Санкт-Петербургский государственный
университет
Факультет прикладной математики —
процессов управления
Программирование и информационные технологии

Теория графов и её приложения

Отчёт по проектному заданию

Преподаватели
Воронкова Е. Б.
Вольф Д. А.

Выполнили
Козырев С. А.
Куклин Д. В.

Санкт-Петербург
2020

Содержание

Постановка задачи и методы решения

В качестве семестрового проекта по теории графов требовалось, используя картографические данные проекта *OpenStreetmap*, построить граф дорог Российского города-миллионера и выполнить задания по оценке удобства размещения зданий и планированию их размещения, используя построенный граф.

Для реализации проекта мы выбрали язык C++, так как

- подавляющее большинство проектов для работы с картами написаны либо на C, либо на C++; либо используют линковку с Python, но также предоставляют C++ интерфейс,
- среди языков, которые используются в картографических проектах, C++ является наиболее быстрым,
- Python простой.

Также выбрали Нижний Новгород в качестве города-миллионера.

Следуя условию задачи, мы использовали данные *OpenStreetMap* (далее *OSM*).

Построение графа дорог

Спецификация *OSM* [1] определяет следующие структуры данных для хранения объектов географических карт: *Node*, *Way* и *Relation* [4].

Структура *Node* (узел) [2] обязана иметь уникальный идентификатор (`Integer`, занимающий 64 бита), а также широту и долготу (рекомендуется использовать `Float` шириной не менее 64 бит). Опционально каждый объект типа *Node* содержит множество меток (*tag*), которые позволяют получить дополнительную информацию об объекте карты (например, метка *highway=crossing* даёт понять, что данный обозначает пешеходный переход).

Структура *Way* (путь) [3] есть упорядоченный список узлов, которая либо имеет по крайней мере одну метку, либо принадлежит *Relation*.

Пути могут быть открытыми, либо закрытыми. У открытых путей не совпадают первый и последний узлы. Многие дороги определяются в *OSM* как открытые пути.

Закрытые пути как правило определяют объект *Area* (к нему относятся дома, окольные пути, стены и т. п.).

Структуру *Relation* (отношение) мы исключили из рассмотрения, так как используемая нами библиотека не имеет широких возможностей для работы с ней. Также, объекты этого типа по большей части не предоставляют требуемую нами информацию.

Для работы с данными *OSM* мы использовали библиотеку *Libosmium* [5], которая предоставляет удобный API на языке C++ для работы с объектами *OSM*.

Первым шагом мы загрузили Protobuf файл карты Нижнего Новгорода с сайта-планировщика велосипедных путей *BBBike* [8] (см. Рис. 1).

