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 № 2.5

з дисципліни
«Алгоритми і структури даних»

Виконав:

студент групи ІМ-22
Коваленко Михайло Володимирович
номер у списку групи: 11

Перевірила:

Молчанова А. А.

Київ 2023

Лабораторна робота 5.

Обхід графа

Мета лабораторної роботи

Метою лабораторної роботи No5. «Обхід графа» є вивчення методу дослідження графа за допомогою обходу його вершин в глибину або в ширину.

Постановка задачі

1. Представити напрямлений граф з заданими параметрами так само, як у лабораторній роботі No3. Відміна: матриця A за варіантом формується за функцією:

A = mulmr((1.0 – n3*0.01 – n4*0.005 − 0.15)*T);

2. Створити програми для обходу в глибину та в ширину. Обхід починати з вершини, яка має вихідні дуги. При цьому у програмі:

— встановити зупинку у точці призначення номеру черговій вершині за допомогою повідомлення про натискання кнопки,

— виводити зображення графа у графічному вікні перед кожною зупинкою.

3. Під час обходу графа побудувати дерево обходу. Вивести побудоване дерево у графічному вікні.

Завдання для варіанту:

Число вершин: 11

Розміщення колом

srand(2211);

A = mulmr((1.0 – 1*0.01 – 1*0.005 − 0.15)*T);

Текст програми:

main.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <math.h>
#include "props.h"
#define verts_amount 11
#define IDC_BUTTON1 1
#define IDC_BUTTON2 2

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

char ProgName[] = "Lab 5 Mykhailo Kovalenko";

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

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 printMatrix(double **matrix, int n, int initialX, int initialY,
HDC hdc) {
  for (int i = 0, y = initialY + 30; i < n; i++, y += 15) {
    for (int j = 0, x = initialX; j < n; j++, x += 13) {
      wchar_t buffer[2];
      swprintf(buffer, 2, L"%lf", matrix[i][j]);
      TextOut(hdc, x, y, (LPCSTR) buffer, 1);
    }
    MoveToEx(hdc, initialX, y, NULL);
  }
}

void drawArch(int startX, int startY, int finishX, int finishY, int
archInterval, HDC hdc) {
  XFORM transformedMatrix;
  XFORM initialMatrix;
  GetWorldTransform(hdc, &initialMatrix);
```

```
  double angle = atan2(finishY - startY, finishX - startX) - M_PI /
2.0;
    transformedMatrix.eM11 = (FLOAT) cos(angle);
    transformedMatrix.eM12 = (FLOAT) sin(angle);
    transformedMatrix.eM21 = (FLOAT) (-sin(angle));
    transformedMatrix.eM22 = (FLOAT) cos(angle);
    transformedMatrix.eDx = (FLOAT) startX;
    transformedMatrix.eDy = (FLOAT) startY;
  SetWorldTransform(hdc, &transformedMatrix);

    const double archWidth = 0.75;
    double length = sqrt((finishX - startX) * (finishX - startX) +
(finishY - startY) * (finishY - startY));
    double radiusOfVertex = 15.0;
    double semiMinorAxis = archWidth * length;
    double semiMajorAxis = length / 2;
    double vertexAreaSquared = semiMajorAxis * semiMajorAxis *
radiusOfVertex * radiusOfVertex;
    double semiAxesSquared = semiMinorAxis * semiMinorAxis *
semiMajorAxis * semiMajorAxis;
    double ellipseStartY = semiMajorAxis;
    double radius = semiMinorAxis * semiMinorAxis * ellipseStartY *
ellipseStartY;
    double distance = semiMinorAxis * semiMinorAxis * radiusOfVertex
* radiusOfVertex;
    double semiMinorAxisPow = pow(semiMinorAxis, 4);
    double crossing = semiMajorAxis * sqrt(vertexAreaSquared -
semiAxesSquared + radius - distance + semiMinorAxisPow);
    double semiMinorAxisSquaredEllipseStartY = semiMinorAxis *
semiMinorAxis * ellipseStartY;
    double denominator = -semiMajorAxis * semiMajorAxis +
semiMinorAxis * semiMinorAxis;
    double contactYRightTop = (semiMinorAxisSquaredEllipseStartY -
crossing) / denominator;
    double contactXRightTop = sqrt(radiusOfVertex * radiusOfVertex -
contactYRightTop * contactYRightTop);
    double contactYBottom = length - contactYRightTop;
    double contactXLeftBottom = -contactXRightTop;

    if (archInterval <= verts_amount / 2) {
        Arc(hdc, -archWidth * length, length, archWidth * length, 0,
0, 0, 0, length);
        double angleOfArrow = -atan2(length - contactYBottom,
contactXLeftBottom) + 0.3 / 3;
        arrow(angleOfArrow, contactXLeftBottom, contactYBottom, hdc);
    } else {
        Arc(hdc, -archWidth * length, length, archWidth * length, 0,
0, length, 0, 0);
        double angleOfArrow = -atan2(length - contactYBottom, -
```

