# Санкт-Петербургский государственный политехнический университет

Кафедра “Прикладная математика и информатика”

Отчет по лабораторной работе 2 (вариант 10 (11))

“Алгоритмы и структуры данных”

Студент группы № 5030102/20001

ФИО: Дрекалов Никита Сергеевич

Выполнил 27.10.2023

Оглавление

[Санкт-Петербургский государственный политехнический университет 1](#_Toc149342661)

[**Постановка задачи** 2](#_Toc149342662)

[**Описание алгоритма** 3](#_Toc149342663)

[Алгоритм построения списка смежностей из матрицы (10): 3](#_Toc149342664)

[Алгоритм построения матрицы смежностей из списка (11): 3](#_Toc149342665)

[**Текст программы** 4](#_Toc149342666)

[list.hpp 4](#_Toc149342667)

[graph.hpp 4](#_Toc149342668)

[graph.cpp 4](#_Toc149342669)

[main.cpp 4](#_Toc149342670)

[**Описание тестирования** 4](#_Toc149342671)

# Постановка задачи

В лабораторной работе требуется на написанной ранее списке реализовать два способа представления графа – как матрицу смежностей и как список – и написать функцию постройки матрицы смежностей на основе списка смежностей (и наоборот…).

Пользователь программы может делать следующие запросы:

* Запустить сам алгоритм. Пользователю предложит ввести путь к файлу с матрицей/список смежностей и путь, в который будет записан обратный вариант.
* Вывести все запросы (на случай, если забудется одна из множества команд)
* Завершить программу.

Количество запросов со стороны пользователя неограниченно.

# Описание алгоритма

### Алгоритм построения списка смежностей из матрицы (10):

1. Считываем матрицу смежностей в переменную типа List<List<int>> с помощью функции readAdjMatrix().
2. Строим список смежностей с помощью функции asAdjList(). Для того, чтобы матрицу смежностей превратить в список, достаточно пройтись по строкам -> столбцам и добавлять в элемент, соответствующий столбцу, n раз элемент строки, где n – значение текущего элемента матрицы.
3. Записываем список в файл с помощью функции writeAdjList();

### Алгоритм построения матрицы смежностей из списка (11):

1. Считываем список смежностей в переменную типа List<List<int>> с помощью функции readAdjList().
2. Строим список смежностей с помощью функции asAdjMatrix(). Для того, чтобы список смежностей превратить в матрицу, достаточно пройтись по строкам -> столбцам матрицы и добавлять число, равное количеству встречаемости в списке у элемента, соответствующего столбцу, связей с элементом, соответствующего строке.
3. Записываем матрицу в файл с помощью функции writeAdjMatrix();

# Текст программы

Проект можно найти и посмотреть тут: [**GitHub**](https://github.com/Forthey/algorithms_ds/tree/main/task2_10_11)

### list.hpp

#pragma once  
#include <iostream>  
#include <string>  
#include <vector>  
  
namespace Alg {  
 /\*  
 \* Default functions  
 \*/  
  
 template<typename Data>  
 int defaultCompareFunction(const Data &data1, const Data &data2) {  
 return data1 - data2;  
 }  
   
 template<typename Data>  
 void defaultDisplayFunction(std::ostream &out, const Data &data) {  
 out << data;  
 }  
  
 /\*  
 \* Class List declaration  
 \*/  
  
 template <typename Data>  
 class List {  
 class ListElement {  
 public:  
 explicit ListElement(Data data, ListElement\* next) : data(data), next(next) {};  
 Data data;  
 ListElement\* next;  
 } \*head = nullptr;  
  
 class cachedElement {  
 public:  
 ListElement\* element = nullptr;  
 size\_t index = 0;  
 };  
  
 cachedElement cached;  
  
 size\_t length = 0;  
  
