Федеральное государственное автономное образовательное учреждение

высшего образования

«Национальный исследовательский университет ИТМО»

Отчёт по лабораторной работе № 8

«Z_апокалипсис»

Выполнила работу

Мижа Виктория

Академическая группа № j3112

Принято

Практикант, Максим Дунаев

```cpp
#include <iostream>
#include <stack>
#include <queue>
#include <unordered_map>
#include <vector>
#include <cmath>
#include <limits>
#include <string>
#include <sstream>
#include <fstream>
#include <iomanip>
#include <algorithm>

#define double long double
#define endl '\n'

using namespace std;

const double INF = numeric_limits<double>::infinity(); // Константа для "бесконечности"

// Хэш для unordered_map с парами
struct PairHash {
    template <class T1, class T2>
    size_t operator()(const pair<T1, T2>& p) const {
        auto h1 = hash<T1>{}(p.first);
        auto h2 = hash<T2>{}(p.second);
        return h1 ^ h2;
    }
};

// Эвристики: Евклидова и Манхэттенская
double euclideanDistance(double lon1, double lat1, double lon2, double lat2) {
    return sqrt(pow(lon2 - lon1, 2) + pow(lat2 - lat1, 2)); // Сложность O(1), память O(1)
}

double manhattan(double lon1, double lat1, double lon2, double lat2) {
    return abs(lon1 - lon2) + abs(lat1 - lat2); // Сложность O(1), память O(1)
}

// Узел графа
struct Node {
    int id = 0;
    double longitude, latitude;
    int state = 0; // Для поиска
    vector<pair<Node*, double>> neighbors;

    Node() = default;
    Node(double lon, double lat) : longitude(lon), latitude(lat) {} // Сложность O(1), память O(1)
};

// Компаратор для очереди с приоритетом
struct DistanceComparator {
    bool operator()(const pair<Node*, double>& a, const pair<Node*, double>& b) {
        return a.second > b.second; // Сравниваем только по значению расстояния
    } // Сложность O(1), память O(1)
};

// Граф
struct Graph {
    int nodeCounter = 0;
    vector<Node*> nodesList;
    unordered_map<pair<double, double>, int, PairHash> existingNodes; // Память O(V)
    unordered_map<int, double> distanceMap; // Память O(V)
    unordered_map<int, int> pathMap; // Память O(V)

    void parseNode(stringstream& ss) {
        string s;
        getline(ss, s, ',');
        double lon = stod(s);
        getline(ss, s, ':');
        double lat = stod(s);
        Node* currentNode = getNode(lon, lat);
```

```
        while (true) {
            try {
                getline(ss, s, ',');
                double neighborLon = stod(s);
                getline(ss, s, ',');
                double neighborLat = stod(s);
                Node* neighborNode = getNode(neighborLon, neighborLat);
                getline(ss, s, ';');
                double weight = stod(s);
                currentNode->neighbors.push_back(make_pair(neighborNode, weight));
            }
            catch (const invalid_argument& e) {
                return; // Если прочитали всю строку
            }
        }
    } // Сложность O(V + E), память O(V)

    void addNode(double& lon, double& lat) {
        Node* newNode = new Node(lon, lat);
        newNode->id = nodeCounter++;
        nodesList.push_back(newNode); // Сложность O(1), память O(1)
    }

    Node* getNode(double& lon, double& lat) {
        if (existingNodes.count(make_pair(lon, lat)) != 0) {
            return nodesList[existingNodes[make_pair(lon, lat)]];
        }
        addNode(lon, lat);
        existingNodes[make_pair(lon, lat)] = nodesList.back()->id;
        return nodesList.back(); // Сложность O(1), память O(1)
    }

    Node* getClosestNode(double& lon, double& lat) {
        double minDistance = INF;
        Node* closestNode = nullptr;

        for (Node* node : nodesList) {
            double currentDistance = pow((node->latitude - lat), 2) + pow((node->longitude - lon),
2);
            if (currentDistance < minDistance) {
                closestNode = node;
                minDistance = currentDistance;
            }
        }
        return closestNode; // Сложность O(V), память O(1)
    }

    void resetStates() {
        for (Node* node : this->nodesList) {
            node->state = 0; // Сложность O(V), память O(1)
        }
    }
    // BFS Algorithm
    double breadthFirstSearch(double lonStart, double latStart, double lonEnd, double latEnd) {
        resetStates();
        Node* start = getClosestNode(lonStart, latStart);
        Node* end = getClosestNode(lonEnd, latEnd);

        for (Node* node : nodesList)
            distanceMap[node->id] = (node == start) ? 0 : INF;

        queue<Node*> searchQueue;
        searchQueue.push(start);

        int roadCount = 0;

        while (!searchQueue.empty()) {
            Node* currentNode = searchQueue.front();
            searchQueue.pop();
            currentNode->state = 1;

            for (auto neighbor : currentNode->neighbors) {
                if (neighbor.first->state == 0) {
                    neighbor.first->state = 1;
                    roadCount++;
                    distanceMap[neighbor.first->id] = distanceMap[currentNode->id] +
neighbor.second;
                    searchQueue.push(neighbor.first);

                    if (neighbor.first->id == end->id) {
```

```
                        cout << "Алгоритм прошёл дорог: " << roadCount << endl;
                        cout << "Пройденное расстояние: ";