```
contactXLeftBottom) - 0.3 / 3;
        arrow(angleOfArrow, -contactXLeftBottom, contactYBottom,
hdc);
    }
    SetWorldTransform(hdc, &initialMatrix);

}

void drawDirectedGraph(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 < verts_amount; i++) {
    for (int j = 0; j < verts_amount; 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 triangleHeight = sqrt(3) * radiusOfVertex / 2.;
            double radiusOfContact = radiusOfGraph + radiusOfLoop /
2.;
            double distance = sqrt(radiusOfContact * radiusOfContact
+ triangleHeight * triangleHeight);
            double angleToContactVertex = atan2(coordinates.ny[i] -
centerY, coordinates.nx[i] - centerX);
            double loopAngle = atan2(triangleHeight,
radiusOfContact);
            double contactCoordX = centerX + distance *
cos(angleToContactVertex + loopAngle);
            double contactCoordY = centerY + distance *
sin(angleToContactVertex + loopAngle);
            double curveAngle = angleToContactVertex + 0.3 / 2.;
            arrow(curveAngle, contactCoordX, contactCoordY, 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) {
          drawArch(coordinates.nx[i], coordinates.ny[i],
coordinates.nx[j], coordinates.ny[j], fabs(i - j), hdc);
      }

    }
  }
}

int dfsIteration = 0;
int bfsIteration = 0;

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) "Lab5 (Mykhailo Kovalenko IM-22)",
                      WS_OVERLAPPEDWINDOW,
                      100,
                      100,
                      1150,
                      800,
                      (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 DFSButton;
  HWND BFSButton;
  const int amountOfVertices = verts_amount;
  int state = 0;

  double **T = randm(verts_amount);
  double coefficient = 1.0 - 0.01 - 0.005 - 0.15;
  double **A = mulmr(coefficient, T, verts_amount);
  int* queue = malloc(verts_amount * sizeof(int));
  int* depth = malloc(verts_amount * sizeof(int));
  int* visited = malloc(verts_amount * sizeof(int));
  int birthVertex = findFirst(A, verts_amount);
  double** treeDFS = createMatrix(verts_amount);
  double** treeBFS = createMatrix(verts_amount);
    fillZero(treeDFS, verts_amount);
    fillZero(treeBFS, verts_amount);

  runDfsForNotVisitedVertices(birthVertex, A, visited, 0, depth,
treeDFS);
  breadthFirstSearch(A, birthVertex, queue, treeBFS);

  switch (messg) {
    case WM_CREATE: {
        DFSButton = CreateWindow(
              (LPCSTR) "BUTTON",
              (LPCSTR) "Step into DFS",
              WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,
              700,
              30,
              160,
              50,
              hWnd,
              (HMENU) IDC_BUTTON1,
              (HINSTANCE) GetWindowLongPtr(hWnd, GWLP_HINSTANCE),
              NULL);
        BFSButton = CreateWindow(
              (LPCSTR) "BUTTON",
              (LPCSTR) "Step into BFS",
              WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,
              900,
              30,
              160,
              50,
              hWnd,
```

```
              (HMENU) IDC_BUTTON2,
              (HINSTANCE) GetWindowLongPtr(hWnd, GWLP_HINSTANCE),
              NULL);
      return 0;
    }
    case WM_COMMAND: {
      switch (LOWORD(wParam)) {

        case IDC_BUTTON1:
          state = 0;
          if (dfsIteration < amountOfVertices) dfsIteration++;
          InvalidateRect(hWnd, NULL, FALSE);
          break;

        case IDC_BUTTON2:
          state = 1;
          if (bfsIteration < amountOfVertices) bfsIteration++;
          InvalidateRect(hWnd, NULL, FALSE);
          break;
      }
    }
    case WM_PAINT :
      hdc = BeginPaint(hWnd, &ps);
          HFONT hFont = CreateFont(16, 0, 0, 0, FW_NORMAL, FALSE,
FALSE, FALSE,
                                   DEFAULT_CHARSET,
OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,
                                   DEFAULT_QUALITY, DEFAULT_PITCH |
FF_DONTCARE, L"Arial");
          SelectObject(hdc, hFont);
          DeleteObject(hFont);
          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 GPen2 = CreatePen(PS_SOLID, 2, RGB(0, 153, 76));
      HPEN CPen = CreatePen(PS_SOLID, 2, RGB(0,206,209));
      HPEN NoPen = CreatePen(PS_NULL, 0, RGB(0, 0, 0));
      SelectObject(hdc, NoPen);
      Rectangle(hdc, 0, 0, 670, 700);


