**Міністерство освіти і науки України  
Національний технічний університет України  
«Київський політехнічний інститут імені Ігоря Сікорського»  
Факультет інформатики та обчислювальної техніки  
Кафедра обчислювальної техніки**

**Лабораторна робота № 2.4**

з дисципліни  
«Алгоритми і структури даних»

Виконав: Перевірила:

студент групи ІМ-22 Молчанова А. А.  
Кушнір Микола Миколайович  
номер у списку групи: 13

Київ 2023

***Постановка задачі***

***1.*** Представити напрямлений граф з заданими параметрами так само, як у

лабораторній роботі №3. Відміна: матриця ***А*** напрямленого графа за варіантом формується за функціями:

srand(n1 n2 n3 n4);

T = randm(n, n);

A = mulmr(( 1.0 – n3 \* 0.01 – n4 \* 0.01 − 0.3) \* T);

Перетворити граф у ненапрямлений.

***2.*** Визначити степені вершин напрямленого і ненапрямленого графів.

Програма на екран виводить степені усіх вершин ненапрямленого графу і

напівстепені виходу та заходу напрямленого графу. Визначити, чи граф є

однорідним та якщо так, то вказати степінь однорідності графу.

***3.*** Визначити всі висячі та ізольовані вершини. Програма на екран

виводить перелік усіх висячих та ізольованих вершин графу.

***4.*** Змінити матрицю графу за функцією

A = mulmr((1.0 – n3 \* 0.005 – n4 \* 0.005 − 0.27) \* T);

Створити програму для обчислення наступних результатів:

*1)* матриця суміжності;

*2)* півстепені вузлів;

*3)* всі шляхи довжини 2 і 3;

*4)* матриця досяжності;

*5)* компоненти сильної зв’язності;

*6)* матриця зв’язності;

*7)* граф конденсації.

Шляхи довжиною **2** і **3** слід шукати за матрицями ***А2*** і ***А3***, відповідно. Матриця

досяжності та компоненти сильної зв’язності слід шукати за допомогою

операції транзитивного замикання.

***Завдання для варіанту 13***

* *n1* = **2**;
* *n2* = **2**;
* *n3* = **1**;
* *n4* = **3**;

Число вершин *n*: 10 + **1** = **11**.

Розміщення вершин: **прямокутником (квадратом)**.

Формування матриці ***А***:

srand(**2** **2** **1** **3**);

T = randm(**11**, **11**);

A = mulmr(( 1.0 – **1.0** \* 0.01 – **3.0** \* 0.01 − 0.3) \* T);

Формування зміненої матриці:

A = mulmr((1.0 – **1.0** \* 0.005 – **3.0** \* 0.005 − 0.27) \* T).

[***Посилання на репозиторій з лабораторною роботою***](https://github.com/kushnirko/asd-labs/tree/main/semester-2/lab-2.4)

***Текст програми***

***Вміст файлу Configurations.h***

*/\*  
 \* Список скорочень, застосованих для найменування деяких ідентифікаторів  
 \* d або D => directed (напрямлений)  
 \* u або U => undirected (ненапрямлений)  
 \* m або M => modified (для позначення модифікованої матриці)  
 \* l або L => leaf (для висячих вершин)  
 \* i або I => isolated (для ізольованих вершин)  
 \*/  
  
/\* N1N2 - номер групи, N3N4 - порядковий номер у списку групи \*/*#define **N1** 2  
#define **N2** 2  
#define **N3** 1  
#define **N4** 3  
*/\* Кількість рядків і стовпців матриць суміжності графів \*/*#define **N** (10 + **N3**)  
*/\* Для позначення осей координат (позиції елементів записані у векторах) \*/*#define **x** 0  
#define **y** 1  
*/\* Значення, що використовуються в обчисленнях \*/*#define **PI** 3.1415926536  
#define **SQRT\_2** 1.4142135624  
  
#define **VERTEX\_RADIUS** 40  
#define **LOOP\_RADIUS** (5 \* **VERTEX\_RADIUS** / 4)  
#define **ONE\_STEP\_LENGTH** (9 \* **VERTEX\_RADIUS** / 2) */\* Найменша відстань між вершинами графа \*/*#define **MAX\_ONE\_STEP\_LENGTH** (3 \* **ONE\_STEP\_LENGTH** / 2)  
#define **WINDOW\_BORDER\_OFFSET** (2 \* **LOOP\_RADIUS** + 10)  
  
const wchar\_t \*vertices\_names[] =  
{  
 L"1", L"2", L"3",  
 L"4", L"5", L"6",  
 L"7", L"8", L"9",  
 L"10", L"11",  
};  
  
#define **GRAPH\_WIDTH** ((int) ((double) (**N** - 5) \* 0.5) \* **ONE\_STEP\_LENGTH**)  
#define **GRAPH\_HEIGHT** (3 \* **ONE\_STEP\_LENGTH**)  
*/\* Щоб граф коректно відобразився у вікні, його висота має бути сталою \*/*const int min\_coords[] =  
{  
 **WINDOW\_BORDER\_OFFSET**,  
 **WINDOW\_BORDER\_OFFSET**};  
const int max\_coords[] =  
{  
 **WINDOW\_BORDER\_OFFSET** + **GRAPH\_WIDTH**,  
 **WINDOW\_BORDER\_OFFSET** + **GRAPH\_HEIGHT**};  
  
const int vertex\_print\_offset[] = { 5, 8 };  
  
*/\* Визначають зміщення центру ребер для огинання вершин або вже намальованих ребер \*/*const double edge\_center\_offset\_dividers[] = { 1, 3.6, 4.5, 6.5, 7 };

***Вміст файлу DrawingDataSetter.h***

#include <math.h>  
#include "Configurations.h"  
#include "WorkWithMatrices.h"  
  
typedef struct DrawingData  
{  
 int edge\_type;  
 int start[2];  
 int center[2];  
 int end[2];  
 double angle;  
 int arrow\_end[2];  
} draw\_data;  
  
*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ПЕРЕЛІК ФУНКЦІЙ \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
/\*\*\*\*\* Функції, що виконують основні обчислення \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*int \*\*SetVerticesCoords (int n);  
draw\_data SetEdgeDrawData (int v1, int v2, int is\_directed, int \*\*coords, int drawn\_lines[**N**][**N**]);  
*/\*\*\*\*\* Допоміжні функції \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*int CheckGraphType (int n, int \*\*graph\_matrix);  
double Pow2 (int value);  
double GetDistance (const int \*v1\_pos, const int \*v2\_pos);  
double ConvertDegreeToRad (double degree\_value);  
*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*int \*\*SetVerticesCoords(int n)  
{  
 int \*\*coords = Create2dIntArr(n, n);  
 int current\_pos[2] = { min\_coords[**x**], min\_coords[**y**] };  
 for (int i = 0; i < n; i++)  
 {  
 coords[i][**x**] = current\_pos[**x**];  
 coords[i][**y**] = current\_pos[**y**];  
 if (current\_pos[**x**] < max\_coords[**x**] && current\_pos[**y**] == min\_coords[**y**])  
 current\_pos[**x**] += **ONE\_STEP\_LENGTH**;  
 else if (current\_pos[**y**] < max\_coords[**y**] && current\_pos[**x**] == max\_coords[**x**])  
 current\_pos[**y**] += **ONE\_STEP\_LENGTH**;  
 else if (current\_pos[**x**] > min\_coords[**x**] && current\_pos[**y**] == max\_coords[**y**])  
 current\_pos[**x**] -= **ONE\_STEP\_LENGTH**;  
 else if (current\_pos[**y**] > min\_coords[**y**] && current\_pos[**x**] == min\_coords[**x**])  
 current\_pos[**y**] -= **GRAPH\_HEIGHT** / (3 - n % 2);  
 */\*  
 \* Якщо к-сть вершин парна та більша ніж 10, то з  
 \* лівого боку буде розміщено дві вершини,  
 \* а якщо непарна - 1  
 \*/* }  
 return coords;  
}  
  
draw\_data SetEdgeDrawData(int v1, int v2, int is\_directed, int \*\*coords, int drawn\_lines[**N**][**N**])  
{  
 draw\_data data;  
 data.edge\_type = 1;  
 data.start[**x**] = coords[v1][**x**];  
 data.start[**y**] = coords[v1][**y**];  
 data.end[**x**] = coords[v2][**x**];  
 data.end[**y**] = coords[v2][**y**];  
 data.center[**x**] = (data.start[**x**] + data.end[**x**]) / 2;  
 data.center[**y**] = (data.start[**y**] + data.end[**y**]) / 2;  
 int dx = data.end[**x**] - data.start[**x**];  
 int dy = data.end[**y**] - data.start[**y**];  
 int index = (int) (GetDistance(data.start, data.end) / **ONE\_STEP\_LENGTH**);  
 int center\_offset[2] =  
 {  
 abs((int) (dy / edge\_center\_offset\_dividers[index])),  
 abs((int) (dx / edge\_center\_offset\_dividers[index]))  
 };  
 int is\_drawn = 0;  
 if (drawn\_lines[v1][v2] == 1 && drawn\_lines[v2][v1] == 1)  
 {  
 is\_drawn = 1;  
 if (!is\_directed)  
 {  
 data.edge\_type = 0;  
 return data;  
 }  
 else  
 {  
 data.center[**x**] += center\_offset[**x**];  
 data.center[**y**] += center\_offset[**y**];  
 }  
 }  
 else  
 drawn\_lines[v1][v2] = drawn\_lines[v2][v1] = 1;  
 int variable\_delta, static\_coord;  
 if (v1 == v2)  
 {  
 data.edge\_type = 2;  
 int loop\_offset\_direction[2] = { 0 };  
 int arrow\_direction[2] = { 0 };  
 if (data.center[**x**] > min\_coords[**x**] && data.center[**y**] == min\_coords[**y**])  
 {  
 --loop\_offset\_direction[**x**];  
 --loop\_offset\_direction[**y**];  
 --arrow\_direction[**y**];  
 data.angle = ConvertDegreeToRad(-87);  
 }  
 else if (data.center[**x**] == max\_coords[**x**] && data.center[**y**] > min\_coords[**y**])  
 {  
 ++loop\_offset\_direction[**x**];  
 --loop\_offset\_direction[**y**];  
 ++arrow\_direction[**x**];  
 data.angle = ConvertDegreeToRad(183);  
 }  
 else if (data.center[**x**] < max\_coords[**x**] && data.center[**y**] == max\_coords[**y**])  
 {  
 ++loop\_offset\_direction[**x**];  
 ++loop\_offset\_direction[**y**];  
 ++arrow\_direction[**y**];  
 data.angle = ConvertDegreeToRad(93);  
 }  
 else if (data.center[**x**] == min\_coords[**x**] && data.center[**y**] < max\_coords[**y**])  
 {  
 --loop\_offset\_direction[**x**];  
 ++loop\_offset\_direction[**y**];  
 --arrow\_direction[**x**];  
 data.angle = ConvertDegreeToRad(3);  
 }  
 int loop\_center\_offset = (int)round(  
 (**VERTEX\_RADIUS** \* **SQRT\_2** / 2 +  
 sqrt(Pow2(**LOOP\_RADIUS**) - Pow2(**VERTEX\_RADIUS**) / 2))  
 / **SQRT\_2**);  
 data.center[**x**] += loop\_center\_offset \* loop\_offset\_direction[**x**];  
 data.center[**y**] += loop\_center\_offset \* loop\_offset\_direction[**y**];  
 data.arrow\_end[**x**] = data.end[**x**] + **VERTEX\_RADIUS** \* arrow\_direction[**x**];  
 data.arrow\_end[**y**] = data.end[**y**] + **VERTEX\_RADIUS** \* arrow\_direction[**y**];  
 return data;  
 }  
 else if ((dx == 0 ? (variable\_delta = dy, static\_coord = **x**, 1) : 0) ||  
 (dy == 0 ? (variable\_delta = dx, static\_coord = **y**, 1) : 0 ))  
 {  
 if (abs(variable\_delta) > **MAX\_ONE\_STEP\_LENGTH**)  
 {  
 if (data.start[static\_coord] == min\_coords[static\_coord])  
 {  
 if (!is\_drawn)  
 data.center[static\_coord] -= center\_offset[static\_coord];  
 }  
 else if (data.start[static\_coord] == max\_coords[static\_coord])  
 {  
 if (!is\_drawn)  
 data.center[static\_coord] += center\_offset[static\_coord];  
 else  
 data.center[static\_coord] -= 2 \* center\_offset[static\_coord];  
 }  
 }  
 }  
 else if (dx == dy && is\_drawn)  
 data.center[**x**] -= center\_offset[**x**];  
 else  
 {  
 if (is\_drawn)  
 {  
 int graph\_center[] =  
 {  
 ((min\_coords[**x**] + max\_coords[**x**]) / 2),  
 ((min\_coords[**y**] + max\_coords[**y**]) / 2)  
 };  
 int alternative\_center[] =  
 {  
 (data.center[**x**] - 2 \* center\_offset[**x**]),  
 (data.center[**y**] - 2 \* center\_offset[**y**])  
 };  
 if (GetDistance(data.center, graph\_center) >  
 GetDistance(alternative\_center, graph\_center))  
 {  
 data.center[**x**] = alternative\_center[**x**];  
 data.center[**y**] = alternative\_center[**y**];  
 }  
 }  
 }  
 if (is\_directed)  
 {  
 int new\_dx = data.end[**x**] - data.center[**x**];  
 int new\_dy = data.end[**y**] - data.center[**y**];  
 double hypotenuse = GetDistance(data.center, data.end);  
 if (new\_dx >= 0 && new\_dy >= 0)  
 data.angle = acos(abs(new\_dx) / hypotenuse) \* -1;  
 else if (new\_dx >= 0 && new\_dy < 0)  
 data.angle = acos(abs(new\_dx) / hypotenuse);  
 else if (new\_dx < 0 && new\_dy >= 0)  
 data.angle = (**PI** - acos(abs(new\_dx) / hypotenuse)) \* -1;  
 else if (new\_dx < 0 && new\_dy < 0)  
 data.angle = **PI** - acos(abs(new\_dx) / hypotenuse);  
 data.arrow\_end[**x**] = data.end[**x**] - (int) round((double) **VERTEX\_RADIUS** \* cos(data.angle));  
 data.arrow\_end[**y**] = data.end[**y**] + (int) round((double) **VERTEX\_RADIUS** \* sin(data.angle));  
 }  
 return data;  
}  
  
int CheckGraphType(int n, int \*\*graph\_matrix)  
{  
 int is\_directed = 0;  
 int i, j;  
 for (i = 0; i < n; i++)  
 {  
 for (j = 0; j < n; j++)  
 {  
 if (graph\_matrix[i][j] != graph\_matrix[j][i])  
 {  
 is\_directed = 1;  
 break;  
 }  
 }  
 }  
 return is\_directed;  
}  
  
double Pow2(int value)  
{  
 return (double)(value \* value);  
}  
  
double GetDistance(const int \*v1\_pos, const int \*v2\_pos)  
{  
 int a = v2\_pos[**x**] - v1\_pos[**x**];  
 int b = v2\_pos[**y**] - v1\_pos[**y**];  
 return sqrt(Pow2(a) + Pow2(b));  
}  
  
double ConvertDegreeToRad(double degree\_value)  
{  
 return **PI** \* degree\_value / 180.0;  
}

***Вміст файлу GraphPainter.h***

#include "DrawingDataSetter.h"  
  
void DrawGraph(int n,  
 int \*\*graph\_matrix,  
 int \*\*coords,  
 HPEN e\_pen,  
 HBRUSH v\_brush,  
 HPEN v\_pen,  
 HDC hdc)  
{  
 */\* Зображаємо ребра \*/* int i, j;  
 int is\_directed = CheckGraphType(n, graph\_matrix);  
 int drawn\_lines[**N**][**N**] = { 0 };  
 SelectObject(hdc, e\_pen);  
 SelectObject(hdc, GetStockObject(**NULL\_BRUSH**));  
 for (i = 0; i < n; i++)  
 {  
 for (j = 0; j < n; j++)  
 {  
 if (graph\_matrix[i][j] == 1)  
 {  
 draw\_data data = SetEdgeDrawData(i, j, is\_directed, coords, drawn\_lines);  
 switch (data.edge\_type)  
 {  
 case 0:  
 break;  
 case 1:  
 MoveToEx(hdc, data.start[**x**], data.start[**y**], **NULL**);  
 LineTo(hdc, data.center[**x**], data.center[**y**]);  
 MoveToEx(hdc, data.center[**x**], data.center[**y**], **NULL**);  
 LineTo(hdc, data.end[**x**], data.end[**y**]);  
 break;  
 case 2:  
 Ellipse(hdc,  
 (data.center[**x**] - **LOOP\_RADIUS**),  
 (data.center[**y**] - **LOOP\_RADIUS**),  
 (data.center[**x**] + **LOOP\_RADIUS**),  
 (data.center[**y**] + **LOOP\_RADIUS**));  
 break;  
 }  
 if (is\_directed)  
 {  
 */\* Якщо граф напрямлений, малюємо стрілку \*/* double fi = **PI** - data.angle;  
 int leftLineEnd[2], rightLineEnd[2];  
 rightLineEnd[**x**] = data.arrow\_end[**x**] + (int) (30 \* cos(fi + 0.3));  
 rightLineEnd[**y**] = data.arrow\_end[**y**] + (int) (30 \* sin(fi + 0.3));  
 leftLineEnd[**x**] = data.arrow\_end[**x**] + (int) (30 \* cos(fi - 0.3));  
 leftLineEnd[**y**] = data.arrow\_end[**y**] + (int) (30 \* sin(fi - 0.3));  
 MoveToEx(hdc, leftLineEnd[**x**], leftLineEnd[**y**], **NULL**);  
 LineTo(hdc, data.arrow\_end[**x**], data.arrow\_end[**y**]);  
 LineTo(hdc, rightLineEnd[**x**], rightLineEnd[**y**]);  
 }  
 }  
 }  
 }  
 */\* Зображаємо вершини \*/* int left, top, right, bottom;  
 int print\_pos[2];  
 for (i = 0; i < n; i++)  
 {  
 left = (coords[i][**x**] - **VERTEX\_RADIUS**);  
 top = (coords[i][**y**] - **VERTEX\_RADIUS**);  
 right = (coords[i][**x**] + **VERTEX\_RADIUS**);  
 bottom = (coords[i][**y**] + **VERTEX\_RADIUS**);  
 if (i > 8)  
 print\_pos[**x**] = coords[i][**x**] - (int)(1.5 \* vertex\_print\_offset[**x**]);  
 */\* Ці елементи складаються з двох цифр, тому зміщення має бути більшим \*/* else  
 print\_pos[**x**] = coords[i][**x**] - vertex\_print\_offset[**x**];  
 print\_pos[**y**] = coords[i][**y**] - vertex\_print\_offset[**y**];  
 SelectObject(hdc, v\_brush);  
 Ellipse(hdc, left, top, right, bottom);  
 SelectObject(hdc, v\_pen);  
 Ellipse(hdc, left, top, right, bottom);  
 **TextOut**(hdc, print\_pos[**x**], print\_pos[**y**], vertices\_names[i], 2);  
 }  
}

