ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 8

«Графы. Поиск кратчайшего пути»

Выполнила работу

Чунихина Валерия

ИСУ: 468026

Академическая группа J3114

Принято

Дунаев Максим Владимирович

Санкт-Петербург

2024

Содержание

[Содержание 2](#_Toc177380279)

[Введение 3](#_Toc177380280)

[Теоретическая подготовка 4](#_Toc177380281)

[Реализация 5](#_Toc177380282)

[Экспериментальная часть 6](#_Toc177380283)

[Заключение 7](#_Toc177380284)

[Приложение 8](#_Toc177380285)

**ВВЕДЕНИЕ**

**Цель работы:** реализовать алгоритмы BFS, DFS, Дейкстра и A\* для поиска кратчайшего пути от места проживания (Общежитие №2, ул. Ленсовета д. 23) до университета ИТМО.

**Задачи, которые необходимо решить:**

1. Реализовать алгоритмы BFS, DFS, Дейкстра и A\*.
2. Измерить время работы каждого алгоритма.
3. Сравнить и проанализировать результаты.

**ТЕОРЕТИЧЕСКАЯ ПОДГОТОВКА**

**Алгоритм BFS (поиск в ширину)** обходит граф, исследуя вершины по уровням, начиная с начальной. Он подходит для поиска кратчайшего пути в невзвешенных графах.

**Алгоритм DFS (поиск в глубину)** углубляется вдоль одного пути до тупика, затем возвращается и продолжает поиск в других направлениях.

**Алгоритм Дейкстры** находит кратчайшие пути от одной вершины ко всем остальным в графе с неотрицательными весами рёбер. Он выбирает вершину с минимальным расстоянием и обновляет значения для соседей, используя приоритетную очередь.

**Алгоритм A\* —** модификация Дейкстры, которая учитывает как текущую стоимость пути, так и эвристическую оценку расстояния до цели, что ускоряет поиск кратчайшего пути.

**4.** Используемые библиотеки:

<cmath> — предоставляет математические функции

<iostream> — используется для ввода и вывода данных в стандартные потоки

<map> — реализует ассоциативный массив (словари)

<vector>— предоставляет динамический массив

<fstream> — используется для работы с файлами

<limits> — содержит информацию о предельных значениях типов данных

<queue> — предоставляет контейнер для работы с очередями

<set> — реализует множество, которое хранит уникальные элементы в отсортированном порядке и обеспечивает быстрый доступ к ним.

<sstream> — позволяет работать со строками как с потоками

<algorithm> — содержит стандартные алгоритмы

<chrono> — предоставляет инструменты для работы с временем, включая измерение продолжительности выполнения кода.

<stack> — реализует структуру данных "стек"

<cassert> - для сравнения значений в тестах

**РЕАЛИЗАЦИЯ**

1. **Представление графа**

using Graph = map<int, vector<pair<int, double>>>; // O(V + E)  
using Coordinates = map<int, pair<double, double>>; // O(V)  
  
// Функция для парсинга графа из файла  
void parseGraph(const string& filename, Graph& graph, Coordinates& coordinates) {  
 ifstream file(filename);  
 if (!file.is\_open()) {  
 cerr << "Error opening file: " << filename << endl;  
 return;  
 }  
  
 map<pair<double, double>, int> coordinateToId; // V\*(8\*2 + 4) байта  
 int nodeId = 0; // 4 байта  
  
 auto getId = [&](double lon, double lat) -> int { // 4 байта  
 pair<double, double> coord = {lon, lat}; // V\*8\*2 байта  
 auto it = coordinateToId.find(coord);  
 if (it == coordinateToId.end()) {  
 coordinateToId[coord] = nodeId;  
 coordinates[nodeId] = coord;  
 return nodeId++;  
 }  
 return it->second;  
 };  
  
 string line; // O(1)  
 while (getline(file, line)) {  
 size\_t colon = line.find(":");  
 if (colon == string::*npos*) continue;  
  
