ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет информатики, математики и компьютерных наук

**Программа подготовки бакалавров по направлению
01.03.02 Прикладная математика и информатика**

Созинов Кирилл Игоревич

КУРСОВАЯ РАБОТА

Алгоритмы проверки планарности графов

Научный руководитель

к.ф-м.н., преп. каф. ПМИ.

Д. С. Талецкий

Нижний Новгород, 2021

Содержание

Введение

Проверка планарности графа — это алгоритмическая задача проверки, является ли данный граф *планарным* (то есть, может ли он быть нарисован на плоскости без пересечения рёбер). Задача хорошо изучена в информатике и для неё было придумано много практических алгоритмов, многие из которых используют современные структуры данных.

Плоские графы имеют ключевую роль во многих задачах вычислительной геометрии. Представление химический структур как планарных графов значительно упрощают определение их изоморфизма. Другим примером является встраивание сети компонентов в чип. Компоненты представлены проводами, и никакие два провода не могут пересекаться без короткого замыкания. Эту проблему можно решить, рассматривая сеть как граф и находя ее плоское вложение. Также можно рассмотреть транспортную сеть, моделирование которой при помощи построения плоского графа позволит избежать лишних пересечений транспортных путей.

Целью данной работы является изучение, сравнительный анализ а также реализация этих алгоритмов.

Теоретическая часть

Определение 1. *Графом* называется пара $G = (V, E)$, где V - множество вершин графа, а E - множество ребер.

Мы будем рассматривать конечные ориентированные и неориентированные графы без петель и кратных ребер. Если каждое ребро представляет собой неупорядоченную пару различных вершин, то граф *неориентированный*. Если каждое ребро представляет собой упорядоченную пару различных вершин, то граф *ориентированный*. Если (v, w) является ребром в ориентированном графе, мы говорим, что ребро покидает v и входит в w.

Определение 2. Граф $G' = (V', E')$ называется *подграфом* графа $G = (V, E)$, если $V' \subseteq V$ и $E' \subseteq E$.

Определение 3. Последовательность вершин $v_i, 1 \leqslant i \leqslant n$, и ребер $e_i, 1 \leqslant i \leqslant n$, таких как $e_i = (v_i, v_i + 1)$, называется *путем* графа G из v_1 в v_n.

Вершина w достижима из вершины v, если существует путь от w до v. Путь называется *простым*, если все вершины в нём различны. Путь от вершины к самой себе называется *замкнутым путем*.

Определение 4. Замкнутый путь от v до v называется *циклом*, если все его ребра различны, и единственная вершина, встречающаяся ровно 2 раза – v.

Определение 5. Неориентированный граф G называется *связным*, если любая вершина в этом графе достижима из любой другой вершины.

Максимальные связные подграфы G называются *компонентами связности*.

Определение 6. Вершина, при удалении которой количество *компонент связяности* возрастает, называется *точкой сочленения* или *шарниром*.

Если граф связный и не содержит точек сочленения, то этот граф *двусвязный*.

Определение 7. *(Направленное, корневое) дерево* T – это ориентированный граф с одной выделенной вершиной, называемой корнем r, такой, что каждая вершна в T достижима из r, при этом никакие ребра не входят в r, и ровно одно ребро входит в каждую другую вершину в T.

Определение 8. *Остовное дерево* — ациклический связный подграф данного связного неориентированного графа, в который входят все его вершины.

Определение 9. В теории графов *изоморфизмом* графов $G = (V_g, E_g)$ и $H = (V_h, E_h)$ называется биекция между множествами вершин графов $f : V_g \longrightarrow V_h$ такая, что любые две вершины u и v графа G смежны тогда и только тогда, когда вершины $f(u)$ и $f(v)$ смежны в графе H.

Если изоморфизм графов установлен, они называются *изоморфными* и обозначаются как $G \simeq H$.

Операция *подразделения ребра* подразумевает собой деление некоторого ребра e с конечными вершинами u, v. Получается граф, содержащий новую вершину w и два ребра u, w и w, v вместо ребра e.

Обратная операция, *исключение вершины* w с инцидентными ей рёбрами (e_1, e_2), заменяет смежные вершине w оба ребра (e_1, e_2) на новое ребро, соединяющее конечные точки пары. Следует подчеркнуть, что данная операция применима только к вершинам, инцидентным в точности двум рёбрам.

Рис. 1: Иллюстрация операции подразделения ребра.

Рис. 2: Иллюстрация операции исключения вершины.

Определение 10. Два графа называются *гомеоморфными*, если от одного можно перейти к другому при помощи операций подразделения ребра и обратных к ним.

Определение 11. Граф обладает *укладкой* в пространстве L, если он изоморфен графу, вершинами которого являются некоторые точки пространства, а ребрами — жордановы кривые, соединяющие соответствующие вершины, причем

- Кривая, являющаяся ребром не проходит через другие вершины графа, кроме вершин, которые она соединяет;
- Две кривые, являющиеся ребрами, пересекаются лишь в вершинах, инцидентных одновременно обоим этим ребрам.

Определение 12. Граф называется *планарным*, если он обладает укладкой на плоскости.

Плоским называется граф уже уложенный на плоскости.

Также для проверки планарности графов нам будет интересна *формула Эйлера*, а точнее её следствия:

Во-первых, все плоские укладки одного графа имеют одинаковое количество граней. Во-вторых, если каждая грань ограничена не менее чем тремя рёбрами (при условии, что в графе больше двух рёбер), а каждое ребро разделяет две грани, то

$$3|F(G)| \leqslant 2|E(G)|$$

следовательно,

$$|E(G)| \leqslant 3|V(G)| - 6$$

Это значит, что при достаточно большом количестве ребер граф *заведомо непланарен*.

Еще в XVIII веке Эйлер доказал, что графы K_5 и $K_{3,3}$ не являются планарными. Это можно доказать, использвуя перебор и следующую теорему.

Теорема 1. (Жордана). Замкнутая несамопересекающаяся прямая (цикл) делит плоскость ровно на две части. (При этом одна часть ограничена, другая неограничена, причем две точки плоскости, не принадлежащие кривой, лежат в одной части тогда и только тогда, когда их можно соединить ломаной, не пересекающей кривой) .

Доказательство этой теоремы в рамках данной работы обсуждаться не будет, т.к. оно очень сложное , но с ним можно ознакомиться в [*].

Теорема 2. (Куратовского). Граф является планарным тогда и только тогда, когда он не содержит подграфа, гомеоморфного графу K_5 или $K_{3,3}$ (рис. 3).

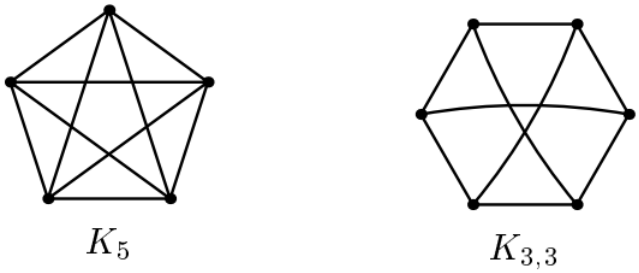

Рис. 3: Графы K_5 и $K_{3,3}$.

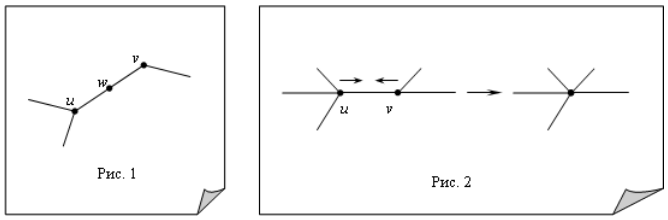

Рис. 4: Иллюстрация операции стягивания ребра.

Операция *стягивания ребра* подразумевает собой слияние вершин ребра в одну и последующее удаление ребра из графа.

Определение 13. Минор графа G — это другой граф H, полученный путём применения операций удаления вершин, удалением и стягиванием рёбер к графу G.

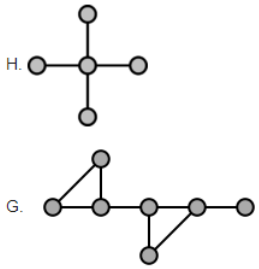

Рис. 5: Иллюстрация минора графа G (граф H).

Теорема 3. (Вагнера). Любой граф либо имеет планарное вложение, либо содержит *минор* одного из двух типов — полный граф K_5 или полный двудольный граф $K_{3,3}$ (граф может иметь оба типа миноров).

