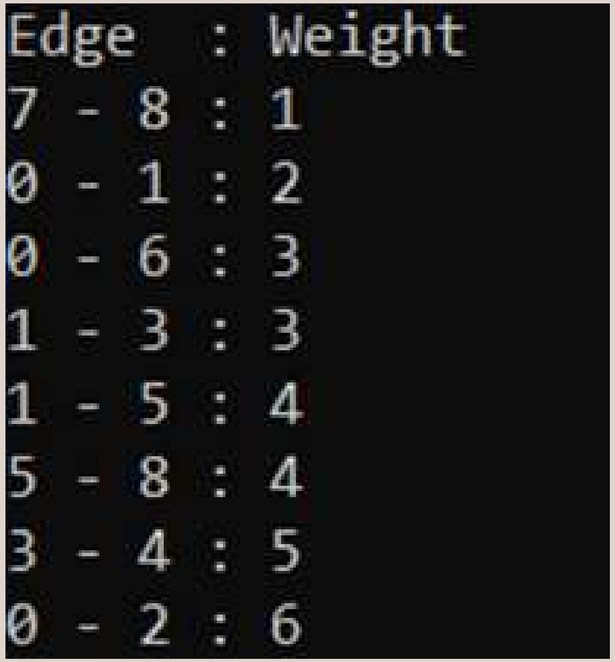
1. Ми вказуємо кількість вершин і додаємо ребра графа, їх ваги та вершини, інцидентні цим ребрам.

```
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.



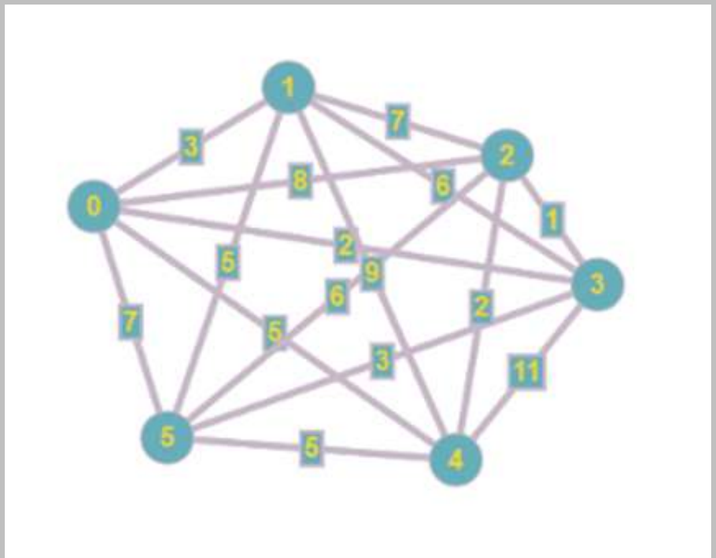
3. Після всіх 1000 повторень алгоритму Крускала отримуємо час виконання всіх повторень.



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

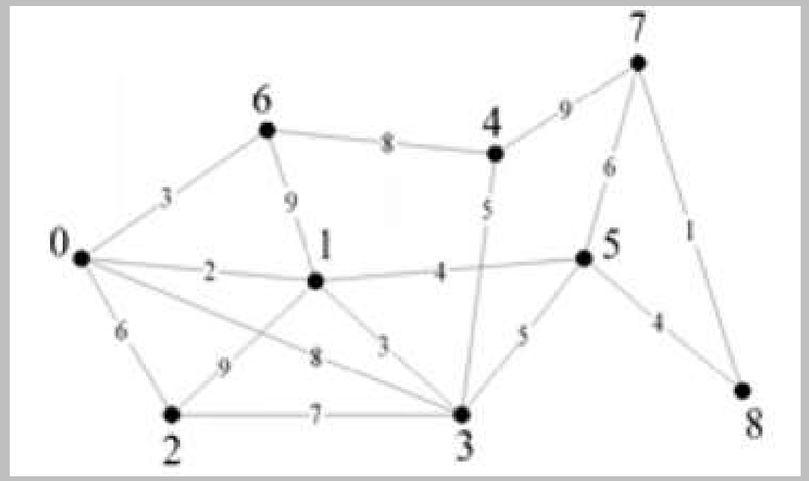
```
Edge   : Weight
0 - 3 :  2
3 - 2 :  1
2 - 4 :  2
0 - 1 :  3
3 - 5 :  3
runtime = 3.088
```

Крускала

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

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

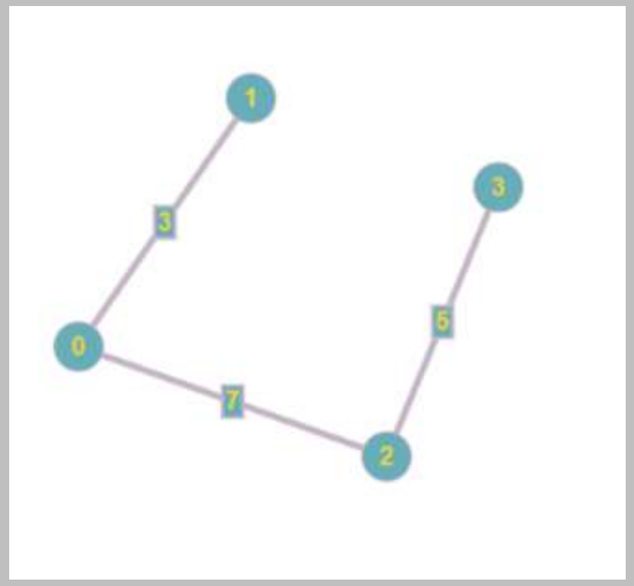
```
Edge   : Weight
7 - 8 : 1
0 - 1 : 2
0 - 6 : 3
1 - 3 : 3
1 - 5 : 4
5 - 8 : 4
3 - 4 : 5
0 - 2 : 6
```

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
```