                        return distanceMap[end->id];
                    }
                }
            }
        }

        cout << "Пути не существует. Расстояние: ";
        return -1; // BFS: сложность O(V + E), пространственная сложность O(V)
    }

    // DFS Algorithm
    double depthFirstSearch(double lonStart, double latStart, double lonEnd, double latEnd) {
        resetStates();

        Node* start = getClosestNode(lonStart, latStart);
        Node* end = getClosestNode(lonEnd, latEnd);

        stack<Node*> searchStack;

        for (Node* node : nodesList)
            distanceMap[node->id] = (node == start) ? 0 : INF;

        searchStack.push(start);

        int roadCount = 0;

        while (!searchStack.empty()) {
            Node* currentNode = searchStack.top();
            searchStack.pop();
            currentNode->state = 1;

            for (auto neighbor : currentNode->neighbors) {
                roadCount++;
                if (neighbor.first->state == 0) {
                    neighbor.first->state = 1;
                    distanceMap[neighbor.first->id] = distanceMap[currentNode->id] +
neighbor.second;
                    searchStack.push(neighbor.first);

                    if (neighbor.first->id == end->id) {
                        cout << "Алгоритм прошёл дорог: " << roadCount << endl;
                        cout << "Пройденное расстояние: ";
                        return distanceMap[end->id];
                    }
                }
            }
        }

        cout << "Пути не существует. Расстояние: ";
        return -1; // DFS: сложность O(V + E), пространственная сложность O(V)
    }

    vector<int> dijkstra(double lonStart, double latStart, double lonEnd, double latEnd) {
        Node* start = getClosestNode(lonStart, latStart);
        Node* end = getClosestNode(lonEnd, latEnd);

        for (Node* node : nodesList) {
            distanceMap[node->id] = (node == start) ? 0 : INF;
        }

        priority_queue<pair<Node*, double>, vector<pair<Node*, double>>, DistanceComparator> pq;
        pq.push(make_pair(start, 0));

        bool pathExists = false;

        while (!pq.empty()) {
            pair<Node*, double> current = pq.top();
            pq.pop();

            if (current.first->id == end->id) {
                pathExists = true;
                break;
            }
```

```
            if (current.second > distanceMap[current.first->id]) {
                continue;
            }
            for (pair<Node*, double> neighbor : current.first->neighbors) {
                double newDistance = distanceMap[current.first->id] + neighbor.second;
                if (newDistance < distanceMap[neighbor.first->id]) {
                    distanceMap[neighbor.first->id] = newDistance;
                    pathMap[neighbor.first->id] = current.first->id;
                    pq.push(make_pair(neighbor.first, newDistance));
                }
            }
        }

        if (!pathExists) {
            cout << "Не существует такого пути \n";
            return {};
        }

        vector<int> pathResult;
        int index = end->id;
        while (index != start->id) {
            pathResult.push_back(index);
            index = pathMap[index];
        }

        reverse(pathResult.begin(), pathResult.end());
        cout << "Расстояние: " << distanceMap[end->id] << endl;
        cout << "Пройденный путь: ";
        for (int i : pathResult) {
            cout << i + 1 << ' ';
        }

        return pathResult; // Дейкстра: сложность O((V + E) * log V), пространственная сложность
O(V)
    }

    vector<int> A_star(double lonStart, double latStart, double lonEnd, double latEnd) {
        Node* start = getClosestNode(lonStart, latStart);
        Node* end = getClosestNode(lonEnd, latEnd);

        for (Node* node : nodesList) {
            distanceMap[node->id] = (node == start) ? 0 : INF;
        }

        vector<double> heuristics(nodesList.size(), 0);

        priority_queue<pair<Node*, double>, vector<pair<Node*, double>>, DistanceComparator> pq;
        pq.push(make_pair(start, 0));

        bool pathExists = false;

        while (!pq.empty()) {
            pair<Node*, double> current = pq.top();
            pq.pop();

            if (current.first->id == end->id) {
                pathExists = true;
                break;
            }

            if (current.second - heuristics[current.first->id] > distanceMap[current.first->id]) {
                continue;
            }

            for (pair<Node*, double> neighbor : current.first->neighbors) {
                // Используем эвристику Манхэттена
                double heuristicValue = manhattan(current.first->longitude, current.first-
>latitude,
                    neighbor.first->longitude, neighbor.first->latitude);

                // Расчет новой дистанции
                double newDistance = distanceMap[current.first->id] + neighbor.second +
heuristicValue;

                if (newDistance < distanceMap[neighbor.first->id]) {
                    distanceMap[neighbor.first->id] = newDistance;
                    pathMap[neighbor.first->id] = current.first->id;
                    heuristics[neighbor.first->id] = heuristicValue;
                    pq.push(make_pair(neighbor.first, newDistance));
                }
```