1. Алгоритмы

1.1. Метод добавления пути Хопкрофта и Тарьяна

Алгоритм Хопкрофта и Тарьяна (H&T), разработанный на основе более раннего алгоритма, предложенного Ауслендером и Партером. В 1971 году Хопкрофт и Тарьян сформулировали ограниченный по времени вариант алгоритма Гольдштейна, который они позже улучшили до $O(V)$ в диссертации Тарьяна и упростили в своей статье «Efficicient planarity testing», хотя поправки к их алгоритму были позже опубликованы Део в 1976 году.

В качестве входных данный алгоритм H&T принимает *двусвязный граф*, однако Тарьян предложил алгоритм линейного времени для разделения связного графа на двусвязные компоненты. Далее работа с этим двусвязным графом проводится в три этапа:

На первом этапе выполняется поиск в глубину по графу, генерируя при этом пальмовое дерево (также известное как дерево DFS) и производит присваивание:

- каждой вершине DFS-индекс;
- каждое ребро обозначается как дерево и заднее ребро(также можно назвать дугой), и задает ему направление.

Во время обратного отслеживания алгоритм вычисляет для каждой вершины самые низкие и вторые по величине вершины, достижимые из частей дерева, являющихся потомками этой вершины. Эти части информации впоследствии используются для:

- Блочной сортировки ребер;
- Создания списка смежности всех ребер, исходящих из этой вершины;
- Для сравнения путей, сгенерированных на втором этапе для проверки на непланарность;

Второй этап включает в себя второй обход в глубину, рассматривающий теперь только исходящие ребра из каждой вершины (которые теперь направлены и отсортированы). Сначала идентифицируется цикл в пределах графа, а затем, когда DFS возвращается и смотрит на новые ветви, идентифицируются последовательные пути. В результате граф разбивается на неразбивающиеся пути.

Заключительная, третья фаза берет эти пути в порядке генерации и рассматривает их вложение, располагая пути по мере их рассмотрения слева ли справа от разделяющего цикла.

Алгоритм Хопкрофта и Тарьяна использует два стека, обозначенных как левый и правый, для представления сторон разделяющего цикла, блок представляет собой группу путей в левом и правом стеках, где перемещение одного пути блока слева направо или наоборот меняет местами стеки для всех путей. Используя эту информацию, задача расположения графа на плоскости сводится к:

- Каждый последующий путь рассматривается по отношению к блокам на стеке таким образом, что если блок определяет положение этого нового пути (т.е. последний путь слева или справа пересекается с новым), тогда он удаляется их стека блоков. При этом:
  - Если и на левом и на правом стеке пути пересекаются, то граф не является планарным.
  - Если левый путь перескает, то пути этого блока меняются местами между левым и правыми стеками, после чего предыдущий блок считается рассмотренным.

– Если левый путь не пересекает, то расположение путей на стеках не изменяется, предыдущий блок считается рассмотренным.

Этот процесс повторяется до тех пор, пока не будет найден блок, который не пересекается с путем с обеих сторон, либо не будет найден непланарный случай. Если непланарный случай не достигается, тогда новый путь добавляется в левый стек, и из этого пути и путей из удаленных блоков формируется новый блок.

- Если достигается состояние, когда путь на вершине стека не влияет на вложение следующего пути, то тогда он может быть удален из стека (как и блок который также может удалится если это был его последний путь), и вложение может рассматриваться относительно предыдущего пути на жтом стеке.

- Дополнительный процесс требуется, когда алгоритм возвращается к точке, где все пути, разделенные циклом обработаны. В этой точке блоки, содержащие пути, ответвляющиеся от этого цикла рассмотрены, с удаленными путями, не имеющими влияния, и проверенные не непланарность (рис. 6). Если есть только пути в левом стеке, то эти блоки объединяются с предыдущим блоком, так как последующие пути могут перекрываться.

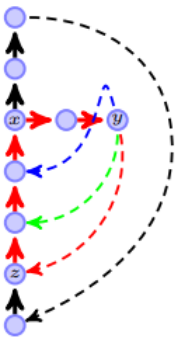

Рис. 6: Пути (зеленый и синий) слева и справа от цикла (красный), образованные путем $(x \longrightarrow y \longrightarrow z)$.

Пример алгоритма H&T. На рис. 7 приведен пример графа, который показывает порядок, в котором поиск по глубине первого этапа алгоритма H&T посещает вершины (в алфавитном порядке по возрастанию) и ребра (в числовом порядке по возрастанию). Пути, сформированные во время второго этапа, показаны на рис. 8 и пронумерованы в порядке возрастания их порядкового номера.

На рис.9 можно увидеть процесс вложения путей из рис. 8, а на рис. 10 состояние стеков во время того же процесса.

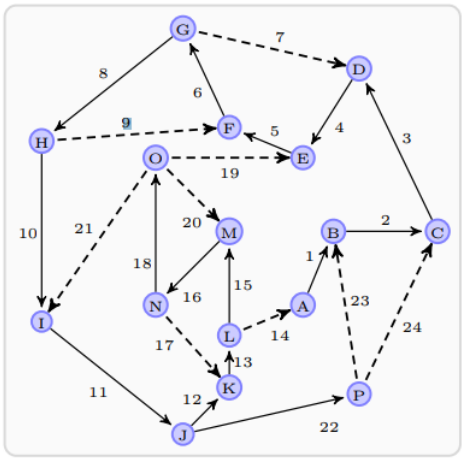

Рис. 7: Пример графа и образец пути, пройденного DFS.

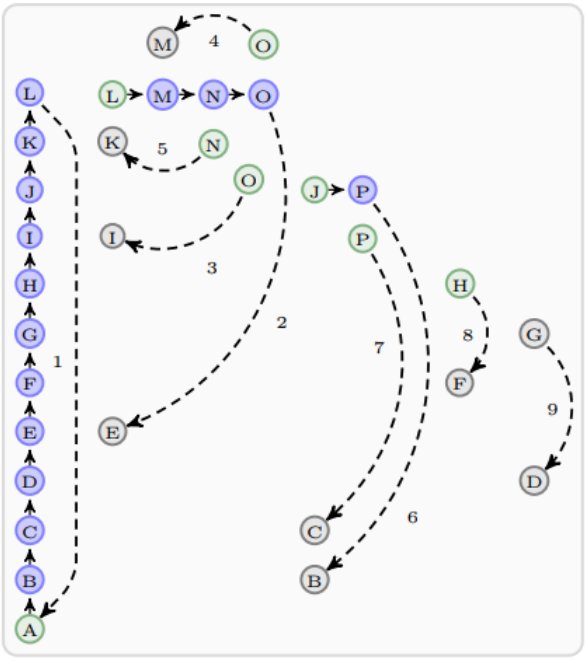

Рис. 8: Непересекающиеся пути, генерируемые второй фазой H&T.

1.2. Метод добавления вершин Ши и Сю

Метод добавления вершин заключается в создании структур данных, представляющих возможные вложения порождённого подграфа данного графа и добавления вершин по одной к этой структуре данных. Эти методы начинались с неэффективного $O(n^2)$ метода, были изменены и доработаны во многих научных статьях. В 1999 году Ши и Сю упростили эти методы, предложив свой вариант, работающий за линейное время. На их версии алгоритма и остановимся поподробнее.

Ши и Сю разработали достаточно простой относительно предыдущих алгоритм проверки планарности графа. Из теории вводится новое понятие, а именно PC-дерево, которое является аналогом PQ-дерева для некорневых деревьев).

Определение 14. *PC-дерево* – дерево, вершины которого могут быть разделены на 2 типа: *P-вершины* и *C-вершины*, где соседи *P-вершин* могут быть переставлены в любом порядке, а соседи *C-вершин* образуют циклический порядок который может быть лишь развернут.