 string parentNode = line.substr(0, colon); // O(m), m - длина строки родительского узла  
 string children = line.substr(colon + 1); // O(m), m - длина строки родительского узла  
  
 double lon1, lat1; // 8 \* 2 байт  
 sscanf(parentNode.c\_str(), "%lf,%lf", &lon1, &lat1);  
 int parentId = getId(lon1, lat1); // 4 байта  
  
 stringstream ss(children);  
 string child; // O(1)  
 while (getline(ss, child, ';')) {  
 double lon2, lat2, weight; // 8 \* 3 байта  
 if (sscanf(child.c\_str(), "%lf,%lf,%lf", &lon2, &lat2, &weight) == 3) {  
 int childId = getId(lon2, lat2); // 4 байта  
 graph[parentId].emplace\_back(childId, weight);  
 graph[childId].emplace\_back(parentId, weight);  
 }  
 }  
 }  
}

В **Graph** и **Coordinates** определяем структуры данных для хранения графа и координат узлов соответственно. Для **Graph** используетм **map** для хранения узлов и их соседей с весами, а в **Coordinates** храним координаты узлов.

Функция **parseGraph**:

Открываем файл и проверяем, успешно ли он открыт.

Создаём отображение **coordinateToId** для сопоставления координат с идентификаторами узлов.

Используем лямбда-функцию **getId** для получения идентификатора узла по его координатам. Если координаты не существуют, добавляем их в отображение.

Читаем файл построчно, разбиваем строки на родительские и дочерние узлы, парсим координаты и веса, и добавляем их в граф.

1. **Нахождение ближайшего узла**

int findNearestNode(const Coordinates& coordinates, double lon, double lat) {  
 int nearestNode = -1; // 4 байта  
 double minDistance = numeric\_limits<double>::*infinity*(); // 8 байт  
  
 // Проходим по всем координатам  
 for (const auto& [node, coord] : coordinates) {  
 // Вычисляем расстояние до текущей координаты  
 double distance = sqrt(pow(coord.first - lon, 2) + pow(coord.second - lat, 2)); // 8 байт  
 if (distance < minDistance) {  
 minDistance = distance;  
 nearestNode = node;  
 }  
 }  
 return nearestNode;  
}

Эта функция принимает координаты и ищет ближайший узел в графе.

Она инициализирует переменные для хранения ближайшего узла и минимального расстояния.

Затем проходит по всем узлам в coordinates, вычисляет расстояние до каждого узла и обновляет ближайший узел, если найдено меньшее расстояние.

Возвращает идентификатор ближайшего узла.

1. **Вывод пути и итоговой суммы**

double printPath(const map<int, int>& parent, int start, int goal, const Graph& graph) {  
 vector<int> path; // 4 \* len(path) байт  
 double weight = 0.0; // 8 байт  
  
 // Восстанавливаем путь от goal до start  
 for (int cur = goal; cur != start; cur = parent.at(cur)) { // 4 байта  
 path.push\_back(cur);  
 // Находим вес текущего ребра  
 for (const auto& [neighbor, w] : graph.at(parent.at(cur))) {  
 if (neighbor == cur) weight += w;  
 }  
 }  
 path.push\_back(start);  
  
 // Переворачиваем путь, чтобы получить его от start до goal  
 reverse(path.begin(), path.end());  
  
 // Печатаем итоговый путь  
 cout << "Final path: ";  
 for (const int node : path) {  
 cout << node << " ";  
 }  
 cout << endl;  
  
 cout << "Final weight: " << weight << endl;  
 return weight;  
}

Эта функция восстанавливает путь от целевого узла до стартового, используя родительские узлы, и вычисляет общий вес пути.

Мы создаём вектор для хранения пути и переменную для хранения веса.

В цикле восстанавливаем путь, добавляя узлы в вектор и вычисляя вес каждого ребра.

После завершения цикла переворачиваем вектор пути и выводим общий вес и итоговый путь.