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

      struct coordinates coordinates;

      double circleRadius = 200;
      double vertexRadius = circleRadius / 11;
```

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

      double circleCenterX = 370;
      double circleCenterY = 360;


      double angleAlpha = 2.0 * M_PI / (double) verts_amount;
      for (int i = 0; i < verts_amount; 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;
      }

      int defaultMatrixX = 700;
      int defaultMatrixY = 100;
      double** dfsDetour = createCrossingMatrix(depth);
      double** bfsDetour = createCrossingMatrix(queue);
      TextOut(hdc, defaultMatrixX, defaultMatrixY, (LPCSTR) L"Initial
Matrix", 28);
      printMatrix(A, verts_amount, defaultMatrixX, defaultMatrixY,
hdc);

      TextOut(hdc, defaultMatrixX, defaultMatrixY + 220, (LPCSTR)
L"DFS Relativity", 28);
      printMatrix(dfsDetour, verts_amount, defaultMatrixX,
defaultMatrixY + 220, hdc);
      TextOut(hdc, defaultMatrixX + 200, defaultMatrixY + 220,
(LPCSTR) L"DFS Tree", 15);
      printMatrix(treeDFS, verts_amount, defaultMatrixX + 200,
defaultMatrixY + 220, hdc);

      TextOut(hdc, defaultMatrixX, defaultMatrixY + 440, (LPCSTR)
L"BFS Relativity", 28);
      printMatrix(bfsDetour, verts_amount, defaultMatrixX,
defaultMatrixY + 440, hdc);
      TextOut(hdc, defaultMatrixX + 200, defaultMatrixY + 440,
(LPCSTR) L"BFS Tree", 15);
      printMatrix(treeBFS, verts_amount, defaultMatrixX + 200,
defaultMatrixY + 440, hdc);

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

```
      SelectObject(hdc, KPen);
          drawDirectedGraph(circleCenterX, circleCenterY,
circleRadius, vertexRadius, loopRadius, angleAlpha,
                            coordinates, A, KPen, GPen, hdc);

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

        for (int i = 0; i < verts_amount; ++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);
        }

        SelectObject(hdc, GPen2);
        SetDCBrushColor(hdc, RGB(0, 153, 76));

      if (state == 0) {
        double** modified = createMatrix(amountOfVertices);
          fillZero(modified, amountOfVertices);
        for (int i = 0; i < dfsIteration; ++i) {
            buildSearchMatrix(treeDFS, depth[i], modified);
            drawDirectedGraph(circleCenterX, circleCenterY,
circleRadius, vertexRadius, loopRadius, angleAlpha,
                              coordinates, modified, GPen2, GPen,
hdc);
          Ellipse(hdc, coordinates.nx[depth[i]] - vertexRadius,
coordinates.ny[depth[i]] - vertexRadius,
                  coordinates.nx[depth[i]] + vertexRadius,
coordinates.ny[depth[i]] + vertexRadius);
        }

        for (int i = 0; i < dfsIteration; i++)
        {
          wchar_t buffer[5];
          swprintf(buffer, 5, L"%d", depth[i] + 1);
          TextOut(hdc, coordinates.nx[depth[i]] - dtx,
coordinates.ny[depth[i]] - vertexRadius / 2, buffer, 3);
        }
        freeMatrix(modified, amountOfVertices);
      }

      SelectObject(hdc, CPen);
      SetDCBrushColor(hdc, RGB(0,206,209));
```

```
      if (state == 1) {
        double** modified = createMatrix(amountOfVertices);
          fillZero(modified, amountOfVertices);

        for (int i = 0; i < bfsIteration; ++i) {
            buildSearchMatrix(treeBFS, queue[i], modified);
            drawDirectedGraph(circleCenterX, circleCenterY,
circleRadius, vertexRadius, loopRadius, angleAlpha,
                              coordinates, modified, CPen, GPen,
hdc);

