Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Объектно – ориентированное программирование

Вариант задания – Алгоритм Дейкстры.

Студент гр. 3331506/90403 П. Е. Симаков

Преподаватель М. С. Ананьевский

Санкт-Петербург

2022

Оглавление

1. Введение.

Алгоритм Дейкстры – алгоритм на графах, предложенный Эдсгером Дейкстрой в 1959 году, позволяющий найти кратчайший путь между заданной вершиной и всеми остальными вершинами графа. Граф — структура из точек-вершин, соединенных ребрами-отрезками. Его можно представить как схему дорог или как компьютерную сеть. Ребра — это связи, по ним можно двигаться от одной вершины к другой. Алгоритм Дейкстры работает для ориентированных графов, у которых нет ребер с отрицательным весом.

Основная задача — поиск кратчайшего пути по схеме, где множество точек соединено между собой отрезками. В виде такой схемы можно представить многие объекты реального мира, поэтому практических примеров использования алгоритма много: построение маршрутов на онлайн-карте, маршрутизация движения данных в компьютерной сети, поиск системой бронирования наиболее быстрых или дешевых билетов, в том числе с возможными пересадками.

2. Описание алгоритма.

Алгоритм Дейкстры пошаговый. Сначала выбирается точка, от которой будут отсчитываться пути. Затем алгоритм поочередно ищет самые короткие маршруты из исходной точки в другие. Вершины, где он уже побывал, отмечает посещенными. Алгоритм использует посещенные вершины, когда рассчитывает пути для непосещенных. Блок схема алгоритма представлена на рисунке 1.

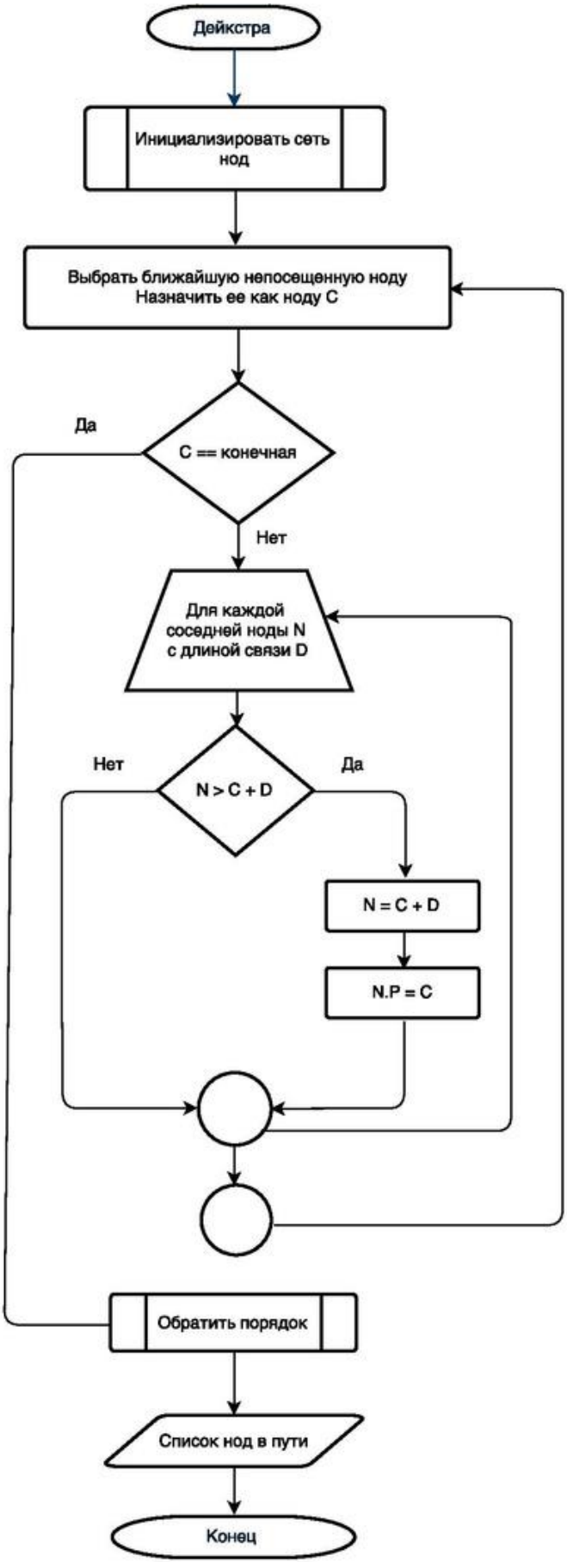
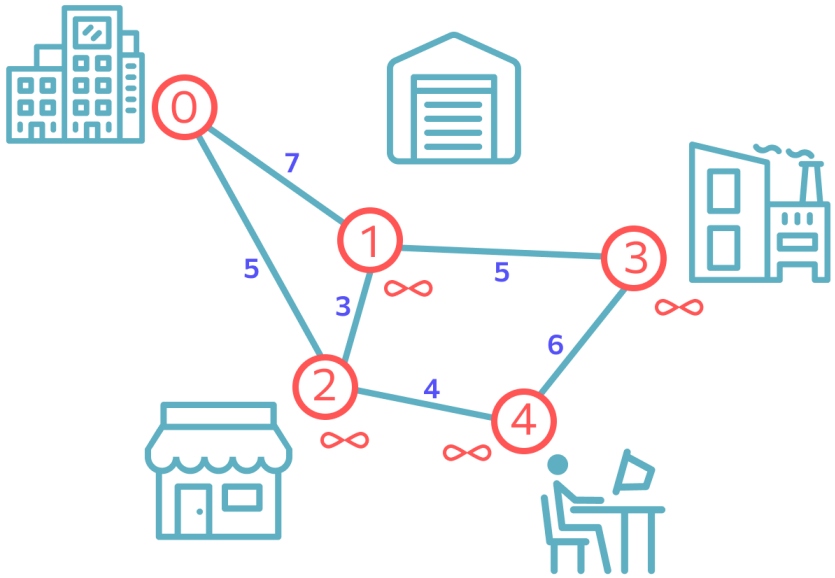


