ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 8

«Графы, алгоритмы поиска кратчайшего пути»

Выполнила работу

Кузнецова Таисия

Академическая группа №J3113

Принято

Ходненко И.В

Санкт-Петербург

2024

**Введение**

**Цель -** научится применять изученные алгоритмы на графах (A\*,Дейкстры, BFS и DFS) в практических, реальных кейсах.

**Задачи**:

1. Повторить изученный материал по данной теме.
2. Проанализировать данный граф.
3. Продумать реализацию каждого алгоритма, с учетом их особенностей.
4. Написать код алгоритмов.
5. Сравнить результаты.
6. **Теоретическая подготовка**

**Граф -** это структура данных, которая состоит из вершин и рёбер. Вершины - это элементы, которые могут содержать данные, а рёбра - это связи между вершинами. Графы часто используются для моделирования различных объектов и их связей, например, для описания маршрутов на карте, сети компьютеров или социальных сетей.В C++ графы могут быть реализованы разными способами:

Список смежности: для каждой вершины хранится список её соседей. Это удобный способ представления, если количество рёбер не очень велико.Матрица смежности: используется двумерный массив, где каждая ячейка определяет наличие или отсутствие рёбер между вершинами. Подходит для плотных графов. В плюсах для хранения графов часто используют контейнеры из библиотеки STL (Standard Template Library). Например:

*std::vector* для хранения вершин и рёбер.

*std::map* или *std::unordered\_map* для быстрого поиска вершин по ключу.

*std::set* или *std::unordered\_set* для хранения уникальных значений, например, для посещённых вершин.

Часто для представления графа используются указатели на объекты, что позволяет динамически выделять память для создания новых вершин, благодаря чему графы на C++ преобладают над теми же графами Python.

**Алгоритм Дейкстры** находит кратчайший путь от одной вершины до всех остальных вершин в графе с положительными весами рёбер.

Он использует приоритетную очередь (например, *std::priority\_queue*) для того, чтобы в каждый момент времени выбирать вершину с минимальным расстоянием.

Начинается с начальной вершины и присваивает ей расстояние 0. Для всех остальных вершин назначается бесконечное расстояние.

Постепенно обновляется расстояние для соседних вершин, если находится более короткий путь.

**BFS** — это алгоритм для поиска в графе, который исследует вершины, начиная с начальной, по уровням. Сначала посещаются все соседние вершины, затем их соседи и так далее.Алгоритм использует очередь ( *std::queue*) для того, чтобы гарантировать, что вершины будут посещаться в порядке их удалённости от начальной.

Он начинается с начальной вершины и добавляет её в очередь. Далее поочерёдно извлекает вершины из очереди и добавляет их соседей.

**DFS** — это алгоритм поиска, который исследует граф, двигаясь по одной ветви как можно дальше до конца, прежде чем вернуться и исследовать другие ветви.Он использует рекурсию или стек для того, чтобы углубиться в граф, не прерываясь, пока не дойдёт до конца ветви.После того как алгоритм достигает вершины, не имеющей соседей, он возвращается к предыдущей вершине и продолжает поиск.

**A\*** — это улучшенная версия алгоритма Дейкстры, которая использует эвристику (предположительное расстояние до цели), чтобы сделать поиск более направленным и быстрее находить кратчайший путь.

Алгоритм работает аналогично Дейкстре, но на каждом шаге использует эвристическую функцию для оценки "приоритетности" вершин.Эта функция — это сумма текущего расстояния от начальной вершины и предполагаемого расстояния до целевой вершины.

1. **Реализация**
2. Выбор представления графа. Поскольку нам необходимо было реализовать алгоритмы, которые обходят граф по соседям, я решила представить граф списком смежности.Для этого в каждой вершине у меня было несколько параметров: идентификатор, широта и долгота. Для хранения соседей я использовала *std::vector<std::pair<Node\*, double>>*, где *Node\** — указатель на соседнюю вершину, а double — вес рёбра, соединяющего текущую вершину с соседом.
3. Чтения координат из файла.Для начала я написала функцию *parseFromFile*, которая читает данные о вершинах и рёбрах из файла. Эти данные включают координаты (широта и долгота) и вес рёбер между соседними вершинами. Я решила хранить координаты в виде строк (с помощью *std::string*), чтобы можно было быстро искать вершины по координатам, используя кэш с помощью *std::unordered\_map*.Мне нужно было убедиться, что если в графе уже существует вершина с такими координатами, я её не добавлю снова, а просто обновлю связи (рёбра) с ней. Для этого использовала кэш для быстрого поиска вершин.
4. Реализация DFS (Поиск в глубину).В C++ я реализовала DFS с использованием стека (*std::stack*), чтобы не использовать рекурсию, что даёт больше контроля над памятью. Стек помогает хранить текущие вершины, и мы поочередно извлекаем их, пока не достигнем конца пути. Я также использовала *std::unordered\_set* для хранения посещённых вершин, чтобы не заходить в одну и ту же вершину несколько раз.
5. Реализация BFS (Поиск в ширину). В отличие от DFS, Поиск в ширину исследует граф уровневым образом, начиная с начальной вершины и исследуя все её соседние вершины, затем соседей этих вершин и так далее. Я использовала очередь (*std::queue*) для хранения вершин, которые нужно исследовать. Очередь гарантирует, что мы будем посещать вершины в порядке их удалённости от начальной.
6. Реализация алгоритма Дейкстры. Алгоритм находит кратчайший путь от одной вершины до всех остальных вершин в графе с положительными весами рёбер. Для этого пришлось использовать приоритетную очередь (или *std::priority\_queue*), которая позволяет всегда извлекать вершину с минимальным расстоянием.

