ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 8

«Графы»

Выполнил работу

Дубовик Никита

Академическая группа J3110

Принято

Ментор, Владислав Вершинин

Санкт-Петербург

2024

**Введение:**

Цель работы – Найти кратчайший путь от места проживания до университета Itmo.

Задачи работы:

Разобрать принципы работы алгоритмов BFS, DFS, Dijkstra, A\*;

Реализовать хранение графа к помощью списка смежности;

Реализовать алгоритмы BFS, DFS, Dijkstra, A\*;

Проанализировать результаты работы и выявить плюсы и минусы алгоритмов.

**Реализация:**

1. Реализация хранения матрицы

Матрица хранится в списке смежности. Вершину задаёт широта и долгота. В словаре с своей по координатам можно получить соответствующую вершину и её соседей.

std::unordered\_map<std::pair<double, double>, std::unique\_ptr<Node>, HashPair> nodes;

Используется библиотека memory для упрощённой и автоматической работы с памятью.

1. Алгоритм BFS

std::vector<Node\*> Graph::bfs(Node\* start\_node, Node\* dest\_node) {

std::deque<Node\*> queue{ start\_node }; // Queue for BFS

std::unordered\_set<Node\*> visited{ start\_node }; // Set to track visited nodes to avoid processing the same node multiple times

std::unordered\_map<Node\*, Node\*> parent; // Map to store the parent of each node for path reconstruction

parent[start\_node] = nullptr;

// Perform BFS until there are no more nodes in the queue

while (!queue.empty()) {

Node\* current\_node = queue.front();

queue.pop\_front();

// Check if we have reached the destination node

if (current\_node == dest\_node) {

return construct\_path(current\_node, parent);

}

// Explore all adjacent nodes (edges) of the current node

for (const auto& edge : current\_node->edges) {

if (visited.insert(edge.first).second) {

parent[edge.first] = current\_node;

queue.push\_back(edge.first);

}

}

}

return {};

}

1. Алгоритм DFS

std::vector<Node\*> Graph::dfs(Node\* start\_node, Node\* dest\_node) {

std::deque<Node\*> stack{ start\_node }; // Stack for DFS

std::unordered\_set<Node\*> visited{ start\_node }; // Set to track visited nodes to avoid processing the same node multiple times

std::unordered\_map<Node\*, Node\*> parent; // Map to store the parent of each node for path reconstruction

parent[start\_node] = nullptr;

// Perform DFS until there are no more nodes in the stack

while (!stack.empty()) {

Node\* current\_node = stack.back();

stack.pop\_back();

// Check if we have reached the destination node

if (current\_node == dest\_node) {

return construct\_path(current\_node, parent);

}

// Explore all adjacent nodes (edges) of the current node

for (const auto& edge : current\_node->edges) {

if (visited.insert(edge.first).second) {

parent[edge.first] = current\_node;

stack.push\_back(edge.first);

}

}

}

return {};

}

1. Алгоритм Дейкстры

std::vector<Node\*> Graph::dijkstra(Node\* start\_node, Node\* dest\_node) {

std::priority\_queue<QueueNode, std::vector<QueueNode>, std::greater<>> priority\_queue; // Priority queue for dijkstra

std::unordered\_map<Node\*, double> path\_cost;

path\_cost.reserve(nodes.size());

for (const auto& node : nodes) {

path\_cost[node.second.get()] = INFINITY;

}

std::unordered\_map<Node\*, Node\*> parent; // Map to store the parent of each node

parent[start\_node] = nullptr;

path\_cost[start\_node] = 0;

priority\_queue.emplace(0, start\_node);

// Perform dijkstra until there are no more nodes in the priority\_queue

while (!priority\_queue.empty()) {

auto current = priority\_queue.top();

priority\_queue.pop();

if (current.weight <= path\_cost[current.node]) {

// Check if we have reached the destination node

if (current.node == dest\_node) {

return construct\_path(current.node, parent);

};

for (const auto& edge : current.node->edges) {

double weights\_sum = current.weight + edge.second;

if (weights\_sum < path\_cost[edge.first]) {

path\_cost[edge.first] = weights\_sum;

priority\_queue.emplace(weights\_sum, edge.first);

parent[edge.first] = current.node;

}

}

}

}

return {};

