ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 8

« Найти путь алгоритмами от места вашего постоянного проживания в спб к университету ИТМО на переданном графе»

Выполнил работу

Лапшин Максим Константинович

Академическая группа J3111

Принято

Ментор, Вершинин Владислав Константинович

Санкт-Петербург

2024

1. Введение

В данной работе мы сосредоточимся на реализации трех основных алгоритмов: Алгоритм поиска в ширину (BFS), Алгоритм поиска в глубину (DFS) и Алгоритм Дейкстры. Эти алгоритмы не только позволяют эффективно находить пути в графах, но и служат основой для более сложных методов и приложений, таких как планирование маршрутов, анализ графов и др. Теоретическая часть

1. Алгоритм поиска в ширину (BFS)

Алгоритм поиска в ширину (BFS, Breadth-First Search) — это алгоритм, который исследует граф, начиная с заданной исходной вершины и последовательно посещая всех её соседей. Затем он переходит к соседям этих соседей и так далее, пока не будут изучены все доступные вершины. BFS реализуется с помощью очереди, что обеспечивает последовательное прохождение уровней графа.

Основные свойства BFS:

- Находит кратчайший путь в невзвешенном графе.
- Работает за O(V + E) времени, где V — количество вершин, а E — количество рёбер в графе.

Приложение А

```
void bfs(const std::vector<Node>& nodes, int currentNodeIndex, int targetNodeIndex, double& minWeight, std::unordered_set<int> visited) {
```

```
    std::queue<std::pair<int, double>> queue; // Храним пару: индекс узла и текущий вес
    queue.push({ currentNodeIndex, 0.0 });
    visited.insert(currentNodeIndex);

    // Устанавливаем начальное значение minWeight в максимально возможное
    minWeight = std::numeric_limits<double>::infinity();

    while (!queue.empty()) {
        auto front = queue.front();
        int nodeIndex = front.first;
        double weight = front.second;
        queue.pop();

        // Если достигли целевого узла, обновляем minWeight
        if (nodeIndex == targetNodeIndex) {
            minWeight = std::min(minWeight, weight);
            break;
        }

        // Обходим все смежные ребра текущего узла
        for (const auto& edge : nodes[nodeIndex].edges) {
            int nextNodeIndex = findNodeIndex(edge.lon, edge.lat, nodes);
            double edgeWeight = edge.weight;
            if (visited.find(nextNodeIndex) == visited.end()) {
                queue.push({ nextNodeIndex, weight + edgeWeight });
                visited.insert(nextNodeIndex);
            }
        }
    }
}
```

Время работы алгоритма BFS

2. Алгоритм поиска в глубину (DFS)

Алгоритм поиска в глубину (DFS, Depth-First Search) — это еще один алгоритм для обхода графов, который исследует ветвь графа, до тех пор, пока есть возможность двигаться вперед, и затем возвращается к предыдущей вершине, чтобы исследовать другие ветви. DFS может быть реализован как с помощью рекурсии, так и с использованием стека.

Основные свойства DFS:

- Подходит для задач, где необходимо исследовать все возможные пути.
- Работает за O(V + E) времени и может использовать меньше памяти, чем BFS в случае разреженных графов.

Приложение В

```
void dfs(const std::vector<Node>& nodes, int currentNodeIndex, int
targetNodeIndex, double currentWeight, double& minWeight,
std::unordered_set<int>& visited) {
   if (visited.find(currentNodeIndex) != visited.end()) {
      return;
   }
   if (currentNodeIndex == targetNodeIndex) {
      minWeight = std::min(minWeight, currentWeight);
      return;
   }
   visited.insert(currentNodeIndex);
   for (const Edge& edge : nodes[currentNodeIndex].edges) {
      for (int i = 0; i < nodes.size(); ++i) {
         if (nodes[i].lon == edge.lon && nodes[i].lat == edge.lat) {
            dfs(nodes, i, targetNodeIndex, currentWeight + edge.weight,
minWeight, visited);
            break;
         }
      }
   }
   visited.erase(currentNodeIndex);
}
```

Время работы алгоритма DFS

3. Алгоритм Дейкстры

Алгоритм Дейкстры — это алгоритм поиска кратчайших путей от одной начальной вершины до всех других вершин в графе с неотрицательными весами рёбер. Он работает путем последовательного выбора вершины с минимальным расстоянием и обновления расстояний до соседних вершин.

Основные свойства алгоритма Дейкстры:

- Находит кратчайшие пути во взвешенных графах.
- Эффективность алгоритма составляет O((V + E) log V) при использовании приоритетной очереди, что делает его подходящим для графов с большим числом вершин.