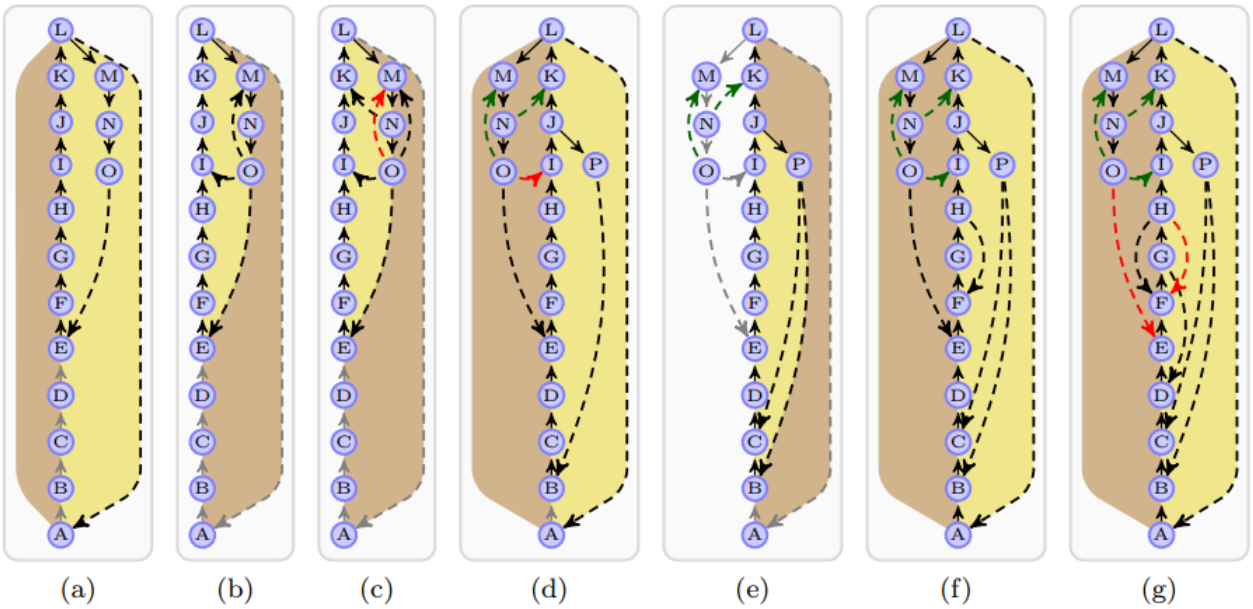

Рис. 9: Вложение путей графа.

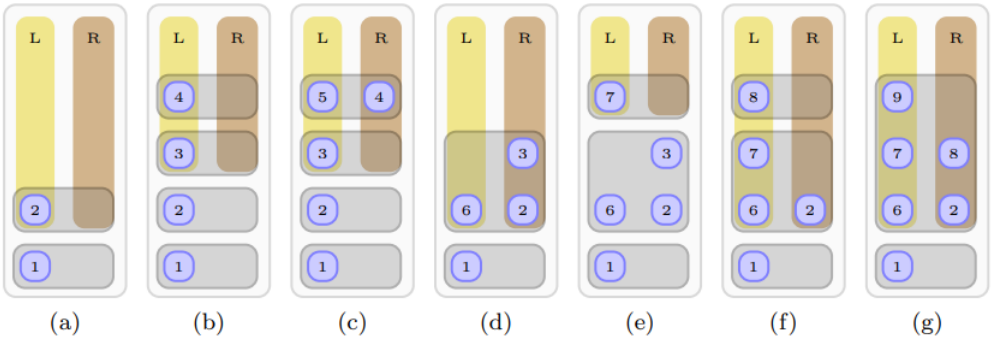

Рис. 10: Состояние левого, правого стеков, а также стека блоков во время работы алгоритма.

Говоря совсем кратко, алгоритм состоит из начальной маркировки вершин, обхода дерева для идентификации специальных *i-деревьев* и их встраивания в двусвязные компоненты.

В дальнейшем предполагаем, что граф является плоским. Обозначим наибольшего соседа узла i через $h(i)$. Каждой вершине v дерева T присвоим метку $b(v)$ следующим образом:

$b(i) = max\{h(v)|v$ - узел в $T_i\}$, где T_i – поддерево дерева T с корнем i. Далее сортируем потомков каждой вершины дерева T в соответствии с их меткой. Пусть i будет первой итерацией, на которой было найдено заднее ребро из потомка к i.

Определение 15. Терминальной вершиной назовем вершину t в T_r, которая удовлетворяет следующим условиям:

1. $b(t) > i$
2. Либо t смежен с i, либо есть потомок с меткой i
3. Ни один другой потомок не удовлетворяет ни 1. ни 2.

Определение 16. Поддерево T_j называется *i-деревом*, если каждая вершина в T_j имеет метку i.

Определение 17. Поддерево T_j называется *i*-деревом*, если каждая вершина в T_j имеет метку выше i.

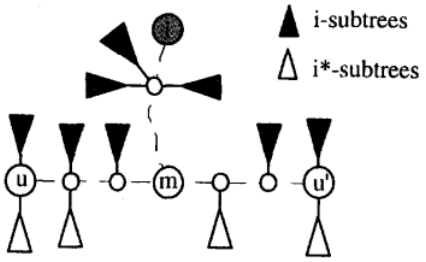

Рис. 11: i-деревья и i*-деревья.

Далее полезной для нас будет информация о том, что специально выделенные уникальные пути от одной вершины до другой находятся на границах *двусвязных компонент*. В дальнейшем эти двусвязные компоненты превращаются в -вершины, которые строятся основываясь на значении меток вершин этих компонент, а именно выделяются так называемые *основные вершины*, и *репрезентативный граничный цикл*, который они вместе с вершиной i образуют(i - первая вершина, в которую вошло заднее ребро). Пример представления C-вершин можно увидеть на рис. 12. На рис. 13 можно увидеть примеры путей дерева в через C-узлы.

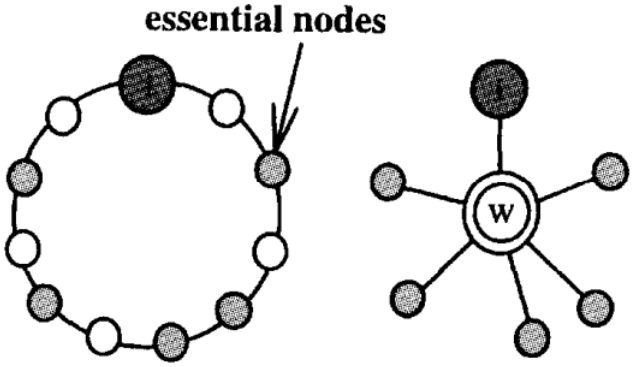

Рис. 12: Представление C-вершин.

Последний этап алгоритма заключается во вложении *PC-деревьев* на плоскость. Однако перед этим надо удалить ребро (i, r). Тогда путь от корня r к *терминальной вершине* m (в случае (i), либо к другой в зависимости от числа терминальных вершин) вместе с поддеревьями потомков образуют дерево $T(m)$. Это дерево удовлетворяет тому же свойству, что и любое *i-дерево*: ни один из узлов в дереве не имеет соседа больше i. А именно, они могут быть только смежными с i. Пример можно увидеть на рис. 14.

На плоскость мы будем укладывать деревья $T[i]$, которые состоят из пути P, *i-поддеревьев* вершин в P и дерева $T(i)$ определенного выше. Кроме того, потомки каждого C-узла в пути P будут соблюдать свой первоначальный порядок, и каждый P-узел j в пути P будет изменен на C-узел с его соседями, ориентированными особым образом.

Теперь вставим PC-дерево $T[i]$ в плоскость. Для этого дается способ упорядочения соседей i в дереве $T[i]$, который допускает плоское вложение ребер, инцидентных i, однако здесь мы его опустим. Если мы упорядочим соседей i в дереве $T[i]$ в соответствии с вышеуказанным порядком DFS, то конечное вложение будет планарным, и вершина i вместе с *терминальными вершинами* будут на внешней грани двусвязной компоненты.

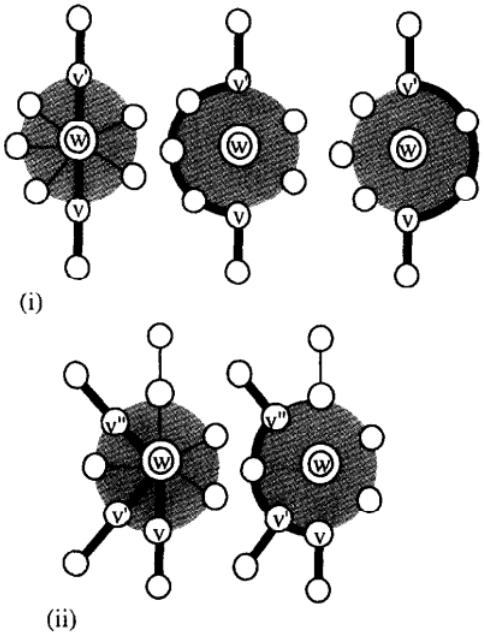

Рис. 13: Представление путей через C-вершину w. (i) – Пути через w от v до v'. (ii) – Пути через w.

1.3. Метод добавления ребер де Фрейсекса, де Мендеса и Розенштиля

Данный алгоритм также основан на DFS и деревьях Тремо, которые им порождаются. Раньше мы называли их пальмовыми деревьями или деревьями DFS.

Для формулировки алгоритма нам понадобится ряд следующих определений и формулировок:

Корневое связующее дерево $\mathcal{T}$ графа $G = (V, E)$ определяет разбиение множества ребер G на два класса: множество *ребер дерева* T и множество *задних ребер* $E \backslash T$. Он также определяет *частичный порядок* $\preceq$ на $V : x \preceq y$, если путь дерева, связывающий y с корнем $\mathcal{T}$, включает x. Корневое дерево $\mathcal{T}$ является *деревом Тремо*, если каждое заднее ребро инцидентно двум сопоставимым вершинам (относительно $\preceq$). *Дерево Тремо* $\mathcal{T}$ делает граф ориентированным: ребро x, y (с отношением $x \preceq y$) ориентировано от x до y (вверх), если это *ребро дерева*, и от y до x (вниз), если это *заднее ребро* дерева. Обозначим через $\omega^+(v)$ множество ребер, инцидентых v. Когда $\mathcal{T}$ является *деревом Тремо*, частичный порядок расширяется до $V \cup E$ (или для краткости до G) следующим образом: для любого ребра $e = (x, y)$, ориентированного из x в y, кладём $x \prec e$, и если $x \prec y$ (то есть: если e-ребро дерева), также кладем $e \prec y$.

Определение 18. Здесь мы определяем low как отображение из E в V :

$$low(e) = \begin{cases} min\{v \in V : \exists (u, v) \in E, (u, v) \succeq e\}, & \text{если e ребро дерева} \\ y & \text{, если e заднее ребро} \end{cases}$$

Определение 19. Край ребра $e = (x, y)$ обозначается как $Fringe(e)$ и определяется как

$$Fringe(e) = \{\ f \in E \backslash T : f \succeq e \text{ и } low(f) \prec x\ \}$$

Определение 20. Пусть e – *ребро дерева*:

- Если $Fringe(e) = \emptyset$, то ребро e является *блокирующим ребром*(пунктир на рисунке);

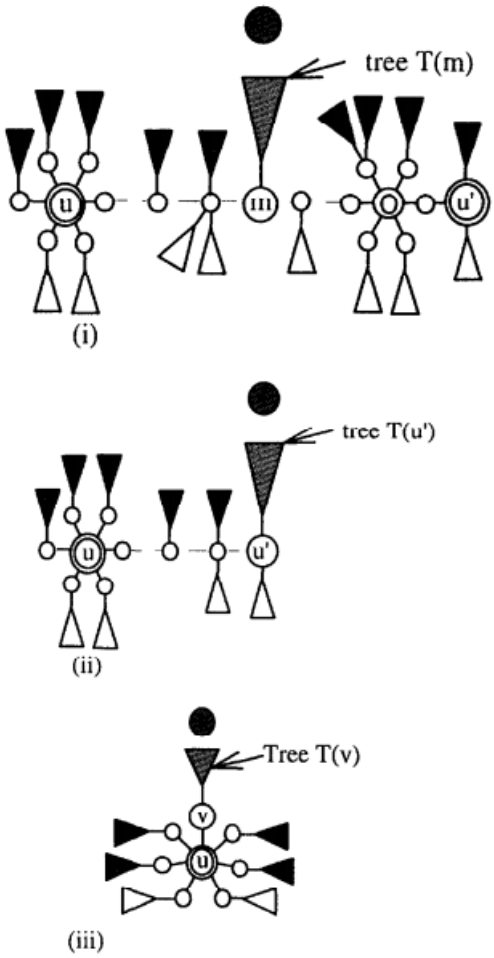

Рис. 14: Поддерево для внутреннего встраивания. (i) случай с двумя *терминальными узлами*, (ii) случай с одним *терминальным узлом* (u не является C-узлом или $u \neq u'$) и (iii) случай с одним *терминальным узлом* (u является C-узлом и $u = u'$).

- В противном случае, если все края в $Fringe(e)$ имеют одинаковый low, край e называется *тонким ребром*(тонким на рисунке);
- В противном случае край e является *толстым ребром* (жирным на рисунке)

Определение 21. *TT-порядок приоритета* $\prec^*$ является *частичным порядком* на E таким, что для любого $v \in V$ и любого $e, f \in \omega^+(v)$:

- если $low(e) \prec low(f)$, то $e \prec^* f$;
- если $low(e) = low(f)$, *f-толстое ребро* дерева, но e-нет, то $e \prec^* f$.

Определение 22. Пусть v – вершина и ребра $e_1, e_2 \in \omega^+(v)$. *Множество переплетений* $Interlaced(e_1, e_2)$ определено как:
$Interlaced(e_1, e_2) = \{f \in Fringe(e_1) : low(f) \succ low(e2)\}$.

Определение 23. Дан граф G и *дерево Тремо* T из G, тогда раскраска $\lambda : E \backslash T \rightarrow \{-1, 1\}$ является *F-раскраской*, если для каждой вершины v и любых ребер $e_1, e_2 \in \omega^+(v) Interlaced($ и $Interlaced(e_2, e_1)$ монохроматичны и окрашены по-разному.

Сам алгоритм состоит из трех шагов. На первом шаге запускается DFS для вычисления low и статуса ребер (*блокирующие/тонкие/толстые*). Второй шаг-это вычисление

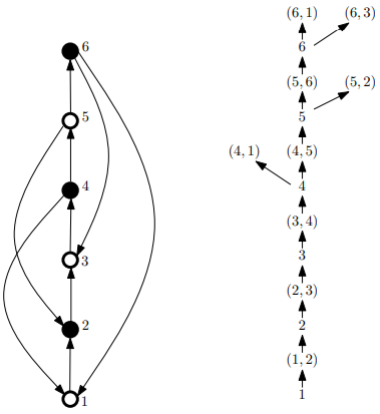

Рис. 15: *Частичный порядок* $\prec$, определенный *деревом Тремо* графа $K_3 3$.

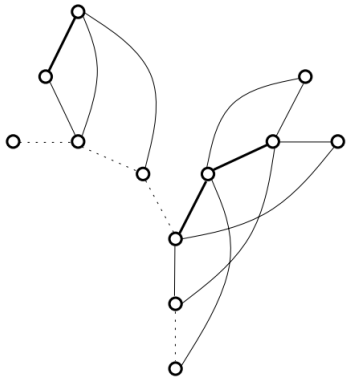

Рис. 16: Иллюстрация классификации ребер дерева.

порядка приоритета TT, который может быть эффективно выполнен с помощью блочной сортировки. Теперь рассмотрим последний шаг алгоритма, который проверяет плоскостность графа.

Мы рассмотрим некоторую структуру данных DS, которая присваивается ребру следующим образом: $DS(e)$ пуст, если e-ребро дерева, и включает e (без ограничений на двухцветность), если e-заднее ребро дерева. Мы говорим, что все задние ребра были обработаны, а ребра дерева все еще не обработаны.

- Пока существует вершина v, отличная от корня, такая, что все ребра в $\omega^+(v)$ были обработаны. Пусть $e = (u, v)$ - ребро дерева, входящее в v. Пусть $e_1 \prec^* e_2 \prec^* \ \prec^* e_k$ - ребра в $\omega^+(v)$ $(k \leqslant 1)$. Мы делаем следующее:
  - Инициализируем $DS(e)$ и $DS(e_1)$.
  - Для $i : 2 \to k$ объединяем $DS(e_i)$ в $DS(e)$ так, что: добавим к $DS(e)$ ребра в D$S(e_i)$ и добавим ограничения *F-раскраски*, соответствующие парам ребер e_j, e_i при $j < i$ (обратите внимание, что все соответствующие задние ребра принадлежат $DS(e)$). Если какое-либо ограничение не может быть выполнено, граф объявляется неплоским.
  - Удаляем из всех $DS(e)$ каждое заднее ребро с меньшей инцидентностью, чем у u.
  - Заявляем, что ребро e было обработано.
- Поскольку все ребра были обработаны, мы объявляем, что граф является планарным.

2. Вычислительные эксперименты

Для проведения вычислительных экспериментов я реализовал два алгоритма:

- Алгоритм добавления пути Хопкрофта и Тарьяна
- Алгоритм добавления ребер де Фрейсекса, де Мендеса и Розенштиля

Итак, эксперемента будет два. В первом мы проверим, действительно ли алгоритмы работают за линейное время. Во втором же эксперименте мы сравним скорость работы алгоритмов.

2.1. Эксперимент 1

Как заявляют создатели алгоритмов, они работают за время $O(n)$. Для проверки этого факта я буду использовать граф определенного типа, который будет масшатабироваться по числу вершин:

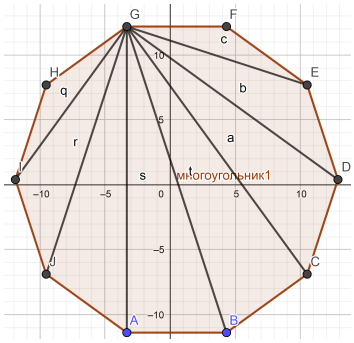

Рис. 17: Граф для Эксперимента 1 с $V = 10$

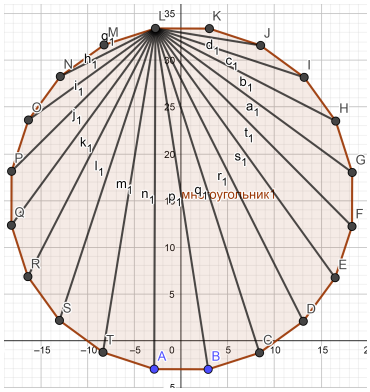

Рис. 18: Граф для Эксперимента 1 с $V = 20$

Далее продемонстированы графики зависимости(рис.19, и рис. 20), где по оси X лежит количество вершин в графе, а по оси Y время в миллисекундах. Первый график построен по результатам работы алгоритма Хопкрофта и Тарьяна, второй по алгоритму де Фрейсекса, де Мендеса и Розенштиля.

Как мы видим, зависимость времени работы от количества вершин графа действительно является линейной.

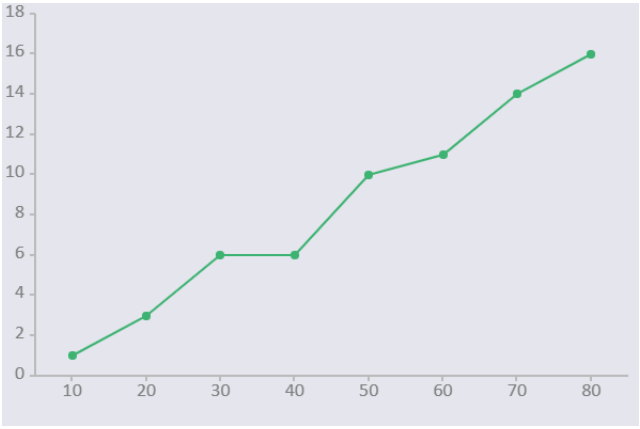

Рис. 19: График для алгоритма Хопкрофта и Тарьяна.

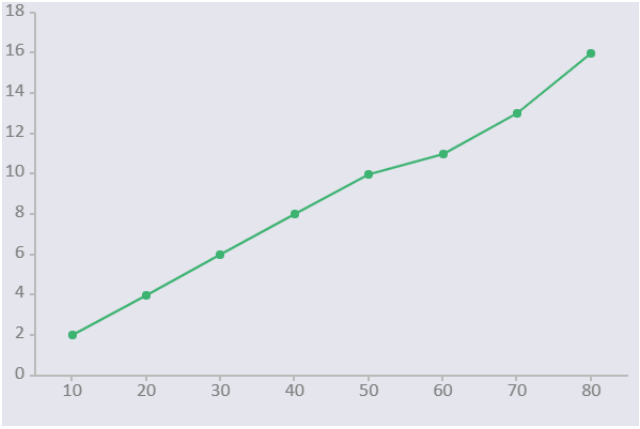

Рис. 20: График для алгоритма де Фрейсекса, де Мендеса и Розенштиля.

2.2. Эксперимент 2

Алгоритм Хопкрофта и Тарьяна, разработанный в 1974 году является первым, а значит самым старым алгоритмом, отыскивающим подграф Куратовского за линейное время. Второй же, в свою очередь, является одним из самых современных. Теоретически, второй алгоритм должен работать быстрее. Сравним время работы этих алгоритмов на различных графах и проверим, так ли это.

Для начала рассмотрим графы, на которых будем замерять скорость работы:

1. **Граф треугольной призмы**(Рис. 21). Первый представленный граф представляет собой призматический граф, заданный произведением декартова графа $C_3 \times P_{\lfloor \frac{n}{3} \rfloor}$, расширенный дополнительными ребрами, соединяющими последовательные концентрические треугольники. При взгляде в перспективе с центральной точкой схода граф выглядит как треугольная призма.

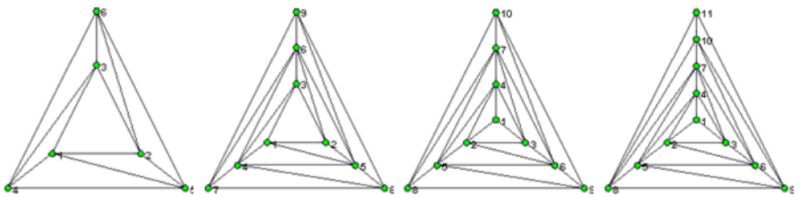

Рис. 21: Примеры графа треугольной призмы.

Тип графа	Время, мс	Время, мс	Время, мс
Граф тр. призмы	2.25	12.8	23.6
Граф плоского колеса	2.05	12.76	23.5
Уп. граф трисекции граней	2.06	13.05	24

Таблица 1: Результаты проверки скорости работы алгоритма H&T

2. **Граф плоского колеса** (Рис. 22). Граф можно описать: одной центральной вершиной (присутствует во всех подграфах колеса); три дороги, расходящиеся от центральной вершины (каждый путь разделяется на два суб-графики колеса); три вершины концентратора (уникальный для каждого подграфа колесо) с соединяющиеся ребрами (спицы колеса) в центральной вершине и в каждой вершине двух соседних радиальных путей, и три ребра, уникальная для каждого подграфа колеса, и завершают цикл два радиальных пути и центральная вершина.

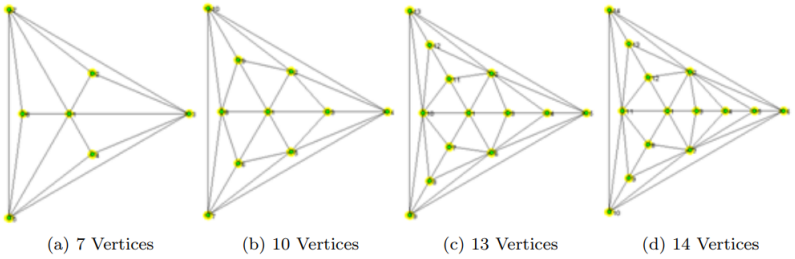

Рис. 22: Примеры графа плоского колеса.

3. **Упорядоченный граф трисекции граней**(Рис. 23). Этот граф начинается с треугольного цикла, образованного тремя вершинами и тремя ребрами, создающими внутреннюю и внешнюю грани; эта внутренняя грань помещается в очередь граней. Когда вершина добавляется к графу: грань, находящаяся в начале очереди, удаляется; новая вершина помещается в центр этой грани; эта грань трисекается тремя ребрами, соединяющими вершины на границе грани с новой вершиной; и три новые (треугольные) грани, образованные этой трисекцией, добавляются в хвост очереди граней.

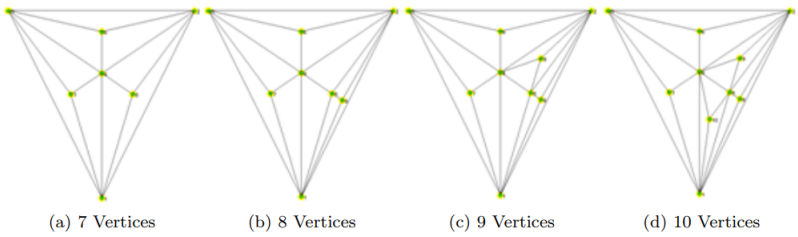

Рис. 23: Примеры упорядоченного графа трисекции граней.

В таблицах представлены результаты измерений времени работы алгоритмов. В первой колонке времени $V = 10$, во второй $V = 50$, в третьей $V = 100$ для более наглядной разницы во времени. Первая таблица для алгоритма Хопкрофта и Тарьяна, вторая для алгоритма де Фрейсекса, де Мендеса и Розенштиля. Итак, обратим внимание на результат.

Тип графа	Время, мс	Время, мс	Время, мс
Граф тр. призмы	2.25	11.3	21.4
Граф плоского колеса	2.05	11.7	20.9
Уп. граф трисекции граней	2.06	11.05	22

Таблица 2: Результаты проверки скорости работы алгоритма добавления ребер.

По результатам измерений можно сделать вывод, что второй алгоритм действительно производит проверку планарности за меньшее время, однако также стоит заметить, что разница достаточно невелика.