Рисунок 1 - Блок-схема алгоритма

Пример использования алгоритма.

1. Инициализация.

Одна из вершин назначается начальной, от которой будут рассчитываться длины маршрутов до других вершин, в данном случае 0. Расстояние до самой себя у этой вершины равно нулю. Расстояние до других вершин неизвестно, поэтому расстояние до них условно принимается за бесконечно большую величину.



Вершина 0 помечается как посещенная, а расстояние до нее помечается равным нулю.

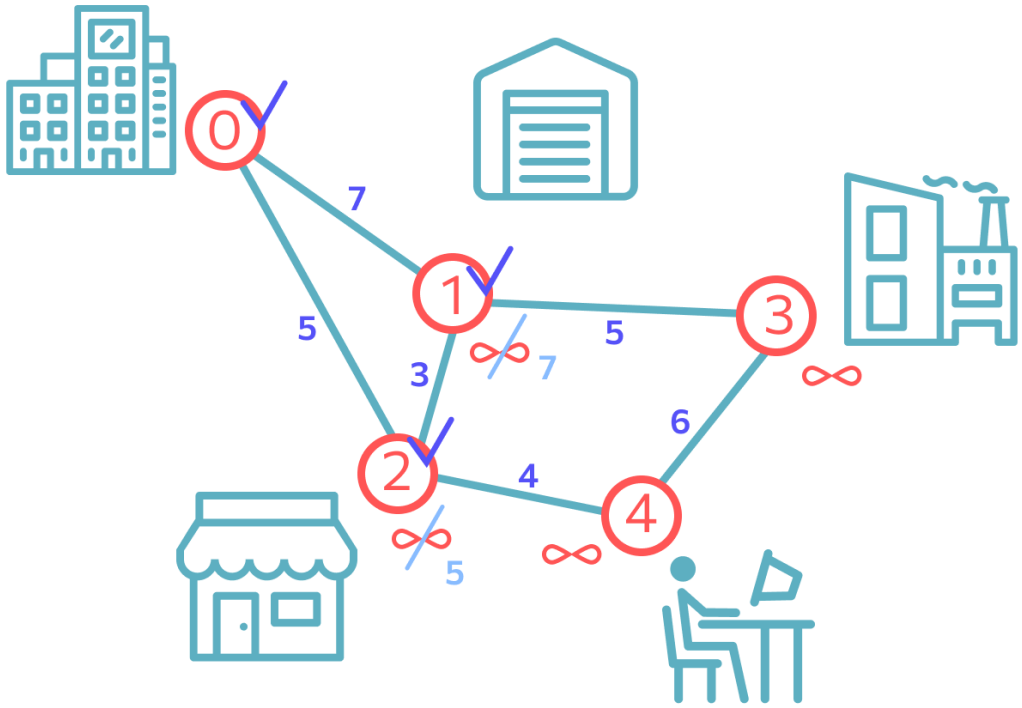
2. Первый шаг алгоритма

Находясь в нулевой вершине выбирается ближайшая вершина, то есть та, чье ребро до нулевой вершины весит меньше остальных – в данном случае это вершина 2. Алгоритм условно переходит в вершину 2 и рассматривает ее соседей.

3. Дальнейшие шаги алгоритма

Для выбранной вершины сравниваются веса ребер до соседних вершин и записывается длина пути до них с учетом уже пройденного на предыдущих

шагах пути. Также учитываются вершины, которые уже помечены как посещенные. Если рассматриваемая вершина имеет несколько общих ребер с уже посещенными вершинами, то записывается минимальное расстояние, учитывающее путь от начальной