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-42
Тєпайкін
номер у списку групи: 27

Перевірила:

Молчанова А. А.

Київ 2025

Постановка задачі

1. Представити напрямлений граф з заданими параметрами так само, як у лабораторній роботі №3.

Відміна: матриця A 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 A^2 і A^3, відповідно. Матриця досяжності та компоненти сильної зв'язності слід шукати за допомогою операції транзитивного замикання.

За варіантом: n1 = 4, n2 = 2, n3 = 2, n4 = 7;

Текст програми

```
#include <stdio.h>
#include <windows.h>
#include <math.h>

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

const char ProgName[] = "Lab 4";
const int n3 = 2;
const int n4 = 7;
const int vert = 11;
int variant;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow) {
  printf("Enter the graph you need to draw: 1 for directed graph, 2 for undirected, 3 for modified\n");
  scanf("%i", &variant);
  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,
        "",
        WS_OVERLAPPEDWINDOW,
        100,
        100,
        1000,
        700,
        (HWND) NULL,
        (HMENU) NULL,
        (HINSTANCE) hInstance,
        (HINSTANCE) NULL
);
ShowWindow(hWnd, nCmdShow);
int b;
while((b = GetMessage(&lpMsg, hWnd, 0, 0))!= 0) {
  if(b == -1)  {
    return lpMsg.wParam;
  }
  else {
    TranslateMessage(&lpMsg);
    DispatchMessage(&lpMsg);
  }
}
return (lpMsg.wParam);
```

```
}

HDC hdc;
PAINTSTRUCT ps;

void arrow(HDC hdc, int x1, int y1, int x2, int y2, int radius) {
  double dx = x2 - x1;
  double dy = y2 - y1;

  int arrowLength = 10;
  double arrowAngle = 36.0 * 3.14 / 180.0;
  double angle = atan2(dy, dx);
  double xIntersec = x2 - radius * cos(angle);
  double yIntersec = y2 - radius * sin(angle);
  double x3 = xIntersec - arrowLength * cos(angle - arrowAngle);
  double y3 = yIntersec - arrowLength * sin(angle - arrowAngle);
  double x4 = xIntersec - arrowLength * cos(angle + arrowAngle);
  double y4 = yIntersec - arrowLength * sin(angle + arrowAngle);

  MoveToEx(hdc, x3, y3, NULL);
  LineTo(hdc, xIntersec, yIntersec);
  LineTo(hdc, x4, y4);
}

double **makeGraph(int n) {
  srand(4227);
  double **graph = (double **) malloc(n * sizeof(double *));
  for (int i = 0; i < n; i++) {
```

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

double **formatGraph(double **arr, int vertices, double multiplier) {
  for (int i = 0; i < vertices; i++) {
    for (int j = 0; j < vertices; j++) {
      arr[i][j] *= multiplier;
      arr[i][j] = arr[i][j] < 1.0 ? 0 : 1;
    }
  }
  return arr;
}

double **mirrorGraph(double **arr, int vertices) {
  double **undirected = (double **) malloc(vertices * sizeof(double *));
  for (int i = 0; i < vertices; i++) {
    undirected[i] = (double *) malloc(vertices * sizeof(double));
    for (int j = 0; j < vertices; j++) {
      undirected[i][j] = arr[i][j];
    }
  }

  for (int i = 0; i < vertices; i++) {
```

```
    for (int j = 0; j < vertices; j++) {
      if (undirected[i][j] != undirected[j][i]) {
        undirected[i][j] = 1;
        undirected[j][i] = 1;
      }
    }
  }

  return undirected;
}

void deleteMatrix(double **arr, int vertices) {
  for (int i = 0; i < vertices; i++) {
    free(arr[i]);
  }
  free(arr);
}

void getInfoForDirected(double **arr, int vertices) {
  int degrees[vertices];
  for (int i = 0; i < vertices; i++) {
    int indegree = 0;
    int outdegree = 0;
    for (int j = 0; j < vertices; j++) {
      if (arr[i][j] == 1) {
        indegree++;
      }
      if (arr[j][i] == 1) {
```

```
        outdegree++;
      }
    }
    degrees[i] = indegree + outdegree;
    printf("Vertex: %d. Indegree: %d, outdegree: %d\n", (i + 1), indegree, outdegree);
  }

  printf("Hanging and isolated vertices:\n");
  for (int i = 0; i < vertices; i++) {
    if (degrees[i] == 0) {
      printf("Vertex %d is isolated\n", (i + 1));
    }
    if (degrees[i] == 1) {
      printf("Vertex %d is hanging\n", (i + 1));
    }
  }

  for (int i = 1; i < vertices; i++) {
    if (degrees[i] != degrees[i - 1]) {
      printf("Graph is irregular\n");
      return;
    }
  }
  printf("Graph is regular. Degree: %d\n", degrees[0]);
}

void getInfoForUndirected(double **arr, int vertices) {
  int degrees[vertices];
```