}

1. Алгоритм A\*

std::vector<Node\*> Graph::a\_star(Node\* start\_node, Node\* dest\_node) {

std::priority\_queue<QueueNode, std::vector<QueueNode>, std::greater<>> priority\_queue; // Priority queue for A\*

std::unordered\_map<Node\*, double> path\_cost;

std::unordered\_map<Node\*, double> total\_estimated\_cost;

path\_cost.reserve(nodes.size());

total\_estimated\_cost.reserve(nodes.size());

for (const auto& node\_p : nodes) {

Node\* node = node\_p.second.get();

path\_cost[node] = INFINITY;

total\_estimated\_cost[node] = INFINITY;

}

std::unordered\_map<Node\*, Node\*> parent;

parent[start\_node] = nullptr;

path\_cost[start\_node] = 0;

total\_estimated\_cost[start\_node] = a\_star\_heuristic(start\_node, dest\_node);

priority\_queue.emplace(total\_estimated\_cost[start\_node], start\_node);

// Perform A\* until there are no more nodes in the priority\_queue

while (!priority\_queue.empty()) {

Node\* current\_node = priority\_queue.top().node;

priority\_queue.pop();

// Check if we have reached the destination node

if (current\_node == dest\_node) {

return construct\_path(current\_node, parent);

};

for (const auto& edge : current\_node->edges) {

double weights\_sum = path\_cost[current\_node] + edge.second;

if (weights\_sum < path\_cost[edge.first]) {

parent[edge.first] = current\_node;

path\_cost[edge.first] = weights\_sum;

total\_estimated\_cost[edge.first] = weights\_sum + a\_star\_heuristic(edge.first, dest\_node);

priority\_queue.emplace(total\_estimated\_cost[edge.first], edge.first);

}

}

}

return {};

}

**Экспериментальная часть:**

BFS:

Time BFS: 0.038253 seconds

DFS:

Time DFS: 0.0990041 seconds

Dijkstra:

Time Dijkstra: 0.125093 seconds

A\*:

Time A\*: 0.201551 seconds

**Заключение:**

В ходе выполнения работы мною был реализован алгоритмы BFS, DFS, Dijkstra, A\*. После получения экспериментальных результатов можно сделать выводы по работе и предназначению данных алгоритмов. DFS как и BFS позволяют находить путь от одной вершины до другой, при этом работаю достаточно быстро, но не позволяют найти кратчайший путь во взвешенном графе. Для нахождения минимального пути по рёбрам можно использовать BFS, DFS для этой задачи подходит плохо в виду его способа обхода. Лучше всего для нахождения кратчайших путей во взвешенном графе подходят алгоритмы Dijkstra, A\*. Особенность дейкстры в том, что его целью является нахождения кратчайшего пути от начальной вершины ко всем остальным вершинам графа, а не к конкретной, а также работает с графами, где все веса рёбер положительны. Особенность A\* в том, что его целью является нахождение пути к одной конкретной конечной точке. Может работать с графами с отрицательными весами при использовании подходящей эвристики. Основная причина почему A\* на моём примере работает медленнее в том, что используюется не вектор для хранения вершин а словарь. Альтернативным решением было бы закодировать каждую вершину индексом и хранить соответствия в словаре. Это позволило бы эффективнее производить обход и сэкономило бы память.

**Приложения:**

ПРИЛОЖЕНИЕ

Исходный код файла main.cpp

#include "graph.hpp"

#include "utils.hpp"

const std::string GRAPH\_FILE = "spb\_graph.txt";

const double START\_LONGITUDE = 30.372067;

const double START\_LATITUDE = 59.928577;

const double DEST\_LONGITUDE = 30.310001;

const double DEST\_LATITUDE = 59.956423;

const std::string ALGO\_BFS = "BFS";

const std::string ALGO\_DFS = "DFS";

const std::string ALGO\_DIJKSTRA = "Dijkstra";

const std::string ALGO\_A\_STAR = "A\*";

std::vector<Node\*> find\_path\_using\_algorithm(Graph& graph, Node\* start\_node, Node\* dest\_node, const std::string& algorithm\_name, const bool print) {

const auto start\_time = std::chrono::high\_resolution\_clock::now();

std::vector<Node\*> path;

if (algorithm\_name == ALGO\_BFS) {

path = graph.bfs(start\_node, dest\_node);

}

else if (algorithm\_name == ALGO\_DFS) {

path = graph.dfs(start\_node, dest\_node);