1. **BFS (поиск в ширину)**

// Функция для BFS  
double BFS(const Graph& graph, int start, int goal) {  
 auto start\_time = high\_resolution\_clock::*now*();  
  
 queue<int> q; // максимум - V \* 4 байта  
 set<int> visited; // максимум - V \* 4 байта  
 map<int, int> parent; // максимум - V \* 4\*2 байта  
 q.push(start);  
  
 while (!q.empty()) {  
 int current = q.front(); // 4 байта  
 q.pop();  
  
 if (visited.count(current)) continue;  
 visited.insert(current);  
  
 if (current == goal) {  
 auto end\_time = high\_resolution\_clock::*now*();  
 double totalWeight = printPath(parent, start, goal, graph); // 8 байт  
 cout << "BFS time: " << duration\_cast<milliseconds>(end\_time - start\_time).count() << " ms\n";  
 return totalWeight;  
 }  
  
 for (const auto& [neighbor, \_] : graph.at(current)) {  
 if (!visited.count(neighbor)) {  
 q.push(neighbor);  
 parent[neighbor] = current;  
 }  
 }  
 }  
 cout << "BFS: Path not found.\n"; // O(1)  
 return -1; // Возвращаем -1, если путь не найден  
}

Инициализация:

Сначала фиксируем время начала выполнения алгоритма.

Создаём три структуры данных:

* + - **queue<int> q** для хранения узлов, которые нужно обработать (память O(V)).
    - **set<int> visited** для отслеживания посещенных узлов (память O(V)).
    - **map<int, int> parent** для хранения родительских узлов, что позволяет восстановить путь после нахождения целевого узла (память O(V)).

Основной цикл:

Пока очередь не пуста, извлекаем текущий узел.

Если узел уже посещен, пропускаем его.

Если текущий узел равен целевому, вызываем функцию printPath для восстановления пути и подсчета его веса, после чего выводим время выполнения и возвращаем общий вес пути.

Если узел не является целевым, перебираем его соседей. Если сосед не был посещен, добавляем его в очередь и устанавливаем его родителя.

1. **DFS (Поиск в глубину)**

double DFS(const Graph& graph, int start, int goal) {  
 auto start\_time = high\_resolution\_clock::*now*();  
  
 stack<int> s; // V\*4 байта  
 set<int> visited; // максимум- V\*4 байта  
 map<int, int> parent; // максимум - V \* 4\*2 байта  
 s.push(start);  
  
 while (!s.empty()) {  
 int current = s.top(); // 4 байта  
 s.pop();  
  
 if (visited.count(current)) continue;  
 visited.insert(current);  
  
 if (current == goal) {  
 auto end\_time = high\_resolution\_clock::*now*();  
 double totalWeight = printPath(parent, start, goal, graph); // 8 байт  
 cout << "DFS time: " << duration\_cast<milliseconds>(end\_time - start\_time).count() << " ms\n";  
 return totalWeight;  
 }  
  
 for (const auto& [neighbor, \_] : graph.at(current)) {  
 if (!visited.count(neighbor)) {  
 s.push(neighbor);  
 parent[neighbor] = current;  
 }  
 }  
 }  
 cout << "DFS: Path not found.\n";  
 return -1;  
}

Инициализация:

* + **start\_time** фиксируе время начала выполнения функции.
  + **stack<int> s** используется для хранения узлов, которые нужно исследовать. Память выделяется по мере добавления узлов.
  + **set<int> visited** хранит узлы, которые уже были посещены, чтобы избежать циклов. Память зависит от количества узлов.
  + **map<int, int> parent** хранит информацию о родительских узлах для восстановления пути. Память также зависит от количества узлов.

Основной цикл:

* + Цикл продолжается, пока стек не пуст.
  + Извлекается текущий узел из стека.
  + Если узел уже посещен, итерация продолжается.
  + Если текущий узел равен целевому, вызывается функция printPath для восстановления пути и подсчета веса, после чего возвращается общий вес.

