Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут ім. Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра автоматизованих систем обробки інформації та управління

ЗВІТ

про виконання лабораторного практикуму №2

Виконав:

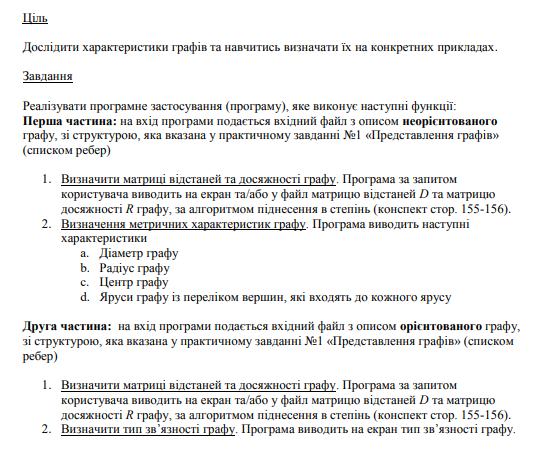
студент 1- го курсу ФІОТ

групи *ІП-91*

*Кінчур Вадим Вікторович*

Київ 2020

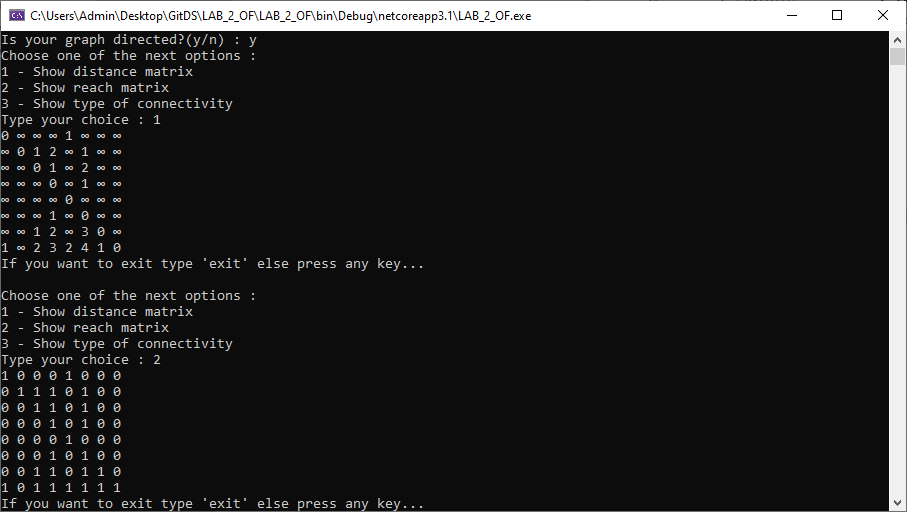
2. Умова лабораторної роботи

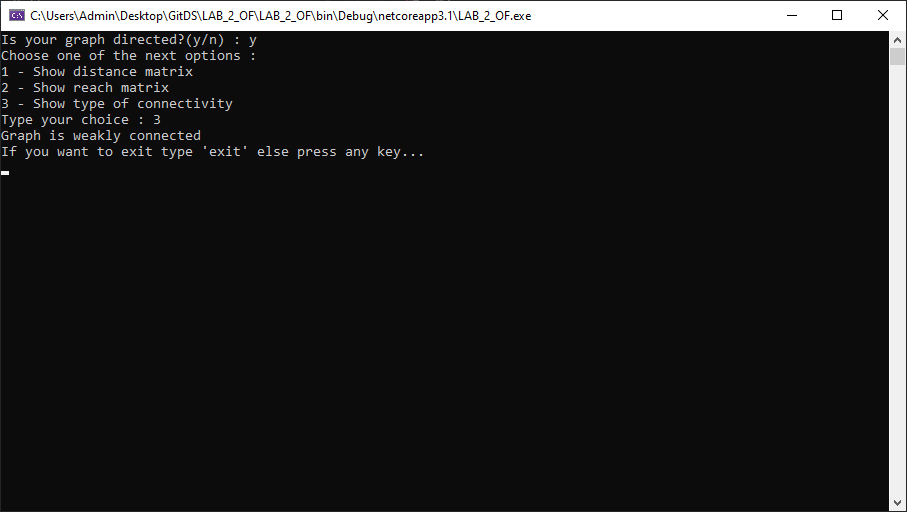


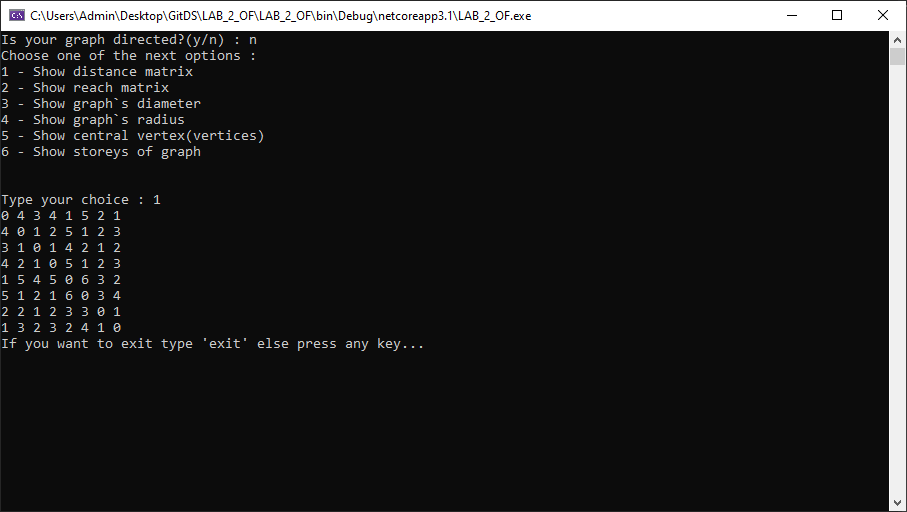
3. Програмний код (C#) :

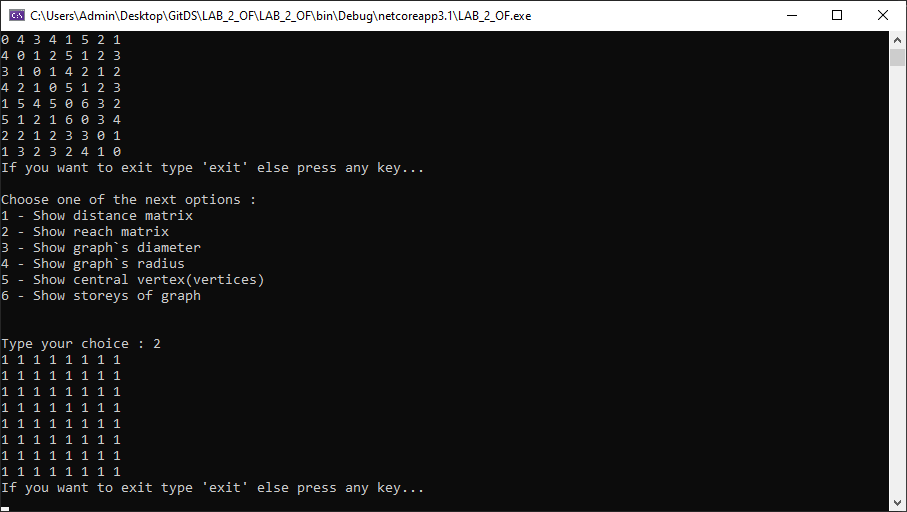
using System;  
using System.IO;  
using System.Collections.Generic;  
using System.Text;  
  
namespace WorkSpace  
**{** class Program  
 {  
 public static void Main(string[] args)  
 {  
 Parser Info = new Parser("input.txt");  
 (int vertexNum, int edgesNum, List<(int, int)> edgesList) = Info.GetInput();  
  
 System.Console.Write("Is your graph directed?(y/n) : ");  
 string graphType = System.Console.ReadLine();  
  
 if (graphType == "y")  
 {  
 DirectedGraph graph = new DirectedGraph(vertexNum, edgesNum, edgesList);  
 graph.StartMenu();  
 }  
 else  
 {  
 NotDirectedGraph graph = new NotDirectedGraph(vertexNum, edgesNum, edgesList);  
 graph.StartMenu();  
 }  
 }  
 }  
  
 class Parser  
 {  
 private readonly StreamReader file;  
 public Parser(string fileName)  
 {  
 string path = "../../../" + fileName;  
 file = new StreamReader(path);  
 }  
  
 private (int, int) ParseRow(string row)  
 {  
 string[] characters = row.Split(" ");  
 int[] numbers = Array.ConvertAll(characters, s => int.Parse(s));  
 return (numbers[0], numbers[1]);  
  