}

else if (algorithm\_name == ALGO\_DIJKSTRA) {

path = graph.dijkstra(start\_node, dest\_node);

}

else if (algorithm\_name == ALGO\_A\_STAR) {

path = graph.a\_star(start\_node, dest\_node);

}

else {

std::cerr << "Unknown algorithm: " << algorithm\_name << "\n";

return {};

}

const auto end\_time = std::chrono::high\_resolution\_clock::now();

print\_path\_info(algorithm\_name, path, end\_time - start\_time, print);

return path;

}

int main() {

Graph graph;

build\_graph\_from\_file(GRAPH\_FILE, graph);

Node\* start\_node = graph.get\_nearest\_node(START\_LATITUDE, START\_LONGITUDE);

Node\* dest\_node = graph.get\_nearest\_node(DEST\_LATITUDE, DEST\_LONGITUDE);

find\_path\_using\_algorithm(graph, start\_node, dest\_node, ALGO\_BFS, false);

find\_path\_using\_algorithm(graph, start\_node, dest\_node, ALGO\_DFS, false);

find\_path\_using\_algorithm(graph, start\_node, dest\_node, ALGO\_DIJKSTRA, false);

find\_path\_using\_algorithm(graph, start\_node, dest\_node, ALGO\_A\_STAR, false);

return 0;

}

Исходный код файла utils.hpp

#pragma once

#include <chrono>

#include "graph.hpp"

bool parse\_node\_coordinates(const std::string& str, double& longitude, double& latitude) {

return sscanf\_s(str.c\_str(), "%lf,%lf", &longitude, &latitude) == 2;

}

bool parse\_edges(const std::string& str, double& longitude, double& latitude, double& weight) {

return sscanf\_s(str.c\_str(), "%lf,%lf,%lf", &longitude, &latitude, &weight) == 3;

}

void build\_graph\_from\_file(const std::string& filename, Graph& graph) {

std::ifstream file(filename);

if (!file.is\_open()) {

throw std::runtime\_error("Could not open file: " + filename);

}

std::string line;

while (std::getline(file, line)) {

std::stringstream ss(line);

std::string node\_info;

if (!std::getline(ss, node\_info, ':')) {

throw std::runtime\_error("Invalid line format: " + line);

}

double start\_node\_longitude;

double start\_node\_latitude;

if (parse\_node\_coordinates(node\_info, start\_node\_longitude, start\_node\_latitude)) {

graph.add\_node(start\_node\_longitude, start\_node\_latitude);

}

else

throw std::runtime\_error("Invalid node format in line: " + line);

std::string neighbors\_info;

while (std::getline(ss, neighbors\_info, ';')) {

double dest\_node\_longitude;

double dest\_node\_latitude;

double edge\_weight;

if (parse\_edges(neighbors\_info, dest\_node\_longitude, dest\_node\_latitude, edge\_weight)) {

graph.add\_node(dest\_node\_longitude, dest\_node\_latitude);

graph.connect\_nodes(start\_node\_longitude, start\_node\_latitude, dest\_node\_longitude, dest\_node\_latitude, edge\_weight, false);

}

else

throw std::runtime\_error("Invalid neighbor format in line: " + line);

}

}

}

void print\_path(const std::vector<Node\*>& path) {

if (path.empty()) {

std::cout << "No path found.\n";

return;

}

for (Node\* node : path) {

std::cout << "[" << node->longitude << "," << node->latitude << "] -> ";

}

std::cout << "END\n";

}

void print\_path\_info(const std::string& algorithm\_name, const std::vector<Node\*>& path, const std::chrono::duration<double>& duration, const bool print) {

std::cout << algorithm\_name << ": \n";

if (print)

print\_path(path);

std::cout << " " << algorithm\_name << " Path Length: " << path.size() << "\n";

std::cout << " " << "Time " << algorithm\_name << ": " << duration.count() << " seconds\n";

}

Исходный код файла graph.hpp

#pragma once

#include <iostream>

#include <fstream>

#include <sstream>

#include <memory>

#include <string>

#include <vector>

#include <unordered\_map>

#include <unordered\_set>

#include <deque>

#include <queue>

struct HashPair {

template <typename T1, typename T2>

std::size\_t operator() (const std::pair<T1, T2>& p) const {

size\_t hash1 = std::hash<T1>{}(p.first);

size\_t hash2 = std::hash<T2>{}(p.second);

return hash1 ^ (hash2 + 0x9e3779b9 + (hash1 << 6) + (hash1 >> 2));

