Міністерство освіти і науки України

Національний університет
“Львівська політехніка”

Кафедра систем штучного інтелекту

Розрахункова робота

з дисципліни

«Дискретна математика»

Варіант № 21

Виконав:

студент групи КН-115
Поставка Маркіян

Викладач:

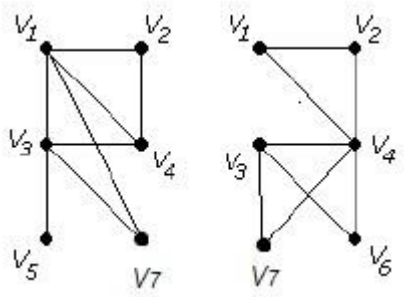
Мельникова Н. І.

Львів – 2019р.

Варіант 21.

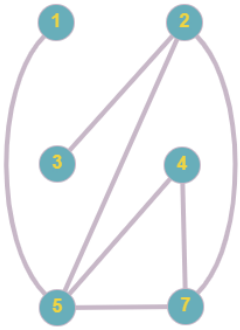
Завдання 1.

Виконати наступні операції над графами: 1) знайти доповнення до першого графу, 2) об'єднання графів, 3) кільцеву сумму G1 та G2 (G1+G2), 4) розмножити вершину у другому графі, 5) виділити підграф А - що скадається з 3-х вершин в G1 6) добуток графів.

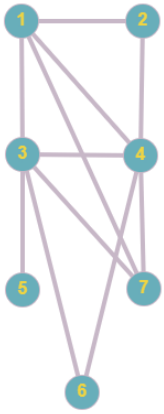


Рішення:

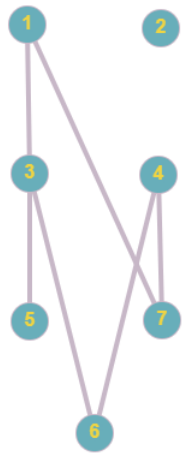
1)



2)

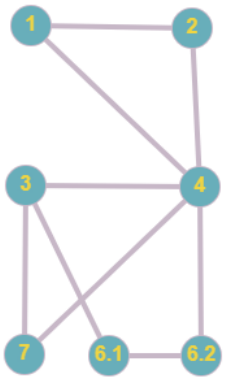


3)



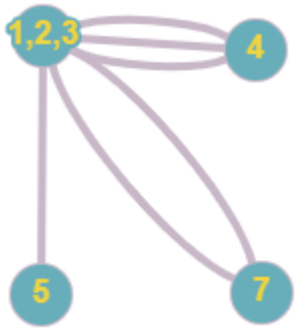
4)

Розмножимо вершину 6.

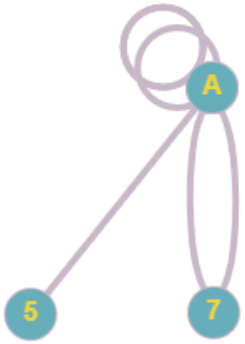


5)

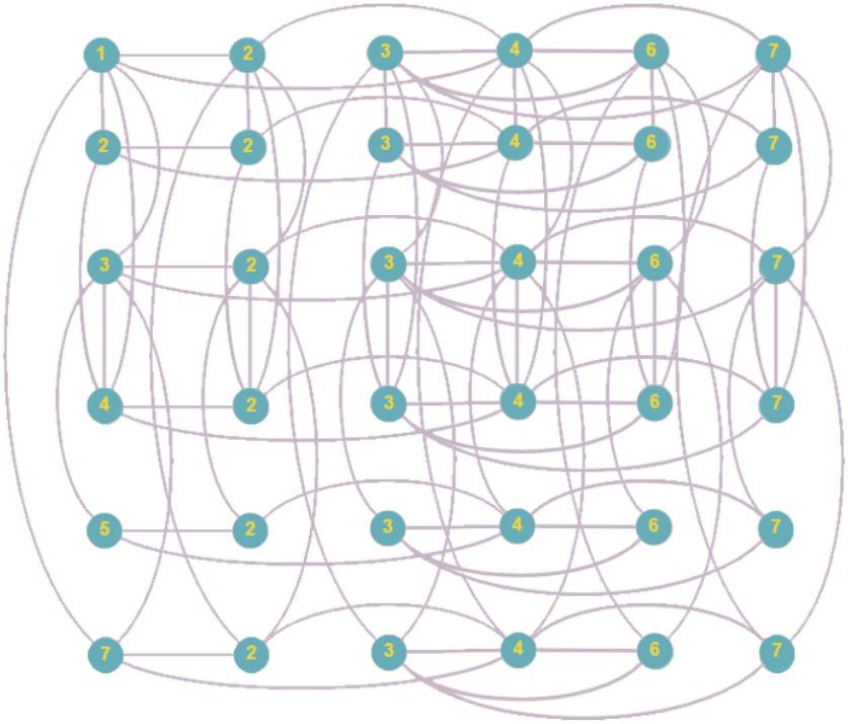
Виділимо підграф А з вершин 1, 2, 3.
Для цього стягнемо 2 в 1 і 3 в 1,2. 1,2,3 – це підграф А.



Стягнемо А в 4.

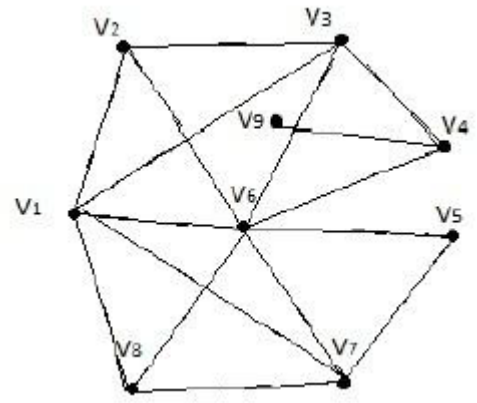


6)



Завдання 2.

Скласти таблицю суміжності для орграфа.



