Практическая работа № 12

НАХОЖДЕНИЕ КРАТЧАЙШИХ ПУТЕЙ В ГРАФЕ

Вариант-12.2.3.2

Постановка задачи

Составить программу нахождения кратчайших путей в графе алгоритмом Йена.

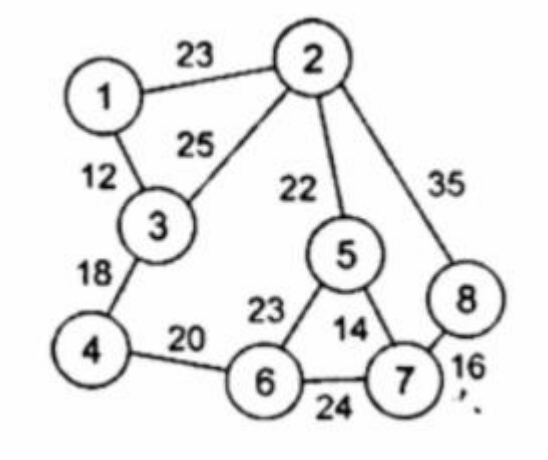
Выбрать и реализовать способ представления графа в памяти.

Предусмотреть ввод с клавиатуры произвольного графа.

Разработать доступный способ (форму) вывода результирующего дерева на экран монитора.

Провести тестовый прогон программы для заданного графа.

Индивидуальное задание:



1. Описание алгоритма

Алгоритм Йена вычисляет K -самые короткие пути без петель с одним источником для графа с неотрицательной стоимостью ребра. Алгоритм был опубликован Jin Y. Yen в 1971 году и использует любой алгоритм кратчайшего пути для поиска наилучшего пути, а затем переходит к поиску K - 1 отклонений от наилучшего пути. Алгоритм может быть разбит на две части, определение первого к-кратчайший путь, и затем определяют все другие K - кратчайшие пути. Предполагается, что контейнер будет содержать k- самый короткий путь, тогда как контейнер будет содержать потенциальные k- самый короткий путь. Чтобы определить, в кратчайшем пути от источника к

раковине, любой эффективный алгоритм кратчайшего пути может быть использован. Чтобы найти, где находится в диапазоне от до, алгоритм предполагает, что все пути от до были ранее найдены. Итерации могут быть разделены на два процесса, находя все отклонения и выбрать минимальную длину пути, чтобы стать. Обратите внимание, что в этой итерации диапазон от до. Первый процесс можно подразделить на три операции: выбор, поиск и добавление в контейнер. Корневой путь выбирается путем нахождения подпути в том, что следует за первыми узлами, где находится в диапазоне от до. Затем, если найден путь, стоимость края от устанавливается на бесконечность. Затем, ответвительный путь находится путем вычисления кратчайшего пути от ответвительного узла, узла, до приемника. Удаление ранее использовавшихся кромок от до гарантирует, что путь ответвления будет другим. , добавление корневого пути и ответвления добавляется к . Затем ребра, которые были удалены, т.е. для которых была установлена бесконечная стоимость, восстанавливаются до своих исходных значений. Второй процесс определяет подходящий путь, находя путь в контейнере с наименьшими затратами. Этот путь удаляется из контейнера и вставляется в контейнер, и алгоритм переходит к следующей итерации.

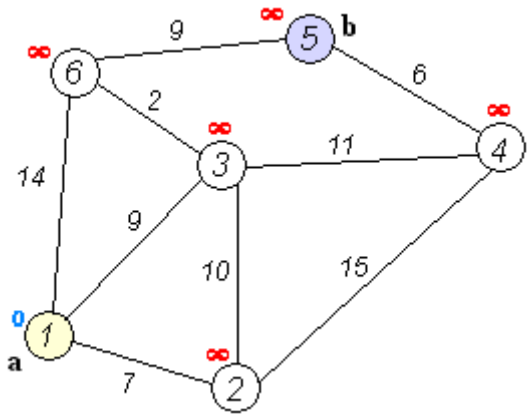


Рис.1 Поиск кратчайшего пути алгоритмом Дейкстры

Функция main вызывает меню.