***Вміст файлу PrimitiveTableOutput.h***

*/\* Типи лінії таблиці \*/*#define **FIRST\_LINE** 1  
#define **MIDDLE\_LINE** 2  
#define **LAST\_LINE** 3  
*/\* ASCII символи для зображення кожного типу лінії таблиці \*/*const int first\_line\_chars[] = {218, 196, 194, 191};  
const int middle\_line\_chars[] = {195, 196, 197, 180};  
const int last\_line\_chars[] = {192, 196, 193, 217};  
  
void PrintTableLine(int cols\_quantity, const int cols\_lengths[cols\_quantity], int line\_type)  
{  
 const int \*pointer;  
 switch (line\_type)  
 {  
 case **FIRST\_LINE** :  
 pointer = first\_line\_chars;  
 break;  
 case **MIDDLE\_LINE** :  
 pointer = middle\_line\_chars;  
 break;  
 case **LAST\_LINE** :  
 pointer = last\_line\_chars;  
 }  
 int symbols[4];  
 int i, j;  
 for (i = 0; i < 4; i++)  
 symbols[i] = pointer[i];  
 printf("%c", symbols[0]);  
 for (i = 0; i < cols\_quantity; i++)  
 {  
 for (j = 0; j < cols\_lengths[i]; j++)  
 printf("%c", symbols[1]);  
 if ((i + 1) != cols\_quantity)  
 printf("%c", symbols[2]);  
 }  
 printf("%c\n", symbols[3]);  
}

***Вміст файлу WorkWithMatrices.h***

#include <stdio.h>  
#include <stdlib.h>  
#include "PrimitiveTableOutput.h"  
  
*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ПЕРЕЛІК ФУНКЦІЙ \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
/\*\*\*\*\* Функції, що задані в умові лабораторної роботи \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*double \*\*randm (int n1, int n2);  
int \*\*mulmr (int n1, int n2, double \*\*matrix\_T, double coefficient);  
*/\*\*\*\*\* Функції, що виконують основні обчислення \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*int \*\*SymmetrizeMatrix (int n, int \*\*matrix\_A);  
int \*\*GetVerticesDegsOfDGraph (int n, int \*\*matrix\_A);  
void PrintDGraphHomogeneityDeg (int n, int \*\*vertices\_degs);  
void PrintLIVerticesOfDGraph (int n, int \*\*vertices\_degs);  
int \*GetVerticesDegsOfUGraph (int n, int \*\*graph\_matrix);  
void PrintUGraphHomogeneityDeg (int n, int \*vertices\_degs);  
void PrintLIVerticesOfUGraph (int n, int \*vertices\_degs);  
int \*\*MultSquareMatrices (int n, int \*\*matrix\_1, int \*\*matrix\_2);  
void PrintPathsWithLen2 (int n, int \*\*matrix\_pow1, int \*\*matrix\_pow2);  
void PrintPathsWithLen3 (int n, int \*\*matrix\_pow1, int \*\*matrix\_pow3);  
int \*\*MultSquareMatricesElemByElem (int n, int \*\*matrix\_1, int \*\*matrix\_2);  
int \*\*TransposeSquareMatrix (int n, int \*\*matrix);  
int \*\*SumSquareMatrices (int n, int \*\*matrix\_1, int \*\*matrix\_2);  
int \*\*GetConnectComponents (int n, int \*\*matrix\_S);  
int \*\*GetReachabilityMatrix (int n, int \*\*matrix\_A);  
int \*\*GetCondensationGraphMatrix (int n, int \*\*components, int \*\*matrix\_mA);  
*/\*\*\*\*\* Допоміжні функції \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*double RandInRange (double min, double max);  
void PrintDouble2dArr (int rows, int cols, double \*\*arr);  
int \*\*Create2dIntArr (int rows, int cols);  
void PrintInt2dArr (int rows, int cols, int \*lengths, char \*\*headings, int \*\*arr);  
void PrintIntArr (int arr\_length, int \*lengths, char \*heading, int \*arr);  
void PrintBooleanMatrix (int n, int \*\*matrix);

void PrintConnectComponents (int n, int \*\*components);  
void FreeInt2dArr (int rows, int \*\*arr);  
void FreeDouble2dArr (int rows, double \*\*arr);  
*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*double \*\*randm(int n1, int n2)  
{  
 double \*\*matrix\_T = (double \*\*) malloc(sizeof(double \*) \* n1);  
 int i, j;  
 for (i = 0; i < n1; i++)  
 {  
 matrix\_T[i] = (double \*) malloc(sizeof(double) \* n2);  
 for (j = 0; j < n2; j++)  
 matrix\_T[i][j] = RandInRange(0.0, 2.0);  
 }  
 return matrix\_T;  
}  
  
int \*\*mulmr(int n1, int n2, double \*\*matrix\_T, double coefficient)  
{  
 int \*\*matrix\_A = Create2dIntArr(n1, n2);  
 int i, j;  
 for (i = 0; i < n1; i++)  
 {  
 for (j = 0; j < n2; j++)  
 matrix\_A[i][j] = (int) (matrix\_T[i][j] \* coefficient);  
 }  
 return matrix\_A;  
}  
  
int \*\*SymmetrizeMatrix(int n, int \*\*matrix\_A)  
{  
 int \*\*matrix = Create2dIntArr(n, n);  
 int i, j;  
 for (i = 0; i < n; i++)  
 for (j = i; j < n; j++)  
 if (matrix\_A[i][j] == 1 || matrix\_A[j][i] == 1)  
 matrix[i][j] = matrix[j][i] = 1;  
 return matrix;  
}  
  
int \*\*GetVerticesDegsOfDGraph(int n, int \*\*matrix\_A)  
{  
 int \*\*degs = Create2dIntArr(n, 2);  
 */\*  
 \* degs[0-10][1] - напівстепінь виходу вершини;  
 \* degs[0-10][0] - напівстепінь заходу вершини;  
 \*/* int i, j;  
 for (i = 0; i < n; i++)  
 for (j = 0; j < n; j++)  
 {  
 if (matrix\_A[i][j])  
 degs[i][0]++;  
 if (matrix\_A[j][i])  
 degs[i][1]++;  
 }  
 return degs;  
}  
  
void PrintDGraphHomogeneityDeg(int n, int \*\*vertices\_degs)  
{  
 int r = 0, i;  
 for (i = 1; i < n; i++)  
 if (vertices\_degs[0][0] != vertices\_degs[i][0])  
 {  
 r = -1;  
 break;  
 }  
 if (r == -1)  
 printf("Graph isn't homogeneous\n");  
 else  
 {  
 r = vertices\_degs[0][0];  
 printf("Graph homogeneity degree: %d\n", r);  
 }  
}  
  
void PrintLIVerticesOfDGraph(int n, int \*\*vertices\_degs)  
{  
 int i;  
 int leaf\_ct = 0, isolated\_ct = 0;  
 printf("Leaf vertices : ");  
 for (i = 0; i < n; i++)  
 if ((vertices\_degs[i][0] == 1 && vertices\_degs[i][1] == 0) ||  
 (vertices\_degs[i][0] == 0 && vertices\_degs[i][1] == 1))  
 {  
 printf("%d ", (i + 1));  
 leaf\_ct++;  
 }  
 if (!leaf\_ct)  
 printf("graph hasn't leaf vertices");  
 printf("\nIsolated vertices: ");  
 for (i = 0; i < n; i++)  
 if (vertices\_degs[i][0] == 0 &&  
 vertices\_degs[i][1] == 0)  
 {  
 printf("%d ", (i + 1));  
 isolated\_ct++;  
 }  
 if (!isolated\_ct)  
 printf("graph hasn't isolated vertices");  
 printf("\n");  
}  
  
int \*GetVerticesDegsOfUGraph(int n, int \*\*graph\_matrix)  
{  
 int \*degs = (int \*) calloc(n, sizeof(int));  
 */\* degs[0-10] - степінь вершини \*/* int i, j;  
 for (i = 0; i < n; i++)  
 for (j = 0; j < n; j++)  
 if (graph\_matrix[i][j])  
 (i == j) ? (degs[i] += 2) : ++degs[i];  
 return degs;  
}  
  
void PrintUGraphHomogeneityDeg(int n, int \*vertices\_degs)  
{  
 int r = 0, i;  
 for (i = 1; i < n; i++)  
 if (vertices\_degs[0] != vertices\_degs[i])  
 {  
 r = -1;  
 break;  
 }  
 if (r == -1)  
 printf("Graph isn't homogeneous\n");  
 else  
 {  
 r = vertices\_degs[0];  
 printf("Graph homogeneity degree: %d\n", r);  
 }  
}  
  
void PrintLIVerticesOfUGraph(int n, int \*vertices\_degs)  
{  
 int i;  
 int leaf\_ct = 0, isolated\_ct = 0;  
 printf("Leaf vertices : ");  
 for (i = 0; i < n; i++)  
 if (vertices\_degs[i] == 1)  
 {  
 printf("%d ", (i + 1));  
 leaf\_ct++;  
 }  
 if (!leaf\_ct)  
 printf("graph hasn't leaf vertices");  
 printf("\nIsolated vertices: ");  
 for (i = 0; i < n; i++)  
 if (vertices\_degs[i] == 0)  
 {  
 printf("%d ", (i + 1));  
 isolated\_ct++;  
 }  
 if (!isolated\_ct)  
 printf("graph hasn't isolated vertices");  
 printf("\n");  
}  
  
int \*\*MultSquareMatrices(int n, int \*\*matrix\_1, int \*\*matrix\_2)  
{  
 int \*\*result = Create2dIntArr(n, n);  
 int i, j, k;  
 for (i = 0; i < n; i++)  
 for (j = 0; j < n; j++)  
 for (k = 0; k < n; k++)  
 result[i][j] += matrix\_1[i][k] \* matrix\_2[k][j];  
 return result;  
}  
  
int \*\*MultSquareMatricesElemByElem(int n, int \*\*matrix\_1, int \*\*matrix\_2)  
{  
 int \*\*result = Create2dIntArr(n, n);  
 int i, j;  
 for (i = 0; i < n; i++)  
 for (j = 0; j < n; j++)  
 result[i][j] = matrix\_1[i][j] \* matrix\_2[i][j];  
 return result;  
}  
  
int \*\*TransposeSquareMatrix(int n, int \*\*matrix)

{  
 int \*\*result = Create2dIntArr(n, n);  
 int i, j;  
 for (i = 0; i < n; i++)  
 for (j = i; j < n; j++)  
 {  
 result[i][j] = matrix[j][i];  
 result[j][i] = matrix[i][j];  
 }  
 return result;  
}  
  
int \*\*SumSquareMatrices(int n, int\*\* matrix\_1, int \*\*matrix\_2)  
{  
 int \*\*result = Create2dIntArr(n, n);  
 int i, j;  
 for (i = 0; i < n; i++)  
 for (j = 0; j < n; j++)  
 result[i][j] = matrix\_1[i][j] + matrix\_2[i][j];  
 return result;  
}  
  
void PrintPathsWithLen2(int n, int \*\*matrix\_pow1, int \*\*matrix\_pow2)  
{  
 int cols\_quantity = 4;  
 int cols\_lengths[] = { 11, 7, 7, 7 };  
 PrintTableLine(cols\_quantity, cols\_lengths, 1);  
 printf("%c No. %c v-1 %c v-2 %c v-3 %c\n",  
 179, 179, 179, 179, 179);  
 int path\_counter = 1;  
 int path[3];  
 int i, j, k;  
 for (i = 0; i < n; i++)  
 {  
 for (j = 0; j < n; j++)  
 {  
 if (matrix\_pow2[i][j])  
 {  
 path[0] = i;  
 path[2] = j;  
 for (k = 0; k < n; k++)  
 {  
 if (matrix\_pow1[i][k] && matrix\_pow1[k][j])  
 {  
 path[1] = k;  
 PrintTableLine(cols\_quantity, cols\_lengths, 2);  
 printf("%c %5d %c %2d %c %2d %c %2d %c\n",  
 179, path\_counter,  
 179, (path[0] + 1),  
 179, (path[1] + 1),  
 179, (path[2] + 1), 179);  
 path\_counter++;  
 }  
 }  
 }  
 }  
 }  
 PrintTableLine(cols\_quantity, cols\_lengths, 3);  
}  
  
void PrintPathsWithLen3(int n, int \*\*matrix\_pow1, int \*\*matrix\_pow3)  
{  
 int cols\_quantity = 5;  
 int cols\_lengths[] = { 11, 7, 7, 7, 7 };  
 PrintTableLine(cols\_quantity, cols\_lengths, 1);  
 printf("%c No. %c v-1 %c v-2 %c v-3 %c v-4 %c\n",  
 179, 179, 179, 179, 179, 179);  
 int path\_counter = 1;  
 int path[4];  
 int i, j, k, l;  
 for (i = 0; i < n; i++)  
 {  
 for (j = 0; j < n; j++)  
 {  
 if (matrix\_pow3[i][j])  
 {  
 path[0] = i;  
 path[3] = j;  
 for (k = 0; k < n; k++)  
 {  
 if (matrix\_pow1[i][k])  
 {  
 for (l = 0; l < n; l++)  
 {  
 if (matrix\_pow1[k][l] && matrix\_pow1[l][j])  
 {  
 path[1] = k;  
 path[2] = l;  
 PrintTableLine(cols\_quantity, cols\_lengths, 2);  
 printf("%c %5d %c %2d %c %2d %c %2d %c %2d %c\n",  
 179, path\_counter,  
 179, (path[0] + 1),  
 179, (path[1] + 1),  
 179, (path[2] + 1),  
 179, (path[3] + 1), 179);  
 path\_counter++;  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
 PrintTableLine(cols\_quantity, cols\_lengths, 3);  
}  
  
int \*\*GetConnectComponents(int n, int \*\*matrix\_S)  
{  
 int \*\*components = Create2dIntArr(n, (n + 1));  
 */\*  
 \* Максимальне число компонент зв'язності = к-сті вершин = n;  
 \* максимальне число вершин у компоненті зв'язності = к-сті вершин = n;  
 \* останній елемент у рядку вказуватиме на першу вільну позицію у components[0-10][]  
 \*/* int counter = 0; */\* лічильник вказуватиме на першу вільну позицію у components[] \*/* int \*included\_vertices = (int \*) calloc(n, sizeof(int));  
 int i, j;  
 for (i = 0; i < n; i++)  
 {  
 if (!included\_vertices[i])  
 {  
 for (j = 0; j < n; j++)  
 {  
 if (matrix\_S[i][j] != 0)  
 {  
 included\_vertices[j]++;  
 components[counter][components[counter][n]] = j + 1;  
 components[counter][n]++;  
 }  
 }  
 counter++;  
 }  
 }  
 free(included\_vertices);  
 return components;  
}  
  
int \*\*GetReachabilityMatrix(int n, int \*\*matrix\_A)  
{  
 int \*\*matrix = Create2dIntArr(n, n);  
 int \*\*t\_closing = Create2dIntArr(n, n);  
 int i, j;  
 for (i = 0; i < n; i++)  
 {  
 matrix[i][i] = 1; */\* додавання одиничної матриці \*/* for (j = 0; j < n; j++)  
 t\_closing[i][j] = matrix\_A[i][j];  
 }  
 for (i = 0; i < n; i++)  
 {  
 matrix = SumSquareMatrices(n, matrix, t\_closing);  
 t\_closing = MultSquareMatrices(n, t\_closing, matrix\_A);  
 }  
 for (i = 0; i < n; i++)  
 {  
 for (j = 0; j < n; j++)  
 if (matrix[i][j] != 0)  
 matrix[i][j] = 1; */\* застосування булевого відображення \*/* }  
 FreeInt2dArr(n, t\_closing);  
 return matrix;  
}  
  
int \*\*GetCondensationGraphMatrix(int n, int \*\*components, int \*\*matrix\_mA)  
{  
 int index = 0;  
 while (index < n && components[index][n] != 0)  
 index++;  
 int \*\*graph\_matrix = Create2dIntArr((index + 1), (index + 1));  
 int i, j, k, l;  
 for (i = 0; i < index; i++)  
 {  
 for (j = 0; j < index; j++)  
 {  
 if (i != j)  
 {  
 for (k = 0; k < components[i][n]; k++)  
 {  
 for (l = 0; l < components[j][n]; l++)  
 {  
 if (matrix\_mA[components[i][k] - 1][components[j][l] - 1])  
 {  
 */\* Пошук ребра, що з'єднує компоненти \*/* graph\_matrix[i][j] = 1;  
 break;  
 }  
 }  
 if (graph\_matrix[i][j] == 1)  
 break;  
 }  
 }  
 }  
 }  
 return graph\_matrix;  
}  
  
double RandInRange(double min, double max)  
{  
 double random = (double) rand() / **RAND\_MAX**;  
 double range = max - min;  
 return min + range \* random;  
}

void PrintDouble2dArr(int rows, int cols, double \*\*arr)  
{  
 int i, j;  
 for (i = 0; i < rows; i++)  
 {  
 for (j = 0; j < cols; j++)  
 printf("%lf ", arr[i][j]);  
 printf("\n");  
 }  
}  
  
int \*\*Create2dIntArr(int rows, int cols)  
{  
 int \*\*arr = (int \*\*) malloc(sizeof(int \*) \* rows);  
 for (int i = 0; i < rows; i++)  
 arr[i] = (int \*) calloc(cols, sizeof(int));  
 return arr;  
}  
  
void PrintInt2dArr(int rows, int cols,  
 int \*lengths, char \*\*headings,  
 int \*\*arr)  
{  
 PrintTableLine((cols + 1), lengths, 1);  
 int i, j, k;  
 printf("%c No. %c", 179, 179);  
 for (i = 0; i < cols; i++)  
 printf(" %s %c", headings[i], 179);  
 printf("\n");  
 int space\_before, space\_after;  
 for (i = 0; i < rows; i++)  
 {  
 PrintTableLine((cols + 1), lengths, 2);  
 printf("%c %2d %c", 179, (i + 1), 179);  
 for (j = 0; j < cols; j++)  
 {  
 space\_before = (int) (lengths[j + 1] / 2) - 1;  
 space\_after = space\_before + (lengths[j + 1] % 2);  
 for (k = 0; k < space\_before; k++)  
 printf(" ");  
 printf("%2d", arr[i][j]);  
 for (k = 0; k < space\_after; k++)  
 printf(" ");  
 printf("%c", 179);  
 }  
 printf("\n");  
 }  
 PrintTableLine((cols + 1), lengths, 3);  
}  
  