точки. Например, в вершину 1 можно попасть из вершины 2 и вершины 0, которые помечены как посещенные. При этом путь 0-1 короче, чем 0-2-1, таким образом в таблицу минимальных путей между графами заносится длина пути 0-1.



Шаги повторяются, пока на графе есть непосещенные точки. Если вершину не посетили, она не участвует в расчетах.

Когда непосещенные вершины заканчиваются, алгоритм прекращает работу. Результатом работы алгоритма может служить расстояние между двумя заданными вершинами или список кратчайших маршрутов от одной заданной вершины.

3. Исследование алгоритма

Вершины хранятся в некоторой структуре данных, поддерживающей операции изменения произвольного элемента и извлечения минимального.

Дан граф с V вершин и E ребер.

Каждая вершина извлекается ровно один раз, то есть, требуется O(V) извлечений. В худшем случае, каждое ребро приводит к изменению одного элемента структуры, то есть, O(E) изменений.

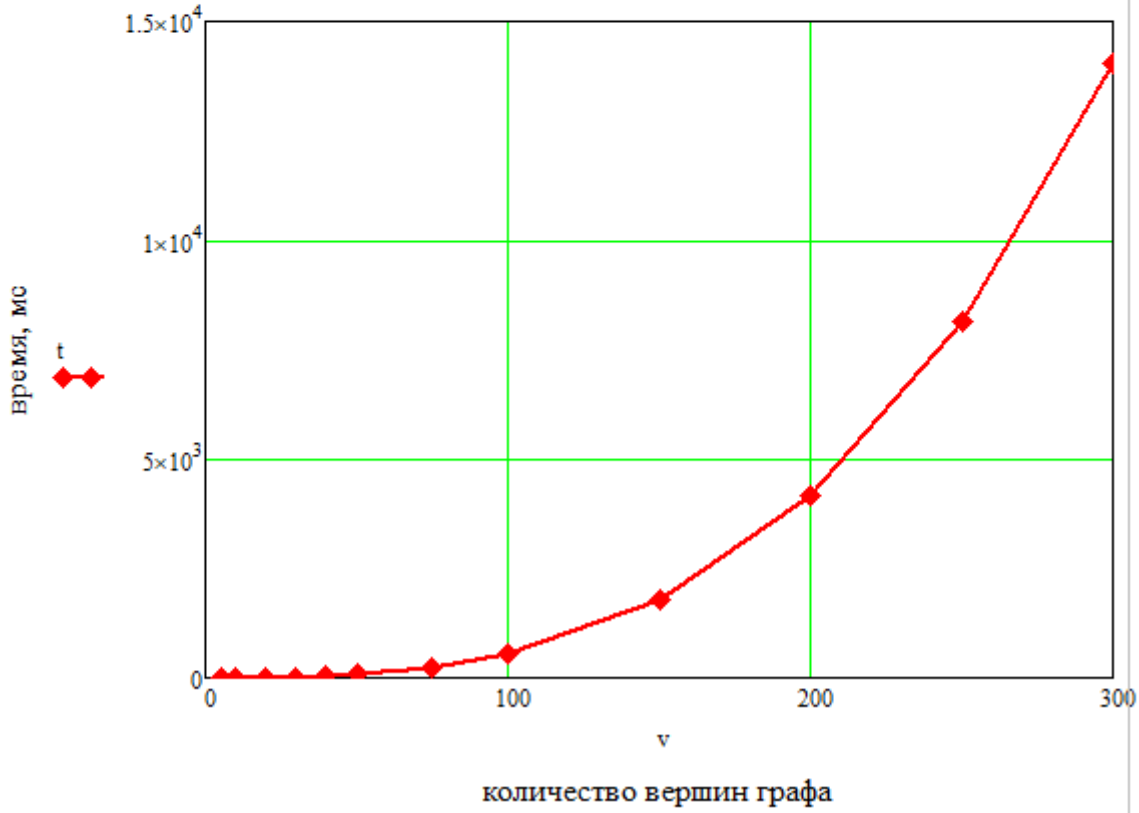
Если вершины хранятся в простом массиве и для поиска минимума используется алгоритм линейного поиска, временная сложность алгоритма Дейкстры составляет $O(V * V + E) = O(V^2)$.