```
for (int i = 0; i < vertices; i++) {
  int degree = 0;
  for (int j = 0; j < vertices; j++) {
    if (arr[i][j] == 1) {
      degree++;
    }
  }
  degrees[i] = degree;
  printf("Vertex: %d. Degree: %d\n", (i + 1), degree);
}

printf("Hanging and isolated vertices:\n");
for (int i = 0; i < vertices; i++) {
  if (degrees[i] == 0) {
    printf("Vertex %d is isolated\n", (i + 1));
  }
  if (degrees[i] == 1) {
    printf("Vertex %d is hanging\n", (i + 1));
  }
}

for (int i = 1; i < vertices; i++) {
  if (degrees[i] != degrees[i - 1]) {
    printf("Graph is irregular\n");
    return;
  }
}
printf("Graph is regular. Degree: %d\n", degrees[0]);
```

```
}

void printGraph(double **arr, int n) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      printf("%.0f ", arr[i][j]);
    }
    printf("\n");
  }
}

double **findTwoLengthWays(double **arr, int vertices) {
  double **twoWay = (double **) malloc(vertices * sizeof(double *));

  for (int i = 0; i < vertices; i++) {
    twoWay[i] = (double *) malloc(vertices * sizeof(double));
    for (int j = 0; j < vertices; j++) {
      for (int k = 0; k < vertices; k++) {
        twoWay[i][j] += arr[i][k] * arr[k][j];
      }
    }
  }

  printf ("Ways of length 2:\n");

  for (int i = 0; i < vertices; i++) {
    for (int j = 0; j < vertices; j++) {
      if (i == j && twoWay[i][j] > 0) {
```

```
        for (int k = 0; k < vertices; k++) {
          if (arr[i][k] == arr[k][i] && arr[i][k] == 1) {
            printf("%d -> %d -> %d\n", (i + 1), (k + 1), (i + 1));
          }
        }
      }
      if (twoWay[i][j] > 0) {
        for (int k = 0; k < vertices; k++) {
          if (arr[k][j] == 1 && arr[i][k] == 1 && j != i) {
            printf("%d -> %d -> %d\n", (i + 1), (k + 1), (j + 1));
          }
        }
      }
    }
  }
  printf("\n");
  return twoWay;
}

double **findThreeLengthWays(double **arr, double **square, int vertices) {
  double **threeWay = (double **) malloc(vertices * sizeof(double *));

  for (int i = 0; i < vertices; i++) {
    threeWay[i] = (double *) malloc(vertices * sizeof(double));
    for (int j = 0; j < vertices; j++) {
      for (int k = 0; k < vertices; k++) {
        threeWay[i][j] += square[i][k] * arr[k][j];
      }
```

```
  }
}

printf ("Ways of length 3:\n");

for (int i = 0; i < vertices; i++) {
  for (int j = 0; j < vertices; j++) {
    if (i == j && threeWay[i][j] > 0) {
      for (int k = 0; k < vertices; k++) {
        if (arr[k][j] == 1) {
          for (int l = 0; l < vertices; l++) {
            if (arr[l][k] == 1) {
              if (arr[i][l] == 1) {
                printf("%d -> %d -> %d -> %d\n", (i + 1), (l + 1), (k + 1), (i + 1));
              }
            }
          }
        }
      }
    }

    if (threeWay[i][j] > 0 && j != i) {
      for (int k = 0; k < vertices; k++) {
        if (arr[k][j] == 1) {
          for (int l = 0; l < vertices; l++) {
            if (arr[l][k] == 1) {
              if (arr[i][l] == 1) {
                printf("%d -> %d -> %d -> %d\n", (i + 1), (l + 1), (k + 1), (j + 1));
```

```
              }
            }
          }
        }
      }
    }
  }
 }

 printf("\n");
 return threeWay;
}

double **getReachabilityMatrix (double **arr, int vertices) {

 double **power = (double **) malloc(vertices * sizeof(double *));
 for (int i = 0; i < vertices; i++) {
  power[i] = (double *) malloc(vertices * sizeof(double));
  for (int j = 0; j < vertices; j++) {
   power[i][j] = arr[i][j];
  }
 }

 double **reachability = (double **) malloc(vertices * sizeof(double *));
 for (int i = 0; i < vertices; i++) {
  reachability[i] = (double *) malloc(vertices * sizeof(double));
  for (int j = 0; j < vertices; j++) {
   if (i == j) reachability[i][j] = 1;
```