          Ellipse(hdc, coordinates.nx[queue[i]] - vertexRadius,
coordinates.ny[queue[i]] - vertexRadius,
                  coordinates.nx[queue[i]] + vertexRadius,
coordinates.ny[queue[i]] + vertexRadius);
            //printf("%d" , queue[i]);
        }
          //printf(" ");

        for (int i = 0; i < bfsIteration; i++)
        {
          wchar_t buffer[5];
          swprintf(buffer, 5, L"%d", queue[i] + 1);
          TextOut(hdc, coordinates.nx[queue[i]] - dtx,
coordinates.ny[queue[i]] - vertexRadius / 2, buffer, 3);
        }
        freeMatrix(modified, amountOfVertices);
      }

        EndPaint(hWnd, &ps);

        freeMatrix(A, verts_amount);
        free(queue);
        free(depth);
        free(visited);
        freeMatrix(dfsDetour, verts_amount);
        freeMatrix(bfsDetour, verts_amount);
        freeMatrix(treeDFS, verts_amount);
        freeMatrix(treeBFS, verts_amount);

        case WM_DESTROY:
          PostQuitMessage(0);
        break;
        default:
          return (DefWindowProc(hWnd, messg, wParam, lParam));
  }
  return 0;
}
```

functions.c:

```c
#include <stdlib.h>
#define verts_amount 11

double **randm(int n) {
  srand(2211);
  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;
}

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

double** createMatrix(int n) {
  double **matrix = (double **) malloc(n * sizeof(double *));
  for (int i = 0; i < n; i++) {
    matrix[i] = (double *) malloc(n * sizeof(double));
  }
  return matrix;
}

void freeMatrix(double **matrix, int n) {
  for (int i = 0; i < n; ++i) {
```

```
    free(matrix[i]);
  }
  free(matrix);
}

double** createCrossingMatrix(const int* array) {
    double** traversalMatrix = createMatrix(verts_amount);
    fillZero(traversalMatrix, verts_amount);
    for (int i = 0; i < verts_amount; i++)
    {
        traversalMatrix[array[i]][i] = 1.0;
    }
    return traversalMatrix;
}

void buildSearchMatrix(double** graph, int sourceVertex, double**
searchMatrix) {
    const int number = verts_amount;
    for (int i = 0; i < number; ++i) {
        if (graph[i][sourceVertex] == 1)
searchMatrix[i][sourceVertex] = 1;
    }
}

int findFirst(double** matrix, int n) {
  int* outgoingCounts = (int*)calloc(n, sizeof(int));
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      if (matrix[i][j] == 1) {
        outgoingCounts[i]++;
      }
    }
  }
  for (int i = 0; i < n; ++i) {
    if (outgoingCounts[i] > 0) {
      free(outgoingCounts);
      return i;
    }
  }
  free(outgoingCounts);
  return -1;
}

void breadthFirstSearch(double** adjacencyMatrix, int startVertex,
int* queue, double** tree) {
    const int number = verts_amount;
    int visited[number];
    for (int i = 0; i < number; i++)
    {
```

```
        visited[i] = 0;
    }
    int queueStart = 0;
    int queueFinish = 0;

    visited[startVertex] = 1;
    queue[queueFinish] = startVertex;

    while (queueStart <= queueFinish) {
        int currentVertex = queue[queueStart++];
        for (int neighborVertex = 0; neighborVertex < number;
neighborVertex++) {
            if (adjacencyMatrix[currentVertex][neighborVertex] == 1
&& visited[neighborVertex] == 0)
            {
                tree[currentVertex][neighborVertex] = 1;
                queue[++queueFinish] = neighborVertex;
                visited[neighborVertex] = 1;
            }
        }
    }
    for (int i = 0; i < number; ++i) {
        if (visited[i] == 0){
            int currentVertex = i;
            for (int neighborVertex = 0; neighborVertex < number;
neighborVertex++) {
                if (adjacencyMatrix[currentVertex][neighborVertex] ==
1 && visited[neighborVertex] == 0 && currentVertex != neighborVertex)
                {
                    queue[++queueFinish] = currentVertex;
                    tree[currentVertex][neighborVertex] = 1;
                    queue[++queueFinish] = neighborVertex;
                    visited[neighborVertex] = 1;
                }
            }
        }
    }
}

void depthFirstSearch(double** adjacencyMatrix, int currentVertex,
int* visited, int* depthVertices, double** tree, int* numVisited) {
    const int number = verts_amount;
    visited[currentVertex] = 1;
    depthVertices[(*numVisited)] = currentVertex;
    (*numVisited)++;
    for (int neighborVertex = 0; neighborVertex < number;
++neighborVertex) {
        if (adjacencyMatrix[currentVertex][neighborVertex] == 1 &&
visited[neighborVertex] == 0) {
```