Перебор соседей:

Для каждого соседа текущего узла проверяется, был ли он посещен. Если нет, он добавляется в стек и устанавливается как дочерний узел текущего.

Завершение:

Если путь не найден, выводится соответствующее сообщение, и функция возвращает -1.

1. **Дейкстра**

// Функция для алгоритма Дейкстры  
double Dijkstra(const Graph& graph, int startNode, int targetNode) {  
 auto startTime = high\_resolution\_clock::*now*();  
  
 priority\_queue<pair<double, int>, vector<pair<double, int>>, greater<>> minHeap; // O(V)  
 map<int, double> shortestDistances; // O(V) в худшем случае  
 map<int, int> predecessors; // O(V) в худшем случае  
  
 // Инициализация расстояний до всех узлов бесконечностью  
 for (const auto& node : graph) {  
 shortestDistances[node.first] = numeric\_limits<double>::*infinity*();  
 }  
 shortestDistances[startNode] = 0;  
  
 minHeap.emplace(0, startNode);  
  
 while (!minHeap.empty()) {  
 auto [currentDistance, currentNode] = minHeap.top();  
 minHeap.pop();  
  
 if (currentNode == targetNode) {  
 auto endTime = high\_resolution\_clock::*now*();  
 double totalWeight = printPath(predecessors, startNode, targetNode, graph); // 8 байт  
 cout << "Dijkstra time: "  
 << duration\_cast<milliseconds>(endTime - startTime).count() << " ms\n";  
 return totalWeight;  
 }  
  
 // Обход соседей текущего узла  
 for (const auto& [neighbor, edgeWeight] : graph.at(currentNode)) {  
 double potentialDistance = currentDistance + edgeWeight; // 8 байт  
  
 if (potentialDistance < shortestDistances[neighbor]) {  
 shortestDistances[neighbor] = potentialDistance;  
 predecessors[neighbor] = currentNode;  
 minHeap.emplace(potentialDistance, neighbor);  
 }  
 }  
 }  
  
 cout << "Dijkstra's algorithm: Path not found.\n";  
 return -1; // Возвращаем -1, если путь не найден  
}

Инициализация:

Сначала фиксируем время начала выполнения алгоритма.

Создаём структуры данных: приоритетная очередь **pq** для хранения узлов по их расстоянию, а также два отображения **distances** и **parent** для хранения расстояний до узлов и их родительских узлов соответственно.

Инициализация расстояний:

Все расстояния инициализируются до бесконечности, кроме стартового узла, который устанавливается в 0.

Основной цикл:

Пока очередь не пуста, извлекается узел с минимальным расстоянием.

Если текущий узел является целевым, восстанавливается путь и выводится время выполнения.

Для каждого соседа текущего узла вычисляется новое расстояние. Если оно меньше, чем текущее, обновляются расстояние и родительский узел, и сосед добавляется в очередь.

Завершение:

Если путь не найден, выводится соответствующее сообщение.

1. **A\***

// Функция для алгоритма A\*  
double AStar(const Graph& graph, int startNode, int goalNode, const Coordinates& coords) {  
 auto startTime = high\_resolution\_clock::*now*();  
  
 // Эвристическая функция для вычисления расстояния до цели  
 auto heuristicFunc = [&](int node) {  
 auto [longitude1, latitude1] = coords.at(node);  
 auto [longitude2, latitude2] = coords.at(goalNode);  
 return sqrt(pow(longitude1 - longitude2, 2) + pow(latitude1 - latitude2, 2));  
 };  
  
 priority\_queue<pair<double, int>, vector<pair<double, int>>, greater<>> openSet; // O(V)  
 map<int, double> gCosts; // O(V) в худшем случае  
 map<int, double> fCosts; // O(V) в худшем случае  
 map<int, int> cameFrom; // O(V) в худшем случае  
  