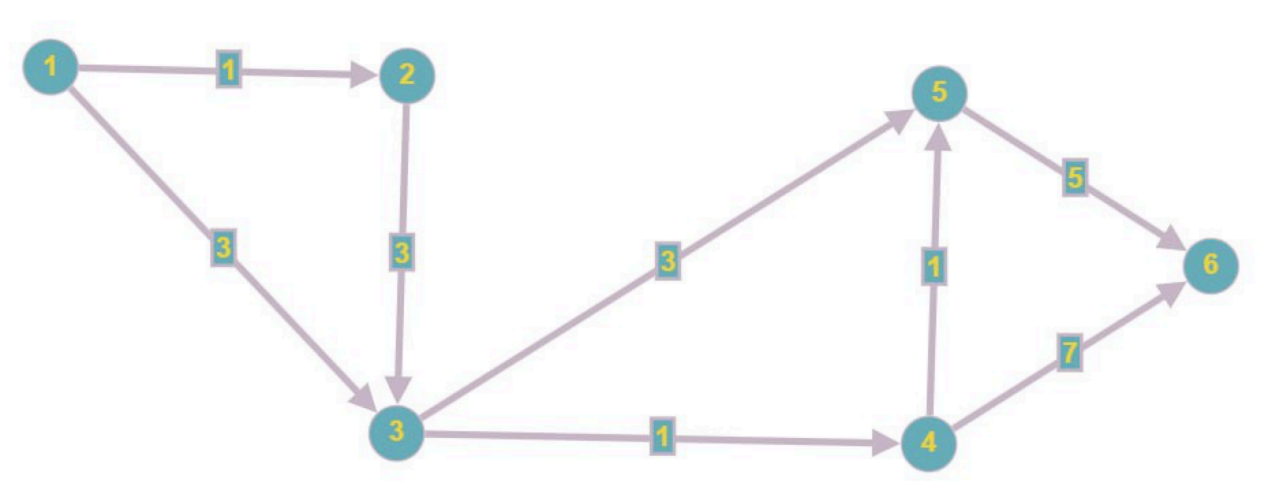
```
            }
        }


        if (!pathExists) {
            cout << "Не существует такого пути \n";
            return {};
        }

        vector<int> pathResult;
        int indexPathResult = end->id;
        while (indexPathResult != start->id) {
            pathResult.push_back(indexPathResult);
            indexPathResult = pathMap[indexPathResult];
        }

        reverse(pathResult.begin(), pathResult.end());
        cout << "Расстояние: " << distanceMap[end->id] << endl;
        cout << "Пройденный путь: ";
        for (int i : pathResult)
            cout << i + 1 << ' ';

        return pathResult;
    }

};

// Функция записи данных в граф
void write_graph(Graph& g, fstream& file) {
    string s = "";
    int cnt = 0;
    while (file >> s) {
        stringstream ss(s);
        g.parseNode(ss);
        cnt++;
        if (cnt % 1000 == 0) {
            cout << "Считано вершин: " << cnt << endl;
        }
    }
    cout << "Всего вершин в графе: " << g.nodesList.size() << '\n';
}

// Функция чтения вектора
void read_vector(string s, vector<int>& vec) {
    int el;
    stringstream ss(s);
    while (ss >> el) {
        vec.push_back(el);
    }
}
// Вспомогательная функция для сравнения векторов
bool compare_vectors(vector<int>& vecA, vector<int>& vecB) {
    if (vecA.size() != vecB.size()) {
        return false;
    };
    for (int i = 0; i < vecA.size(); i++) {
        if (vecA[i] + 1 != vecB[i]) {
            return false;
        }
    }
    return true;
}

void testGraph(const string& graphFilePath, const string& expectedFilePath) {
    fstream graphFile(graphFilePath);
    fstream expectedFile(expectedFilePath);

    if (!graphFile.is_open() || !expectedFile.is_open()) {
        cerr << "Ошибка при открытии файлов." << endl;
        return;
    }

    Graph g; // Создание экземпляра графа
    write_graph(g, graphFile); // Чтение графа из файла

    double lon1, lat1, lon2, lat2;
    expectedFile >> lon1 >> lat1 >> lon2 >> lat2; // Чтение координат

    double dfs_expected, bfs_expected;
    expectedFile >> dfs_expected >> bfs_expected; // Ожидаемые результаты
```

```
    vector<int> dijkstra_expected;
    vector<int> a_star_expected;

    string line;
    while (getline(expectedFile, line)) {
        read_vector(line, dijkstra_expected);
        read_vector(line, a_star_expected);
    }

    // Запуск алгоритмов
    double dfs_result = g.depthFirstSearch(lon1, lat1, lon2, lat2);
    double bfs_result = g.breadthFirstSearch(lon1, lat1, lon2, lat2);

    vector<int> dijkstra_result = g.dijkstra(lon1, lat1, lon2, lat2);
    vector<int> a_star_result = g.A_star(lon1, lat1, lon2, lat2);

    // Проверка результатов
    bool condition = (bfs_result == bfs_expected) &&
        (dfs_result == dfs_expected) &&
        compare_vectors(dijkstra_result, dijkstra_expected) &&
        compare_vectors(a_star_result, a_star_expected);

    if (condition) {
        cout << "Тест пройден успешно!" << endl;
    }
    else {
        cerr << "Тест не пройден!" << endl;
    }

    graphFile.close();
    expectedFile.close();
}

int main() {
    // Для тестов:
    //testGraph("path_test.txt","path_expected.txt");
    setlocale(LC_ALL, "rus");