Если же используется очередь с приоритетами, реализованная на основе двоичной кучи, то мы получаем $O(V \log V + E \log E) = O(E \log V)$.

Если же очередь с приоритетами была реализована на основе кучи Фибоначчи, получается наилучшая оценка сложности $O(V \log V + E)$.

Для эмпирического исследования алгоритма измерим время выполнения для графов с разным количеством вершин.

$$v := \begin{pmatrix} 5 \\ 10 \\ 20 \\ 30 \\ 40 \\ 50 \\ 75 \\ 100 \\ 150 \\ 200 \\ 250 \\ 300 \end{pmatrix} \quad t := \begin{pmatrix} 0 \\ 1 \\ 5 \\ 16 \\ 37 \\ 72 \\ 232 \\ 538 \\ 1771 \\ 4168 \\ 8129 \\ 14020 \end{pmatrix}$$



4. Заключение.

В работе был рассмотрен алгоритм Дейкстры для нахождения кратчайшего пути в связном графе без отрицательных ребер. Алгоритм отличается высокой скоростью работы по сравнению с алгоритмами решающими аналогичные задачи, такими как алгоритма Флойда-Уоршелла ($O(n^3)$) или алгоритм Форда Беллмана ($O(VE)$).

Список литературы.

1. Тим Рафгарден Совершенный алгоритм. Графовые алгоритмы и структуры данных. - СПб.: Прогресс книга, 2019. - 256 с.
2. Алгоритм Дейкстры // SkillFactory.Блог URL: https://blog.skillfactory.ru/glossary/algoritm-dejkstry/ (дата обращения: 16.05.2022).
3. Графы для самых маленьких: Dijkstra // Habr URL: https://habr.com/ru/post/202314/ (дата обращения: 16.05.2022).