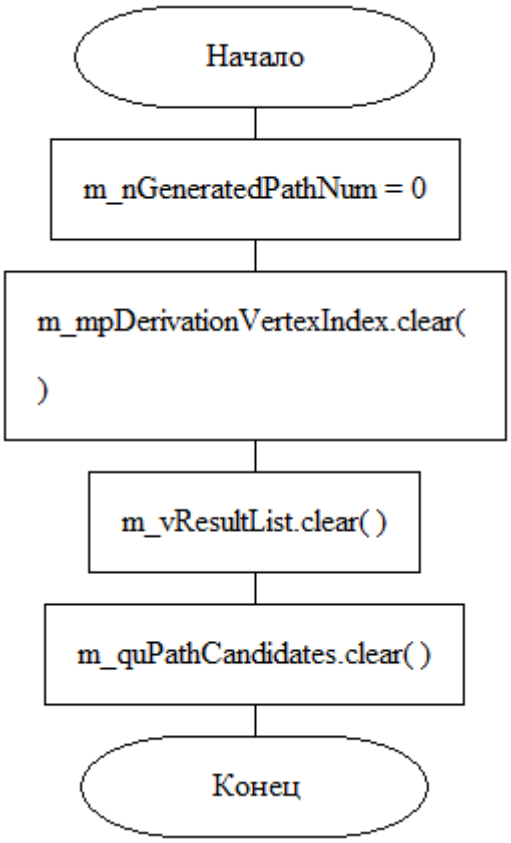


Рис.2 Схема алгоритма функции clear

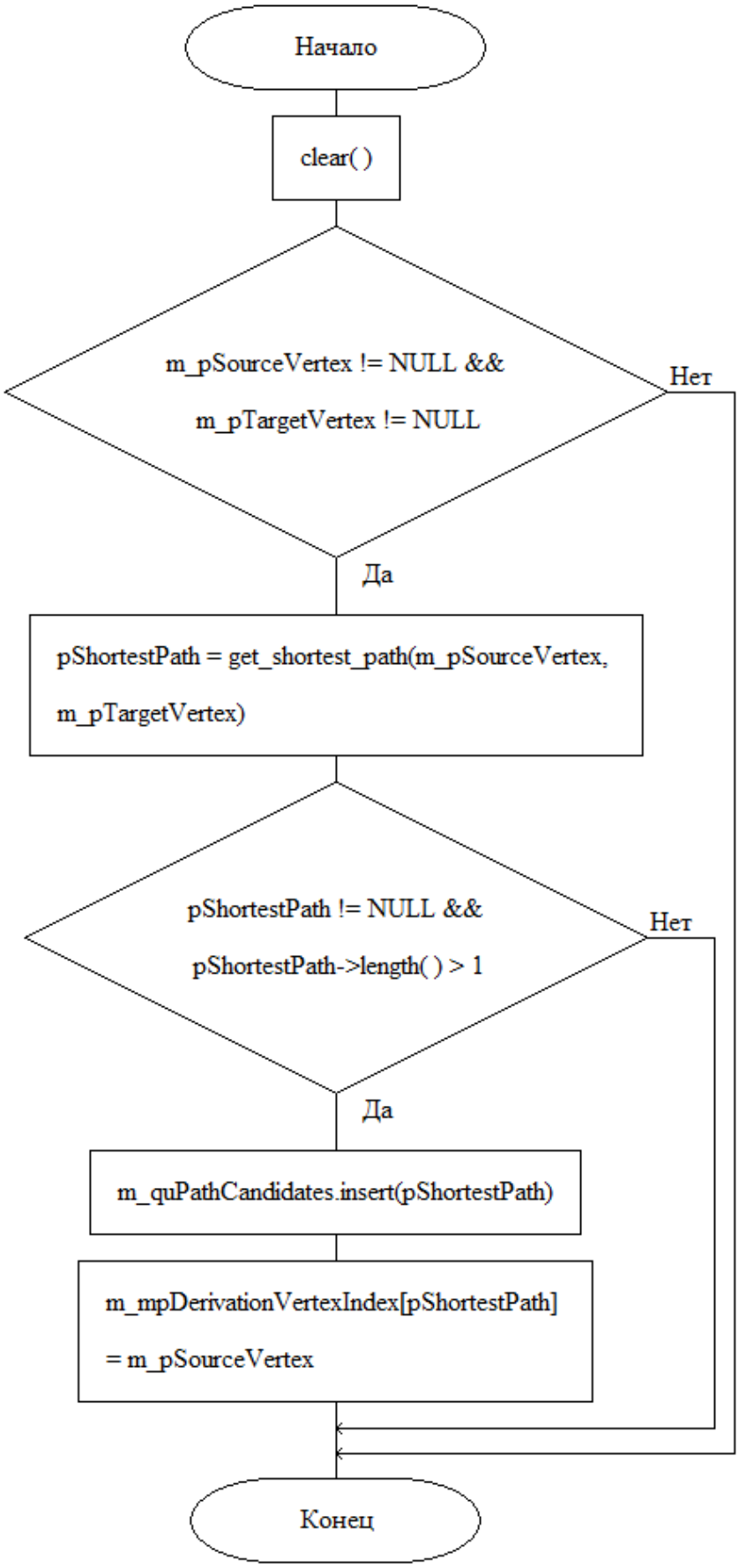


Рис.3 Схема алгоритма функции init

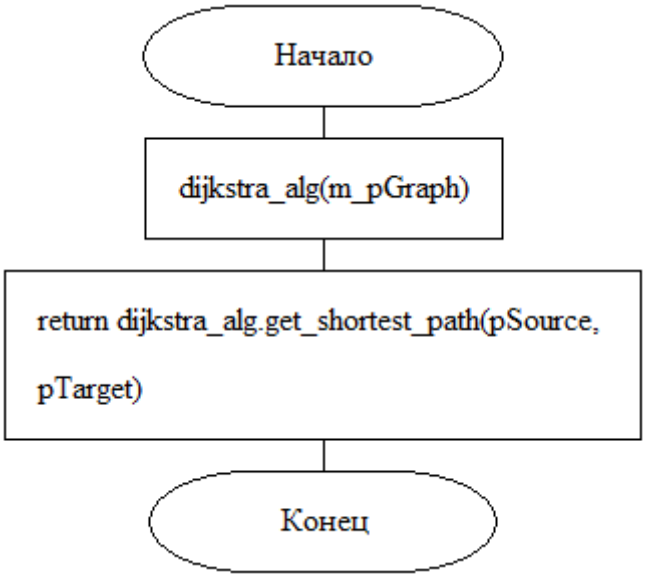


Рис.4 Схема алгоритма функции get_shortest_path

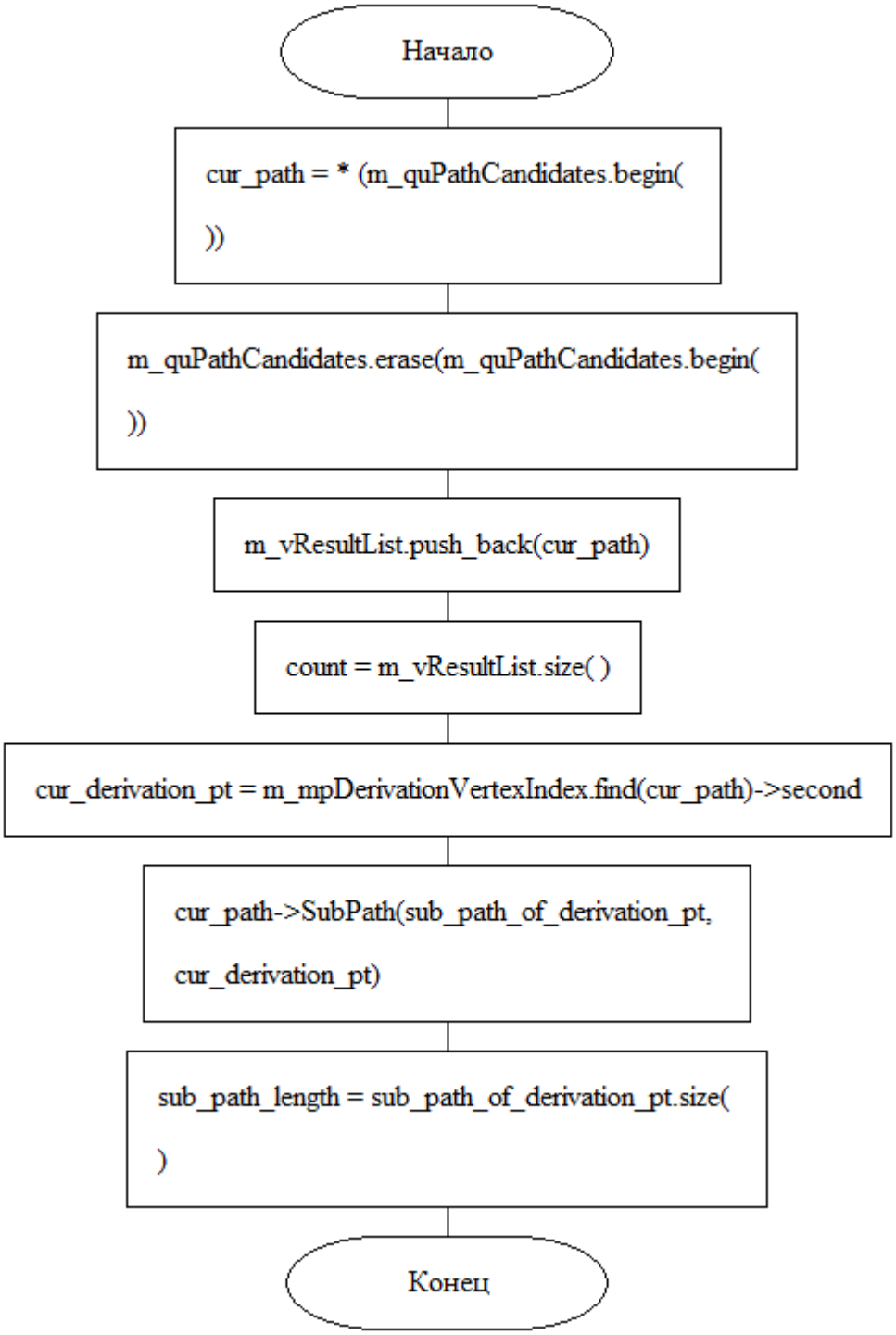


Рис.5 Схема алгоритма функции next

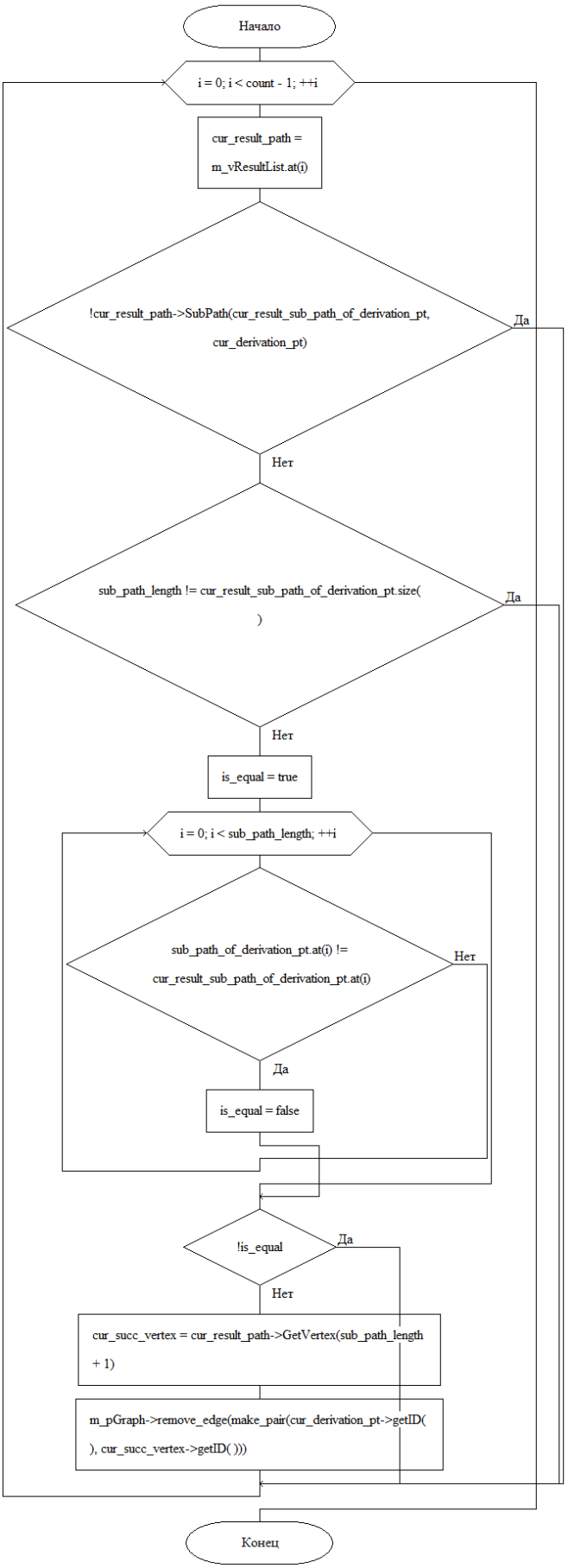


Рис.6 Схема алгоритма функции поиска следующего пути

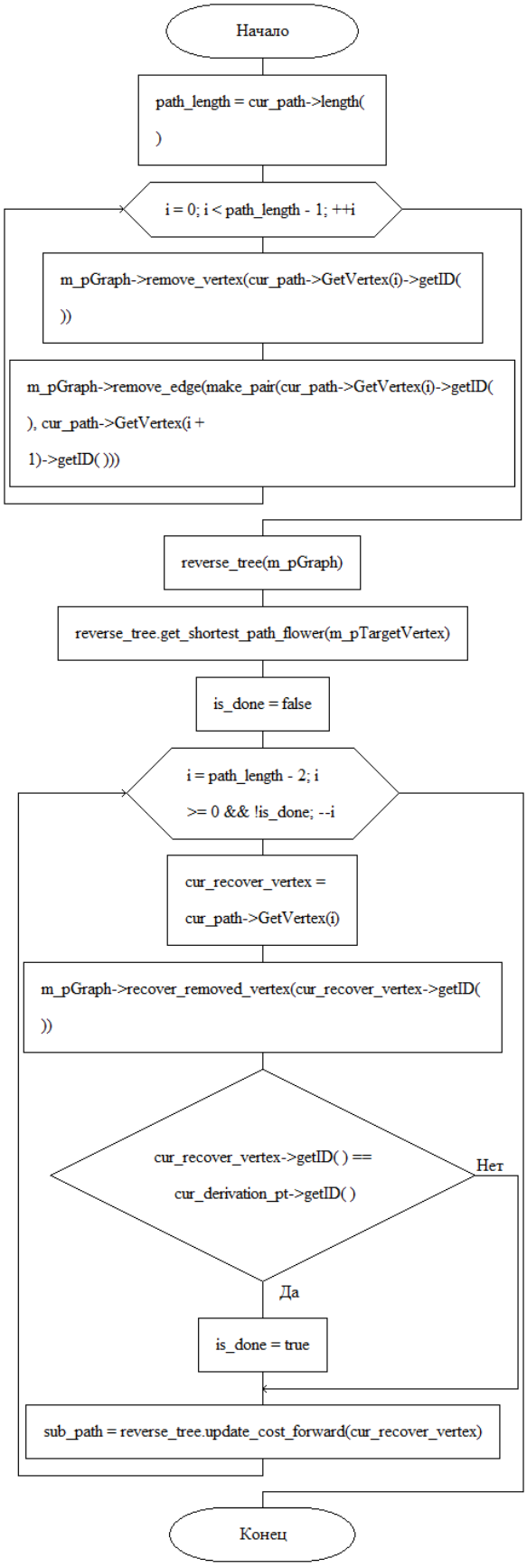


Рис.5 Схема алгоритма функции удаления векторов

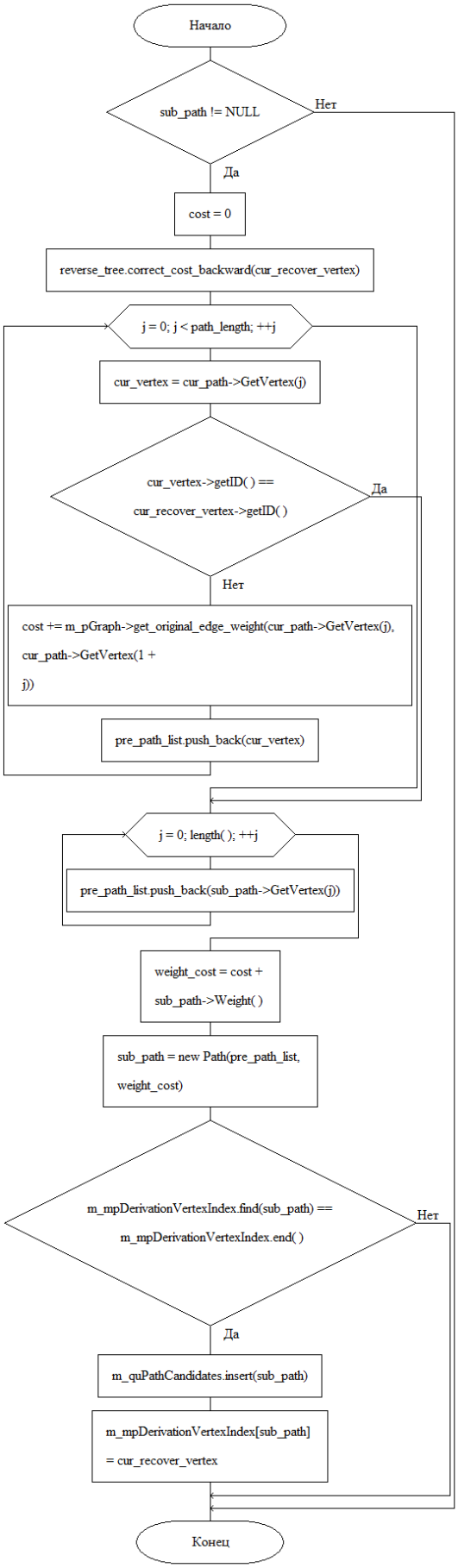


Рис.6 Схема алгоритма функции удаления векторов

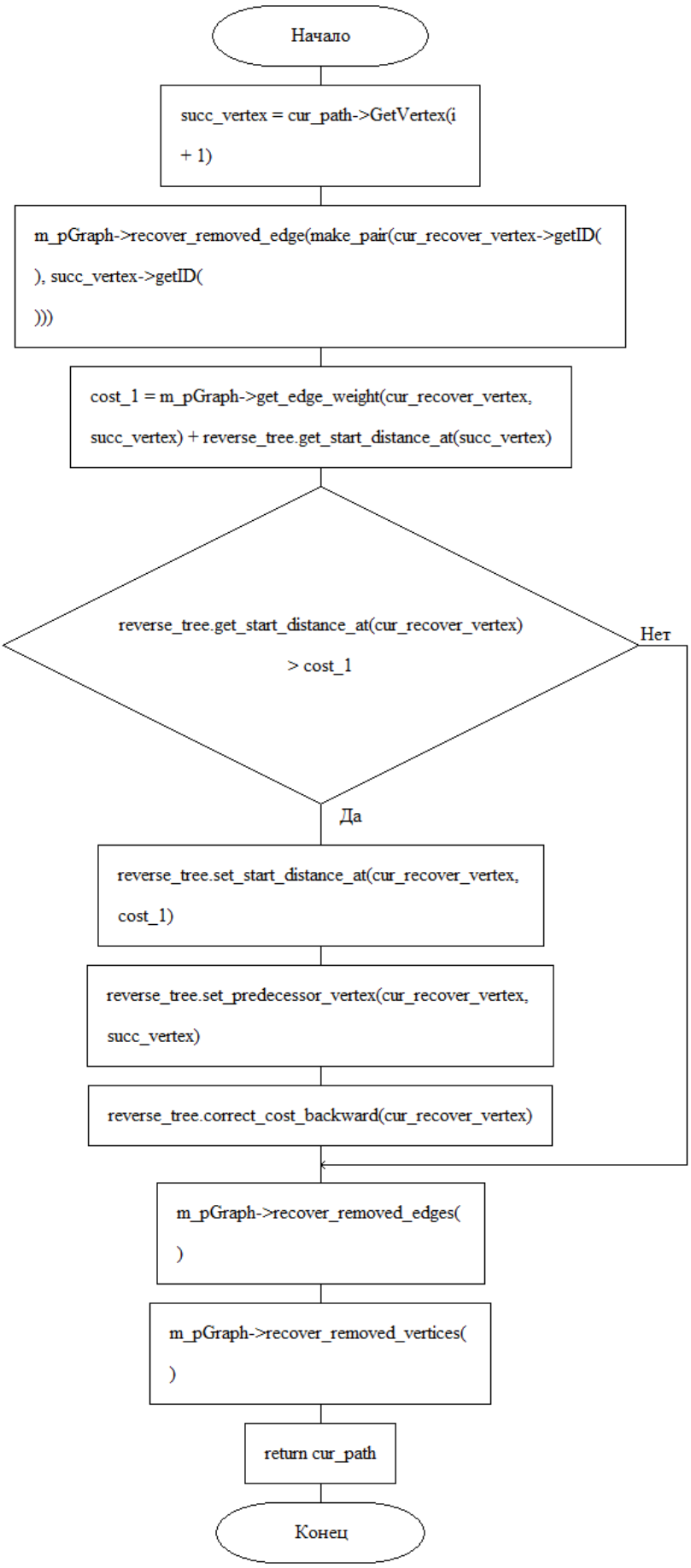


Рис.7 Схема алгоритма функции разворота дерева

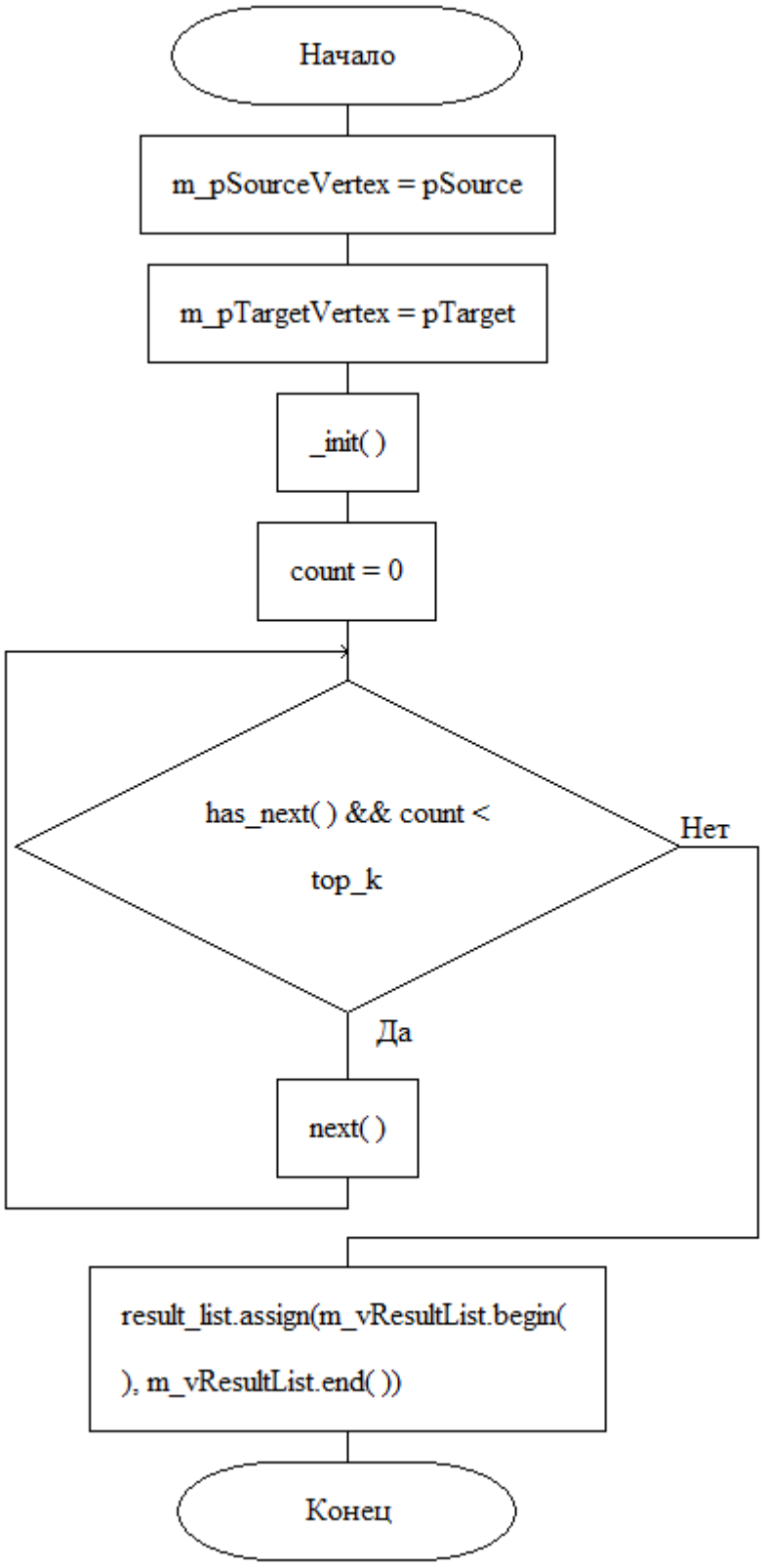


Рис.8 Схема алгоритма функции get_shortest_paths

Реализация алгоритма

Текст исходного кода программы

main.cpp

```
#include <limits>
#include <set>
#include <map>
#include <queue>
#include <string>
#include <vector>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <algorithm>
#include "GraphElements.h"
#include "Graph.h"
#include "Yen.h"

using namespace std;

void testYenAlg()
{
	Graph my_graph("yen_8.txt");

	Yen yenAlg(my_graph, my_graph.get_vertex(0),
		my_graph.get_vertex(7));

	int i = 0;
	while (yenAlg.has_next())
	{
		++i;
		yenAlg.next()->PrintOut(cout);
	}
}

void menu() {
	int choice = 0, count = 0, heights = 0;
	int v1 = 0, v2 = 0, weigth = 0;
	bool flag = true;
	string line = "", str = "";
	while (flag == true) {
		cout << "\n[1] - Добавить элемент в граф\n[2] - Вывод графа в виде