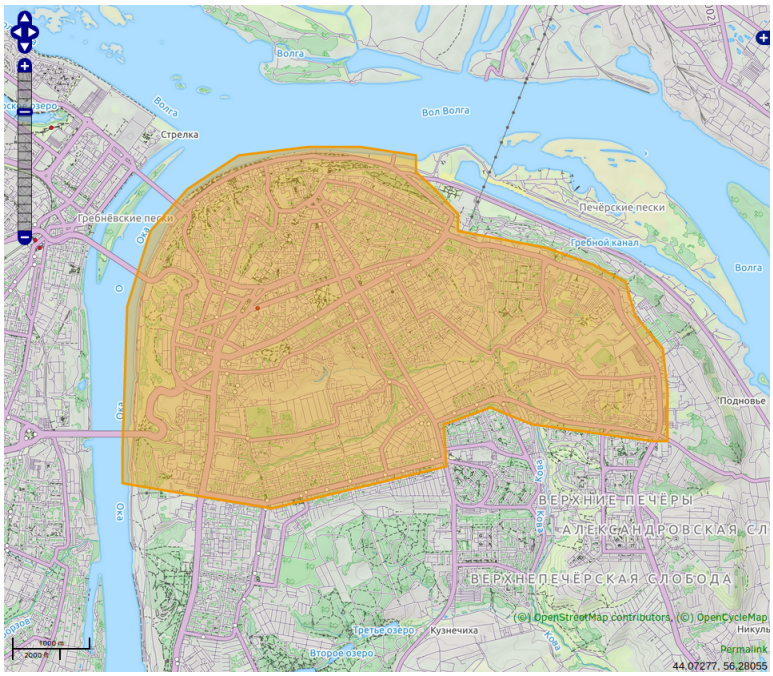

Рис. 1: Используемая карта центра Нижнего Новгорода

Далее требовалось создать карту узлов, используя [5]. Задача осложня-

лась тем, что пути могли иметь пересечения в некоторых узлах, значит, необходимо было разделить пути по узлам, лежащим на пересечениях. Также, многие рёбра могли быть использованы без пользы, например в тупиках, но вносить весомый вклад в производительность.

Мы использовали способ, предложенный на форуме [7], что позволило сократить число узлов почти в два раза. Для простоты здесь и далее из кода убраны обозначения, не связанные с его логикой.

```
unordered_map<uint64_t, bool> marked {};

void way(Way& way) noexcept {
    if (!way.has_key("highway")) { return; }
    for (auto& node: way.nodes()) {
        if (marked.contains(node)) {
            marked[node] |= true;
        } else {
            marked.insert({ node, false });
        }
    }
}
```

Для каждого пути на карте мы помечали каждый узел пути, если он встречался более одного раза (таким образом, он лежит на пересечении). При следующей итерации по каждому пути, мы разделяли путь по узлу, имеющему отметку. Если узел не помечен, то в граф добавляем ребро между предыдущим и текущим узлами.

Так как у каждого узла известна широта и долгота, то расстояние между двумя узлами можно вычислить, используя формулу гаверсинуса [9]:

$$\eta(\Theta) = \eta(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\eta(\lambda_2 - \lambda_1),$$
$$d = 2r \arcsin(\sqrt{\eta(\Theta)}),$$

где Θ — центральный угол, $\varphi_{1,2}$ — широты в радианах, $\lambda_{1,2}$ — высоты в радианах и $\eta(\theta) = \sin^2\left(\frac{\theta}{2}\right)$ для произвольного угла θ.

```
for (auto& curr: way.nodes()) {
    if (curr != first && curr != last) {
        if (marked.at(curr)) {
            auto d = ... // One-way or two-way.
```

```
            auto w = haversine(pred, curr);
            routes.add_edge({ pred, curr }, w, d);
            pred = &curr;
        }
    }
}

auto w = haversine(pred, last), d = ...;
routes.add_edge({ pred, last }, w, d);
```

Структура `Node` содержит, как и в спецификации, поля уникального идентификатора в *OSM*, широту и высоту.

```
struct Node {
private:
    using Angle = long double;
    uint64_t m_id = 0;
    Angle m_latitude = 0;
    Angle m_longitude = 0;
};
```

Граф представлен списком смежности.

```
struct Graph {
private:
    using Edge = pair<Node, Node>;
    using OutEdges = unordered_map<Node, Distance>;
    using AdjList = unordered_map<Node, OutEdges>;

    AdjList m_data {};
};
```

Так как и жилые дома, и объекты инфраструктуры определяются в *OSM* как пути, то их сложно связать в одной структуре с графом дорог. Для примера, мы могли помещать каждый узел здания в граф и помечать его особым образом, чтобы выделять среди узлов-частей дорог. В таком случае усложнилась бы реализация большинства алгоритмов и вспомогательных структур.

