

## Лабораторна робота №5

### Алгоритми на графах. Мінімальне кістякове дерево

Метою виконання лабораторної роботи є набуття практичних навичок із проектування, реалізації, тестування та аналізу алгоритмів Прима (Prim) і Крускала (Kruskal) для побудови мінімального кістякового дерева

Зміст:

1. Представлення графів
2. Визначення мінімального кістякового дерева графа
3. Алгоритм Прима
4. Алгоритм Крускала
5. Завдання
6. Вимоги до представлення звіту

#### 1. Представлення графів, обходи графів.

Граф – абстрактний математичний об'єкт, що являє собою множину *вершин* графа і набір *ребер*, тобто з'єднань між парами вершин. Наприклад, як множину вершин можна прийняти множину аеропортів, що обслуговуються деякою авіакомпанією, а як множину ребер представити регулярні рейси цієї авіакомпанії в ці аеропорти. У деяких графах потрібно встановити напрямок ребрам як у випадку з аеропортом; такі графи називаються *орієнтованими* графами, чи *орграфами*. В іншому випадку ми маємо справу з *неорієнтованими* графами. Якщо в графі є шлях від одного вузла до іншого, такий граф називається *зв'язним*. Інакше йдеться про *незв'язний* граф. Графи із циклами називаються циклічними, а графи без циклів – ациклічними. Досить широке застосування мають орієнтовані ациклічні графи (ОАД).

Як правило, множина вершин позначається як  $V$ , а множина ребер –  $E$ . У такому разі граф  $G$  являє собою  $G = (V, E)$ . У неорієнтованих графах множина  $E$  це множина, що складається з двох наборів елементів  $(V_i, V_j)$  для кожного ребра між двома вершинами графа  $V_i$  і  $V_j$ . Для графа на рисунку 5,а множина вершин та ребер і відображається так:

$$\begin{aligned}V &= \{0, 1, 2, 3\}. \\E &= \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}\end{aligned}$$

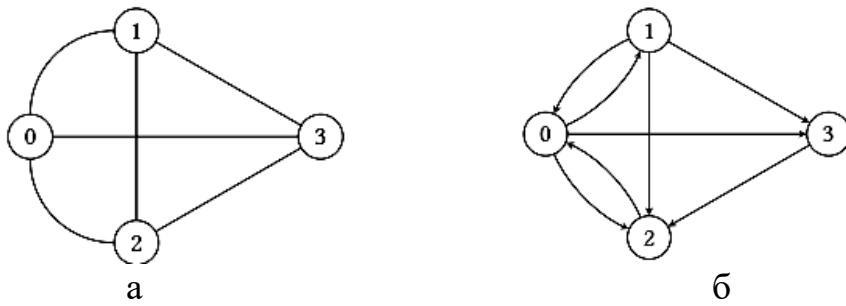


Рисунок 5.1 – Приклади неорієнтованого (а) та орієнтованого (б) графів

У орієнтованих графах множина  $E$  – це множина, що складається з кортежів двох елементів  $\langle V_i, V_j \rangle$  для кожного ребра між двома вершинами графа  $V_i$  та

$V_j$ . Порядок  $V_i$  та  $V_j$  важливий, так як він відповідає за напрямок зв'язку між вершинами..

Граф на рисунку 5.1,б відображається як:

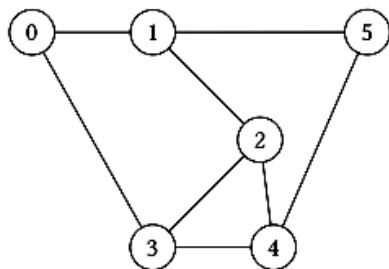
$$V = \{0, 1, 2, 3\}$$

$$E = \{\langle 0, 1 \rangle, \langle 0, 2 \rangle, \langle 0, 3 \rangle, \langle 1, 0 \rangle, \langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 0 \rangle, \langle 3, 2 \rangle\}$$

При розробці алгоритмів з використанням структур даних у вигляді графів їх представляють двома способами:

- у вигляді матриці суміжності (вагових коефіцієнтів);
- у вигляді зв'язаних списків суміжних вершин.

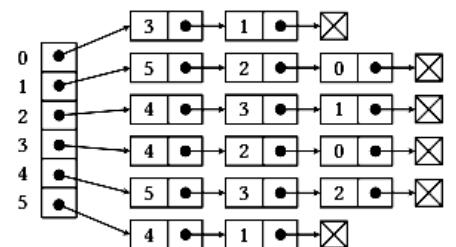
Матриця суміжності графа є квадратною матрицею, в якій для кожної вершини відведені рядок та стовпець. У матриці суміжності на перетині рядка під номером  $i$  та стовпця  $j$  міститься «0», якщо між вершинами графа  $V_i$  і  $V_j$  відсутнє ребро та «1», якщо таке ребро є. Матрицю суміжності для графа, зображеного на рисунку 5.2,а, показано на рисунку 2,б. Як бачимо, розмір матриці суміжності  $V^2$



а

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

б



в

Рисунок 5.2 – Вигляд графа та його представлень

Щоб зменшити обсяг необхідної пам'яті для представлення вершин графа використовують масив, кожен елемент якого позначає одну з вершин та запускає кортеж, що складається з вершин, що є сусідами з обраною вершиною. Такий кортеж називається списком суміжних вершин графа (Рис.5.2,в). Список – це структура даних, у якій містяться елементи – вузли списку, які складаються з двох частин. Перша частина містить дані, що описують елемент, а друга частина містить покажчик на наступний елемент списку. Щоб створити представлення графа за допомогою списку суміжних вершин, необхідні наступні процедури

- $L \leftarrow CreateList()$  створює та повертає новий, порожній список.
- $InsertInList(L, p, d)$  додає до списку  $L$  вузол, що містить дані  $d$ , після вузла  $p$ . Якщо  $p = null$ , тоді нам потрібен новий вузол, який стане новою голововою списку.