 // Инициализация стоимости g и f для всех узлов  
 for (const auto& node : graph) {  
 gCosts[node.first] = numeric\_limits<double>::*infinity*();  
 fCosts[node.first] = numeric\_limits<double>::*infinity*();  
 }  
 gCosts[startNode] = 0;  
 fCosts[startNode] = heuristicFunc(startNode);  
  
 openSet.emplace(fCosts[startNode], startNode);  
  
 while (!openSet.empty()) {  
 auto [currentFCost, currentNode] = openSet.top();  
 openSet.pop();  
  
 if (currentNode == goalNode) {  
 auto endTime = high\_resolution\_clock::*now*();  
 double totalWeight = printPath(cameFrom, startNode, goalNode, graph); // 8 байт  
 cout << "A\* execution time: " << duration\_cast<milliseconds>(endTime - startTime).count() << " ms\n";  
 return totalWeight;  
 }  
  
 for (const auto& [neighbor, edgeWeight] : graph.at(currentNode)) {  
 double tentativeGCost = gCosts[currentNode] + edgeWeight; // 8 байт  
  
 if (tentativeGCost < gCosts[neighbor]) {  
 gCosts[neighbor] = tentativeGCost;  
 fCosts[neighbor] = tentativeGCost + heuristicFunc(neighbor);  
 cameFrom[neighbor] = currentNode;  
 openSet.emplace(fCosts[neighbor], neighbor);  
 }  
 }  
 }  
 cout << "A\*: Path not found.\n";  
 return -1; // Возвращаем -1, если путь не найден  
}

Эвристическая функция:

Определяется лямбда-функция heuristicFunc, которая вычисляет эвристическую стоимость (предполагаемое расстояние) от текущего узла до целевого узла на основе координат. Используется евклидова метрика для расчета расстояния.

Структуры данных:

•**openSet**: приоритетная очередь для хранения узлов, которые нужно исследовать. Узлы сортируются по их стоимости f, которая равна сумме g (стоимость пути от начального узла до текущего) и эвристической оценки h (стоимость от текущего узла до цели).

• **gCosts**: карта, хранящая стоимость пути от начального узла до каждого узла.

• **fCosts**: карта, хранящая общую стоимость (g + h) для каждого узла.

• **cameFrom**: карта, отслеживающая предшественников каждого узла для восстановления пути после завершения алгоритма.

Инициализация:

Все стоимости g и f для узлов инициализируются бесконечностью, за исключением начального узла, для которого gCosts устанавливается в 0 и fCosts вычисляется с помощью эвристической функции.

Основной цикл:

• Цикл продолжается до тех пор, пока есть узлы в openSet.

• Извлекается узел с наименьшей стоимостью f.

• Если текущий узел является целевым, происходит восстановление пути с помощью функции printPath, и выводится время выполнения алгоритма.

• Для каждого соседа текущего узла вычисляется потенциальная стоимость tentativeGCost. Если она меньше известной стоимости для соседа, обновляются стоимости g и f, а также предшественник соседа. Узел добавляется в openSet.

Если все узлы исследованы и целевой узел не найден, выводится сообщение о том, что путь не найден, и функция возвращает -1.

**ЭКСПЕРЕМЕНТАЛЬНАЯ ЧАСТЬ**

1. **Результаты:**
2. **BFS (Поиск в ширину):**

**•** Финальный вес: **0.47283**

**•** Время выполнения: **947 мс**

Поиск в ширину (BFS) — это алгоритм, который исследует все узлы на текущем уровне перед тем, как перейти к следующему. В данном случае, полученный вес пути выше, чем у других алгоритмов, что может указывать на то, что BFS не нашел оптимальный путь, особенно если веса ребер графа различны. Время выполнения достаточно высокое, что может быть связано с тем, что алгоритм исследует больше узлов.

