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 №4

з дисципліни
«Алгоритми і структури даних»

Виконав

Студент групи ІМ-22
Тимофеєв Даниіл Костянтинович
номер у списку групи: 20

Перевірила:

Молчанова А. А.

Київ 2023

Постановка задачі : 1. Представити напрямлений граф з заданими параметрами так само, як у лабораторній роботі No3.

Відміна: матриця A напрямленого графа за варіантом формується за функціями:

srand(п1 п2 п3 п4); 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 A2 і A3 , відповідно. Матриця досяжності та компоненти сильної зв’язності слід шукати за допомогою операції транзитивного замикання.

Завдання варіанту № 20 :

Номер – 2220

Число вершин = 10 +2 = 12.

Розміщення вершин колом , бо n4 = 0;

Srand(2220)

Текст програми мовою С.

Файл *main.c* :

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <math.h>
#include "props.h"
#define vertices 12
#define IDC_BUTTON 1
#define IDC_BUTTON2 2
#define IDC_BUTTON3 3
#define IDC_BUTTON4 4

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

char ProgName[] = "Lab 4";

struct coordinates {
    double nx[vertices];
    double ny[vertices];
    double loopX[vertices];
    double loopY[vertices];
};




void arrow(double fi, double px, double py, HDC hdc) {
  double lx, ly, rx, ry;
  lx = px + 15 * cos(fi + 0.3);
  rx = px + 15 * cos(fi - 0.3);
  ly = py + 15 * sin(fi + 0.3);
  ry = py + 15 * sin(fi - 0.3);
  MoveToEx(hdc, lx, ly, NULL);
  LineTo(hdc, px, py);
  LineTo(hdc, rx, ry);
}

void depictArch(int startX, int startY, int finalX, int finalY, int
archInterval, HDC hdc) {
  XFORM transformMatrix;
  XFORM initialMatrix;
  GetWorldTransform(hdc, &initialMatrix);

  double angle = atan2(finalY - startY, finalX - startX) - M_PI / 2.0;
  transformMatrix.eM11 = (FLOAT) cos(angle);
  transformMatrix.eM12 = (FLOAT) sin(angle);
  transformMatrix.eM21 = (FLOAT) (-sin(angle));
  transformMatrix.eM22 = (FLOAT) cos(angle);
  transformMatrix.eDx = (FLOAT) startX;
  transformMatrix.eDy = (FLOAT) startY;
  SetWorldTransform(hdc, &transformMatrix);

  const double archWidthRatio = 0.75;
  double archLength = sqrt((finalX - startX) * (finalX - startX) + (finalY -
startY) * (finalY - startY));
  double radiusOfVertex = 15.0;
  double semiMinorAxis = archWidthRatio * archLength;
  double semiMajorAxis = archLength / 2;

  double ellipseStartY = semiMajorAxis;
```

```
  double vertexAreaSquared = semiMajorAxis * semiMajorAxis * radiusOfVertex *
radiusOfVertex;
  double semiAxesSquared = semiMinorAxis * semiMinorAxis * semiMajorAxis *
semiMajorAxis;
  double distanceFromCenter = semiMinorAxis * semiMinorAxis * ellipseStartY *
ellipseStartY;
  double distanceFromVertex = semiMinorAxis * semiMinorAxis * radiusOfVertex
* radiusOfVertex;
  double semiMinorAxisPow = pow(semiMinorAxis, 4);

  double intersection = semiMajorAxis *
                        sqrt(vertexAreaSquared - semiAxesSquared +
distanceFromCenter - distanceFromVertex +
                             semiMinorAxisPow);
  double semiMinorAxisSquaredEllipseStartY = semiMinorAxis * semiMinorAxis *
ellipseStartY;
  double denominator = -semiMajorAxis * semiMajorAxis + semiMinorAxis *
semiMinorAxis;


  double contactYRightTop = (semiMinorAxisSquaredEllipseStartY -
intersection) / denominator;
  double contactXRightTop = sqrt(radiusOfVertex * radiusOfVertex -
contactYRightTop * contactYRightTop);
  double contactYBottom = archLength - contactYRightTop;
  double contactXLeftBottom = -contactXRightTop;

  if (archInterval <= vertices / 2) {
    Arc(hdc, -archWidthRatio * archLength, archLength, archWidthRatio *
archLength, 0, 0, 0, 0, archLength);
    double angleOfArrow = -atan2(archLength - contactYBottom,
contactXLeftBottom) + 0.3 / 3;
    arrow(angleOfArrow, contactXLeftBottom, contactYBottom, hdc);
  } else {
    Arc(hdc, -archWidthRatio * archLength, archLength, archWidthRatio *
archLength, 0, 0, archLength, 0, 0);
    double angleOfArrow = -atan2(archLength - contactYBottom, -
contactXLeftBottom) - 0.3 / 3;
    arrow(angleOfArrow, -contactXLeftBottom, contactYBottom, hdc);
  }

  SetWorldTransform(hdc, &initialMatrix);

}

void depictDirectedGraph(int n, int centerX, int centerY, int radiusOfGraph,
int radiusOfVertex, int radiusOfLoop, double angle,
                         struct coordinates coordinates, double **matrix,
                         HPEN KPen, HPEN GPen, HDC hdc) {


  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      MoveToEx(hdc, coordinates.nx[i], coordinates.ny[i], NULL);
      if ((j >= i && matrix[i][j] == 1) || (j <= i && matrix[i][j] == 1 &&
matrix[j][i] == 0)) {
        if (i == j) {
          SelectObject(hdc, GPen);

          Ellipse(hdc, coordinates.loopX[i] - radiusOfLoop,
coordinates.loopY[i] - radiusOfLoop,
                  coordinates.loopX[i] + radiusOfLoop, coordinates.loopY[i] +
radiusOfLoop);
```

```
          double radiusOfContact = radiusOfGraph + radiusOfLoop / 2.;
          double triangleHeight = sqrt(3) * radiusOfVertex / 2.;
          double loopAngle = atan2(triangleHeight, radiusOfContact);
          double contactDistance = sqrt(radiusOfContact * radiusOfContact +
triangleHeight * triangleHeight);
          double angleToContactVertex = atan2(coordinates.ny[i] - centerY,
coordinates.nx[i] - centerX);

          double contactPointX = centerX + contactDistance *
cos(angleToContactVertex + loopAngle);
          double contactPointY = centerY + contactDistance *
sin(angleToContactVertex + loopAngle);

          double curvatureAngle = angleToContactVertex + 0.3 / 2.;
          arrow(curvatureAngle, contactPointX, contactPointY, hdc);
          SelectObject(hdc, KPen);
        } else {
          LineTo(hdc, coordinates.nx[j], coordinates.ny[j]);
          double line_angle = atan2(coordinates.ny[i] - coordinates.ny[j],
coordinates.nx[i] - coordinates.nx[j]);
          arrow(line_angle, coordinates.nx[j] + radiusOfVertex *
cos(line_angle),
                coordinates.ny[j] + radiusOfVertex * sin(line_angle), hdc);
        }
      } else if (j < i && matrix[i][j] == 1 && matrix[j][i] == 1) {
        depictArch(coordinates.nx[i], coordinates.ny[i], coordinates.nx[j],
coordinates.ny[j], fabs(i - j), hdc);
      }

    }
  }

}

void depictUndirectedGraph(int centerX, int centerY, int radiusOfGraph, int
radiusOfVertex, int radiusOfLoop, double angle,
                           struct coordinates coordinates, double **matrix,
                           HPEN KPen, HPEN GPen, HDC hdc) {
  for (int i = 0; i < vertices; ++i) {

    for (int j = 0; j < vertices; ++j) {
      MoveToEx(hdc, coordinates.nx[i], coordinates.ny[i], NULL);

      if (matrix[i][j] == 1) {

        if (i == j) {
          SelectObject(hdc, GPen);
          Ellipse(hdc, coordinates.loopX[i] - radiusOfLoop,
coordinates.loopY[i] - radiusOfLoop,
                  coordinates.loopX[i] + radiusOfLoop, coordinates.loopY[i] +
radiusOfLoop);
          SelectObject(hdc, KPen);
        } else {
          LineTo(hdc, coordinates.nx[j], coordinates.ny[j]);
        }