```
    else reachability[i][j] = 0;
  }
}

for (int a = 0; a < 10; a++) {

  for (int i = 0; i < vertices; i++) {
    for (int j = 0; j < vertices; j++) {
      reachability[i][j] += power[i][j];
    }
  }

  for (int i = 0; i < vertices; i++) {
    for (int j = 0; j < vertices; j++) {
      for (int k = 0; k < vertices; k++) {
        power[i][j] += power[i][k] * power[k][j];
      }
    }
  }

}

for (int i = 0; i < vertices; i++) {
  for (int j = 0; j < vertices; j++) {
    reachability[i][j] += power[i][j];
    reachability[i][j] = reachability[i][j] != 0;
  }
}
```

```
  return reachability;
}

double **findStrongComponents (double **arr, int vertices) {
  double **reachSquare = (double **) malloc(vertices * sizeof(double *));

  for (int i = 0; i < vertices; i++) {
    reachSquare[i] = (double *) malloc(vertices * sizeof(double));
    for (int j = 0; j < vertices; j++) {
      for (int k = 0; k < vertices; k++) {
        reachSquare[i][j] += arr[i][k] * arr[k][j];
      }
    }
  }

  for (int i = 1; i <= vertices; i++) {
    int used[vertices];
    int currentIndex = 0;
    for (int j = 0; j < vertices; j++) {
      if (reachSquare[j][j] == i) {
        used[currentIndex] = j;
        currentIndex++;

        if (j == 10) {
          printf("\nCount of bonds in component: %d\n", i);
          printf("Vertices in the component: ");
          for (int k = 0; k < currentIndex; k++) {
```

```
          printf("%d ", (used[k] + 1));
         }
         printf("\n");
        }


       }
      }
     }

     double **flippedReach = (double **) malloc(vertices * sizeof(double *));
     for (int i = 0; i < vertices; i++) {
      flippedReach[i] = (double *) malloc(vertices * sizeof(double));
      for (int j = 0; j < vertices; j++) {
       flippedReach[i][j] = arr[j][i];
      }
     }

     double **strongConnect = (double **) malloc(vertices * sizeof(double *));
     for (int i = 0; i < vertices; i++) {
      strongConnect[i] = (double *) malloc(vertices * sizeof(double));
      for (int j = 0; j < vertices; j++) {
       strongConnect[i][j] = arr[i][j] * flippedReach[i][j];
      }
     }