Щоб переглянути елементи списку  $L$ , нам потрібно викликати:

$n \leftarrow GetNextListNode(L, null)$  та отримати перший елемент; потім, поки  $n \neq null$ . Можемо повторно призначити  $n \leftarrow GetNextListNode(L, n)$ . Можна отримати доступ до даних усередині вузла, наприклад, за допомогою функції  $GetData(n)$ , яка повертає дані  $d$ , що зберігаються у вузлі  $n$ .

Наприклад, якщо є числа 3, 1 і 0 і ми в такому порядку додаємо їх на початок списку, то список збільшиться від [ ] до [0, 1, 3], як показано на рисунку 5.3.

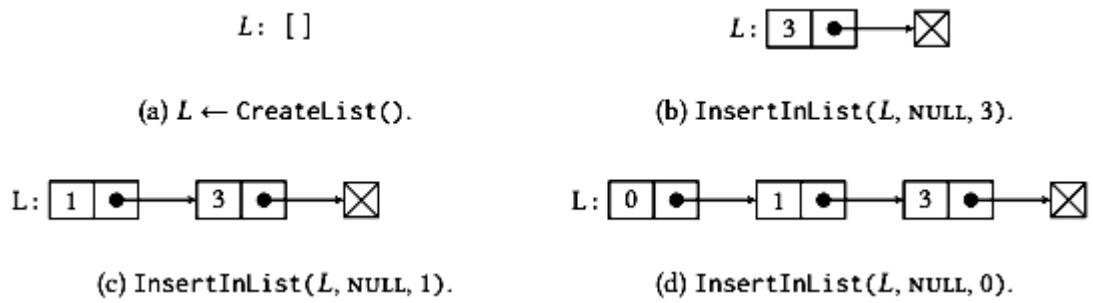


Рисунок 5.3 Додавання вузлів на початок списку

Таким чином, за допомогою структури даних список ми створюємо представлення графа у вигляді списку суміжності. У цьому представленні ми маємо один список суміжності для кожної вершини. Усі вони об'єднані в масив, який вказує на початок списків суміжних вершин. Якщо є масив  $A$ , об'єкт  $A[i]$  даного масиву вказує на початок списку суміжності вузла  $i$ . Якщо поруч із вузлом  $i$  немає інших вузлів,  $A[i]$  вказуватиме на null.

Для графа на рисунку 5.2, а показаний список суміжності на рисунку 5.2, в. Зліва на рисунку 5.4, в розміщений масив, що містить голови списку суміжності графа; кожному об'єкту масиву відповідає певна вершина. Наприклад, третій елемент масиву містить голову списку суміжності для вершини 2. Кожен список суміжності складається із сусідніх вершин, розташованих у числовому порядку. Наприклад, щоб створити список суміжності для вершини 1, ми викликаємо додавання вузлів на початок списку  $\text{InsertInList}(1, \text{null}, d)$  три рази: для вершин 0, 2 та 5, саме в такому порядку. Таким чином, додавши їх у порядку 0, 2, 5, ми у результаті отримаємо список [5, 2, 0].

## 2. Визначення мінімального кістякового дерева графа

У багатьох практичних ситуаціях виникає така задача: потрібно з'єднати  $n$  даних точок таким чином, щоб з *кожної* точки існував шлях до будь-якої іншої, причому *сумарна довжина з'єднань повинна бути мінімальною*. Точки можна представити вершинами графа, а довжини з'єднань – вагами ребер. У такій моделі задача перетворюється на задачу про мінімальне кістякове дерево, формально визначається наступним чином.

*Кістякове (покриваюче рос. оставное) дерево (spanning tree)* зв'язного графа є зв'язним ацикличним підграфом (тобто деревом), яке містить всі вершини графа. *Мінімальне кістякове дерево (minimum spanning tree MTS)* зваженого зв'язкового графа є кістякове дерево з найменшою довжиною (вагою), де довжина дерева визначається як сума довжин усіх його ребер. Задача про мінімальне кістякове дерево є задача пошуку мінімального кістякового дерева для даного зваженого зв'язкового графа (рис. 5.4)

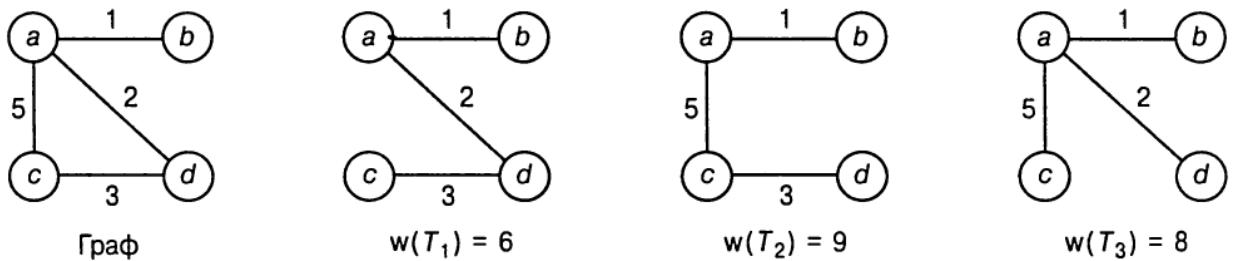


Рисунок – 5.4 Граф та його кістякові дерева.  $T_1$  – мінімальне кістякове дерево