Рішення:

	V1	V2	V3	V4	V5	V6	V7	V8	V9
V1	0	1	1	0	0	1	1	1	0
V2	1	0	1	0	0	1	0	0	0
V3	1	1	0	1	0	1	0	0	0
V4	0	0	1	0	0	1	0	0	1
V5	0	0	0	0	0	1	1	0	0
V6	1	1	1	1	1	0	1	1	0
V7	1	0	0	0	1	1	0	1	0
V8	1	0	0	0	0	1	1	0	0
V9	0	0	0	1	0	0	0	0	0

Завдання 3.

Для графа з другого завдання знайти діаметр.

Рішення:

Діаметр графа: 3

V1 – V3 – V4 – V9

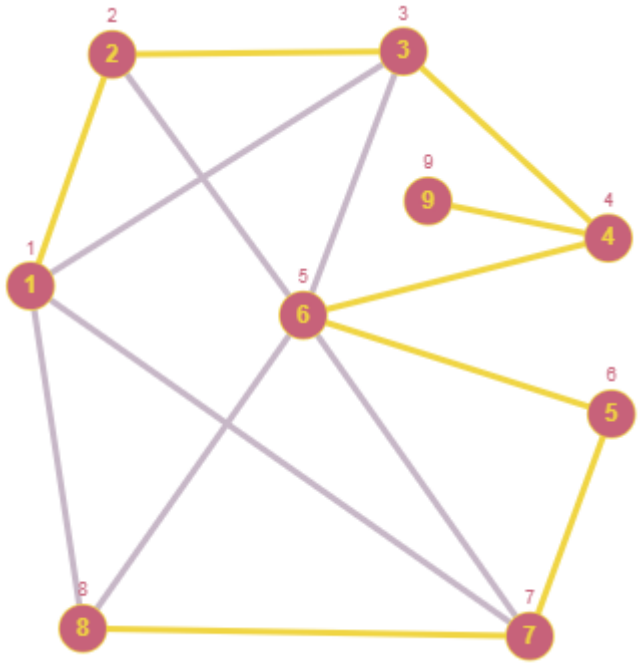
Завдання 4.

Для графа з другого завдання виконати обхід дерева вглиб (варіант закінчується на непарне число) або вшир (закінчується на парне число).

Рішення:

Порядок обходу: 1,2,3,4,6,5,7,8,9

На рисунку підписано якою по порядку ми беремо вершину при обході.

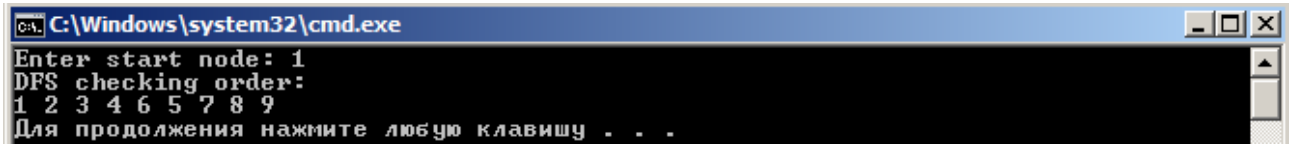


Код програми:

```
#include "stdafx.h"
#include <iostream>
#include <stack>

using namespace std;

const int N = 9;

void DFS(int g[N][N], int visited[N], int ac) // ac == active
{
	int r;
	cout << ac + 1 << " ";
	visited[ac] = true; //Mark this node visited cause we will not call from it dfs anymore,
                                    //everything will be done in a next for cycle
	for (r = 0; r <= N; r++) // Returning if dfs has checked all sumish nodes to the CALLER and serching for next sumish
                                                    // cause for is not ended
		if ((g[ac][r] != 0) && (!visited[r]))
```

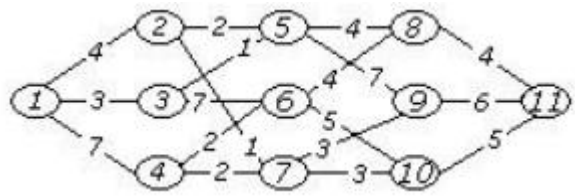
```
                    DFS(g, visited, r); // seeking if node is sumish and not visited
}

int main()
{
	int start_node;
	int graph[N][N] =
	{	{0,	1,	1,	0,	0,	1,	1,	1,	0},
		{1,	0,	1,	0,	0,	1,	0,	0,	0},
		{1,	1,	0,	1,	0,	1,	0,	0,	0},
		{0,	0,	1,	0,	0,	1,	0,	0,	1},
		{0,	0,	0,	0,	0,	1,	1,	0,	0},
		{1,	1,	1,	1,	1,	0,	1,	1,	0},
		{1,	0,	0,	0,	1,	1,	0,	1,	0},
		{1,	0,	0,	0,	0,	1,	1,	0,	0},
		{0,	0,	0,	1,	0,	0,	0,	0,	0,
		} };
	int visited[N];
	for (int i = 0; i < N; i++)
	{
		visited[i] = 0;
	}
	cout << "Enter start node: ";
	cin >> start_node;
	cout << "DFS checking order:" << endl;
	DFS(graph, visited, start_node - 1);
    return 0;
}
```

Результат роботи програми:



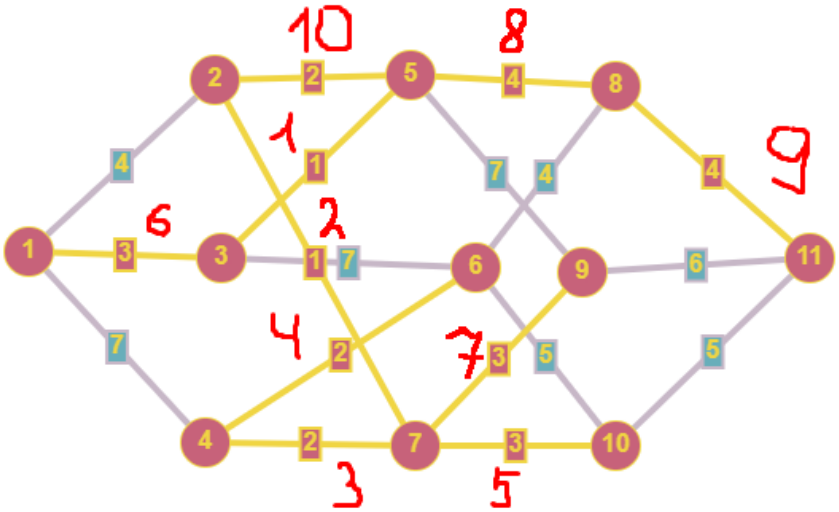
Завдання 5.