      }
    }
  }

}
```

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpszCmdLine, int nCmdShow) {
  WNDCLASS w;
  w.lpszClassName = ProgName;
  w.hInstance = hInstance;
  w.lpfnWndProc = WndProc;
  w.hCursor = LoadCursor(NULL, IDC_ARROW);
  w.hIcon = 0;
  w.lpszMenuName = 0;
  w.hbrBackground = WHITE_BRUSH;
  w.style = CS_HREDRAW | CS_VREDRAW;
  w.cbClsExtra = 0;
  w.cbWndExtra = 0;
  if (!RegisterClass(&w)) {
    return 0;
  }
  HWND hWnd;
  MSG lpMsg;
  hWnd = CreateWindow(ProgName,
                      (LPCSTR) "Lab 4.  by Daniil Timofeev IM-22",
                      WS_OVERLAPPEDWINDOW,
                      100,
                      100,
                      1200,
                      700,
                      (HWND) NULL,
                      (HMENU) NULL,
                      (HINSTANCE) hInstance,
                      (HINSTANCE) NULL);
  ShowWindow(hWnd, nCmdShow);
  while (GetMessage(&lpMsg, hWnd, 0, 0)) {
    TranslateMessage(&lpMsg);
    DispatchMessage(&lpMsg);
  }
  return (lpMsg.wParam);
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT messg, WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  PAINTSTRUCT ps;
  HWND Button_directed;
  HWND Button_undirected;
  HWND Button_modified;
  int state = 0;
  switch (messg) {
    case WM_CREATE: {
      Button_directed = CreateWindow(
              (LPCSTR) "BUTTON",
              (LPCSTR) "Switch to Directed",
              WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,
              700,
              30,
              160,
              50,
              hWnd,
              (HMENU) IDC_BUTTON,
              (HINSTANCE) GetWindowLongPtr(hWnd, GWLP_HINSTANCE),
              NULL);
      Button_undirected = CreateWindow(
```

```
            (LPCSTR) "BUTTON",
            (LPCSTR) "Switch to Undirected",
            WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,
            700,
            600,
            160,
            50,
            hWnd,
            (HMENU) IDC_BUTTON2,
            (HINSTANCE) GetWindowLongPtr(hWnd, GWLP_HINSTANCE),
            NULL);
    Button_modified = CreateWindow(
            (LPCSTR) "BUTTON",
            (LPCSTR) "Switch to Modified",
            WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,
            1000,
            30,
            160,
            50,
            hWnd,
            (HMENU) IDC_BUTTON3,
            (HINSTANCE) GetWindowLongPtr(hWnd, GWLP_HINSTANCE),
            NULL);
    Button_modified = CreateWindow(
            (LPCSTR) "BUTTON",
            (LPCSTR) "Switch to Condensation",
            WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,
            1000,
            600,
            160,
            50,
            hWnd,
            (HMENU) IDC_BUTTON4,
            (HINSTANCE) GetWindowLongPtr(hWnd, GWLP_HINSTANCE),
            NULL);
    return 0;
}
  case WM_COMMAND: {
    switch (LOWORD(wParam)) {

      case IDC_BUTTON:
        state = 0;
        InvalidateRect(hWnd, NULL, FALSE);
        break;

      case IDC_BUTTON2:
        state = 1;
        InvalidateRect(hWnd, NULL, FALSE);
        break;

      case IDC_BUTTON3:
        state = 2;
        InvalidateRect(hWnd, NULL, FALSE);
        break;
      case IDC_BUTTON4:
        state = 3;
        InvalidateRect(hWnd, NULL, FALSE);
        break;
    }
  }
  case WM_PAINT :
    hdc = BeginPaint(hWnd, &ps);
    SetGraphicsMode(hdc, GM_ADVANCED);
    HPEN BPen = CreatePen(PS_SOLID, 2, RGB(50, 0, 255));
```

```
      HPEN KPen = CreatePen(PS_SOLID, 1, RGB(20, 20, 5));
      HPEN GPen = CreatePen(PS_SOLID, 2, RGB(0, 255, 0));
      HPEN NoPen = CreatePen(PS_NULL, 0, RGB(0, 0, 0));
      SelectObject(hdc, NoPen);
      Rectangle(hdc, 0, 0, 670, 700);


      char *nn[vertices] = {"1", "2", "3", "4", "5", "6", "7", "8", "9",
"10\0", "11\0", "12\0"};



      struct coordinates coordinates;

      double circleRadius = 200;
      double vertexRadius = circleRadius / 12;

      double loopRadius = vertexRadius;
      double dtx = vertexRadius / 2.5;

      double circleCenterX = 370;
      double circleCenterY = 360;


      double angleAlpha = 2.0 * M_PI / (double) vertices;
      for (int i = 0; i < vertices; i++) {

        double sinAlpha = sin(angleAlpha * (double) i);
        double cosAlpha = cos(angleAlpha * (double) i);
        coordinates.nx[i] = circleCenterX + circleRadius * sinAlpha;
        coordinates.ny[i] = circleCenterY - circleRadius * cosAlpha;
        coordinates.loopX[i] = circleCenterX + (circleRadius + loopRadius) *
sinAlpha;
        coordinates.loopY[i] = circleCenterY - (circleRadius + loopRadius) *
cosAlpha;

      }


      double **T = randm(vertices);
      double coefficient = 1.0 - 0.01 - 0.01 - 0.3;
      double **A = mulmr(coefficient, T, vertices);

      int initialXOofRandMatrix = 750;
      int initialYofRandMatrix = 150;


      double **R = randm(vertices);
      double **C = symmetricMatrix(mulmr(coefficient, R, vertices),
vertices);
      int initialXOofSymmMatrix = initialXOofRandMatrix;
      int initialYofSymmMatrix = initialYofRandMatrix + 210;



      double** K = randm(vertices);
      double modifiedCoefficient = 1.0 - 0.005 - 0.005 - 0.27;
      double** D = mulmr(modifiedCoefficient, K, vertices);
      int initialXOofModMatrix = initialXOofRandMatrix + 210;
      int initialYofModMatrix = initialYofRandMatrix;


      double **condensationMatrix =
condensationMatrixWithCoef(modifiedCoefficient);
```

```
      double **matrix =
generateAdjacencyMatrixFromStrongComponents(condensationMatrix);
      double** reachabilityMatrix = calculateReachabilityMatrix(D);
      double** connectivityMatrix =
strongConnectivityMatrix(reachabilityMatrix);

      int amount = countStrongComponents(connectivityMatrix);
      printf("%d", amount);



      SelectObject(hdc, GetStockObject(HOLLOW_BRUSH));
      SelectObject(hdc, KPen);


     if (state == 0) {
          depictDirectedGraph(vertices,circleCenterX, circleCenterY,
circleRadius, vertexRadius, loopRadius, angleAlpha,
                              coordinates, A, KPen, GPen, hdc);
        }
     if (state == 1) {
          depictUndirectedGraph(circleCenterX, circleCenterY, circleRadius,
vertexRadius, loopRadius, angleAlpha,
                                coordinates, C, KPen, GPen, hdc);
      }
     if (state == 2) {
       depictDirectedGraph(vertices,circleCenterX, circleCenterY,
circleRadius, vertexRadius, loopRadius, angleAlpha,
                           coordinates, D, KPen, GPen, hdc);
     }

      if (state == 3) {
        depictDirectedGraph(amount,circleCenterX, circleCenterY,
circleRadius, vertexRadius, loopRadius, angleAlpha,
                           coordinates, matrix, KPen, GPen, hdc);
      }



      SelectObject(hdc, BPen);
      SelectObject(hdc, GetStockObject(DC_BRUSH));
      SetDCBrushColor(hdc, RGB(204, 204, 255));
      SetBkMode(hdc, TRANSPARENT);

      int length = state == 3
              ? amount
              : vertices;

      for (int i = 0; i < length; ++i) {
        Ellipse(hdc, coordinates.nx[i] - vertexRadius, coordinates.ny[i] -
vertexRadius,
                coordinates.nx[i] + vertexRadius, coordinates.ny[i] +
vertexRadius);
        TextOut(hdc, coordinates.nx[i] - dtx, coordinates.ny[i] -
vertexRadius / 2, nn[i], 2);
      }


      EndPaint(hWnd, &ps);

      freeMatrix(A, vertices);
      freeMatrix(C, vertices);
      freeMatrix(D, vertices);
      freeMatrix(matrix, vertices);
```

```
      freeMatrix(condensationMatrix, vertices);
      freeMatrix(connectivityMatrix, vertices);
      freeMatrix(reachabilityMatrix, vertices);
    case WM_DESTROY:
      PostQuitMessage(0);
      break;
    default:
      return (DefWindowProc(hWnd, messg, wParam, lParam));
  }
  return 0;
}
```