 Data& min(int(\*compareFunction)(const Data&, const Data&), size\_t a, size\_t b);  
 Data& max(int(\*compareFunction)(const Data&, const Data&), size\_t a, size\_t b);  
 void swap(Data& data1, Data& data2);  
 public:  
 explicit List();  
 List(const std::vector<Data>& array);  
 List(const List<Data> &list);  
  
 List<Data>& pushBack(const Data& data);  
 List<Data>& pushBack(const List<Data>& list);  
 List<Data>& pushFront(const Data& data);  
 List<Data>& pushFront(const List<Data>& list);  
 List<Data>& pushAt(size\_t index, const Data& data);  
 List<Data>& pushAt(size\_t index, const List<Data>& list);  
 List<Data>& pushSorted(const Data& data, int(\*compareFunction)(const Data&, const Data&) = defaultCompareFunction);  
 List<Data>& pushSorted(const List<Data>& list, int(\*compareFunction)(const Data&, const Data&) = defaultCompareFunction);  
  
 List<Data>& popBack();  
 List<Data>& popBack(Data& oldData);  
 List<Data>& popFront();  
 List<Data>& popFront(Data& oldData);  
 List<Data>& popAt(size\_t index);  
 List<Data>& popAt(size\_t index, Data& oldData);  
 void clear();  
 void resize(size\_t newSize);  
  
 bool empty() const;  
 size\_t size() const;  
 bool sorted(int(\*compareFunction)(const Data&, const Data&) = defaultCompareFunction) const;  
  
 void display(std::ostream& out = std::cout, void(\*displayFunction)(std::ostream &, const Data &) = defaultDisplayFunction) const;  
   
 size\_t find(const Data& data) const;  
  
 Data& min(int(\*compareFunction)(const Data&, const Data&) = defaultCompareFunction) const;  
 Data& max(int(\*compareFunction)(const Data&, const Data&) = defaultCompareFunction) const;  
 List<Data>& sort(int(\*compareFunction)(const Data&, const Data&) = defaultCompareFunction);  
  
 List<Data>& operator<<(const Data& data);  
 List<Data>& operator<<(const List<Data>& list);  
 List<Data>& operator>>(const Data& data);  
 List<Data>& operator=(const List<Data>& list);  
 List<Data> operator+(const List<Data>& list) const;  
 Data& operator[](size\_t index);  
  
 ~List();  
 };  
  
  
 /\*  
 \* Class List's functions definitions  
 \*/  
  
  
 template<typename Data>  
 List<Data>::List() {  
 }  
  
 template<typename Data>  
 List<Data>::List(const List<Data> &list) {  
 \*this = list;  
 }  
 template<typename Data>  
 List<Data>::List(const std::vector<Data>& array) {  
 for (const Data& element : array) {  
 pushBack(element);  
 }  
 }  
  
 template<typename Data>  
 List<Data>::~List() {  
 clear();  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::pushBack(const Data& data) {  
 ListElement\* newListElement = new ListElement(data, nullptr);  
  
 if (newListElement == nullptr)  
 return \*this;  
  
 if (head == nullptr) {  
 head = newListElement;  
 length++;  
 return \*this;  
 }  
  
 ListElement\* listElement = head;  
  
 for (size\_t i = 0; i < length - 1; i++)  
 listElement = listElement->next;  
 listElement->next = newListElement;  
  
 length++;  
  
 return \*this;  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::pushBack(const List<Data>& list) {  
 ListElement\* listElement = list.head;  
 for (size\_t i = 0; i < list.length; i++) {  
 (\*this).pushBack(listElement->data);  
 listElement = listElement->next;  
 }  
  
 return \*this;  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::pushFront(const Data& data) {  
 ListElement\* newListElement = new ListElement(data, head);  
  
 if (newListElement == nullptr)  
 return \*this;  
  
 length++;  
  
 head = newListElement;  
  
 return \*this;  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::pushFront(const List<Data>& list) {  
 ListElement\* listElement = list.head;  
 for (size\_t i = 0; i < list.length; i++) {  
 (\*this).pushAt(i, listElement->data);  
 listElement = listElement->next;  
 }  
  
 return \*this;  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::pushAt(size\_t index, const Data& data) {  
 if (head == nullptr) {  
 return pushBack(data);  
 }  
  
 if (index >= length) {  
 index = length - 1;  
 }  
  
 if (index == 0) {  
 return pushFront(data);  
 }  
  
 ListElement\* listElement = head;  
 for (size\_t i = 0; i < index - 1; i++) {  
 listElement = listElement->next;  
 }  
  