 }  
  
 public (int, int, List<(int, int)>) GetInput()  
 {  
 string line = file.ReadLine();  
 (int n, int m) = ParseRow(line);  
 List<(int, int)> edgesList = new List<(int, int)>();  
  
 for (int i = 0; i < m; ++i)  
 {  
 line = file.ReadLine();  
 if (line != null)  
 edgesList.Add(ParseRow(line));  
 }  
 return (n, m, edgesList);  
 }  
 }  
  
 class Matrix  
 {  
 private readonly int sizeRows;  
 private readonly int sizeColumns;  
 private readonly int[,] matrix;  
 public Matrix(int n, int m)  
 {  
 sizeRows = (n >= 0) ? n : 0;  
 sizeColumns = (m >= 0) ? m : 0;  
 matrix = new int[sizeRows, sizeColumns];  
 Console.OutputEncoding = Encoding.Unicode;  
 }  
 private bool NotOutOfTheRange(int row\_index, int column\_index)  
 {  
 return (row\_index >= 0 && row\_index < sizeRows) && (column\_index >= 0 && column\_index < sizeColumns);  
 }  
 public int this[int indexRow, int indexColumn]  
 {  
 set  
 {  
 matrix[indexRow, indexColumn] = (NotOutOfTheRange(indexRow, indexColumn)) ? value : 0;  
 }  
 get  
 {  
 return (NotOutOfTheRange(indexRow, indexColumn)) ? matrix[indexRow, indexColumn] : 0;  
 }  
 }  
 public int GetRowSize() => sizeRows;  
 public int getColSize() => sizeColumns;  
 public static Matrix operator \*(Matrix A, Matrix B)  
 {  
 Matrix C = new Matrix(A.GetRowSize(), B.getColSize());  
  
 for (int i = 0; i < A.GetRowSize(); ++i)  
 {  
 for (int j = 0; j < B.getColSize(); ++j)  
 {  
 for (int k = 0; k < A.getColSize(); ++k)  
 {  
 C[i, j] += A[i, k] \* B[k, j];  
 }  
 }  
 }  
 return C;  
 }  
 public void Print()  
 {  
 for (int i = 0; i < sizeRows; ++i)  
 {  
 for (int j = 0; j < sizeColumns; ++j)  
 {  
 if (matrix[i, j] == int.MaxValue)  
 Console.Write("\u221e ");  
 else  
 Console.Write($"{matrix[i, j]} ");  
 }  
 Console.WriteLine();  
 }  
 }  
 public void FillInf()  
 {  
 for (int i = 0; i < sizeRows; ++i)  
 {  
 for (int j = 0; j < sizeColumns; ++j)  
 {  
 matrix[i, j] = int.MaxValue;  
 }  
 }  
  
 }  
 public int GetMaximalElement()  
 {  
 int maximal = matrix[0, 0];  
 foreach (var value in matrix)  
 {  
 if (value > maximal)  
 {  
 maximal = value;  
 }  
 }  
 return maximal;  
 }  
 public int GetMaxInRow(int rowIndex)  
 {  
 int result = matrix[0, 0];  
 for (int j = 0; j < sizeColumns; ++j)  
 {  
 if (matrix[rowIndex, j] > result)  
 result = matrix[rowIndex, j];  
 }  
 return result;  
 }  
 public bool ContainsInRow(int rowIndex, int value)  
 {  
 for (int j = 0; j < sizeColumns; ++j)  
 {  
 if (matrix[rowIndex, j] == value)  
 return true;  
 }  
 return false;  
 }  
 }  
  
 class Graph  
 {  
 protected readonly int vertexNum;  
 protected readonly int edgesNum;  
 protected readonly List<(int, int)> edgesList;  
  
 public Graph(int n, int m, List<(int, int)> edgesArray)  
 {  
 vertexNum = (n >= 0) ? n : 0;  
 edgesNum = (m >= 0) ? m : 0;  
 edgesList = edgesArray;  
 }  
 protected virtual void ShowMenu()  
 {  
 Console.WriteLine("Choose one of the next options : ");  
 Console.WriteLine("1 - Show distance matrix\n" +  
 "2 - Show reach matrix");  
 }  
 protected virtual void WorkWithNumber(int number)  
 {  
 switch (number)  
 {  
 case 1:  
 GetDistanceMatrix().Print();  
 break;  
 case 2:  
 GetReachMatrix().Print();  
 break;  
 }  
 }  
 protected int EnterNumber(int down, int up)  
 {  
 int number = 0;  
  