Знайти двома методами (Краскала і Прима) мінімальне остове дерево графа.



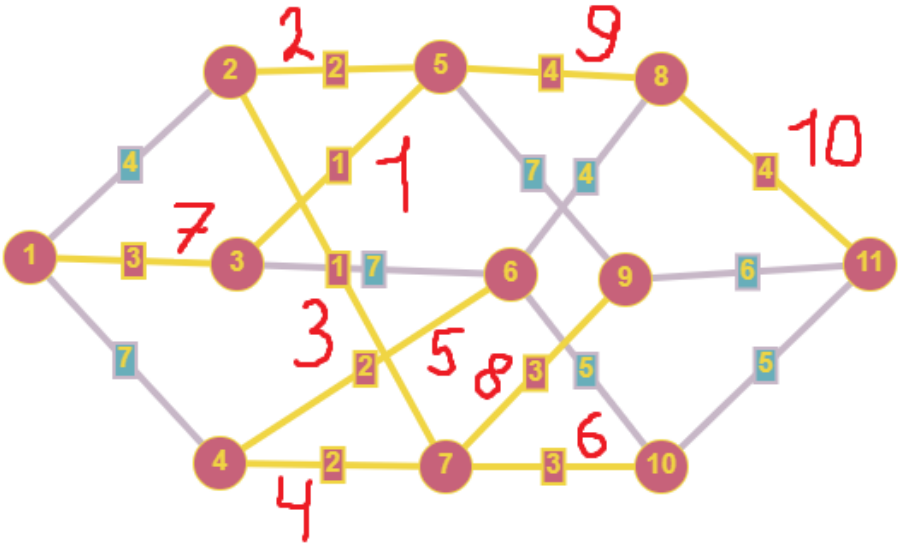
Рішення:

1) Краскала:
   На рисунку наведено порядок додання ребер до дерева.
   Спочатку сортуємо всі ребра по вагам за зростанням.
   Вважаємо що кожна вершина це дерево.
   Об’єднуємо між собою дерева ребрами за зростанням(найкоротшими можливими ребрами):
   1) Якщо два дерева не з’єднані ні з ким то їх з’єднуємо.
   2) Ми під’єднуємо до одних дерев інші найкоротшими ребрами поки не утвориться єдине дерево, що об’єднує всі вершини ребрами з найменшою довжиною. Воно і буде мінімальним кістяковим деревом.



2) Прима:
   На рисунку наведено порядок додання ребер до дерева.
   Спочатку сортуємо всі ребра по вагам за зростанням.
   Найменше ребро додають в дерево.
   Продовжуючи прохід по ребрам від найменшого, вибирають наступне ребро одна вершина якого належить дереву а інша ні.
   Це ребро долучаємо до дерева і знову проходимось по усім ребрам за зростанням від найменшого і шукаємо яке ребро додати.

Коли всі вершини додані, то робота завершена і знайдене дерево і є кістяковим.



Код програми:

Краскала:

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>

using namespace std;

struct graph
{
	int a;
	int b;
	int w;
	bool ck_a;
	bool ck_b;

	graph()
	{
	}

	graph(int a, int b, int w) : a{ a }, b{ b }, w{ w }
	{
		this->ck_a = 0;
		this->ck_b = 0;
	}
};

//Initialisation of graph
void initgraph(graph *g, int nodes, int sides)
{
	g[0] = { 1,2,4 };
	g[1] = { 2,5,2 };
	g[2] = { 5,8,4 };
	g[3] = { 8,11,4 };
	g[4] = { 11,10,5 };
	g[5] = { 10,7,3 };
	g[6] = { 7,4,2 };
	g[7] = { 4,1,7 };
	g[8] = { 1,3,3 };
	g[9] = { 3,5,1 };
	g[10] = { 5,9,7 };
	g[11] = { 9,11,6 };
	g[12] = { 9,7,3 };
	g[13] = { 2,7,1 };
	g[14] = { 3,6,7 };
	g[15] = { 6,4,2 };
	g[16] = { 6,10,5 };
	g[17] = { 6,8,4 };
}

void sort(graph *g, int sides)
{
	graph t;
	for (int j = 0; j < sides - 1; j++)
	{
		for (int i = 0; i < sides - 1; i++)
		{
			if (g[i + 1].w < g[i].w)
			{
				t = g[i];
				g[i] = g[i + 1];
				g[i + 1] = t;
			}
		}
```

```
	}

	for (int i = 0; i < sides; i++)
	{
		cout << i << "\t" << g[i].a << "\t" << g[i].b << "\t" << g[i].w << "\t" <<
endl;
	}
	cout << endl;
}


void MOD_P(graph *g, int nodes, int sides)
{
	sort(g, sides);

	int *ck_node = new int[nodes]; // Масив перевірки на приналежність
	int *tree = new int[nodes-1];
	int tree_i = 0;// Кількість доданих ребер у дерево
	int ck_i = 0; // Кількість записаних

	for (int i = 0; i < nodes; i++)
	{
		ck_node[i] = 0;
	}

	cout << "Min ost tree" << endl;
	cout << "A\t" << "B\t" << "Weight\t" << endl;
	cout << endl;
	cout << g[0].a << "\t" << g[0].b << "\t" << g[0].w << "\t" << endl;
	g[0].ck_a = 1;	g[0].ck_b = 1;
	ck_node[ck_i] = g[0].a;
	ck_i++;
	ck_node[ck_i] = g[0].b;
	ck_i++;


	int active_i = 0; // Значення яке провіряється

	while (ck_i < nodes)
	{
		for (int i = 1; i < sides; i++) // Записуємо якщо наша вершина входить в
		перевірені
		{
			for (int j = 0; j < ck_i; j++)
			{
				if (g[i].a == ck_node[j])
				{
					g[i].ck_a = 1;
					break;
				}
			}

			for (int j = 0; j < ck_i; j++)
			{
				if (g[i].b == ck_node[j])
				{
					g[i].ck_b = 1;
					break;
				}
			}
		}

		for (int i = 1; i < sides; i++)
		{
			if ((g[i].ck_a == 0 && g[i].ck_b == 0)) // Якщо сторона не в дереві то
			долучаємо
			{
				g[i].ck_a = 1;
```