}

};

struct Node {

double longitude;

double latitude;

std::vector<std::pair<Node\*, double>> edges;

Node(double longitude, double latitude)

: longitude(longitude), latitude(latitude) {

}

void add\_edge(Node\* neighbor, double weight) {

edges.emplace\_back(neighbor, weight);

}

};

struct QueueNode {

double weight;

Node\* node;

QueueNode(double weight, Node\* node) : weight(weight), node(node) {}

bool operator>(const QueueNode& other) const {

return weight > other.weight;

}

};

class Graph {

public:

void add\_node(const double longitude, const double latitude);

void connect\_nodes(double longitude1, double latitude1, double longitude2, double latitude2, double weight, const bool orientation);

std::vector<Node\*> bfs(Node\* start\_node, Node\* dest\_node);

std::vector<Node\*> dfs(Node\* start\_node, Node\* dest\_node);

std::vector<Node\*> dijkstra(Node\* start\_node, Node\* dest\_node);

std::vector<Node\*> a\_star(Node\* start\_node, Node\* dest\_node);

Node\* get\_nearest\_node(const double latitude, const double longitude);

private:

std::unordered\_map<std::pair<double, double>, std::unique\_ptr<Node>, HashPair> nodes;

double a\_star\_heuristic(const Node\* a, const Node\* b);

std::vector<Node\*> construct\_path(Node\* current\_node, const std::unordered\_map<Node\*, Node\*>& parent);

};

Исходный код файла graph.cpp

#include "graph.hpp"

void Graph::add\_node(const double longitude, const double latitude) {

nodes.emplace(std::make\_pair(latitude, longitude), std::make\_unique<Node>(longitude, latitude));

}

void Graph::connect\_nodes(double start\_longitude, double start\_latitude, double dest\_longitude, double dest\_latitude, double weight, const bool orientation) {

auto start\_node\_it = nodes.find({ start\_latitude, start\_longitude });

auto dest\_node\_it = nodes.find({ dest\_latitude, dest\_longitude });

if (start\_node\_it == nodes.end() || dest\_node\_it == nodes.end()) {

return;

}

Node\* start\_node = start\_node\_it->second.get();

Node\* dest\_node = dest\_node\_it->second.get();

start\_node->add\_edge(dest\_node, weight);

if (!orientation)

dest\_node->add\_edge(start\_node, weight);

}

std::vector<Node\*> Graph::bfs(Node\* start\_node, Node\* dest\_node) {

std::deque<Node\*> queue{ start\_node }; // Queue for BFS

std::unordered\_set<Node\*> visited{ start\_node }; // Set to track visited nodes to avoid processing the same node multiple times

std::unordered\_map<Node\*, Node\*> parent; // Map to store the parent of each node for path reconstruction

parent[start\_node] = nullptr;

// Perform BFS until there are no more nodes in the queue

while (!queue.empty()) {

Node\* current\_node = queue.front();

queue.pop\_front();

// Check if we have reached the destination node

if (current\_node == dest\_node) {

return construct\_path(current\_node, parent);

}

// Explore all adjacent nodes (edges) of the current node

for (const auto& edge : current\_node->edges) {

if (visited.insert(edge.first).second) {

parent[edge.first] = current\_node;

queue.push\_back(edge.first);

}

}

}

return {};

}

std::vector<Node\*> Graph::dfs(Node\* start\_node, Node\* dest\_node) {

std::deque<Node\*> stack{ start\_node }; // Stack for DFS

std::unordered\_set<Node\*> visited{ start\_node }; // Set to track visited nodes to avoid processing the same node multiple times

std::unordered\_map<Node\*, Node\*> parent; // Map to store the parent of each node for path reconstruction

parent[start\_node] = nullptr;

// Perform DFS until there are no more nodes in the stack

while (!stack.empty()) {

Node\* current\_node = stack.back();

stack.pop\_back();

// Check if we have reached the destination node

if (current\_node == dest\_node) {

return construct\_path(current\_node, parent);

}

// Explore all adjacent nodes (edges) of the current node

for (const auto& edge : current\_node->edges) {

if (visited.insert(edge.first).second) {

parent[edge.first] = current\_node;

stack.push\_back(edge.first);

}

}

}

return {};