таблицы\n[3] - Вывод K-кратчайших путей во взвешенном графе\n[4] - Выход из
программы\n\nВведите номер команды: ";
		cin >> choice;
		if (choice == 1) {
			std::ofstream f;
			f.open("yen_8.txt", std::ios::app);

				cout << endl;
				cout << "Введите номер начальной вершины: ";
				cin >> v1;
				cout << "Введите номер конечной вершины: ";
				cin >> v2;
				cout << "Введите вес ребра между вершинами: ";
				cin >> weigth;
```

```
                        f << endl << v1 << " " << v2 << "  " << weigth << "     ";

                f.close();
                cout << endl;
        }
        else if (choice == 2) {
                cout << endl;
                ifstream fin;
                fin.open("yen_8.txt");
                while (!fin.eof()) {
                        if (count == 0) {
                                cout << "Кол-во вершин в графе:";
                                count++;
                        }
                        else if (count == 1) {
                                cout << "_____________________" << endl;
                                cout << "|начало| конец |вес|" << endl;
                                count++;
                        }
                        getline(fin, str);
                        cout << "|" << str << " |" << endl;
                }
                cout << "--------------------" << endl;
                fin.close();
                cout << endl;
        }
        else if (choice == 3) {
                testYenAlg();
                cout << endl;
        }
        else if (choice == 4) {