 ListElement\* newListElement = new ListElement(data, listElement->next);  
 if (newListElement == nullptr)  
 return \*this;  
  
 listElement->next = newListElement;  
  
 length++;  
  
 return \*this;  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::pushAt(size\_t index, const List<Data>& list) {  
 ListElement\* listElement = list.head;  
 for (size\_t i = 0; i < list.length; i++) {  
 (\*this).pushAt(i + index, listElement->data);  
 listElement = listElement->next;  
 }  
  
 return \*this;  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::pushSorted(const Data& data, int(\*compareFunction)(const Data&, const Data&)) {  
 if (head == nullptr) {  
 return pushBack(data);  
 }  
  
 ListElement\* listElement = head;  
 for (size\_t i = 0; i < length; i++) {  
 if (compareFunction(listElement->data, data) > 0) {  
 return pushAt(i, data);  
 }  
 listElement = listElement->next;  
 }  
  
 return pushBack(data);  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::pushSorted(const List<Data>& list, int(\*compareFunction)(const Data&, const Data&)) {  
 ListElement\* listElement = list.head;  
 for (size\_t i = 0; i < list.length; i++) {  
 pushSorted(listElement->data, compareFunction);  
 listElement = listElement->next;  
 }  
 return \*this;  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::popBack(Data& oldData) {  
 if (length == 0)  
 return \*this;  
  
 ListElement\* listElement = head;  
 ListElement\* previousListElement = nullptr;  
  
 for (size\_t i = 0; i < length - 1; i++) {  
 previousListElement = listElement;  
 listElement = listElement->next;  
 }  
  
 oldData = listElement->data;  
 delete listElement;  
  
 if (previousListElement != nullptr)  
 previousListElement->next = nullptr;  
 else  
 head = nullptr;  
  
 length--;  
  
 return \*this;  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::popBack() {  
 Data tmp;  
 return popBack(tmp);  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::popFront(Data& oldData) {  
 if (head == nullptr)  
 return \*this;;  
  
 ListElement\* oldElement = head;  
 oldData = oldElement->data;  
 head = head->next;  
  
 delete oldElement;  
  
 length--;  
  
 return \*this;  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::popFront() {  
 Data tmp;  
 return popFront(tmp);  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::popAt(size\_t index, Data& oldData) {  
 if (head == nullptr)  
 return \*this;  
  
 if (index >= length)  
 index = length - 1;  
  
 if (length == 1)  
 return popFront(oldData);  
  
 ListElement\* listElement = head;  
  
 for (size\_t i = 0; i < index - 1; i++)   
 listElement = listElement->next;  
  
 ListElement\* oldElement = listElement->next;  
 listElement->next = oldElement->next;  
  
 delete oldElement;  
  
 return \*this;  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::popAt(size\_t index) {  
 Data tmp;  
 return popAt(index, tmp);  
 }  
  
  
 template<typename Data>  
 void List<Data>::clear() {  
 while (!popFront().empty());  
 }  
  
 template<typename Data>  
 void List<Data>::resize(size\_t newSize) {  
 if (length > newSize) {  
 while (length != newSize) {  
 popBack();  
 }  
 }  
 else {  
 while (length != newSize)  
 {  
 pushBack(Data());  
 }  
 }  
 }  
  
 template<typename Data>  
 bool List<Data>::empty() const {  
 return length == 0;  
 }  
  
 template<typename Data>  
 size\_t List<Data>::size() const {  
 return length;  
 }  
  