Мы решили выбрать иной подход к решению задачи. Для каждого здания на карте

1. вычисляем барицентр здания,
2. находим ближайший к зданию узел дороги,
3. в структуре *Building* связываем со зданием найденный узел.

```
struct Building {
private:
    enum class Type {
        House,
        Facility
    };

    uint64_t m_id = 0;
    Type m_type;
    Angle m_latitude = 0;
    Angle m_longitude = 0;
    Node m_closest_node; // Closest road node.
};
```

Финальная структура, содержащая всю необходимую информацию о карте города, имеет следующий вид:

```
struct Map {
    struct Path {
    private:
        Building m_from, m_to;
        Distance m_distance;
    };

    struct TracedPath: public Path {
    private:
        vector<Node> m_trace;
    };

private:
    Buildings m_buildings;
    Graph m_graph;
};
```

Очевидно, что время составления карты оказывается довольно велико. Во-первых, библиотека *Libosmium* каждый раз производит чтение файла с картой и последующее заполнение собственных объектов данных. Вопрос итоговой сложности всех библиотечных операций является открытым, но любой желающий может получить на него ответ, изучив исходный код библиотеки [6].

Во-вторых, сложность составления карты равна $O(n \cdot (m+n))$, где m — число узлов-домов, а n — число узлов-дорог. Таким образом, даже при 16 тысячах узлов (примерное число узлов на карте центра Нижнего Новгорода), карта заполняется несколько минут, что неприемлемо для real-time приложения.

Мы решили эту проблему, применив кэширование к полученной структуре данных `Map`. Для простоты реализации используем популярную библиотеку *Boost*.

```
template<typename T>
bool serialize(fs::path& filename, T&& data) {
    ofstream binary { filename };
    boost::binary_oarchive archive { binary };
    archive << data;
    binary.close();
    return true;
}
```

```
template<typename T>
bool deserialize(fs::path& filename, T&& data) {
    if (!fs::exists(filename)) { return false; }
    ifstream binary { filename };
    boost::binary_iarchive archive { binary };
    archive >> data;
    binary.close();
    return true;
}
```

Подход оправдывает себя, так как большая часть карты остается статичной и при добавлении в базу данных новых узлов каждый раз заново обходить все пути не требуется. В итоге, скорость загрузки карты сократилась с нескольких минут до долей секунды (более точные измерения см. в Результаты), и граф дорог Нижнего Новгорода построен.

Оценка удобства размещения объектов инфраструктуры

Требовалось выбрать M произвольных объектов инфраструктуры и N жилых зданий оценить удобство их размещения, используя расстояние как метрику. При обходе всех путей (*Way* в *OSM*), мы выбирали пути с соответствующими метками.

```
const vector<string> houses =
    { "apartments", "bungalow", "cabin", "detached",
      "dormitory", "farm", "ger", "hotel", "house",
      "houseboat", "residential", "terrace" };
const vector<string> facilities =
    { "fire_station", "hospital", "retail",
      "kiosk", "supermarket" };
```

Как было указано выше, для каждого здания мы запоминаем ближайший узел.

```
void way(Way& way) noexcept {
    const auto location = barycenter(way);
    auto closest = min_element(routes.nodes(),
        [&](const auto& lhs, const auto& rhs) {
            return
                haversine(lhs, location) <
                haversine(rhs, location);
        });
    auto b = Building { way, location, closest };
    buildings.push(b);
}
```

Для определения кратчайших путей от объекта карты до другого естественным кажется применение алгоритма Дейкстры. Нами был реализован алгоритм с временной сложностью $O(n \log n)$ (подробнее см. Результаты).

```
auto dijkstra(Node s) -> ShortestPaths {
    unordered_map<Node, Distance> ds;
    set<pair<Distance, Node>> set;

    for (const auto&[v, _]: nodes()) { ds[v] = INF; }
    ds[s] = 0;
    set.insert({ ds[s], s });
```

```
    while (!set.empty()) {
        auto [v, _] = set.begin();
        set.erase(set.begin());
        for (auto& u: nodes()[v]) {
            auto [to, length] = u;
            if (ds[v] + length < ds[to]) {
                set.erase({ ds[to], to });
                ds[to] = ds[v] + length;
                set.insert({ ds[to], to });
            }
        }
    }

    return ds;
}
```

Используя результаты работы алгоритма несложно как и определить ближайшие объекты, так и построить дерево кратчайших путей.