                cout << endl;
                cout << "Подтвердите выход\n1 - Да | 2 - Нет\nВведите выбранную
команду: ";
                cin >> choice;
                if (choice == 1)
                {
                        cout << endl;
                        flag = false;
                }
                else if (choice == 2)
                {
                        flag = true;
                        cout << endl;
                }
        }
    }
}

int main()
{
    setlocale(0, "");

    menu();
}
```

Graph.h

```
#pragma once

using namespace std;

class Path : public BasePath
{
public:
```

```
	Path(const std::vector<BaseVertex*>& vertex_list, double weight)
:BasePath(vertex_list, weight) {}


	void PrintOut(std::ostream& out_stream) const
	{
		out_stream << "w:" << m_dWeight << "d:" << m_vtVertexList.size() <<
std::endl;
		for (std::vector<BaseVertex*>::const_iterator pos = m_vtVertexList.begin();
pos != m_vtVertexList.end(); ++pos)
		{
			out_stream << (*pos)->getID() << " ";
		}
		out_stream << std::endl <<
"_____" << std::endl;
	}
};

class Graph
{
public:

	const static double DISCONNECT;

	typedef set<BaseVertex*>::iterator VertexPtSetIterator;
	typedef map<BaseVertex*, set<BaseVertex*>*>::iterator BaseVertexPt2SetMapIterator;

protected:


	map<BaseVertex*, set<BaseVertex*>*> m_mpFanoutVertices;
	map<BaseVertex*, set<BaseVertex*>*> m_mpFaninVertices;
	map<int, double> m_mpEdgeCodeWeight;
	vector<BaseVertex*> m_vtVertices;
	int m_nEdgeNum;
	int m_nVertexNum;

	map<int, BaseVertex*> m_mpVertexIndex;


	set<int> m_stRemovedVertexIds;
