Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Автоматический анализ визуального представления графов размеченных» систем переходов»

Студент _____ _____ _____
(Группа) (Подпись, дата) (И.О. Фамилия)

Руководитель _____ _____
(Подпись, дата) (И.О. Фамилия)

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **4**

1 Обзор предметной области **7**
1.1 Основная терминология 7
1.2 Классы непланарных графов 8
1.3 Кривая Безье 9
1.3.1 Пересечение кривых Безье 9
1.3.2 Алгоритм де Кастельжо 10

2 Конструкторский раздел **12**
2.1 Модуль геометрических вычислений 12
2.1.1 Точка 12
2.1.2 Прямоугольник 13
2.1.3 Кривая Безье 13
2.2 Модуль анализа диаграмм 14
2.2.1 Вершина 15
2.2.2 Вектор 15
2.2.3 Ребро 15
2.2.4 Топологический граф 16
2.3 Модуль интеграции 19
2.3.1 Graphviz 19

3 Технологический раздел **21**
3.1 Руководство администратора 21
3.2 API 22
3.2.1 Структура `Point` 22
3.2.2 Класс `Rectangle` 24
3.2.3 Класс `Curve` 24
3.2.4 Стуктура `Vertex` 24
3.2.5 Структура `Vector` 25
3.2.6 Класс `Edge` 26
3.2.7 Класс `Graph` 27

4 Тестирование **29**
4.1 Анализ диаграммы графа K_{10} 29
4.2 Анализ геометрического графа 32

5 Заключение **33**

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ **34**

ПРИЛОЖЕНИЕ **35**

ВВЕДЕНИЕ

Теория визуального представления непланарных графов формируется с относительно недавних пор. С технологическим прорывом последних десятилетий возникает потребность в визуализации всё бóльших объёмов взаимосвязанных данных, что порождает, как правило, огромные графы. Классические методы визуализации здесь мало применимы: они, с целью наибольшего приближения класса диаграммы к планарному, считающимся наиболее простым для восприятия, используют эвристики минимизации пересечения рёбер, что в случае графов с большим числом пересечений может плохо сказаться на читаемости. Теория визуального представления непланарных графов помимо числа пересечения рёбер оперирует топологическими и геометрическими свойствами этих пересечений. В соответствии с возможностью представить граф диаграммой, не содержащей определённые сочетания пересечений рёбер, его относят к одному или нескольким классам непланарных графов.

Рассмотрим две диаграммы двудольного графа. Первая (рисунок 1) содержит минимально возможное число пересечений — 24, а вторая (рисунок 2) – 34 пересечения, но обладает топологическим свойством skewness-4: это означает, что удаление 4 рёбер графа позволит сделать его планарным. Такой пример приводился на научном семинаре Дагштуль, посвящённом непланарным графам, и большинство специалистов признали второй вариант более читаемым.