### 3. Алгоритм Пріма

Алгоритм Пріма (Prim's algorithm) був розроблений у 1957 році. Згідно з алгоритмом, мінімальне кістякове дерево будується як послідовність піддерев, що розширяються. Початкове піддерево такої послідовності складається з єдиної вершини, довільно вибраної з множини вершин графа  $V$ . На кожній ітерації ми розширяємо поточне дерево *жадібним* чином, додаючи до нього найближчу вершину, тобто з'єднану з вихідною вершиною дерева ребром з мінімальною довжиною (вагою), що не входить у дерево. Алгоритм завершує роботу після того, як всі вершини вже є включеними в дерево, яке буде відповідати оптимальним критеріям.

*Жадібний алгоритм* – алгоритм, що полягає у прийнятті локально оптимальних рішень на кожному етапі, допускаючи, що кінцеве рішення також виявиться оптимальним.

#### Алгоритм Prim (G)

```

// Вхідні дані: Зважений зв'язний граф  $G = (V, E)$ 
// Вихідні дані:  $E_T$ , множина ребер, що складають MST графа  $G$ 
// Вибір довільної вершини  $v_0 \in V$ 
 $V_T \leftarrow \{v_0\}$  // Множина вершин, що вже включені до MST
 $E_T \leftarrow \emptyset$  // Множина ребер MST
// MST має містити  $|V| - 1$  ребро
for  $i \leftarrow 1$  to  $|V| - 1$  do
    // 1. Пошук найдешевшого ребра (жадібний вибір) в черзі з пріоритетами
    Пошук ребра з мінімальною вагою  $e^* = (v^*, u^*)$  серед всіх
    ребер  $(v, u)$  таких, що:
         $v \in V_T$  // Ребро виходить із MST
         $u \in V - V_T$  // Ребро входить у ще не включену вершину
    // 2. Включення ребра та нової вершини до MST
     $V_T \leftarrow V_T \cup \{u^*\}$ 
     $E_T \leftarrow E_T \cup \{e^*\}$ 
return  $E_T$ 
```

Дамо наступні пояснення. Псевдокод описує жадібний (Greedy) алгоритм Пріма для знаходження MST у зваженому зв'язному графі  $G$ . Алгоритм послідовно "ростить" дерево від однієї початкової вершини, завжди вибираючи найдешевше ребро, яке розширяє дерево. Після ініціалізації в циклі скануються всі ребра, які є "примежовими" (тобто, з'єднують побудоване дерево  $V_T$  із зовнішнім світом  $V - V_T$ ). При цьому Вибирається ребро  $e^*$  з

найменшою вагою серед усіх можливих кандидатів. Цей локально оптимальний вибір гарантує глобально оптимальне рішення (MST). Множини  $V_T$  та  $V - V_T$ : розділяють ребра на "допустимі" ( $v \in V_T, u \in V - V_T$ ) і "недопустимі" (наприклад, ребро, що з'єднує дві вершини у  $V_T$ , оскільки це створить цикл).

Хоча псевдокод використовує множини  $V_T$  та  $V - V_T$ , згадка про чергу з пріоритетом в коментарі вказує на оптимізовану реалізацію. У такій реалізації черга з пріоритетом використовується для швидкого пошуку цього мінімального ребра за  $O(\log I)$  замість повільного  $O(E)$  перебору.

Наприкінці виконують додавання обраного ребра та вершини до MST. Вершина  $u^*$ , до якої вело мінімальне ребро, переміщується зі множини непереглянутих вершин  $V - V_T$  до множини вершин MST  $V_T$ , обране мінімальне ребро  $e^*$  остаточно включається до результатуючої множини ребер MST.

У таблиці 5.1 показаний хід побудови мінімального кістякового дерева за алгоритмом Прима

Позначення:

$V_T$  – вершини MST

$E$  – множина ребер примежових переглянутим вершинам

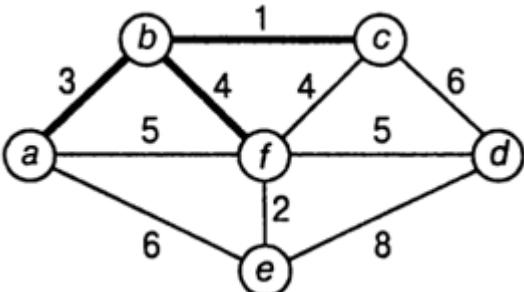
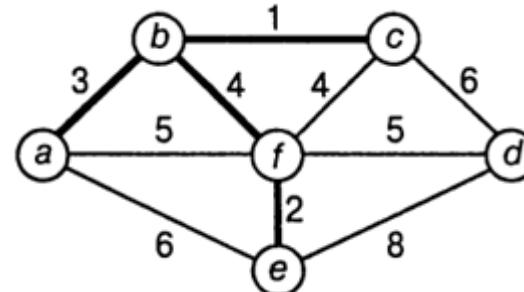
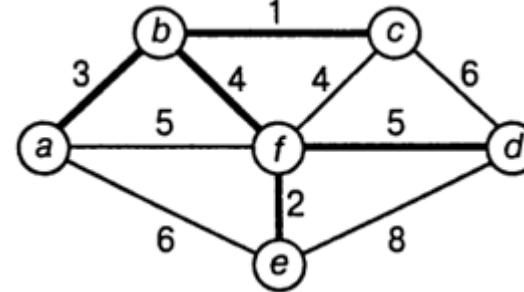
$V - V_T$  – непереглянуті вершини графа  $G$

