



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

<b>Постановка задачи и методы решения</b>	<b>2</b>
Построение графа дорог . . . . .	2
Оценка удобства размещения объектов инфраструктуры . . . . .	8
Выбор зданий . . . . .	8
Дерево кратчайших путей . . . . .	9
Планирование новых объектов . . . . .	11
Разбиение на кластеры . . . . .	11
Центроиды и дерево кратчайших путей для кластеров . . .	15
<b>Результаты</b>	<b>15</b>
<b>Список литературы</b>	<b>16</b>

## 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* [3] определяет следующие структуры данных для хранения объектов географических карт: *Node*, *Way* и *Relation* [6].

Структура *Node* (узел) [4] 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) [5] есть упорядоченный список узлов, которая либо имеет по крайней мере одну метку, либо принадлежит *Relation*.



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 [9], 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 гаверсина [11]:

$$\eta(\Theta) = \eta(\varphi_2 - \varphi_1) + \cos(\varphi_1) \cos(\varphi_2) \eta(\lambda_2 - \lambda_1),$$

$$d = 2r \arcsin(\sqrt{\eta(\Theta)}),$$

где  $\Theta$  — 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  $\theta$ .

```
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 [8].

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

### Выбор зданий

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  $O(n \log n)$  [1].

```
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 определить ближайшие объекты, так и построить дерево кратчайших путей.

### Дерево кратчайших путей

Для построения дерева кратчайших путей от объекта до выбранных узлов требуется выполнить обход, запоминая для каждой вершины её предка. Введём дополнительный тип данных, в котором для каждого узла будем хранить его предка.

```

using Trail = unordered_map<Node, Node>;

auto dijkstra(Node s) -> pair<ShortestPaths, Trail> {
    ...
    Trail trail;
    ...
    if (ds[v] + length < ds[to]) {
        set.erase({ ds[to], to });
        ds[to] = ds[v] + length;
        trail[to] = v;
    }
}

```

```

        set.insert({ ds[to], to });
    }
    ...
    return { ds, trail };
}

```

Далее преобразуем Trail в TracedPath.

```

auto shortest_paths(Building from, Buildings& to) {
    auto [distances, trail] = G.dijkstra(from);
    TracedPaths result;

    for (const auto& to: tos) {
        auto distance = distances[to];

        // Reconstruct path.
        vector<Node> path;
        for (auto v = node;
             v != from; v = trail[v]) {
            path.push(v);
        }
        path.push(from);
        reverse(path.begin(), path.end());

        result.push({ from, to, path, distance });
    }

    return result;
}

```

Пути объединяем в новый объект карты Map:

```

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

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

```

```

        auto pred = path.begin();

        for (auto curr: path) {
            if (curr == pred) { continue; }
            // Weight of the edge from pred to curr.
            auto w = map.find(pred).find(curr);
            routes.add_edge({ pred, curr }, w);
            pred = curr;
        }
    }

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

    return Map { buildings, routes };
}

```

В итоге получено дерево кратчайших путей. Для нахождения общей длины дерева достаточно сложить веса рёбер.

## Планирование новых объектов

### Разбиение на кластеры

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

В `Cluster` хранится информация о кластере: размер, индексы элементов и т. д. Так как на каждом шаге метода полной связи мы объединяем 2 кластера, то будем их хранить.

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 потребуется матрица расстояний:

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

Матрицу мы заполняем результатами вызова алгоритма Дейкстры для каждой вершины.

Теперь перейдём к самой реализации [12]. 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.

```

unordered_set<Cluster> ac(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.

