МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет ИТМО»
(Университет ИТМО)

Лабораторная работа №8
по дисциплине
«Алгоритмы и структуры данных»

по теме:
ГРАФЫ

Студент:
Группа № J311 *А.П. Бабич*

Предподаватель:
ментор, *В.К. Вершинин*

Санкт-Петербург 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы является изучение применения алгоритмов поиска кратчайшего пути в графах на примере задачи нахождения кратчайшего пути от места проживания до университета ИТМО. Для решения этой задачи используются различные алгоритмы поиска пути на графе дорог.

В процессе выполнения работы необходимо решить следующие задачи:

1. Изучить принципы работы популярных алгоритмов поиска пути в графах, таких как:
   - **BFS** (поиск в ширину),
   - **DFS** (поиск в глубину),
   - Алгоритм **Дейкстры**,
   - Алгоритм **A***.
2. Разработать и реализовать данные алгоритмы на языке `C++` для поиска кратчайшего пути между двумя точками в графе.
3. Проанализировать затраты времени для каждого из реализованных алгоритмов и сравнить полученные результаты.

В работе будет проведено практическое исследование эффективности различных алгоритмов поиска кратчайшего пути, а также их сравнительный анализ с точки зрения скорости и точности поиска.

1 ТЕОРЕТИЧЕСКАЯ ПОДГОТОВКА

В теории графов существует несколько популярных алгоритмов поиска, которые используются для решения различных задач, связанных с нахождением путей, исследованием связности и т. д. Рассмотрим основные из них.

1.1 Поиск в ширину (BFS)

Алгоритм поиска в ширину (BFS) представляет собой метод исследования графа, который начинается с исходной вершины и исследует все вершины, расположенные на одинаковом уровне от начальной. Этот алгоритм гарантирует нахождение кратчайшего пути в невзвешенных графах, так как последовательно посещает вершины, основываясь на их удаленности от начальной точки. Алгоритм проходит по всем вершинам на одном уровне перед тем, как перейти на следующий, что обеспечивает равномерное покрытие графа.

1.2 Поиск в глубину (DFS)

Алгоритм поиска в глубину (DFS) углубляется в граф, двигаясь по пути до тех пор, пока не достигнет тупика (вершины, не имеющей соседей). Это рекурсивный алгоритм, который не гарантирует нахождение кратчайшего пути, но полезен для исследования связности графа и поиска различных путей между вершинами. DFS обычно применяется для поиска всех возможных путей или для решения задач, связанных с топологической сортировкой графов.

1.3 Алгоритм Дейкстры

Алгоритм Дейкстры является жадным методом, предназначенным для нахождения кратчайших путей от одной вершины до всех остальных вершин в графах с положительными весами рёбер. Он работает путем выбора вершины с минимальным расстоянием на каждом шаге, используя структуру данных, такую как приоритетная очередь, для эффективного выбора ближайшей

вершины. Этот алгоритм находит оптимальные решения в графах, где все рёбра имеют положительные веса.

1.4 Алгоритм A*

Алгоритм A* представляет собой улучшенную версию алгоритма Дейкстры, добавляющую эвристическую функцию для ускорения процесса поиска. Вместо того чтобы просто выбирать вершины с минимальным расстоянием, A* использует оценку, которая сочетает в себе стоимость пути от начальной вершины и приближенную оценку расстояния до цели. Это позволяет значительно ускорить процесс поиска, особенно в случаях, когда приближенное расстояние до цели известно заранее. Алгоритм A* эффективно решает задачи поиска кратчайших путей в графах, включая те, где необходимо учесть дополнительную информацию, такую как препятствия или другие специфичные условия.

2 РЕАЛИЗАЦИЯ АЛГОРИТМОВ

В этом разделе представлена реализация четырех популярных алгоритмов поиска в графах: BFS (поиск в ширину), DFS (поиск в глубину), алгоритм Дейкстры и A*. Полный код для каждого из этих алгоритмов можно найти в приложениях 5.1, 5.2, 5.3 и 5.4.