	set<pair<int, int> > m_stRemovedEdge;

public:


	Graph(const string& file_name);
	Graph(const Graph& rGraph);
	~Graph(void);

	void clear();

	BaseVertex* get_vertex(int node_id);

	int get_edge_code(const BaseVertex* start_vertex_pt, const BaseVertex*
end_vertex_pt) const;
	set<BaseVertex*>* get_vertex_set_pt(BaseVertex* vertex_, map<BaseVertex*,
set<BaseVertex*>*>& vertex_container_index);

	double get_original_edge_weight(const BaseVertex* source, const BaseVertex* sink);

	double get_edge_weight(const BaseVertex* source, const BaseVertex* sink);
	void get_adjacent_vertices(BaseVertex* vertex, set<BaseVertex*>& vertex_set);
	void get_precedent_vertices(BaseVertex* vertex, set<BaseVertex*>& vertex_set);
```

```
        void remove_edge(const pair<int, int> edge)
        {
                m_stRemovedEdge.insert(edge);
        }

        void remove_vertex(const int vertex_id)
        {
                m_stRemovedVertexIds.insert(vertex_id);
        }

        void recover_removed_edges()
        {
                m_stRemovedEdge.clear();
        }

        void recover_removed_vertices()
        {
                m_stRemovedVertexIds.clear();
        }

        void recover_removed_edge(const pair<int, int> edge)
        {
                m_stRemovedEdge.erase(m_stRemovedEdge.find(edge));
        }

        void recover_removed_vertex(int vertex_id)
        {
                m_stRemovedVertexIds.erase(m_stRemovedVertexIds.find(vertex_id));
        }

private:
        void _import_from_file(const std::string& file_name);

};
```