Приклад 2

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

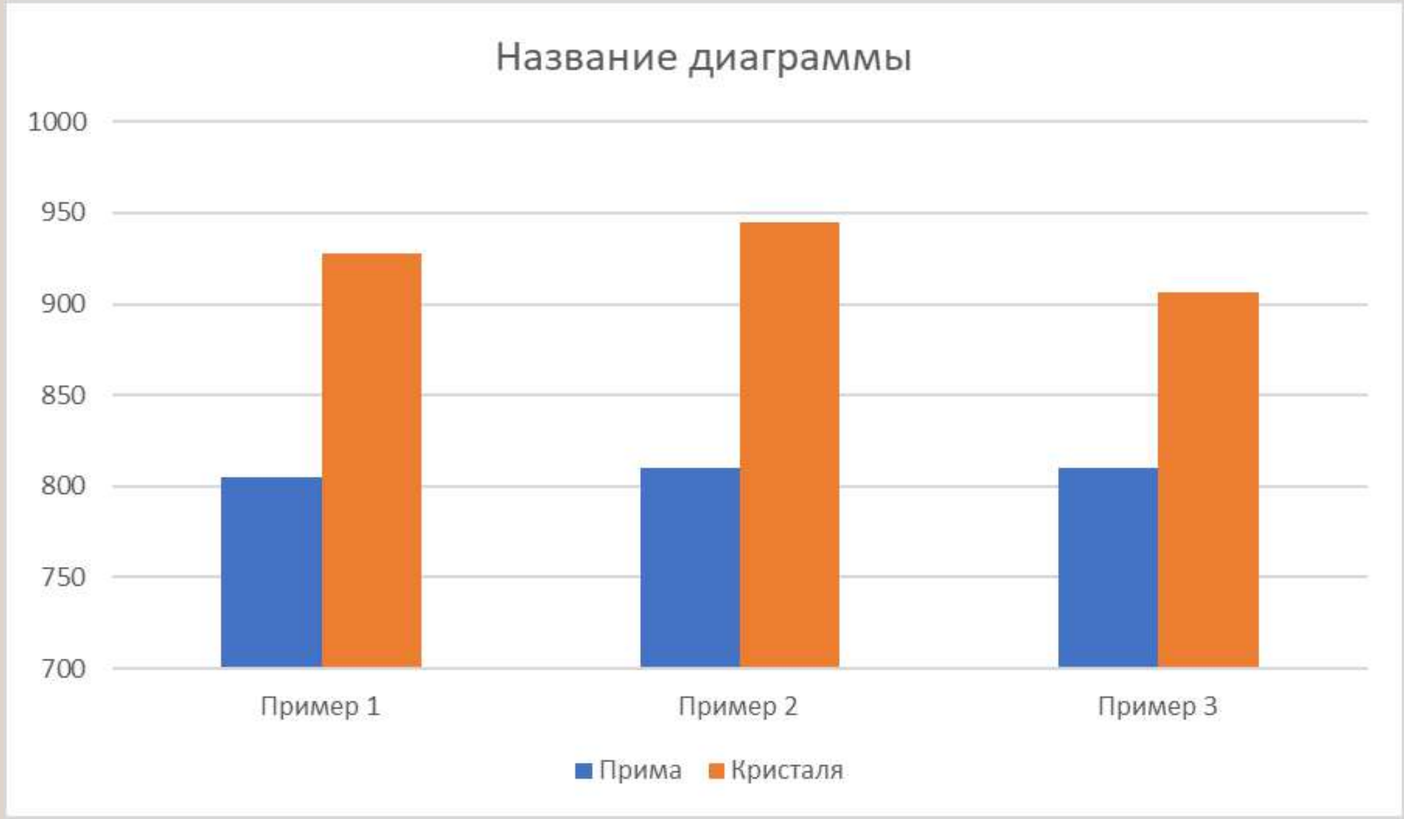
```
Edge   : Weight
0 - 1 :  3
0 - 2 :  7
2 - 3 :  5
runtime = 2.063
```