```
                        ck_node[ck_i] = g[i].a;
                        ck_i++;

                        g[i].ck_b = 1;
                        ck_node[ck_i] = g[i].b;
                        ck_i++;

                        cout << g[i].a << "\t" << g[i].b << "\t" << g[i].w << "\t" <<
                        endl;
                        tree[tree_i] = i;
                        tree_i++;
                        break;
                }

                if ((g[i].ck_a == 1 && g[i].ck_b == 0) || (g[i].ck_a == 0 && g[i].ck_b
== 1)) // Під'єднуємо до дерева ті ребра які інцидентні і мають мінімальну вагу
        // Але це ще не обов'язково кістякове дерево
                {
                        if (g[i].ck_a == 1)
                        {
                                cout << g[i].a << "\t" << g[i].b << "\t" << g[i].w << "\t"
                                << endl;
                                g[i].ck_b = 1;
                                ck_node[ck_i] = g[i].b;
                                ck_i++;
                                tree[tree_i] = i;
                                tree_i++;
                                break;
                        }

                        if (g[i].ck_b == 1)
                        {
                                cout << g[i].a << "\t" << g[i].b << "\t" << g[i].w << "\t"
                                << endl;
                                g[i].ck_a = 1;
                                ck_node[ck_i] = g[i].a;
                                ck_i++;
                                tree[tree_i] = i;
                                tree_i++;
                                break;
                        }
                }
        }

        }

        bool tree_flag;
        for (int i = 1; i < sides; i++) // Об'єднуємо окремі дерева в одне яке і є кістяковим
        {
                tree_flag = 0;
                for (int j = 0; j < tree_i; j++)
                {
                        if ((g[i].ck_a == 1 && g[i].ck_b == 1)) // Якщо 2 вершини сторони
                        перевірені але не додані у дерево
                        // То додаємо їх у дерево
                        {
                                if ((i == tree[j]))
                                {
                                        tree_flag = 1; // Повідомляємо що ця сторона не додана
                                        break;
                                }

                        }
                }
                if (tree_flag == 0) // Додаємо цю сторону
                {
                        ck_i++;
                        ck_i++;
```

```
                        cout << g[i].a << "\t" << g[i].b << "\t" << g[i].w << "\t" << endl;
                        tree[tree_i] = i;
                        tree_i++;
                }
                if (tree_i == (nodes - 1 - 1))
                {
                        break; // Дивимося чи вже всі додали, якщо так то виходимо
                }
        }

        delete[] ck_node;
        delete[] tree;
}


int main()
{
        int nodes = 11;
        int sides = 18;
        graph *g = new graph[sides];
        cout << "Initialized graph" << endl;
        cout << "Side\t" << "A\t" << "B\t" << "Weight\t" << endl;
        cout << endl;
        initgraph(g, nodes, sides);

        MOD_P(g, nodes, sides);

        delete[] g;
        _getch();
        return 0;
}
```

Прима:

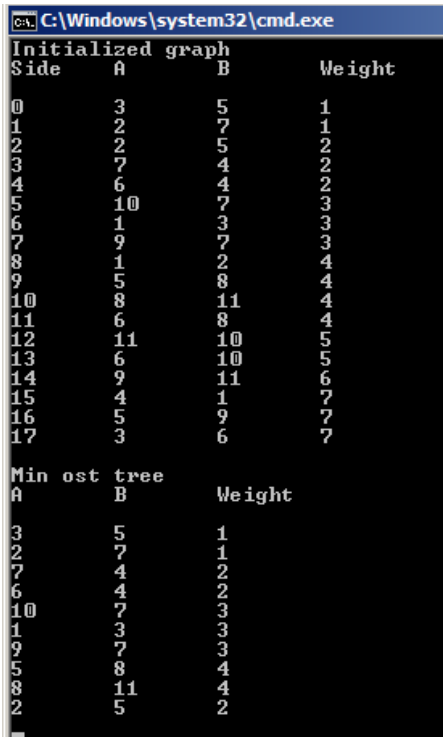
```
#include "stdafx.h"
#include <iostream>
#include <conio.h>

using namespace std;

struct graph
{
        int a;
        int b;
        int w;
        bool ck_a;
        bool ck_b;

        graph()
        {
        }

        graph(int a, int b, int w) : a{ a }, b{ b }, w{ w }
        {
                this->ck_a = 0;
                this->ck_b = 0;
        }
};

//Initialisation of graph
void initgraph(graph *g, int nodes, int sides)
{
        g[0] = { 1,2,4 };
        g[1] = { 2,5,2 };
        g[2] = { 5,8,4 };
        g[3] = { 8,11,4 };
```