     return strongConnect;
    }
```

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT messg, WPARAM wParam,
LPARAM lParam) {
  switch (messg) {
    case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      double **directed = makeGraph(vert);
      directed = formatGraph(directed, vert, (1.0 - n3 * 0.01 - n4 * 0.01 - 0.3));
      printf ("\nThe directed graph matrix:\n");
      printGraph(directed, vert);

      double **undirected = mirrorGraph(directed, vert);
      printf("\nThe undirected graph matrix:\n");
      printGraph(undirected, vert);

      printf("\nInformation about directed graph:\n");
      getInfoForDirected(directed, vert);

      printf("\nInformation about undirected graph:\n");
      getInfoForUndirected(undirected, vert);

      double **modifiedDirected = makeGraph(vert);
      modifiedDirected = formatGraph(modifiedDirected, vert, (1.0 - n3 * 0.005 - n4 *
0.005 - 0.27));
      printf ("\nThe modified directed graph matrix:\n");
      printGraph(modifiedDirected, vert);
```

```
printf("\nInformation about modified directed graph:\n");
getInfoForDirected(modifiedDirected, vert);

double **twoWay = findTwoLengthWays(modifiedDirected, vert);
printf("In the form of matrix:\n");
printGraph(twoWay, vert);

double **threeWay = findThreeLengthWays(modifiedDirected, twoWay, vert);
printf("In the form of matrix:\n");
printGraph(threeWay, vert);

printf("\nReachability matrix:\n");
double **reachability = getReachabilityMatrix(modifiedDirected, vert);
printGraph(reachability, vert);

double **connectivity = findStrongComponents(reachability, vert);
printf("\nStrong connectivity matrix: \n");
printGraph(connectivity, vert);

char *nn[11] = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11"};
int circleRadius = 250;
int Xcenter = 500;
int Ycenter = 300;
int nx[11], ny[11] = {};
for (int i = 0; i < 10; i++) {
  double angle = i * 36;
  nx[i] = Xcenter + (int) (circleRadius * sin(angle * (3.14 / 180)));
```

```
    ny[i] = Ycenter - (int) (circleRadius * cos(angle * (3.14 / 180)));
  }
  nx[10] = Xcenter;
  ny[10] = Ycenter;
  int dx = 16, dy = 16, dtx = 5;
  HPEN BPen = CreatePen(PS_SOLID, 2, RGB(50, 0, 255));
  HPEN KPen = CreatePen(PS_SOLID, 1, RGB(20, 20, 5));

  SelectObject(hdc, KPen);

  if (variant == 1) {
    for (int i = 0; i < vert; i++) {
      for (int j = 0; j < vert; j++) {
        boolean left = (nx[j] < Xcenter);

        if (directed[i][j] == 1 && j != i && directed[j][i] != directed[i][j]) {
          int avgX = (nx[i] + nx[j]) / 2;
          int avgY = (ny[i] + ny[j]) / 2;
          MoveToEx(hdc, nx[i], ny[i], NULL);

          if (avgX >= Xcenter - 1 && avgX <= Xcenter + 1 && avgY >= Ycenter - 1
&& avgY <= Ycenter + 1) {
            LineTo(hdc, nx[10] + 30, (int) ((ny[10] + ny[j]) / 2.04));
            MoveToEx(hdc, nx[10] + 30, (int) ((ny[10] + ny[j]) / 2.04), NULL);
            LineTo(hdc, nx[j], ny[j]);
            arrow(hdc, nx[10] + 30, (int) ((ny[10] + ny[j]) / 2.04), nx[j], ny[j], dx);
          } else {
            LineTo(hdc, nx[j], ny[j]);
```

```
      arrow(hdc, nx[i], ny[i], nx[j], ny[j], dx);
    }
  }
  if ((j == i) && (directed[i][j] == 1)) {
    if (left) {
      Ellipse(hdc, nx[i] - 2 * dx, ny[i] - 2 * dy, nx[i], ny[i]);
      arrow(hdc, nx[i] - 0.2 * dx, ny[i] - 2.1 * dy, nx[j] + 2, ny[j] - 0.1 * dy, dx);
    } else {
      Ellipse(hdc, nx[i], ny[i], nx[i] + 2 * dx, ny[i] + 2 * dy);
      arrow(hdc, nx[i] + 0.2 * dx, ny[i] + 2.1 * dy, nx[j] - 2, ny[j] + 0.1 * dy, dx);
    }
  }
  if (directed[i][j] == 1 && j != i && directed[j][i] == directed[i][j]) {
    if (i < j) {
      MoveToEx(hdc, nx[i], ny[i], NULL);
      LineTo(hdc, ((nx[i] + nx[j]) / 2 + dx), ((ny[i] + ny[j]) / 2 + dy));
      MoveToEx(hdc, ((nx[i] + nx[j]) / 2 + dx), ((ny[i] + ny[j]) / 2 + dy), NULL);
      LineTo(hdc, nx[j], ny[j]);
      arrow(hdc, ((nx[i] + nx[j]) / 2 + dx), ((ny[i] + ny[j]) / 2 + dy), nx[i], ny[i], dx);
    }
    if (i > j) {
      MoveToEx(hdc, nx[j], ny[j], NULL);
      LineTo(hdc, ((nx[i] + nx[j]) / 2 - dx), ((ny[i] + ny[j]) / 2 - dy));
      MoveToEx(hdc, ((nx[i] + nx[j]) / 2 - dx), ((ny[i] + ny[j]) / 2 - dy), NULL);
      LineTo(hdc, nx[i], ny[i]);
      arrow(hdc, ((nx[i] + nx[j]) / 2 - dx), ((ny[i] + ny[j]) / 2 - dy), nx[i], ny[i], dx);
    }
  }
```

```
      }
    }
  }

  if (variant == 2) {
    for (int i = 0; i < vert; i++) {
      for (int j = i; j < vert; j++) {
        boolean left = (nx[j] < Xcenter);

        if (undirected[i][j] == 1 && j != i) {
          int avgX = (nx[i] + nx[j]) / 2;
          int avgY = (ny[i] + ny[j]) / 2;
          MoveToEx(hdc, nx[i], ny[i], NULL);

          if (avgX >= Xcenter - 1 && avgX <= Xcenter + 1 && avgY >= Ycenter - 1
&& avgY <= Ycenter + 1) {
            LineTo(hdc, nx[10] + 30, (int) ((ny[10] + ny[j]) / 2.1));
            MoveToEx(hdc, nx[10] + 30, (int) ((ny[10] + ny[j]) / 2.1), NULL);
            LineTo(hdc, nx[j], ny[j]);
          } else {
            LineTo(hdc, nx[j], ny[j]);
          }
        }
        if ((j == i) && (undirected[i][j] == 1)) {
          if (left) {
            Ellipse(hdc, nx[i] - 2 * dx, ny[i] - 2 * dy, nx[i], ny[i]);
          } else {
            Ellipse(hdc, nx[i], ny[i], nx[i] + 2 * dx, ny[i] + 2 * dy);
```

```
            }
          }
        }
      }
    }

    if (variant == 3) {
      for (int i = 0; i < vert; i++) {
        for (int j = 0; j < vert; j++) {
          boolean left = (nx[j] < Xcenter);

          if (modifiedDirected[i][j] == 1 && j != i && modifiedDirected[j][i] !=
modifiedDirected[i][j]) {
            int avgX = (nx[i] + nx[j]) / 2;
            int avgY = (ny[i] + ny[j]) / 2;
            MoveToEx(hdc, nx[i], ny[i], NULL);

            if (avgX >= Xcenter - 1 && avgX <= Xcenter + 1 && avgY >= Ycenter - 1
&& avgY <= Ycenter + 1) {
              LineTo(hdc, nx[10] + 30, (int) ((ny[10] + ny[j]) / 2.04));
              MoveToEx(hdc, nx[10] + 30, (int) ((ny[10] + ny[j]) / 2.04), NULL);
              LineTo(hdc, nx[j], ny[j]);
              arrow(hdc, nx[10] + 30, (int) ((ny[10] + ny[j]) / 2.04), nx[j], ny[j], dx);
            } else {
              LineTo(hdc, nx[j], ny[j]);
              arrow(hdc, nx[i], ny[i], nx[j], ny[j], dx);
            }
          }
          if ((j == i) && (modifiedDirected[i][j] == 1)) {
```

```
    if (left) {
      Ellipse(hdc, nx[i] - 2 * dx, ny[i] - 2 * dy, nx[i], ny[i]);
      arrow(hdc, nx[i] - 0.2 * dx, ny[i] - 2.1 * dy, nx[j] + 2, ny[j] - 0.1 * dy, dx);
    } else {
      Ellipse(hdc, nx[i], ny[i], nx[i] + 2 * dx, ny[i] + 2 * dy);
      arrow(hdc, nx[i] + 0.2 * dx, ny[i] + 2.1 * dy, nx[j] - 2, ny[j] + 0.1 * dy, dx);
    }
  }
  if (modifiedDirected[i][j] == 1 && j != i && modifiedDirected[j][i] ==
modifiedDirected[i][j]) {
    if (i < j) {
      MoveToEx(hdc, nx[i], ny[i], NULL);
      LineTo(hdc, ((nx[i] + nx[j]) / 2 + dx), ((ny[i] + ny[j]) / 2 + dy));
      MoveToEx(hdc, ((nx[i] + nx[j]) / 2 + dx), ((ny[i] + ny[j]) / 2 + dy), NULL);
      LineTo(hdc, nx[j], ny[j]);
      arrow(hdc, ((nx[i] + nx[j]) / 2 + dx), ((ny[i] + ny[j]) / 2 + dy), nx[i], ny[i], dx);
    }
    if (i > j) {
      MoveToEx(hdc, nx[j], ny[j], NULL);
      LineTo(hdc, ((nx[i] + nx[j]) / 2 - dx), ((ny[i] + ny[j]) / 2 - dy));
      MoveToEx(hdc, ((nx[i] + nx[j]) / 2 - dx), ((ny[i] + ny[j]) / 2 - dy), NULL);
      LineTo(hdc, nx[i], ny[i]);
      arrow(hdc, ((nx[i] + nx[j]) / 2 - dx), ((ny[i] + ny[j]) / 2 - dy), nx[i], ny[i], dx);
    }
  }
 }
}
```