$e^*$  – ребро з мінімальною вагою, обране на поточному кроці

$E_T$ , множина ребер MST

Таблиця 5.1

Зображення	$V_T$	$E$	$V - V_T$	$e^*$	$E_T$
	{a}	(a,b) = 3 (a,f) = 5 (a,e) = 6	{b,c,d,e,f}	-	$\emptyset$
	{a,b}	(a,f) = 5 (a,e) = 6 (b,c) = 1 (b,f) = 4	{c,d,e,f}	(a,b) = 3	(a,b) = 3
	{a,b,c}	(a,f) = 5 (a,e) = 6 (b,f) = 4 (c,f) = 4 (c,d) = 6	{d,e,f}	(b,c) = 1	(a,b) = 3 (b,c) = 1

	{a,b,c,f}	$(a,f) = 5$ $(a,e) = 6$ $(c,f) = 4$ $(c,d) = 6$ $(f,e) = 2$ $(f,d) = 5$	{d,e}	$(b,f) = 4$	$(a,b) = 3$ $(b,c) = 1$ $(b,f) = 4$
	{a,b,c,f,e}	$(a,f) = 5$ $(a,e) = 6$ $(c,f) = 4$ $(c,d) = 6$ $(f,d) = 5$	{d}	$(f,e) = 2$	$(a,b) = 3$ $(b,c) = 1$ $(b,f) = 4$ $(f,e) = 2$
	{a,b,c,f,e,d}	$(a,f) = 5$ $(a,e) = 6$ $(c,f) = 4$ $(c,d) = 6$	{}	$(f,d) = 5$	$(a,b) = 3$ $(b,c) = 1$ $(b,f) = 4$ $(f,e) = 2$ $(f,d) = 5$

Мінімальне кістякове дерево побудовано.

$(a,b) = 3$   
 $(b,c) = 1$   
 $(b,f) = 4$   
 $(f,e) = 2$   
 $(f,d) = 5$

Його вартість становить:

$$e^* = \sum_{i=0}^{|E_T|-1} e_i^* = 15.$$

Код алгоритму Пріма:

```

1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4. #define INF 9999999
5. // number of vertices in graph
6. #define V 6
7. // create a 2d array of size 6x6
8. //for adjacency matrix to represent graph
9. int G[V][V] = {
10.   {INF, 3, INF, INF, 6, 5},
11.   {3, INF, 1, INF, INF, 4},
12.   {INF, 1, INF, 6, INF, 4},
13.   {INF, INF, 6, INF, 8, 5},
14.   {6, INF, INF, 8, INF, 8},
15.   {5, 4, 4, 5, 2, INF},
16. };
17. int main () {
18.   int no_edge;           // number of edge
19.   // create a array to track selected vertex
20.   // selected will become true otherwise false
21.   int selected[V];
22.   // set selected false initially
23.   memset (selected, false, sizeof (selected));
24.   // set number of edge to 0
25.   no_edge = 0;
```

```

26. // the number of egde in minimum spanning tree will be
27. // always less than (V - 1), where V is number of vertices in
28. //graph
29. // choose 0th vertex and make it true
30. selected[0] = true;
31. int x; // row number
32. int y; // col number
33. // print for edge and weight
34. cout << "Edge" << " : " << "Weight";
35. cout << endl;
36. while (no_edge < V - 1) {
37. //For every vertex in the set S, find the all adjacent vertices
38. // , calculate the distance from the vertex selected at step 1.
39. // if the vertex is already in the set S, discard it otherwise
40. //choose another vertex nearest to selected vertex at step 1.
41.     int min = INF;
42.     x = 0;
43.     y = 0;
44.     for (int i = 0; i < V; i++) {
45.         if (selected[i]) {
46.             for (int j = 0; j < V; j++) {
47.                 if (!selected[j] && G[i][j]) { // not in selected and there is an edge
48.                     if (min > G[i][j]) {
49.                         min = G[i][j];
50.                         x = i;
51.                         y = j;
52.                     }
53.                 }
54.             }
55.         }
56.     }
57.     cout << x << " - " << y << " : " << G[x][y];
58.     cout << endl;
59.     selected[y] = true;
60.     no_edge++;
61. }
62. return 0;
63. }

```

Запустивши наведений вище код, ми отримаємо виведення у вигляді:

```

64. Edge : Weight
65. 0 - 1 : 3
66. 1 - 2 : 1
67. 1 - 5 : 4
68. 5 - 4 : 2
69. 5 - 3 : 5

```

#### 4. Алгоритм Крускала

Алгоритм Крускала (Kruskal's algorithm) також відноситься до класу «жадібних». Алгоритм Крускала шукає мінімальне кістякове дерево зваженого зв'язного графа  $G = (V, E)$  як ацикличний підграф з  $|V| - 1$  ребрами, сума ваг яких мінімальна. При цьому алгоритм будує мінімальне кістякове дерево як послідовність підграфів, що розширюється, які завжди ацикличні, але на проміжних стадіях не завжди зв'язні. Алгоритм починає з сортування ребер графа в неспадному порядку їх ваг. Потім, починаючи з порожнього поточного підграфа, переглядає відсортований список і додає чергове ребро до списку поточного підграфа, якщо не створюється цикл; інакше ребро просто пропускається.