Файл *components*.c :

```
#define vertices 12
#include <stdlib.h>
#include <stdio.h>
#include "props.h"



double **randm(int n) {
  srand(2220);
  double **matrix = (double **) malloc(sizeof(double *) * n);
  for (int i = 0; i < n; i++) {
    matrix[i] = (double *) malloc(sizeof(double) * n);
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      matrix[i][j] = (double) (rand() * 2.0) / (double) RAND_MAX;
    }
  }
  return matrix;

}

double **mulmr(double coef, double **matrix, int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      matrix[i][j] *= coef;
      matrix[i][j] = matrix[i][j] < 1 ? 0 : 1;
    }
  }
  return matrix;
}


double **symmetricMatrix(double **matrix, int n) {
  double **symmetrical = (double **) malloc(n * sizeof(double *));
  for (int i = 0; i < n; ++i) {
    symmetrical[i] = (double *) malloc(n * sizeof(double));
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      symmetrical[i][j] = matrix[i][j];
    }
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (symmetrical[i][j] != symmetrical[j][i]) {
        symmetrical[i][j] = 1;
        symmetrical[j][i] = 1;
```

```
      }
    }
  }
  return symmetrical;
}

void freeMatrix(double **matrix, int n) {
  for (int i = 0; i < n; ++i) {
    free(matrix[i]);
  }
  free(matrix);
}


int* graphDegrees(double** matrix) {
  const int number = vertices;
  int* vertexDegree;
  vertexDegree = malloc(number * sizeof(int));
  int vertexDegreeCounter;
  for (int i = 0; i < number; ++i) {
    vertexDegreeCounter = 0;
    for (int j = 0; j < number; ++j) {
      if(matrix[i][j] && i == j) vertexDegreeCounter += 2;
      else if(matrix[i][j]) vertexDegreeCounter++;
    }
    vertexDegree[i] = vertexDegreeCounter;
  }
  return vertexDegree;
}



int* halfDegreeEntry(double** matrix) {
  const int number = vertices;
  int* vertexDegree;
  vertexDegree = malloc(number * sizeof(int));
  int vertexDegreeCounter;
  for (int j = 0; j < number; ++j) {
    vertexDegreeCounter = 0;
    for (int i = 0; i < number; ++i) {
      if(matrix[i][j] ) vertexDegreeCounter++;
    }
    vertexDegree[j] = vertexDegreeCounter;
  }
  return vertexDegree;
}

int* halfDegreeExit(double** matrix) {
  const int number = vertices;
  int* vertexDegree;
  vertexDegree = malloc(number * sizeof(int));
  int vertexDegreeCounter;
  for (int i = 0; i < number; ++i) {
    vertexDegreeCounter = 0;
    for (int j = 0; j < number; ++j) {
      if(matrix[i][j]) vertexDegreeCounter++;
    }
    vertexDegree[i] = vertexDegreeCounter;
  }
  return vertexDegree;
}

int* summarizeHalfDegrees(const int* exit, const int* entry) {
  const int number = vertices;
```

```
  int* vertexDegree = malloc(number * sizeof(int));
  for (int i = 0; i < number; ++i) {
    vertexDegree[i] = exit[i] + entry[i];
  }

  return vertexDegree;
}



int* findTerminalVertices(const int* degreesArray) {
  const int number = vertices;
  int* terminalVertices = calloc(number, sizeof(int));
  int position = 0;
  terminalVertices[position++] = 0;
  for (int i = 0; i < number; ++i) {
    if(degreesArray[i] == 1) terminalVertices[position++] = i + 1;
  }
  return terminalVertices;
}

int* findIsolatedVertices(const int* degreesArray) {
  const int number = vertices;
  int* isolatedVertices = calloc(number, sizeof(int));
  int position = 0;
  isolatedVertices[position++] = 0;

  for (int i = 0; i < number; ++i) {
    if(degreesArray[i] == 0) isolatedVertices[position++] = i + 1;
  }

  return isolatedVertices;
}

int checkHomogeneity(const int* degreesArray) {
  const int number = vertices;
  int firstDegree = degreesArray[0];
  for (int i = 1; i < number; ++i) {
    if(degreesArray[i] != firstDegree) return 0;
    firstDegree = degreesArray[i];
  }
  return 1;
}

double** summarizeMatrices(double** AMatrix, double** BMatrix) {
  const int number = vertices;
  double **summedMatrix = (double **) malloc(sizeof(double *) * number);
  for (int i = 0; i < number; i++) {
    summedMatrix[i] = (double *) malloc(sizeof(double) * number);
    for (int j = 0; j < number; ++j) {
      summedMatrix[i][j] = AMatrix[i][j] + BMatrix[i][j];
    }
  }
  return summedMatrix;
}

double** multiplyMatrices(double** AMatrix, double** BMatrix) {
  const int number = vertices;
  double **multipliedMatrix = (double **) malloc(sizeof(double *) * number);
  for (int i = 0; i < number; ++i) {
    multipliedMatrix[i] = (double *) malloc(sizeof(double) * number);
    for (int j = 0; j < number; ++j) {
      multipliedMatrix[i][j] = 0;
      for (int e = 0; e < number; ++e) {
```

```
        multipliedMatrix[i][j] += AMatrix[i][e] * BMatrix[e][j];
      }
    }
  }
  return multipliedMatrix;
}

double** copyMatrix(double** matrix) {
  const int number = vertices;
  double **copiedMatrix = (double **) malloc(sizeof(double *) * number);
  for (int i = 0; i < number; ++i) {
    copiedMatrix[i] = (double *) malloc(sizeof(double) * number);
    for (int j = 0; j < number; ++j) {
      copiedMatrix[i][j] = matrix[i][j];
    }
  }
  return copiedMatrix;
}


void booleanConversion(double** matrix) {
  const int number = vertices;
  for (int i = 0; i < number; ++i) {
    for (int j = 0; j < number; ++j) {
      if (matrix[i][j]) matrix[i][j] = 1;
    }
  }
}

double **directedMatrix(double K) {
  double **T = randm(vertices);
  double **A = mulmr(K, T,vertices);
  return A;
}

double** calculateReachabilityMatrix(double** matrix) {
  const int number = vertices;
  double **copy = copyMatrix(matrix);
  double **sum = copy;
  double **prev = copy;
  double **tempPrev, **tempSum;

  for (int i = 1; i < number - 1; i++) {
    tempPrev = multiplyMatrices(prev, matrix);
    tempSum = summarizeMatrices(sum, tempPrev);
    freeMatrix(sum,number);
    freeMatrix(prev,number);
    prev = tempPrev;
    sum = tempSum;
  }

  for (int i = 0; i < number; i++) {
    sum[i][i] += 1;
  }

  freeMatrix(prev,number);
  booleanConversion(sum);

  return sum;
}


void depthFirstSearch(double** connectivityMatrix, int startVertex, double*
component, int* visited) {
```

```
  const int number = vertices;
  visited[startVertex] = 1;
  component[startVertex] = 1;
  for (int adjacentVertex = 0; adjacentVertex < number; ++adjacentVertex) {
    if(!visited[adjacentVertex] &&
connectivityMatrix[startVertex][adjacentVertex]) {
      depthFirstSearch(connectivityMatrix, adjacentVertex,component,visited);
    }
  }
}

double** transposeMatrix(double** matrix,  int number) {
  number = vertices;
  double **transposedMatrix = malloc(number * sizeof(double*));
  for (int i = 0; i < number; ++i) {
    transposedMatrix[i] = malloc(number * sizeof(double ));
    for (int j = 0; j < number; ++j) {
      transposedMatrix[i][j] = matrix[j][i];
    }
  }
  return transposedMatrix;

}

int countNonZeroEntries(double **matrix) {
  int numVertices = vertices;
  int count = 0;

  for (int row = 0; row < numVertices; row++) {
    for (int col = 0; col < numVertices; col++) {
      if (matrix[row][col]) {
        count++;
        row++;
      }
    }
  }
  return count;
}


double** findStrongComponents(double** strongMatrix) {
  const int number = vertices;
  int* visitedVertex = calloc(number, sizeof(int));
  double** connectedComponents = calloc(number, sizeof(double *));
  for (int i = 0; i < number; i++) {
    connectedComponents[i] = calloc(number, sizeof(double));
  }

  for (int i = 0; i < number; ++i) {
    if(!visitedVertex[i]) {
      depthFirstSearch(strongMatrix, i, connectedComponents[i],
visitedVertex);
    }
  }

  free(visitedVertex);
  return connectedComponents;
}

double** strongConnectivityMatrix(double **reachabilityMatrix) {
  const int number = vertices;
  double** transposedMatrix = transposeMatrix(reachabilityMatrix, number);
  double **strongConnectivityMatrix = malloc(number* sizeof(double*));
  for (int i = 0; i < number; i++) {
```

```
    strongConnectivityMatrix[i] = malloc(number* sizeof(double));
    for (int j = 0; j < number; j++) {
      strongConnectivityMatrix[i][j] = reachabilityMatrix[i][j] *
transposedMatrix[i][j];
    }
  }
  freeMatrix(transposedMatrix, number);
  return strongConnectivityMatrix;
}

void condensationMatrix(double** strongComponents) {
  int numComponents = countNonZeroEntries(strongComponents);

  double **adjacencyMatrix = calloc(numComponents, sizeof(double*));
  for (int i = 0; i < numComponents; i++) {
    adjacencyMatrix[i] = calloc(numComponents, sizeof(double));
  }

  int position = 1;
  for (int i = 0; i < vertices; ++i) {
    if(!strongComponents[0][i]) {
      adjacencyMatrix[0][position] = 1;
      position++;
    }
  }

  for (int i = 0; i < numComponents; i++) {
    for (int j = 0; j < numComponents; j++) {
      printf("%.0lf  ", adjacencyMatrix[i][j]);
    }
    printf("\n");
  }

  freeMatrix(adjacencyMatrix, vertices);

}
double **generateAdjacencyMatrixFromStrongComponents(double **components) {
  const int number = vertices;
  double **matrix = calloc(number, sizeof(size_t*));
  for (int i = 0; i < number; i++) {
    matrix[i] = calloc(number, sizeof(double));
  }

  for(int i = 0; i < number; i++) {
    if (!components[0][i]) matrix[1][i+1] = 1;
  }

  return matrix;
}



void dfss(double** strongMatrix, int vertex, int* visitedVertex) {
  visitedVertex[vertex] = 1;

  for (int i = 0; i < vertices; ++i) {
    if (strongMatrix[vertex][i] && !visitedVertex[i]) {
      dfss(strongMatrix, i, visitedVertex);
    }
  }
}

double **condensationMatrixWithCoef(double K) {
  double **matrix = directedMatrix(K);
```

```
  double **reachability = calculateReachabilityMatrix(matrix);
  double **connectivity = strongConnectivityMatrix(reachability);
  double **components = findStrongComponents(connectivity);

  freeMatrix(matrix, vertices);
  freeMatrix(reachability,vertices);
  freeMatrix(connectivity, vertices);
  return components;
}

int countStrongComponents(double** strongMatrix) {
  const int number = vertices;
  int* visitedVertex = calloc(number, sizeof(int));
  int count = 0;

  for (int i = 0; i < number; ++i) {
    if (!visitedVertex[i]) {
      dfss(strongMatrix, i,  visitedVertex);
      count++;
    }
  }

  free(visitedVertex);
  return count;
}
```