 while (number > up || number < down)  
 {  
 Console.Write("\nType your choice : ");  
 bool checker = int.TryParse(Console.ReadLine(), out number);  
 if (number > up || number < down)  
 Console.WriteLine("Invalid input!");  
 }  
 return number;  
 }  
 protected virtual Matrix GetAdjacencyMatrix()  
 {  
 Matrix adjacencyMatrix = new Matrix(vertexNum, vertexNum);  
  
 foreach (var value in edgesList)  
 {  
 int start = value.Item1 - 1;  
 int finish = value.Item2 - 1;  
 adjacencyMatrix[start, finish] = 1;  
 adjacencyMatrix[finish, start] = 1;  
 }  
 return adjacencyMatrix;  
 }  
 protected void FillPath(ref Matrix ToFill, Matrix A, int path\_length)  
 {  
 for (int i = 0; i < A.GetRowSize(); ++i)  
 {  
 for (int j = 0; j < A.getColSize(); ++j)  
 {  
 if (i == j)  
 {  
 ToFill[i, j] = 0;  
 }  
 else if ((ToFill[i, j] == int.MaxValue) && (A[i, j] != 0))  
 {  
 ToFill[i, j] = path\_length;  
 }  
  
 }  
 }  
 }  
 protected Matrix GetDistanceMatrix()  
 {  
 Matrix resultMatrix = new Matrix(vertexNum, vertexNum);  
 Matrix toMultiple = GetAdjacencyMatrix();  
 Matrix adjacencyMatrix = GetAdjacencyMatrix();  
 resultMatrix.FillInf();  
  
 for (int i = 0; i < vertexNum; ++i)  
 {  
 FillPath(ref resultMatrix, toMultiple, i + 1);  
 toMultiple \*= adjacencyMatrix;  
 }  
 return resultMatrix;  
 }  
 protected void FillReach(ref Matrix ToFill, Matrix A)  
 {  
 for (int i = 0; i < vertexNum; ++i)  
 {  
 for (int j = 0; j < vertexNum; ++j)  
 {  
 if ((i == j) || (A[i, j] != 0))  
 ToFill[i, j] = 1;  
 }  
 }  
 }  
 public Matrix GetReachMatrix()  
 {  
 Matrix reachMatrix = new Matrix(vertexNum, vertexNum);  
 Matrix adjacencyMatrix = GetAdjacencyMatrix();  
 Matrix toMultiple = GetAdjacencyMatrix();  
  
 for (int i = 0; i < vertexNum; ++i)  
 {  
 FillReach(ref reachMatrix, toMultiple);  
 toMultiple \*= adjacencyMatrix;  
 }  
 return reachMatrix;  
 }  
 public virtual void StartMenu(int down, int up)  
 {  
 string answer = "";  
 while (answer != "exit")  
 {  
 ShowMenu();  
 WorkWithNumber(EnterNumber(down, up));  
 Console.WriteLine("If you want to exit type 'exit' else press any key...");  
 answer = Console.ReadLine();  
 }  
 }  
 }  
  
 class DirectedGraph : Graph  
 {  
 public DirectedGraph(int n, int m, List<(int, int)> edges\_list) :  
 base(n, m, edges\_list)  
 { }  
 protected override Matrix GetAdjacencyMatrix()  
 {  
 Matrix adjacencyMatrix = new Matrix(vertexNum, vertexNum);  
  
 foreach (var value in edgesList)  
 {  
 int start = value.Item1 - 1;  
 int finish = value.Item2 - 1;  
 adjacencyMatrix[start, finish] = 1;  
 }  
 return adjacencyMatrix;  
 }  
 protected override void ShowMenu()  
 {  
 base.ShowMenu();  
 Console.Write("3 - Show type of connectivity");  
 }  
 protected override void WorkWithNumber(int number)  
 {  
 base.WorkWithNumber(number);  
 switch (number)  
 {  
 case 3:  
 Console.WriteLine($"Graph is {GetConnectedType()}");  
 break;  
 }  
 }  
 public void StartMenu() => base.StartMenu(1, 3);  
  
 protected bool IsStrongConnected()  
 {  
 Matrix reachMatrix = GetReachMatrix();  
  