```
	g[4] = { 11,10,5 };
	g[5] = { 10,7,3 };
	g[6] = { 7,4,2 };
	g[7] = { 4,1,7 };
	g[8] = { 1,3,3 };
	g[9] = { 3,5,1 };
	g[10] = { 5,9,7 };
	g[11] = { 9,11,6 };
	g[12] = { 9,7,3 };
	g[13] = { 2,7,1 };
	g[14] = { 3,6,7 };
	g[15] = { 6,4,2 };
	g[16] = { 6,10,5 };
	g[17] = { 6,8,4 };
}

void sort(graph *g, int sides)
{
	graph t;
	for (int j = 0; j < sides - 1; j++)
	{
		for (int i = 0; i < sides - 1; i++)
		{
			if (g[i + 1].w < g[i].w)
			{
				t = g[i];
				g[i] = g[i + 1];
				g[i + 1] = t;
			}
		}
	}

	for (int i = 0; i < sides; i++)
	{
		cout << i<< "\t" << g[i].a << "\t" << g[i].b << "\t" << g[i].w << "\t" <<
endl;
	}
	cout << endl;
}

void MOD_P(graph *g, int nodes, int sides)
{
	sort(g, sides);

	int *ck_node = new int[nodes]; // Масив перевірки на приналежність

	int ck_i = 0; // Кількість записаних

	for (int i = 0; i < nodes; i++)
	{
		ck_node[i] = 0;
	}
	cout << "Min ost tree" << endl;
	cout << "A\t" << "B\t" << "Weight\t" << endl;
	cout << endl;
	cout << g[0].a << "\t" << g[0].b << "\t" << g[0].w << "\t" << endl;
	g[0].ck_a = 1;	g[0].ck_b = 1;
	ck_node[ck_i] = g[0].a;
	ck_i++;
	ck_node[ck_i] = g[0].b;
	ck_i++;

	int active_i = 0; // Значення яке провіряється

	while (ck_i < nodes)
	{
		for (int i = 1; i < sides; i++) // Записуємо якщо наша вершина входить в
		перевірені
```

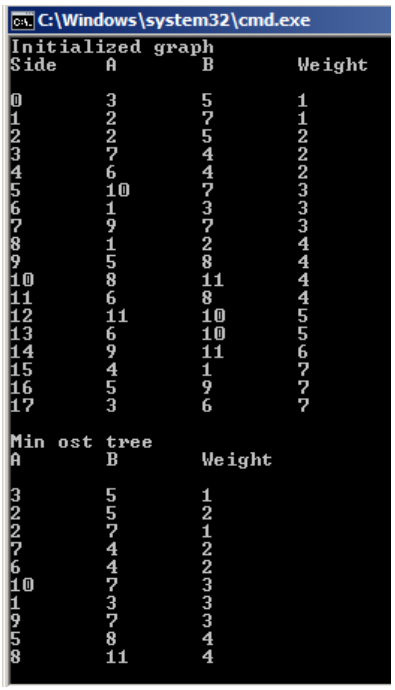
```
		{
			for (int j = 0; j < ck_i; j++)
			{
				if (g[i].a == ck_node[j])
				{
					g[i].ck_a = 1;
					break;
				}
			}

			for (int j = 0; j < ck_i; j++)
			{
				if (g[i].b == ck_node[j])
				{
					g[i].ck_b = 1;
					break;
				}
			}
		}

		for (int i = 1; i < sides; i++) // рахуємо кількість суміжних вершин
		{
			if ( (g[i].ck_a == 1 && g[i].ck_b == 0) || (g[i].ck_a == 0 && g[i].ck_b
			== 1) )
			{
				if (g[i].ck_a == 1)
				{
					cout << g[i].a << "\t" << g[i].b << "\t" << g[i].w << "\t"
					<< endl;
					g[i].ck_b = 1;
					ck_node[ck_i] = g[i].b;
					ck_i++;
					break;
				}

				if (g[i].ck_b == 1)
				{
					cout << g[i].a << "\t" << g[i].b << "\t" << g[i].w << "\t"
					<< endl;
					g[i].ck_a = 1;
					ck_node[ck_i] = g[i].a;
					ck_i++;
					break;
				}
			}
		}
	}

	delete[] ck_node;
}

int main()
{
	int nodes = 11;
	int sides = 18;
	graph *g = new graph[sides];
	cout << "Initialized graph" << endl;
	cout <<"Side\t" << "A\t" << "B\t" << "Weight\t" << endl;
	cout << endl;
	initgraph(g, nodes, sides);

	MOD_P(g, nodes, sides);