Graph.cpp

```
#include <limits>
#include <set>
#include <map>
#include <string>
#include <vector>
#include <fstream>
#include <iostream>
#include <algorithm>
#include "GraphElements.h"
#include "Graph.h"


const double Graph::DISCONNECT = (numeric_limits<double>::max)();


Graph::Graph(const string& file_name)
{
        _import_from_file(file_name);
}

Graph::Graph(const Graph& graph)
{
        m_nVertexNum = graph.m_nVertexNum;
        m_nEdgeNum = graph.m_nEdgeNum;
        m_vtVertices.assign(graph.m_vtVertices.begin(), graph.m_vtVertices.end());
```

```
	m_mpFaninVertices.insert(graph.m_mpFaninVertices.begin(),
graph.m_mpFaninVertices.end());
	m_mpFanoutVertices.insert(graph.m_mpFanoutVertices.begin(),
graph.m_mpFanoutVertices.end());
	m_mpEdgeCodeWeight.insert(graph.m_mpEdgeCodeWeight.begin(),
graph.m_mpEdgeCodeWeight.end());
	m_mpVertexIndex.insert(graph.m_mpVertexIndex.begin(),
graph.m_mpVertexIndex.end());
}

Graph::~Graph(void)
{
	clear();
}


void Graph::_import_from_file(const string& input_file_name)
{
	const char* file_name = input_file_name.c_str();


	ifstream ifs(file_name);
	if (!ifs)
	{
		cerr << "Graph.cpp1 " << file_name << " Graph.cpp1!!!" << endl;
		exit(1);
	}

	clear();


	ifs >> m_nVertexNum;

	int start_vertex, end_vertex;
	double edge_weight;
	int vertex_id = 0;

	while (ifs >> start_vertex)
	{
		if (start_vertex == -1)
		{
			break;
		}
		ifs >> end_vertex;
		ifs >> edge_weight;


		BaseVertex* start_vertex_pt = get_vertex(start_vertex);
		BaseVertex* end_vertex_pt = get_vertex(end_vertex);


		m_mpEdgeCodeWeight[get_edge_code(start_vertex_pt, end_vertex_pt)] =
edge_weight;


		get_vertex_set_pt(end_vertex_pt, m_mpFaninVertices)-
>insert(start_vertex_pt);


		get_vertex_set_pt(start_vertex_pt, m_mpFanoutVertices)-
>insert(end_vertex_pt);

	}

	if (m_nVertexNum != m_vtVertices.size())
```

```
	{
		cerr << "Graph.cpp: " << m_vtVertices.size() << "Graph.cpp:" <<
m_nVertexNum << endl;
		exit(1);
	}

	m_nVertexNum = m_vtVertices.size();
	m_nEdgeNum = m_mpEdgeCodeWeight.size();

	ifs.close();
}

BaseVertex* Graph::get_vertex(int node_id)
{
	if (m_stRemovedVertexIds.find(node_id) != m_stRemovedVertexIds.end())
	{
		return NULL;
	}
	else
	{
		BaseVertex* vertex_pt = NULL;
		const map<int, BaseVertex*>::iterator pos = m_mpVertexIndex.find(node_id);
		if (pos == m_mpVertexIndex.end())
		{
			int vertex_id = m_vtVertices.size();
			vertex_pt = new BaseVertex();
			vertex_pt->setID(node_id);
			m_mpVertexIndex[node_id] = vertex_pt;

			m_vtVertices.push_back(vertex_pt);
		}
		else
		{
			vertex_pt = pos->second;
		}

		return vertex_pt;
	}
}

void Graph::clear()
{
	m_nEdgeNum = 0;
	m_nVertexNum = 0;

	for (map<BaseVertex*, set<BaseVertex*>*>::const_iterator pos =
m_mpFaninVertices.begin();
		pos != m_mpFaninVertices.end(); ++pos)
	{
		delete pos->second;
	}
	m_mpFaninVertices.clear();

	for (map<BaseVertex*, set<BaseVertex*>*>::const_iterator pos =
m_mpFanoutVertices.begin();
		pos != m_mpFanoutVertices.end(); ++pos)
	{
		delete pos->second;
	}
	m_mpFanoutVertices.clear();


	m_mpEdgeCodeWeight.clear();
```

```
	for_each(m_vtVertices.begin(), m_vtVertices.end(), DeleteFunc<BaseVertex>());
	m_vtVertices.clear();
	m_mpVertexIndex.clear();

	m_stRemovedVertexIds.clear();
	m_stRemovedEdge.clear();
}

int Graph::get_edge_code(const BaseVertex* start_vertex_pt, const BaseVertex*
end_vertex_pt) const
{

	return start_vertex_pt->getID() * m_nVertexNum + end_vertex_pt->getID();
}


set<BaseVertex*>* Graph::get_vertex_set_pt(BaseVertex* vertex_, map<BaseVertex*,
set<BaseVertex*>*>& vertex_container_index)
{
	BaseVertexPt2SetMapIterator pos = vertex_container_index.find(vertex_);

	if (pos == vertex_container_index.end())
	{
		set<BaseVertex*>* vertex_set = new set<BaseVertex*>();
		pair<BaseVertexPt2SetMapIterator, bool> ins_pos =
			vertex_container_index.insert(make_pair(vertex_, vertex_set));

		pos = ins_pos.first;
	}

	return pos->second;
}


double Graph::get_edge_weight(const BaseVertex* source, const BaseVertex* sink)
{
	int source_id = source->getID();
	int sink_id = sink->getID();

	if (m_stRemovedVertexIds.find(source_id) != m_stRemovedVertexIds.end()
		|| m_stRemovedVertexIds.find(sink_id) != m_stRemovedVertexIds.end()
		|| m_stRemovedEdge.find(make_pair(source_id, sink_id)) !=
m_stRemovedEdge.end())
	{
		return DISCONNECT;
	}
	else
	{
		return get_original_edge_weight(source, sink);
	}
}


void Graph::get_adjacent_vertices(BaseVertex* vertex, set<BaseVertex*>& vertex_set)
{
	int starting_vt_id = vertex->getID();

	if (m_stRemovedVertexIds.find(starting_vt_id) == m_stRemovedVertexIds.end())
	{
		set<BaseVertex*>* vertex_pt_set = get_vertex_set_pt(vertex,
m_mpFanoutVertices);
		for (set<BaseVertex*>::const_iterator pos = (*vertex_pt_set).begin();
			pos != (*vertex_pt_set).end(); ++pos)
		{
			int ending_vt_id = (*pos)->getID();
```

```
                        if (m_stRemovedVertexIds.find(ending_vt_id) !=
m_stRemovedVertexIds.end()
                                || m_stRemovedEdge.find(make_pair(starting_vt_id,
ending_vt_id)) != m_stRemovedEdge.end())
                        {
                                continue;
                        }

                        vertex_set.insert(*pos);
                }
        }
}

void Graph::get_precedent_vertices(BaseVertex* vertex, set<BaseVertex*>& vertex_set)
{
        if (m_stRemovedVertexIds.find(vertex->getID()) == m_stRemovedVertexIds.end())
        {
                int ending_vt_id = vertex->getID();
                set<BaseVertex*>* pre_vertex_set = get_vertex_set_pt(vertex,
m_mpFaninVertices);
                for (set<BaseVertex*>::const_iterator pos = (*pre_vertex_set).begin();
                        pos != (*pre_vertex_set).end(); ++pos)
                {
                        int starting_vt_id = (*pos)->getID();
                        if (m_stRemovedVertexIds.find(starting_vt_id) !=
m_stRemovedVertexIds.end()
                                || m_stRemovedEdge.find(make_pair(starting_vt_id,
ending_vt_id)) != m_stRemovedEdge.end())
                        {
                                continue;
                        }

                        vertex_set.insert(*pos);
                }
        }
}

double Graph::get_original_edge_weight(const BaseVertex* source, const BaseVertex* sink)
{
        map<int, double>::const_iterator pos =
                m_mpEdgeCodeWeight.find(get_edge_code(source, sink));

        if (pos != m_mpEdgeCodeWeight.end())
        {
                return pos->second;
        }
        else
        {
                return DISCONNECT;
        }
}
```

Yen.h

```
#pragma once

using namespace std;

class Yen
{
        Graph* m_pGraph;
```

```
	vector<BasePath*> m_vResultList;
	map<BasePath*, BaseVertex*> m_mpDerivationVertexIndex;
	multiset<BasePath*, WeightLess<BasePath> > m_quPathCandidates;

	BaseVertex* m_pSourceVertex;
	BaseVertex* m_pTargetVertex;

	int m_nGeneratedPathNum;

private:

	void _init();

public:

	Yen(const Graph& graph)
	{
		Yen(graph, NULL, NULL);
	}

	Yen(const Graph& graph, BaseVertex* pSource, BaseVertex* pTarget)
		:m_pSourceVertex(pSource), m_pTargetVertex(pTarget)
	{
		m_pGraph = new Graph(graph);
		_init();
	}

	~Yen(void) { clear(); }

	void clear();
	bool has_next();
	BasePath* next();

	BasePath* get_shortest_path(BaseVertex* pSource, BaseVertex* pTarget);
	void get_shortest_paths(BaseVertex* pSource, BaseVertex* pTarget, int top_k,
		vector<BasePath*>&);
	};
```

Yen.cpp

```
#include <set>
#include <map>
#include <queue>
#include <vector>
#include "GraphElements.h"
#include "Graph.h"
#include "Dx.h"
#include "Yen.h"

using namespace std;

void Yen::clear()
{
	m_nGeneratedPathNum = 0;
	m_mpDerivationVertexIndex.clear();
	m_vResultList.clear();
	m_quPathCandidates.clear();
}

void Yen::_init()
{
	clear();
```

```
	if (m_pSourceVertex != NULL && m_pTargetVertex != NULL)
	{
		BasePath* pShortestPath = get_shortest_path(m_pSourceVertex,
m_pTargetVertex);
		if (pShortestPath != NULL && pShortestPath->length() > 1)
		{
			m_quPathCandidates.insert(pShortestPath);
			m_mpDerivationVertexIndex[pShortestPath] = m_pSourceVertex;
		}
	}
}

BasePath* Yen::get_shortest_path(BaseVertex* pSource, BaseVertex* pTarget)
{
	Dx dijkstra_alg(m_pGraph);
	return dijkstra_alg.get_shortest_path(pSource, pTarget);
}

bool Yen::has_next()
{
	return !m_quPathCandidates.empty();
}

BasePath* Yen::next()
{

	BasePath* cur_path = *(m_quPathCandidates.begin());


	m_quPathCandidates.erase(m_quPathCandidates.begin());
	m_vResultList.push_back(cur_path);

	int count = m_vResultList.size();

	BaseVertex* cur_derivation_pt = m_mpDerivationVertexIndex.find(cur_path)->second;
	vector<BaseVertex*> sub_path_of_derivation_pt;
	cur_path->SubPath(sub_path_of_derivation_pt, cur_derivation_pt);
	int sub_path_length = sub_path_of_derivation_pt.size();


	for (int i = 0; i < count - 1; ++i)
	{
		BasePath* cur_result_path = m_vResultList.at(i);
		vector<BaseVertex*> cur_result_sub_path_of_derivation_pt;

		if (!cur_result_path->SubPath(cur_result_sub_path_of_derivation_pt,
cur_derivation_pt)) continue;

		if (sub_path_length != cur_result_sub_path_of_derivation_pt.size())
continue;

		bool is_equal = true;
		for (int i = 0; i < sub_path_length; ++i)
		{
			if (sub_path_of_derivation_pt.at(i) !=
cur_result_sub_path_of_derivation_pt.at(i))
			{
				is_equal = false;
				break;
			}
		}
		if (!is_equal) continue;
```

```
			BaseVertex* cur_succ_vertex = cur_result_path->GetVertex(sub_path_length +
1);
			m_pGraph->remove_edge(make_pair(cur_derivation_pt->getID(),
cur_succ_vertex->getID()));
		}


		int path_length = cur_path->length();
		for (int i = 0; i < path_length - 1; ++i)
		{
			m_pGraph->remove_vertex(cur_path->GetVertex(i)->getID());
			m_pGraph->remove_edge(make_pair(
				cur_path->GetVertex(i)->getID(), cur_path->GetVertex(i + 1)-
>getID()));
		}


		Dx reverse_tree(m_pGraph);
		reverse_tree.get_shortest_path_flower(m_pTargetVertex);


		bool is_done = false;
		for (int i = path_length - 2; i >= 0 && !is_done; --i)
		{

			BaseVertex* cur_recover_vertex = cur_path->GetVertex(i);
			m_pGraph->recover_removed_vertex(cur_recover_vertex->getID());


			if (cur_recover_vertex->getID() == cur_derivation_pt->getID())
			{
				is_done = true;
			}


			BasePath* sub_path = reverse_tree.update_cost_forward(cur_recover_vertex);


			if (sub_path != NULL)
			{
				++m_nGeneratedPathNum;

				double cost = 0;
				reverse_tree.correct_cost_backward(cur_recover_vertex);

				vector<BaseVertex*> pre_path_list;
				for (int j = 0; j < path_length; ++j)
				{
					BaseVertex* cur_vertex = cur_path->GetVertex(j);
					if (cur_vertex->getID() == cur_recover_vertex->getID())
					{

						break;
					}
					else
					{
						cost += m_pGraph->get_original_edge_weight(
							cur_path->GetVertex(j), cur_path->GetVertex(1 +
j));

						pre_path_list.push_back(cur_vertex);
					}
				}

				for (int j = 0; j < sub_path->length(); ++j)
				{
```

```
                        pre_path_list.push_back(sub_path->GetVertex(j));
                }


                double weight_cost = cost + sub_path->Weight();
                delete sub_path;
                sub_path = new Path(pre_path_list, weight_cost);


                if (m_mpDerivationVertexIndex.find(sub_path) ==
m_mpDerivationVertexIndex.end())
                {
                        m_quPathCandidates.insert(sub_path);
                        m_mpDerivationVertexIndex[sub_path] = cur_recover_vertex;
                }
        }


        BaseVertex* succ_vertex = cur_path->GetVertex(i + 1);
        m_pGraph->recover_removed_edge(make_pair(cur_recover_vertex->getID(),
succ_vertex->getID()));


        double cost_1 = m_pGraph->get_edge_weight(cur_recover_vertex, succ_vertex)
                + reverse_tree.get_start_distance_at(succ_vertex);

        if (reverse_tree.get_start_distance_at(cur_recover_vertex) > cost_1)
        {
                reverse_tree.set_start_distance_at(cur_recover_vertex, cost_1);
                reverse_tree.set_predecessor_vertex(cur_recover_vertex, succ_vertex);
                reverse_tree.correct_cost_backward(cur_recover_vertex);
        }
    }


    m_pGraph->recover_removed_edges();
    m_pGraph->recover_removed_vertices();

    return cur_path;
}

void Yen::get_shortest_paths(BaseVertex* pSource,
    BaseVertex* pTarget, int top_k, vector<BasePath*>& result_list)
{
    m_pSourceVertex = pSource;
    m_pTargetVertex = pTarget;

    _init();
    int count = 0;
    while (has_next() && count < top_k)
    {
        next();
        ++count;
    }

    result_list.assign(m_vResultList.begin(), m_vResultList.end());
    }
```

2. Тестирование программы

```
[1] - Добавить элемент в граф
[2] - Вывод графа в виде таблицы
[3] - Вывод K-кратчайших путей во взвешенном графе
[4] - Выход из программы

Введите номер команды: 2

Кол-во вершин в графе:|8 |
_____________________
|начало| конец |вес|
|0      1       23 |
|0      2       12 |
|1      2       24 |
|1      4       22 |
|1      7       35 |
|2      3       18 |
|3      4       20 |
|4      5       23 |
|4      6       14 |
|5      6       24 |
|6      7       16 |
--------------------
```

Рис.9 Скриншот вывода графа в виде таблицы

```
[1] - Добавить элемент в граф
[2] - Вывод графа в виде таблицы
[3] - Вывод K-кратчайших путей во взвешенном графе
[4] - Выход из программы

Введите номер команды: 1

Введите номер начальной вершины: 10
Введите номер конечной вершины: 20
Введите вес ребра между вершинами: 99


[1] - Добавить элемент в граф
[2] - Вывод графа в виде таблицы
[3] - Вывод K-кратчайших путей во взвешенном графе
[4] - Выход из программы

Введите номер команды: 2

Кол-во вершин в графе:|8 |
_____________________
|начало| конец |вес|
|0      1       23 |
|0      2       12 |
|1      2       24 |
|1      4       22 |
|1      7       35 |
|2      3       18 |
|3      4       20 |
|4      5       23 |
|4      6       14 |
|5      6       24 |
|6      7       16 |
|10     20      99       |
--------------------
```

Рис.9 Скриншот добавления связи в граф

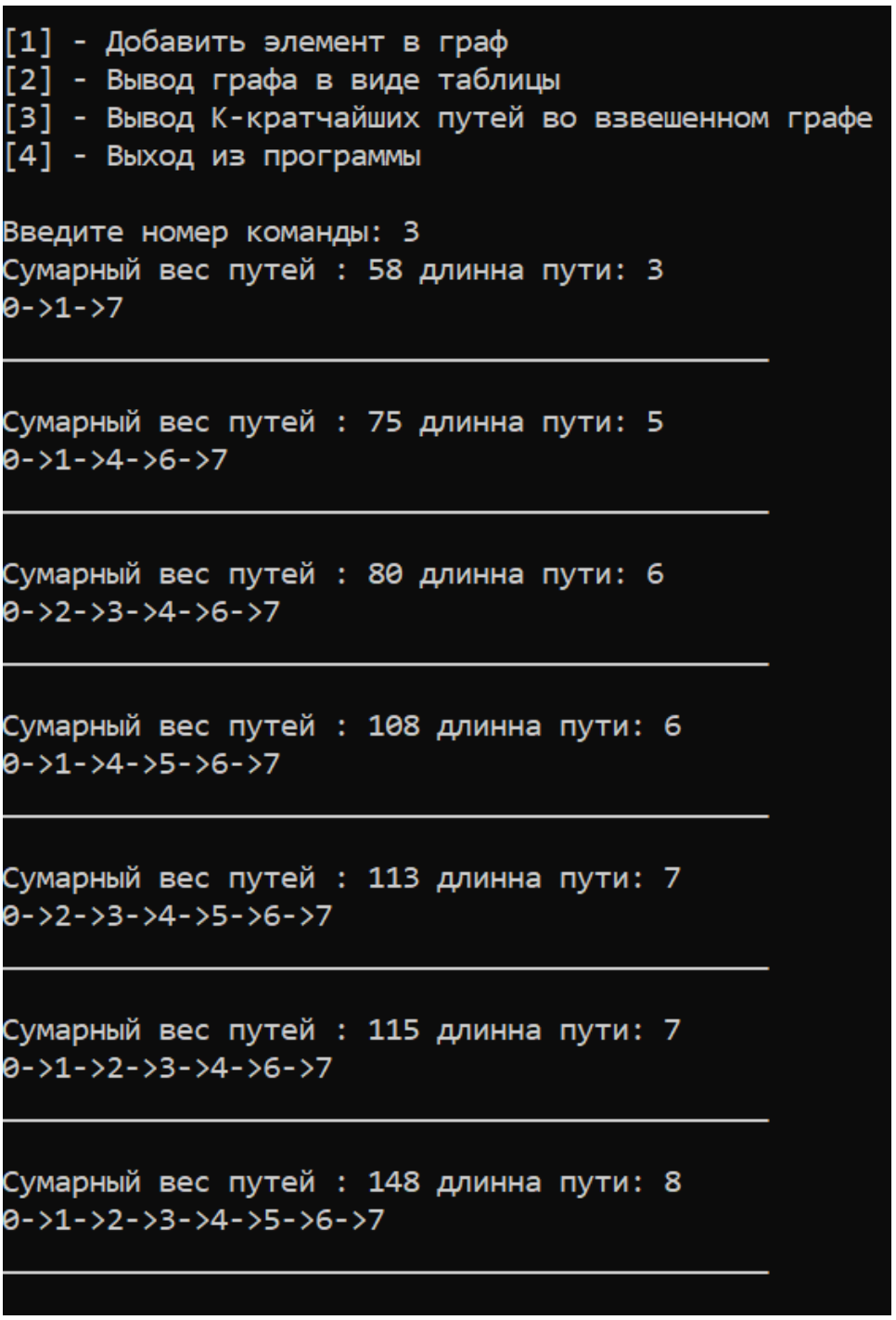


Рис.10 Скриншот вывода k- кратчайших путей во взвешенном графе

Выводы

1. В ходе работы была создана программа для работы с графами.
2. Также были реализованы функции добавления, вывода исходного графа, вывода k-кратчайших путей во взвешенном графе.
3. Были изучены особенности алгоритма Йена и алгоритма Дейкстры:
4. Преимущества: Алгоритм Йена позволяет найти наиболее оптимальные пути не перебирая все возможные.
5. Недостатки: Сложность времени алгоритм Йена напрямую зависит от сложности алгоритма кратчайшего пути. В данном случае это алгоритм Дейкстры, который в худшем случае имеет сложность O(n^2)
6. Таким образом, была изучена работа Алгоритма Йена и принцип представления графов в памяти компьютера .

Список используемых информационных источников

1. Сыромятников В.П. Структуры и алгоритмы обработки данных, лекции, РТУ МИРЭА, Москва, 2020/2021 уч./год.
2. Документация по языку программирования C++, интернет-ресурс: https://en.cppreference.com/w/ (Дата обращения – 10.12.2020)
3. Интегрированная среда разработки для языков программирования C и C++, разработанная компанией JetBrains - CLion / Copyright © 2000-2020 JetBrains s.r.o., интернет-ресурс: https://www.jetbrains.com/clion/learning-center/ (Дата обращения – 10.12.2020).
4. ГОСТ 19.701-90 ЕСПД. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения. Интернет-ресурс: http://docs.cntd.ru/document/gost-19-701-90-espd (Дата обращения – 10.12.2020).
5. Описание алгоритма Йена. интернет-ресурс: https://ru.qaz.wiki/wiki/Yen%27s_algorithm (Дата обращения – 10.12.2020).
6. Описание алгоритма Дейкстры. интернет-ресурс: https://ru.wikipedia.org/wiki/Алгоритм_Дейкстры (Дата обращения – 10.12.2020).
7. Описание оформления таблицы в консоли C++. интернет-ресурс: https://ru.stackoverflow.com/questions/437788 (Дата обращения – 10.12.2020).