Приложение C

```
std::vector<double> dijkstra(int startNodeIndex, const std::vector<Node>&
nodes, std::unordered_set<int> visited) {
   std::vector<double> distances(nodes.size(),
std::numeric_limits<double>::infinity());
   distances[startNodeIndex] = 0;
   std::priority_queue<Distance, std::vector<Distance>, std::greater<Distance>>
queue;
   queue.push({ startNodeIndex, 0 });
   while (!queue.empty()) {
      int currentNodeIndex = queue.top().nodeIndex;
      double currentDistance = queue.top().distance;
      queue.pop();
      // Если узел уже посещён, пропускаем его
      if (visited.find(currentNodeIndex) != visited.end()) {
         continue;
      }
      // Добавляем узел к посещённым
      visited.insert(currentNodeIndex);
      // Проходим все ребра текущего узла
      for (const auto& edge : nodes[currentNodeIndex].edges) {
         double newDistance = currentDistance + edge.weight;
         int targetIndex = findNodeIndex(edge.lon, edge.lat, nodes);
         // Если найден более короткий путь до смежного узла
         if (newDistance < distances[targetIndex]) {
            distances[targetIndex] = newDistance;
            queue.push({ targetIndex, newDistance });
         }
      }
   }
   return distances; // Возвращаем массив расстояний
}
```

```
Time taken: 24.2677 milliseconds
```

Время работы алгоритма Дейкстры

4. Заключение

В данной работе будет осуществлена реализация алгоритмов BFS, DFS и Дейкстры на языке программирования C++. Они послужат основой для дальнейшего анализа и решения более сложных задач, связанных с графовыми структурами. Понимание этих алгоритмов и их эффективная реализация имеет критическое значение для выполнения множества прикладных задач в информатике и смежных дисциплинах

Приложение D

#include #include #include #include #include #include #include #include #include #include
struct Edge { double lon; double lat; double weight; };
struct Node { double lon; double lat; std::vector edges; };
struct Distance { int nodeIndex; double distance;
```
bool operator>(const Distance& other) const {
    return distance > other.distance; // Для min-очереди
}

};
```
std::vector parseDataFromFile(const std::string& filename) { std::vector nodes; std::ifstream file(filename); std::string line;
```
if (!file.is_open()) {
    std::cerr << "Error opening file: " << filename << std::endl;
    return nodes;
}

auto addNode = [&nodes](double lon, double lat) {
    Node node{ lon, lat };
    nodes.push_back(node);
    return nodes.size() - 1; // Возвращаем индекс нового узла
    };

while (std::getline(file, line)) {
    std::stringstream ss(line);
    std::string nodePart, edgesPart;
    std::getline(ss, nodePart, ':');

    std::stringstream nodeStream(nodePart);
    double lon, lat;
    char comma;

    if (!(nodeStream >> lon >> comma >> lat) || comma != ',') {
        std::cerr << "Invalid node coordinates in line: " << line <<
std::endl;
        continue;
    }
```

```
    int currentNodeIndex = addNode(lon, lat);
    std::getline(ss, edgesPart);
    std::stringstream edgesStream(edgesPart);
    std::string edgeStr;

    while (std::getline(edgesStream, edgeStr, ';')) {
        std::stringstream edgeStream(edgeStr);
        double edgeLon, edgeLat, weight;
        char edgeComma1, edgeComma2;

        if (!(edgeStream >> edgeLon >> edgeComma1 >> edgeLat >> edgeComma2
>> weight) ||
            (edgeComma1 != ',' || edgeComma2 != ',')) {
            std::cerr << "Invalid edge data in edge: " << edgeStr <<
std::endl;
            continue;
        }

        // Проверяем, существует ли целевой узел
        auto it = std::find_if(nodes.begin(), nodes.end(), [&](const Node&
n) {
            return n.lon == edgeLon && n.lat == edgeLat;
            });

        int targetNodeIndex = (it != nodes.end()) ?
std::distance(nodes.begin(), it) : addNode(edgeLon, edgeLat);
        nodes[currentNodeIndex].edges.push_back({ edgeLon, edgeLat,
weight }); // Добавляем ребро
    }
}

return nodes;

}
int findNodeIndex(double lon, double lat, const std::vector& nodes) { for (int i = 0; i < nodes.size();
++i) { if (nodes[i].lon == lon && nodes[i].lat == lat) { return i; } } return -1; }
void dfs(const std::vector& nodes, int currentNodeIndex, int targetNodeIndex, double currentWeight,
double& minWeight, std::unordered_set& visited) { if (visited.find(currentNodeIndex) !=
visited.end()) { return; }
if (currentNodeIndex == targetNodeIndex) {
    minWeight = std::min(minWeight, currentWeight);
    return;
}

visited.insert(currentNodeIndex);

for (const Edge& edge : nodes[currentNodeIndex].edges) {
    for (int i = 0; i < nodes.size(); ++i) {
        if (nodes[i].lon == edge.lon && nodes[i].lat == edge.lat) {
            dfs(nodes, i, targetNodeIndex, currentWeight + edge.weight,
minWeight, visited);
            break;
        }
```

```
    }
}

visited.erase(currentNodeIndex);

}
void bfs(const std::vector& nodes, int currentNodeIndex, int targetNodeIndex, double& minWeight,
std::unordered_set visited) { std::queue<std::pair<int, double>> queue; // Храним пару: индекс узла
и текущий вес queue.push({ currentNodeIndex, 0.0 }); visited.insert(currentNodeIndex);
// Устанавливаем начальное значение minWeight в максимально возможное
minWeight = std::numeric_limits<double>::infinity();

while (!queue.empty()) {
    auto front = queue.front();
    int nodeIndex = front.first;
    double weight = front.second;
    queue.pop();

    // Если достигли целевого узла, обновляем minWeight
    if (nodeIndex == targetNodeIndex) {
        minWeight = std::min(minWeight, weight);
        break;
    }

    // Обходим все смежные ребра текущего узла
    for (const auto& edge : nodes[nodeIndex].edges) {
        int nextNodeIndex = findNodeIndex(edge.lon, edge.lat, nodes);
        double edgeWeight = edge.weight;
        if (visited.find(nextNodeIndex) == visited.end()) {
            // Добавляем соседний узел в очередь и отмечаем его как
посещенный
            queue.push({ nextNodeIndex, weight + edgeWeight });
            visited.insert(nextNodeIndex);
        }
    }
}

}
std::vector dijkstra(int startNodeIndex, const std::vector& nodes, std::unordered_set visited)
{ std::vector distances(nodes.size(), std::numeric_limits::infinity()); distances[startNodeIndex] = 0;
std::priority_queue<Distance, std::vector<Distance>,
std::greater<Distance>> queue;
queue.push({ startNodeIndex, 0 });
while (!queue.empty()) {
    int currentNodeIndex = queue.top().nodeIndex;
    double currentDistance = queue.top().distance;
    queue.pop();

    // Если узел уже посещён, пропускаем его
    if (visited.find(currentNodeIndex) != visited.end()) {
        continue;
    }
```

```
    // Добавляем узел к посещённым
    visited.insert(currentNodeIndex);

    // Проходим все ребра текущего узла
    for (const auto& edge : nodes[currentNodeIndex].edges) {
        double newDistance = currentDistance + edge.weight;
        int targetIndex = findNodeIndex(edge.lon, edge.lat, nodes);

        // Если найден более короткий путь до смежного узла
        if (newDistance < distances[targetIndex]) {
            distances[targetIndex] = newDistance;
            queue.push({ targetIndex, newDistance });
        }
    }
}

return distances; // Возвращаем массив расстояний

}
int main() { const std::string filename = "graph.txt"; std::vector nodes =
parseDataFromFile(filename);
double startLon = 30.4141326;
double startLat = 59.9470649;
double targetLon = 30.4140936;
double targetLat = 59.9469059;

int startNodeIndex = findNodeIndex(startLon, startLat, nodes);
int targetNodeIndex = findNodeIndex(targetLon, targetLat, nodes);

if (startNodeIndex == -1 || targetNodeIndex == -1) {
    std::cout << "Invalid start or target node." << std::endl;
    return 1;
}

double minWeight = std::numeric_limits<double>::infinity();
std::unordered_set<int> visited;

dfs(nodes, startNodeIndex, targetNodeIndex, 0.0, minWeight, visited);
if (minWeight < std::numeric_limits<double>::infinity()) {
    std::cout << "minWeight: " << minWeight << std::endl;
}
else {
    std::cout << "None." << std::endl;
}

bfs(nodes, startNodeIndex, targetNodeIndex, minWeight, visited);
if (minWeight < std::numeric_limits<double>::infinity()) {
    std::cout << "minWeight: " << minWeight << std::endl;
}
else {
    std::cout << "None." << std::endl;
}
```

```
std::vector<double> distances = dijkstra(startNodeIndex, nodes, visited);
if (distances[targetNodeIndex] < std::numeric_limits<double>::infinity())
{
    std::cout << "minWeight: " << minWeight << std::endl;
}
else {
    std::cout << "None." << std::endl;
}

return 0;

}
```