Файл *console.c* :

```
#include <stdio.h>
#include <stdlib.h>
#define vertices 12
#include "props.h"



void typeMatrix(double **matrix) {
  const int number = vertices;
  for (int i = 0; i < number; i++) {
    for (int j = 0; j < number; j++) {
      printf("%.0lf  ", matrix[i][j]);
    }
    printf("\n");
  }
}

void printDegrees(int *degrees) {
  const int number = vertices;
  printf("{  ");
  for (int i = 0; i < number; i++) {
    printf("%d  ", degrees[i]);
  }
  printf("}\n");
}

void printVertices(int* verticesNumber) {
  printf("{ ");
  for (int i = 0; verticesNumber[i] != 0; ++i) {
    printf("%d ", verticesNumber[i]);
  }
  printf("}\n");
}
```

```
void printPathways(double** matrix) {
  const int numbers = vertices;
  for (int i = 0; i < numbers; i++) {
    for (int j = 0; j < numbers; j++) {
      if (*(*(matrix + i) + j) != 0) {
        printf("%d -> %d;  ", i + 1, j + 1);
      }
    }
    printf("\n");
  }
}

void printComponents(double **matrix, int number) {
  int n = vertices;
  int componentCount = 1;
  for (int i = 0; i < n; i++) {
    int isNewComponent = 1;
    for (int j = 0; j < n; j++) {
      if(matrix[i][j]) {
        if (isNewComponent) printf("Component %d: [  ", componentCount);
        printf("%d  ", j + 1);
        isNewComponent = 0;
      }
    }
    if (!isNewComponent) {
      componentCount++;
      printf("]\n");
    }
  }
}


void aboutDirectedGraph() {

  double **T = randm(vertices);
  double coefficient = 1.0 - 0.01 - 0.01 - 0.3;
  double **A = mulmr(coefficient, T, vertices);

  int* entry = halfDegreeEntry(A);
  int* exit = halfDegreeExit(A);
  int* summedDegrees = summarizeHalfDegrees(exit, entry);

  int* isolated = findIsolatedVertices(summedDegrees);
  int* terminal = findTerminalVertices(summedDegrees);



  printf("\nThis is a Directed Graph \n");
  printf("\n\tInitial matrix\n");
  typeMatrix(A);


  printf("Exit degree : ");
  printDegrees(exit);

  printf("Entry degree : ");
  printDegrees(entry);

  printf("\nIs the graph homogeneous?\n");
  if(checkHomogeneity(summedDegrees)) {
    printf("%d\n", summedDegrees[0]);
  } else {
    printf("\tThe graph is not homogeneous ");
```

```
  }

  printf("\nFind isolated vertices\n");
  printf("Isolated vertices : ");
  if(isolated[0]) {
    printVertices(isolated);
  } else {
    printf("\tNo isolated vertices");
  }

  printf("\nFind terminal vertices\n");
  printf("Terminal vertices : ");
  if(terminal[0]) {
    printVertices(isolated);
  } else {
    printf("\tNo terminal vertices");
  }
  freeMatrix(A, vertices);

  free(entry);
  free(exit);

  free(summedDegrees);

  free(isolated);
  free(terminal);

}

void aboutUndirectedGraph() {
  double coefficient = 1.0 - 0.01 - 0.01 - 0.3;
  double **R = randm(vertices);
  double **C = symmetricMatrix(mulmr(coefficient, R, vertices), vertices);
  int* degree = graphDegrees(C);

  int* isolated = findIsolatedVertices(degree);
  int* terminal = findTerminalVertices(degree);

  printf("\nThis is a Undirected Graph \n");
  printf("\n\tMatrix for Undirected Graph\n");
  typeMatrix(C);

  printf("Undirected graph degrees : ");
  printDegrees(degree);

  printf("\nIs the graph homogeneous?\n");
  if(checkHomogeneity(degree)) {
    printf("%d\n", degree[0]);
  } else {
    printf("\tThe graph is not homogeneous ");
  }

  printf("\nFind isolated vertices\n");
  printf("Isolated vertices : ");
  if(isolated[0]) {
    printVertices(isolated);
  } else {
    printf("\tNo isolated vertices");
  }

  printf("\nFind terminal vertices\n");
  printf("Terminal vertices : ");
  if(terminal[0]) {
    printVertices(isolated);
```

```
  } else {
    printf("\tNo terminal vertices");
  }
  freeMatrix(C, vertices);

  free(degree);
  free(isolated);
  free(terminal);

}

void aboutModifiedGraph() {
  double** K = randm(vertices);
  double modifiedCoefficient = 1.0 - 0.005 - 0.005 - 0.27;
  double** D = mulmr(modifiedCoefficient, K, vertices);
  int* entry = halfDegreeEntry(D);
  int* exit = halfDegreeExit(D);
  int* summedDegrees = summarizeHalfDegrees(exit, entry);
  double** squaredMatrix = multiplyMatrices(D,D);
  double** cubedMatrix = multiplyMatrices(squaredMatrix, D);

  int* isolated = findIsolatedVertices(summedDegrees);
  int* terminal = findTerminalVertices(summedDegrees);

  double** reachabilityMatrix = calculateReachabilityMatrix(D);
  double** connectivityMatrix = strongConnectivityMatrix(reachabilityMatrix);

  double** strongComponents = findStrongComponents(connectivityMatrix);
  printf("\nThis is a Modified Graph \n");
  printf("\n\tMatrix for Modified Graph\n");
  typeMatrix(D);

  printf("Exit degree : ");
  printDegrees(exit);

  printf("Entry degree : ");
  printDegrees(entry);

  printf("\nIs the graph homogeneous?\n");
  if(checkHomogeneity(summedDegrees)) {
    printf("%d\n", summedDegrees[0]);
  } else {
    printf("\tThe graph is not homogeneous ");
  }

  printf("\nFind isolated vertices\n");
  printf("Isolated vertices : ");
  if(isolated[0]) {
    printVertices(isolated);
  } else {
    printf("\tNo isolated vertices");
  }

  printf("\nFind terminal vertices\n");
  printf("Terminal vertices : ");
  if(terminal[0]) {
    printVertices(isolated);
  } else {
    printf("\tNo terminal vertices");
  }

  printf("\nMatrix squared : 2\n");
  typeMatrix(squaredMatrix);
```

```
  printf("\nPathways with length : 2\n");
  printPathways(squaredMatrix);

  printf("\nMatrix cubed : 3\n");
  typeMatrix(cubedMatrix);

  printf("\nPathways with length : 3\n");
  printPathways(cubedMatrix);

  printf("\nReachability Matrix of Mod graph\n");
  typeMatrix(reachabilityMatrix);

  printf("\nConnected Matrix of Mod graph\n");
  typeMatrix(connectivityMatrix);

  printf("\nStrongly Connected Components of Mod Graph\n");
  printComponents(strongComponents,vertices);

  printf("\nMatrix of Condensation Graph\n");
  condensationMatrix(strongComponents);


  freeMatrix(D,vertices);
  freeMatrix(squaredMatrix, vertices);
  freeMatrix(cubedMatrix, vertices);
  freeMatrix(reachabilityMatrix, vertices);
  freeMatrix(connectivityMatrix, vertices);
  freeMatrix(strongComponents, vertices);


  free(isolated);
  free(terminal);
  free(summedDegrees);
  free(entry);
  free(exit);
}


int main() {

  aboutDirectedGraph();
  aboutModifiedGraph();
  aboutUndirectedGraph();

}
```