#### Алгоритм Kruskal (G)

```

// Вхідні дані: Зважений зв'язний граф G = (V, E)
// Вихідні дані:  $E_T$ , множина ребер, що складають мінімальне
// кістякове дерево G
// Сортування множини E в неспадному порядку ваг ребер
 $e_0^* \leq \dots e_{|E|-1}^*$ 

```

```

 $E_T \leftarrow \emptyset$ 
 $eCounter \leftarrow 0$  //лічильник ребер, відстежує кількість ребер, доданих до  $E_T$ .
 $k \leftarrow 0$  // лічильник ітерацій, відстежує, яке за порядком відсортоване ребро ми
переглядаємо
while  $eCounter < |V| - 1$  do
     $k = k + 1$ 
    if  $E_T \cup \{e_k^*\}$  – ациклічний граф
         $E_T \leftarrow E_T \cup \{e_k^*\}; eCounter = eCounter + 1$ 
return  $E_T$ 

```

Дамо наступні пояснення. Після сортування ребер та ініціалізації  $E_T$  (MST), як порожнього та лічильників запускається основний цикл - жадібний вибір. На кожній ітерації ми розглядаємо наступне найдешевше ребро  $e_k^*$  з відсортованого списку  $E$ . Виконується перевірка на циклічність **if**  $E_T \cup \{e_k^*\}$  – ациклічний граф. Ребро додається, лише якщо воно не утворює циклу з ребрами, які вже є у  $E_T$ . Якщо цикл не утворився, ребро  $e_k^*$  додається до  $E_T$ , і лічильник  $eCounter$  збільшується.

Для перевірки на ациклічність використовується структура даних DISJOINT SET UNION (DSU), відома також як система непересічних множин або UNION-FIND.

На початку кожна вершина є окремою множиною DSU.

При перевірці на цикл вважаємо, що ребро  $(u, v)$  утворює цикл, якщо вершини  $u$  та  $v$  вже належать до однієї тієї ж множини (одного компоненту зв'язності). Ця перевірка виконується операцією  $\text{FIND}(u) = \text{FIND}(v)$ .

Якщо цикл не утворюється, ми об'єднуємо множини, що містять  $u$  та  $v$ , операцією  $\text{UNION}(u, v)$ .

У таблиці 5.2 показаний хід побудови мінімального кістякового дерева за алгоритмом Крускала

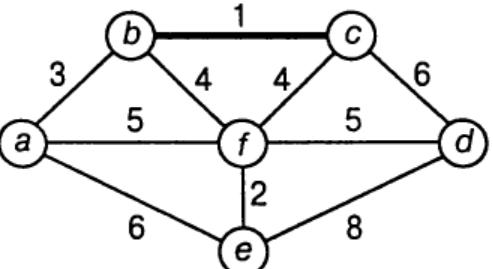
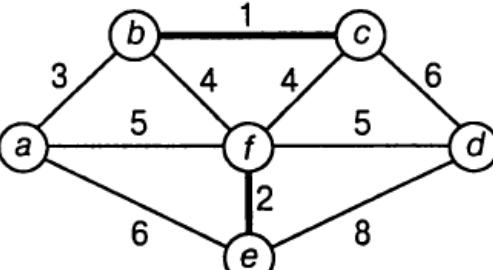
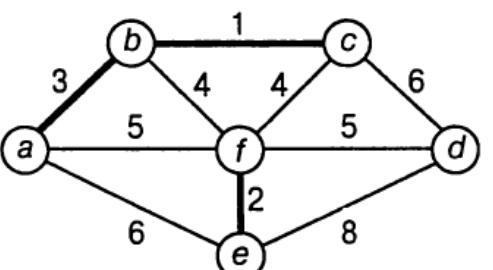
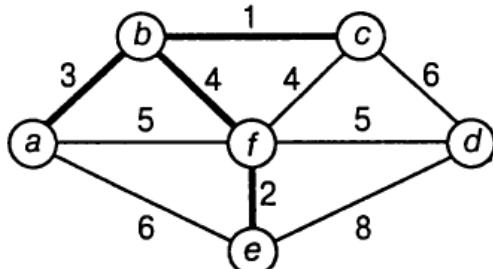
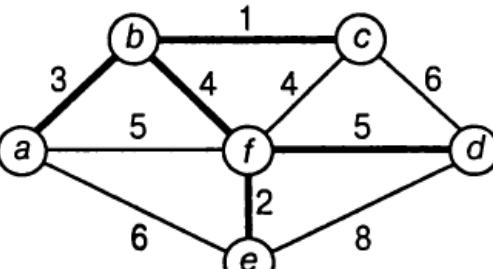
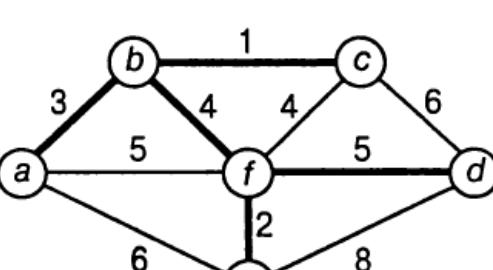
Позначення:

$E$  – множина сортованих неспадному порядку ваг ребер

$E_T$  – множина ребер MST

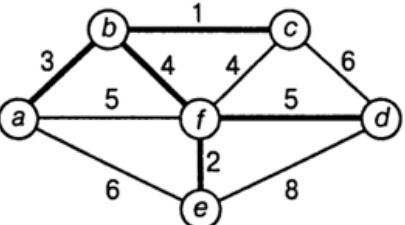
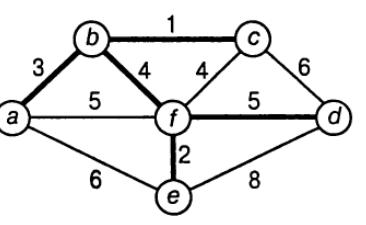
Таблиця 2

Зображення	$E_T$	$E$
	{}	$\{bc = 1\}$ $ef = 2$ $ab = 3$ $bf = 4$ $cf = 4$ $af = 5$ $df = 6$ $ae = 6$ $cd = 8$ $de = 8\}$

	{ bc =1 }	{bc =1 ef=2 ab=3 bf =4 cf=4 af=5 df=5 ae=6 cd=6 de =8}
	{ bc =1 ef=2 }	{bc =1 ef=2 ab=3 bf =4 cf=4 af=5 df=5 ae=6 cd=6 de =8}
	{ bc =1 ef=2 ab=3 }	{bc =1 ef=2 ab=3 bf =4 cf=4 af=5 df=5 ae=6 cd=6 de =8}
	{ bc =1 ef=2 ab=3 bf =4 }	{bc =1 ef=2 ab=3 bf =4 cf=4 af=5 df=5 ae=6 cd=6 de =8}
	{ bc =1 ef=2 ab=3 bf =4 df=5 }	{bc =1 ef=2 ab=3 bf =4 cf=4 (c) af=5 (c) df=5 ae=6 cd=6 de =8}
	{ bc =1 ef=2 ab=3 bf =4 df=5 }	{bc =1 ef=2 ab=3 bf =4 cf=4 (c) af=5 (c) df=5 ae=6 (c) cd=6(c) de =8 (c)} Дерево побудовано

Мінімальне кістякове дерево побудовано. Його вартість становить:

$e^* = \sum_{i=0}^{|E_T|-1} e_i^* = 15$ . І воно повністю збігається з мінімальним кістяковим деревом, побудованим за алгоритмом Прима. Але відрізняється за послідовністю ребер

Алгоритм Прима	Алгоритм Крускала
 <p>(a,b) = 3      (b,c) = 1      (b,f) = 4      (f,e) = 2      (f,d) = 5</p>	 <p>{ bc =1 ef=2      ab=3      bf =4 df =5      }</p>

Код алгоритму Крускала:

```

1. #include <iostream>
2. #include <vector>
3. #include <algorithm>
4. using namespace std;
5. #define edge pair<int,int>
6. class Graph {
7. private:
8.     vector<pair<int, edge>> G; // graph
9.     vector<pair<int, edge>> T; // mst
10.    int *parent;
11.    int V; // number of vertices/nodes in graph
12. public:
13.     Graph(int V);
14.     void AddWeightedEdge(int u, int v, int w);
15.     int find_set(int i);
16.     void union_set(int u, int v);
17.     void kruskal();
18.     void print();
19. };
20. Graph::Graph(int V) {
21.     parent = new int[V];
22.     //i 0 1 2 3 4 5
23.     //parent[i] 0 1 2 3 4 5
24.     for (int i = 0; i < V; i++)
25.         parent[i] = i;
26.     G.clear();
27.     T.clear();
28. }
29. void Graph::AddWeightedEdge(int u, int v, int w) {
30.     G.push_back(make_pair(w, edge(u, v)));
31. }
32. int Graph::find_set(int i) {
33.     // If i is the parent of itself
34.     if (i == parent[i])
35.         return i;
36.     else
37.         // Else if i is not the parent of itself
38.         // Then i is not the representative of his set,
39.         // so we recursively call Find on its parent
40.         return find_set(parent[i]);
41. }
42. void Graph::union_set(int u, int v) {
43.     parent[u] = parent[v];
44. }
45. void Graph::kruskal() {
46.     int i, uRep, vRep;
47.     sort(G.begin(), G.end()); // increasing weight
48.     for (i = 0; i < G.size(); i++) {
49.         uRep = find_set(G[i].second.first);
50.         vRep = find_set(G[i].second.second);