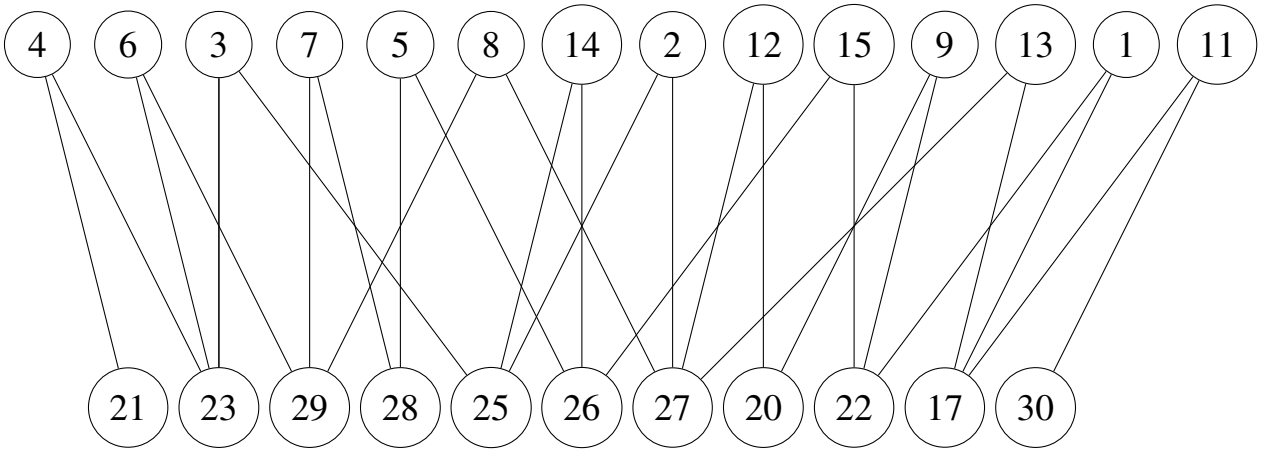

Рисунок 1 — Граф с 24 пересечениями рёбер (минимально возможным)

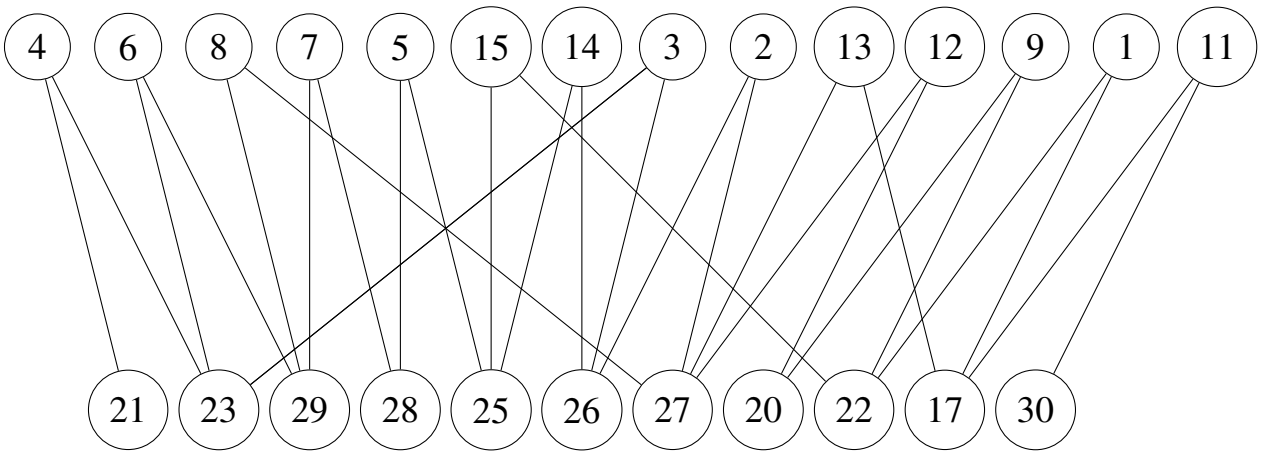

Рисунок 2 — Граф с 34 пересечениями рёбер (минимальное skewness)

Одно из актуальных исследовательских направлений — распознавание принадлежности непланарного графа заданному классу, что в общем случае является вычислительно трудной задачей. Можно рассмотреть распознавание принадлежности *конкретной* диаграммы заданному классу. Эта проблема мало изучена, и в современной литературе нет систематизированных алгоритмов её решения. Однако автоматическое выявление компонентов диаграммы, не удовлетворяющих заданному классу и, следовательно, снижающих читаемость в смысле определения класса, полезно в вопросах дальнейшей коррекции диаграммы и качественном анализе результата работы алгоритмов визуализации непланарных графов.

Целью настоящей работы является реализация библиотеки функций выявления компонентов диаграммы, снижающих её читаемость. Поставлена более общая по сравнению с вопросом распознавания задача: необходимо не просто определить принадлежность классу, но и предъявить *все* компоненты диаграммы, не удовлетворяющие ему. Таким образом, работа в большей своей части является исследовательской, и первостепенным ставится вопрос построения алгоритмов выявления недопустимых (применительно к данному классу) компонентов диаграммы. Алгоритмы компьютерной графики используются здесь в качестве прикладного инструмента для достижения поставленной цели.