```
            tree[currentVertex][neighborVertex] = 1;
            depthFirstSearch(adjacencyMatrix, neighborVertex,
visited, depthVertices, tree, numVisited);
        }
    }
}

void runDfsForNotVisitedVertices(int currentVertex, double**
adjacencyMatrix,
                                 int* visited, int amount, int*
depthVertices, double** graph ) {
  const int number = verts_amount;
  for (int i = 0; i < number; ++i) {
    visited[i] = 0;
  }
  for (int i = 0; i < number; ++i) {
    if (visited[i] == 0) {
      depthFirstSearch(adjacencyMatrix, i, visited,
depthVertices,graph, &amount);
    }
  }
}
```

props.h:

```
#ifndef LAB_2_5_PROPS_H
#define LAB_2_5_PROPS_H
double **randm(int n);
double **mulmr(double coef, double **matrix, int n);
void fillZero(double** matrix, int n);
double** createMatrix(int n);
void freeMatrix(double **matrix, int n);
double** createCrossingMatrix(const int* array);
void buildSearchMatrix(double** graph, int sourceVertex, double**
searchMatrix);
int findFirst(double** matrix, int n);
void breadthFirstSearch(double** adjacencyMatrix, int startVertex,
int* queue, double** tree);
void depthFirstSearch(double** adjacencyMatrix, int currentVertex,
int* visited, int* depthVertices, double** tree, int* numVisited);
void runDfsForNotVisitedVertices(int currentVertex, double**
adjacencyMatrix, int* visited, int amount, int* depthVertices,
double** graph );
#endif
```

Згенерована матриця суміжності:

Initial Matrix

```
0 0 0 1 0 0 0 0 0 0 0
1 1 0 0 0 1 0 1 1 1 0
1 0 1 0 1 1 0 0 0 0 0
1 1 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0 0
1 1 0 0 0 0 0 0 0 0 0
0 1 0 1 0 1 0 0 1 1 0
0 0 0 1 0 0 0 1 0 0 1
1 0 0 0 0 0 1 0 0 1 0
0 0 0 1 0 1 1 0 0 1 0
```

Матриця дерева обходу і матриця відповідності вершин і одержаної нумерації:

DFS Relativity

```
1 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0
```

DFS Tree

```
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 1 0 0 1
0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
```

BFS Relativity

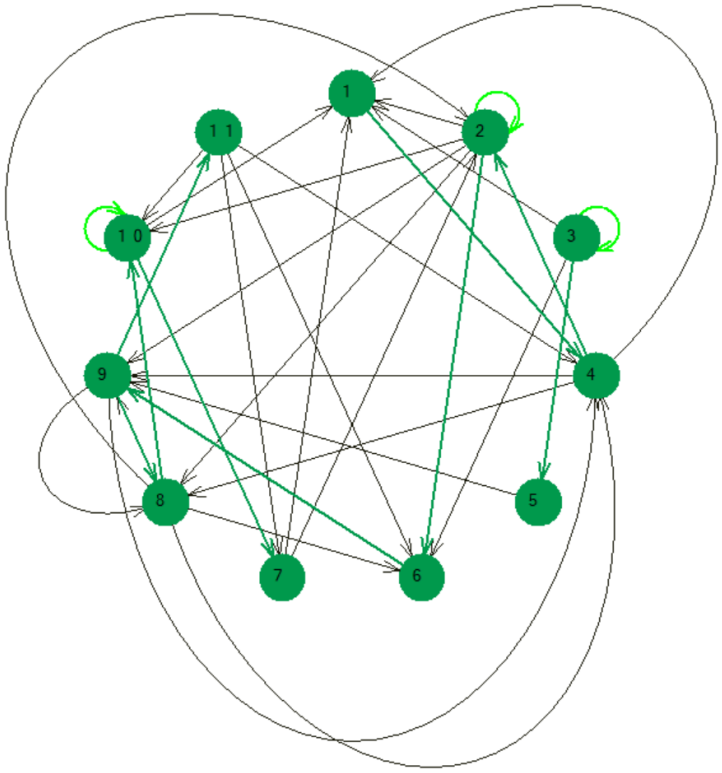
```
1 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0
```

BFS Tree

```
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 1 0
0 0 0 0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
```

Скріншоти зображення графа з одержаною нумерацією та дерева обходу:

DFS:



BFS: