Проверка планарности графов

Рагулин Владимир

Оглавление

1. Введение

В этой работе будет рассмотрена *задача о плоской укладке графа*, сформулированная следующим образом: Как изобразить данный граф на плоскости так, чтобы никакие два его ребра не пересекались.

Область применения алгоритма для решения этой задачи достаточно велика. В качестве примеров можно привести прокладывание различных коммуникаций, проектирование и изготовление печатных плат в электротехнике.

Нас будет интересовать ответ на вопрос, при каких условиях граф можно нарисовать на плоскости, и как это сделать.

2. Основные определения

2.1. Графы

[1] Пусть V — *множество вершин*, E — *множество рёбер* (неупорядоченных пар $(u, v), u, v \in V$). Тогда $G = (V, E)$ называется *неориентированным графом*.

Если E — мультимножество, то G — *мультиграф*, а рёбра, встречающиеся в E больше одного раза называются *кратными*. Ребро (v, v) называется *петлёй*.

Граф называется *простым*, если он не содержит ни петель, ни кратных рёбер.

Вершины u и v называются *соседними* если существует ребро (u, v).

Ребро (u, v) *инцидентно* вершинам u, v и наоборот. *Степень* вершины означает количество инцидентных ей рёбер.

Путь $p : v \rightsquigarrow w$ — это последовательность вершин и соединяющих их рёбер, начинающаяся в v и заканчивающаяся в w. Путь называется *простым*, если все его вершины различны.

Путь $p : v \rightsquigarrow w$ и ребро (w, v) образуют *цикл*. Два цикла различны, если невозможно получить один из другого циклическим сдвигом.

Если $E' \subseteq E$, а $V' = V(E')$ содержит все вершины из V, инцидентные хотя бы одному ребру из E', то $G' = (V', E')$ является *подграфом* $G = (V, E)$.

2.2. Связность и двусвязные компоненты

Граф называется *связным*, если между любыми двумя вершинами существует путь.

Связный граф *двусвязен*, если удаление из графа любой вершины не нарушает связности графа.

Мостом называется ребро, удаление которого нарушает связность графа.

Точкой сочленения называется вершина, удаление которой нарушает связность графа.

Утверждение 1: Вершины моста являются точками сочленения

Утверждение 2: Граф, в котором нет точек сочленения двусвязен

Двусвязной компонентой графа называется такой его двусвязный подграф, что если добавить к нему любую вершину графа, то он перестанет быть двузвязным.

Утверждение 3: Разбиение графа на двусвязные компоненты единственно

2.3. Укладки и планарные графы

Укладкой графа на плоскость называется сопоставление вершинам графа точек на плоскости, а рёбрам — жордановых кривых, соединяющих соответствующие вершины, при которой кривые, соответствующие рёбрам пересекаются только в вершине, инцедентной обоим рёбрам.

Граф, у которого существует укладка на плоскость называется *планарным*.

Теорема 4*(Фари)[2]***:** У планарного графа существует укладка, в которой все рёбра сопоставлены прямолинейным отрезкам

Плоский граф — граф, уже уложенный на плоскости.

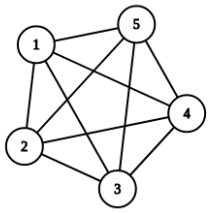
Плоский граф разделяет плоскость на связные области — *грани*.

Теорема 5*(Формула Эйлера)[3]*: Для простого связного плоского графа с n вершинами, m рёбрами и f гранями выполняется соотношение:

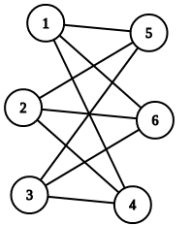
$$n - m + f = 2$$

Ровно одна из граней неограничена. Она называется *внешней*. Остальные грани называются *внутренними*.

Самые известные примеры непланарных графов:



K_5 — полный граф на пяти вершинах



$K_{3,3}$ — Полный двудольный граф с тремя вершинами в каждой доле

Теорема 6*(Понтрягин-Куратовский) [4]*: Граф планарен тогда и только тогда, когда не содержит подграфов, гомеоморфных K_5 и $K_{3,3}$

Это условие красиво, но не создаёт эффективного способа проверки на планарность.

3. Описание алгоритма

3.1. Проверка на планарность

3.1.1. Дополнительные условия

Мы будем пользоваться *гамма-алгоритмом* [5]. Он требует, чтобы графы, подаваемые на вход, обладали следующими дополнительными свойствами:

1. Граф связный
2. Граф содержит хотя бы один цикл
3. Граф не содержит точек сочленения

Заметим, что эти свойства эквивалентны следующему: граф двусвязен.

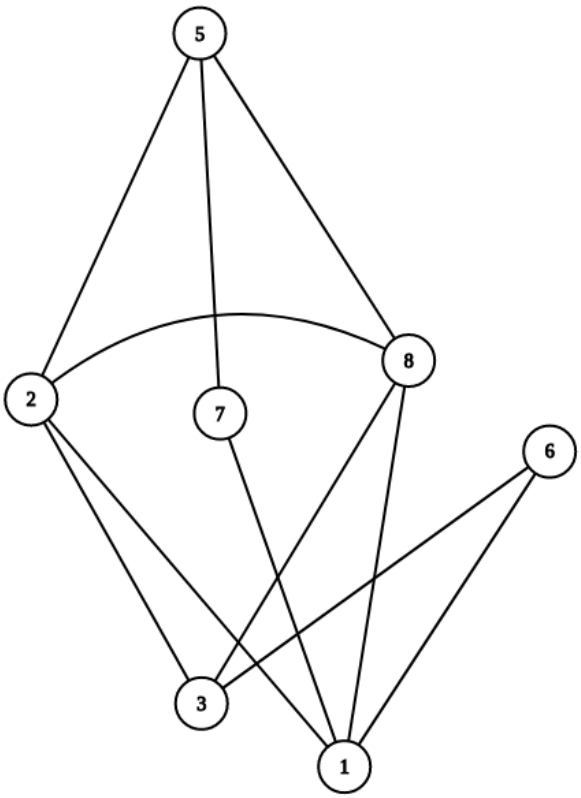
Если нарушено первое свойство, то граф нужно укладывать отдельно по компонентам связности.

Если нарушено второе свойство, то граф — дерево и его укладка тривиальна.

Если нарушено третье свойство, то уложим граф по компонентам двусвязности. Будем рисовать граф по компонентам: каждую новую компоненту будем рисовать в грани, содержащей общую точку сочленению. Мы сможем получить плоскую укладку, потому что граф со сжатыми компонентами двусвязности представляет собой дерево.

3.1.2. Пример выполнения

Опишем шаги алгоритма на примере:



В алгоритме мы будем поддерживать частичную укладку графа и первый шаг — её создание.

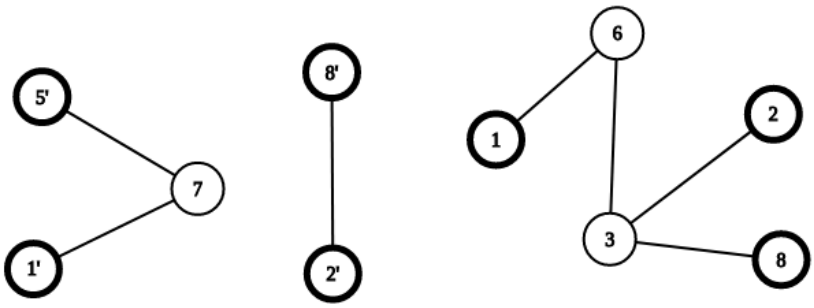
Выбираем любой простой цикл в G (возьмём, например, $\{1, 2, 5, 8\}$), укладываем его на плоскость. Теперь в нашей частичной укладке есть две одинаковые по составу вершин грани — внешняя и внутренняя.

Будем обозначать уже уложенную часть G'.

На каждом шаге будем поддерживать множество *сегментов*. Все сегменты можно разделить на два типа:

- ребро, оба конца которого уже уложены, но оно само — ещё нет.
- связная компонента графа $G \,/\, G'$, дополненная всеми рёбрами G, один конец которых принадлежит компоненте, а второй — G'

Контактными назовем вершины, принадлежащие одновременно G' и какому-то сегменту.



Сегменты графа после инициализации. Жирным помечены контактные вершины

Исходный граф двусвязен, поэтому каждый сегмент имеет хотя бы две контактные вершины.

Будем говорить, что грань Γ *вмещает* сегмент S если она содержит все его контактные вершины.

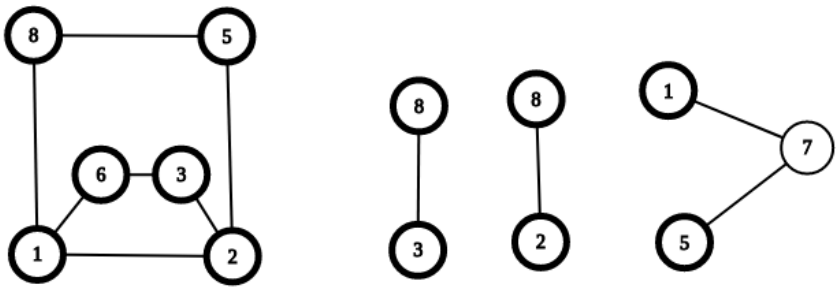
Множество граней, вмещающих наш сегмент, назовём $\Gamma(S)$.

Общий шаг алгоритма:

- Выделим все сегменты S_i
- Если сегментов нет, значит граф полностью уложен и решение найдено
- Для каждого сегмента определим $|\Gamma(S_i)|$
- Если есть сегмент, для которого $|\Gamma(S_i)| = 0$, то мы не можем его никуда вписать и граф не планарен
- Иначе возьмём сегмент S с минимальным $|\Gamma(S_i)|$, выберем какие-нибудь две его контактные вершины u, v, найдём путь $p : u \rightsquigarrow w$ в сегменте и добавим этот путь в какую-нибудь грань $\Gamma \in \Gamma(S)$. При этом Γ будет разделена на две грани, а S исчезнет или распадётся на новые сегменты

В нашем примере все сегменты вмещаются в обе грани, значит можем выбрать любой. Выберем, например, больший и найдём в нём путь между двумя контактными вершинами (пусть это будут $1, 2$).

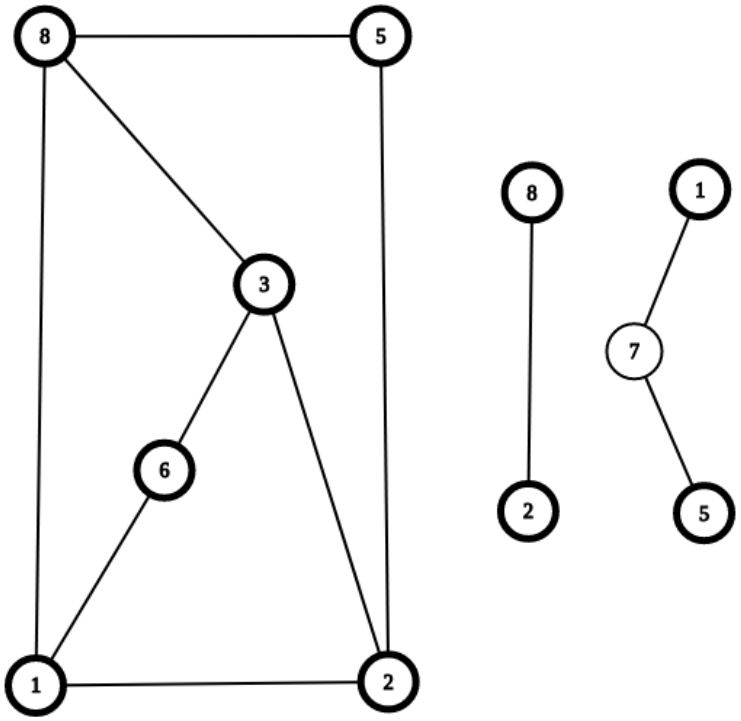
Добавим в укладку путь $\{1, 6, 3, 2\}$ и посмотрим, как изменилось множество сегментов:



Промежуточная укладка и сегменты графа после первого шага. Жирным помечены контактные вершины

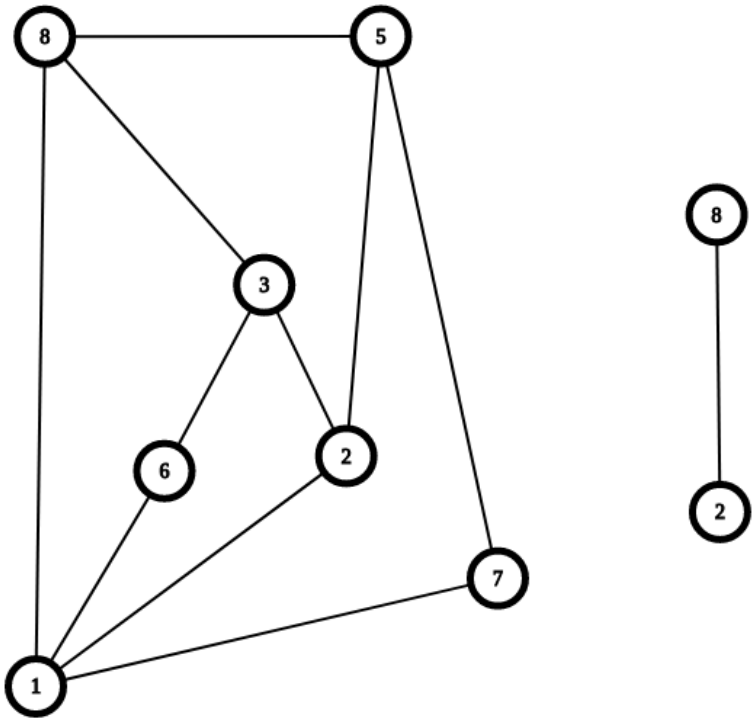
Теперь для сегментов $(2, 8)$ и $(1, 5, 7)$ всё ещё существует по две грани, вмещающие их, а для появившегося сегмента $(3, 8)$ существует только одна.

Значит, сейчас мы должны вписать этот сегмент:



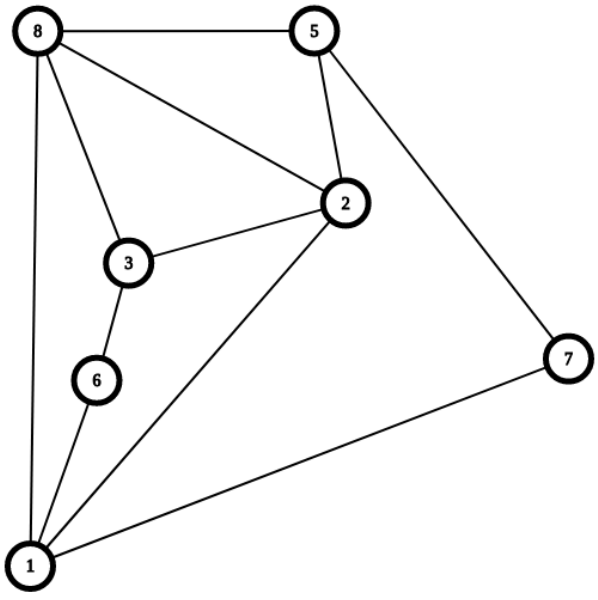
Промежуточная укладка и сегменты графа после второго шага

Для обоих сегментов $|\Gamma(S)| = 2$, поэтому выберем любой и впишем, например, во внешнюю грань:



Промежуточная укладка и сегменты графа после третьего шага

Впишем последний сегмент в грань, вмещающую его и получим укладку графа на плоскость.



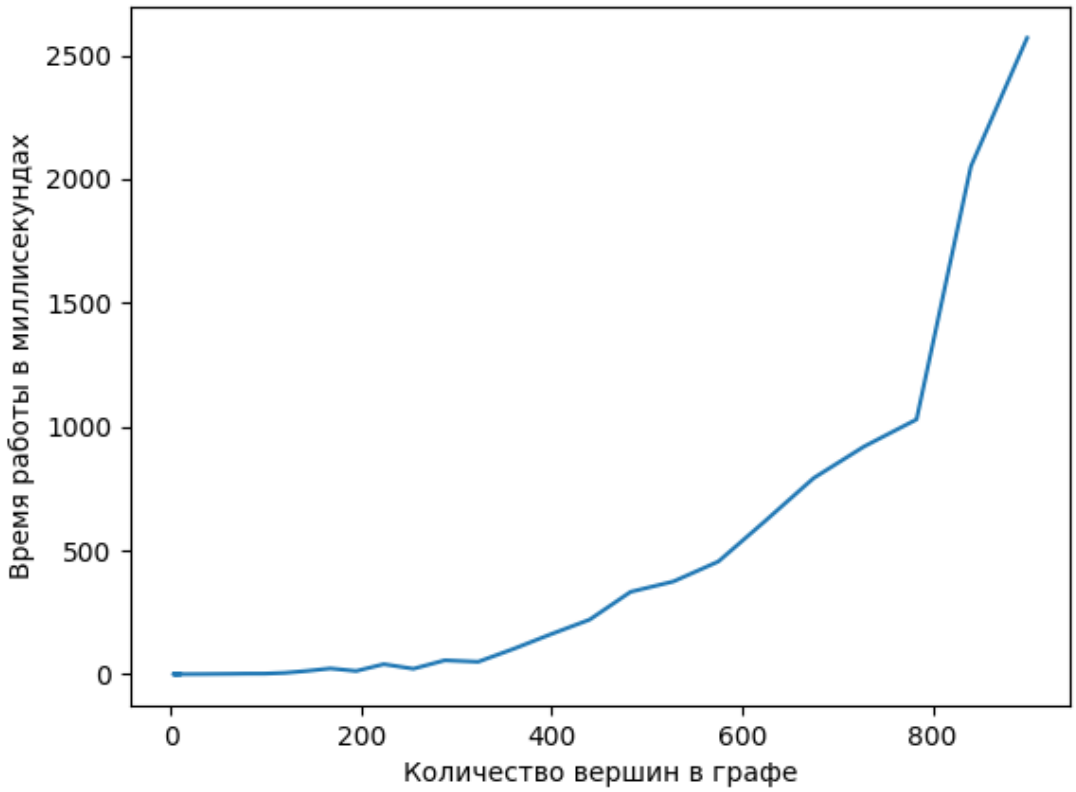
Финальная укладка

3.1.3. Краткое описание алгоритма

1. *Инициализация*. Выберем произвольный цикл и уложим его не плоскость, определим сегменты и вмещающие их грани.
2. Пока множество сегментов не пусто:
   - *Общий шаг*
   - Выберем сегмент S с минимальным $|\Gamma(S)|$. Если $|\Gamma(S)| = 0$, то граф не планарен. Конец.
   - Выберем две произвольные контактные вершины u, v сегмента S.
   - Найдём в сегменте путь $p : u \rightsquigarrow w$.
   - Добавим этот путь в любую грань, вмещающую S.
   - Обновим множество граней и множество сегментов.
3. *Завершение*. Мы определили планарность графа и, если он планарен, получили список его граней.

3.1.4. Ассимптотика

Мы можем честно оценить сверху время работы нашего алгоритма как $O(n^3)$, впрочем, некоторые исследователи утверждают, что определённые реализации работают за $O(n^2)$.



А вот такую картину дают экспериментальные результаты

Этот алгоритм не является оптимальным по времени. В 1974 году Хопкрофт и Тарьян описали линейный алгоритм проверки на планарность [6]. Но он чрезвычайно сложен как для понимания, так и для реализации.

3.1.5. Корректность алгоритма и замечания

Корректность алгоритма доказана в источниках.

Некоторые источники утверждают, что если на общем шаге выбирать сегмент не с минимальным числов вмещающих граней, а по следующему принципу:

- Если есть сегмент S, $|\Gamma(S)| = 1$, то выберем его (из таких — любой).
- Иначе можно брать любой сегмент.

то алгоритм тоже будет корректно работать.

3.2. Визуализация планарных графов

Заметим, что из гамма-алгоритма, несмотря на его простоту и понятность, достаточно сложно получить конкретную отрисовку графа (сопоставление вершинам точек на плоскости).

В работе было попробовано несколько вероятностных методов отрисовки графа (не опирающихся на результат гамма-алгоритма), но в итоге визуализация реализована с использованием встроенных методов библиотеки *networkx*.

4. Реализация

Код и комментарии к нему можно найти в репозитории на GitHub.

Реализация алгоритма несколько осложнена необходимостью предварительно разделить его на двусвязные компоненты. Поэтому план обработки графа такой:

1. Выделить компоненты связности, для каждой компоненты выполнять независимо
2. Найти и удалить мосты
3. Рекурсивно (с помощью поиска в глубину) красить компоненты двусвязности, меняя цвет при проходе через точку сочленения и в момент выхода из компоненты проверять её на планарность гамма-алгоритмом

4.1. Поиск мостов

Поиск мостов и точек сочленения — одна из стандартных задач на графы.

```
vector<pair<int, int>> bridges;
void find_bridges(int v, int parent) {
    color[v] = 1;
    dp[v] = height[v];
    for (int to : g[v]) if (to != parent) {
        if (color[to] == 0) {
            height[to] = height[v] + 1;
            find_bridges(to, v);
            dp[v] = min(dp[v], dp[to]);
            if (dp[to] > height[v]) bridges.emplace_back(v, to);
        } else {
            dp[v] = min(dp[v], height[to]);
        }
    }
}
```

4.2. Окрашивание компонент двусвязности

Будем обходить граф поиском в глубину, проходя через точку сочленения меняем цвет. Выходя из компоненты двусвязности, запускаем `check()` — гамма-алгоритм, проверяющий компоненту на планарность.

```
int max_color = 1;
void dfs(int v, int c) {
    color[v] = c;
    for (int to : g[v]) if (color[to] == 0) {
        if (dp[to] >= height[v]) {
            int new_color = max_color++;
            dfs(to, new_color);
            color[v] = new_color;
            if (!check(new_color)) output_result(false);
            color[v] = c;
        } else {
            dfs(to, c);
        }
    }
}
```

4.3. Непосредственно Гамма-алгоритм

Реализуем структуры граней и сегментов:

```
//Для грани храним множество вершин и вершины в порядке обхода.
struct Face {
    set<int> vts;
    vector<int> tour;
    explicit Face(vector<int> & a) : tour(a) {
        for (int el : tour) vts.insert(el);
    }
    Face() = default;
};
//Для сегмента храним множество контактных вершин, множество вмещающих граней и
одно из рёбер
struct Segment {
    set<int> touch;
    vector<int> good_faces;
    int v_start, id_start;
    Segment(int v, int id) : v_start(v), id_start(id) {}
};
```

Реализуем функцию нахождения пути:

```
//Эта функция находит путь из v в targ и записывает его рёбра в cur_path
bool find_path(int v, int targ, vector<int> & cur_path) {
    color2[v] = max_color2;
    for (int id : g2[v]) if (id != cur_path.back()) {
        int to = v ^ E[id];
        if (to == targ) {
            cur_path.emplace_back(id);
            return true;
        }
        if (!placed.contains(to) && color2[to] != max_color2) {
            cur_path.emplace_back(id);
            if (find_path(to, targ, cur_path)) return true;
            cur_path.pop_back();
        }
    }
    return false;
}
```

И функцию, которая будет помечать рёбра сегмента (и выписывать его контактные вершины):

```
void paint_segment(int v, int c, set<int> & touching_vtx) {
    for (int id : g2[v]) if (state[id] != c) {
        state[id] = c;
        int to = v ^ E[id];
        if (!placed.contains(to)) paint_segment(to, c, touching_vtx);
        else touching_vtx.insert(to);
    }
}
```

Основная функция программы `bool check()` выглядит достаточно монструозно:

```
/*
 * This is most important (and the longest) function
 * it checks if current two-connected component (its vertexes should be colored
with c) is planar
 */
bool check(int c) {
    /* Initializing */
    vtx.clear();
    vtx_set.clear();
    placed.clear();
    int top = 0;
    for (auto & ar : g2) ar.clear();
    for (int v = 0; v < n; ++v) {
        if (color[v] == c) {
            vtx.emplace_back(v);
            vtx_set.insert(v);
        }
    }
    for (int v : vtx) for (int to : g[v]) if (vtx_set.contains(to)) {
        if (v < to) {
            g2[v].emplace_back(top);
            g2[to].emplace_back(top);
            E[top] = v ^ to;
            top++;
        }
    }
    { /* For planar two-connected graph m <= 3n - 6. Checking this. */
        int edges_cnt = 0;
        for (int v: vtx) edges_cnt += sz(g2[v]);
        int n_cur = sz(vtx);
        edges_cnt /= 2;
        if (edges_cnt + 1 == n_cur) return true;
        if (edges_cnt > 3 * n_cur - 6) return false;
    }
    vector<Face> faces;
    { /* Adding primary cycle and faces */
        vector<int> cur_path, cycle;
        int v = vtx[0];
        assert(!g2[v].empty());
        cur_path.emplace_back(g2[v][0]);
        max_color2++;
        assert(find_path(v ^ E[g2[v][0]], v, cur_path));
        for (int id : cur_path) {
            placed.insert(v);
            cycle.emplace_back(v);
            v ^= E[id];
            state[id] = 2;
        }
        faces.emplace_back(cycle);
        faces.emplace_back(cycle);
    }
    vector<Segment> segments;
    { /* Computing segments */
        for (int v : placed) {
            for (int id : g2[v]) if (state[id] == 0) {
```

```
                segments.emplace_back(v, id);
                int to = v ^ E[id];
                if (placed.contains(to)) {
                    segments.back().touch = {v, to};
                    state[id] = 1;
                } else {
                    paint_segment(v ^ E[id], 1, segments.back().touch);
                }
            }
        }
    }
    while (!segments.empty()) {
        /* COMMON STEP */
        int id = 0;
        int s = sz(segments);
        /* For each segment s we compute Gamma(s) (segments[i].good_faces) */
        for (int i = 0; i < s; ++i) {
            segments[i].good_faces.clear();
            for (int j = 0; j < sz(faces); ++j) {
                bool ok = true;
                for (int x : segments[i].touch) {
                    if (!faces[j].vts.contains(x)) {
                        ok = false;
                        break;
                    }
                }
                if (ok) {