void PrintIntArr(int arr\_length,  
 int \*lengths, char \*heading,  
 int \*arr)  
{  
 PrintTableLine(2, lengths, 1);  
 int i, j;  
 printf("%c No. %c %s %c\n", 179, 179, heading, 179);  
 int space\_before = (int) (lengths[1] / 2) - 1;  
 int space\_after = space\_before + (lengths[1] % 2);  
 for (i = 0; i < arr\_length; i++)  
 {  
 PrintTableLine(2, lengths, 2);  
 printf("%c %2d %c", 179, (i + 1), 179);  
 for (j = 0; j < space\_before; j++)  
 printf(" ");  
 printf("%2d", arr[i]);  
 for (j = 0; j < space\_after; j++)  
 printf(" ");  
 printf("%c\n", 179);  
 }  
 PrintTableLine(2, lengths, 3);  
}  
  
void PrintBooleanMatrix(int n, int \*\*matrix)  
{  
 int i, j;  
 int cols\_quantity = n;  
 int \*cols\_lengths = (int \*) calloc(cols\_quantity, sizeof(int));  
 printf(" %c", 179);  
 for (i = 0; i < n; i++)  
 {  
 cols\_lengths[i] = 3;  
 printf("%2d %c", (i + 1), 179);  
 }  
 printf("\n");  
 for (i = 0; i < n; i++)  
 {  
 printf(" %c%c%c", 196, 196, 196);  
 PrintTableLine(cols\_quantity, cols\_lengths, 2);  
 printf(" %2d %c", (i + 1), 179);  
 for (j = 0; j < n; j++)  
 printf("%2d %c", matrix[i][j], 179);  
 printf("\n");  
 }  
 printf(" %c%c%c", 196, 196, 196);  
 PrintTableLine(cols\_quantity, cols\_lengths, 3);  
 free(cols\_lengths);  
}

void PrintConnectComponents(int n, int \*\*components)  
{  
 int cols\_lengths[] = { 5, 44 };  
 PrintTableLine(2, cols\_lengths, 1);  
 printf("%c No. %c Vertices in component %c\n",  
 179, 179, 179);  
 int components\_ct = 0;  
 int i;  
 while (components\_ct < n && components[components\_ct][n] != 0)  
 {  
 PrintTableLine(2, cols\_lengths, 2);  
 printf("%c %2d %c", 179, (components\_ct + 1), 179);  
 for (i = 0; i < n; i++)  
 {  
 if (components[components\_ct][i] != 0)  
 printf(" %2d ", components[components\_ct][i]);  
 else  
 printf(" ");  
 }  
 printf("%c\n", 179);  
 components\_ct++;  
 }  
 PrintTableLine(2, cols\_lengths, 3);  
}

void FreeInt2dArr(int rows, int \*\*arr)  
{  
 for (int i = 0; i < rows; i++)  
 free(arr[i]);  
 free(arr);  
}  
  
void FreeDouble2dArr(int rows, double \*\*arr)  
{  
 for (int i = 0; i < rows; i++)  
 free(arr[i]);  
 free(arr);  
}

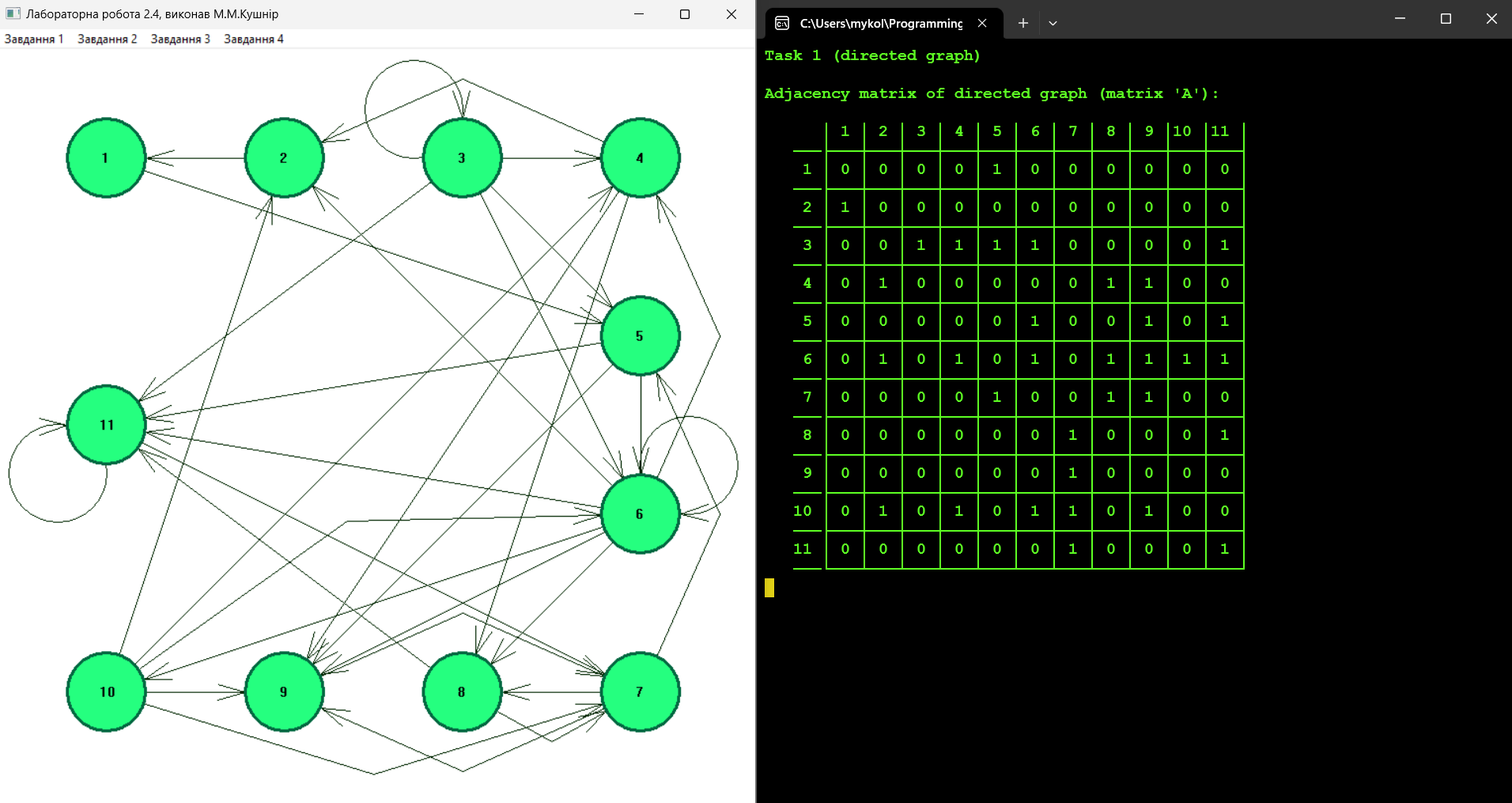
***Вміст файлу main.c***

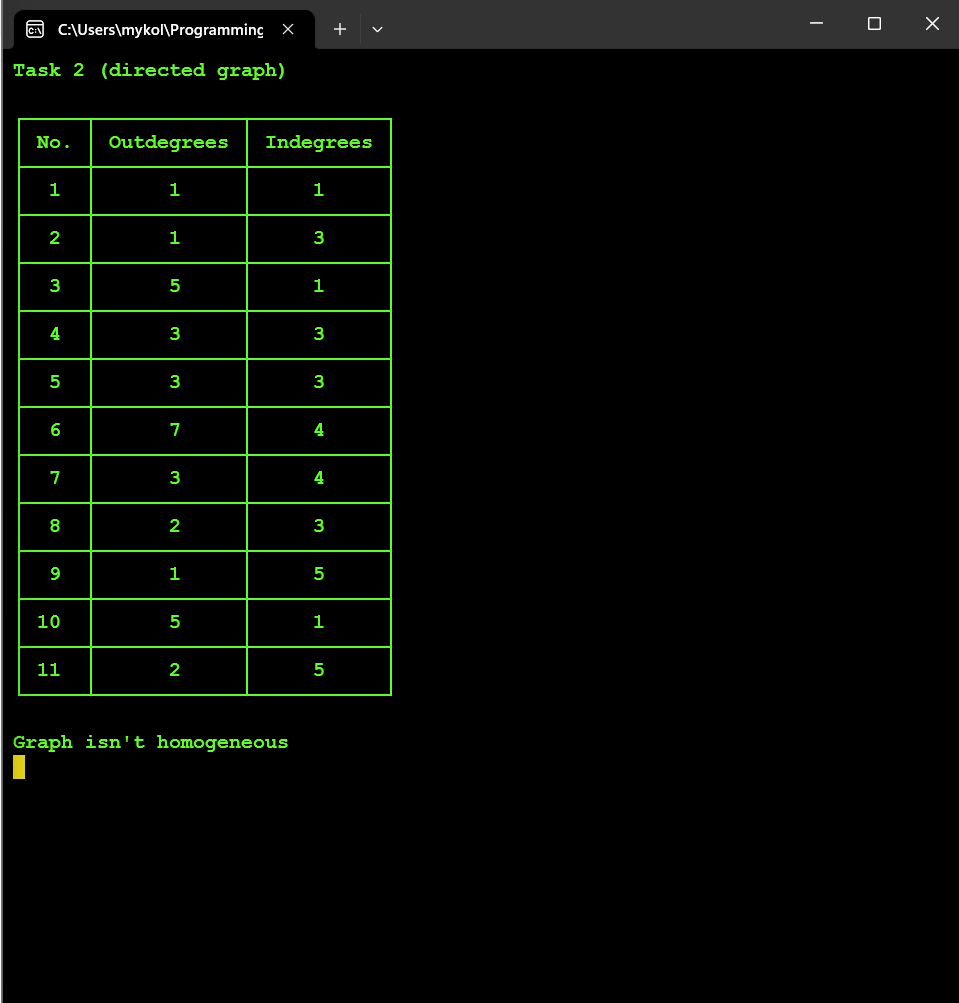
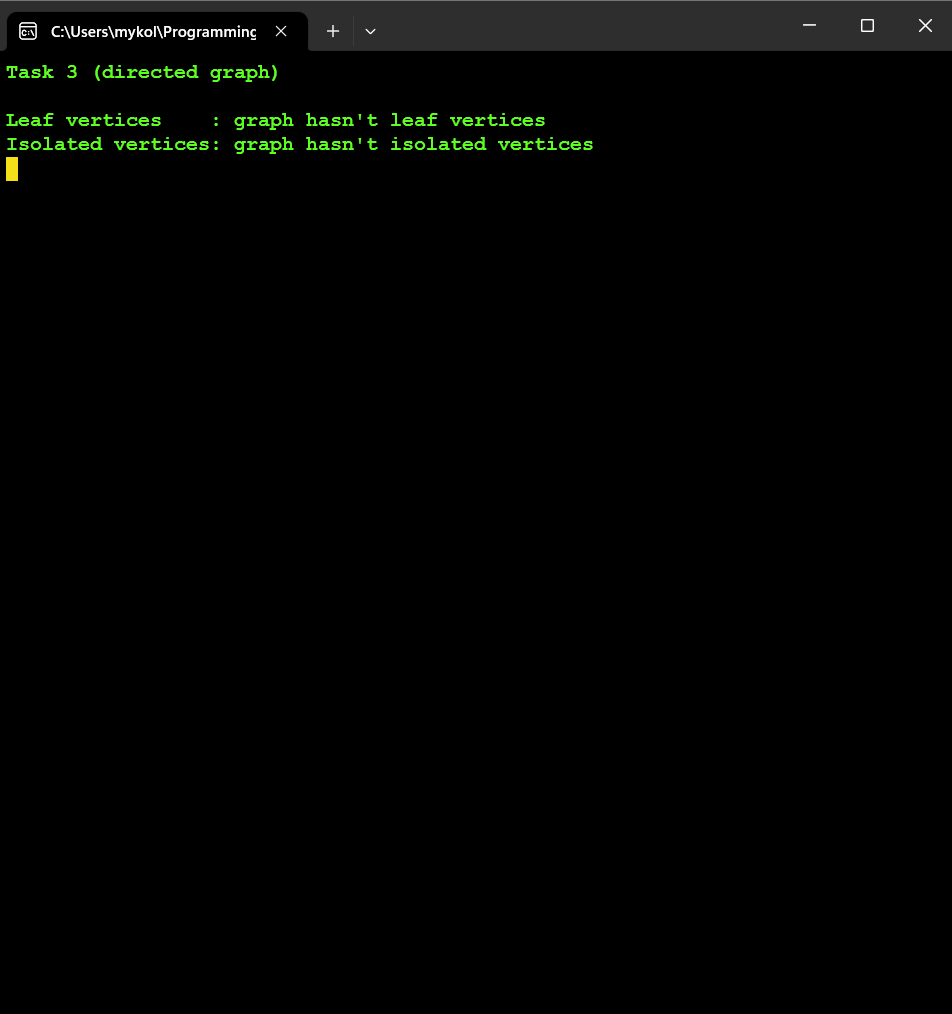
#ifndef **UNICODE**#define **UNICODE**#endif  
  
#include <windows.h>  
#include "GraphPainter.h"  
  
#define **TASK\_1\_DIRECTED\_GRAPH** 1  
#define **TASK\_1\_UNDIRECTED\_GRAPH** 2  
#define **TASK\_2\_DIRECTED\_GRAPH** 3  
#define **TASK\_2\_UNDIRECTED\_GRAPH** 4  
#define **TASK\_3\_DIRECTED\_GRAPH** 5  
#define **TASK\_3\_UNDIRECTED\_GRAPH** 6  
#define **TASK\_4\_1** 7  
#define **TASK\_4\_2** 8  
#define **TASK\_4\_3\_1** 9  
#define **TASK\_4\_3\_2** 10  
#define **TASK\_4\_4** 11  
#define **TASK\_4\_5** 12  
#define **TASK\_4\_6** 13  
#define **TASK\_4\_7** 14  
  
int current\_task = 0;  
int \*\*A;  
int \*\*uA;  
int \*\*mA;  
int \*\*current\_matrix;  
int \*\*vertices\_coords;  
int \*\*components\_coords;  
int \*\*current\_coords;  
int current\_n;  
  
LRESULT **CALLBACK** WndProc(HWND, UINT, WPARAM, LPARAM);  
  
void AddMenu(HWND hwnd);  
  
int **WINAPI** WinMain(HINSTANCE hInstance,  
 HINSTANCE hPrevInstance,  
 LPSTR lpCmdLine,  
 int nCmdShow)  
{  
 WNDCLASS wndClass;  
 wndClass.lpszClassName = L"Лабораторна робота 2.4";  
 wndClass.hInstance = hInstance;  
 wndClass.lpfnWndProc = WndProc;  
 wndClass.hCursor = **LoadCursor**(**NULL**, **IDC\_ARROW**);  
 wndClass.hIcon = 0;  
 wndClass.lpszMenuName = 0;  
 wndClass.hbrBackground = (HBRUSH)GetStockObject(**WHITE\_BRUSH**);  
 wndClass.style = **CS\_HREDRAW** | **CS\_VREDRAW**;  
 wndClass.cbClsExtra = 0;  
 wndClass.cbWndExtra = 0;  
 if (!**RegisterClass**(&wndClass)) return 0;  
 HWND hWnd;  
 MSG lpMsg;  
 hWnd = **CreateWindow**(L"Лабораторна робота 2.4",  
 L"Лабораторна робота 2.4, виконав М.М.Кушнір",  
 WS\_OVERLAPPEDWINDOW,  
 0, 0,  
 (WINDOW\_BORDER\_OFFSET \* 2 + GRAPH\_WIDTH + 40),  
 (WINDOW\_BORDER\_OFFSET \* 2 + GRAPH\_HEIGHT + 60),  
 (HWND)NULL,  
 (HMENU)NULL,  
 (HINSTANCE)hInstance,  
 (HINSTANCE)NULL);  
 ShowWindow(hWnd, nCmdShow);  
 UpdateWindow(hWnd);  
 int GetMessage\_res;  
 while ((GetMessage\_res = **GetMessage**(&lpMsg, hWnd, 0, 0)) != 0)  
 {  
 if (GetMessage\_res == -1)  
 return lpMsg.wParam;  
 else  
 {  
 TranslateMessage(&lpMsg);  
 **DispatchMessage**(&lpMsg);  
 }  
 }  
}  
  