Приложение 1.

```
#include <vector>
#include <list>
#include <stack>

typedef enum{
    UNDIRECTED,
    DIRECTED,
} EdgeType;

// Получить "бесконечность" для типа T
// (целочисленный тип - максимальное значение, с плавающей запятой - inf)
template<typename T>
T get_inf();

// переменная, содержащая "бесконечность", т.е. такой вес ребра,
// который эквивалентен отсутствию этого ребра
template <typename T>
T INF = get_inf<T>();

template <typename T> class Edge;
template <typename T> class Vertex;
template <typename T> class BaseGraph;


template <typename T>
class Vertex {
    friend BaseGraph<T>;
private:
    int id;
    std::list<Edge<T>> edges;
public:
    explicit Vertex(int id = 0);
    Vertex(const Vertex &other);
    ~Vertex() = default;
public:
    void add_edge(Vertex<T>* neighbor, T distance);
    void remove_edge(Vertex<T>* neighbor);
public:
    int get_id() {return id;}
    std::list<Edge<T>> get_edges() {return edges;}
};


template <typename T>
class Edge{
private:
    Vertex<T>* neighbor;
    T distance;
public:
    explicit Edge(Vertex<T>* neighbor = nullptr, T distance = 0);
    ~Edge() = default;
public:
    bool operator== (const Edge<T> &other);
public:
    Vertex<T>* get_neighbor() {return neighbor;}
    T get_distance() {return distance;}
};


// Класс графа с базовой функциональностью
template <typename T>
```

```
class BaseGraph{
protected:
    int id_counter = 0;
    std::list<Vertex<T>*> vertices;
    std::vector<std::vector<T>> adjacency_matrix;    // Матрица смежности
    std::vector<std::list<int>> adjacency_list;       // Список смежности
public:
    explicit BaseGraph(int num_of_vertices = 0);
    explicit BaseGraph(std::list<Vertex<T>*> &vertices);
    explicit BaseGraph(std::vector<std::vector<T>> &adjacency_matrix);
    explicit BaseGraph(std::vector<std::list<int>> &adjacency_list);
    BaseGraph(const BaseGraph &other);
    BaseGraph(BaseGraph &&other) noexcept;
    virtual ~BaseGraph();

// Методы для взаимодействия с графом (т.е. геттеры, сеттеры и т.д.)
public:
    Vertex<T>* find_vertex(int id);
    int add_edge(int source_id, int target_id, int weight = 0, EdgeType
edge_type = UNDIRECTED);
    int remove_edge(int source_id, int target_id, EdgeType edge_type =
UNDIRECTED);
    int add_vertex();
    int remove_vertex(int id);
public:
    int get_id_counter() {return id_counter;}
    std::list<Vertex<T>*> get_vertices() {return vertices;}
    std::vector<std::vector<T>> get_adjacency_matrix() {return
adjacency_matrix;}
    std::vector<std::list<int>> get_adjacency_list() {return adjacency_list;}
public:
    void actualize_adjacency_list();
    void actualize_adjacency_matrix();
};

template<typename T>
class GraphDijkstra: virtual public BaseGraph<T>{
public:
    std::vector<std::vector<T>> dijkstra();
    int dijkstra_log(int top_from, int top_to);
};
```

```
template<typename T>
class Graph:
        virtual public GraphFloydWarshall<T>,
        virtual public GraphTarjansBridges<T>,
        virtual public GraphDijkstra<T>,
        virtual public GraphTraversal<T>,
        virtual public GraphTarjansSCCalgorithm<T> {
public:
    explicit Graph(int num_of_vertices = 0) : BaseGraph<T>(num_of_vertices)
{};
    explicit Graph(std::vector<std::vector<T>> &adjacency_matrix) :
BaseGraph<T>(adjacency_matrix) {};
    explicit Graph(std::vector<std::list<int>> &adjacency_list) :
BaseGraph<T>(adjacency_list) {};
    Graph(const Graph &other) : BaseGraph<T>(other) {};
    Graph(Graph &&other) noexcept : BaseGraph<T>(other) {};
    virtual ~Graph() = default;
};
```