 for (int i = 0; i < vertexNum; ++i)  
 {  
 for (int j = 0; j < vertexNum; ++j)  
 {  
 if (reachMatrix[i, j] != 1 || reachMatrix[j, i] != 1)  
 {  
 return false;  
 }  
 }  
 }  
 return true;  
 }  
  
 protected bool IsOneSideConnected()  
 {  
 Matrix reachMatrix = GetReachMatrix();  
  
 for (int i = 0; i < vertexNum; ++i)  
 {  
 for (int j = 0; j < vertexNum; ++j)  
 {  
 if (reachMatrix[i, j] != 1 && reachMatrix[j, i] != 1)  
 {  
 return false;  
 }  
 }  
 }  
 return true;  
 }  
  
 protected bool IsWeaklyConnected()  
 {  
 Graph A = new Graph(vertexNum, edgesNum, edgesList);  
 Matrix reachMatrix = A.GetReachMatrix();  
  
 for (int i = 0; i < vertexNum; ++i)  
 {  
 for (int j = 0; j < vertexNum; ++j)  
 {  
 if (reachMatrix[i, j] != 1)  
 return false;  
 }  
 }  
 return true;  
 }  
  
 protected string GetConnectedType()  
 {  
 if (IsStrongConnected()) return "strong connected";  
 if (IsOneSideConnected()) return "one side connected";  
 if (IsWeaklyConnected()) return "weakly connected";  
 return "incorehent";  
 }  
 }  
  
 class NotDirectedGraph : Graph  
 {  
 public NotDirectedGraph(int n, int m, List<(int, int)> edgesList) :  
 base(n, m, edgesList)  
 { }  
  
 protected int GetRadius()  
 {  
 Matrix distanceMatrix = GetDistanceMatrix();  
 int radius = distanceMatrix.GetMaxInRow(0);  
  
 for (int i = 1; i < vertexNum; ++i)  
 {  
 int currentEccentricity = distanceMatrix.GetMaxInRow(i);  
 if (currentEccentricity < radius)  
 {  
 radius = currentEccentricity;  
 }  
 }  
 return radius;  
 }  
 protected int GetDiameter()  
 {  
 Matrix distanceMatrix = GetDistanceMatrix();  
 return distanceMatrix.GetMaximalElement();  
 }  
 protected List<int> GetCenter()  
 {  
 int radius = GetRadius();  
 Matrix distanceMatrix = GetDistanceMatrix();  
 List<int> centresList = new List<int>();  
 for (int i = 0; i < vertexNum; ++i)  
 {  
 int currentEccentricity = distanceMatrix.GetMaxInRow(i);  
 if (currentEccentricity == radius)  
 centresList.Add(i + 1);  
 }  
 return centresList;  
 }  
 protected override Matrix GetAdjacencyMatrix() => base.GetAdjacencyMatrix();  
 protected List<int> GetDistances()  
 {  
 List<int> distances = new List<int>();  
 Matrix distanceMatrix = GetDistanceMatrix();  
  
 for (int i = 0; i < vertexNum; ++i)  
 {  
 for (int j = 0; j < vertexNum; ++j)  
 {  
 if (!distances.Contains(distanceMatrix[i, j]))  
 {  
 distances.Add(distanceMatrix[i, j]);  
 }  
 }  
 }  
 return distances;  
 }  
 protected void PrintStoreys()  
 {  
 List<int> distances = GetDistances();  
 Matrix distancesMatrix = GetDistanceMatrix();  
 distances.Sort();  
  