```

```

51.         if (uRep != vRep) {
52.             T.push_back(G[i]); // add to tree
53.             union_set(uRep, vRep);
54.         }
55.     }
56. }
57. void Graph::print() {
58.     cout << "Edge :" << " Weight" << endl;
59.     for (int i = 0; i < T.size(); i++) {
60.         cout << T[i].second.first << " - " << T[i].second.second << " : "
61.             << T[i].first;
62.         cout << endl;
63.     }
64. }
65. int main() {
66.     Graph g(6);
67.     g.AddWeightedEdge(0, 1, 3);
68.     g.AddWeightedEdge(0, 4, 6);
69.     g.AddWeightedEdge(0, 5, 5);
70.     g.AddWeightedEdge(1, 0, 3);
71.     g.AddWeightedEdge(1, 2, 1);
72.     g.AddWeightedEdge(1, 5, 4);
73.     g.AddWeightedEdge(2, 1, 1);
74.     g.AddWeightedEdge(2, 3, 6);
75.     g.AddWeightedEdge(2, 5, 4);
76.     g.AddWeightedEdge(3, 2, 6);
77.     g.AddWeightedEdge(3, 4, 8);
78.     g.AddWeightedEdge(3, 5, 5);
79.     g.AddWeightedEdge(4, 2, 4);
80.     g.AddWeightedEdge(4, 0, 6);
81.     g.AddWeightedEdge(4, 2, 8);
82.     g.AddWeightedEdge(4, 5, 2);
83.     g.AddWeightedEdge(5, 0, 5);
84.     g.AddWeightedEdge(5, 1, 4);
85.     g.AddWeightedEdge(5, 2, 4);
86.     g.AddWeightedEdge(5, 3, 5);
87.     g.AddWeightedEdge(5, 4, 2);
88.     g.kruskal();
89.     g.print();
90.     return 0;
91. }

```

## 5. Завдання до лабораторної роботи

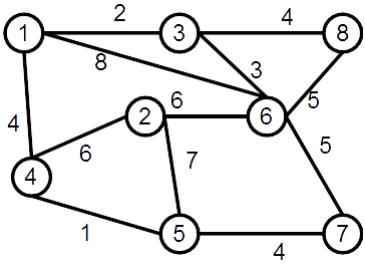
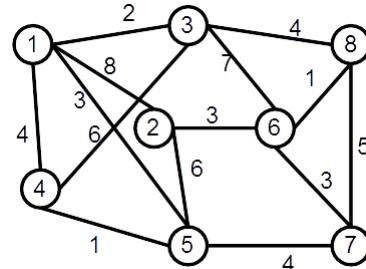
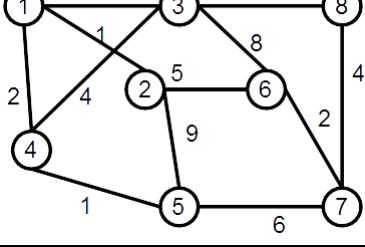
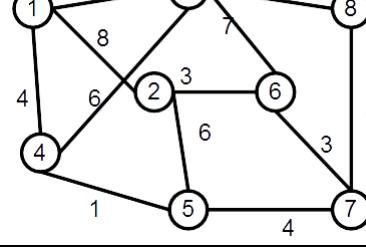
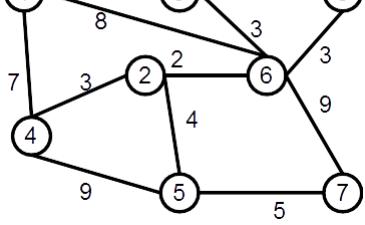
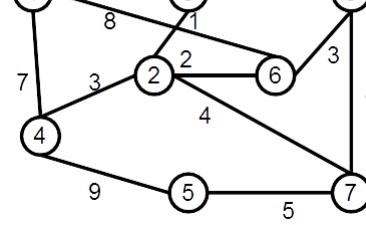
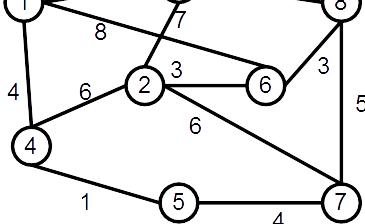
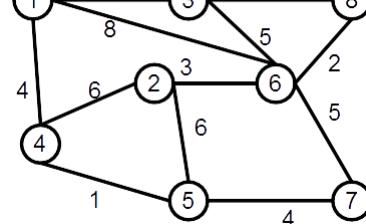
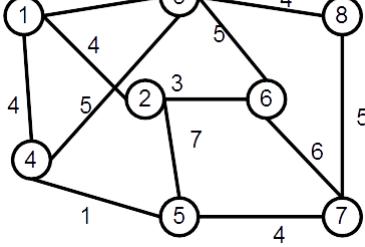
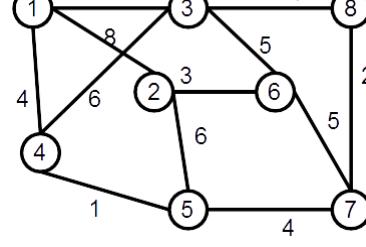
- Побудувати матриці та списки суміжності для зважених неорієнтованих графів за варіантами завдань
- Побудувати за допомогою алгоритму Прима мінімальне кістякове дерево (МКД) для графа за варіантами завдань та показати графічно хід побудови МКД за алгоритмом Пріма (Таблиця 5.1)
- Запрограмувати алгоритм Пріма. Запустити програму, показати результат за варіантами завдань. Показати, що результати ручної та автоматизованої побудови МКД співпадають.
- Побудувати за допомогою алгоритму Крускала МКД для графа за варіантами завдань та показати графічно хід побудови МКД за алгоритмом Крускала (Таблиця 5.2).
- Запрограмувати алгоритм Крускала. Запустити програму, показати результат за варіантами завдань. Показати, що результати ручної та автоматизованої побудови МКД співпадають.
- Порівняти результати побудови МКД за двома алгоритмами, поясніти різницю, якщо вона буде