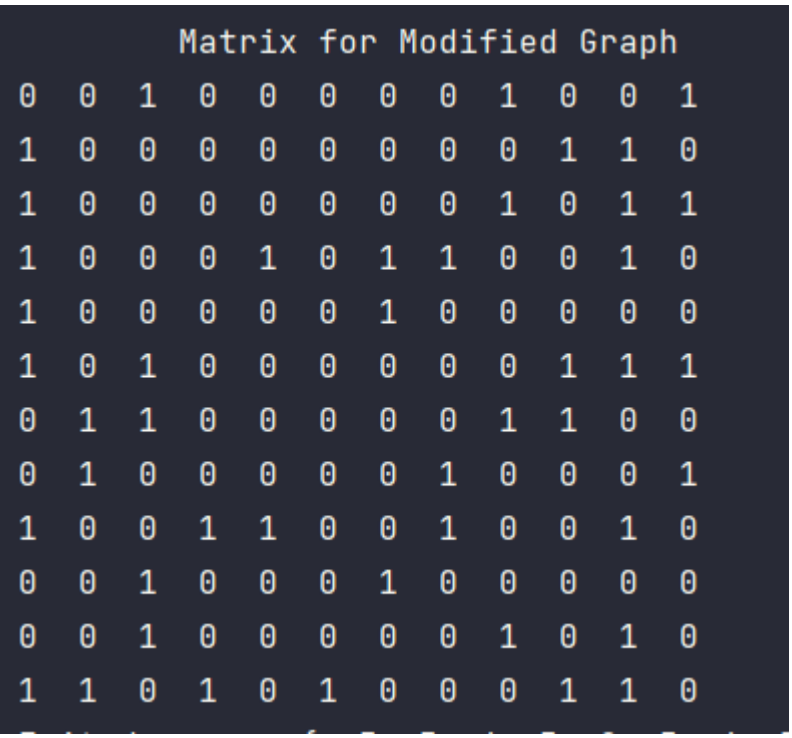
Згенеровані матриці суміжності для **орієнтованого** графа :

```
This is a Directed Graph

        Initial matrix
0  0  1  0  0  0  0  0  1  0  0  1
1  0  0  0  0  0  0  0  0  1  1  0
1  0  0  0  0  0  0  0  1  0  0  1
1  0  0  0  0  0  1  1  0  0  1  0
1  0  0  0  0  0  1  0  0  0  0  0
1  0  1  0  0  0  0  0  0  1  1  1
0  1  0  0  0  0  0  0  1  1  0  0
0  0  0  0  0  0  0  1  0  0  0  1
1  0  0  1  1  0  0  0  0  0  1  0
0  0  1  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  1  0  1  0
1  1  0  0  0  1  0  0  0  1  1  0
```

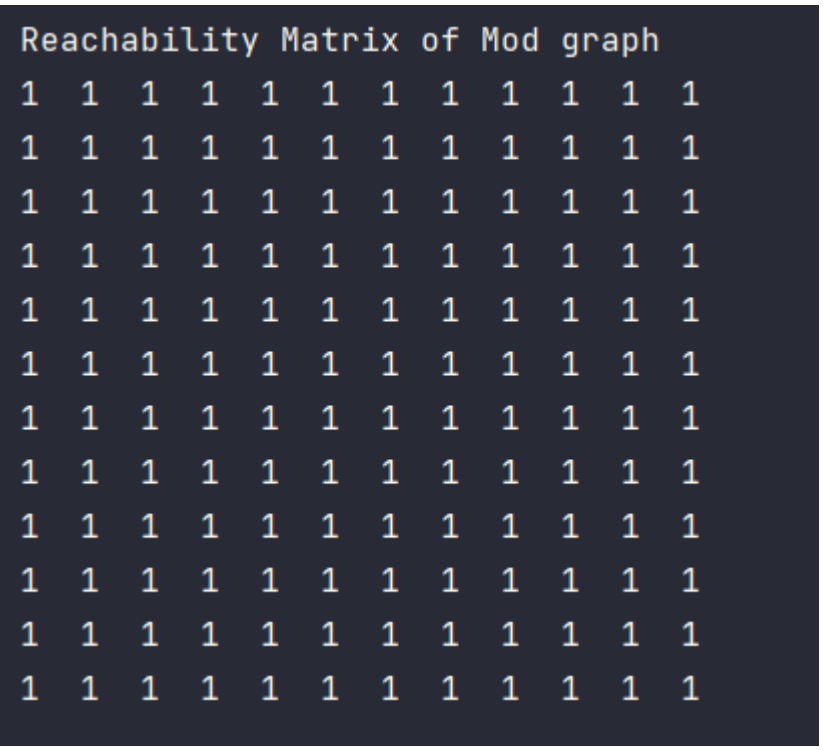
Згенеровані матриці суміжності для **неорієнтованого** графа :

```
This is a Undirected Graph

        Matrix for Undirected Graph
0  1  1  1  1  1  0  0  1  0  0  1
1  0  0  0  0  0  1  0  0  1  1  1
1  0  0  0  0  1  0  0  1  1  0  1
1  0  0  0  0  0  1  1  1  0  1  0
1  0  0  0  0  0  1  0  1  0  0  0
1  0  1  0  0  0  0  0  0  1  1  1
0  1  0  1  1  0  0  0  1  1  0  0
0  0  0  1  0  0  0  1  0  0  0  1
1  0  1  1  1  0  1  0  0  0  1  0
0  1  1  0  0  1  1  0  0  0  0  1
0  1  0  1  0  1  0  0  1  0  1  1
1  1  1  0  0  1  0  1  0  1  1  0
```

Згенеровані матриці суміжності для **модифікованого** графа :



Згенеровані матриці *досяжності* для **модифікованого** графа :



Згенеровані матриці *зв'язності* для **модифікованого** графа :

```
Connected Matrix of Mod graph
1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  1  1  1  1
```

Згенерована матриця *суміжності* для графа **конденсації** :

```
Matrix of Condensation Graph
0
```

Шлях довжиною 2 :

```
Pathways with length : 2
1 -> 1;  1 -> 2;  1 -> 4;  1 -> 5;  1 -> 6;  1 -> 8;  1 -> 9;  1 -> 10;  1 -> 11;  1 -> 12;
2 -> 3;  2 -> 7;  2 -> 9;  2 -> 11;  2 -> 12;
3 -> 1;  3 -> 2;  3 -> 3;  3 -> 4;  3 -> 5;  3 -> 6;  3 -> 8;  3 -> 9;  3 -> 10;  3 -> 11;  3 -> 12;
4 -> 1;  4 -> 2;  4 -> 3;  4 -> 7;  4 -> 8;  4 -> 9;  4 -> 10;  4 -> 11;  4 -> 12;
5 -> 2;  5 -> 3;  5 -> 9;  5 -> 10;  5 -> 12;
6 -> 1;  6 -> 2;  6 -> 3;  6 -> 4;  6 -> 6;  6 -> 7;  6 -> 9;  6 -> 10;  6 -> 11;  6 -> 12;
7 -> 1;  7 -> 3;  7 -> 4;  7 -> 5;  7 -> 7;  7 -> 8;  7 -> 9;  7 -> 10;  7 -> 11;  7 -> 12;
8 -> 1;  8 -> 2;  8 -> 4;  8 -> 6;  8 -> 8;  8 -> 10;  8 -> 11;  8 -> 12;
9 -> 1;  9 -> 2;  9 -> 3;  9 -> 5;  9 -> 7;  9 -> 8;  9 -> 9;  9 -> 11;  9 -> 12;
10 -> 1;  10 -> 2;  10 -> 3;  10 -> 9;  10 -> 10;  10 -> 11;  10 -> 12;
11 -> 1;  11 -> 3;  11 -> 4;  11 -> 5;  11 -> 8;  11 -> 9;  11 -> 11;  11 -> 12;
12 -> 1;  12 -> 3;  12 -> 5;  12 -> 7;  12 -> 8;  12 -> 9;  12 -> 10;  12 -> 11;  12 -> 12;
```