 template<typename Data>  
 bool List<Data>::sorted(int(\*compareFunction)(const Data&, const Data&)) const {  
 ListElement\* listElement = head;  
 for (size\_t i = 0; i < length - 1; i++) {  
 if (compareFunction(listElement->data, listElement->next->data) > 0) {  
 return false;  
 }  
 listElement = listElement->next;  
 }  
 return true;  
 }  
  
 template<typename Data>  
 void List<Data>::display(std::ostream& out, void(\*displayFunction)(std::ostream&, const Data&)) const {  
 ListElement\* element = head;  
  
 if (length == 0) {  
 out << "<Empty list>" << std::endl;  
 }  
  
 for (size\_t i = 0; i < length && element != nullptr; i++) {  
 displayFunction(out, element->data);  
 out << std::endl;  
 element = element->next;  
 }  
 out << std::endl;  
 }  
  
  
 template<typename Data>  
 size\_t List<Data>::find(const Data& data) const {  
 ListElement\* listElement = head;  
 size\_t count = 0;  
  
 for (size\_t index = 0; index < length; index++) {  
 if (listElement->data == data)  
 count++;  
 listElement = listElement->next;  
 }  
  
 return count;  
 }  
  
 template<typename Data>  
 Data& List<Data>::min(int(\*compareFunction)(const Data&, const Data&)) const {  
 return min(compareFunction, 0, length);  
 }  
  
 template<typename Data>  
 Data& List<Data>::max(int(\*compareFunction)(const Data&, const Data&)) const {  
 return max(compareFunction, 0, length);  
 }  
  
 template<typename Data>  
 Data& List<Data>::min(int(\*compareFunction)(const Data&, const Data&), size\_t a, size\_t b) {  
 if (length == 0 || a >= b)  
 throw std::runtime\_error("Cannot find the max element of an empty list");  
  
 ListElement\* listElement = head;  
 for (size\_t i = 0; i < a; i++)  
 listElement = listElement->next;  
 Data\* min = &listElement->data;  
 listElement = listElement->next;  
  
 for (size\_t i = a + 1; i < b; i++) {  
 if (compareFunction(listElement->data, \*min) < 0)  
 min = &listElement->data;  
 listElement = listElement->next;  
 }  
  
 return \*min;  
 }  
  
 template<typename Data>  
 Data& List<Data>::max(int(\*compareFunction)(const Data&, const Data&), size\_t a, size\_t b) {  
 if (length == 0 || a >= b)  
 throw std::runtime\_error("Cannot find the max element of an empty list");  
  
 ListElement\* listElement = head;  
 for (size\_t i = 0; i < a; i++)  
 listElement = listElement->next;  
 Data\* max = &listElement->data;  
 listElement = listElement->next;  
  
 for (size\_t i = a + 1; i < b; i++) {  
 if (compareFunction(listElement->data, \*max) > 0)  
 max = &listElement->data;  
 listElement = listElement->next;  
 }  
  
 return \*max;  
 }  
  
 template<typename Data>  
 void List<Data>::swap(Data& data1, Data& data2) {  
 Data tmp = data1;  
 data1 = data2;  
 data2 = tmp;  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::sort(int(\*compareFunction)(const Data&, const Data&)) {  
 if (length == 0)  
 return \*this;  
  
 // Selection sort  
 ListElement\* listElement = head;  
  
 for (size\_t i = 0; i < length; i++) {  
 swap(listElement->data, min(compareFunction, i, length));  
 listElement = listElement->next;  
 }  
  
 return \*this;  
 }  
  
  
 template<typename Data>  
 List<Data>& List<Data>::operator<<(const Data& data) {  
 return pushBack(data);  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::operator<<(const List<Data>& list) {  
 return pushBack(list);  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::operator>>(const Data& data) {  
 return popBack(data);  
 }  
  
 template<typename Data>  
 List<Data>& List<Data>::operator=(const List<Data>& list) {  
 clear();  
  