Планирование новых объектов

Дерево кратчайших путей

Для построения дерева кратчайших путей от объекта до выбранных узлов требуется требуется выполнить обход алгоритмом Дейкстры, запоминая для каждой вершины её предка. Здесь нам понадобится дополнительный тип данных: `using Trail = std::unordered_map<Node, Node>;`. `Trail` для каждого узла хранит его предка. Теперь приведём алгоритм Дейкстры с запоминанием предка:

```
auto dijkstra(Node s) -> pair<ShortestPaths, Trail> {
    std::unordered_map<Node, Distance> ds;
    std::set<pair<Distance, Node>> set;
    std::unordered_map<Node, Node> previous;

    for (const auto&[v, _]: nodes()) { ds[v] = INF; }

    ds[s] = 0;
```

```
    set.insert({ ds[s], s });
    while (!set.empty()) {
        auto[_, v] = *set.begin();
        set.erase(set.begin());
        for (const auto& u: nodes().at(v)) {
            auto[to, length] = u;
            if (ds[v] + length < ds[to]) {
                set.erase({ ds[to], to });
                ds[to] = ds[v] + length;
                previous[to] = v;
                set.insert({ ds[to], to });
            }
        }
    }

    return { ds, previous };
}
```

После этого мы преобразуем `Trail` уже в структуру `TracedPath`:

```
auto shortest_paths_with_trace(Building from,
    const Buildings& to) -> TracedPaths {
    auto[distances, trail] =
        m_graph.dijkstra(from.closest());
    TracedPaths result {};

    for (const auto& building: to) {
        auto node_to = building.closest(),
             node_from = from.closest();
        auto distance = distances.at(node_to);

        // Reconstruct path.
        std::vector<Node> path;
        for (auto v = node_to; v != from.closest();
            v = trail[v]) {
            path.push_back(v);
        }
        path.push_back(node_from);
        reverse(path.begin(), path.end());
```

```
        // Build path in place.
        result.emplace_back(from, building, path,
            distance);
    }

    return result;
}
```

А теперь пути объединяем в новый `Map`:

```
Map paths_to_map(Map& map, Map::TracedPaths& paths) {
    std::unordered_set<Building> set;
    Buildings buildings;
    Graph routes;

    for (const auto& path: paths) {
        auto[from, to] = path.ends();
        set.insert(from);
        set.insert(to);

        auto pred = *path.path().begin();
        for (const auto curr: path.path()) {
            if (curr == pred) { continue; }
            //weight of edge from pred to curr
            auto w = map.nodes().find(pred)
                ->second.find(curr)->second;
            routes.add_edge_one_way({ pred, curr },
                w);
            pred = curr;
        }
    }

    buildings.insert(buildings.end(), set.begin(),
        set.end());

    return Map { buildings, routes };
}
```

В итоге получили дерево кратчайших путей. Чтобы найти сумму длинн кратчайших путей, надо на шаге, где мы получили `TracedPaths`, сло-

жить их длины. Для нахождения общей длины дерева нужно просто сложить веса рёбер.

Разбиение на кластеры

Для разбиения на кластеры мы будем использовать дополнительные структуры: `Cluster` и `ClusterStructure`.

В `Cluster` мы храним информацию о кластере: его размер, индексы его элементов, по которым мы можем их найти, и т.д. Так как на каждом шаге метода полной связи мы объединяем 2 кластера, то также будем хранить их.

В `ClusterStructure` хранится дерево кластеров и различная информация о нём.

```
struct Cluster {
    static size_t overall_clusters_num;
    Cluster* m_left;    //  subclusters
    Cluster* m_right;   //
    size_t m_size; // cluster size
    size_t m_first; // index of the first element
    size_t m_last; // index of the last element
    size_t m_id; // cluster id
    Building m_center;
};
```

```
struct ClusterStructure {
    const Cluster* m_root = nullptr;
    Buildings m_data;
    Clusters m_clusters {};
    DMatrix<Building> m_dm_buildings;
    matrix<uint64_t> m_dm_clusters;
    std::vector<int64_t> _m_next;
    size_t m_clusters_num;
    const Map& m_map;
};
```

Также нам потребуется матрица расстояний:

```
using DMatrix = std::unordered_map<std::pair<T, T>, double>;
```

Её мы заполняем результатами вызова Дейкстры для каждой вершины.