    cout << fixed << setprecision(20);

    Graph g;

    fstream fs;

    fs.open("/Users/vitamija/Desktop/hi_graf/spb_graph.txt");

    write_graph(g, fs);

    fs.close();

    while (true) {
        string action;

        cout << "Введите 0), чтобы выйти из приложения\n";
        cout << "Введите 1), чтобы найти ближайшую вершину\n";
        cout << "Введите 2), чтобы найти путь с помощью DFS\n";
        cout << "Введите 3), чтобы найти путь с помощью BFS\n";
        cout << "Введите 4), чтобы найти путь с помощью Дейкстры\n";
        cout << "Введите 5), чтобы найти путь с помощью A*\n";

        cin >> action;

        clock_t startTime = clock();

        double lonStart, latStart;
        double lonEnd, latEnd;

        if (action == "0") {
            return 0;
        }
        else if (action == "1") {
            cout << "Введите широту и долготу: ";
            cin >> lonStart >> latStart;

            startTime = clock();

            Node* closest = g.getClosestNode(lonStart, latStart);

            cout << "Итоговая широта и долгота: ";
```

```
            cout << closest->longitude << ' ' << closest->latitude << '\n';
        }
        else if (action == "2") {
            cout << "Введите широту и долготу для стартовой вершины поиска DFS: ";
            cin >> lonStart >> latStart;

            cout << "Введите широту и долготу для конечной вершины: ";
            cin >> lonEnd >> latEnd;

            startTime = clock();

            cout << g.depthFirstSearch(lonStart, latStart, lonEnd, latEnd) << endl;
        }
        else if (action == "3") {
            cout << "Введите широту и долготу для стартовой вершины поиска BFS: ";
            cin >> lonStart >> latStart;

            cout << "Введите широту и долготу для конечной вершины: ";
            cin >> lonEnd >> latEnd;

            startTime = clock();

            cout << g.breadthFirstSearch(lonStart, latStart, lonEnd, latEnd) << endl;
        }
        else if (action == "4") {
            cout << "Введите широту и долготу для стартовой вершины поиска Дейкстры: ";
            cin >> lonStart >> latStart;
            cout << "Введите широту и долготу для конечной вершины: ";
            cin >> lonEnd >> latEnd;

            startTime = clock();

            g.dijkstra(lonStart, latStart, lonEnd, latEnd);

            cout << endl;
        }
        else if (action == "5") { // Вызов алгоритма A*
            cout << "Введите широту и долготу для стартовой вершины поиска A*: ";
            cin >> lonStart >> latStart;

            cout << "Введите широту и долготу для конечной вершины: ";
            cin >> lonEnd >> latEnd;

            startTime = clock();

            g.A_star(lonStart, latStart, lonEnd, latEnd);

            cout << endl;
        }
        clock_t end = clock();
        double duration = (end - startTime) * 1000 / CLOCKS_PER_SEC;
        cout << "Время выполнения алгоритма: " << duration << " милисекунд \n";
    }

    return 0;
}
```

```
2
Введите широту и долготу для стартовой вершины поиска DFS: 30.296019 59.94393 30.467719 59.920543
Введите широту и долготу для конечной вершины: Алгоритм прошёл дорог: 89711
Пройденное расстояние: 6.98426969902042227289
Время выполнения алгоритма: 129.00000000000000000000 милисекунд
```