LRESULT **CALLBACK** WndProc(HWND hWnd,  
 UINT message,  
 WPARAM wParam,  
 LPARAM lParam)  
{  
 HDC hdc;  
 PAINTSTRUCT ps;  
 switch (message)  
 {  
 case **WM\_COMMAND**:  
 {  
 system("cls");  
 RedrawWindow(hWnd, **NULL**, **NULL**,**RDW\_ERASE** | **RDW\_INVALIDATE**);  
 switch (wParam)  
 {  
 case **TASK\_1\_DIRECTED\_GRAPH**:  
 {  
 current\_task = **TASK\_1\_DIRECTED\_GRAPH**;  
 current\_matrix = A;  
 current\_coords = vertices\_coords;  
 current\_n = **N**;  
  
 printf("Task 1 (directed graph)\n\n");  
 printf("Adjacency matrix of directed graph (matrix 'A'):\n\n");  
 PrintBooleanMatrix(**N**, A);  
 break;  
 }  
 case **TASK\_1\_UNDIRECTED\_GRAPH**:  
 {  
 current\_task = **TASK\_1\_UNDIRECTED\_GRAPH**;  
 current\_matrix = uA;  
 current\_coords = vertices\_coords;  
 current\_n = **N**;  
  
 printf("Task 1 (undirected graph)\n\n");  
 printf("Adjacency matrix of undirected graph (matrix 'uA'):\n\n");  
 PrintBooleanMatrix(**N**, uA);  
 break;  
 }  
 case **TASK\_2\_DIRECTED\_GRAPH**:  
 {  
 current\_matrix = A;  
 current\_coords = vertices\_coords;  
 current\_task = **TASK\_2\_DIRECTED\_GRAPH**;  
 current\_n = **N**;  
  
 printf("Task 2 (directed graph)\n\n");  
 int \*\*vertices\_degs = GetVerticesDegsOfDGraph(**N**, A);  
 int cols\_lengths[] = { 5,12, 11 };  
 char \*headings[] = { "Outdegrees", "Indegrees" };  
 PrintInt2dArr(**N**, 2,  
 cols\_lengths, headings,  
 vertices\_degs);  
 printf("\n");  
 PrintDGraphHomogeneityDeg(**N**, vertices\_degs);  
 FreeInt2dArr(**N**, vertices\_degs);  
 break;  
 }  
 case **TASK\_2\_UNDIRECTED\_GRAPH**:  
 {  
 current\_task = **TASK\_2\_UNDIRECTED\_GRAPH**;  
 current\_matrix = uA;  
 current\_coords = vertices\_coords;  
 current\_n = **N**;  
  
 printf("Task 2 (undirected graph)\n\n");  
 int \*vertices\_degs = GetVerticesDegsOfUGraph(**N**, uA);  
 int cols\_lengths[] = { 5,9 };  
 PrintIntArr(**N**, cols\_lengths, "Degrees", vertices\_degs);  
 printf("\n");  
 PrintUGraphHomogeneityDeg(**N**, vertices\_degs);  
 free(vertices\_degs);  
 break;  
 }  
 case **TASK\_3\_DIRECTED\_GRAPH**:  
 {  
 current\_task = **TASK\_3\_DIRECTED\_GRAPH**;  
 current\_matrix = A;  
 current\_coords = vertices\_coords;  
 current\_n = **N**;  
  
 printf("Task 3 (directed graph)\n\n");  
 int \*\*vertices\_degs = GetVerticesDegsOfDGraph(**N**, A);  
 PrintLIVerticesOfDGraph(**N**, vertices\_degs);  
 FreeInt2dArr(**N**, vertices\_degs);  
 break;  
 }  
 case **TASK\_3\_UNDIRECTED\_GRAPH**:  
 {  
 current\_task = **TASK\_3\_UNDIRECTED\_GRAPH**;  
 current\_matrix = uA;  
 current\_coords = vertices\_coords;  
 current\_n = **N**;  
  
 printf("Task 3 (undirected graph)\n\n");  
 int \*vertices\_degs = GetVerticesDegsOfUGraph(**N**, uA);  
 PrintLIVerticesOfUGraph(**N**, vertices\_degs);  
 free(vertices\_degs);  
 break;  
 }  
 case **TASK\_4\_1**:  
 {  
 current\_task = **TASK\_4\_1**;  
 current\_matrix = mA;  
 current\_coords = vertices\_coords;  
 current\_n = **N**;  
  
 printf("Task 4.1\n\n");  
 printf("Modified adjacency matrix of graph (matrix 'mA'):\n\n");  
 PrintBooleanMatrix(**N**, mA);  
 break;  
 }  
 case **TASK\_4\_2**:  
 {  
 current\_task = **TASK\_4\_2**;  
 current\_matrix = mA;  
 current\_coords = vertices\_coords;  
 current\_n = **N**;  
  
 printf("Task 4.2\n\n");  
 int \*\*vertices\_degs = GetVerticesDegsOfDGraph(**N**, mA);  
 int cols\_lengths[] = { 5,12, 11 };  
 */\* Перший стовпчик буде використаний для нумерації рядків \*/* char \*headings[] = { "Outdegrees", "Indegrees" };  
 PrintInt2dArr(**N**, 2,  
 cols\_lengths, headings,  
 vertices\_degs);  
 FreeInt2dArr(**N**, vertices\_degs);  
 break;  
 }  
 case **TASK\_4\_3\_1**:  
 {  
 current\_task = **TASK\_4\_3\_1**;  
 current\_matrix = mA;  
 current\_coords = vertices\_coords;  
 current\_n = **N**;  
  
 printf("Task 4.3.1\n\n");  
 printf("Paths of length 2:\n");  
 int \*\*mA2 =MultSquareMatrices(**N**, mA, mA);  
 PrintPathsWithLen2(**N**, mA, mA2);  
 FreeInt2dArr(**N**, mA2);  
 break;  
 }  
 case **TASK\_4\_3\_2**:  
 {  
 current\_task = **TASK\_4\_3\_2**;  
 current\_matrix = mA;  
 current\_coords = vertices\_coords;  
 current\_n = **N**;  
  
 printf("Task 4.3.2\n\n");  
 printf("Paths of length 3:\n");  
 int \*\*mA2 = MultSquareMatrices(**N**, mA, mA);  
 int \*\*mA3 = MultSquareMatrices(**N**,mA2,mA);  
 PrintPathsWithLen3(**N**, mA, mA3);  
 FreeInt2dArr(**N**, mA2);  
 FreeInt2dArr(**N**, mA3);  
 break;  
 }  
 case **TASK\_4\_4**:  
 {  
 current\_task = **TASK\_4\_4**;  
 current\_matrix = mA;  
 current\_coords = vertices\_coords;  
 current\_n = **N**;  
  
 printf("Task 4.4\n\n");  
 printf("Reachability matrix:\n\n");  
 int \*\*R = GetReachabilityMatrix(**N**, mA);  
 PrintBooleanMatrix(**N**, R);  
 FreeInt2dArr(**N**, R);  
 break;  
 }  
 case **TASK\_4\_5**:  
 {  
 current\_task = **TASK\_4\_5**;  
 current\_matrix = mA;  
 current\_coords = vertices\_coords;  
 current\_n = **N**;  
  
 printf("Task 4.5\n\n");  
 printf("Components of strong connectivity:\n");  
 int \*\*R = GetReachabilityMatrix(**N**, mA);  
 int \*\*Rt = TransposeSquareMatrix(**N**, R);  
 int \*\*S = MultSquareMatricesElemByElem(**N**, R, Rt);  
 int \*\*components = GetConnectComponents(**N**, S);  
 PrintConnectComponents(**N**, components);  
 FreeInt2dArr(**N**, R);  
 FreeInt2dArr(**N**, Rt);  
 FreeInt2dArr(**N**, S);  
 FreeInt2dArr(**N**, components);  
 break;  
 }  
 case **TASK\_4\_6**:  
 {  
 current\_task = **TASK\_4\_6**;  
 current\_matrix = mA;  
 current\_coords = vertices\_coords;  
 current\_n = **N**;  
  
 printf("Task 4.6\n\n");  
 printf("Connectivity matrix:\n\n");  
 int \*\*R = GetReachabilityMatrix(**N**, mA);  
 int \*\*Rt = TransposeSquareMatrix(**N**, R);  
 int \*\*S = MultSquareMatricesElemByElem(**N**, R, Rt);  
 PrintBooleanMatrix(**N**, S);  
 FreeInt2dArr(**N**, R);  
 FreeInt2dArr(**N**, Rt);  
 FreeInt2dArr(**N**, S);  
 break;  
 }  
 case **TASK\_4\_7**:  
 {  
 current\_task = **TASK\_4\_7**;  
 printf("Task 4.7\n\n");  
 int \*\*R = GetReachabilityMatrix(**N**, mA);  
 int \*\*Rt = TransposeSquareMatrix(**N**, R);  
 int \*\*S = MultSquareMatricesElemByElem(**N**, R, Rt);  
 int \*\*components = GetConnectComponents(**N**, S);  
 int \*\*graph\_matrix = GetCondensationGraphMatrix(**N**, components, mA);  
 int components\_ct = 0;  
 while (components\_ct < **N** && components[components\_ct][**N**] != 0)  
 components\_ct++;  
 current\_n = components\_ct;  
 printf("Adjacency matrix of the condensation graph:\n\n");  
 PrintBooleanMatrix(current\_n, graph\_matrix);  
 components\_coords = SetVerticesCoords(current\_n);  
 current\_matrix = graph\_matrix;  
 current\_coords = components\_coords;  
 FreeInt2dArr(**N**, R);  
 FreeInt2dArr(**N**, Rt);  
 FreeInt2dArr(**N**, S);  
 FreeInt2dArr(**N**, components);  
 break;  
 }  
 }  
 }  
 case **WM\_PAINT**:  
 {  
 hdc = BeginPaint(hWnd, &ps);  
 SetBkMode(hdc, **TRANSPARENT**);  
 HPEN ePen = CreatePen(**PS\_SOLID**, 1, **RGB**(0, 38, 0));  
 HPEN vOutlinePen = CreatePen(**PS\_SOLID**, 3, **RGB**(3, 104, 65));  
 HBRUSH vFillBrush = CreateSolidBrush(**RGB**(37, 255, 127));  
 if (current\_task)  
 DrawGraph(current\_n,  
 current\_matrix,  
 current\_coords,  
 ePen,  
 vFillBrush,  
 vOutlinePen,  
 hdc);

if (current\_n != **N**)  
 FreeInt2dArr(current\_n, components\_coords);  
 DeleteObject(ePen);  
 DeleteObject(vOutlinePen);  
 DeleteObject(vFillBrush);  
 EndPaint(hWnd, &ps);  
 break;  
 }  
 case **WM\_CREATE**:  
 AddMenu(hWnd);  
 vertices\_coords = SetVerticesCoords(**N**);  
 srand(**N1** \* 1000 + **N2** \* 100 + **N3** \* 10 + **N4**);  
 double \*\*T = randm(**N**, **N**);  
 A = mulmr(**N**, **N**, T,  
 (1.0 - **N3** \* 0.01 - **N4** \* 0.01 - 0.3));  
 uA = SymmetrizeMatrix(**N**, A);  
 mA = mulmr(**N**, **N**, T,  
 (1.0 - **N3** \* 0.005 - **N4** \* 0.005 - 0.27));  
 printf("Matrix T: \n\n");  
 PrintDouble2dArr(**N**, **N**, T);  
 printf("\n\*You can select an option from the window pop-up menu\n");  
 FreeDouble2dArr(**N**, T);  
 break;  
 case **WM\_DESTROY**:  
 FreeInt2dArr(**N**, vertices\_coords);  
 FreeInt2dArr(**N**, A);  
 FreeInt2dArr(**N**, uA);  
 FreeInt2dArr(**N**, mA);  
 PostQuitMessage(0);  
 break;  
 default :  
 return **DefWindowProc**(hWnd, message, wParam, lParam);  
 }  
}  
  
void AddMenu(HWND hwnd)  
{  
 HMENU hMenu = CreateMenu();  
 */\* Підменю для завдання 1 \*/* HMENU hTask1 = CreateMenu();  
 **AppendMenu**(hTask1, **MF\_STRING**, **TASK\_1\_DIRECTED\_GRAPH**,  
 L"1) намалювати напрямлений граф");  
 **AppendMenu**(hTask1, **MF\_STRING**, **TASK\_1\_UNDIRECTED\_GRAPH**,  
 L"2) намалювати ненапрямлений граф");  
 **AppendMenu**(hMenu, **MF\_POPUP**, (UINT\_PTR)hTask1,  
 L"Завдання 1");  
 */\* Підменю для завдання 2 \*/* HMENU hTask2 = CreateMenu();  
 **AppendMenu**(hTask2, **MF\_STRING**, **TASK\_2\_DIRECTED\_GRAPH**,  
 L"1) визначити напівстепені вершин напрямленого графа");  
 **AppendMenu**(hTask2, **MF\_STRING**, **TASK\_2\_UNDIRECTED\_GRAPH**,  
 L"2) визначити степені вершин ненапрямленого графа");  
 **AppendMenu**(hMenu, **MF\_POPUP**, (UINT\_PTR)hTask2,  
 L"Завдання 2");  
 */\* Підменю для завдання 3 \*/* HMENU hTask3 = CreateMenu();  
 **AppendMenu**(hTask3, **MF\_STRING**, **TASK\_3\_DIRECTED\_GRAPH**,  
 L"1) визначити висячі та ізольовані вершини напрямленого графа");  
 **AppendMenu**(hTask3, **MF\_STRING**, **TASK\_3\_UNDIRECTED\_GRAPH**,  
 L"2) визначити висячі та ізольовані вершини ненапрямленого графа");  
 **AppendMenu**(hMenu, **MF\_POPUP**, (UINT\_PTR)hTask3,  
 L"Завдання 3");  
 */\* Підменю для завдання 4 \*/* HMENU hTask4 = CreateMenu();  
 **AppendMenu**(hTask4, **MF\_STRING**, **TASK\_4\_1**,  
 L"1) обчислити змінену матрицю суміжності");  
 **AppendMenu**(hTask4, **MF\_STRING**, **TASK\_4\_2**,  
 L"2) визначити напівстепені вузлів");  
 */\* Підменю для завдання 4-3) \*/* HMENU hTask4\_3 = CreateMenu();  
 **AppendMenu**(hTask4\_3, **MF\_STRING**, **TASK\_4\_3\_1**,  
 L" 2 ");  
 **AppendMenu**(hTask4\_3, **MF\_STRING**, **TASK\_4\_3\_2**,  
 L" 3 ");  
 **AppendMenu**(hTask4, **MF\_POPUP**, (UINT\_PTR)hTask4\_3,  
 L"3) знайти всі шляхи довжиною ...");  
 */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/* **AppendMenu**(hTask4, **MF\_STRING**, **TASK\_4\_4**,  
 L"4) обчислити матрицю досяжності");  
 **AppendMenu**(hTask4, **MF\_STRING**, **TASK\_4\_5**,  
 L"5) знайти компоненти сильної зв'язності");  
 **AppendMenu**(hTask4, **MF\_STRING**, **TASK\_4\_6**,  
 L"6) обчислити матрицю зв'язності");  
 **AppendMenu**(hTask4, **MF\_STRING**, **TASK\_4\_7**,  
 L"7) намалювати граф конденсації");  
 **AppendMenu**(hMenu, **MF\_POPUP**, (UINT\_PTR)hTask4,  
 L"Завдання 4");  
 SetMenu(hwnd, hMenu);  
}