Крускала

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

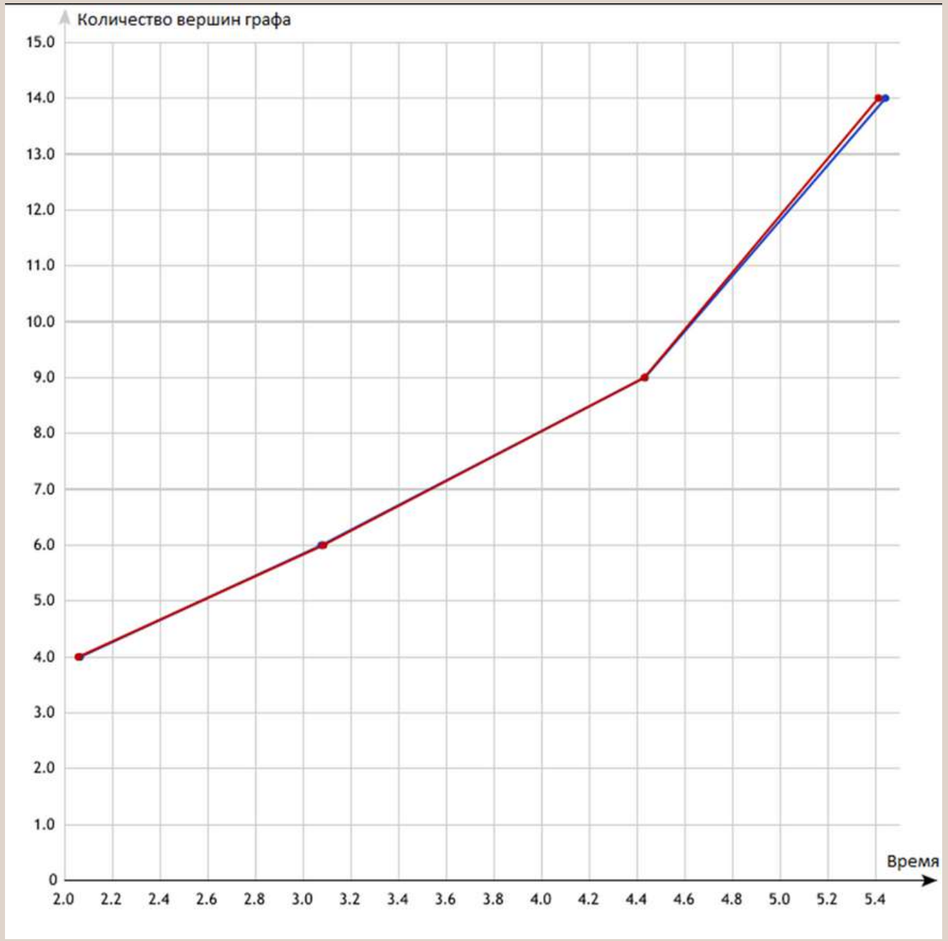
Аналіз отриманих результатів та висновки щодо досягнення мети РГР

График по памяти:



Графік за часом і кількістю вершин графа

(Синій алгоритм Прима, червоний алгоритм Крускала)



Алгоритм Пріма та Крускала

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.