```

pair<size_t, size_t> min = { -1, -1 };
for (auto& c1: ac) {
    for (auto& c2: ac) {
        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: ac) {
    m_dm_clusters(new_cluster.id(), c.id()) =
        min(m_dm_clusters(x, c.id()),
            m_dm_clusters(y, c.id()));
    m_dm_clusters(c.id(), new_cluster.id()) =
        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);
ac.insert(new_cluster);
ac.erase(m_clusters[x]);
ac.erase(m_clusters[y]);

```

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

```

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

```

## Центроиды и дерево кратчайших путей для кластеров

Для нахождения центроида кластера требуется найти среднее значение координат домов, входящих в него. Однако, если для двух кластеров, у которых мы делаем новый, уже известны центроиды, то найти центроиду для нового кластера можно по следующей формуле:

$$c_{new} = \frac{c_1 s_1 + c_2 s_2}{s_1 + s_2}$$

где:  $c_{new}$  — новая центроида,  $c_1, c_2, s_1, s_2$  — центроиды и размеры кластеров соответственно.

После нахождения геометрической центроида найдём центроиду на карте, то есть постройку, которая находится ближе всего к геометрической центроиде.

```
Location loc;  
loc.first = (  
    cl1.centroid().location().first * cl1.size() +  
    cl2.centroid().location().first * cl2.size())  
    / (cl1.size() + cl2.size());  
loc.second = (  
    cl1.centroid().location().second * cl1.size() +  
    cl2.centroid().location().second * cl2.size())  
    / (cl1.size() + cl2.size());  
  
auto b = find_nearest_building(m_map, loc);
```

Нахождение дерева кратчайших путей и прочих характеристик обсуждались ранее (см. Дерево кратчайших путей).

## Результаты

Таким образом, мы, используя данные проекта *OpenStreetMap* и свободно распространяемые библиотеки,

- построили граф дорог Нижнего Новгорода и связали с ним здания, расположенные в городе,



- провели оценку удобства размещения объектов инфраструктуры города, используя расстояние между объектами как основную метрику измерения удобства,
- спланировали размещение новых объектов, используя кластеры,
- предоставили средства визуализации для просмотра отмеченных узлов и путей.

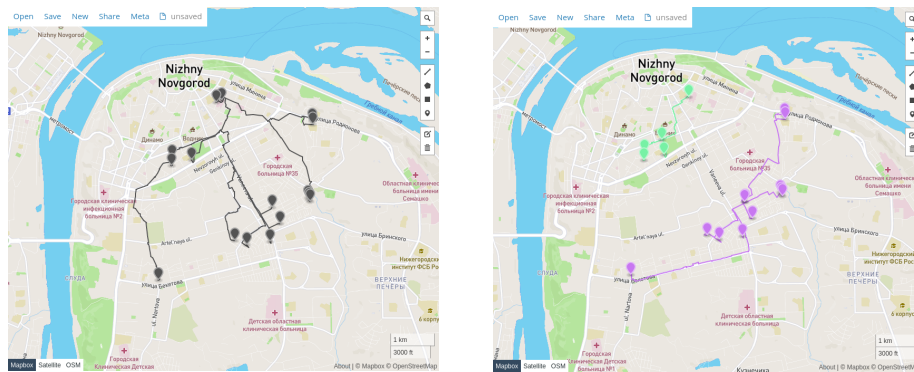


Рис. 2: Примеры кластеров

Исходный код вместе с инструкцией по сборке и использованию размещен на *GitHub* [2]. Каждый желающий может воспользоваться исходным кодом приложения, следуя условиям свободной лицензии, и воспроизвести получившиеся результаты.

## Список литературы

- [1] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.
- [2] KOZYREV, S., AND KUKLIN, D. Source code.
- [3] OSM CONTRIBUTORS. OpenStreetMap main page.
- [4] OSM CONTRIBUTORS. OpenStreetMap Node.
- [5] OSM CONTRIBUTORS. OpenStreetMap Way.

- [6] OSM CONTRIBUTORS. OpenStreetMap Wiki.
- [7] OSMCODE COMMUNITY. Libosmium main page.
- [8] OSMCODE COMMUNITY. Libosmium source code.
- [9] RAMM, F. OSM Help.
- [10] SCHNEIDER, W. BBBike extract page.
- [11] VENESS, C. Calculate distance and bearing between latitude/longitude points.
- [12] WIKIPEDIA. Complete-linkage clustering.