Заключение

В курсовой работе изучались, сравнивались и реализовывались алгоритмы проверки планарности графов. Были изучены метод добавления путей Хопкрофта и Тарьяна, метод добавления вершин Ши и Сю и метод добавления ребер де Фрейсекса, де Мендеса и Розенштиля. В ходе проведения вычислительных экспериментов было установлено, что алгоритмы действительно работают за линейное время, и более современный метод добавления ребер имеет более высокую скорость работы, чем первый линейный алгоритм проверки планарности в истории.

Литература

1. Аносов Д.В. *Отображения окружности, векторные поля и их применения.* // М.:МЦНМО, 2003.

2. Куратовский К. *Топология.* // М.: Мир, 1969, Т. 1,2.

3. John Hopcroft, Robert E. Tarjan. *Efficient planarity testing* // Journal of the Association for Computing Machinery. — 1974. — Т. 21, вып. 4. — С. 549–568.

4. Martyn G. Taylor. *Planarity Testing by Path Addition.* — University of Kent, 2012. — (Ph.D.).

5. A. Lempel, S. Even, I. Cederbaum. *An algorithm for planarity testing of graphs* // P. Rosenstiehl (Ed.), Theory of Graphs, Gordon and Breach, New York, pp. 215-232.

6. Shih W. K., Hsu W. L. *A new planarity test* // Theoretical Computer Science. — 1999. — Т. 223, вып. 1–2. — С. 179–191.

7. John M. Boyer, Wendy J. Myrvold. *On the cutting edge: simplified O(n) planarity by edge addition* // Journal of Graph Algorithms and Applications. — 2004. — Т. 8, вып. 3. — С. 241–273.

8. S. Even. *Graph Algorithms.* // Computer Science Press, Rockville, MD, 1979.

9. H. de Fraysseix and P. Ossona de Mendez. *A characterization of DFS cotree critical graphs.* // In Graph Drawing, volume 2265 of Lecture notes in Computer Science, pages 84–95, 2002.

10. H. de Fraysseix and P. Ossona de Mendez. *On cotree-critical and DFS cotreecritical graphs.* // Journal of Graph Algorithms and Applications, 7(4):411–427, 2003.

11. de Fraysseix H., Ossona de Mendez P., Rosenstiehl P. *Trémaux Trees and Planarity* // International Journal of Foundations of Computer Science. — 2006. — Т. 17, вып. 5. — С. 1017–1030.