Теперь перейдём к самой реализации. Кластеризацию мы производим в конструкторе структуры `ClusterStructure`. Начинаем с того, что для каждой вершины создаём кластер и для кластеров заполняем матрицу расстояний, просто переписывая матрицу расстояний для домов.

```
ClusterStructure(const Map& map,
    Buildings&& buildings, DMatrix<Building>&& dm)
    : m_data(buildings)
    , m_dm_buildings(dm)
    , m_dm_clusters(2 * buildings.size() - 1,
        2 * buildings.size() - 1)
    , m_clusters_num(0)
    , m_map(map) {

//        for every building create cluster
    m_clusters.reserve(2 * m_data.size() - 1);
    _m_next.resize(2 * m_data.size() - 1);
    for (size_t i = 0; i < m_data.size(); ++i) {
        cluster_from_element(i, m_data[i]);
    }

//        create distance matrix for clusters
    for (auto& c1: m_clusters) {
        for (auto& c2: m_clusters) {
            m_dm_clusters(c1.id(), c2.id()) =
                m_dm_buildings.at({
                    m_data[c1.first()],
                    m_data[c2.first()]
                });
        }
    }
```

Затем создадим `unordered_set`, в котором будем хранить все кластеры, которые пока ещё не находятся в других кластерах. Будем называть их активными кластерами.

```
std::unordered_set<Cluster>
```

```
    active_clusters(m_clusters.begin(),
    m_clusters.end());
```

Далее в цикле, пока количество активных кластеров больше одного, будем повторять следующие действия:

1. находим активные кластеры, расстояние между которыми минимально;
2. объединяем их в новый кластер, находим расстояния между новым кластером и остальными, добавляем их в матрицу расстояний;
3. добавляем его в активные кластеры, из активных кластеров удаляем кластеры из первого шага.

Находим минимальное расстояние:

```
std::pair<size_t, size_t> min = { -1, -1 };
for (auto& c1: active_clusters) {
    for (auto& c2: active_clusters) {
        if (c1.id() == c2.id()) { continue; }

        if (min.first > m_clusters.size()) {
            min = { c1.id(), c2.id() };
            continue;
        }

        if (m_dm_clusters(c1.id(), c2.id()) <
            m_dm_clusters(min.first, min.second)) {
            min = { c1.id(), c2.id() };
        }
    }
}
```

Добавляем новый кластер и расстояния:

```
auto[x, y] = min;
auto new_cluster = merge_clusters(x, y);
for (auto& c: active_clusters) {
    m_dm_clusters(new_cluster.id(), c.id()) =
```

```
        std::min(m_dm_clusters(x, c.id()),
                 m_dm_clusters(y, c.id()));
    m_dm_clusters(c.id(), new_cluster.id()) =
        std::min(m_dm_clusters(c.id(), x),
                 m_dm_clusters(c.id(), y));
}
m_dm_clusters(new_cluster.id(), new_cluster.id()) = 0;
```

Добавляем новый кластер в активные кластеры, удаляем старые:

```
m_clusters.emplace_back(new_cluster);
active_clusters.insert(new_cluster);
active_clusters.erase(m_clusters[x]);
active_clusters.erase(m_clusters[y]);
```

После прохождения цикла берём последний добавленный кластер за корень дерева кластеров:

```
m_root = &(*m_clusters.rbegin());
```

Результаты

Список литературы

[1] OSM Contributors. OpenStreetMap main page.

[2] OSM Contributors. OpenStreetMap Node.

[3] OSM Contributors. OpenStreetMap Way.

[4] OSM Contributors. OpenStreetMap Wiki.

[5] OSMCode Community. Libosmium main page.

[6] OSMCode Community. Libosmium source code.

[7] Ramm, F. OSM Help.

[8] Schneider, W. BBBike extract page.

[9] Veness, C. Calculate distance and bearing between latitude/longitude points.