	delete[] g;
	_getch();
	return 0;
}
```

Результат роботи програми:

1) Краскала:



2) Прима:



Завдання 6.

Розв'язати задачу комівояжера для повного 8-ми вершинного графа методом «іди у найближчий», матриця вагів якого має вигляд:

21)

	1	2	3	4	5	6	7	8
1	∞	1	3	5	1	5	3	2
2	1	∞	6	6	6	1	5	5
3	3	6	∞	7	3	5	4	1
4	5	6	7	∞	5	5	5	1
5	1	6	3	5	∞	6	6	6
6	5	1	5	5	6	∞	5	2
7	3	5	4	5	6	5	∞	2
8	2	5	1	1	6	2	2	∞

Рішення:

1 –> 5 = 1

	2	3	4	15	6	7	8
2		6	6	6	1	5	5
3	6		7	3	5	4	1
4	6	7		5	5	5	1
15	6	3	5		6	6	6
6	1	5	5	6		5	2
7	5	4	5	6	5		2
8	5	1	1	6	2	2	

Альтернативний шлях

1 –> 2 = 1

	12	3	4	5	6	7	8
12		6	6	6	1	5	5
3	6		7	3	5	4	1
4	6	7		5	5	5	1
5	6	3	5		6	6	6
6	1	5	5	6		5	2
7	5	4	5	6	5		2
8	5	1	1	6	2	2	

1 –> 5–>3 = 4

	2	153	4	6	7	8
2		6	6	1	5	5
153	6		7	5	4	1
4	6	7		5	5	1
6	1	5	5		5	2
7	5	4	5	5		2
8	5	1	1	2	2	

1 –> 5–>3–>8 = 5

	2	4	6	7	1538
2		6	1	5	5
4	6		5	5	1
6	1	5		5	2
7	5	5	5		2
1538	5	1	2	2	

1 –> 5–>3–>8–>4 = 6

	2	15384	6	7
2		6	1	5
15384	6		5	5
6	1	5		5
7	5	5	5	

1 –> 5–>3–>8–>4–>6 = 11

	2	153846	7
2		1	5
153846	1		5
7	5	5	

1 –> 5–>3–>8–>4–>6–>2 = 12

	1538462	7
1538462		5
7	5	

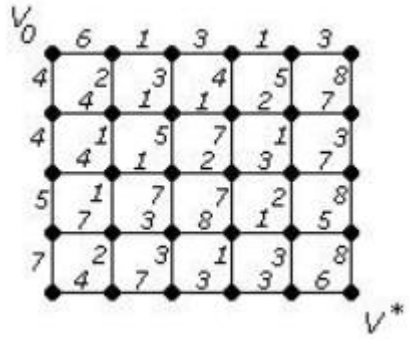
1 –> 5–>3–>8–>4–>6–>2–> 7= 17

	15384627
15384627	

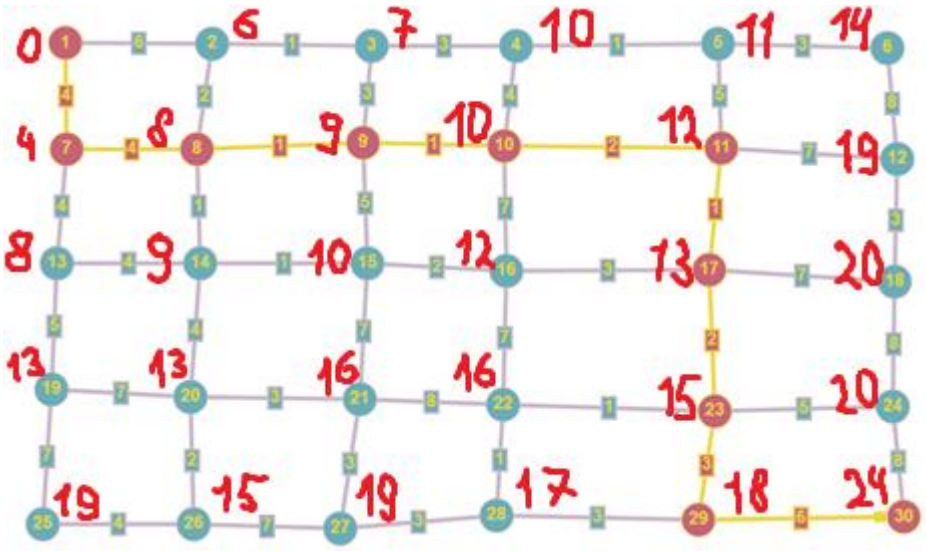
Ми виконали одну ітерацію методом іди у найближчий і отримали шлях довжиною 17. Цей розв’язок задачі комівояжера можливо не є найбільш оптимальним, тому що для загального випадку щоб отримати найбільш оптимальний розв’язок потрібно використовувати повний перебір.

Завдання 7.

За допомогою алгоритму Дейкстри знайти найкоротший шлях у графі між парою вершин V_0 і V^*.



Рішення:



Відстань між вершинами = 24.

Маршрут: 1 –> 7 –>8–>9–>10–>11–>17–>23–>29–>30

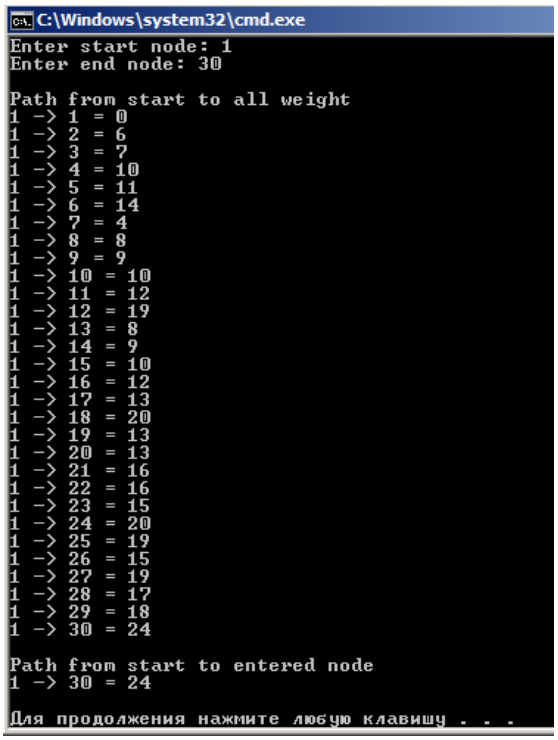
Код програми:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

const int N = 30; // Num of nodes
const int inf = 1000; // infinity

void Dijkstra(int[N][N], int, int);

int main()
{
	int start_node;
	int end_node;

	// Initializing here beacuse to complicated to do this in function
	int gaph[N][N] =
	{ { 0, 6, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
	{ 6, 0, 1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
```

```
        { 0, 1, 0, 3, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 3, 0, 1, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 1, 0, 3, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
        { 4, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 2, 0, 0, 0, 0, 4, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 3, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 4, 0, 0, 0, 0, 1, 0, 2, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 5, 0, 0, 0, 0, 2, 0, 7, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 4, 0, 1, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 1, 0, 2, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 2, 0, 3, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 3, 0, 7, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 7, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 7, 0, 3, 0, 0, 0, 0, 2, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 3, 0, 8, 0, 0, 0, 0, 3, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 8, 0, 1, 0, 0, 0, 0, 1 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 1, 0, 5, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 4, 0, 0, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 4, 0, 7, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 7, 0, 3 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 3, 0 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 3 },
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0 } };

        cout << "Enter start node: ";     cin >> start_node;
        cout << "Enter end node: ";              cin >> end_node;

        Dijkstra(gaph, start_node - 1, end_node-1);

        system("pause");
        return 0;
}

void Dijkstra(int g[N][N], int st, int ed)
{
        int weight[N]; // Array of nodes weights
        int k;
        int i;
        int u; // Active node
        bool ck[N]; // If visited

        for (i = 0; i < N; i++)
        {
                weight[i] = inf; ck[i] = false;
        }

        weight[st] = 0;
        for (k = 0; k < N - 1; k++)
        {
                int min = inf;
                for (i = 0; i < N; i++)
                        if (!ck[i] && weight[i] <= min) // If not checked and weight != inf
                        {
                                min = weight[i]; u = i;
                        }

                ck[u] = true;
                for (i = 0; i < N; i++)
                {
                        if (!ck[i] && (g[u][i]) && (weight[u] != inf) &&         // If !ck,
sumish(g[u][i]!=0), d[u]!=max -> pereN na shumishnist,
                                (weight[u] + g[u][i] < weight[i]))                                      //
If actiNe_w+cur_w<inf || actiNe_w+cur_w<preN_w

                                 //(if current_node weight is higher than base+current_side weight we change
current_node weight)
                        {
                                weight[i] = weight[u] + g[u][i];
                        }
                }

        }

        cout <<"\nPath from start to all weight\n";
        for (i = 0; i < N; i++) if (weight[i] != inf)
```

```
                cout << st + 1 << " -> " << i + 1 << " = " << weight[i] << endl;
        else cout << st + 1 << " > " << i + 1 << " = " << "No way from start to this node" << endl;

        cout <<"\nPath from start to entered node\n";
        if (weight[ed] != inf)
                cout << st + 1 << " -> " << ed + 1 << " = " << weight[ed] << endl;
        else cout << st + 1 << " -> " << ed + 1 << " = " << "No way from start to this node" << endl;
        cout << endl;
}
```