 ListElement\* listElement = list.head;  
 for (size\_t i = 0; i < list.length; i++) {  
 pushBack(listElement->data);  
 listElement = listElement->next;  
 }  
  
 return \*this;  
 }  
  
 template<typename Data>  
 List<Data> List<Data>::operator+(const List<Data>& list) const {  
 List<Data> resultList(\*this);  
 resultList.pushBack(list);  
 return resultList;  
 }  
  
 template<typename Data>  
 Data& List<Data>::operator[](size\_t index) {  
 if (index >= length)  
 index = length - 1;  
 ListElement\* listElement = head;  
 if (cached.element == nullptr || cached.index > index) {  
 for (size\_t i = 0; i < index; i++) {  
 listElement = listElement->next;  
 }  
 }  
 else {  
 listElement = cached.element;  
 for (size\_t i = cached.index; i < index; i++) {  
 listElement = listElement->next;  
 }  
 }  
  
 cached.element = listElement;  
 cached.index = index;  
  
 return listElement->data;  
 }  
}

### graph.hpp

#pragma once  
#include "list.hpp"  
  
namespace Alg {  
 typedef List<List<int>> AdjacencyMatrix;  
 typedef List<List<int>> AdjacencyList;  
 class Graph {  
 AdjacencyList adjList;  
 AdjacencyMatrix adjMatrix;  
 bool listSet = false;  
 bool matrixSet = false;  
 int vertexNumber = 0;  
  
 static int vtoi(int v) ;  
 static int itov(int i) ;  
  
 void readAdjMatrix(std::ifstream& file);  
 void readAdjList(std::ifstream& file);  
 void writeAdjMatrix(std::ofstream& file);  
 void writeAdjList(std::ofstream& file);  
 public:  
 enum TypeCode {  
 *NONE* = -1,  
 *ADJ\_LIST* = 0,  
 *ADJ\_MATRIX* = 1  
 };  
  
 Graph() = default;  
 Graph(const std::string &filename);  
  
 TypeCode read(const std::string& filename);  
 bool write(const std::string& filename, TypeCode as = *ADJ\_LIST*);  
  
 const AdjacencyList& asAdjList();  
 const AdjacencyMatrix& asAdjMatrix();  
 };  
}

### graph.cpp

#include <fstream>  
  
#include "graph.hpp"  
  
namespace Alg {  
 Graph::Graph(const std::string& filename)  
 {  
 read(filename);  
 }  
  
 Graph::TypeCode Graph::read(const std::string& filename)  
 {  
 std::ifstream file(filename);  
 if (!file.is\_open()) {  
 return *NONE*;  
 }  
 int code = 0;  
  
 file >> code;  
 switch (code)  
 {  
 case *ADJ\_LIST*:  
 readAdjList(file);  
 return *ADJ\_LIST*;  
 case *ADJ\_MATRIX*:  
 readAdjMatrix(file);  
 return *ADJ\_MATRIX*;  
 default:  
 return *NONE*;  
 }  
 }  
  
 void Graph::readAdjMatrix(std::ifstream& file) {  
 int vertexNumber = 0;  
  
 file >> vertexNumber;  
 this->vertexNumber = vertexNumber;  
 adjMatrix.resize(vertexNumber);  
 for (int row = 0; row < vertexNumber; row++) {  
 adjMatrix[row].resize(vertexNumber);  
 for (int col = 0; col < vertexNumber; col++) {  
 int linksNumber;  
 file >> linksNumber;  
 adjMatrix[row][col] = linksNumber;  
 }  
 }  
 matrixSet = true;  
 }  
  
 void Graph::readAdjList(std::ifstream& file) {  
 int vertexNumber = 0;  
  
 file >> vertexNumber;  
 this->vertexNumber = vertexNumber;  
 adjList.resize(vertexNumber);  
 for (int i = 0; i < vertexNumber; i++) {  
 int vertex, linkVertex;  
 file >> vertex;  
 while (file.peek() != '\n' && !file.eof()) {  
 file >> linkVertex;  
 adjList[vtoi(vertex)].pushSorted(vtoi(linkVertex));  
 }  
 }  
 listSet = true;  
 }  
  