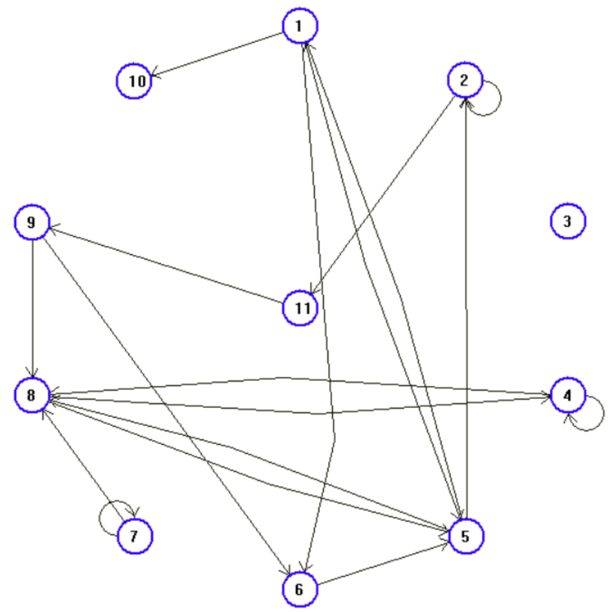
```
int components = 0;
int connected;
int diff;
for (int i = 1; i <= vert; i++) {
  connected = 0;
  int used[vert];
  int currentIndex = 0;
  diff = -1;
  for (int j = 0; j < vert; j++) {
    if (reachability[j][j] == i) {
      used[currentIndex] = j;
      currentIndex++;
      if (j == 10) {
        components++;
      }
    }
  }
}

char *cond[11] = {"K1", "K2", "K3", "K4", "K5", "K6", "K7", "K8", "K9", "K10",
"K11"};
SelectObject(hdc, BPen);
int cx[11], cy[11] = {};
int rCon = 50;
int xconCen = 800;
int yconCen = 600;
for (int i = 0; i < 10; i++) {
  double angle = i * (360 / 11);
```

```
    cx[i] = xconCen + (int) (rCon * sin(angle * (3.14 / 180)));
    cy[i] = yconCen - (int) (rCon * cos(angle * (3.14 / 180)));
  }

  double **condMatrix = (double **) malloc(components * sizeof(double *));
  for (int i = 0; i < components; i++) {
    condMatrix[i] = (double *) malloc(components * sizeof(double));
    for (int j = 0; j < components; j++) {
      if (j != i) {
        MoveToEx(hdc, cx[i], cy[i], NULL);
        LineTo(hdc, cx[j], cy[j]);
        arrow(hdc, cx[i], cy[i], cx[j], cy[j], dx);
      }
    }
  }
  printf("Condensation matrix:\n");
  printGraph(condMatrix, components);

  for (int i = 0; i < components; i++) {
    Ellipse(hdc, cx[i] - dx, cy[i] - dy, cx[i] + dx, cy[i] + dy);
    TextOut(hdc, cx[i] - dtx, cy[i] - dy / 2, cond[i], 2);
  }
  deleteMatrix(condMatrix, components);
}

SelectObject(hdc, BPen);
for (int i = 0; i < 11; i++) {
  Ellipse(hdc, nx[i] - dx, ny[i] - dy, nx[i] + dx, ny[i] + dy);
```