Приложение 2.

```
#include<iostream>
#include <vector>
#include <queue>
#include <stack>
#include <algorithm>
#include <cstdint>
#include <tuple>
#include "graph.h"

#define EXPLICIT_INSTANTIATION(CLASSNAME)    \
    template class CLASSNAME<int8_t>;        \
    template class CLASSNAME<int16_t>;       \
    template class CLASSNAME<int32_t>;       \
    template class CLASSNAME<int64_t>;       \
                                             \
    template class CLASSNAME<float>;         \
    template class CLASSNAME<double>;        \


EXPLICIT_INSTANTIATION(Edge);
EXPLICIT_INSTANTIATION(Vertex);
EXPLICIT_INSTANTIATION(BaseGraph);
EXPLICIT_INSTANTIATION(GraphDijkstra);
EXPLICIT_INSTANTIATION(Graph);



template <typename T>
Vertex<T>::Vertex(int id) {
    this->id = id;
    this->edges = std::list<Edge<T>>();
}


template <typename T>
Vertex<T>::Vertex(const Vertex &other) {
    id = other.id;
    edges = other.edges;
}
```

```
template <typename T>
void Vertex<T>::add_edge(Vertex<T>* neighbor, T distance) {
    for (auto it = edges.begin(); it != edges.end(); it++) {
        if (it->get_neighbor() == neighbor) {
            edges.erase(it);
            break;
        }
    }
    Edge<T> edge(neighbor, distance);       //
    edges.push_back(edge);
}

template<typename T>
void Vertex<T>::remove_edge(Vertex<T>* neighbor) {
    for (auto & it : edges) {
        if (it.get_neighbor() == neighbor) {
            edges.remove(it);           // change!!! It works, but very
strange
            break;
        }
    }
}

template<typename T>
bool Edge<T>::operator== (const Edge<T> &other) {
    return (neighbor == other.neighbor) && (distance == other.distance);
}

template <typename T>
Edge<T>::Edge(Vertex<T>* neighbor, T distance) {
    this->neighbor = neighbor;
    this->distance = distance;
}



template <typename T>
BaseGraph<T>::BaseGraph(int num_of_vertices) {
    for (id_counter = 0; id_counter < num_of_vertices; id_counter++) {
        auto v = new Vertex<T>(id_counter);
        vertices.push_back(v);
    }
}


template <typename T>
BaseGraph<T>::BaseGraph(std::list<Vertex<T>*> &vertices) {
    this->vertices = vertices;
}

template <typename T>
BaseGraph<T>::BaseGraph(std::vector<std::vector<T>> &adjacency_matrix){
    this->adjacency_matrix = adjacency_matrix;
    this->id_counter = adjacency_matrix.size();

    for (int i = 0; i < adjacency_matrix.size(); i++) {
        auto v = new Vertex<T>(i);
        vertices.push_back(v);
    }
    for (int i = 0; i < adjacency_matrix.size(); i++) {
        for (int j = 0; j < adjacency_matrix.size(); j++) {
            if (adjacency_matrix[i][j] != INF<T>) {
                auto source = find_vertex(i);
                auto target = find_vertex(j);
```

```
                source->add_edge(target, adjacency_matrix[i][j]);
            }
        }
    }
}

template <typename T>
BaseGraph<T>::BaseGraph(std::vector<std::list<int>> &adjacency_list){
    this->adjacency_list = adjacency_list;
    this->id_counter = adjacency_list.size();

    for (int i = 0; i < adjacency_list.size(); i++) {
        auto v = new Vertex<T>(i);
        vertices.push_back(v);
    }
    for (int i = 0; i < adjacency_list.size(); i++) {
        for (int j = 0; j < adjacency_list.size(); j++) {
            auto it = std::find(adjacency_list[i].begin(),
adjacency_list[i].end(), j);
            if (it != adjacency_list[i].end()) {
                auto source = find_vertex(i);
                auto target = find_vertex(j);
                source->add_edge(target, 0);
            }
        }
    }
}

template <typename T>
BaseGraph<T>::BaseGraph(const BaseGraph &other) {
    adjacency_matrix = other.adjacency_matrix;
    adjacency_list = other.adjacency_list;
}

template <typename T>
BaseGraph<T>::BaseGraph(BaseGraph &&other) noexcept {
    adjacency_matrix = other.adjacency_matrix;
    adjacency_list = other.adjacency_list;
}

template <typename T>
BaseGraph<T>::~BaseGraph() {
    for (auto & v : vertices) {
        delete v;
    }
}

template <typename T>
int BaseGraph<T>::add_edge(int source_id, int target_id, int weight, EdgeType
edge_type){

    // Edge, where start and end are the same vertex, is unexpected
    if (source_id == target_id) {
        return 1;
    }

    Vertex<T>* source = find_vertex(source_id);
    Vertex<T>* target = find_vertex(target_id);

    // If some vertex is not exists, then an error is considered
    if (source == nullptr || target == nullptr) {
        return 1;