2.1 Алгоритм поиска в ширину (BFS)

Алгоритм BFS используется для поиска кратчайшего пути в невзвешенных графах. Он исследует все вершины, расположенные на одинаковом уровне от начальной вершины, что гарантирует нахождение кратчайшего пути в случае, если все рёбра графа имеют одинаковый вес. Объявление алгоритма BFS на C++:

```
1 std::vector<Node *> bfs(Node *start, Node *goal);
```

Полный код можно найти в приложении 5.1.

2.2 Алгоритм поиска в глубину (DFS)

Алгоритм DFS используется для поиска путей в графах и проверки связности. Он углубляется в граф, исследуя как можно дальше по пути до тех пор, пока не достигнет тупика, после чего возвращается и исследует другие возможные пути. Объявление алгоритма DFS на C++:

```
1 std::vector<Node *> dfs(Node *start, Node *goal);
```

Полный код можно найти в приложении 5.2.

2.3 Алгоритм Дейкстры

Алгоритм Дейкстры используется для нахождения кратчайших путей от одной вершины до всех остальных вершин в графах с положительными весами рёбер. Он работает жадным методом, выбирая на каждом шаге вершину с минимальным расстоянием.

Объявление алгоритма Дейкстры на C++:

```
std::vector<Node *> dijkstra(Graph &graph, Node *start, Node *goal);
```

Полный код можно найти в приложении 5.3.

2.4 Алгоритм A*

Алгоритм A* улучшает алгоритм Дейкстры, добавляя эвристическую функцию для ускорения поиска. Это особенно полезно, когда известно приближенное расстояние от текущей вершины до целевой.

Объявление алгоритма A* на C++:

```
std::vector<Node *> A_Star(Graph &graph, Node *start, Node *goal);
```

Полный код можно найти в приложении 5.4.

2.5 Контекстная ссылка на полные реализации

Полные реализации алгоритмов BFS, DFS, Дейкстры и A* на C++ можно найти в приложениях 5.1, 5.2, 5.3 и 5.4.

3 ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ

3.1 Сравнение времени выполнения алгоритмов

Для оценки эффективности алгоритмов поиска в графах были проведены измерения времени их выполнения. Результаты представлены в таблице 1.

Таблица 1 — Сравнение времени выполнения алгоритмов

Алгоритм	**Время (мс)**
BFS	982
DFS	1259
Dijkstra	1909
A*	1127

3.2 Графики времени выполнения алгоритмов

На рисунках 1–4 представлены графики, отображающие время выполнения каждого из алгоритмов.

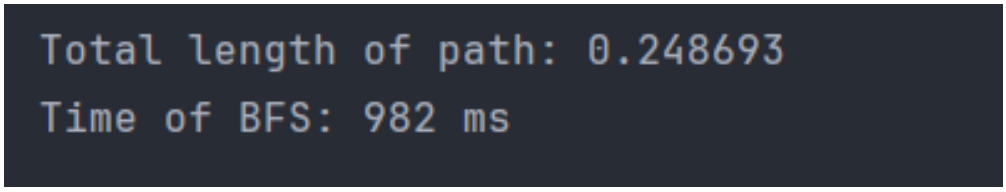

Рисунок 1 — График времени выполнения BFS

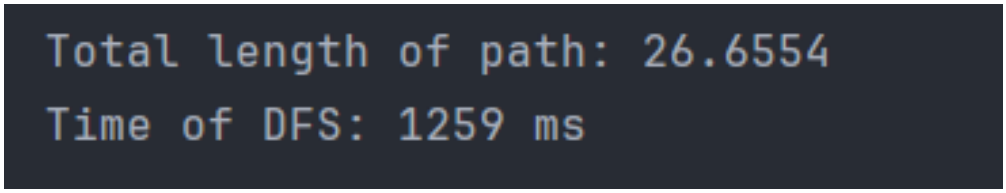

Рисунок 2 — График времени выполнения DFS

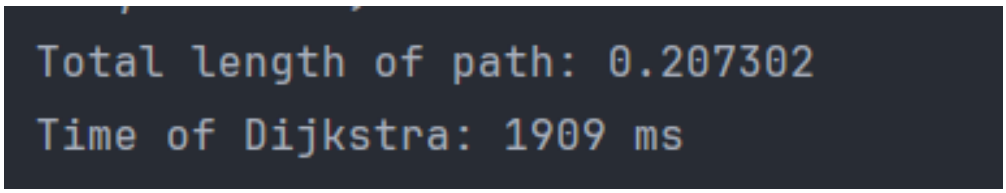

Рисунок 3 — График времени выполнения алгоритма Dijkstra

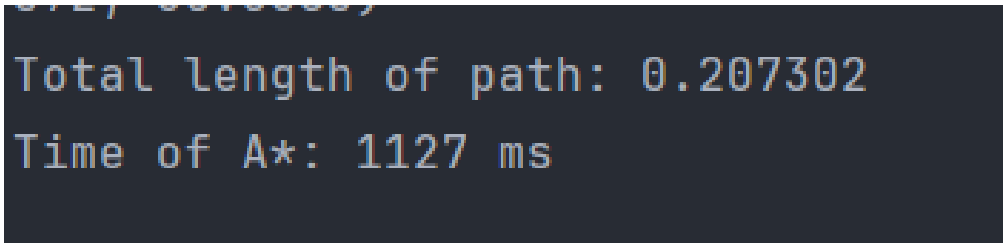

Рисунок 4 — График времени выполнения алгоритма A*

3.3 Асимптотическая сложность алгоритмов

Таблица 2 содержит сравнение асимптотической сложности алгоритмов по времени и памяти.

Таблица 2 — Асимптотическая сложность алгоритмов

Алгоритм	**Временная сложность**	**Память**
BFS	$O(V+E)$	$O(V)$
DFS	$O(V+E)$	$O(V)$
Dijkstra	$O((V+E)\log V)$	$O(V+E)$
A*	$O((V+E)\log V)$	$O(V+E)$

где:

– V — количество вершин в графе.
– E — количество рёбер в графе.

3.4 Вывод по асимптотике

Алгоритмы BFS и DFS имеют линейную сложность по времени ($O(V+E)$), что делает их подходящими для задач обхода графа. Однако в общем случае они не учитывают веса рёбер, что ограничивает их применение для поиска кратчайших путей.

Алгоритмы Dijkstra и A* демонстрируют более высокую временную сложность ($O((V+E)\log V)$) из-за использования приоритетных очередей,

что обеспечивает эффективность при работе с взвешенными графами. Алгоритм A*, в отличие от Dijkstra, дополнительно использует эвристическую функцию, что делает его более предпочтительным в задачах с заранее известной целевой вершиной.

По использованию памяти BFS и DFS менее ресурсоёмки ($O(V)$), так как хранят только информацию о посещённых вершинах. Dijkstra и A* требуют больше памяти ($O(V + E)$), так как дополнительно хранят структуру для обработки весов рёбер.

4 ЗАКЛЮЧЕНИЕ

В ходе эксперимента были протестированы четыре популярных алгоритма поиска в графах: BFS (поиск в ширину), DFS (поиск в глубину), алгоритм Дейкстры и алгоритм A*. Время выполнения каждого из них измерено, а результаты представлены в Таблице 1. Асимптотическая сложность этих алгоритмов описана в Таблице 2.

Результаты эксперимента

BFS: Время выполнения составило 982 миллисекунды. Это результат, соответствующий теоретическим ожиданиям, поскольку BFS эффективно работает в графах с равными весами рёбер. Его временная сложность $O(V+E)$ делает этот алгоритм быстрым при работе с большими графами.

DFS: Время выполнения составило 1259 миллисекунд. Несмотря на схожую временную сложность с BFS ($O(V+E)$), DFS не гарантирует нахождение кратчайшего пути, что ограничивает его применение. Алгоритм может быть полезен в задачах обхода или поиска любого достижимого пути.

Алгоритм Дейкстры: Этот алгоритм показал наибольшее время выполнения — 1909 миллисекунд. Временная сложность $O((V+E)\log V)$ обусловлена использованием приоритетной очереди. Хотя Дейкстра более ресурсоёмок, он гарантирует нахождение кратчайшего пути в графах с неотрицательными весами рёбер. Этот алгоритм предпочтителен для задач с неравномерными весами рёбер.

A*: Время выполнения составило 1127 миллисекунд, что сопоставимо с BFS. Однако благодаря использованию эвристической функции, A* часто работает быстрее, чем алгоритм Дейкстры, избегая ненужных путей. В задачах с заранее известной целевой вершиной и подходящей эвристикой A* является более эффективным выбором.

Вывод

Исследование времени позволило оценить производительность и особенности четырёх популярных алгоритмов поиска в графах: BFS, DFS, алгоритма Дейкстры и A*.

Каждый алгоритм обладает своими преимуществами и ограничениями, зависящими от структуры графа и требований задачи:

BFS продемонстрировал высокую эффективность в графах с равными весами рёбер. Этот алгоритм подходит для задач, где важно гарантированно найти кратчайший путь, но отсутствуют требования к учёту весов. DFS оказался универсальным инструментом для обхода графов, однако его ограничение в виде отсутствия гарантии нахождения кратчайшего пути делает его менее предпочтительным в задачах поиска оптимальных решений. Алгоритм Дейкстры подтвердил свою надёжность в нахождении кратчайших путей в графах с неравномерными весами рёбер. Однако его высокая временная сложность делает его менее подходящим для обработки больших графов. A*, благодаря использованию эвристики, обеспечил баланс между точностью и производительностью. Этот алгоритм оптимален для задач направленного поиска, особенно если доступна качественная эвристическая оценка. Анализ асимптотической сложности (Таблица 2) подтвердил, что выбор алгоритма зависит от соотношения количества вершин и рёбер графа, а также от распределения весов.

Можно сделать следующий вывод: универсального решения для всех графов не существует. Алгоритм следует выбирать, исходя из особенностей задачи:

Если важно равномерное покрытие графа или поиск кратчайшего пути в равновесных графах — предпочтителен BFS. Для обхода произвольного пути — DFS. Если требуются точные решения с учётом весов, стоит использовать Дейкстру. Для ускорения поиска с известной целью и качественной эвристикой лучшим выбором будет A*.

5 ПРИЛОЖЕНИЕ: РЕАЛИЗАЦИЯ АЛГОРИТМОВ

В данном приложении приведены реализации четырёх алгоритмов поиска в графах, которые использовались для нахождения пути от ПМЖ до ИТМО.

5.1 Алгоритм BFS (Обход в ширину)

Ниже приведена реализация алгоритма поиска в ширину, который используется для нахождения кратчайшего пути от стартового узла до целевого в графе, если он существует.

Листинг 5.1 — BFS - Поиск в ширину

```
std::vector<Node *> bfs(Node *start, Node *goal) {
    if (!start || !goal) {
        return {};
    }

    std::queue<Node *> queue;
    std::unordered_map<Node *, Node *> came_from;
    std::unordered_map<Node *, bool> visited;

    queue.push(start);
    visited[start] = true;

    while (!queue.empty()) {
        Node *current = queue.front();
        queue.pop();

        if (current == goal) {
            std::vector<Node *> path;
            for (Node *at = goal; at != nullptr; at = came_from[at]) {
                path.push_back(at);
            }
            std::reverse(path.begin(), path.end());
            return path;
        }

        for (const auto &edge: current->edges) {
            Node *neighbor = edge.first;
            if (!visited[neighbor]) {
```

```
                queue.push(neighbor);
                visited[neighbor] = true;
                came_from[neighbor] = current;
            }
        }
    }

    return {};
}
```

5.2 Алгоритм DFS (Поиск в глубину)

Пример реализации алгоритма поиска в глубину для нахождения пути от стартового узла до целевого.

Листинг 5.2 — DFS - Поиск в глубину

```
std::vector<Node *> dfs(Node *start, Node *goal) {
    if (!start || !goal) {
        return {};
    }

    std::unordered_map<Node *, bool> visited;
    std::unordered_map<Node *, Node *> came_from;
    std::stack<Node *> stack;

    stack.push(start);
    visited[start] = true;

    while (!stack.empty()) {
        Node *current = stack.top();
        stack.pop();

        if (current == goal) {
            std::vector<Node *> path;
            for (Node *at = goal; at != nullptr; at = came_from[at]) {
                path.push_back(at);
            }
            std::reverse(path.begin(), path.end());
            return path;
        }

        for (const auto &edge: current->edges) {
            Node *neighbor = edge.first;
            if (!visited[neighbor]) {
                stack.push(neighbor);
                visited[neighbor] = true;
                came_from[neighbor] = current;
            }
        }
    }

    return {};
}
```

5.3 Алгоритм Дейкстры

Алгоритм Дейкстры для нахождения кратчайшего пути в графе с использованием очереди с приоритетом.

Листинг 5.3 — Dijkstra - Алгоритм Дейкстры

```
std::vector<Node *> dijkstra(Graph &graph, Node *start, Node *goal) {
    if (!start || !goal) {
        return {};
    }

    std::unordered_map<Node *, double> distances;
    std::unordered_map<Node *, Node *> came_from;
    std::unordered_map<Node *, bool> visited;

    for (const auto &node: graph.nodes) {
        distances[node.get()] = std::numeric_limits<double>::infinity()
           ;
    }
    distances[start] = 0.0;

    using PQElement = std::pair<double, Node *>;
    std::priority_queue<PQElement, std::vector<PQElement>, std::greater
       <>> pq;
    pq.emplace(0.0, start);

    while (!pq.empty()) {
        auto [current_distance, current_node] = pq.top();
        pq.pop();

        if (visited[current_node]) {
            continue;
        }
        visited[current_node] = true;

        if (current_node == goal) {
            std::vector<Node *> path;
            for (Node *at = goal; at != nullptr; at = came_from[at]) {
                path.push_back(at);
            }
            std::reverse(path.begin(), path.end());
            return path;
        }

```

```
37          for (const auto &edge: current_node->edges) {
38              Node *neighbor = edge.first;
39              double weight = edge.second;
40
41              double new_distance = current_distance + weight;
42              if (new_distance < distances[neighbor]) {
43                  distances[neighbor] = new_distance;
44                  came_from[neighbor] = current_node;
45                  pq.emplace(new_distance, neighbor);
46              }
47          }
48      }
49
50      return {};
51  }
```

5.4 Алгоритм A*

Реализация алгоритма A* с использованием эвристической функции для поиска кратчайшего пути.

Листинг 5.4 — A* - Алгоритм A*

```
double heuristic(Node *a, Node *b) {
    return std::sqrt(std::pow(a->lat - b->lat, 2) + std::pow(a->lon - b
       ->lon, 2));
}

std::vector<Node *> A_Star(Graph &graph, Node *start, Node *goal) {
    if (!start || !goal) {
        return {};
    }

    std::unordered_map<Node *, double> gScore;
    std::unordered_map<Node *, double> fScore;
    std::unordered_map<Node *, Node *> came_from;

    for (const auto &node: graph.nodes) {
        gScore[node.get()] = std::numeric_limits<double>::infinity();
        fScore[node.get()] = std::numeric_limits<double>::infinity();
    }
    gScore[start] = 0.0;
    fScore[start] = heuristic(start, goal);

    using PQElement = std::pair<double, Node *>;
    std::priority_queue<PQElement, std::vector<PQElement>, std::greater
       <>> openSet;
    openSet.emplace(fScore[start], start);

    while (!openSet.empty()) {
        Node *current = openSet.top().second;
        openSet.pop();

        if (current == goal) {
            std::vector<Node *> path;
            for (Node *at = goal; at != nullptr; at = came_from[at]) {
                path.push_back(at);
            }
            std::reverse(path.begin(), path.end());
            return path;
        }
```

```
37
38          for (const auto &edge: current->edges) {
39              Node *neighbor = edge.first;
40              double weight = edge.second;
41
42              double tentative_gScore = gScore[current] + weight;
43              if (tentative_gScore < gScore[neighbor]) {
44                  came_from[neighbor] = current;
45                  gScore[neighbor] = tentative_gScore;
46                  fScore[neighbor] = tentative_gScore + heuristic(
                        neighbor, goal);
47
48                  openSet.emplace(fScore[neighbor], neighbor);
49              }
50          }
51      }
52
53      return {};
54  }
```

Здесь представлены исходные коды четырёх популярных алгоритмов поиска кратчайшего расстояния в графах: BFS, DFS, Дейкстры и A*. Для выполнения задачи поиска кратчайшего пути от ПМЖ до ИТМО. Так же, на гитхабе залит полный код реализации классов и функций