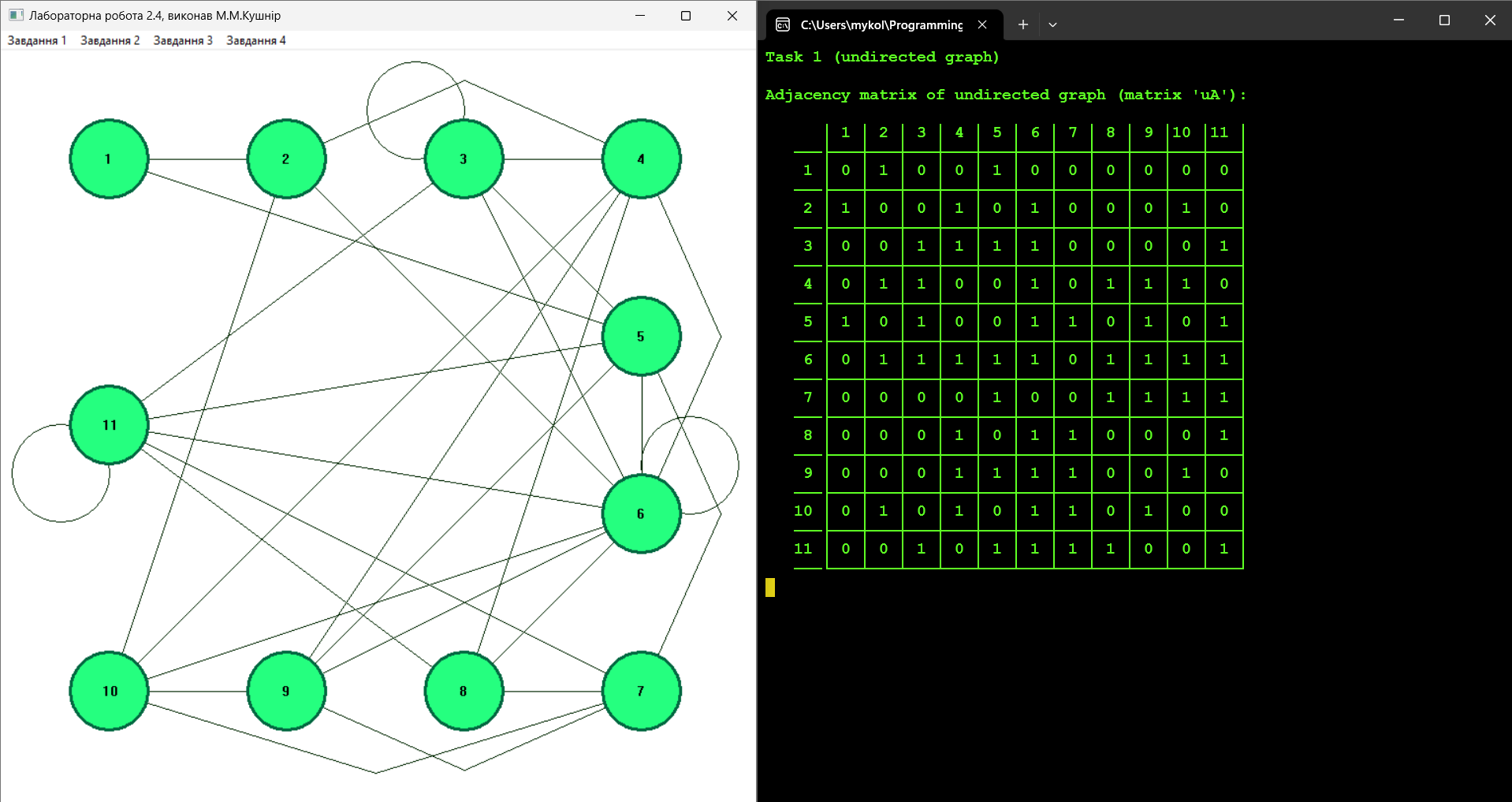
***Результати тестування програми***

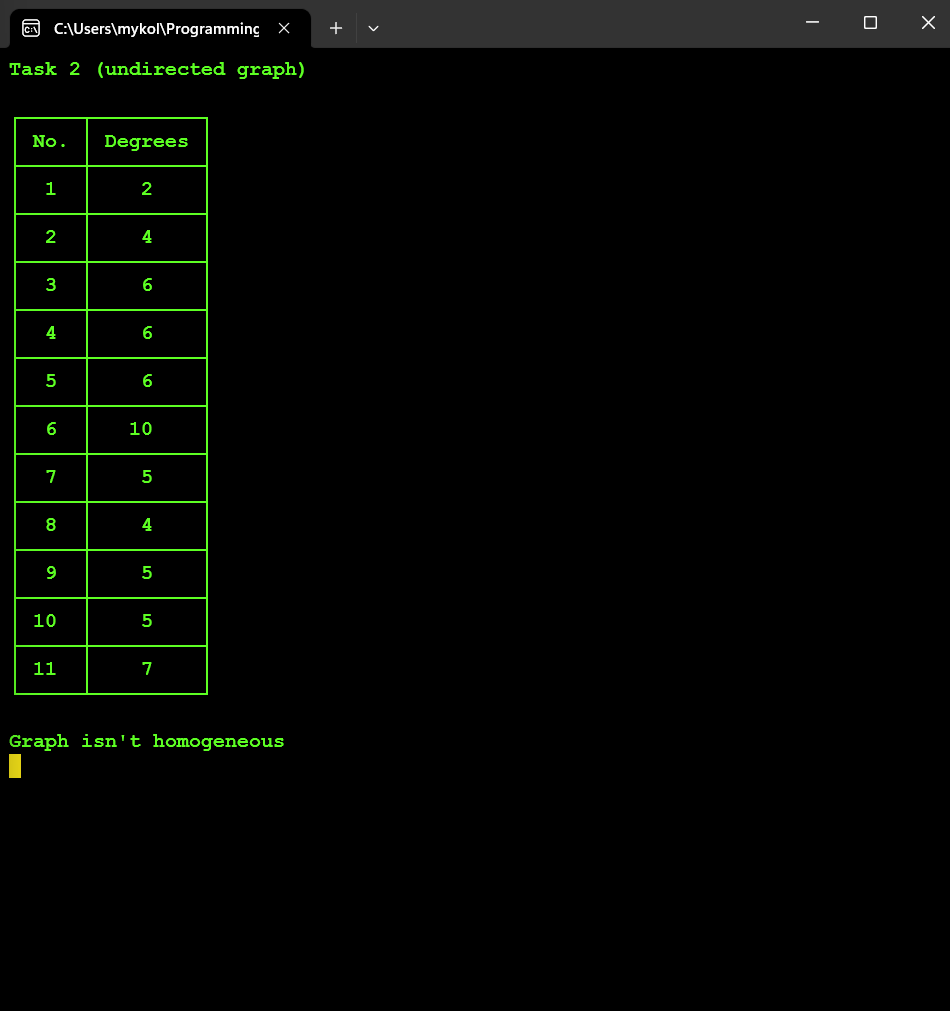
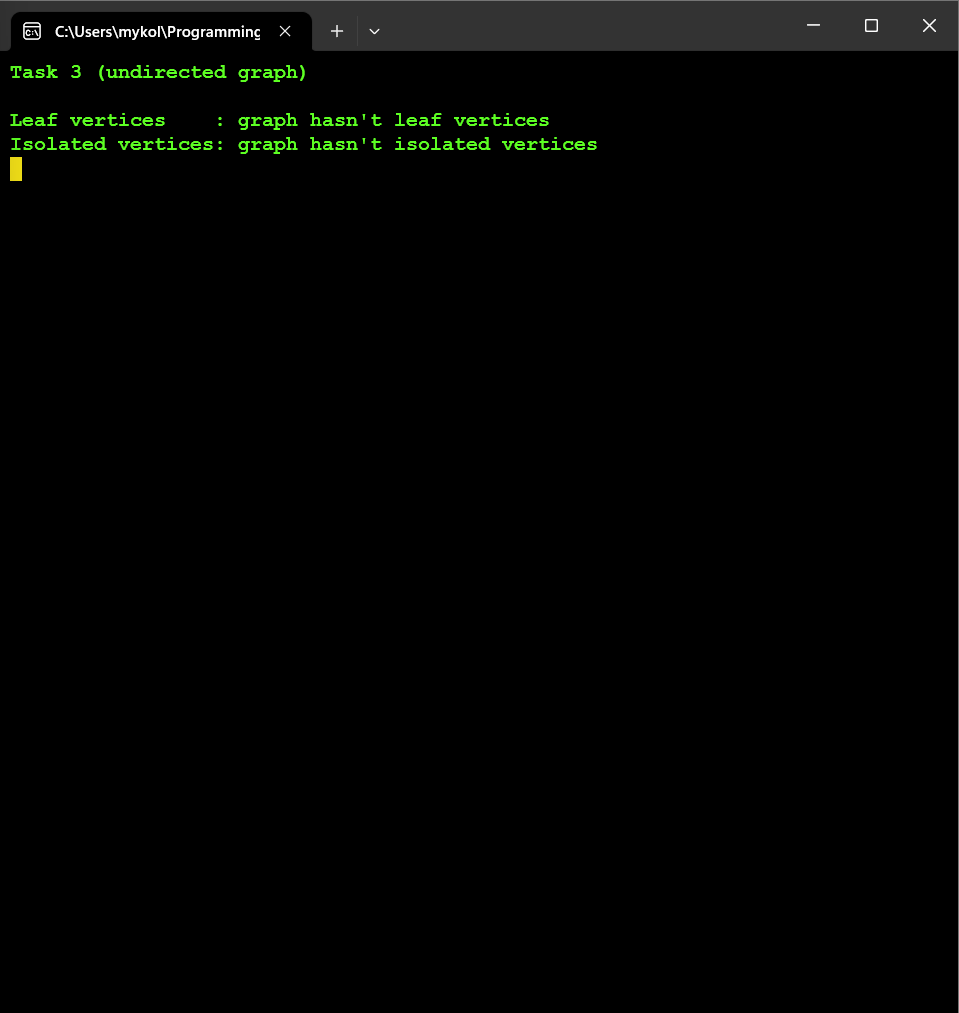
***Напрямлений (не модифікований) граф, його матриця суміжності, таблиця напівстепенів вузлів, перелік висячих та ізольованих вершин***

******

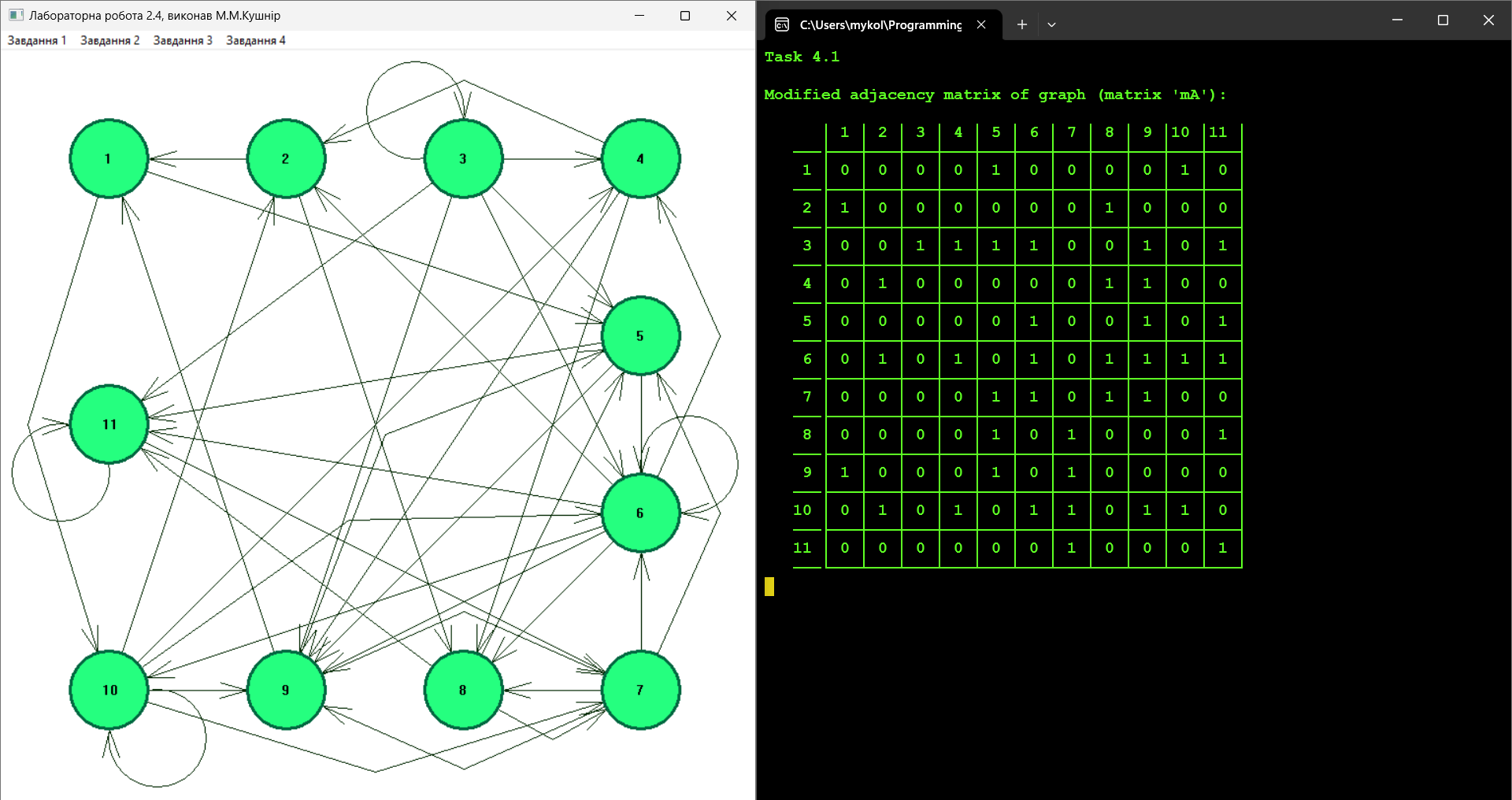
****** 

***Ненапрямлений граф, його матриця суміжності, таблиця степенів вузлів, перелік висячих та ізольованих вершин***

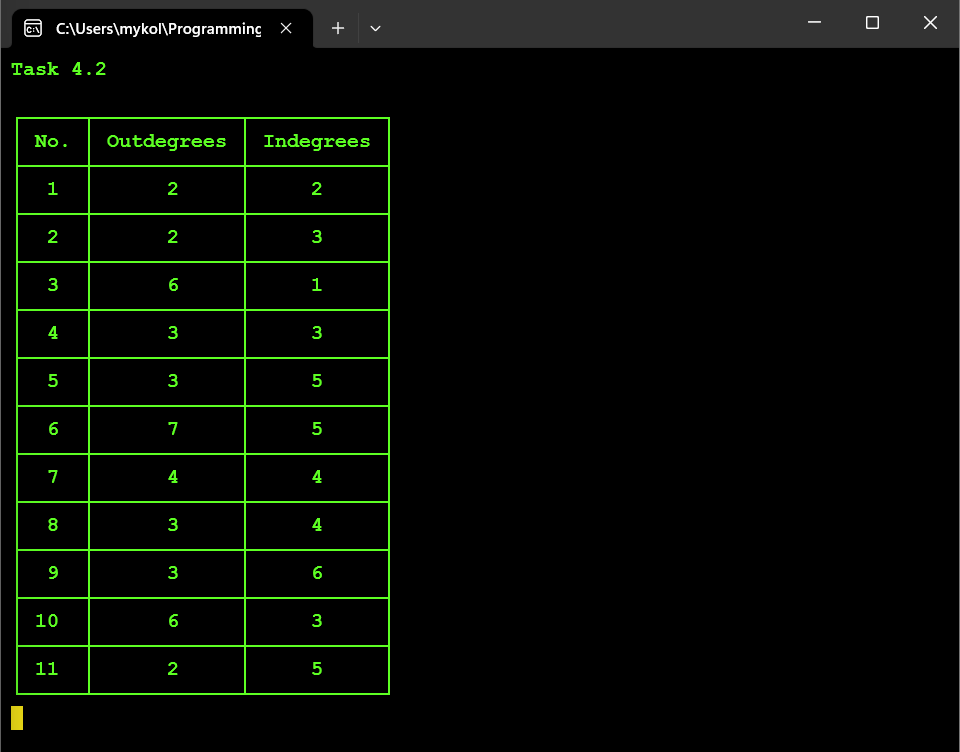
******

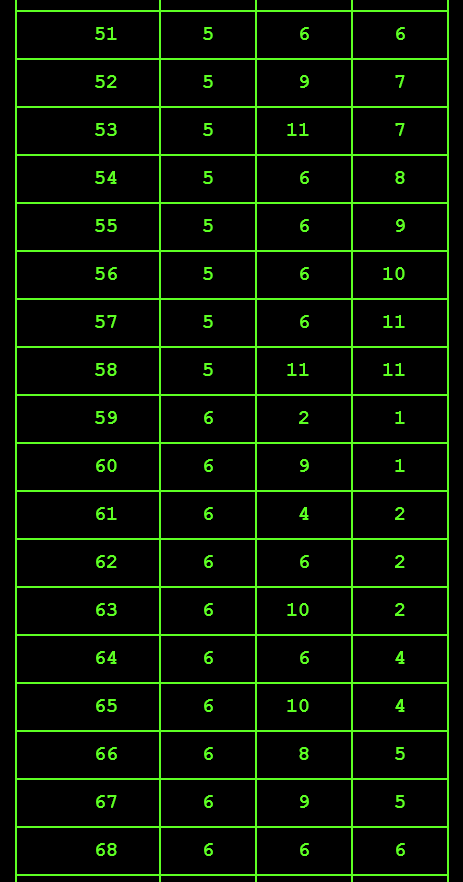
****** ******

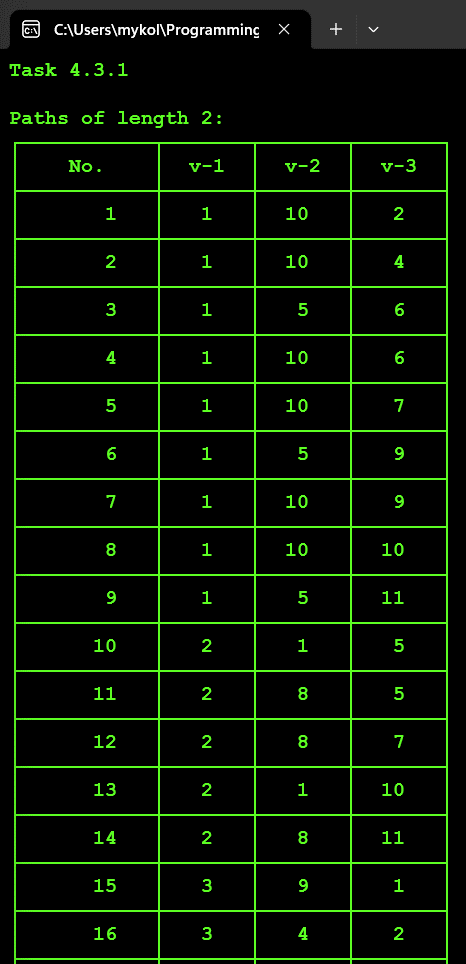
***Модифікований напрямлений граф та його матриця суміжності***

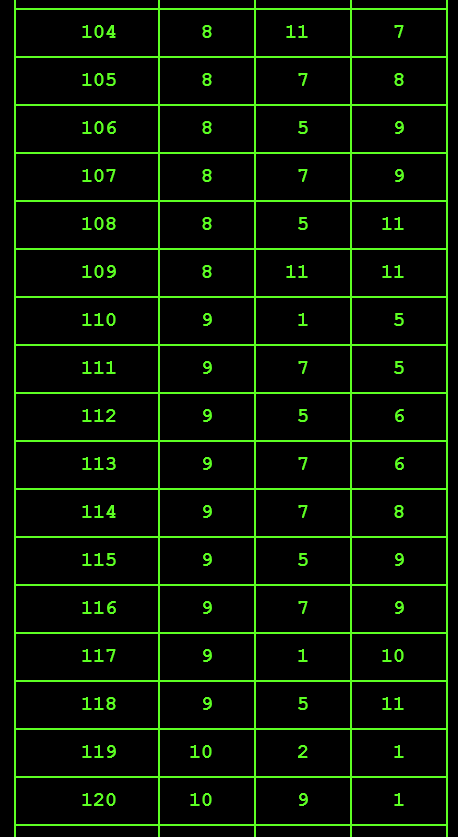


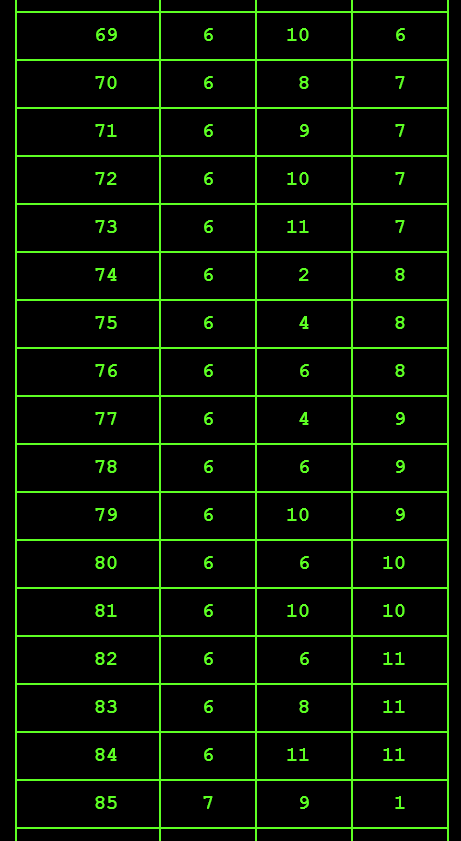
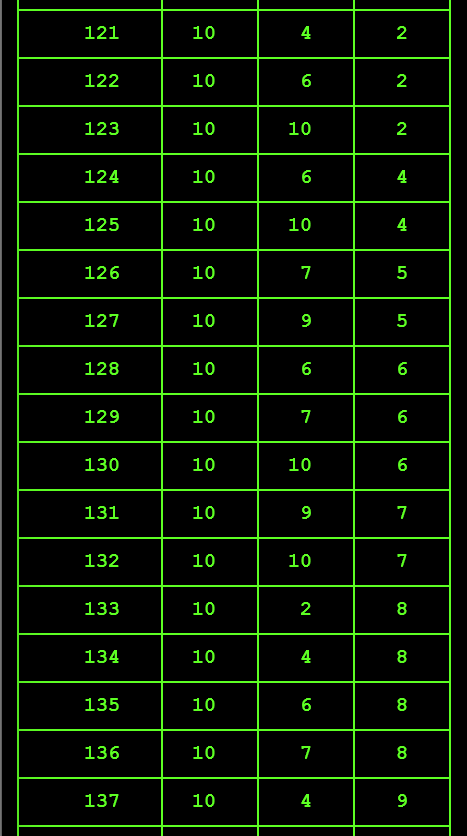
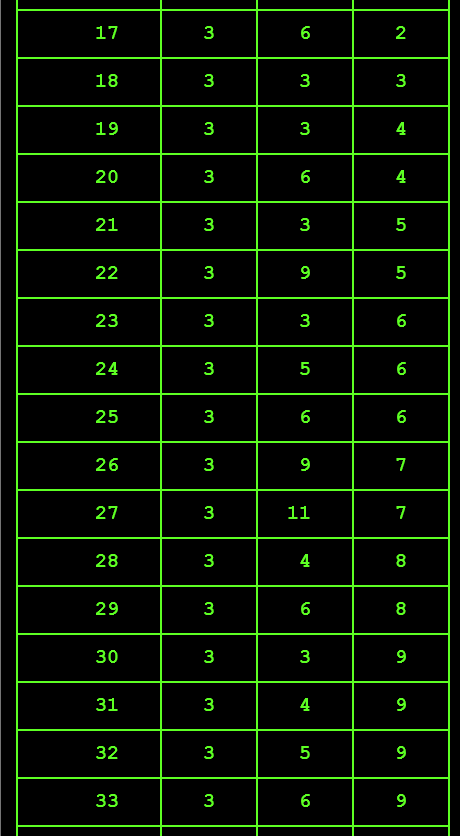
***Таблиця степенів вузлів модифікованого графа***