}

std::vector<Node\*> Graph::dijkstra(Node\* start\_node, Node\* dest\_node) {

std::priority\_queue<QueueNode, std::vector<QueueNode>, std::greater<>> priority\_queue; // Priority queue for dijkstra

std::unordered\_map<Node\*, double> path\_cost;

path\_cost.reserve(nodes.size());

for (const auto& node : nodes) {

path\_cost[node.second.get()] = INFINITY;

}

std::unordered\_map<Node\*, Node\*> parent; // Map to store the parent of each node

parent[start\_node] = nullptr;

path\_cost[start\_node] = 0;

priority\_queue.emplace(0, start\_node);

// Perform dijkstra until there are no more nodes in the priority\_queue

while (!priority\_queue.empty()) {

auto current = priority\_queue.top();

priority\_queue.pop();

if (current.weight <= path\_cost[current.node]) {

// Check if we have reached the destination node

if (current.node == dest\_node) {

return construct\_path(current.node, parent);

};

for (const auto& edge : current.node->edges) {

double weights\_sum = current.weight + edge.second;

if (weights\_sum < path\_cost[edge.first]) {

path\_cost[edge.first] = weights\_sum;

priority\_queue.emplace(weights\_sum, edge.first);

parent[edge.first] = current.node;

}

}

}

}

return {};

}

std::vector<Node\*> Graph::a\_star(Node\* start\_node, Node\* dest\_node) {

std::priority\_queue<QueueNode, std::vector<QueueNode>, std::greater<>> priority\_queue; // Priority queue for A\*

std::unordered\_map<Node\*, double> path\_cost;

std::unordered\_map<Node\*, double> total\_estimated\_cost;

path\_cost.reserve(nodes.size());

total\_estimated\_cost.reserve(nodes.size());

for (const auto& node\_p : nodes) {

Node\* node = node\_p.second.get();

path\_cost[node] = INFINITY;

total\_estimated\_cost[node] = INFINITY;

}

std::unordered\_map<Node\*, Node\*> parent;

parent[start\_node] = nullptr;

path\_cost[start\_node] = 0;

total\_estimated\_cost[start\_node] = a\_star\_heuristic(start\_node, dest\_node);

priority\_queue.emplace(total\_estimated\_cost[start\_node], start\_node);

// Perform A\* until there are no more nodes in the priority\_queue

while (!priority\_queue.empty()) {

Node\* current\_node = priority\_queue.top().node;

priority\_queue.pop();

// Check if we have reached the destination node

if (current\_node == dest\_node) {

return construct\_path(current\_node, parent);

};

for (const auto& edge : current\_node->edges) {

double weights\_sum = path\_cost[current\_node] + edge.second;

if (weights\_sum < path\_cost[edge.first]) {

parent[edge.first] = current\_node;

path\_cost[edge.first] = weights\_sum;

total\_estimated\_cost[edge.first] = weights\_sum + a\_star\_heuristic(edge.first, dest\_node);

priority\_queue.emplace(total\_estimated\_cost[edge.first], edge.first);

}

}

}

return {};

}

Node\* Graph::get\_nearest\_node(const double latitude, const double longitude) {

double min\_distance\_squared = std::numeric\_limits<double>::max();

Node\* closest\_node = nullptr;

for (const auto& it : nodes) {

Node\* node = it.second.get();

double delta\_lat = node->latitude - latitude;

double delta\_lon = node->longitude - longitude;

double distance\_squared = delta\_lat \* delta\_lat + delta\_lon \* delta\_lon;

if (distance\_squared == 0)

return node;

if (distance\_squared < min\_distance\_squared) {

closest\_node = node;

min\_distance\_squared = distance\_squared;

}

}

return closest\_node;

}

double Graph::a\_star\_heuristic(const Node\* node\_1, const Node\* node\_2) {

const double lat\_diff = node\_1->latitude - node\_2->latitude;

const double lon\_diff = node\_1->longitude - node\_2->longitude;

return lat\_diff \* lat\_diff + lon\_diff \* lon\_diff;

}

std::vector<Node\*> Graph::construct\_path(Node\* current\_node, const std::unordered\_map<Node\*, Node\*>& parent) {

std::vector<Node\*> path;

// Backtrack to construct the path

for (Node\* node = current\_node; node != nullptr; node = parent.at(node)) {

path.push\_back(node);

}

// Reverse to get the path from start to destination

std::reverse(path.begin(), path.end());

return path;

}