 foreach (var value in distances)  
 {  
 List<int> temp = new List<int>();  
 for (int i = 0; i < vertexNum; ++i)  
 {  
 if (distancesMatrix.ContainsInRow(i, value))  
 {  
 temp.Add(i + 1);  
 }  
 }  
 if (value != int.MaxValue)  
 Console.Write($"Distance = {value} : ");  
 else  
 Console.Write("Distance = \u221e : ");  
 temp.ForEach(el => Console.Write($"{el} "));  
 Console.WriteLine();  
 }  
 }  
 protected override void ShowMenu()  
 {  
 Console.WriteLine("Choose one of the next options : ");  
 Console.WriteLine("1 - Show distance matrix\n" +  
 "2 - Show reach matrix\n" +  
 "3 - Show graph`s diameter\n" +  
 "4 - Show graph`s radius\n" +  
 "5 - Show central vertex(vertices)\n" +  
 "6 - Show storeys of graph\n");  
 }  
 protected override void WorkWithNumber(int number)  
 {  
 base.WorkWithNumber(number);  
 switch (number)  
 {  
 case 3:  
 int diameter = GetDiameter();  
 if (diameter != int.MaxValue)  
 Console.WriteLine($"Diameter = {diameter}");  
 else  
 Console.WriteLine("Diameter = \u221e ");  
 break;  
 case 4:  
 int radius = GetRadius();  
 if (radius != int.MaxValue)  
 Console.WriteLine($"Radius = {radius}");  
 else  
 Console.WriteLine("Radius = \u221e ");  
 break;  
 case 5:  
 GetCenter().ForEach(el => Console.Write($"{el} "));  
 Console.WriteLine();  
 break;  
 case 6:  
 PrintStoreys();  
 break;  
 }  
 }  
 public void StartMenu()  
 {  
 base.StartMenu(1, 6);  
 }  
 }  
**}**

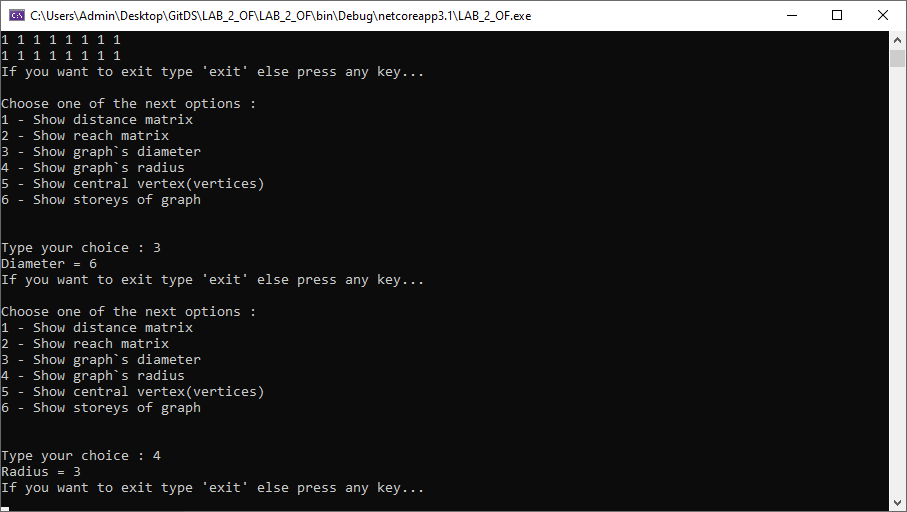
4. Результати виконання

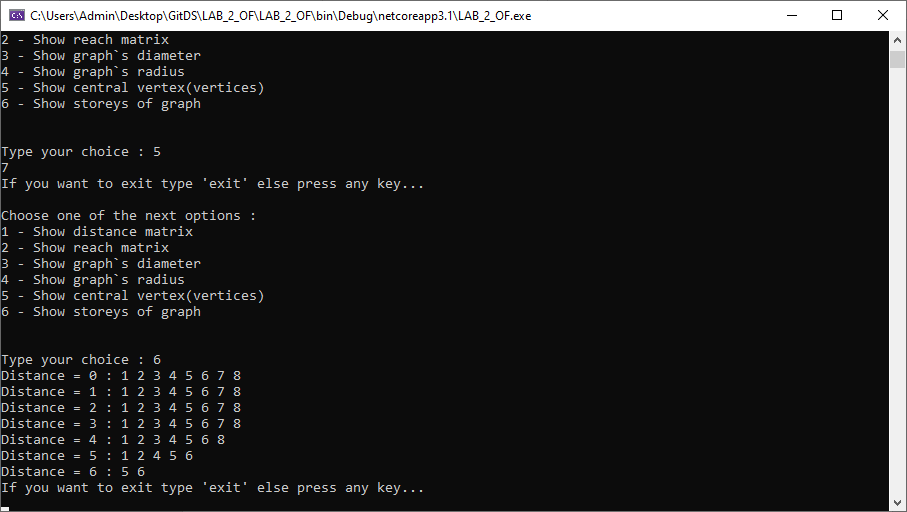












**Вхідні дані :**

8 9

1 5

2 6

2 3

3 4

6 4

4 6

7 3

8 7

8 1