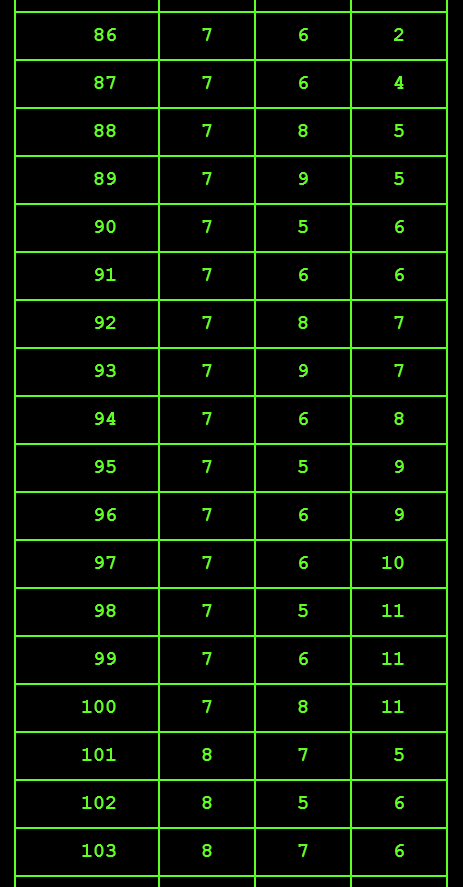
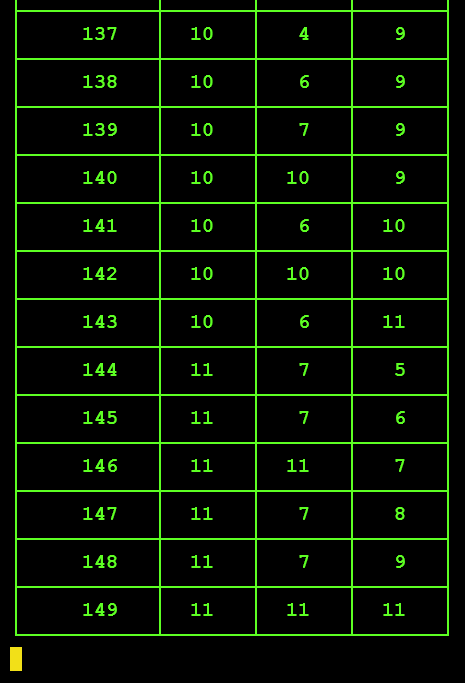


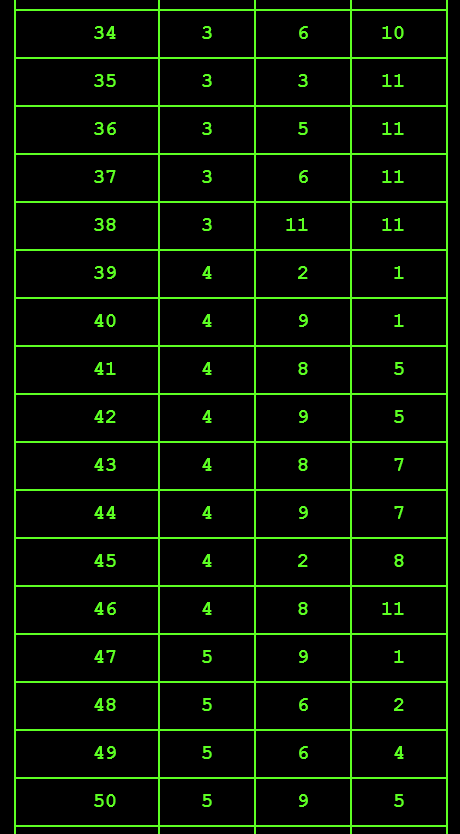
***Шляхи довжиною 2***

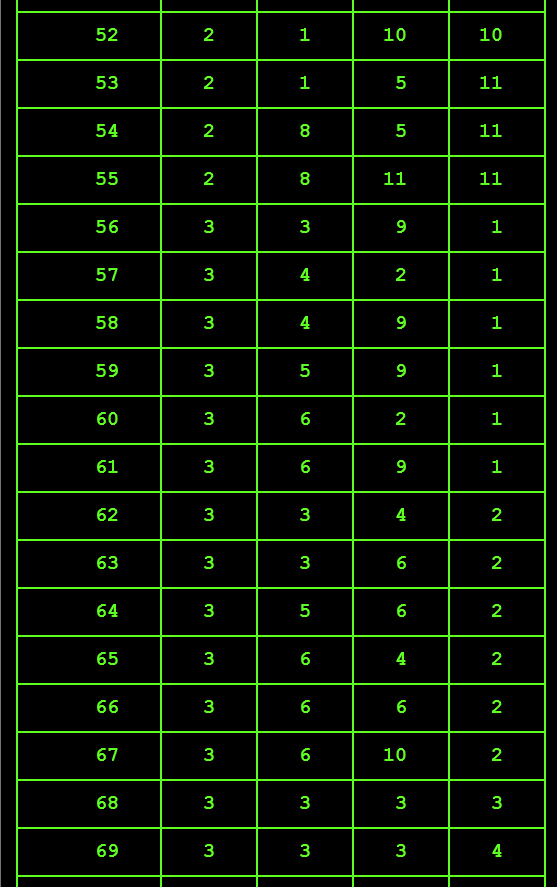
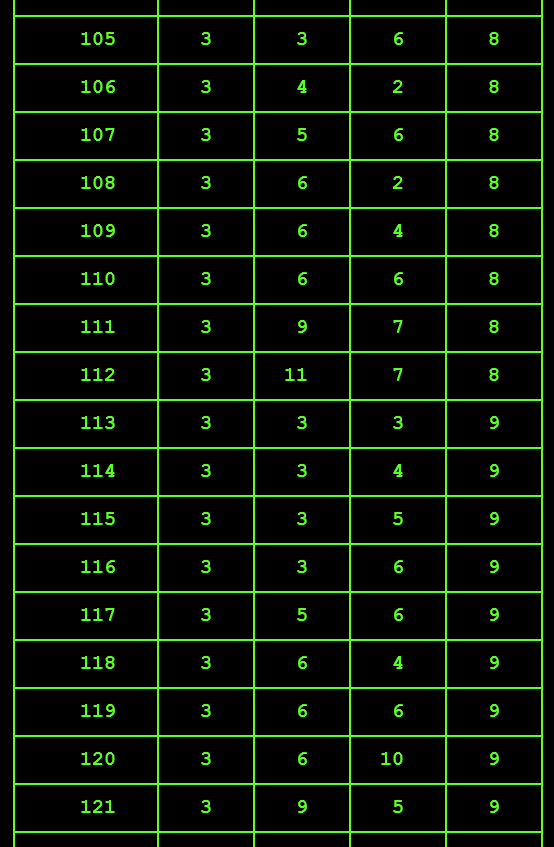
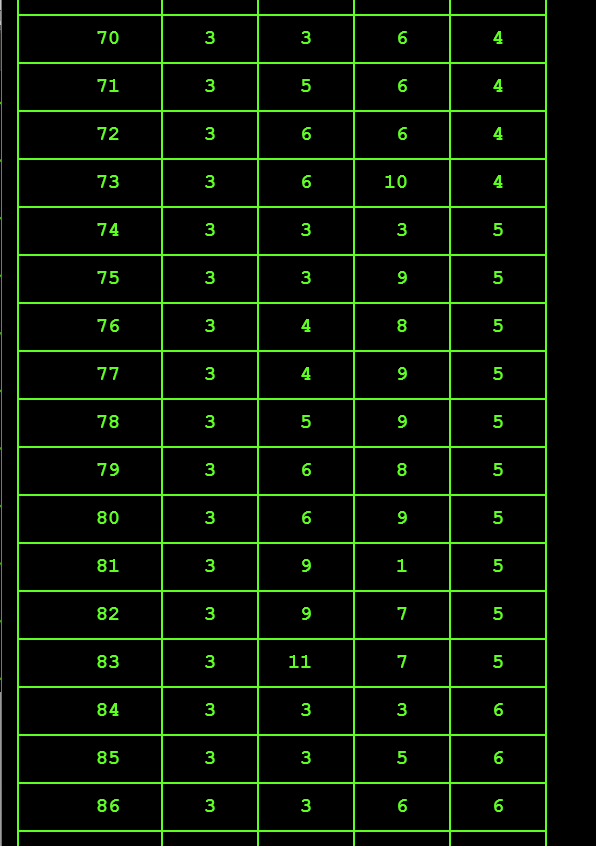
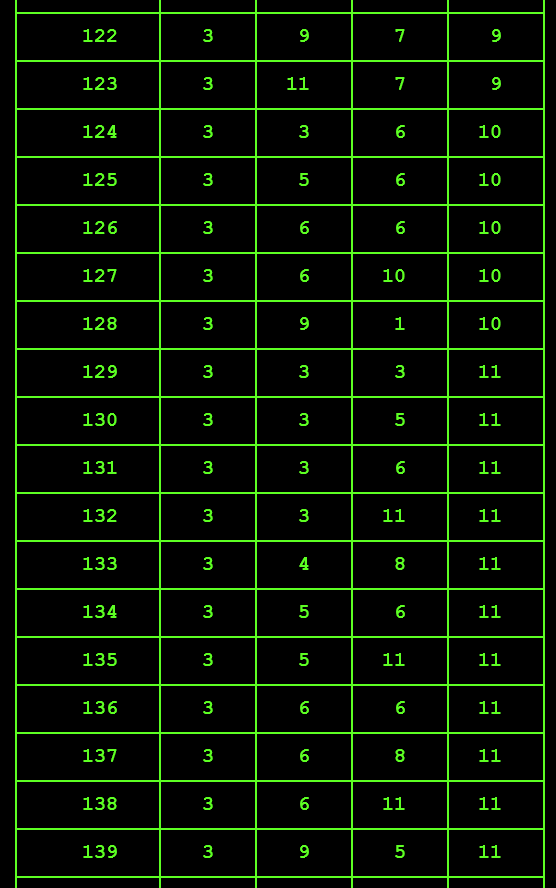
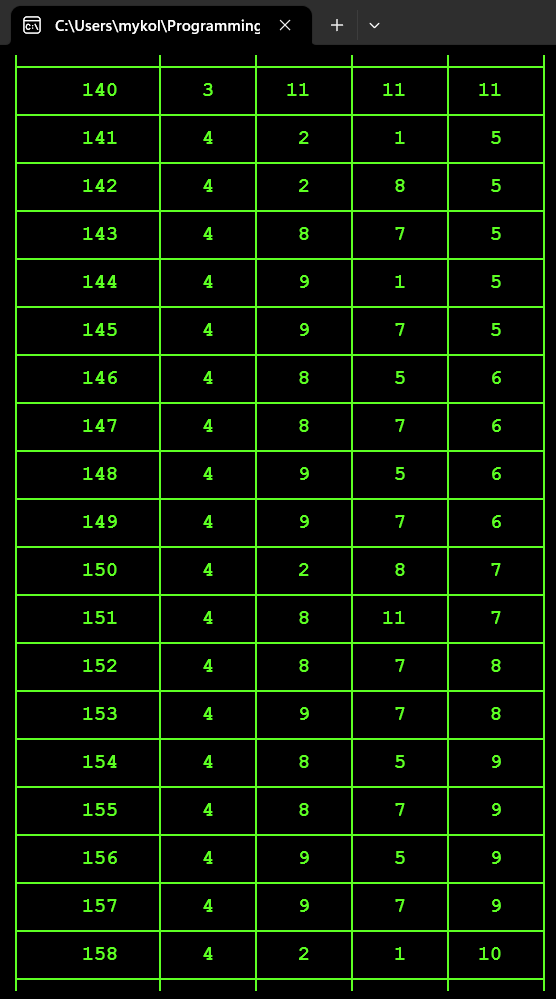
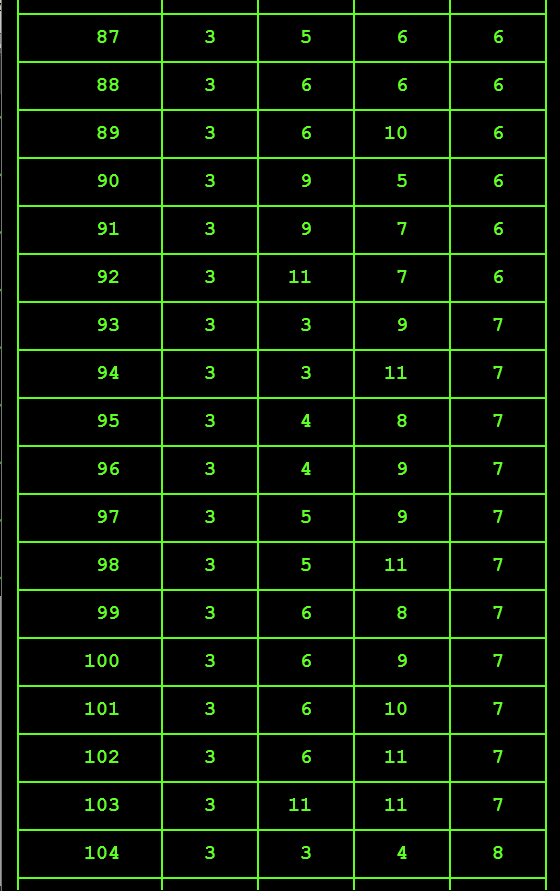
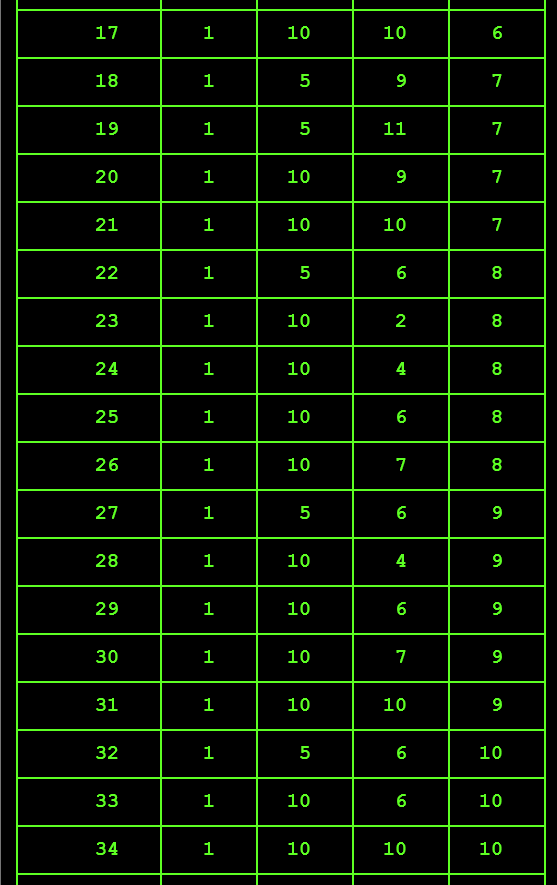
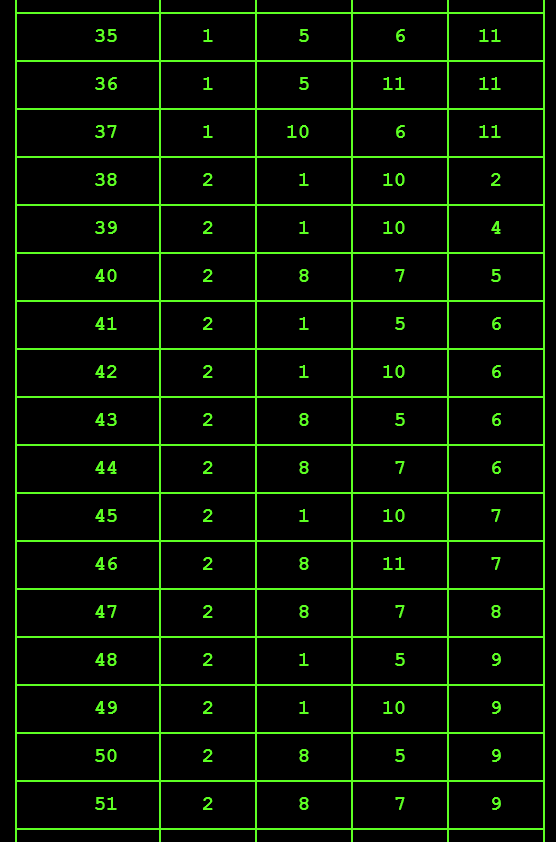
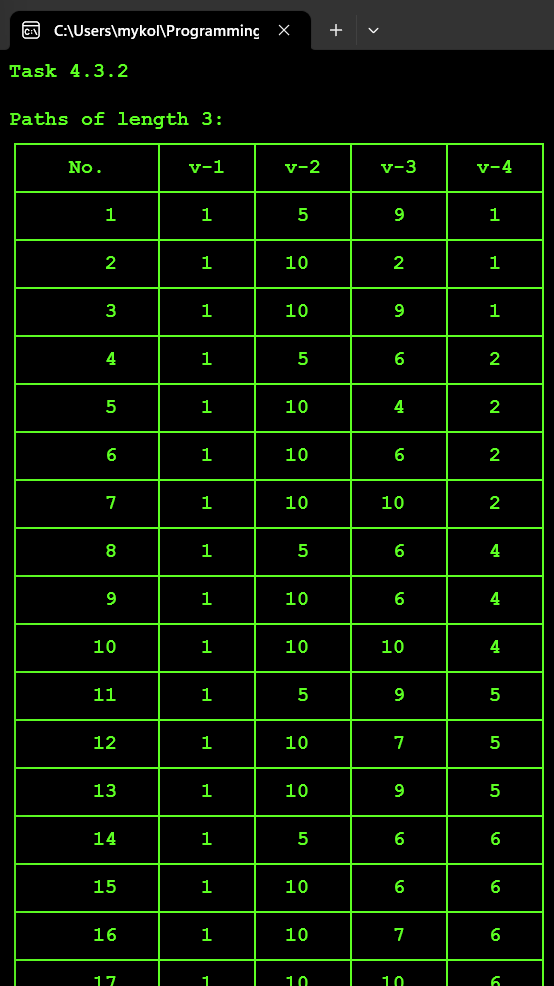


