

Балтийский государственный технический университет «Военмех» им. Д. Ф. Устинова

Кафедра И5
«Информационные системы и программная инженерия»

ЛАБОРАТОРНАЯ РАБОТА № 2
По дисциплине «**ТЕОРИЯ АВТОМАТОВ И ФОРМАЛЬНЫХ ЯЗЫКОВ**»
На тему
Построение орграфа.

Вариант № 3

Выполнил:
Студент Васильев Н.А.
Группа И967

Преподаватель:
Суслов В.П.

Санкт-Петербург
2019

Задание

Написать программу построения обычного и орграфа. Для обычного графа сделать обход для нечетных вариантов в ширину, для четных в глубину. Последовательность обхода вершин вывести на экран монитора. Для орграфа построить матрицу инцидентности. Граф задается матрицей смежности и списком смежности. Исходные данные хранятся во внешнем форматном файле. Выбор способа задания графа определяется пунктом меню программы.

Особенности реализации

Описывается абстрактный класс `Graph`, содержащий виртуальные методы для загрузки данных из файла, вывода информации на экран, работы с матрицей инцидентности и обходов графа в ширину и глубину.

Классы `matrixGraph` и `listGraph` наследуются от `Graph` и представляют собой реализации интерфейса для задания графа с помощью матрицы смежности и списка смежности.

Программа использует следующие контейнеры из библиотеки стандартных шаблонов (STL): `vector`, `stack`, `queue`, `list`.

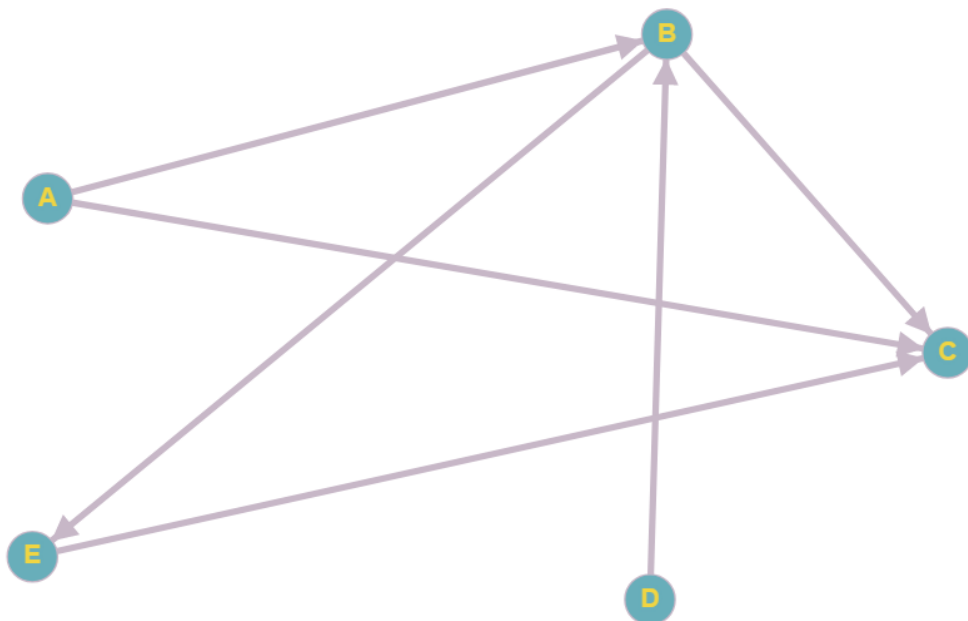


Рисунок 1 – ориентированный граф

Исходный код программы

```
#include <fstream>
#include <iostream>
#include <sstream>
#include <vector>
#include <conio.h>
#include <windows.h>
#include <queue>
#include <stack>
#include <list>
#include <iomanip>
using namespace std;
const char verticesNames[5] = { 'A', 'B', 'C', 'D', 'E'};

class Graph {
protected:
    int vCounter = 0;
    int eCounter = 0;
    vector<vector<int> > incidenceMatrix;
    virtual void printGraph() {};
    virtual void dfs(int startNode) {};
    virtual void bfs(int startNode) {};
    virtual void makeIncidenceMatrix() {};
public:
    virtual void loadGraph(string filename) {};

    void printIncidenceMatrix() {
        cout << "\n ";
        for (int i = 0; i < eCounter; i++) {
            int from = -1, to = -1;
            for (int j = 0; j < vCounter; j++) {
                incidenceMatrix[j][i] == 1 ? from = j : incidenceMatrix[j][i]
== -1 ? to = j: NULL;
            }
            cout << setw(4) << verticesNames[from] << " -> " <<
verticesNames[to];
        }
        cout << endl;
        for (int i = 0; i < vCounter; i++) {
            cout << verticesNames[i];
            for (int j = 0; j < eCounter; j++) {
                cout << right << setw(8) << incidenceMatrix[i][j] << " ";
            }
        }
    }
};
```

```

        }
        cout << endl;
    }
};

void outputAll(bool directed) {
    cout << "\nGraph:";
    printGraph();
    if (directed) {
        cout << "\nIncidence matrix:";
        makeIncidenceMatrix();
        printIncidenceMatrix();
    }
    cout << "\nBFS:\n";
    for (int i = 0; i < vCounter; i++)
        bfs(i);
    cout << "\nDFS:\n";
    for (int i = 0; i < vCounter; i++)
        dfs(i);
    cout << endl;
}

};

class matrixGraph: public Graph {
private:
    vector<vector<int> > adjacencyMatrix;

    void printGraph() {
        cout << "\n ";
        for (int i = 0; i < vCounter; i++) {
            cout << setw(3) << verticesNames[i];
        }
        cout << endl;
        for (int i = 0; i < vCounter; i++) {
            cout << verticesNames[i];
            for (int j = 0; j < vCounter; j++)
                cout << setw(3) << adjacencyMatrix[i][j];
            cout << '\n';
        }
    }
}