 bool Graph::write(const std::string& filename, TypeCode as)  
 {  
 std::ofstream file(filename);  
 if (!file.is\_open()) {  
 return false;  
 }  
  
 switch (as)  
 {  
 case Alg::Graph::*ADJ\_LIST*:  
 writeAdjList(file);  
 return true;  
 case Alg::Graph::*ADJ\_MATRIX*:  
 writeAdjMatrix(file);  
 return true;  
 default:  
 return false;  
 break;  
 }  
 return true;  
 }  
  
 void Graph::writeAdjMatrix(std::ofstream& file) {  
 file << *ADJ\_MATRIX* << " " << vertexNumber << std::endl;  
  
 if (!matrixSet) {  
 asAdjMatrix();  
 }  
  
 for (int row = 0; row < vertexNumber; row++) {  
 for (int col = 0; col < vertexNumber; col++) {  
 file << adjMatrix[row][col] << " ";  
 }  
 file << std::endl;  
 }  
 }  
  
 void Graph::writeAdjList(std::ofstream& file) {  
 if (!listSet) {  
 asAdjList();  
 }  
  
 file << *ADJ\_LIST* << " " << adjList.size() << std::endl;  
 for (int row = 0; row < vertexNumber; row++) {  
 file << itov(row) << " ";  
 for (int i = 0; i < adjList[row].size(); i++) {  
 file << itov(adjList[row][i]) << " ";  
 }  
 file << std::endl;  
 }  
 }  
  
 const AdjacencyList& Graph::asAdjList() {  
 if (!listSet) {  
 adjList.clear();  
 adjList.resize(adjMatrix.size());  
 for (int row = 0; row < adjMatrix.size(); row++) {  
 for (int col = 0; col < adjMatrix[row].size(); col++) {  
 size\_t linksNum = adjMatrix[row][col];  
 for (int i = 0; i < linksNum; i++) {  
 adjList[row].pushBack(col);  
 }  
 }  
 }  
 listSet = true;  
 }  
  
 return adjList;  
 }  
  
 const AdjacencyMatrix& Graph::asAdjMatrix() {  
 if (!matrixSet) {  
 adjMatrix.clear();  
 adjMatrix.resize(adjList.size());  
 for (int row = 0; row < adjList.size(); row++) {  
 adjMatrix[row].resize(adjList.size());  
 for (int col = 0; col < adjList.size(); col++) {  
 adjMatrix[row][col] = adjList[row].find(col);  
 }  
 }  
 matrixSet = true;  
 }  
  
 return adjMatrix;  
 }  
  
 inline int Graph::vtoi(int v) {  
 return v - 1;  
 }  
  
 inline int Graph::itov(int i) {  
 return i + 1;  
 }  
}

### main.cpp

#include <iostream>  
#include "graph.hpp"  
  
void graph\_menu() {  
 Alg::Graph graph;  
 std::string inputFile, outputFile;  
 const std::string srcFolder = "graphs/";  
 Alg::Graph::TypeCode saveType;  
  
 std::cout << "Enter the path to the file" << std::endl;  
 std::cout << "> ";  
 std::cin >> inputFile;  
  
 switch (graph.read(srcFolder + inputFile)) {  
 case Alg::Graph::*NONE*:  
 std::cout << "File doesn't exist..." << std::endl;  
 return;  
 case Alg::Graph::*ADJ\_MATRIX*:  
 std::cout << "Graph was read as adjacency matrix and will be saved as adjacency list" << std::endl;  
 saveType = Alg::Graph::*ADJ\_LIST*;  
 break;  
 case Alg::Graph::*ADJ\_LIST*:  
 std::cout << "Graph was read as adjacency list and will be saved as adjacency matrix" << std::endl;  
 saveType = Alg::Graph::*ADJ\_MATRIX*;  
 break;  
 }  
  