A. Исходный код программы

A.1. Исходный код алгоритма H&T

```
from collections import deque  # module for working with deque
from typing import List, Dict  # module for working with annotations
from datetime import datetime  # module for working with time
start_time = datetime.now()  # marking the time of beginning


class Vertex:
def __init__(self, name):
self.name = name  # Unique Name of vertex in graph.
self.index = 0  # Index of a vertex assigned during the DFS.
self.neighbours = list()  # List of neighbours on graph.
self.parent = None  # Ancestor of vertex in graph.
self.lowPoint1 = self  # Variables assigned during the DFS
self.lowPoint2 = self  # for bucket sorting.
self.firstPath = None  # denotes the number of the first
# path containing v

def add_neighbour(self, v) -> None:
if v not in self.neighbours:
self.neighbours.append(v)


class Edge:
def __init__(self, v, u):
self.index = 0  # Index of an edge assigned during the DFS.
self.type = None  # "TREE" or "BACK", None if not visited in DFS
self.pair = (v, u)  # A pair of vertices defining an edge

def sort_f(self) -> None:
"""
Function for bucket sort.
"""
v = self.pair[0]
w = self.pair[1]
if self.type == "BACK":
return 2 * w.index
elif self.type == "TREE" and w.lowPoint2 >= v.index:
return 2 * w.lowPoint1
elif self.type == "TREE" and w.lowPoint2 < v.index:
return 2 * w.lowPoint1 + 1


class Path:
def __init__(self, edge):
# list of vertices in path
self.path_vertices = [edge.pair[0], edge.pair[1]]
```

```
# list of edges in path
self.path_edges = [edge]
self.index = 0

def path_append(self, edge) -> None:
"""
This method adds an edge to the current path.
"""
self.path_vertices.append(edge.pair[0])
self.path_vertices.append(edge.pair[1])
self.path_edges.append(edge)


class Graph:
vertices = {}  # Dict{key = vertex : value = set of adjacent vertexes}
edges = {}  # Dict{key = vertex : value = list of adjacent edges}
bucket = []
adj_list = {}
paths = deque()
global DFCount  # Counter for DFS
DFCount = 0
global cur_start
cur_start = None
start_vertex: Vertex  # The vertex from which DFS was launched
global counter
counter = 0

def add_vertex(self, v) -> bool:
if isinstance(v, Vertex) and v.name not in self.vertices:
self.vertices[v] = set()
return True
else:
return False

def add_edge(self, u, v) -> bool:
if u in self.vertices and v in self.vertices and u != v:
for key, value in self.vertices.items():
if key == u:
u.add_neighbour(v)
self.vertices[u].add(v)
if self.edges.get(u) is None:
self.edges[u] = set()
self.edges[u].add(Edge(u, v))
if key == v:
v.add_neighbour(u)
self.vertices[v].add(u)
if self.edges.get(v) is None:
self.edges[v] = set()
self.edges[v].add(Edge(v, u))
```

```
return True
else:
return False

def print_adjacency_list_ns(self):
"""
This method prints adjacency lists
of vertices in graph (self.vertices).
"""
for key in list(self.vertices.keys()):
print(f"{key.name}: [", end=' ')
for neigh in self.vertices[key]:
print(neigh.name, end=' ')
print(']')

def print_adjacency_list_s(self):
"""
This method prints adjacency lists
of edges (key: Vertex) in graph (self.edges).
"""
for key in list(self.adj_list.keys()):
print(key.name + ': [', end=' ')
for neigh in self.adj_list[key]:
print(neigh.name, end=' ')
print(']')

def get_number_of_edges(self) -> int:
i = 0
for arr in self.edges.values():
i += len(arr)
return i

def dfs(self, vertex) -> None:
self.start_vertex = vertex
self._dfs(vertex)

def _dfs(self, vertex) -> None:
global DFCount
DFCount += 1
vertex.index = DFCount
vertex.lowPoint1 = vertex.lowPoint2 = vertex.index
for v in vertex.neighbours:
if v.index == 0:
v.parent = vertex
self.vertices[v].remove(vertex)  # make edge is oriented
for e in self.edges[vertex]:  # mark edge as a tree arc
if e.pair == (vertex, v):
e.type = "TREE"
e.index += DFCount
for e_b in self.edges[v]:
```

```
if e_b.pair == (v, vertex):
self.edges[v].remove(e_b)
break
self._dfs(v)

# Backtracking. Here we calculate lowpoints of vertex
if v.lowPoint1 < vertex.lowPoint1:  # dummy statement b
vertex.lowPoint2 = min(vertex.lowPoint1, v.lowPoint2)
vertex.lowPoint1 = v.lowPoint1
elif v.lowPoint1 == vertex.lowPoint1:
vertex.lowPoint2 = min(vertex.lowPoint2, v.lowPoint2)
else:
vertex.lowPoint2 = min(vertex.lowPoint2, v.lowPoint1)

elif v.index < vertex.index and \
v != vertex.parent:  # to avoid exploring an edge in both directions
self.vertices[v].remove(vertex)  # make edge oriented
for e in self.edges[vertex]:  # mark edge as a back edge and make oriented
if e.pair == (vertex, v):
e.type = "BACK"
e.index += DFCount
for e_b in self.edges[v]:
if e_b.pair == (v, vertex):
self.edges[v].remove(e_b)
break
# Calculating lowpoints
if v.index < vertex.lowPoint1:
vertex.lowPoint2 = vertex.lowPoint1
vertex.lowPoint1 = v.index
elif v.index > vertex.lowPoint1:
vertex.lowPoint2 = min(vertex.lowPoint2, v.index)

def bucket_sort(self):
"""
This function sorts the vertices in
the dictionary(self.adj_list) in a specific
way for further method self.embed().
"""
for i in range(2 * len(self.vertices.keys()) + 1):
self.bucket.append(list())
for e_set in self.edges.values():
for e in e_set:
self.bucket[e.sort_f()].append(e)
for v in self.vertices.keys():
self.adj_list[v] = list()
for i in range(2 * len(self.vertices.keys()) + 1):
for e in self.bucket[i]:
self.adj_list[e.pair[0]].append(e.pair[1])

def embed(self) -> bool:
```

```
"""
This is the main function for checking whether all graph
segments can be embedded on a plane. Before calling it,
you need to call self.DFS() and sort adjacency list by
self.bucket_sort().
"""

def pathfinder(vertex: Vertex):
nonlocal Stack, Next, \
Block, path_counter, path_start, free
for w in self.adj_list[vertex]:
for edge in self.edges[vertex]:
if edge.pair == (vertex, w):
if edge.type == "TREE":
if path_start == 0:
path_start = vertex
path_counter += 1
self.paths.append(Path(edge))  # +
w.firstPath = path_counter
self.paths[-1].path_append(edge)  # +
print(vertex.name, w.name, "tree")  # +
pathfinder(w)
# delete stack entries and blocks corresponding
# to vertices no smaller than v;
for block in Block:
x = block[0]
y = block[1]
if ((x in Stack and x >= vertex.index) or x == 0) \
and ((y in Stack and y >= vertex.index) or y == 0):
Block.remove(block)
for block in Block:
x = block[0]
y = block[1]
if x in Stack and x >= vertex.index:
block[0] = 0
block[1] = y
if y in Stack and y >= vertex.index:
block[0] = x
block[1] = 0
while Next[0] != 0 and Stack[Next[0]] >= vertex.index:
Next[0] = Next[Next[0]]
while Next[1] != 0 and Stack[Next[1]] >= vertex.index:
Next[1] = Next[Next[1]]
if w.firstPath != vertex.firstPath:
#  all of segment with first edge (v, w) has
# been embedded. New blocks must be
#  moved from right to left
left_ = 0
for block in Block:
x = block[0]
```

```
y = block[1]
if (x in Stack and/
x > self.paths[w.firstPath].path_vertices[-1].index) or (
y in Stack and y > self.paths[w.firstPath].path_vertices[-1].index) \
and Stack[Next[0]] != 0:
if x in Stack and/
 x > self.paths[w.firstPath].path_vertices[-1].index:
if y in Stack and/
 y > self.paths[w.firstPath].path_vertices[-1].index:
return False  # nonplanar
left_ = x
else:  # y in Stack and y.index > w.firstPath.path_vertices[-1]
save = Next[left_ + 1]
Next[left_ + 1] = Next[y + 1]
Next[y + 1] = save
left_ = y
Block.remove(block)
# block on B must be combined with new blocks just deleted;
if Block:
block = Block[-1]
x = block[0]
y = block[1]
Block.remove(block)
if x != 0:
Block.append(block)
elif left_ != 0 or y != 0:
Block.append([left_, y])
# delete end-of-stack marker on right stack;
Next[0] = Next[Next[0]]
# v -- -> w. Current path is complete. Path is normal if f(PATH(s)) < w;
elif edge.type == "BACK":
if path_start == 0 or path_start.index == 0:
path_counter += 1
path_start = vertex
self.paths.append(Path(edge))
self.paths[path_counter - 1].path_append(edge)
print(vertex.name, w.name, "back")  # +
# switch blocks of entries from left
# to right so that p may be embedded on left;
left_ = 0
right_ = -1
while (Next[left_ + 1] != 0 and Stack[Next[left_ + 1]] > w.index) or \
(Next[right_ + 1] != 0 and Stack[Next[right_ + 1]] > w.index):
for block in Block:
x = block[0]
y = block[1]
if x != 0 and y != 0:
if Stack[Next[left_ + 1]] > w.index:
if Stack[Next[left_ + 1]] > w.index:
return False  # nonplanar
```

```
save = Next[right_ + 1]
Next[right_ + 1] = Next[left_ + 1]
Next[left_ + 1] = save
save = Next[x + 1]
Next[x + 1] = Next[y + 1]
Next[y + 1] = save
left_ = y
right_ = x
else:  # STACK(NEXT(R')) > w;
left_ = x
right_ = y
elif x != 0:  # STACK (NEXT(L')) > w;
save = Next[x + 1]
Next[x + 1] = Next[right_ + 1]
Next[right_ + 1] = Next[left_ + 1]
Next[left_ + 1] = save
right_ = x
elif y != 0:
right_ = y
Block.remove(block)
# add P to left stack if p is normal;
if self.paths[path_start.firstPath - 1].path_vertices[-1].index < w.index:
if left_ == 0:
left_ = free

Stack[free] = self.paths[path_start.firstPath - 1].path_vertices[
-1].index  # ?
Next[free + 1] = Next[1]
Next[1] = free
free += 1
# Add new block corresponding to combined old blocks.
# New block may be empty.
# If segment containing current path is not a single frond;
if right_ == -1:
right_ = 0
if left_ != 0 or right_ != 0 or vertex.index != path_start.index:
Block.append([left_, right_])
# if segment containing current path
# is not a single frond, add an end-of-stack
# marker to right stack;
if vertex.index != path_start.index:
Stack[free] = 0
Next[free + 1] = Next[0]
Next[0] = free
free += 1
path_start.index = 0

# Initialization
Stack = [0 for i in range(self.get_number_of_edges())]
print(len(Stack))
```

```
Next = [0 for i in range(self.get_number_of_edges() + 1)]
Block = []
Next.append(0)  # Next[0] = Next[1] = 0
Next.append(0)
free = 1
Stack.append(0)
self.start_vertex.firstPath = 1
path_start = 0
path_counter = 0
pathfinder(self.start_vertex)
return True


def main() -> None:
# Initializing an example graph
vertices = []
for i in range(100):
vertices.append(Vertex(str(i)))
graph = Graph()
for i in vertices:
graph.add_vertex(i)
for i in range(0, len(vertices) - 1):
for j in range(1, len(vertices)):
if j - i == 1:
graph.add_edge(vertices[i], vertices[j])
print("Added edge between", vertices[i].name,
"and", vertices[j].name)
for j in range(1, len(vertices)):
if 2 <= j <= len(vertices) - 1:
graph.add_edge(vertices[0], vertices[j])
print("Added edge between", vertices[0].name,
"and", vertices[j].name)

# Logs to the console
print("Adjacency list of original graph:")
graph.print_adjacency_list_ns()
print(f"This graph has {graph.get_number_of_edges()} edges, so each edge
 has both directions.")
graph.dfs(vertices[0])
print("Adjacency list of modified graph after first DFS:")
graph.print_adjacency_list_ns()
print(f"This oriented graph now has {graph.get_number_of_edges()} edges.")
print("These are LowPoint1 and LowPoint2 of each vertex:")
for v in graph.vertices.keys():
print(f"{v.name}: [{v.lowPoint1}, {v.lowPoint2}]")
print("Before pathfinding we need bucket sort. Now adjacency list is:")
graph.bucket_sort()
graph.print_adjacency_list_s()
graph.vertices.clear()  # was replaced by adj_list
print(graph.embed())
```

```
if __name__ == '__main__':
main()
end_time = datetime.now()  # marking the time of the end
print('Duration: {0:.3f} milliseconds'.format(
end_time.microsecond / 1000 - start_time.microsecond / 1000))
```