Граф для тестов



Цель работы

Основная задача — разработать и реализовать алгоритмы BFS, DFS, Дейкстры и A* для изучения работы с графами. В качестве исходного графа используется дорожная сеть города Санкт-Петербург.

Поиск в глубину (DFS):

Описание:
Алгоритм DFS начинает обход с заданной вершины, углубляясь в граф до тех пор, пока не достигнет конца пути или тупика. При обнаружении тупика алгоритм возвращается назад и продолжает исследовать другие возможные маршруты.

Сложность:

- **Временная сложность:** $O(V+E)$.
- **Пространственная сложность:** $O(V)$ из-за использования стека для хранения текущего пути.

Применение:
DFS применяется для задач, требующих полного обхода графа, таких как:

- Решение головоломок (например, лабиринтов).
- Анализ связности графов.
- Поиск мостов и точек сочленения в графах.

```
2
Введите широту и долготу для стартовой вершины поиска DFS: 30.296019 59.94393 30.467719 59.920543
Введите широту и долготу для конечной вершины: Алгоритм прошёл дорог: 89711
Пройденное расстояние: 6.98426969902042227289
Время выполнения алгоритма: 129.00000000000000000000 милисекунд
```

Теоретическая подготовка

Поиск в ширину (BFS):

Описание:
Алгоритм BFS предназначен для обхода графов. Он исследует вершины слоями, начиная с корневой, затем переходя к её соседям и далее к соседям следующего уровня, пока не найдётся целевая вершина.

Сложность:

- **Временная сложность:** $O(V+E)$, где:
V — количество вершин, E — количество рёбер.
- **Пространственная сложность:** $O(V)$ из-за использования очереди для хранения вершин.

Применение:
BFS активно применяется для поиска кратчайшего пути в графах без весов, в задачах маршрутизации и анализа связей в социальных сетях.

```
3
Введите широту и долготу для стартовой вершины поиска BFS: 30.296019 59.94393 30.467719 59.920543
Введите широту и долготу для конечной вершины: Алгоритм прошёл дорог: 202917
Пройденное расстояние: 0.82146914454336150690
Время выполнения алгоритма: 129.00000000000000000000 милисекунд
```

Алгоритм Дейкстры:

Описание:
Дейкстра — это алгоритм для нахождения кратчайшего пути из одной вершины во все остальные в взвешенном графе с неотрицательными весами рёбер. Он использует жадный подход, выбирая на каждом шаге вершину с минимальным расстоянием и обновляя пути к её соседям.