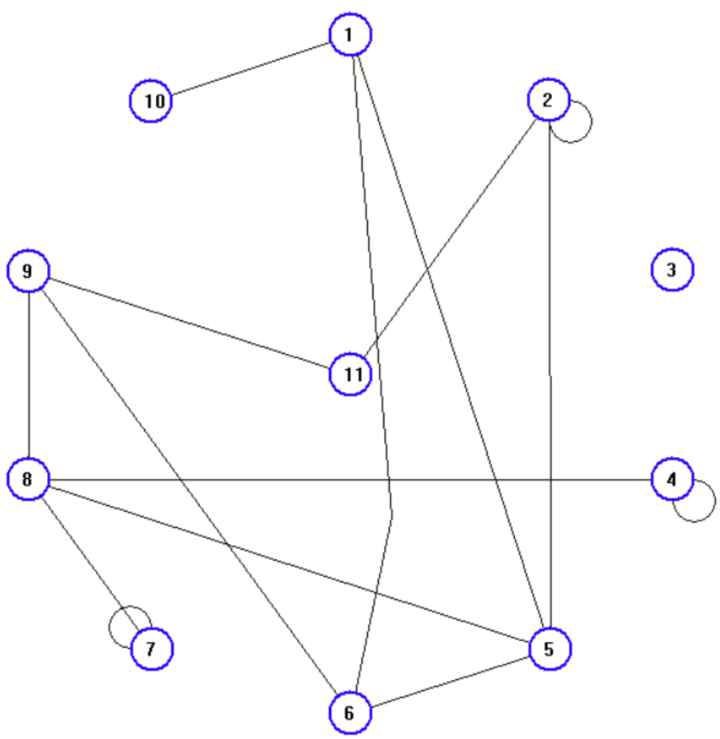
```
        TextOut(hdc, nx[i] - dtx, ny[i] - dy / 2, nn[i], 2);
      }

      EndPaint(hWnd, &ps);
      deleteMatrix(directed, vert);
      deleteMatrix(undirected, vert);
      deleteMatrix(modifiedDirected, vert);
      deleteMatrix(twoWay, vert);
      deleteMatrix(threeWay, vert);
      deleteMatrix(reachability, vert);
      deleteMatrix(connectivity, vert);
      break;
    case WM_DESTROY:
      PostQuitMessage(0);
      break;
    default:
      return (DefWindowProc(hWnd, messg, wParam, lParam));
  }
}
```