A.2. Исходный код метода добавления ребер

```
import collections
from typing import List, Dict, Set  # module for working with annotations
from datetime import datetime  # module for working with time
start_time = datetime.now()  # marking the time of beginning


class Vertex:
def __init__(self, name):
self.name = name  # Unique Name of vertex in graph.
self.index = 0  # Index of a vertex assigned during the DFS.
self.neighbours = list()  # List of neighbours on graph.
self.parent = None  # Ancestor of vertex in graph.

def add_neighbour(self, v) -> None:
if v not in self.neighbours:
self.neighbours.append(v)


class Edge:
def __init__(self, v, u):
self.index = 0  # Index of an edge assigned during the DFS.
self.type = None  # "TREE" or "BACK", None if not visited in DFS
self.pair = (v, u)  # A pair of vertices defining an edge
self.low = None  # low(e)
self.fringe = set()  # Fringe(e) : set(Edge)
self.status = ""  # Edge status: {"BLOCK", "THIN", "THICK"}
self.color = 0  # colors = {-1, 1}, 0 is no color
self.isProcessed = False  # is edge processed is Graph.embed()
self.DS = []  # DS(e) : List[Edge]


def interlaced(edge1: Edge, edge2: Edge) -> List[Edge]
res = list()
for f in edge1.fringe:
if f.low.index > edge2.low.index:
res.append(f)
return res
```

```
class Graph:
vertices = {}  # Dict{key = vertex : value = set of adjacent vertexes}
edges = {}  # Dict{key = vertex : value = list of adjacent edges}
DFCount = 0  # Counter for DFS
start_vertex: Vertex  # The vertex from which DFS starts
previous_edge = None  # The variable required to compute Fringe(e)
ttt = 0  # int counter for indexing edges

def add_vertex(self, v) -> bool:
if isinstance(v, Vertex) and v.name not in self.vertices:
self.vertices[v] = set()
return True
else:
return False

def add_edge(self, u, v) -> bool:
if u in self.vertices and v in self.vertices and u != v:  #
for key, value in self.vertices.items():
if key == u:
u.add_neighbour(v)
self.vertices[u].add(v)
if self.edges.get(u) is None:
self.edges[u] = list()
self.edges[u].append(Edge(u, v))
# self.edges[u].add(Edge(v, u))
if key == v:
v.add_neighbour(u)
self.vertices[v].add(u)
if self.edges.get(v) is None:
self.edges[v] = list()
self.edges[v].append(Edge(v, u))
# self.edges[v].add(Edge(u, v))

return True
else:
return False

def print_adjacency_list_vertex(self):
"""
This method prints adjacency lists
of vertices in graph (self.vertices).
"""
for key in list(self.vertices.keys()):
print(f"{key.name}: [", end=' ')
for neigh in self.vertices[key]:
print(neigh.name, end=' ')
print(']')
```

```
def print_adjacency_list_edges(self):
"""
This method prints adjacency lists
of edges (key: Vertex) in graph (self.edges).
"""
for vertex, edges in self.edges.items():
print(vertex.name + ': [', end=' ')
for edge in edges:
print(edge.index, end=' ')
print(']')

def get_number_of_edges(self) -> int:
i = 0
for arr in self.edges.values():
i += len(arr)
return i

def dfs(self, vertex: Vertex) -> None:
self.start_vertex = vertex
self._dfs(vertex)

def _dfs(self, vertex: Vertex) -> None:
self.DFCount += 1
vertex.index = self.DFCount
temp_counter = -1
for v in vertex.neighbours:
if v.index == 0:
self.ttt += 1
print(vertex.name, v.name, "tree")
v.parent = vertex
self.vertices[v].remove(vertex)  # make edge is oriented
for e in self.edges[vertex]:  # mark edge as a tree arc
if e.pair == (vertex, v):
e.type = "TREE"
e.index = self.ttt  # vertex.index  # e.index = self.DFCount
for e_b in self.edges[v]:
if e_b.pair == (v, vertex):
self.edges[v].remove(e_b)
break
self._dfs(v)

# calculating low(e)
for edge in self.edges[vertex]:
if edge.pair == (vertex, v):
for e in self.edges[v]:
if edge.low is None or e.low.index <= edge.low.index:
edge.low = e.low
# to avoid exploring an edge in both directions
elif v.index < vertex.index and v != vertex.parent:
self.ttt += 1
```

```
temp_counter += 1
print(vertex.name, v.name, "back")
self.vertices[v].remove(vertex)  # make edge is oriented
for e in self.edges[vertex]:  # mark edge as a back edge and make oriented
if e.pair == (vertex, v):
e.type = "BACK"
e.index = self.ttt  # vertex.index + temp_counter
e.low = v  # calculating low(e)
for e_b in self.edges[v]:
if e_b.pair == (v, vertex):
self.edges[v].remove(e_b)
break

# calculating Fringe(e)
for edge in self.edges[vertex]:
if edge.pair == (vertex, v):
# copy suitable edges from previous recursion step
if self.previous_edge is not None:
for f in self.previous_edge.fringe:
if f.low.index < vertex.index:
edge.fringe.add(f)
edges = self.edges[v] + [edge]  # the edge itself can also be
for e in edges:
if e.index > edge.index and e.type == "BACK" and
 e.low.index < vertex.index:
edge.fringe.add(e)
self.previous_edge = edge

def compute_edge_status(self):
"""
This method assign status of each edge:
{"BLOCK", "THIN", "THICK"}.
"""
def is_low_equal(fringe: List[Edge]) -> bool:
"""
This is an auxiliary function for checking
the equality of low (e) of each vertex from Fringe(e).
"""
isFirst = True
comparable = None
for e in fringe:
if isFirst:
comparable = e.low
isFirst = False
if e.low != comparable:
return False
return True

for edges in self.edges.values():
for edge in edges:
```

```
if edge.fringe == {}:
edge.status = "BLOCK"
elif is_low_equal(edge.fringe):
edge.status = "THIN"
else:
edge.status = "THICK"

def bucket_sort(self) -> None:  # The second step of
# algorithm - sorting
for edges in self.edges.values():
edges.sort(key=lambda edge: edge.low.index)

def coloring_constraint(self, edge1: Edge, edge2: Edge) -> bool:
"""
This method checks F-coloring constraints.
"""
interlaced_e1_e2 = interlaced(edge1, edge2)
interlaced_e2_e1 = interlaced(edge2, edge1)
colored = []
for e_interlaced in interlaced_e1_e2:
edge = None
for edges_list in self.edges.values():
for ed in edges_list:
if ed == e_interlaced:
edge = ed
break
if edge is not None:
if edge.color == 0:
edge.color = -1
elif edge.color == -1:
edge.color = 1
elif edge.color == 1:
edge.color = -1
colored.append(edge)
for i in range(len(colored) - 1):
if colored[i].color != colored[i+1].color:
return False
colored.clear()

for e_interlaced in interlaced_e2_e1:
edge = None
for edges_list in self.edges.values():
for ed in edges_list:
if ed == e_interlaced:
edge = ed
break
if edge is not None:
if edge.color == 0:
edge.color = -1
elif edge.color == -1:
```

```
edge.color = 1
elif edge.color == 1:
edge.color = -1
colored.append(edge)
for i in range(len(colored) - 1):
if colored[i].color != colored[i+1].color:
return False
colored.clear()

return True

def embed(self) -> bool:
"""
This method is the third and main step of the algorithm.
It checks the planarity of the graph.
If function returns True - graph is planar.
If method returns False - non planar.
"""
def isAllProcessed() -> bool:
"""
This function will be used to check the
processing of all edges coming from the vertices, except for the root
"""
for v in self.edges.keys():
if self.edges[v] != self.edges[self.start_vertex]:
for edge in self.edges[v]:
if not edge.isProcessed:
return False
return True

for v in self.edges.keys():
for edge in self.edges[v]:  # We say that all the cotree edges have
if edge.type == "BACK":  # been processed and that the tree edges
# are still unprocessed.
edge.DS.append(edge)  # DS(e) is empty if e is a tree edge
# and includes e (with no bicoloration
edge.isProcessed = True   # constraints) if e is a cotree edge

while not isAllProcessed():
for v in self.edges.keys():
if self.edges[v] != self.edges[self.start_vertex]:
parent = v.parent
for ed in self.edges[parent]:
if ed.pair == (parent, v):
e = ed
temp_counter = 0
prev_edge: Edge
for e_i in self.edges[v]:
for e_ds in e_i.DS:
temp_counter += 1
```

```
prev_edge = e_ds
if temp_counter == 1:
e.DS.append(e_ds)
elif not self.coloring_constraint(e_ds, prev_edge):
return False  # nonplanar
temp_counter = 0
for edge in e.DS:
if edge.type == "BACK" and edge.pair[0] == e.pair[0]:
e.DS.remove(edge)
e.isProcessed = True
return True


def main() -> None:
# Initializing an example graph
vertices = []
for i in range(80):
vertices.append(Vertex(str(i)))
graph = Graph()
for i in vertices:
graph.add_vertex(i)
for i in range(0, len(vertices) - 1):
for j in range(1, len(vertices)):
if j - i == 1:
graph.add_edge(vertices[i], vertices[j])
print("Added edge between", vertices[i].name,
"and", vertices[j].name)
for j in range(1, len(vertices)):
if 2 <= j <= len(vertices) - 1:
graph.add_edge(vertices[0], vertices[j])
print("Added edge between", vertices[0].name,
 "and", vertices[j].name)

# Logs to the console
print("Adjacency list of original graph:")
graph.print_adjacency_list_vertex()
print(f"This graph has {graph.get_number_of_edges()} edges,
 so each edge has both directions.")
graph.edges = collections.OrderedDict(
sorted(graph.edges.items(), key=lambda pair: pair[0].name))
graph.print_adjacency_list_edges()
graph.dfs(vertices[0])
print("Adjacency list of modified graph after first DFS:")
graph.print_adjacency_list_vertex()
print(f"This oriented graph now has
{graph.get_number_of_edges()} edges.")
graph.compute_edge_status()
graph.print_adjacency_list_edges()
graph.bucket_sort()
print("Before embedding we need bucket sort. Now adjacency list is:")
```

```
graph.print_adjacency_list_edges()
print(graph.embed())


if __name__ == '__main__':
main()
end_time = datetime.now()  # marking the time of the end
print('Duration: {0:.3f} milliseconds'.format(
end_time.microsecond / 1000 - start_time.microsecond / 1000))
```