К результату работы предъявляются и другие требования:

— библиотекой принимаются диаграммы, рёбра которых представляются произвольными жордановыми дугами;

— функции библиотеки соответствуют большинству изученных на данный момент классов непланарных графов;

— библиотека поддерживает непосредственный ввод результата работы программного обеспечения Graphviz, использующегося для визуализации графа по его описанию на языке DOT;

1 Обзор предметной области

1.1 Основная терминология

Теоретическая основа текущего и следующего подразделов выстроена на исследовании “A Survey on Graph Drawing Beyond Planarity“ [1].

Необходимо определить основную терминологию, использующуюся при визуализации графов. Пусть $G = (V, E)$ – граф. *Диаграммой* или *топологическим графом* Γ называют инъективное отображение, ставящее в соответствие каждой вершине графа $v \in V$ точку плоскости p_v, и всякому ребру $(u, v) \in E$ — жорданову дугу с точками p_u и p_v на концах. Традиционно в терминологии и обозначениях не различают вершины, рёбра графа и точки, дуги на плоскости соответственно их представляющие, поэтому указанные понятия мы будем использовать взаимозаменяемо. Два ребра Γ пересекаются, если у них есть общая точка, не совпадающая с точками на концах рёбер; указанная точка — *точка пересечения*, или просто *пересечение* рёбер. Будем предполагать, что никакое ребро не содержит вершину, отличную от его конечных точек, никакие два ребра не пересекаются по касательной и никакие три ребра не делят персечение.

Диаграмма Γ графа G делит плоскость на топологически связанные *области*. Неограниченную область называют *внешней*, остальные — *внутренними*. Границе области могут принадлежать вершины графа или пересечения. *Вложением* G называют класс эквивалентности диаграммы при гомеоморфизме плоскости, то есть класс диаграмм G, определяющих одно и то же множество внешних и внутренних областей. Граф с фиксированным вложением — *вложенный граф*. Диаграмма без пересечений рёбер называется *планарной*. *Планарный граф* допускает представление планарной диаграммой и называется *плоскостным графом* при фиксированном планарном вложении.

Рассмотрим случай, когда вершины диаграммы Γ связываются ломаными, то есть рёбра представляются отрезками прямых. *Изломом* назовём точку, соединяющую соседние отрезки ломаной. Если всякое ребро Γ содержит не более k изломов, то диаграмма типа *излома* k. Особенно интересен случай $k = 0$, когда всякое ребро представляется отрезком прямой. Такую диаграмму называют *геометрическим графом*.

1.2 Классы непланарных графов

Классы непланарных графов описывают множества графов, допускающих представление диаграммой, не содержащей не удовлетворяющих определённым топологическим или геометрическим свойствам сочетаний рёбер. Для каждого класса X далее даётся определение диаграммы типа X. Граф принадлежит классу X, если допускает представление диаграммой типа X. Названия многих классов не имеют канонического русскоязычного перевода, поэтому приводятся в оригинале на английском языке.

k-planar или *k-планарные* диаграммы $(k \geq 1)$ не содержат рёбра, пересекаемые более k раз. Первым был изучен случай $k = 1$; случаи $k > 1$ изучались позднее с целью определения минимально возможного числа пересечений в таких диаграммах.

k-quasi planar диаграммы $(k \geq 3)$ не содержат k взаимно пересекающихся рёбер. Первые исследования k-quasi-planar графов проводились с целью определения максимально возможного числа их рёбер.

В *skewness-k* диаграммах $(k \geq 1)$ удаление самое большее k рёбер позволяет сделать диаграмму планарной, то есть диаграмма не содержит набора пересечений, не покрываемых (максимум) k рёбрами. В основном изучен случай $k = 1$ — такие графы называются *почти планарными* или *близкими к планарным*.