Шлях довжиною 3 :

```
Pathways with length : 3
1 -> 1;  1 -> 2;  1 -> 3;  1 -> 4;  1 -> 5;  1 -> 6;  1 -> 7;  1 -> 8;  1 -> 9;  1 -> 10;  1 -> 11;  1 -> 12;
2 -> 1;  2 -> 2;  2 -> 3;  2 -> 4;  2 -> 5;  2 -> 6;  2 -> 8;  2 -> 9;  2 -> 10;  2 -> 11;  2 -> 12;
3 -> 1;  3 -> 2;  3 -> 3;  3 -> 4;  3 -> 5;  3 -> 6;  3 -> 7;  3 -> 8;  3 -> 9;  3 -> 10;  3 -> 11;  3 -> 12;
4 -> 1;  4 -> 2;  4 -> 3;  4 -> 4;  4 -> 5;  4 -> 6;  4 -> 7;  4 -> 8;  4 -> 9;  4 -> 10;  4 -> 11;  4 -> 12;
5 -> 1;  5 -> 2;  5 -> 3;  5 -> 4;  5 -> 5;  5 -> 6;  5 -> 7;  5 -> 8;  5 -> 9;  5 -> 10;  5 -> 11;  5 -> 12;
6 -> 1;  6 -> 2;  6 -> 3;  6 -> 4;  6 -> 5;  6 -> 6;  6 -> 7;  6 -> 8;  6 -> 9;  6 -> 10;  6 -> 11;  6 -> 12;
7 -> 1;  7 -> 2;  7 -> 3;  7 -> 4;  7 -> 5;  7 -> 6;  7 -> 7;  7 -> 8;  7 -> 9;  7 -> 10;  7 -> 11;  7 -> 12;
8 -> 1;  8 -> 2;  8 -> 3;  8 -> 4;  8 -> 5;  8 -> 6;  8 -> 7;  8 -> 8;  8 -> 9;  8 -> 10;  8 -> 11;  8 -> 12;
9 -> 1;  9 -> 2;  9 -> 3;  9 -> 4;  9 -> 5;  9 -> 6;  9 -> 7;  9 -> 8;  9 -> 9;  9 -> 10;  9 -> 11;  9 -> 12;
10 -> 1;  10 -> 2;  10 -> 3;  10 -> 4;  10 -> 5;  10 -> 6;  10 -> 7;  10 -> 8;  10 -> 9;  10 -> 10;  10 -> 11;  10 -> 12
;
11 -> 1;  11 -> 2;  11 -> 3;  11 -> 4;  11 -> 5;  11 -> 6;  11 -> 7;  11 -> 8;  11 -> 9;  11 -> 10;  11 -> 11;  11 -> 12
;
12 -> 1;  12 -> 2;  12 -> 3;  12 -> 4;  12 -> 5;  12 -> 6;  12 -> 7;  12 -> 8;  12 -> 9;  12 -> 10;  12 -> 11;  12 -> 12
;
```

Степені усіх вершин ненапрямленого графу :

```
        Matrix for Undirected Graph
 0  1  1  1  1  1  0  0  1  0  0  1
 1  0  0  0  0  0  1  0  0  1  1  1
 1  0  0  0  0  1  0  0  1  1  0  1
 1  0  0  0  0  0  1  1  1  0  1  0
 1  0  0  0  0  0  1  0  1  0  0  0
 1  0  1  0  0  0  0  0  0  1  1  1
 0  1  0  1  1  0  0  0  1  1  0  0
 0  0  0  1  0  0  0  1  0  0  0  1
 1  0  1  1  1  0  1  0  0  0  1  0
 0  1  1  0  0  1  1  0  0  0  0  1
 0  1  0  1  0  1  0  0  1  0  1  1
 1  1  1  0  0  1  0  1  0  1  1  0
Undirected graph degrees : {  7  5  5  5  3  5  5  4  6  5  7  7  }
```

Напівстепені виходу та заходу напрямленого графу :

```
        Initial matrix
0  0  1  0  0  0  0  0  1  0  0  1
1  0  0  0  0  0  0  0  0  1  1  0
1  0  0  0  0  0  0  0  1  0  0  1
1  0  0  0  0  0  1  1  0  0  1  0
1  0  0  0  0  0  1  0  0  0  0  0
1  0  1  0  0  0  0  0  0  1  1  1
0  1  0  0  0  0  0  0  1  1  0  0
0  0  0  0  0  0  0  1  0  0  0  1
1  0  0  1  1  0  0  0  0  0  1  0
0  0  1  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  1  0  1  0
1  1  0  0  0  1  0  0  0  1  1  0
Exit degree : {  3  3  3  4  2  5  3  2  4  1  2  5  }
Entry degree : {  7  2  3  1  1  1  2  2  4  4  6  4  }
```

Напівстепені виходу та заходу модифікованого графу :

```
        Matrix for Modified Graph
 0  0  1  0  0  0  0  0  1  0  0  1
 1  0  0  0  0  0  0  0  0  1  1  0
 1  0  0  0  0  0  0  0  1  0  1  1
 1  0  0  0  1  0  1  1  0  0  1  0
 1  0  0  0  0  0  1  0  0  0  0  0
 1  0  1  0  0  0  0  0  0  1  1  1
 0  1  1  0  0  0  0  0  1  1  0  0
 0  1  0  0  0  0  0  1  0  0  0  1
 1  0  0  1  1  0  0  1  0  0  1  0
 0  0  1  0  0  0  1  0  0  0  0  0
 0  0  1  0  0  0  0  0  1  0  1  0
 1  1  0  1  0  1  0  0  0  1  1  0
 Exit degree : {  3  3  4  5  2  5  4  3  5  2  3  6  }
 Entry degree : {  7  3  5  2  2  1  3  3  4  4  7  4  }
```

```
Matrix squared : 2
3  1  0  2  1  1  0  1  1  1  3  1
0  0  3  0  0  0  1  0  2  0  1  1
2  1  2  2  1  1  0  1  2  1  3  1
1  2  3  0  0  0  1  1  3  1  1  2
0  1  2  0  0  0  0  0  2  1  0  1
2  1  3  1  0  1  1  0  3  1  3  2
3  0  1  1  1  0  1  1  1  1  3  1
2  2  0  1  0  1  0  1  0  2  2  1
2  1  2  0  1  0  2  2  2  0  2  2
1  1  1  0  0  0  0  0  2  1  1  1
2  0  1  1  1  0  0  1  2  0  3  1
3  0  4  0  1  0  2  1  2  2  4  2
```

Другий степінь модифікованої матриці суміжності :

Третій степінь модифікованої матриці суміжності :

```
Matrix cubed : 3
7  2  8  2  3  1  4  4  6  3  9  5
6  2  2  3  2  1  0  2  5  2  7  3
10  2  7  3  4  1  4  5  7  3  12  6
10  4  4  5  3  2  1  4  6  5  11  5
6  1  1  3  2  1  1  2  2  2  6  2
11  3  8  5  4  2  2  4  9  5  14  6
5  3  8  2  2  1  3  3  8  2  7  5
5  2  7  1  1  1  3  2  4  4  7  4
8  6  6  4  2  2  1  4  8  5  9  6
5  1  3  3  2  1  1  2  3  2  6  2
6  2  5  3  3  1  2  4  6  1  8  4
9  5  11  4  2  2  3  3  13  4  12  8
```

Перевірка на однорідність; чи є висячі та ізольовані вершини :

Напрямлений граф :

```
Is the graph homogeneous?
        The graph is not homogeneous
Find isolated vertices
Isolated vertices :     No isolated vertices
Find terminal vertices
Terminal vertices :     No terminal vertices
```

Ненапрямлений граф :

```
Is the graph homogeneous?
        The graph is not homogeneous
Find isolated vertices
Isolated vertices :     No isolated vertices
Find terminal vertices
Terminal vertices :     No terminal vertices
```