Результати тестування:

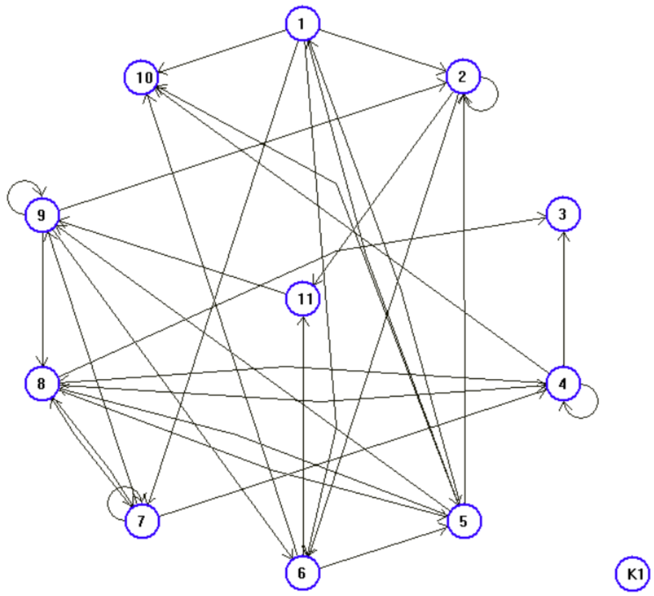
1 для направленого



2 для ненаправленого



3 модифікований




```
The directed graph matrix:
0 0 0 0 1 1 0 0 0 1 0
0 1 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 1 0 0 0
1 1 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 1 1 0 0 0
0 0 0 1 1 0 0 0 0 0 0
0 0 0 0 0 1 0 1 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

The undirected graph matrix:
0 0 0 0 1 1 0 0 0 1 0
0 1 0 0 1 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 1 0 0 0
1 1 0 0 0 1 0 1 0 0 0
1 0 0 0 1 0 0 0 1 0 0
0 0 0 0 0 0 1 1 0 0 0
0 0 0 1 1 0 1 0 1 0 0
0 0 0 0 0 1 0 1 0 0 1
1 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 1 0 0
```

```
Information about directed graph:
Vertex: 1. Indegree: 3, outdegree: 1
Vertex: 2. Indegree: 2, outdegree: 2
Vertex: 3. Indegree: 0, outdegree: 0
Vertex: 4. Indegree: 2, outdegree: 2
Vertex: 5. Indegree: 3, outdegree: 3
Vertex: 6. Indegree: 1, outdegree: 2
Vertex: 7. Indegree: 2, outdegree: 1
Vertex: 8. Indegree: 2, outdegree: 4
Vertex: 9. Indegree: 2, outdegree: 1
Vertex: 10. Indegree: 0, outdegree: 1
Vertex: 11. Indegree: 1, outdegree: 1
Hanging and isolated vertices:
Vertex 3 is isolated
Vertex 10 is hanging
Graph is irregular

Information about undirected graph:
Vertex: 1. Degree: 3
Vertex: 2. Degree: 3
Vertex: 3. Degree: 0
Vertex: 4. Degree: 2
Vertex: 5. Degree: 4
Vertex: 6. Degree: 3
Vertex: 7. Degree: 2
Vertex: 8. Degree: 4
Vertex: 9. Degree: 3
Vertex: 10. Degree: 1
Vertex: 11. Degree: 2
Hanging and isolated vertices:
Vertex 3 is isolated
Vertex 10 is hanging
Graph is irregular
```

```
The modified directed graph matrix:
0 1 0 0 1 1 1 0 0 1 0
0 1 0 0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 0 0 0 1 0 1 0
1 1 0 0 0 0 0 1 1 1 0
0 0 0 0 1 0 0 0 0 1 1
0 0 0 1 0 0 1 1 1 0 0
0 0 1 1 1 0 1 0 0 0 0
0 1 0 0 0 1 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0

Information about modified directed graph:
Vertex: 1. Indegree: 5, outdegree: 1
Vertex: 2. Indegree: 3, outdegree: 4
Vertex: 3. Indegree: 0, outdegree: 2
Vertex: 4. Indegree: 4, outdegree: 3
Vertex: 5. Indegree: 5, outdegree: 3
Vertex: 6. Indegree: 3, outdegree: 3
Vertex: 7. Indegree: 4, outdegree: 3
Vertex: 8. Indegree: 4, outdegree: 4
Vertex: 9. Indegree: 4, outdegree: 4
Vertex: 10. Indegree: 0, outdegree: 4
Vertex: 11. Indegree: 1, outdegree: 2
Hanging and isolated vertices:
Graph is irregular
```

```
Reachability matrix:
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

Count of bonds in component: 11
Vertices in the component: 1 2 3 4 5 6 7 8 9 10 11

Strong connectivity matrix:
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
Condensation matrix:
0
```

