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. 3](#_Toc104657159)

[2. Описание алгоритма. 3](#_Toc104657160)

[3. Исследование алгоритма 6](#_Toc104657161)

[4. Заключение. 8](#_Toc104657162)

# 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.

# 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.

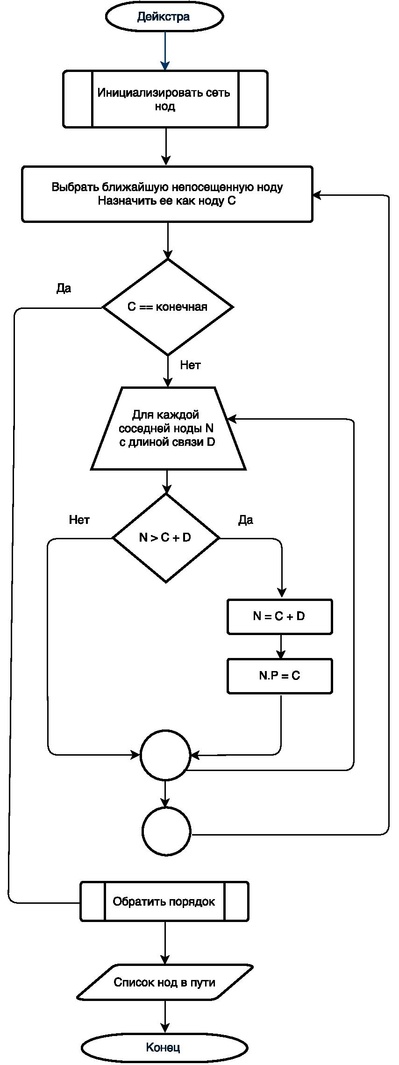


Рисунок - Блок-схема алгоритма

*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.

Изображение выглядит как текст

Автоматически созданное описание

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

1. 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.

1. 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.

Изображение выглядит как текст

Автоматически созданное описание

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.

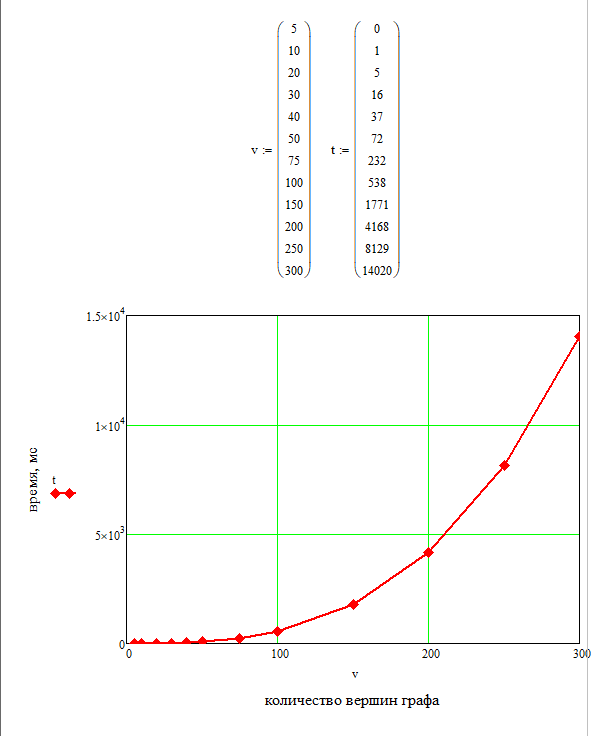
Каждая вершина извлекается ровно один раз, то есть, требуется O(V) извлечений. В худшем случае, каждое ребро приводит к изменению одного элемента структуры, то есть, O(E) изменений.

Если вершины хранятся в простом массиве и для поиска минимума используется алгоритм линейного поиска, временная сложность алгоритма Дейкстры составляет O(V \* V + E) = O(V²).

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.



# 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(n3)) 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;  
}

}