В начале я инициализировала все расстояния как бесконечность, а для начальной вершины поставила расстояние 0. Затем я итеративно обновляла минимальные расстояния для всех соседей текущей вершины. Приоритетная очередь помогает эффективно извлекать вершину с минимальным расстоянием.

1. Реализация А\* .Алгоритм A\* — это модификация алгоритма Дейкстры с использованием эвристики. Эвристика помогает направлять поиск к целевой вершине, что делает A\* более быстрым в некоторых случаях, чем стандартный Дейкстра.Поэтому использовала стандартную формулу A\*:

*f(n) = g(n) + h(n)*

где:

g(n) — стоимость пути от начальной вершины до вершины n,

h(n) — эвристика, которая оценивает стоимость пути от вершины n до целевой вершины.

Для реализации алгоритма я использовала *std::priority\_queue,* как и в Дейкстре, но теперь каждый элемент в очереди хранит не только минимальное расстояние, но и значение функции f(n).

1. Подсчёт времени работы алгоритмов.Для каждого алгоритма я добавила измерение времени работы с помощью *std::chrono::high\_resolution\_clock*. Я решила использовать этот инструмент, чтобы точно видеть, сколько времени занимает выполнение каждого алгоритма на реальных данных.Для BFS и DFS я выбрала контейнеры *std::queue* и *std::stack*, потому что они идеально подходят для этих алгоритмов: очередь гарантирует правильный порядок обхода для BFS, а стек — для DFS.Для Дейкстры и A\* я использовал приоритетную очередь, потому что это ключевой элемент для оптимизации поиска кратчайшего пути, особенно когда нужно часто извлекать минимальный элемент.
2. **Экспериментальная часть**

**Затраты по памяти:**

Дейкстра = O (V+E)

Dfs = O(V)

Bfs = O(V)

A\* = O(V+E)

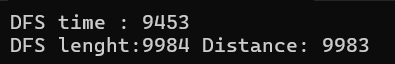
**Затраты по времени:**

Дейкстра = O( V log V + E log V)

изображение №1 - Результат и время выполнения Дейкстра

Dfs = O(V+E)

Изображение №2 - Результат выполнения DFS



Bfs = O(V+E)

Изображение №3 - Результат и время выполнения BFS



A\* = O(V log V + E log V)

Изображение №4 - Результат и время выполнения A\*



Изображение №5 - Гистограма работы Алгоритма



**Вывод:**

В рамках лабораторной работы были реализованы четыре алгоритма поиска кратчайшего пути: DFS (поиск в глубину), BFS (поиск в ширину), Дейкстра и A\*. Алгоритмы были протестированы на большом графе, представляющем собой модель карты с узлами и рёбрами. На основе результатов эксперимента построена гистограмма, показывающая время выполнения каждого алгоритма.

DFS (поиск в глубину)показал худшее время работы. Алгоритм глубоко углубляется в граф и только после этого возвращается назад, чтобы проверить другие пути. Это приводит к большому числу лишних обходов, особенно на больших графах.DFS не оптимизирован для поиска кратчайших путей, так как он не учитывает веса рёбер и полностью обходит подграфы, даже если путь уже найден.Хотя DFS прост в реализации, он неэффективен для задач поиска кратчайшего пути, особенно в графах большого размера.BFS (поиск в ширину)показал более эффективное время работы по сравнению с DFS. BFS обходит граф уровнями, что позволяет находить кратчайший путь в графе с одинаковыми весами рёбер.Алгоритм подходит для невзвешенных графов, но теряет эффективность на больших графах из-за большого потребления памяти для хранения вершин уровня.BFS лучше подходит для задач, где веса рёбер одинаковы или не имеют значения.Дейкстра:Один из лучших результатов. Алгоритм гарантирует нахождение кратчайшего пути во взвешенном графе за счёт использования приоритетной очереди для работы с вершинами.Подходит для графов с положительными весами. Однако сложность O(V log V + E log V) делает его менее оптимальным на больших графах по сравнению с A\*.Алгоритм хорошо справляется с задачами поиска кратчайшего пути в графах среднего размера.A\*показал лучший результат. A\* использует эвристику, чтобы направлять поиск в сторону целевой вершины, избегая лишнего обхода.Он сочетает стоимость пути от начальной вершины до текущей (g(n)) с предположительной стоимостью от текущей вершины до целевой (h(n)), что делает его очень эффективным.A\* является оптимальным выбором для поиска кратчайшего пути на больших графах, особенно если эвристика хорошо подобрана. DFS оказался самым медленным и неэффективным алгоритмом, который не подходит для задачи поиска кратчайшего пути.

BFS работает лучше DFS, но ограничен задачами с равными весами рёбер.

Дейкстра справился с задачей быстрее и точнее BFS, но проигрывает A\* в эффективности на больших графах.A\* показал себя самым быстрым и эффективным алгоритмом благодаря использованию эвристики.

Таким образом, для задач поиска кратчайшего пути на больших графах с весами рёбер оптимальным выбором является алгоритм A\*, который демонстрирует превосходство над другими алгоритмами как по времени работы, так и по точности.

**Приложение:**

Изображение №6 - Листинг кода