Ways of length 2:

2 -> 2 -> 2

5 -> 1 -> 6

5 -> 2 -> 6

5 -> 9 -> 6

5 -> 1 -> 7

5 -> 8 -> 7

7 -> 9 -> 6

7 -> 7 -> 7

7 -> 8 -> 7

9 -> 2 -> 2

9 -> 9 -> 2

9 -> 8 -> 3

9 -> 8 -> 4

9 -> 6 -> 5

9 -> 8 -> 5

9 -> 2 -> 6

9 -> 9 -> 6

9 -> 8 -> 7

9 -> 9 -> 8

9 -> 9 -> 9

9 -> 6 -> 10

9 -> 2 -> 11

9 -> 6 -> 11

11 -> 9 -> 2

11 -> 9 -> 6

11 -> 9 -> 8

11 -> 9 -> 9

Ways of length 3:

9 -> 6 -> 5 -> 1

9 -> 8 -> 5 -> 1

9 -> 2 -> 2 -> 2

9 -> 9 -> 2 -> 2

9 -> 6 -> 5 -> 2

9 -> 8 -> 5 -> 2

9 -> 9 -> 9 -> 2

9 -> 8 -> 4 -> 3

9 -> 9 -> 8 -> 3

9 -> 8 -> 4 -> 4

9 -> 8 -> 7 -> 4

9 -> 9 -> 8 -> 4

9 -> 2 -> 6 -> 5

9 -> 9 -> 6 -> 5

9 -> 9 -> 8 -> 5

9 -> 2 -> 2 -> 6

9 -> 9 -> 2 -> 6

9 -> 9 -> 9 -> 6

9 -> 8 -> 7 -> 7

9 -> 9 -> 8 -> 7

9 -> 8 -> 4 -> 8

9 -> 6 -> 5 -> 8

9 -> 8 -> 5 -> 8

9 -> 8 -> 7 -> 8

9 -> 9 -> 9 -> 8

9 -> 6 -> 5 -> 9

9 -> 8 -> 5 -> 9

9 -> 8 -> 7 -> 9

9 -> 9 -> 9 -> 9

9 -> 2 -> 11 -> 9

9 -> 6 -> 11 -> 9

9 -> 8 -> 4 -> 10

9 -> 6 -> 5 -> 10

9 -> 8 -> 5 -> 10

9 -> 2 -> 6 -> 10

9 -> 9 -> 6 -> 10

9 -> 2 -> 2 -> 11

9 -> 9 -> 2 -> 11

9 -> 2 -> 6 -> 11

9 -> 9 -> 6 -> 11

11 -> 9 -> 2 -> 2

11 -> 9 -> 9 -> 2

11 -> 9 -> 8 -> 3

11 -> 9 -> 8 -> 4

11 -> 9 -> 6 -> 5

11 -> 9 -> 8 -> 5

11 -> 9 -> 2 -> 6

11 -> 9 -> 9 -> 6

11 -> 9 -> 8 -> 7

11 -> 9 -> 9 -> 8

11 -> 9 -> 9 -> 9

11 -> 9 -> 6 -> 10

11 -> 9 -> 2 -> 11

11 -> 9 -> 6 -> 11

Висновки:

У ході виконання цієї лабораторної роботи я ознайомився з характеристиками графів та методом транзитивного замикання. Були вивчені основні поняття, такі як степені вершин, висячі та ізольовані вершини, однорідність графу, матриця суміжності, півстепені вузлів, шляхи довжиною 2 і 3, матриця досяжності, компоненти сильної зв'язності, матриця зв'язності та граф конденсації. Здобуті знання та навички можуть бути застосовані для вирішення різноманітних задач, пов'язаних з аналізом та моделюванням графів у різних сферах, включаючи комп'ютерні науки, транспортні мережі, соціальні мережі та багато інших.