1. **DFS (Поиск в глубину):**

**•** Финальный вес: **30.2946**

**•** Время выполнения: **499 мс**

Поиск в глубину (DFS) не гарантирует нахождение кратчайшего пути, поскольку он может углубляться в один из путей, не исследуя другие возможности. Высокий вес найденного пути указывает на то, что алгоритм прошел через множество узлов, не находя оптимального решения. Время выполнения DFS меньше, чем у BFS, что может быть связано с тем, что он не исследует все узлы на каждом уровне.

1. **3. Дейкстра:**

**•** Финальный вес: **0.104106**

**•** Время выполнения: **407 мс**

Алгоритм Дейкстры предназначен для нахождения кратчайшего пути в графах с неотрицательными весами. Полученный вес является наименьшим среди всех алгоритмов, что подтверждает его эффективность для данной задачи. Время выполнения также достаточно низкое, что делает его хорошим выбором для поиска кратчайшего пути.

1. **A\*:**

**•** Финальный вес: **0.104106**

**•** Время выполнения: **337 мс**

Алгоритм A\* использует эвристические функции для оптимизации поиска и также гарантирует нахождение кратчайшего пути при правильном выборе эвристики. Он показал аналогичный результат по весу с алгоритмом Дейкстры, но с меньшим временем выполнения, что делает его наиболее эффективным из представленных алгоритмов.

1. **Сравнение времени работы алгоритмов**

|  |  |
| --- | --- |
| **Алгоритм** | **Время, ms** |
| BFS | 990 |
| DFS | 498 |
| Дейкстра | 419 |
| **A\*** | **343** |

1. **Подсчет асимптотики**

**Поиск в ширину (BFS)**

В худшем случае BFS посещает все узлы и все рёбра графа. Если граф содержит V вершин (узлов) и E рёбер (связей), то временная сложность будет:

**O(V + E)**

Это связано с тем, что каждый узел добавляется в очередь и обрабатывается один раз, а каждое ребро рассматривается при обходе соседей.

**Поиск в глубину (DFS)**

В худшем случае DFS также посещает все узлы и рёбра графа. Аналогично BFS, временная сложность будет:

**O(V + E)**

Каждый узел обрабатывается один раз, и каждое ребро рассматривается.

**Алгоритм Дейкстры**

Основная операция в алгоритме Дейкстры — это извлечение узла с минимальным расстоянием из приоритетной очереди и обновление расстояний соседних узлов.

В худшем случае алгоритм проходит по всем вершинам V и всем рёбрам E:

• Извлечение узла с минимальным расстоянием: O(V log V)

• Обход всех рёбер: O(E log V)

Таким образом, общая временная сложность: **O((V + E) log V)**

**Алгоритм A\***

A\* также использует приоритетную очередь для выбора узлов с минимальной стоимостью f(n) = g(n) + h(n) , где g(n) — стоимость пути от начального узла до узла n , а h(n) — эвристическая оценка стоимости пути от узла n до целевого узла.

Временная сложность аналогична алгоритму Дейкстры, так как A\* также проходит по всем вершинам и рёбрам:

• Извлечение узла с минимальной стоимостью: O(V log V)

• Обход всех рёбер: O(E log V)

Таким образом, общая временная сложность алгоритма A\*: **O((V + E) log V)**

1. **Подсчёт памяти**

**Поиск в ширину (BFS)**

BFS использует очередь для хранения узлов, которые нужно посетить, а также множество для отслеживания посещённых узлов. В худшем случае, когда граф представляет собой полный граф или дерево, память может достигать:

**O(V)**

Это связано с тем, что в очереди могут храниться все узлы на одном уровне

**Поиск в глубину (DFS)**

DFS использует стек (или рекурсию), чтобы отслеживать узлы. В худшем случае, когда граф представляет собой длинную цепь (например, линейный граф), память может достигать:

**O(V)**

Однако в случае использования рекурсии, стек вызовов может занять дополнительное место, что также составляет O(V) .