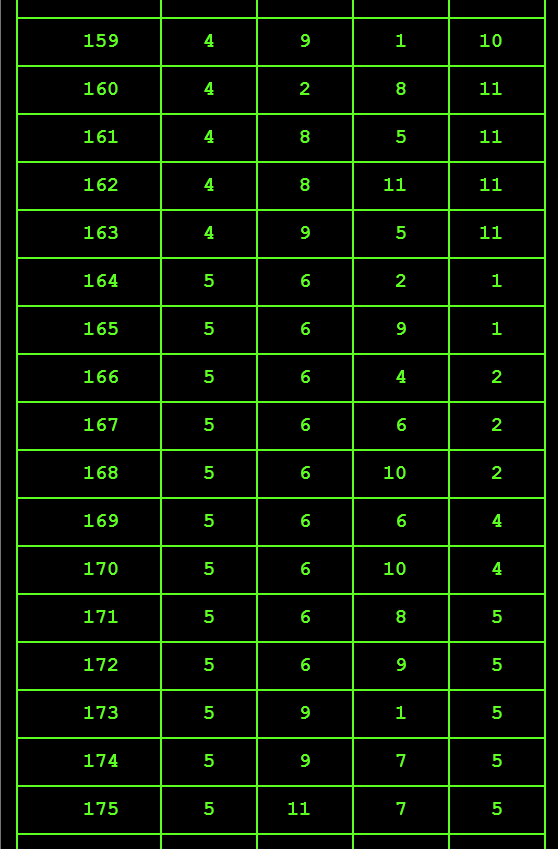
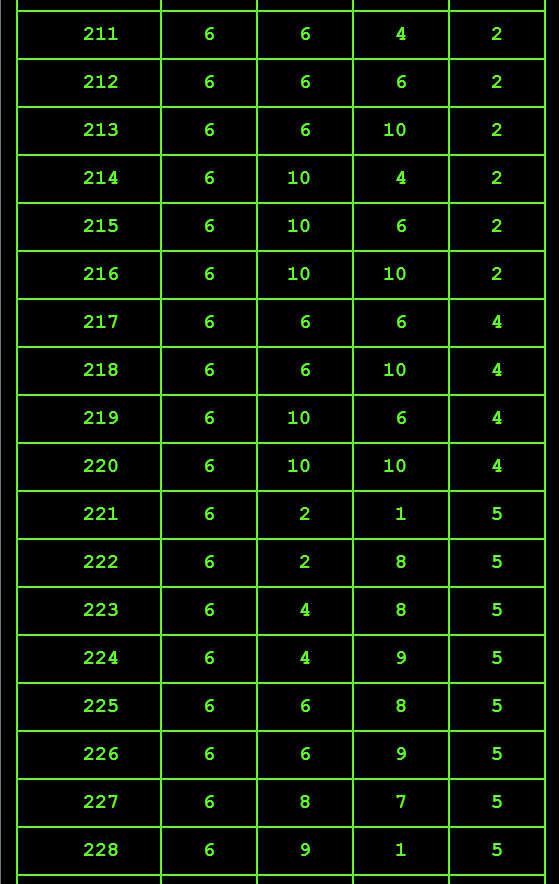
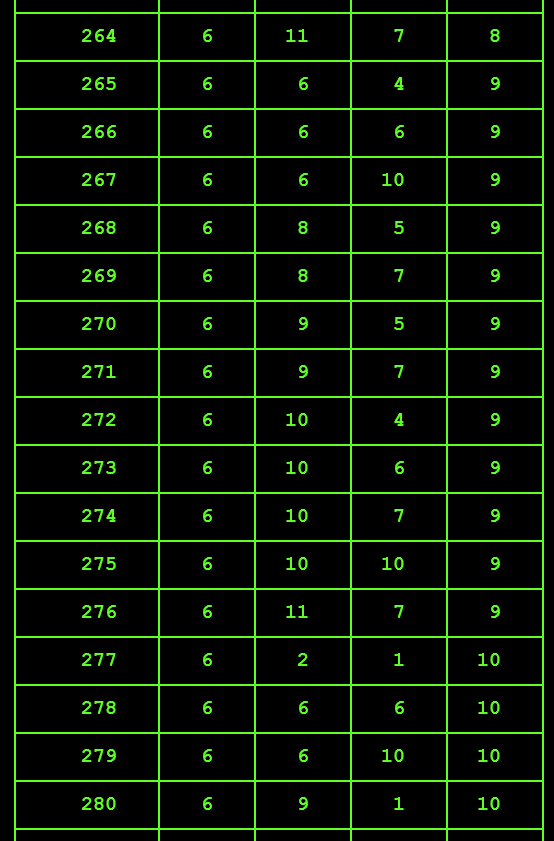
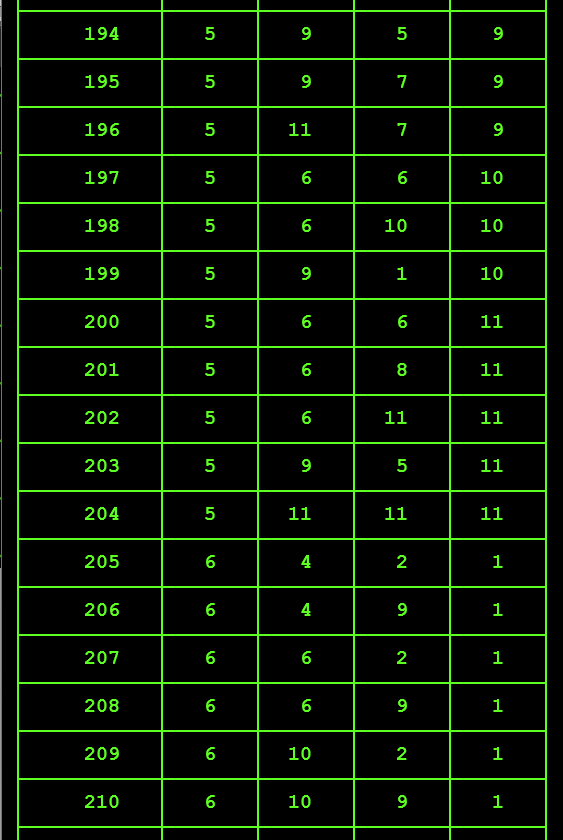
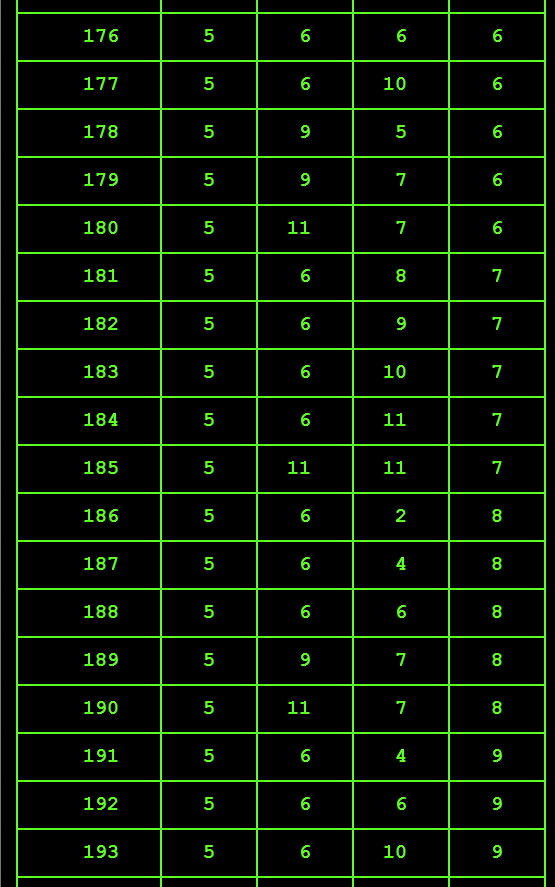
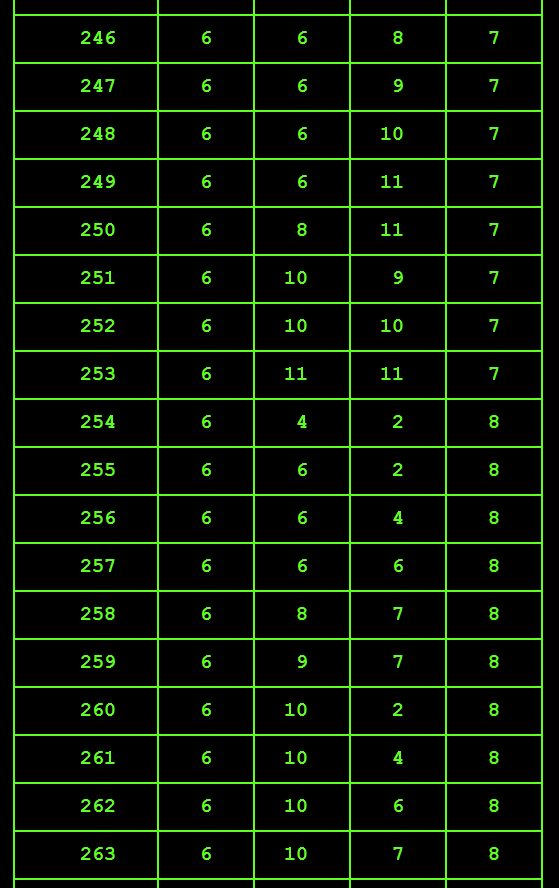
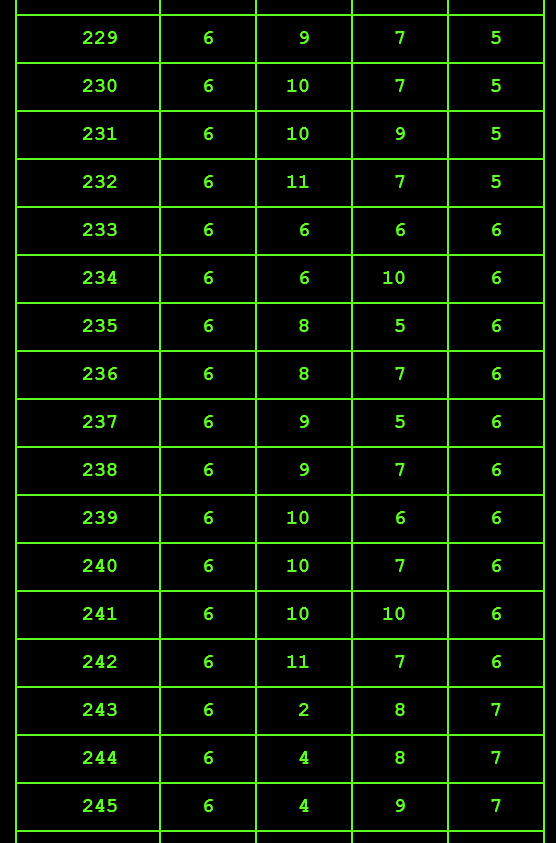
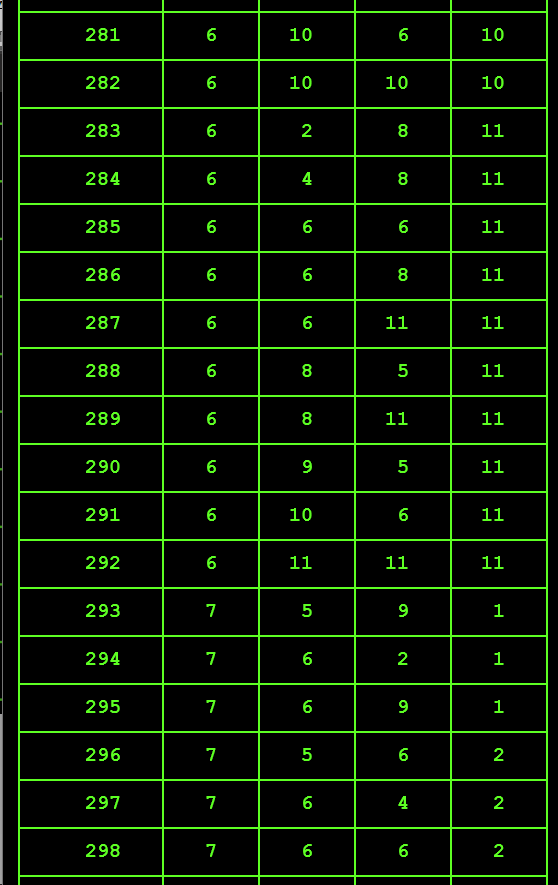
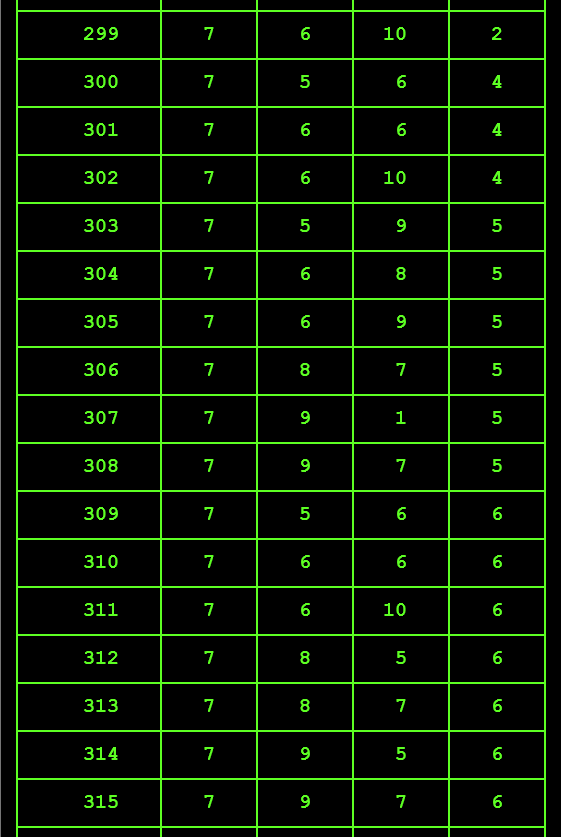