    }
```

```
    switch (edge_type) {
        case UNDIRECTED:
            source->add_edge(target, weight);
            target->add_edge(source, weight);
            break;
        case DIRECTED:
            source->add_edge(target, weight);
            break;
        default:
            break;
    }

    return 0;
}

template <typename T>
int BaseGraph<T>::remove_edge(int source_id, int target_id, EdgeType
edge_type) {

    // Edge, where start and end are the same vertex, is unexpected
    if (source_id == target_id) {
        return 1;
    }

    Vertex<T>* source = find_vertex(source_id);
    Vertex<T>* target = find_vertex(target_id);

    // If some vertex is not exists, then an error is considered
    if (source == nullptr || target == nullptr) {
        return 1;
    }

    switch (edge_type) {
        case UNDIRECTED:
            source->remove_edge(target);
            target->remove_edge(source);
            break;
        case DIRECTED:
            source->remove_edge(target);
            break;
        default:
            break;
    }

    return 0;
}



template <typename T>
int BaseGraph<T>::add_vertex() {
    auto v = new Vertex<T>(id_counter);
    vertices.push_back(v);
    id_counter++;
    return v->get_id();
}

template <typename T>
int BaseGraph<T>::remove_vertex(int id) {

    auto vtx = find_vertex(id);

    // if `vtx` is not in `vertices`, then just return
```

```
    if (vtx == nullptr) {
        return 1;
    }

    // delete all edges linking `vtx` with other vertices
    auto edges = vtx->get_edges();
    for (Edge<T> edge : edges) {
        Vertex<T>* neighbor = edge.get_neighbor();
        neighbor->remove_edge(vtx);
    }

    // delete `vtx` from list of all vertices
    auto it = std::find(vertices.begin(), vertices.end(), vtx);
    vertices.erase(it);

    return 0;
}


template <typename T>
Vertex<T>* BaseGraph<T>::find_vertex(int id) {
    for (auto v = vertices.begin(); v != vertices.end(); v++) {
        if ((*v)->id == id) {
            return *v;
        }
    }
    return nullptr;
}

template <typename T>
void BaseGraph<T>::actualize_adjacency_list() {
    adjacency_list = std::vector<std::list<int>>(id_counter,
std::list<int>());

    for (int id = 0; id < id_counter; id++) {
        Vertex<T>* vtx = find_vertex(id);

        if (vtx == nullptr)
            continue;

        std::list<Edge<T>> edges = vtx->get_edges();
        for (Edge<T> edge : edges) {
            Vertex<T>* neighbor = edge.get_neighbor();
            adjacency_list[id].push_back(neighbor->get_id());
        }
    }
}

template <typename T>
void BaseGraph<T>::actualize_adjacency_matrix() {

    // initially adjacency_matrix is filled with infinities
    adjacency_matrix = std::vector<std::vector<T>>(id_counter,
std::vector<T>(id_counter, INF<T>));

    for (int id = 0; id < id_counter; id++) {
        Vertex<T>* vtx = find_vertex(id);

        if (vtx == nullptr)
            continue;

        std::list<Edge<T>> edges = vtx->get_edges();
        for (Edge<T> edge : edges) {
            Vertex<T>* neighbor = edge.get_neighbor();
```

```
            T distance = edge.get_distance();
            adjacency_matrix[id][neighbor->get_id()] = distance;
        }
    }
}

template<typename T>
int GraphDijkstra<T>::dijkstra_log(int top_from, int top_to)
{
    //number of tops of the graph
    int tops = BaseGraph<T>::adjacency_matrix.size();
    top_from--;
    top_to--;
    // vector of infs
    std::vector <int> distances(tops, INF<T>);
    distances[top_from] = 0;
    std::priority_queue <std::pair <int, int > > q;
    q.push(std::make_pair(0, top_from));
    while (!q.empty())
        {
        int first_length = -q.top().first;
        int first_top = q.top().second;
        q.pop();
        if (first_length > distances[first_top]) continue;
        for (int i = 0; i < tops; i++)
            {
            int to = i;
            int length = BaseGraph<T>::adjacency_matrix[first_top][i];
            if (distances[to] > distances[first_top] + length)
            {
                distances[to] = distances[first_top] + length;
                q.push(std::make_pair(-distances[to], to));
            }
            }
        }
    if (distances[top_to] == INF<T>) return -1;
    else return distances[top_to];
}



template<typename T>
T get_inf() {
    T inf = std::numeric_limits<T>::infinity();
    if (inf == (T) 0) {
        return std::numeric_limits<T>::max();
    }
    return inf;
}
}
```