Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

КУРСОВАЯ РАБОТА

по дисциплине «Объектно-ориентированное программирование»

Алгоритм Беллмана-Форда, алгоритм Флойда-Уоршелла

Студент группы
3331506/20401 Т. Г. Галанин

Преподаватель,
доцент, к.т.н. М. С. Ананьевский

Санкт-Петербург
2025

СОДЕРЖАНИЕ

1 Введение

Алгоритм Беллмана-Форда (*Bellman-Ford algorithm*) позволяет решить задачу о кратчайшем пути из одной вершины в общем случае, когда вес каждого из ребер может быть отрицательным. Для заданного графа алгоритм возвращает логические значение, указывающее на то, содержится ли в графе цикл с отрицательным весом (сумма весов ребер меньше нуля), достижимый из начальной вершины. Если такой цикл существует, то в алгоритме указывается, что решения не существует. Если же таких циклов нет, алгоритм выдает кратчайшие пути и их вес.

Алгоритм Флойда-Уоршелла (*Floyd-Warshall algorithm*) решает задачу о поиске кратчайших путей между всеми парами вершин в графе. Как и в алгоритме Беллмана-Форда, наличие ребер с отрицательным весом допускается, но предполагается, что циклы с отрицательным весом отсутствуют.

В курсовой работе будут разработаны 3 программы на языке C++, реализующие работу алгоритмов. Будет проведен анализ скорости выполнения алгоритмов.

2 Основная часть

2.1 Алгоритм Беллмана-Форда

Процесс ослабления *(relaxation)* заключается в проверке, нельзя ли улучшить найденный до сих пор кратчайший путь к вершине. В этом алгоритме используется ослабление, в результате которого величина, представляющая собой оценку веса кратчайшего пут из начальной вершины (истока) к каждой из вершин, уменьшается до тех пор, пока она не станет равна фактическому весу кратчайшего пути. Максимальное количество итераций при условии отсутствия циклов с отрицательным весом равно количеству вершин графа минус один. Рассмотрим пример реализации алгоритма для графа, представленного на рисунке 2.1.1.

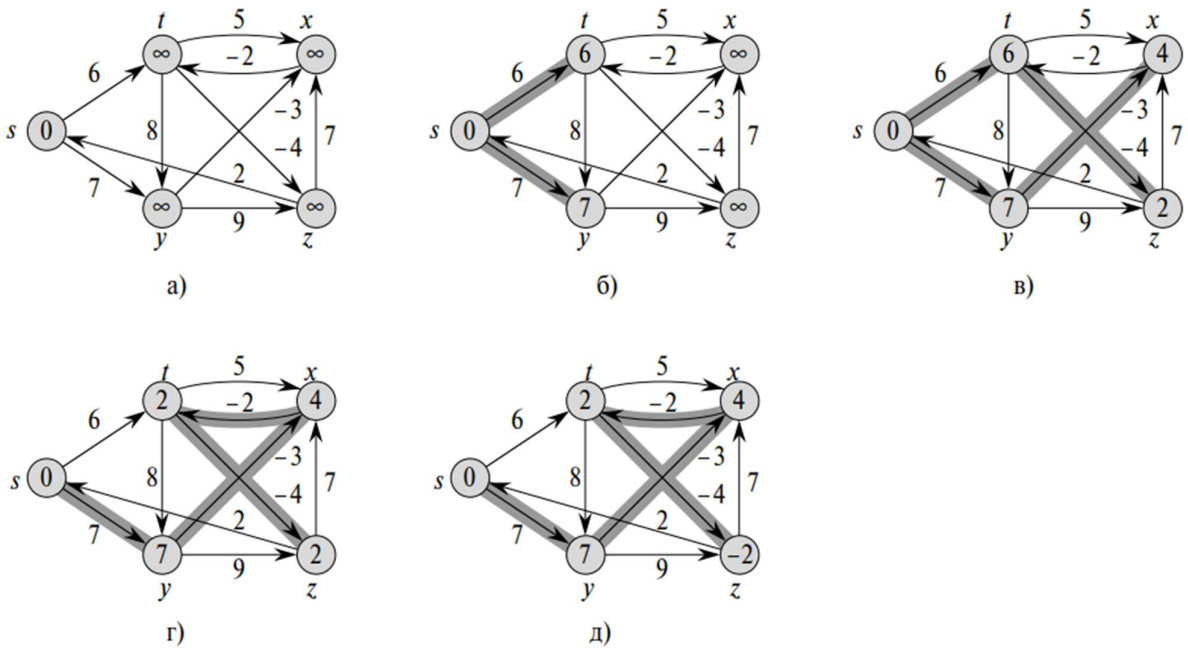

Рисунок 2.1.1 – Выполнение алгоритма Беллмана-Форда

На рисунке 2.1.1.а показана ситуация перед первым проходом: для истока значение расстояния равно нулю, для остальных вершин – бесконечности. Далее в каждой итерации для каждой вершины будем запоминать значение кратчайшего расстояния после ослабления и предыдущую вершину. При каждом проходе ребра будем ослаблять в следующем порядке: (*t, x*), (*t, y*), (*t, z*), (*x, t*), (*y, x*), (*y, z*), (*z, x*), (*z, s*), (*s, t*), (*s, y*).

Таблица 2.1.1 – Значения кратчайшего расстояния до вершины и предшественника до первой итерации.

Вершина	s	t	x	y	z
D	0	∞	∞	∞	∞
П	-	-	-	-	-

В первой итерации (рис. 2.1.1.б) ослабляются только ребра (*s, t*) и (*s, y*).

Таблица 2.1.2 – Значения кратчайшего расстояния до вершины и предшественника после первой итерации.

Вершина	s	t	x	y	z
D	0	6	∞	7	∞
П	-	s	-	s	-

Во второй итерации (рис. 2.1.1.в) ослабляются следующие ребра:

- (*t, x*): D(*x*) = D(*t*) + (*t, x*) = 6 + 5 = 11. Полученное значение D(*x*) меньше предыдущего (11 < ∞). Записываем новое значение расстояния для *x* и нового предшественника – *t.*
- (*t, z*): D(*z*) = D(*t*) + (*t, z*) = 6 – 4 = 2. Полученное значение D(*z*) меньше предыдущего (2 < ∞). Записываем новое значение расстояния для *z* и нового предшественника – *t.*
- (*y, x*): D(*x*) = D(*y*) + (*y, x*) = 7 – 3 = 4. Полученное значение D(*z*) меньше предыдущего (2 < 11). Записываем новое значение расстояния для *x* и нового предшественника – *y.*

Для остальных ребер ослабления не происходит, так как полученные расстояния для них больше предыдущих, например:

- (*t, y*): D(*y*) = D(*t*) + (*t, y*) = 6 + 8 = 14. Полученное значение D(*y*) больше предыдущего (14 > 7). Оставляем значения для *y* без изменений.

Таблица 2.1.3 – Значения кратчайшего расстояния до вершины и предшественника после второй итерации.

Вершина	*s*	*t*	*x*	*y*	*z*
D	0	6	4	7	2
П	-	*s*	*y*	*s*	*t*

В третьей итерации (рис 2.1.1.г) происходит ослабление только одного ребра:

- (x, t): $D(t) = D(x) + (x, t) = 4 - 2 = 2$. Полученное значение $D(t)$ меньше предыдущего ($2 < 6$). Записываем новое значение расстояния для *t* и нового предшественника – *x*.

Значения для остальных вершин остаются без изменений.

Таблица 2.1.4 – Значения кратчайшего расстояния до вершины и предшественника после третьей итерации.

Вершина	*s*	*t*	*x*	*y*	*z*
D	0	2	4	7	2
П	-	*x*	*y*	*s*	*t*

В четвертой итерации (рис 2.1.1.д) происходит ослабление только одного ребра:

- (t, z): $D(z) = D(t) + (t, z) = 2 - 4 = -2$. Полученное значение $D(z)$ меньше предыдущего ($-2 < 2$). Записываем новое значение расстояния для *z* и нового предшественника – *t*.

Значения для остальных вершин остаются без изменений.

Таблица 2.1.3 – Значения кратчайшего расстояния до вершины и предшественника после второй итерации.

Вершина	*s*	*t*	*x*	*y*	*z*
D	0	2	4	7	–2
П	-	*x*	*y*	*s*	*t*

Значения расстояний и предшественников после четвертой итерации являются окончательными, и дальнейшая работа алгоритма не изменит результат, если в графе нет отрицательных циклов. Если же при последующих итерациях можно улучшить расстояния, граф содержит отрицательные циклы – решения не существует.

2.2 Алгоритм Флойда-Уоршелла

Один из методов, позволяющих строить кратчайшие пути в алгоритме Флойда-Уоршелла – вычисление матрицы D, содержащей веса кратчайших путей, с последующим конструированием на ее основе матрицы предшествования П. Каждый элемент $d(i, j)$ матрицы D является кратчайшим путем из вершины i до вершины j. Каждый элемент $\pi(i, j)$ матрицы П является предшественником вершины j на кратчайшем пути из вершины i. Если граф содержит N вершин, алгоритм реализуется за N итераций.

Для примера рассмотрим граф, представленный на рисунке 2.2.1. Перед первой итерацией запишем известные значения расстояний и предшественников.

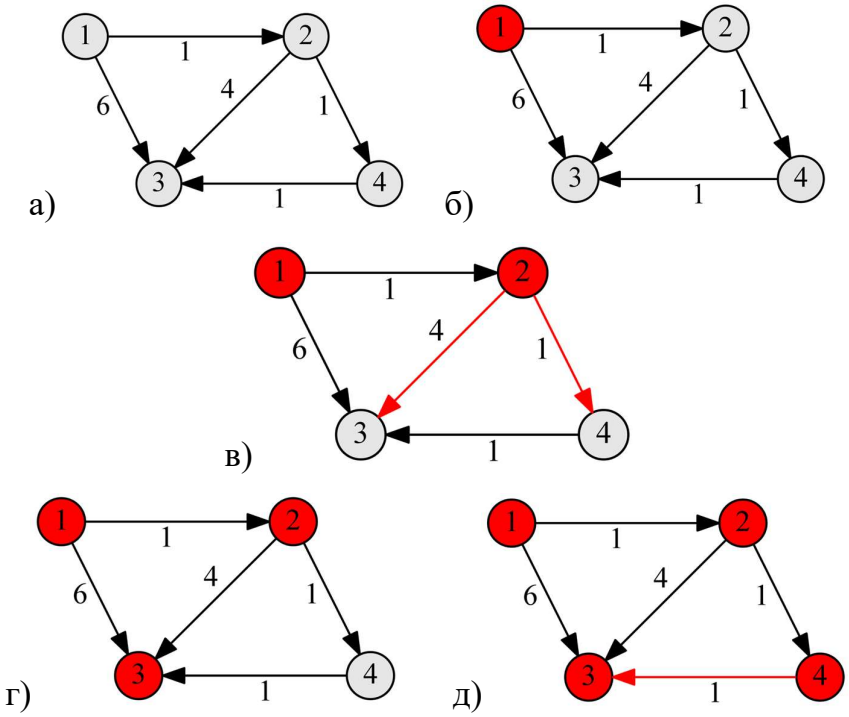

Рисунок 2.2.1 – Алгоритм Флойда-Уоршелла

Таблица 2.2.1 – Матрицы D и П перед первой итерации

D					П				
i \ *j*	1	2	3	4	*i* \ *j*	1	2	3	4
1	x	1	6	∞	1	x	1	1	-
2	∞	x	4	1	2	-	x	2	2
3	∞	∞	x	∞	3	-	-	x	-
4	∞	∞	1	x	4	-	-	4	x

Значения отсутствующих расстояний примем равными бесконечности. В каждой *k*-той итерации будем сравнивать значение имеющего пути из *i* в *j* со значением пути через вершину *k*. Результат первой итерации:

Таблица 2.2.2 – Матрицы D и П после первой итерации

D					П				
i \ *j*	1	2	3	4	*i* \ *j*	1	2	3	4
1	x	1	6	∞	1	x	1	1	-
2	∞	x	4	1	2	-	x	2	2
3	∞	∞	x	∞	3	-	-	x	-
4	∞	∞	1	x	4	-	-	4	x

После первой итерации (рис. 2.2.1.б) матрицы не изменились. Это связано с тем, что при $k = 1$ путь из каждой вершины в вершину k равен бесконечности, следовательно, получившееся расстояние в любом случае будет больше имеющегося.

После второй итерации (рис. 2.2.1.в) произойдет изменение в строке $i = 1$, так как во всех остальных случаях при $k = 2$ путь из i-той в k-тую вершину будет равен бесконечности. Аналогично пройдем третью и четвертую итерации.

Таблица 2.2.3 – Матрицы D и П после второй итерации

D					П				
$i \setminus j$	1	2	3	4	$i \setminus j$	1	2	3	4
1	x	1	5	2	1	x	1	2	2
2	∞	x	4	1	2	-	x	2	2
3	∞	∞	x	∞	3	-	-	x	-
4	∞	∞	1	x	4	-	-	4	x

Таблица 2.2.4 – Матрицы D и П после третьей итерации

D					П				
$i \setminus j$	1	2	3	4	$i \setminus j$	1	2	3	4
1	x	1	5	2	1	x	1	2	2
2	∞	x	4	1	2	-	x	2	2
3	∞	∞	x	∞	3	-	-	x	-
4	∞	∞	1	x	4	-	-	4	x

Таблица 2.2.5 – Матрицы D и П после четвертой итерации

D					П				
$i \setminus j$	1	2	3	4	$i \setminus j$	1	2	3	4
1	x	1	3	2	1	x	1	4	2
2	∞	x	2	1	2	-	x	4	2
3	∞	∞	x	∞	3	-	-	x	-
4	∞	∞	1	x	4	-	-	4	x

2.3 Программная реализация алгоритмов

Программа, реализующая алгоритмов Беллмана-Форда и Флойда-Уоршелла представлена в приложениях А и Б соответственно. Проведем тестирование программы и измерим зависимость среднего времени выполнения алгоритма от количества вершин графа.

Таблица 2.3.1 – Время выполнения алгоритма Беллмана-Форда

Количество вершин	10	100	1000	10000
Время работы (мкс)	40	1005	19024	309975
	40	834	17529	308725
	46	1318	22870	1114398
	29	931	15386	196008
	45	1141	21070	44390
	73	1030	17198	1195896
	69	1043	5104	206400
	41	885	18620	1278917
	46	244	3421	120177
	39	981	18161	318590
Среднее время работы	46.8	941.2	15838.3	509347.6

Таблица 2.3.2 – Время выполнения алгоритма Флойда-Уоршелла

Количество вершин	10	100	1000
Время работы (мкс)	34	9065	7108289
	32	8432	6736456
	46	7628	6222711
	34	8292	6933185
	22	6457	6832652
	49	6523	6886631
	37	8660	7116568
	38	7131	6793786
	37	9699	6527853
	52	6938	6326745
Среднее время работы	38.1	7882.5	6748487.6

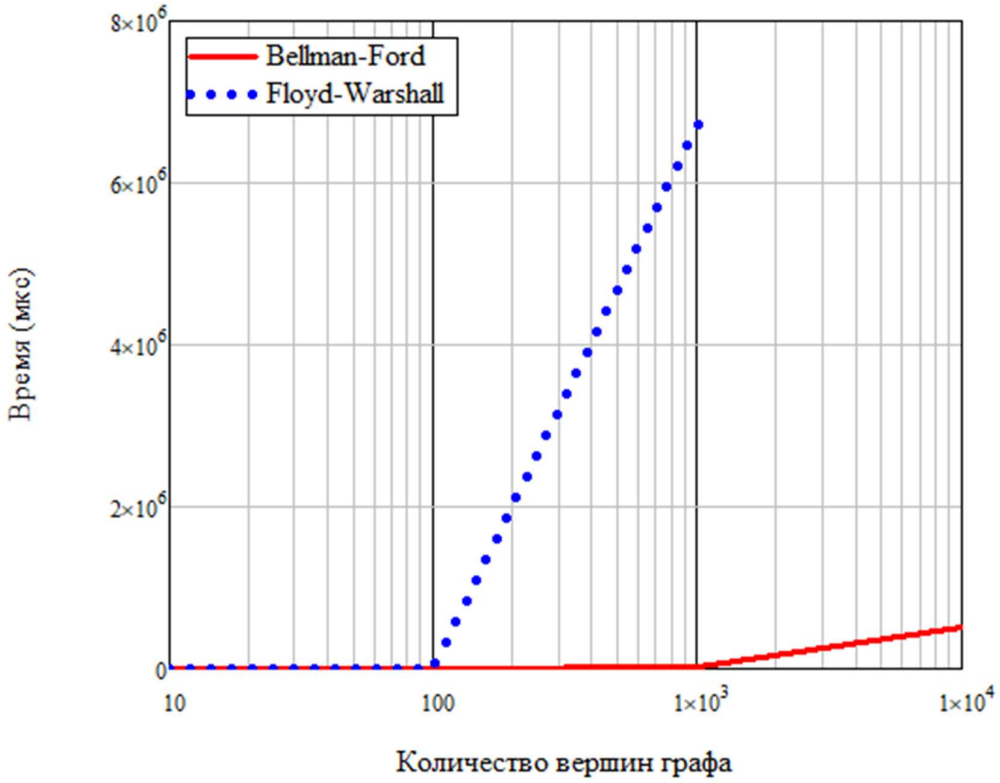

Рисунок 2.3.1 – График зависимости времени выполнения программ от количества вершин

Заключение

В ходе выполнения курсовой работы были успешно реализованы и проанализированы два алгоритма работы с графами: алгоритм Беллмана-Форда и алгоритм Флойда-Уоршелла. Реализация на языке C++ продемонстрировала их эффективность и соответствие теоретическим ожиданиям. Можно заметить, что реализованный алгоритм Флойда-Уоршелла оказывается неэффективен при количестве вершин больше 1000. Экспериментальное тестирование и построение графиков показали линейную зависимость времени выполнения программы от количества вершин (на рисунке 2.3.1 график представлен в полулогарифмическом масштабе).

Практическая значимость работы заключается в возможности использования этих алгоритмов в реальных задачах, таких как анализ сетевых структур, планирование процессов с зависимостями и оптимизация маршрутов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Кормен, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд. Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. – М. : Издательские дом «Вильямс», 2011. – 1296 с.
2. Левитин А. В. Алгоритмы. Введение в разработку и анализ – М. : Издательские дом «Вильямс», 2006. – 576 с.

ПРИЛОЖЕНИЕ А

Программная реализация алгоритма Беллмана-Форда

```
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <set>
#include <climits>
#include <algorithm>
#include <stdexcept>
#include <random>
#include <chrono>

using namespace std;

struct Edge {
    string init;
    string final;
    int weight;
};

class Graph {
private:
    vector<string> vertices;
    vector<Edge> edges;
    unordered_map<string, int> vertex_to_index;

public:
    Graph(vector<string> vertices, vector<Edge> edges);
    void BellmanFord(string start);
    void printGraph();
};

Graph::Graph(vector<string> A_vertices, vector<Edge> A_edges) {
    if (A_vertices.empty()) {
        throw invalid_argument("Vertex list is empty.");
    }

    set<string> unique_verts(A_vertices.begin(), A_vertices.end());
    if (unique_verts.size() != A_vertices.size()) {
        throw invalid_argument("Duplicate vertices found.");
    }

    vertices = A_vertices;
    for (size_t i = 0; i < vertices.size(); ++i) {
        vertex_to_index[vertices[i]] = i;
    }

    set<string> vert_set(vertices.begin(), vertices.end());
```

```
    for (const Edge& e : A_edges) {
        if (vert_set.find(e.init) == vert_set.end() || vert_set.find(e.final) == vert_set.end()) {
            throw invalid_argument("Edge connects non-existing vertex.");
        }
    }
    edges = A_edges;
}

void Graph::BellmanFord(string start) {
    auto it = find(vertices.begin(), vertices.end(), start);
    if (it == vertices.end()) {
        throw invalid_argument("Start vertex '" + start + "' not found.");
    }

    int v = vertices.size();
    int inf = INT_MAX;
    vector<int> distances(v, inf);
    vector<int> predecessors(v, -1);
    int start_idx = vertex_to_index[start];
    distances[start_idx] = 0;

    bool changed;
    for (int i = 0; i < v - 1; ++i) {
        changed = false;
        for (const Edge& e : edges) {
            int u = vertex_to_index[e.init];
            int v_idx = vertex_to_index[e.final];
            int w = e.weight;

            if (distances[u] != inf && distances[u] + w < distances[v_idx]) {
                distances[v_idx] = distances[u] + w;
                predecessors[v_idx] = u;
                changed = true;
            }
        }
        if (!changed) break;
    }

    bool has_negative_cycle = false;
    for (const Edge& e : edges) {
        int u = vertex_to_index[e.init];
        int v_idx = vertex_to_index[e.final];
        int w = e.weight;

        if (distances[u] != inf && distances[u] + w < distances[v_idx]) {
            has_negative_cycle = true;
            break;
        }
    }

    if (has_negative_cycle) {
        throw runtime_error("Negative-weight cycle detected. No solution exists.");
```

```
    }
}

void Graph::printGraph() {
        printf("Vertices:\n");
        for (const auto& v : vertices) {
            printf("%s ", v.c_str());
        }
        printf("\n\nEdges:\n");

        for (const Edge& e : edges) {
            printf("%s -> %s (weight: %d)\n",
                   e.init.c_str(),
                   e.final.c_str(),
                   e.weight);
        }
    }

Graph generate_random_graph(int num_vertices, int max_edges_per_vertex,
                            int min_weight, int max_weight) {
    if (num_vertices <= 0) {
        throw invalid_argument("Number of vertices must be positive.");
    }
    if (max_edges_per_vertex < 0) {
        throw invalid_argument("Max edges per vertex cannot be negative.");
    }

    vector<string> vertices;
    for (int i = 0; i < num_vertices; ++i) {
        vertices.push_back("V" + to_string(i));
    }

    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> weight_dist(min_weight, max_weight);

    int actual_max_edges = min(max_edges_per_vertex, num_vertices - 1);
    uniform_int_distribution<> edge_count_dist(0, actual_max_edges);

    set<pair<string, string>> unique_edges;
    vector<Edge> edges;

    for (const string& u : vertices) {
        int edge_count = edge_count_dist(gen);

        vector<string> possible_targets;
        for (const string& v : vertices) {
            if (v != u) possible_targets.push_back(v);
        }

        shuffle(possible_targets.begin(), possible_targets.end(), gen);
```

```
        int created_edges = 0;
        for (const string& v : possible_targets) {
            if (created_edges >= edge_count) break;

            if (unique_edges.insert({u, v}).second) {
                Edge e;
                e.init = u;
                e.final = v;
                e.weight = weight_dist(gen);
                edges.push_back(e);
                created_edges++;
            }
        }
    }

    return Graph(vertices, edges);
}

int main() {
    try {
        Graph g = generate_random_graph(10, 5, -10, 10);
        g.BellmanFord("V0");

    } catch (const invalid_argument& e) {
        cerr << "Error: " << e.what() << endl;
        return 1;
    } catch (const runtime_error& e) {
        cerr << "Error: " << e.what() << endl;
        return 2;
    }
    return 0;
}
```

ПРИЛОЖЕНИЕ Б

Программная реализация алгоритма Флойда-Уоршелла

```
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <set>
#include <climits>
#include <algorithm>
#include <stdexcept>
#include <random>
#include <chrono>
#include <unordered_map>

using namespace std;

const int INF = INT_MAX;

struct Edge {
    string init;
    string final;
    int weight;
};

class Graph {
private:
    vector<string> vertices;
    vector<Edge> edges;
    unordered_map<string, int> vertex_to_index;

public:
    Graph(vector<string> vertices, vector<Edge> edges);
    void BellmanFord(string start);
    void FloydWarshall();
    void printGraph();
};

Graph::Graph(vector<string> A_vertices, vector<Edge> A_edges) {
    if (A_vertices.empty()) {
        throw invalid_argument("Vertex list is empty.");
    }

    set<string> unique_verts(A_vertices.begin(), A_vertices.end());
    if (unique_verts.size() != A_vertices.size()) {
        throw invalid_argument("Duplicate vertices found.");
    }

    vertices = A_vertices;
    for (size_t i = 0; i < vertices.size(); ++i) {
```

```
        vertex_to_index[vertices[i]] = i;
    }

    set<string> vert_set(vertices.begin(), vertices.end());
    for (const Edge& e : A_edges) {
        if (vert_set.find(e.init) == vert_set.end() || vert_set.find(e.final) == vert_set.end()) {
            throw invalid_argument("Edge connects non-existing vertex.");
        }
    }

    edges = A_edges;
}

void Graph::FloydWarshall() {
    int N = vertices.size();
    vector<vector<int>> distance(N, vector<int>(N, INT_MAX));
    vector<vector<int>> predecessors(N, vector<int>(N, -1));

    for (int i = 0; i < N; ++i) {
        distance[i][i] = 0;
        predecessors[i][i] = i;
    }

    for (const Edge& e : edges) {
        int u = vertex_to_index[e.init];
        int v = vertex_to_index[e.final];
        distance[u][v] = e.weight;
        predecessors[u][v] = u;
    }

    for (int k = 0; k < N; ++k) {
        for (int i = 0; i < N; ++i) {
            for (int j = 0; j < N; ++j) {
                if (distance[i][k] != INT_MAX && distance[k][j] != INT_MAX) {
                    if (distance[i][j] > distance[i][k] + distance[k][j]) {
                        distance[i][j] = distance[i][k] + distance[k][j];
                        predecessors[i][j] = predecessors[k][j];
                    }
                }
            }
        }
    }

    for (int i = 0; i < N; ++i) {
        if (distance[i][i] < 0) {
            throw runtime_error("Graph contains a negative weight cycle.");
        }
    }
}

Graph generate_random_graph(int num_vertices, int max_edges_per_vertex,
                            int min_weight, int max_weight) {
```

```
    if (num_vertices <= 0) {
        throw invalid_argument("Number of vertices must be positive.");
    }
    if (max_edges_per_vertex < 0) {
        throw invalid_argument("Max edges per vertex cannot be negative.");
    }

    vector<string> vertices;
    for (int i = 0; i < num_vertices; ++i) {
        vertices.push_back("V" + to_string(i));
    }

    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> weight_dist(min_weight, max_weight);

    int actual_max_edges = min(max_edges_per_vertex, num_vertices - 1);
    uniform_int_distribution<> edge_count_dist(0, actual_max_edges);

    set<pair<string, string>> unique_edges;
    vector<Edge> edges;

    for (const string& u : vertices) {
        int edge_count = edge_count_dist(gen);

        vector<string> possible_targets;
        for (const string& v : vertices) {
            if (v != u) possible_targets.push_back(v);
        }

        shuffle(possible_targets.begin(), possible_targets.end(), gen);

        int created_edges = 0;
        for (const string& v : possible_targets) {
            if (created_edges >= edge_count) break;

            if (unique_edges.insert({u, v}).second) {
                Edge e;
                e.init = u;
                e.final = v;
                e.weight = weight_dist(gen);
                edges.push_back(e);
                created_edges++;
            }
        }
    }

    return Graph(vertices, edges);
}

int main() {
    try {
```

```
        Graph g = generate_random_graph(10, 5, -10, 10)
        g.FloydWarshall();

    } catch (const invalid_argument& e) {
        cerr << "Error: " << e.what() << endl;
        return 1;
    } catch (const runtime_error& e) {
        cerr << "Error: " << e.what() << endl;
        return 2;
    }

    return 0;
}
```