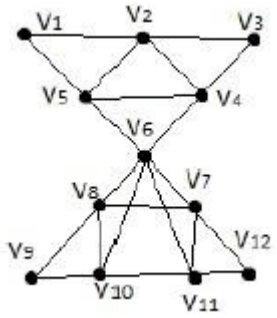
Результат роботи програми:



Програма рахує за нас відстані від початкової вершини до інших вершин. Таким чином вона обрахувала відстань від вершини 1 до вершини 30, яка дорівнює 24. Щоб перевірити правильність шляху ми можемо пройтись від кінцевої вершини і перевіряти чи обрані нами суміжні вершини мають мінімальний шлях серед суміжних. Таким чином ми дійдемо до першої вершини і доведемо, що шлях вірний.

Завдання 8.

Знайти ейлеровий цикл в ейлеровому графі двома методами: а) Флері; б) елементарних циклів.

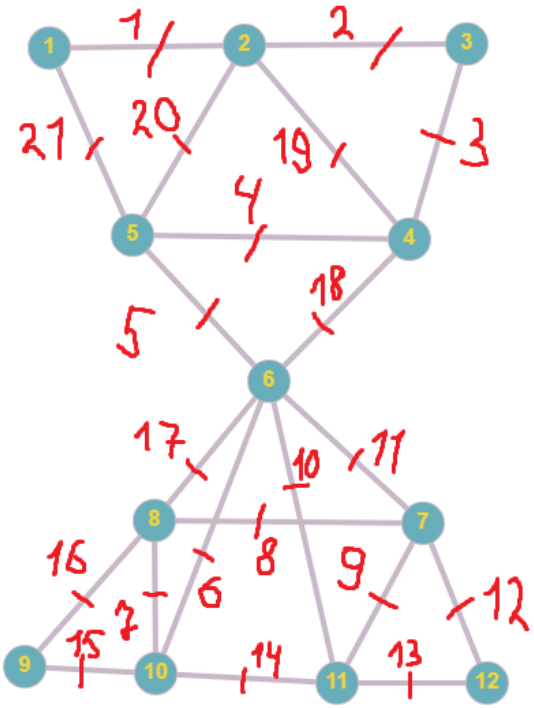


Рішення:

Метод Флері:

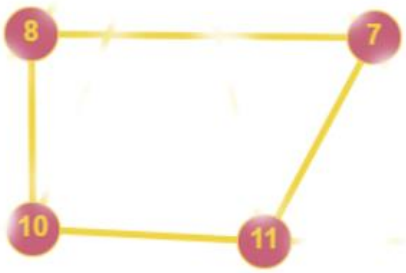
На малюнку позначено порядок проходу по графу. Після проходження викреслюємо вершину. Як висновок, граф містить ейлерів цикл.

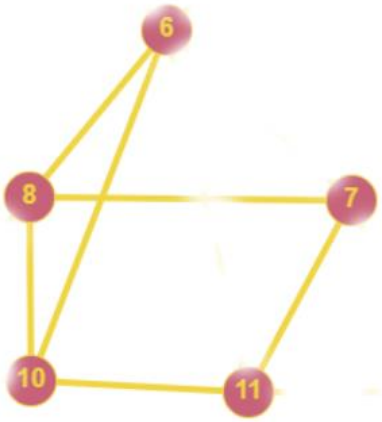
Ейлерів цикл:
1⇒2⇒3⇒4⇒5⇒6⇒10⇒8⇒7⇒11⇒6⇒7⇒12⇒11⇒10⇒9⇒8⇒6⇒4⇒2⇒5⇒1

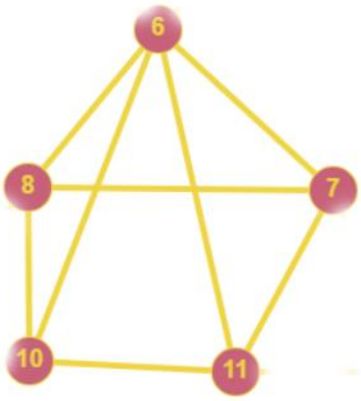


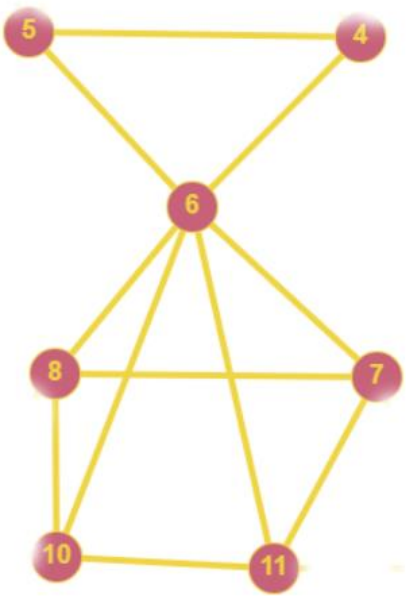
Метод елементарних циклів:

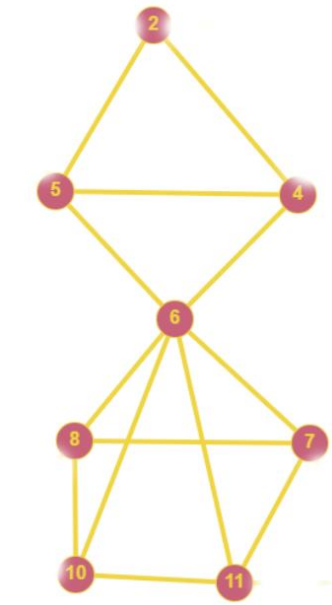
Об’єднуємо по-черзі цикли графа в один.

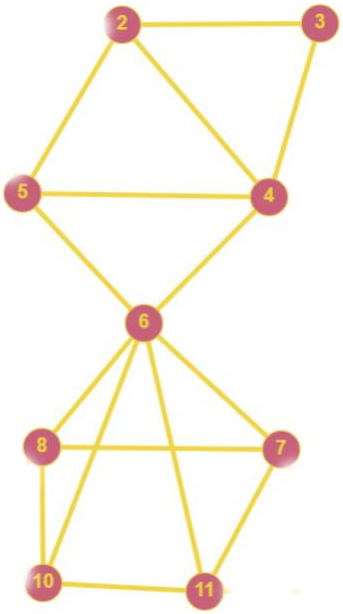


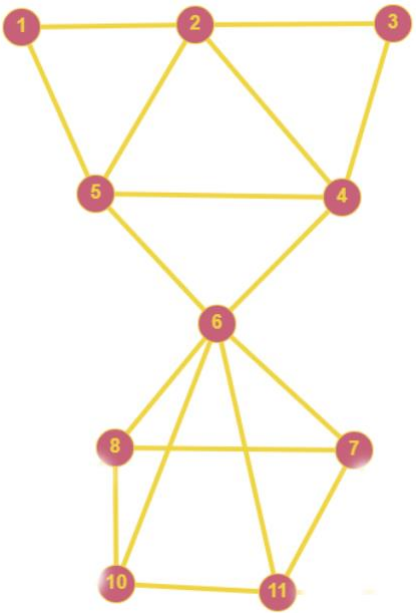


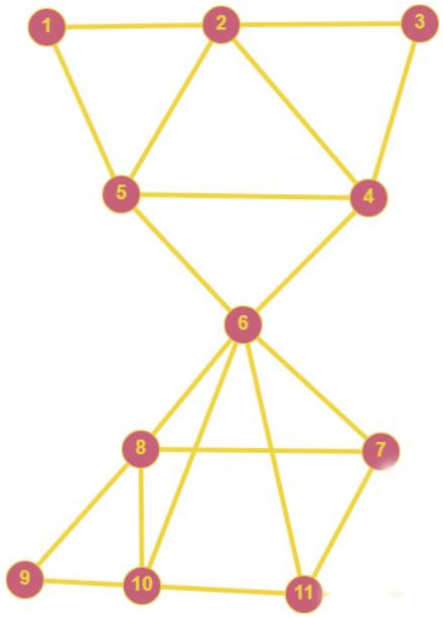


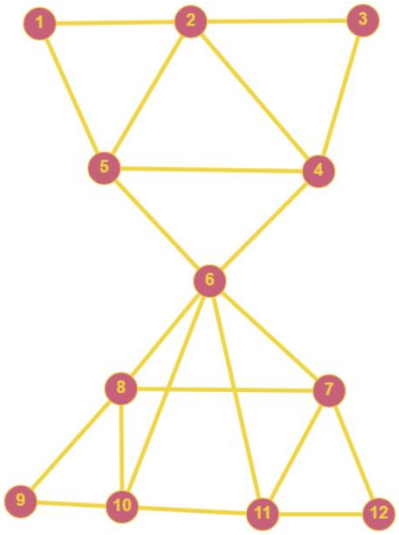



2
5
4
6
8
7
10
11


2
3
5
4
6
8
7
10
11


1
2
3
5
4
6
8
7
10
11





Ми отримали вихідний граф.

Висновок: граф містить ейлерів цикл.

Завдання 9.

Спростити формули (привести їх до скороченої ДНФ).

$$(x \vee \bar{y} \vee \bar{z})(\bar{x} \vee y)$$

Рішення:

¬∧∨

(x∨¬y∨¬z)(¬x∨y) =
(x¬x)∨(xy)∨(¬y¬x)∨(¬yy)∨(¬z¬x)∨(¬zy) =
F ∨(xy)∨(¬y¬x)∨(¬yy)∨(¬z¬x)∨(¬zy) =
(xy)∨(¬y¬x)∨(¬yy)∨(¬z¬x)∨(¬zy) =
(xy)∨(¬y¬x)∨ F ∨(¬z¬x)∨(¬zy) =
(xy)∨(¬y¬x)∨(¬z¬x)∨(¬zy)

Далі мінімізуємо за картою Карно

x\yz	00	01	11	10
0	1	1		1
1			1	1

xy

x\yz	00	01	11	10
0	1	1		1
1			1	1

¬x¬y

x\yz	00	01	11	10
0	1	1		1
1			1	1

¬x¬z

Відповідь: (xy)∨(¬y¬x)∨(¬z¬x)∨(¬zy) =
(xy)∨(¬x¬y)∨(¬x¬z)