 std::cout << "Enter the path to the output file" << std::endl;  
 std::cout << "> ";  
 std::cin >> outputFile;  
 std::cout << "Writing the graph into the file..." << std::endl;  
 if (graph.write(srcFolder + outputFile, saveType)) {  
 std::cout << "Graph has been saved in " << outputFile << std::endl;  
 } else {  
 std::cout << "Some error has happened. Please, try again" << std::endl;  
 }  
}  
  
void menu() {  
 std::cout << "begin - start converting" << std::endl;  
 std::cout << "help - list all commands" << std::endl;  
 std::cout << "exit - exit program" << std::endl;  
}  
  
void start() {  
 std::string response;  
 bool open = true;  
  
 menu();  
 while (open) {  
 std::cout << "> ";  
 std::cin >> response;  
  
 if (response == "begin") {  
 graph\_menu();  
 } else if (response == "exit") {  
 open = false;  
 } else if (response == "help") {  
 menu();  
 } else {  
 std::cout << "Wrong input, please try again (or use 'help' command)" << std::endl;  
 }  
 }  
}  
  
int main() {  
 start();  
 return 0;  
}

# Описание тестирования

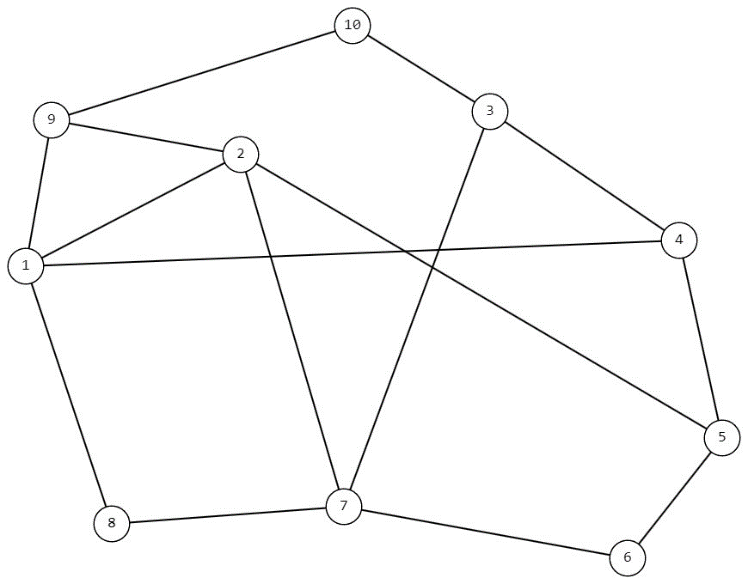
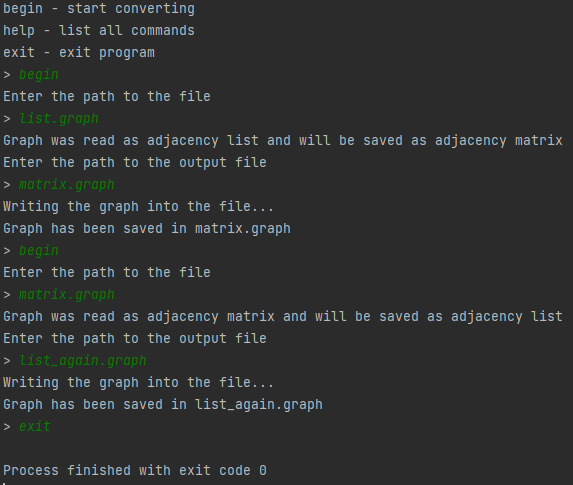
Для проверки корректности работы программы были построены графы с помощью [сайта](https://programforyou.ru/graph-redactor), чтобы была возможность визуального оценивания решения (да и вообще так проще нормальный граф строить…).