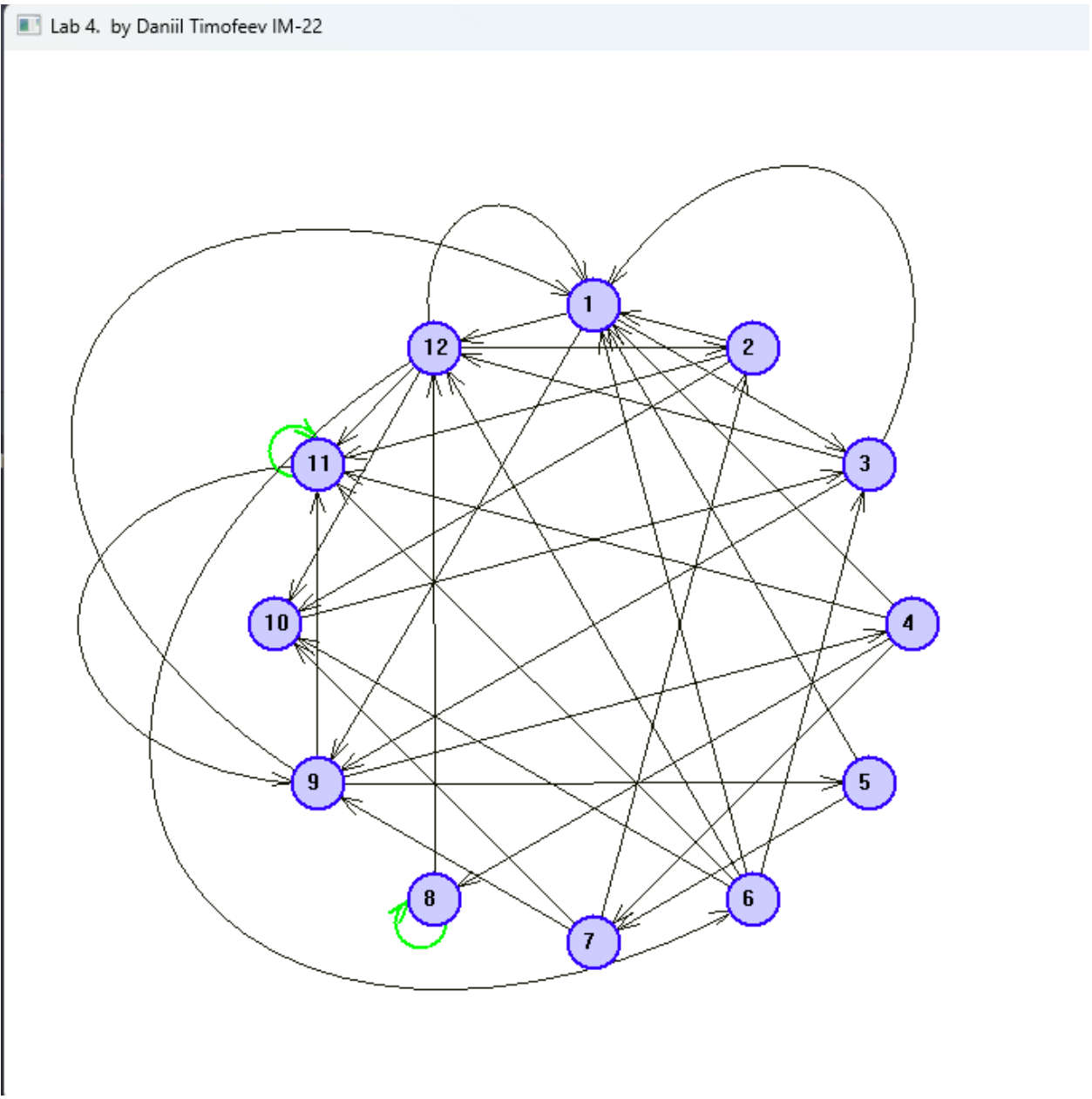
Модифікований граф :

```
Is the graph homogeneous?
        The graph is not homogeneous
Find isolated vertices
Isolated vertices :     No isolated vertices
Find terminal vertices
Terminal vertices :     No terminal vertices
```

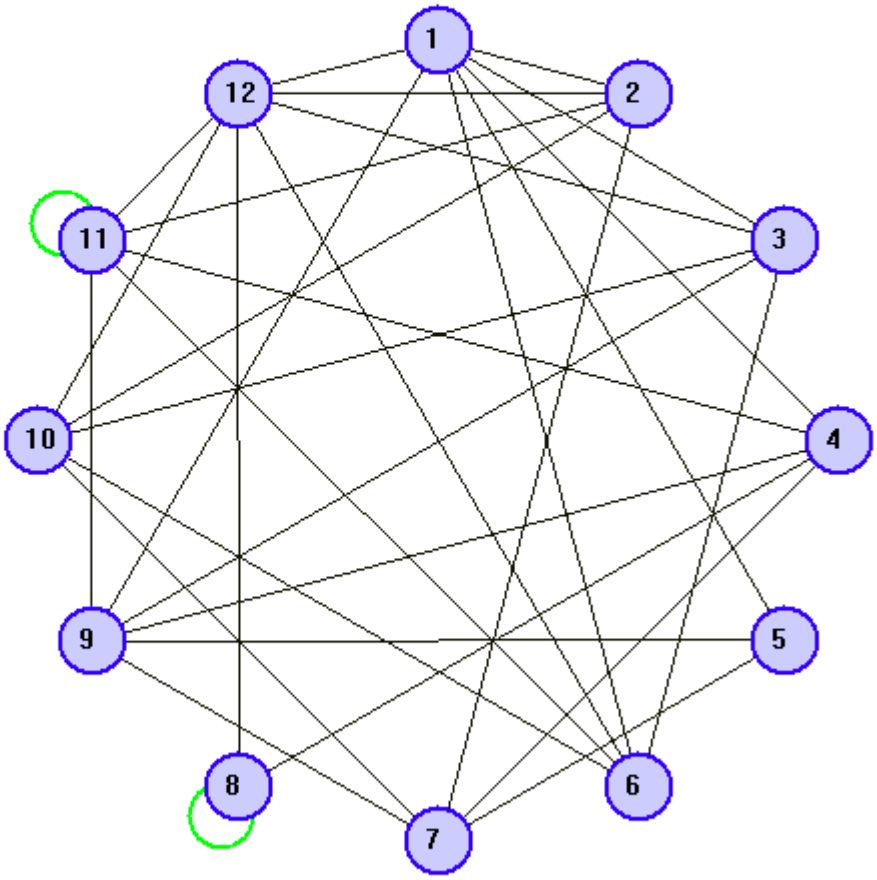
Компоненти сильної зв’язності :

```
Strongly Connected Components of Mod Graph
Component 1: [  1  2  3  4  5  6  7  8  9  10  11  12  ]
```