**Алгоритм Дейкстры**

• Храним расстояния до всех вершин: O(V)

• Храним предшественников: O(V)

• Храним приоритетную очередь: O(V)

Итого пространственная сложность: **O(V)**

**Алгоритм A\***

* Храним стоимости g и f для всех вершин: O(V)
* Храним предшественников: O(V)
* Храним приоритетную очередь: O(V)

Итого пространственная сложность: **O(V)**

**ЗАКЛЮЧЕНИЕ**

В ходе лабораторной работы были протестированы четыре различных алгоритма поиска кратчайшего пути в графе: BFS, DFS, Дейкстра и A\*. Результаты показали значительные различия в эффективности и качестве найденных путей.

Алгоритмы BFS и DFS продемонстрировали недостаточную эффективность в условиях графа с различными весами ребер. BFS обеспечил более высокий вес пути по сравнению с Дейкстрой и A\*, а DFS показал наибольший вес, что указывает на его неспособность находить оптимальные решения.

Алгоритм Дейкстры продемонстрировал хорошую производительность и обеспечил нахождение кратчайшего пути, но время выполнения было выше по сравнению с A\*.

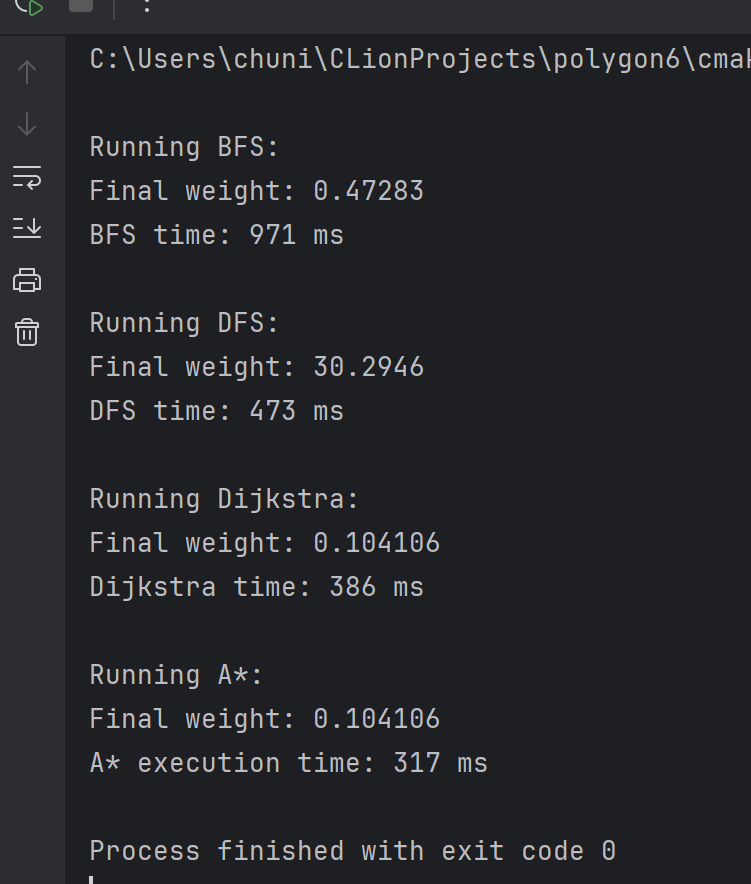
Алгоритм A\* оказался наиболее эффективным как по качеству найденного пути, так и по времени выполнения.

На основании полученных данных можно сделать вывод, что для поиска кратчайших путей в графах с неотрицательными весами рёбер предпочтительными являются алгоритмы Дейкстры и A\*, при этом A\* является более эффективным выбором благодаря своей способности использовать эвристики для сокращения времени выполнения. Это делает его особенно подходящим для задач, требующих быстрого нахождения решений в больших графах.

Данная лабораторная работа подтвердила важность выбора правильного алгоритма в зависимости от характеристик графа и требований к времени выполнения.

**ПРИЛОЖЕНИЕ**

<https://github.com/ITMO-ML-algorithms-and-data-structures/polygon/pull/1091> **-** исходный код

****