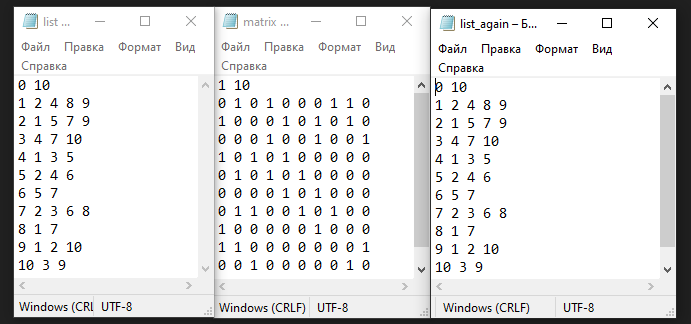
Проверялось:

* Стабильность работы программы при одинаковых входных данных;
* Отсутствие “падений” и “зависаний”;
* Корректное выполнение всех заявленных процедур;
* Корректное завершение программы;

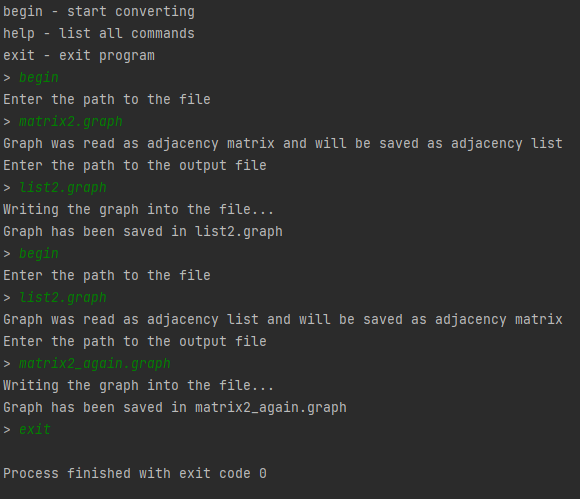
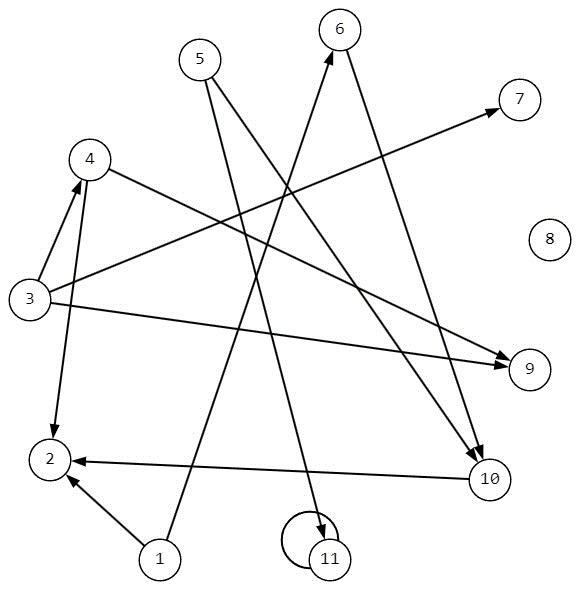
Приведем пример тестирования на скриншотах, данных ниже

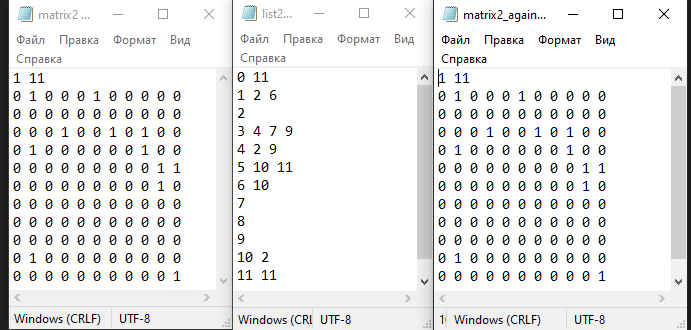
Тест 1.





Тест 2.





При проведении тестирования такого рода никаких проблем обнаружено не было, что позволяет судить о корректности работы программы в целом.