                    segments[i].good_faces.emplace_back(j);
                }
            }
            /* we want to find id that sz(segments[id].good_faces) is minimal */
            if (sz(segments[i].good_faces) < sz(segments[id].good_faces)) id = i;
        }
        /* if |Gamma(s)| == 0 our graph is not planar */
        if (segments[id].good_faces.empty()) return false;
        swap(segments[id], segments[s - 1]);
        auto & [touch, good_faces, v_start, id_start] = segments.back();
        int targ = *touch.begin();
        if (targ == v_start) targ = *touch.rbegin();
        /* now we want to add path v_start -> targ to our embedding */
        if (!placed.contains(v_start ^ E[id_start]))
            paint_segment(v_start ^ E[id_start], 0, touch);
        vector<int> cur_path{id_start};
        max_color2++;
        /* if our path is not a single edge we call find_path */
        if (!placed.contains(v_start ^ E[id_start]))
            assert(find_path(v_start ^ E[id_start], targ, cur_path));
        /* we list all vertexes of path to path_vtx */
        int cur_v = v_start;
        vector<int> path_vtx{cur_v};
        for (int x : cur_path) {
            state[x] = 2;
            cur_v = cur_v ^ E[x];
            path_vtx.emplace_back(cur_v);
            placed.insert(cur_v);
```

```
        }
        /* this code splits Face into two by our path */
        vector<int> fst, snd;
        auto & f = faces[good_faces[0]];
        int pos = 0;
        while (f.tour[pos] != v_start) ++pos;
        for (pos = (pos + 1) % sz(f.tour); f.tour[pos] != targ; pos = (pos + 1) %
sz(f.tour)) {
            snd.emplace_back(f.tour[pos]);
        }
        for (pos = (pos + 1) % sz(f.tour); f.tour[pos] != v_start; pos = (pos + 1)
% sz(f.tour)) {
            fst.emplace_back(f.tour[pos]);
        }
        for (int v : path_vtx) fst.emplace_back(v);
        std::reverse(path_vtx.begin(), path_vtx.end());
        for (int v : path_vtx) snd.emplace_back(v);
        swap(faces[good_faces[0]], faces.back());
        /* remove old face and segment, add new faces */
        faces.pop_back();
        faces.emplace_back(fst);
        faces.emplace_back(snd);
        segments.pop_back();
        /* compute new segments */
        for (int v : path_vtx) {
            for (int x : g2[v]) if (state[x] == 0) {
                segments.emplace_back(v, x);
                int to = v ^ E[x];
                if (!placed.contains(to)) {
                    paint_segment(to, 1, segments.back().touch);
                } else {
                  segments.back().touch = {v, to};
                }
            }
        }
    }
    return true;
}
```

5. Руководство пользователя

Чтобы проверить ваш собственный граф на планарность и нарисовать его укладку можно поступить следующим образом:

1. Склонировать репозиторий на GitHub
2. Записать свой граф в файл `stream.in` в следующем формате:
   - в первой строке должно находиться два целых числа `n, m` — число вершин и число рёбер графа соответственно
   - каждая из следующих `m` строк должна содержать пару чисел `a, b` $(1 \leq a, b \leq n)$ — концы очередного ребра
3. Запустить `run.py`
4. Результат проверки будет выведен в консоль, и если граф планарен, то его укладка будет сохранена в `mygraph.svg`

6. Используемая литература

[1] F. Harary, “Graph theory,” 1973.

[2] D. Wood, “A simple proof of the fary-wagner theorem,” 2005.

[3] М. Асанов, В. Баранский, and В. Расин, “Дискрентная математика. Графы, матроиды, алгоритмы,” 2001.

[4] К. Куратовский, “Топология,” 1966.

[5] В. Каширин, and А. Иринёв, “Алгоритм плоской укладки графов,” 2006.

[6] J. Hopcroft, and R. Tarjan, “Efficient planarity testing,” 1974.