Сложность:

- Временная сложность зависит от используемой структуры данных:
  - В лабораторной работе применяется реализация с приоритетной очередью, обеспечивающая сложность $O((V+E)\cdot \log V)$.

Применение:
Алгоритм Дейкстры широко используется в:

- Навигационных системах.
- Маршрутизации в компьютерных сетях.
- Оптимизации транспортных потоков.

```
4
Введите широту и долготу для стартовой вершины поиска Дейкстры: 30.296019 59.94393 30.467719 59.920543
Введите широту и долготу для конечной вершины: Расстояние: 0.18667037639778027991
Пройденный путь: 189008 189009 177684 177708 177681 49863 49864 49866 49869 49872 188453 177686 188454 177
150069 150068 150189 218031 150178 150179 15409 22958 22960 22963 120883 120885 150091 150095 120978 12098
150105 150101 150082 150103 150456 150459 150460 150463 129346 150469 23060 23062 150494 150496 150499 578
8483 11736 11737 11739 51811 150719 19078 60643 60644 60646 60648 60649 60650 19204 19206 27724 27726 1885
305042 242849 242851 242853 150850 226548 226549 226551 167517 167518 35899 167497 167523 35902 167498 488
Время выполнения алгоритма: 330.00000000000000000000 милисекунд
```

Алгоритм A:*

Описание:
Алгоритм A* расширяет Дейкстру, добавляя эвристическую функцию, которая оценивает приближенную стоимость пути до цели. Он сочетает свойства BFS и Дейкстры, обеспечивая более эффективный поиск путей за счёт использования информации о целевой вершине.

Сложность:

- Временная сложность зависит от качества эвристики.
- В худшем случае сложность совпадает с Дейкстрой — $O((V+E)\cdot \log V)$.
- При удачном выборе эвристики производительность может значительно улучшиться.

Применение:
Алгоритм A* применяется в:

- Системах навигации и прокладки маршрутов.
- Игровом ИИ для поиска оптимальных стратегий.
- Планировании движения роботов и автономных систем.

```
5
Введите широту и долготу для стартовой вершины поиска A*: 30.296019 59.94393 30.467719 59.920543
Введите широту и долготу для конечной вершины: Расстояние: 0.41454267639777808707
Пройденный путь: 189008 189009 177684 177708 177681 49863 49864 49866 49869 49872 188453 177686 188454
150069 150068 150189 218031 150178 150179 15409 22958 22960 22963 120883 120885 150091 150095 120978 1
150105 150101 150082 150103 150456 150459 150460 150463 129346 150469 23060 23062 150494 150496 150499
8483 11736 11737 11739 51811 150719 19078 60643 60644 60646 60648 60649 60650 19204 19206 27724 27726
305042 242849 242851 242853 150850 226548 226549 226551 167517 167518 35899 167497 167523 35902 167498
Время выполнения алгоритма: 283.00000000000000000000 милисекунд
```

Вывод

Во время выполнения лабораторной работы были реализованы и протестированы четыре популярных алгоритма поиска путей на графе дорог Санкт-Петербурга: A*, Дейкстра, BFS и DFS. Каждый из них показал свои сильные и слабые стороны в зависимости от задачи.

Алгоритмы BFS и DFS сильно зависят от расположения вершин в графе. Если вершины находятся далеко друг от друга, DFS может тратить больше времени на обход, а BFS обходит граф более равномерно. Дейкстра, в свою очередь, хорошо работает с взвешенными графами, но может уступать по скорости алгоритму A*, если для последнего выбрана подходящая эвристика.

A* действительно выделился на фоне остальных. Благодаря эвристике он смог быстрее находить пути, направляясь сразу к цели, вместо того чтобы проверять все вершины подряд. В нашем случае правильно подобранная манхэттенская эвристика дала ему ощутимое преимущество — A* оказался быстрее Дейкстры.

В итоге все четыре алгоритма доказали свою полезность в разных сценариях и могут применяться для решения широкого круга задач, связанных с графами и поиском оптимальных маршрутов.