```

```

void makeIncidenceMatrix() {
    for (int i = 0; i < vCounter; i++) {
        for (int j = 0; j < i; j++) {
            if (adjacencyMatrix[i][j] || adjacencyMatrix[j][i]) {
                eCounter++;
            }
        }
    }
    for (int i = 0; i < vCounter; i++) {
        incidenceMatrix.push_back(vector<int>(eCounter));
        for (int j = 0; j < eCounter; j++) {
            incidenceMatrix[i][j] = 0;
        }
    }
    int edge = 0;
    for (int i = 0; i < vCounter; i++) {
        for (int j = 0; j < i; j++) {
            if (adjacencyMatrix[i][j] == 1 && adjacencyMatrix[j][i] == 0)
{
                incidenceMatrix[i][edge] = 1;
                incidenceMatrix[j][edge++] = -1;
            }
            else if (adjacencyMatrix[i][j] == 0 && adjacencyMatrix[j][i] ==
1) {
                incidenceMatrix[i][edge] = -1;
                incidenceMatrix[j][edge++] = 1;
            }
        }
    }
}

```

```

void bfs(int startNode) {
    queue<int> q;
    vector<bool> visited(vCounter, false);
    q.push(startNode);
    while (!q.empty()) {
        startNode = q.front();
        q.pop();
        if (!visited[startNode]) {

```

```

        cout << verticesNames[startNode] << " ";
        visited[startNode] = true;
    }
    for (int i = 0; i < vCounter; i++) {
        if (!visited[i] && adjacencyMatrix[startNode][i] != 0) {
            q.push(i);
        }
    }
}
cout << endl;
}

void dfs(int startNode) {
    stack<int> s;
    vector<bool> visited(vCounter, false);
    s.push(startNode);
    while (!s.empty()) {
        startNode = s.top();
        s.pop();
        if (!visited[startNode]) {
            cout << verticesNames[startNode] << " ";
            visited[startNode] = true;
        }
        for (int i = 0; i < vCounter; i++) {
            if (!visited[i] && adjacencyMatrix[startNode][i] != 0) {
                s.push(i);
            }
        }
    }
    cout << endl;
}

public:
    void loadGraph(string filename) {
        string line;
        ifstream matrixStream(filename);
        if(!matrixStream) {
            cout << "Error\n";
            return;
        }
        while(getline(matrixStream, line)) {

```

```

        adjacencyMatrix.push_back(vector<int>());
        vCounter++;
        for(int i = 0; i < line.size(); i++) {
            if (line[i] != 32) {
                adjacencyMatrix.back().push_back((int)line[i] - (int)'0');
            }
        }
    }
};

```

```

class listGraph: public Graph {
private:
    vector<list<int> > adjacencyList;

    void printGraph() {
        cout << "\n";
        for (int i = 0; i < vCounter; i++) {
            cout << verticesNames[i] << ":";
            list<int>::iterator it;
            for (it = adjacencyList[i].begin(); it != adjacencyList[i].end();
it++)
                cout << setw(2) << verticesNames[*it];
            cout << '\n';
        }
    }
}

```

```

void makeIncidenceMatrix() {
    for (int i = 0; i < vCounter; i++) {
        eCounter += adjacencyList[i].size();
    }
    for (int i = 0; i < vCounter; i++) {
        incidenceMatrix.push_back(vector<int>(eCounter));
        for (int j = 0; j < eCounter; j++) {
            incidenceMatrix[i][j] = 0;
        }
    }
    int edge = 0;
    for (int i = 0; i < vCounter; i++) {
        for (int j = 0; j < adjacencyList[i].size(); j++) {