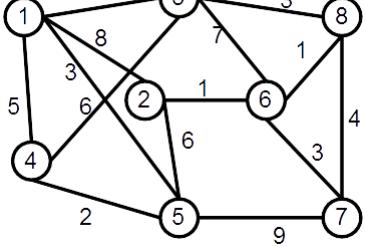
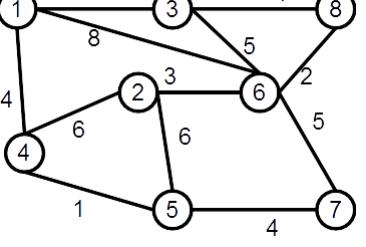
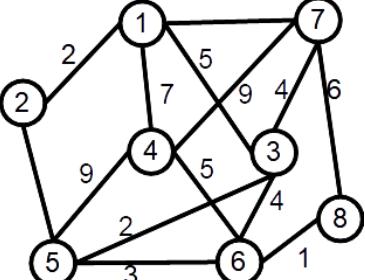
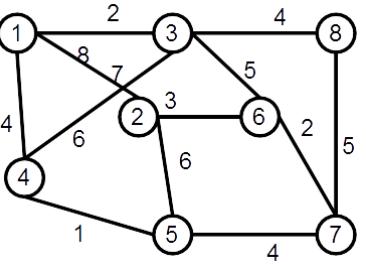
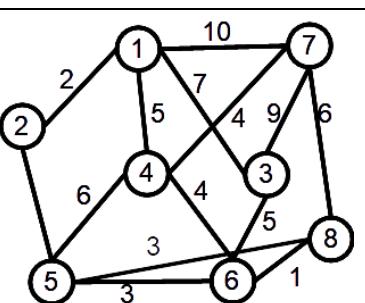
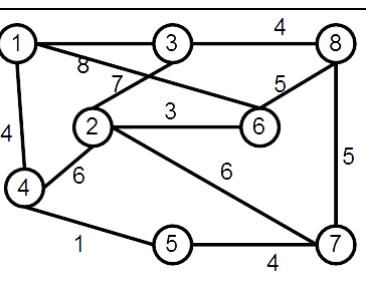
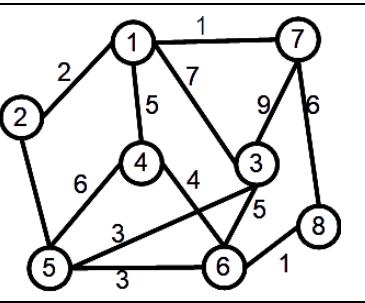
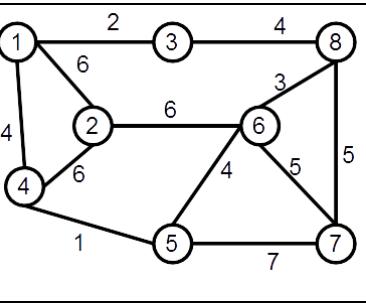
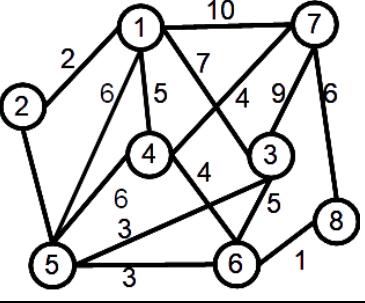
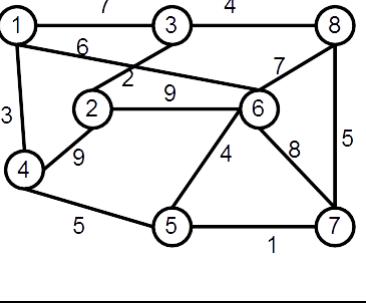
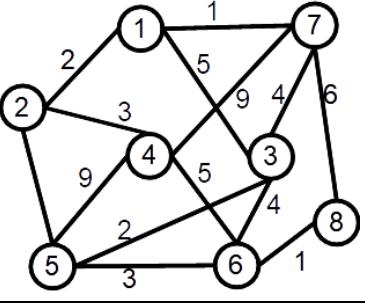
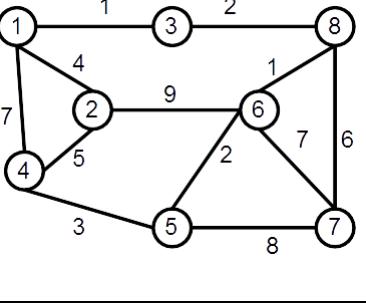
## 6. Звіт повинен містити:

- матриці суміжності та списки суміжності графів за варіантами завдань;

- хід побудови МКД за алгоритмами Пріма та Крускала,
- лістинги програм, результати виконання програм з контрольного прикладу за варіантами завдань,
- порівняння алгоритмів Пріма та Крускала в висновках щодо лабораторної роботи №5.

Варіанти завдань\*:

№	Граф	№	Граф
1		17	
2		18	
9		25	
10		26	
11		27	

12		28	
13		29	
14		30	
3		19	
4		20	
5		21	

6		22	
7		23	
8		24	
15		31	
16		32	

\*Якщо в якомусь графі пропущена довжина ребра або значення довжини дублюються студент може віправити цю помилку зважаючи на власні міркування.