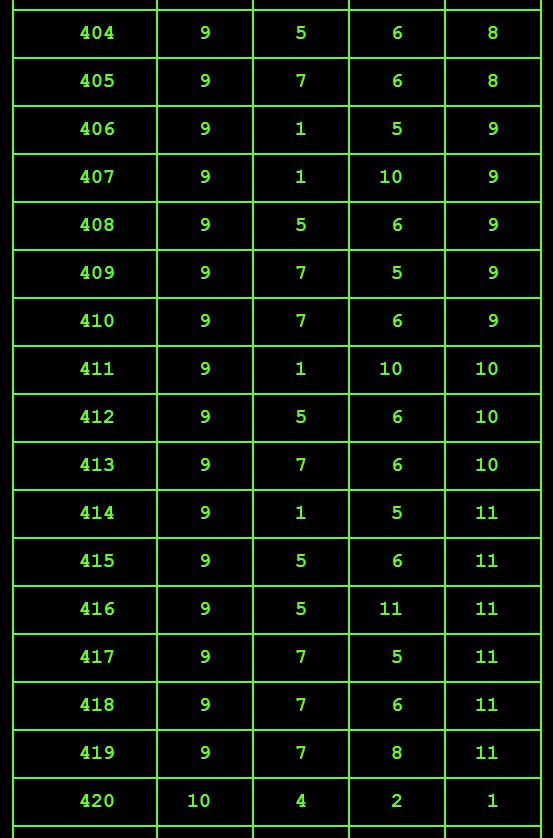
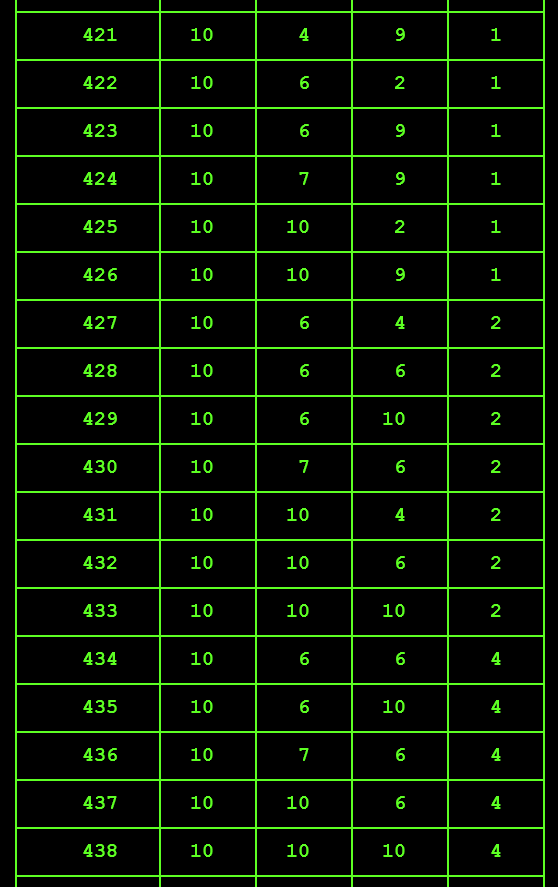
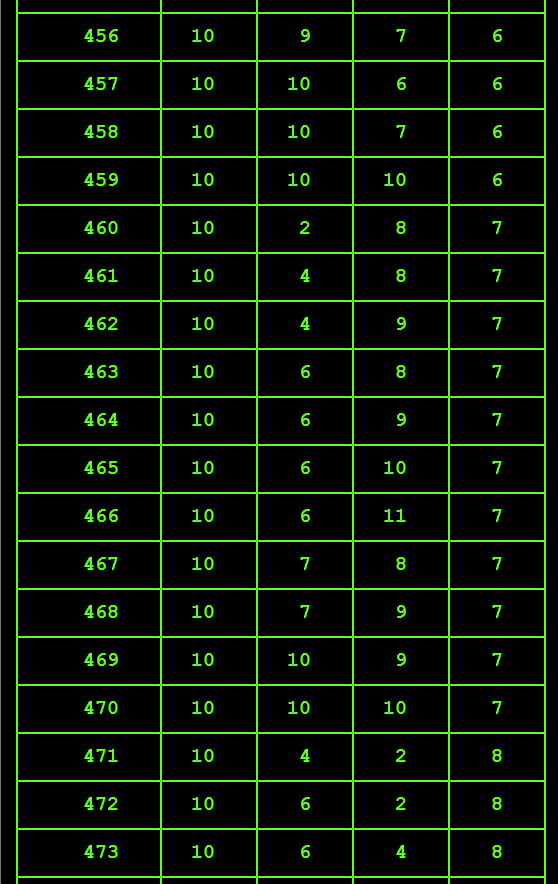
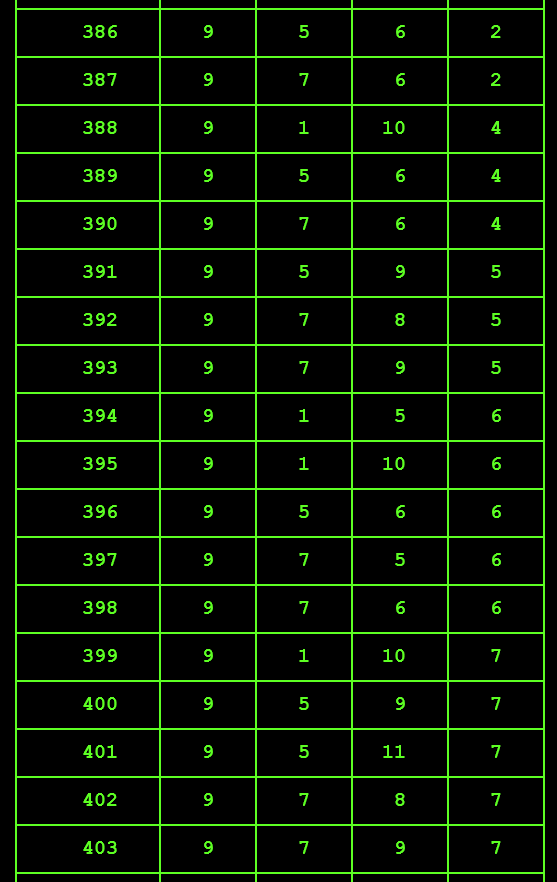
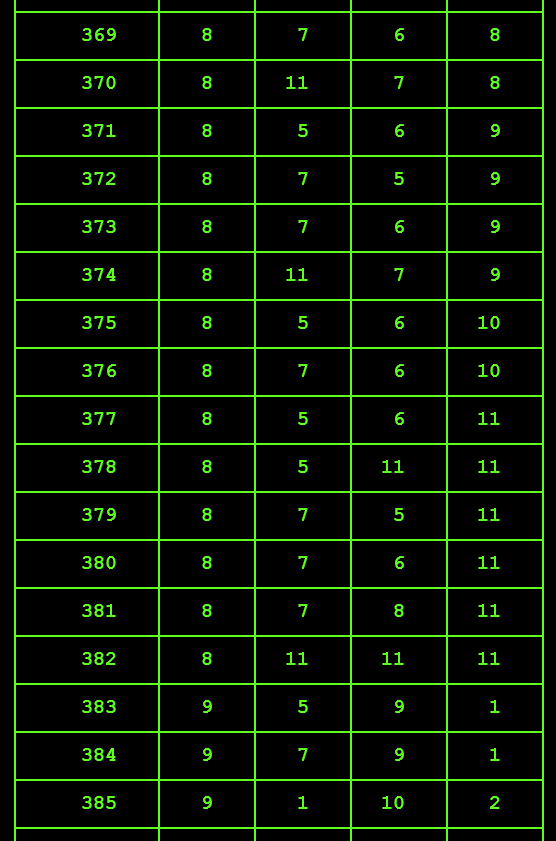
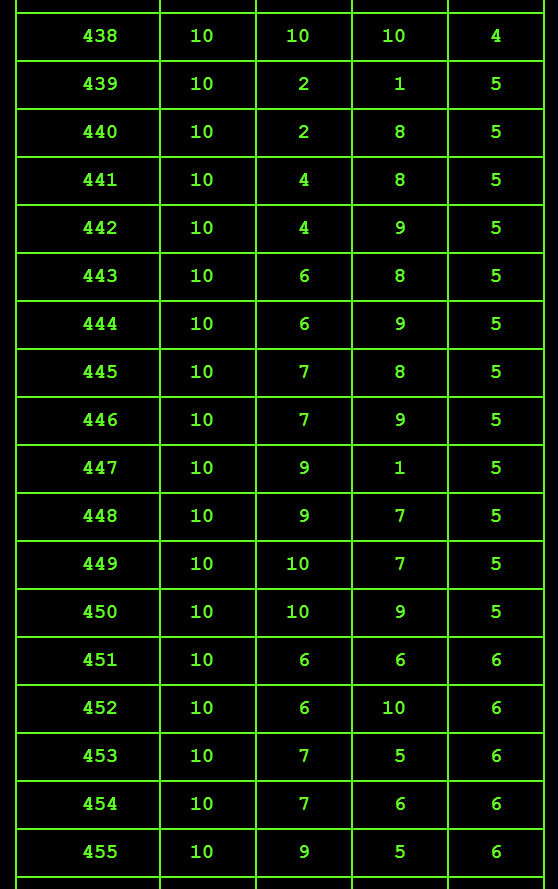
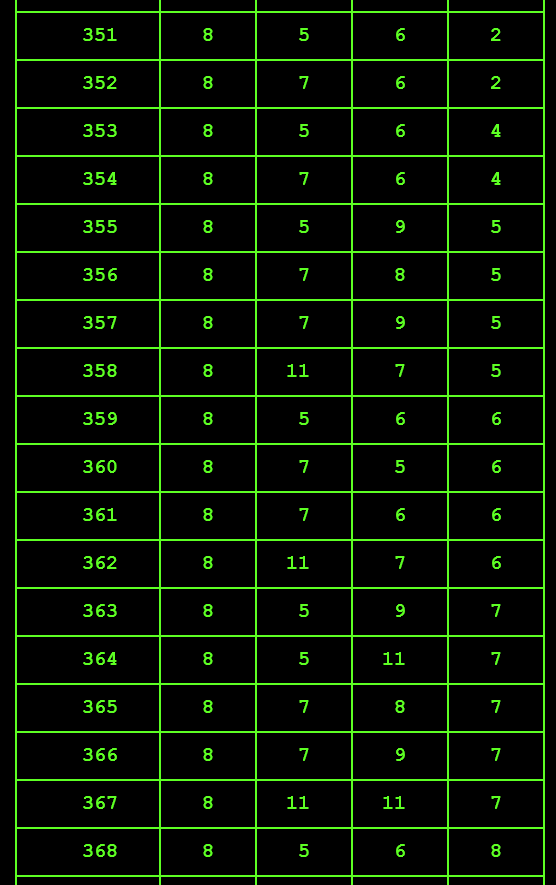
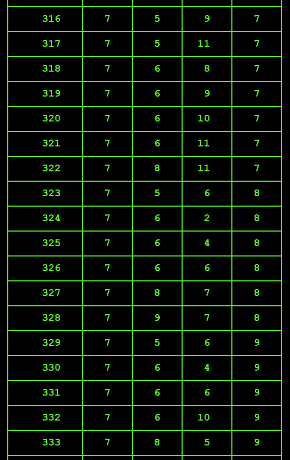
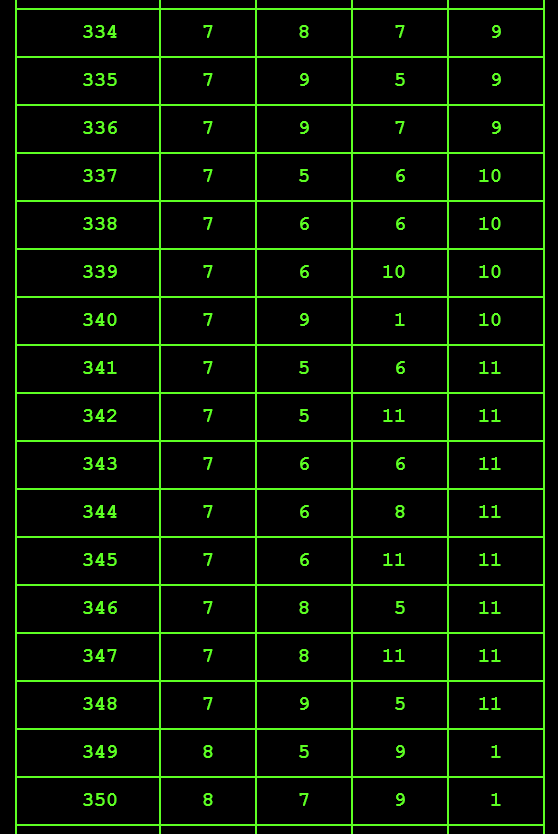
******

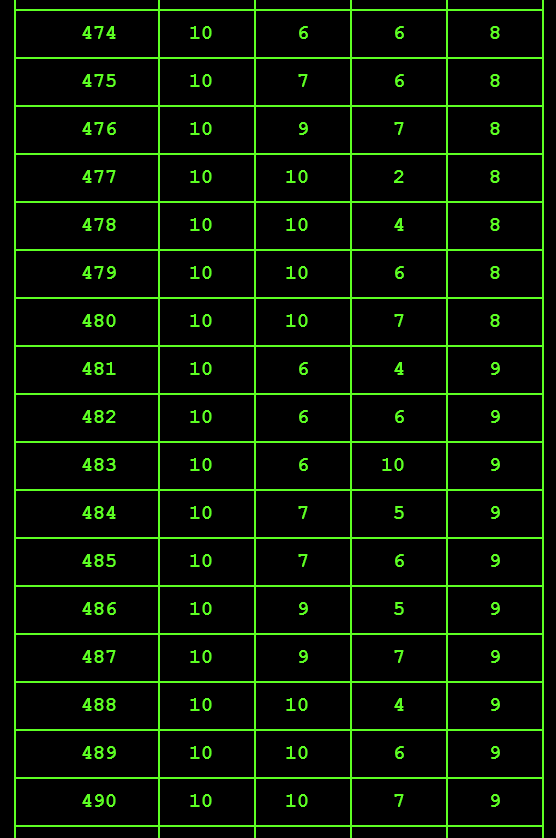
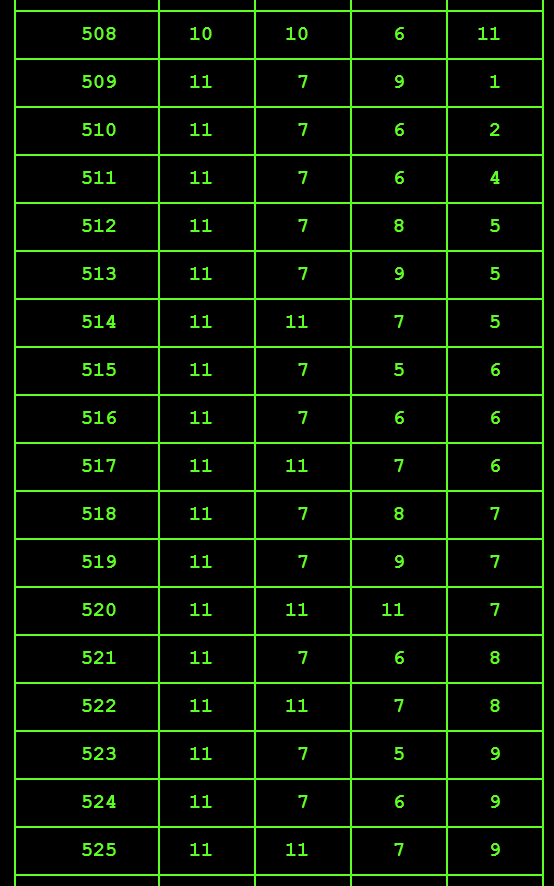


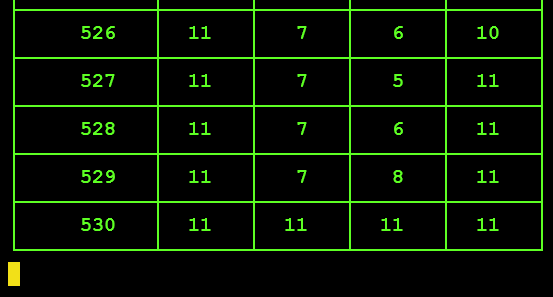
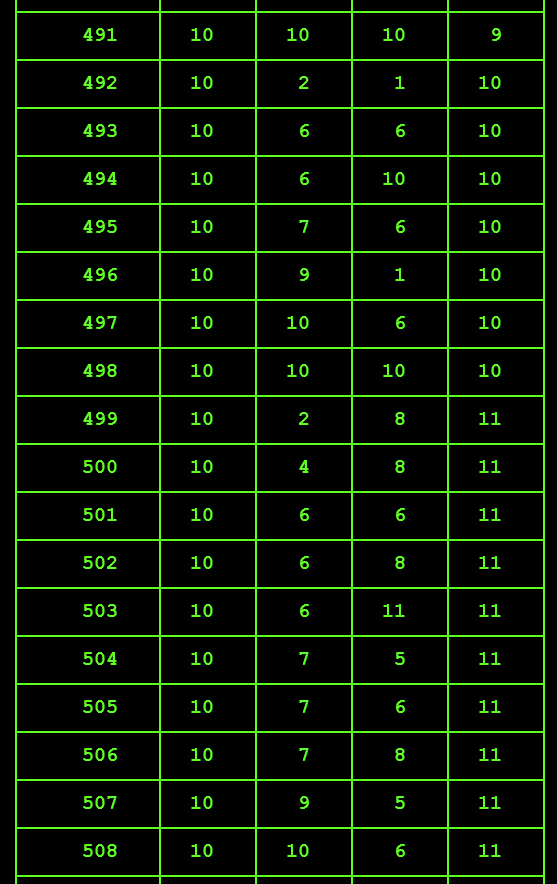
******

***Шляхи довжиною 3***

******

********

******

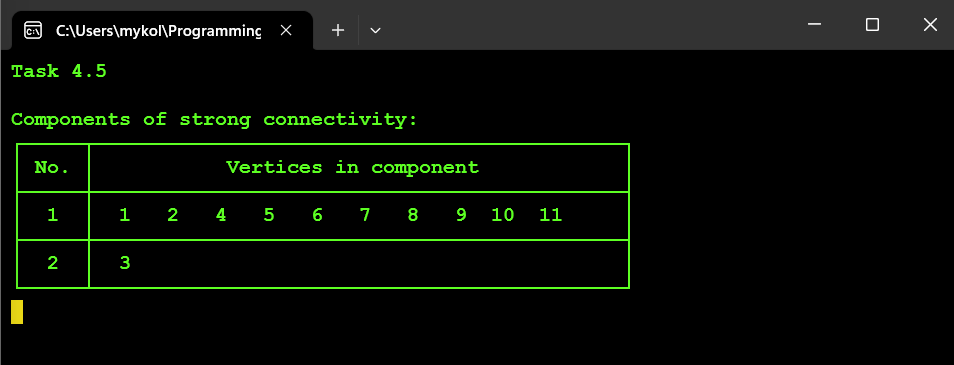
******

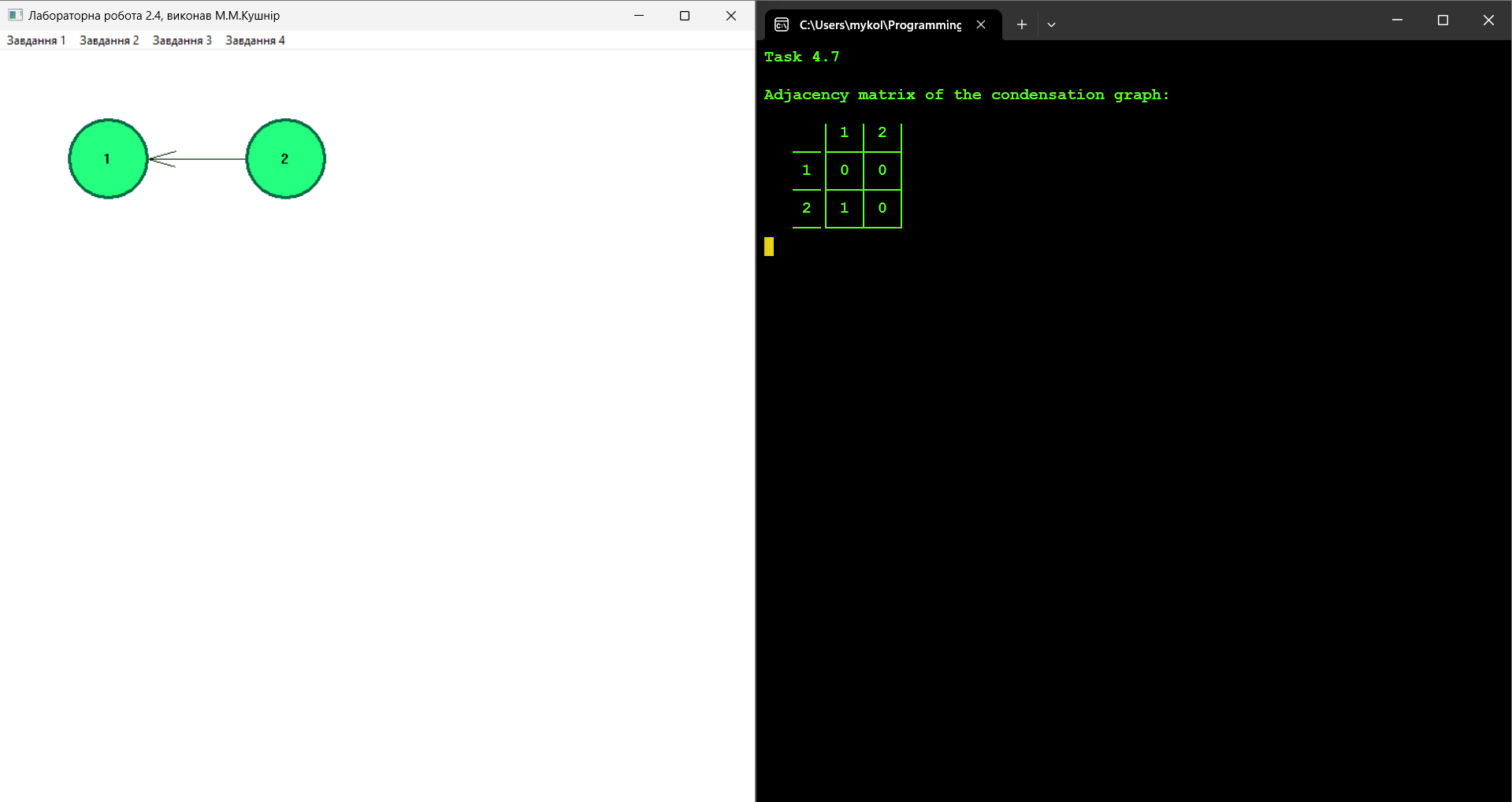
***Матриці досяжності та зв’язності модифікованого графа***

******

******

***Компоненти сильної зв’язності, граф конденсації та його матриця суміжності***

******



***Висновки***

Графи – це дуже корисна структура даних, яка має багато прикладних застосувань у сучасному програмуванні. У ході виконання цієї лабораторної роботи, я досліджував графи, задані матрицями суміжності. Я мав змогу попрактикуватися у знаходженні степенів ненапрямленого та напівстепенів напрямленого графа, утворенні матриць досяжності і зв’язності, побудові графа конденсації, утвореного з компонент сильної зв’язності. Найцікавішим завданням для мене було написання алгоритму для пошуку проміжних вузлів, на шляху від вершини А до вершини Б. Також виведення інформації, закодованої матрицею, у графічне вікно, дозволило наочно переконатися у істинності обчислень та отриманого на лекціях теоретичного матеріалу. Впевнений, що ці навички стануть у нагоді при роботі з графовими структурами даних.