```

```

        incidenceMatrix[i][edge] = 1;
        list<int>::iterator it = adjacencyList[i].begin();
        advance(it, j);
        incidenceMatrix[*it][edge] = -1;
        edge++;
    }
}

}

void bfs(int startNode) {
    queue<int> q;
    vector<bool> visited(vCounter, false);
    q.push(startNode);
    while (!q.empty()) {
        startNode = q.front();
        q.pop();
        if (!visited[startNode]) {
            cout << verticesNames[startNode] << " ";
            visited[startNode] = true;
        }
        for (list<int>::iterator it = adjacencyList[startNode].begin(); it
!= adjacencyList[startNode].end(); it++) {
            if (!visited[*it]) {
                q.push(*it);
            }
        }
    }
    cout << endl;
}

void dfs(int startNode) {
    stack<int> s;
    vector<bool> visited(vCounter, false);
    s.push(startNode);
    while (!s.empty()) {
        startNode = s.top();
        s.pop();
        if (!visited[startNode]) {
            cout << verticesNames[startNode] << " ";
            visited[startNode] = true;

```



```

        }
        for (list<int>::iterator it = adjacencyList[startNode].begin(); it
!= adjacencyList[startNode].end(); it++) {
            if (!visited[*it]) {
                s.push(*it);
            }
        }
    }
    cout << endl;
}

public:
    void loadGraph(string filename) {
        string line;
        ifstream matrixStream(filename);
        if(!matrixStream) {
            cout << "Error\n";
            return;
        }
        while(getline(matrixStream, line)) {
            adjacencyList.push_back(list<int>());
            vCounter++;
            for(int i = 1; i < line.size(); i++) {
                if (line[i] != 32) {
                    adjacencyList.back().push_back((int)line[i] - (int)'0');
                }
            }
        }
    }
};

```

```

int main() {
    char switcher;
    while(switcher != 27) {
        system("cls");
        cout << "1 - Matrix (Directed)\n"
            << "2 - Matrix (Undirected)\n"
            << "3 - List (Directed)\n"
            << "4 - List (Undirected)\n"
            << "ESC - Quit\n";
        switcher = getch();
    }
}

```

```

switch(switcher) {
    case '1': {
        matrixGraph graph = matrixGraph();
        graph.loadGraph("graphMatrix_directed.txt");
        graph.outputAll(true);
    }
    break;
    case '2': {
        matrixGraph graph = matrixGraph();
        graph.loadGraph("graphMatrix_undirected.txt");
        graph.outputAll(false);
    }
    break;
    case '3': {
        listGraph graph = listGraph();
        graph.loadGraph("graphList_directed.txt");
        graph.outputAll(true);
    }
    break;
    case '4': {
        listGraph graph = listGraph();
        graph.loadGraph("graphList_undirected.txt");
        graph.outputAll(false);
    }
    break;
}
system("pause");
}
return 0;
}

```

Результаты работы программы

```
1 - Matrix (Directed)
2 - Matrix (Undirected)
3 - List (Directed)
4 - List (Undirected)
ESC - Quit
█
```

Рисунок 2 – меню

```
Graph:
  A B C D E
A 0 1 1 0 0
B 0 0 1 0 1
C 0 0 0 0 0
D 0 1 0 0 0
E 0 0 1 0 0

Incidence matrix:
  A -> B  A -> C  B -> C  D -> B  B -> E  E -> C
A      1      1      0      0      0      0
B     -1      0      1     -1      1      0
C      0     -1     -1      0      0     -1
D      0      0      0      1      0      0
E      0      0      0      0     -1      1

BFS:
A B C E
B C E
C
D B C E
E C

DFS:
A C B E
B E C
C
D B E C
E C

Для продолжения нажмите любую клавишу . . . █
```

Рисунок 3 – матрица смежности, матрица инцидентности и обходы ориентированного графа

```

Graph:
  A B C D E
A 0 1 1 0 0
B 1 0 1 1 1
C 1 1 0 0 1
D 0 1 0 0 0
E 0 1 1 0 0

BFS:
A B C D E
B A C D E
C A B E D
D B A C E
E B C A D

DFS:
A C E B D
B E C A D
C E B D A
D B E C A
E C B D A

Для продолжения нажмите любую клавишу . . . █

```

Рисунок 4 – матрица смежности и обходы неориентированного графа

```

Graph:
A: B C
B: C E
C:
D: B
E: C

Incidence matrix:
  A -> B  A -> C  B -> C  B -> E  D -> B  E -> C
A      1      1      0      0      0      0
B     -1      0      1      1     -1      0
C      0     -1     -1      0      0     -1
D      0      0      0      0      1      0
E      0      0      0     -1      0      1

BFS:
A B C E
B C E
C
D B C E
E C

DFS:
A C B E
B E C
C
D B E C
E C

Для продолжения нажмите любую клавишу . . . █

```

Рисунок 5 – список смежности, матрица инцидентности и обходы ориентированного графа

```
Graph:
A: B C
B: A C D E
C: A B E
D: B
E: B C

BFS:
A B C D E
B A C D E
C A B E D
D B A C E
E B C A D

DFS:
A C E B D
B E C A D
C E B D A
D B E C A
E C B D A

Для продолжения нажмите любую клавишу . . .
```

Рисунок 6 – список смежности и обходы неориентированного графа