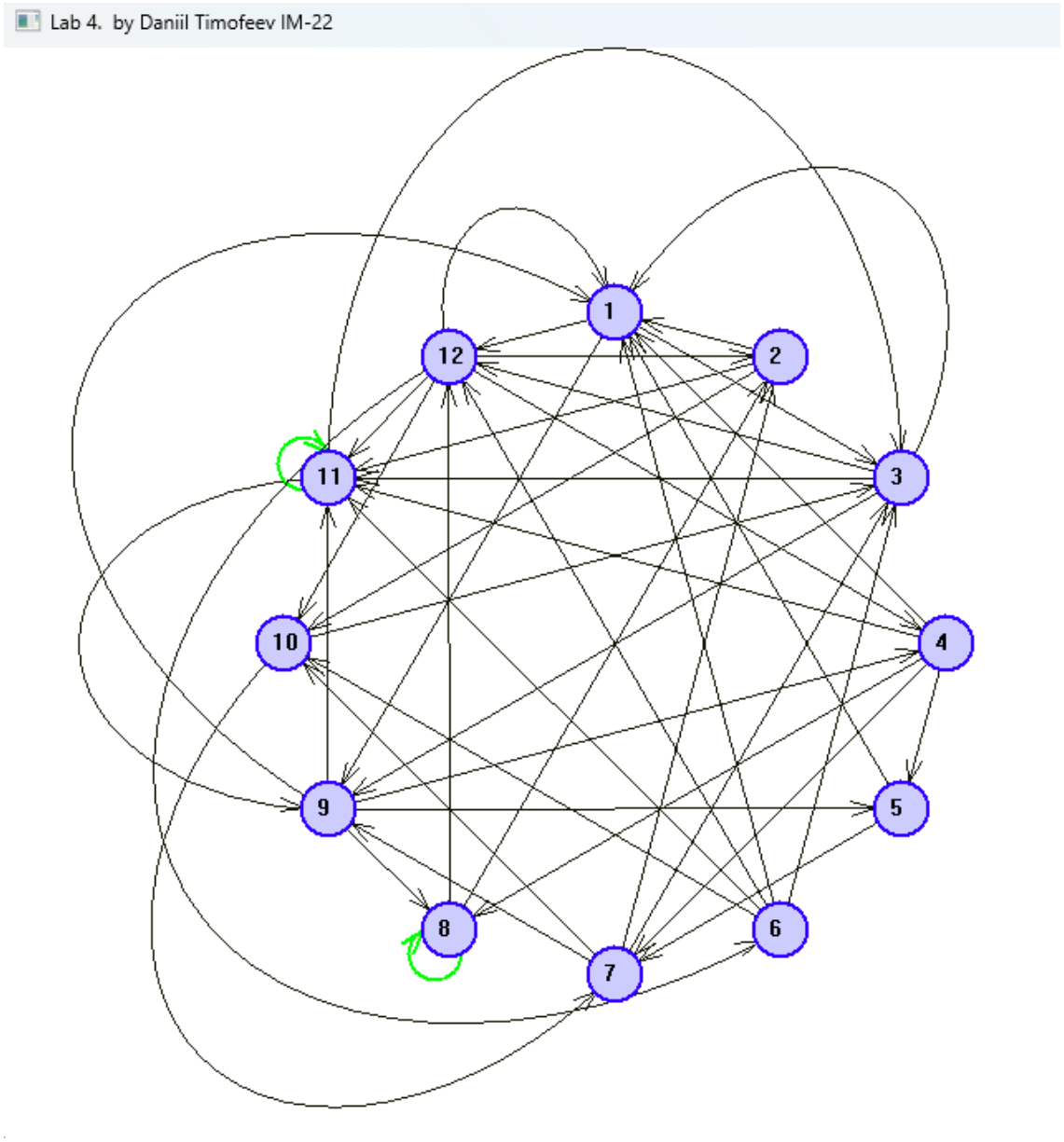
Орієнтований граф:



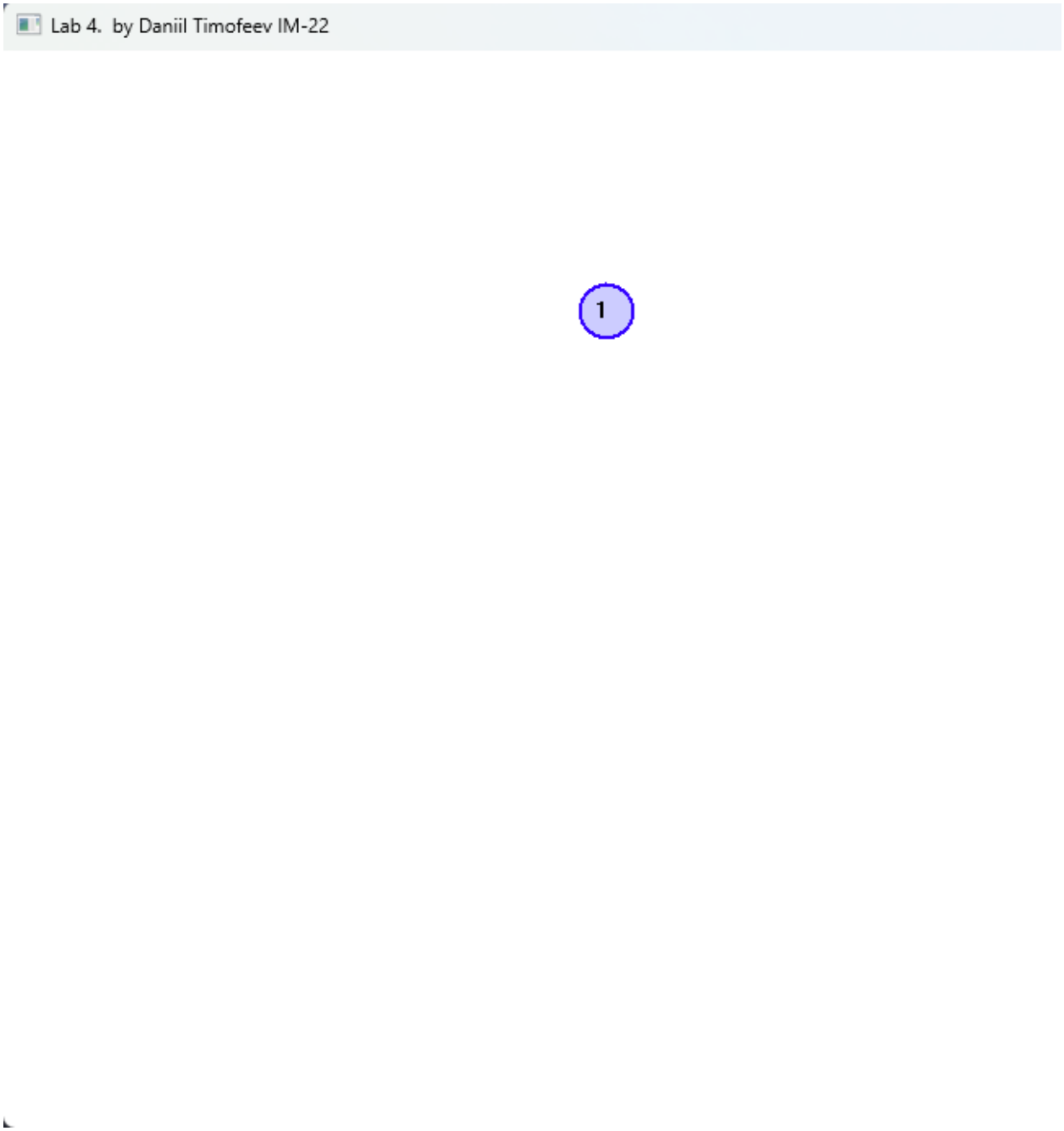
Неорієнтований граф:



Модифікований граф:



Граф конденсації:



Краще запускати так : gcc .\components.c .\console.c -o console.exe – для консольного виводу; gcc .\components.c .\main.c -mwindows -o picture.exe – для виводу графів

Висновки :

Завдяки лабораторній роботі №4 я вдосконалив навичку розбиття коду на окремі функції. Також ознайомився з алгоритмами знаходження : матриць досяжності, зв’язності суміжності графа конденсації; висячих та ізольованих вершин. Наприклад, для знаходження матриці досяжності, замість об’єднання степенів матриць суміжності виконав їх додавання. Тоді слід виконати корекцію одержаної матриці за допомогою булевого перетворення

згідно з теоремою. Виникли труднощі зі знаходженням компонент сильної зв’язності.

Ба більше, було використано алгоритм обходу графа (пошук у глиб) для знаходження компонент сильної зв’язності. Звичайно, можна було використати й інші алгоритми, такий як алгоритм Косараю. Але я обрав саме пошук у глиб, бо це достатньо простий алгоритм, який легко реалізувати. Завдяки простій рекурсивній структурі пошук у глиб є досить ефективним для пошуку компонент сильної зв’язності, адже в середньому його складність становить O(V + E), де V - кількість вершин, а E - кількість ребер у графі. Також пошук у глибину працює на принципі глибинного перебору, що дозволяє виявляти компоненти сильної зв'язності, які мають глибше розташування у графі. Це також дало ознайомлення з важливими алгоритмами з графами.

Покращив контроль за динамічною пам’яттю : правильно виділяти(за допомогою функцій malloc/calloc) та вивільняти (за допомогою функцій free та своєї функції, така як freeMatrix).

У цій лабораторній було створену структуру coordinates для збереження координат для кожної вершини графа.

Більш того, при написанні лабораторної роботи були декілька варіантів для реалізації деяких алгоритмів. Наприклад, для знаходження матриці зв’язності. Спочатку я думав реалізувати це так : розглядати кожну вершину в сильно зв'язних компонентах і перевіряти з'єднання цих вершин з вершинами в інших компонентах. На основі цих з'єднань заповнювати матрицю зв'язності. Але зупинився на використанні переданих сильно зв'язні компоненти безпосередньо для формування матриці зв'язності. Я використовую інформацію про наявність зв'язків між вершинами в компонентах, щоб встановити значення 1 відповідних елементів матриці зв'язності. Вибрав останній варіант, бо він більш ефективний оскільки я вже маю цю інформацію. Просто використовую її для створення матриці зв'язності, що може бути швидше.