(k, l)-grid-free диаграммы $(k, l \geq 1)$ не содержат двух групп из k и l рёбер соответственно таких, что всякое ребро первой группы пересекает каждое ребро второй группы. Если k рёбер инциденты одной и той же вершине, то (k, l)-сетка называется *радиальной*; если l рёбер инциденты этой же вершине, (k, l)-сетка *бирадиальная*. Наконец, (k, l)-сетка *натуральная*, если все рёбра независимы и никакие два ребра из одной группы не пересекаются.

Диаграмма геометрического графа, любые пересекающиеся рёбра которого формируют угол $\frac{\pi}{2}$ в точке пересечения, называется *геометрическим RAC* (*right angle crossing*) графом. Появление RAC диаграмм связано когнитивными исследованиями, показавшими положительное влияние большой величины углов пересечения рёбер на читаемость диаграммы.

ACEα (*angle crossing equal α*) и *ACLα* (*angle crossing at least α*) — варианты RAC диаграмм с параметризованным углом $\alpha \in (0, \frac{\pi}{2})$. В геометрическом ACE$\alpha$ графе любые два пересекающиеся ребра образуют угол, *равный* α, а в гео-

метрическом ACLα графе значения углов всех пересекающихся рёбер *хотя бы* α.

1.3 Кривая Безье

Для представления рёбер произвольных топологических графов могут использоваться *кривые Безье*. Использование *сплайна Безье* — ряда кривых Безье, где конечная точка всякой кривой совпадает с началом последующей — позволяет с достаточной точностью описать любую жорданову дугу. Описываемые определения и алгоритмы взяты из онлайн-книги “A Primer on Bézier Curves“ [2], посвященной практическому использованию кривых Безье.

Параметрически кривая Безье $B(t)$ представляется в виде

$$B(t) = \sum_{i=0}^{n} b_{i,n}(t) P_i, \quad t \in [0, 1],$$

где P_i, $i = 0, ..., n$, есть *контрольные точки* кривой Безье, а многочлен $b_{i,n}(t)$ — *полином Бернштейна* или *базисная функция* кривой Безье, соответствующая контрольной точке P_i:

$$b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, ..., n.$$

Многоугольник, образованный последовательным соединением контрольных точек P_i, $i = 0, ...,n$, отрезками прямых называется *многоугольником Безье*. Всякая выпуклая оболочка многоугольника Безье содержит исходную кривую.

1.3.1 Пересечение кривых Безье

Перейдём к вопросу о пересечении кривых Безье. Рассмотрим следующий алгоритм пересечения, основывающийся на подходе “разделяй и властвуй“.

1. Выбирается пара кривых Безье (C_1, C_2). Если выпуклые оболочки этих кривых не пересекаются, между C_1 и C_2 пересечения нет. Алгоритм завершается.
2. Пусть выпуклые оболочки кривых пересекаются. Если C_1 и C_2 настолько “малы“, что для них выполняется некоторое определяемое пользователем

условие завершаемости, пересечение считается найденным, и алгоритм завершается.

3. Пусть указанное условие не выполнено. Делением C_1 и C_2 пополам получаем новые кривые $C_{1.1}, C_{1.2}, C_{2.1}, C_{2.2}$, и для каждой пары $(C_{1.1}, C_{2.1})$, $(C_{1.1}, C_{2.2})$, $(C_{1.2}, C_{2.1})$ и $(C_{1.2}, C_{2.2})$ перезапускаем алгоритм сначала.

Условие завершаемости, определяемое пользователем в п. 2. алгоритма, может самым различным образом отвечать нужной степени точности (например, с заданным ε могут сравниваться длины кривых C_1 и C_2). Алгоритм легко модифицируется в соответствии с тем, достаточно ли просто выявить факт пересечения, или также нужно узнать соответствующие точки. Во втором случае можно использовать различные методы приближения решения. Здесь же становится виден недостаток алгоритма: к одной и той же точке пересечения по разным направлениям могут вести различные ветви рекурсии, в результате чего можно получить несколько различных приближений данной точки, лежащих в достаточно малой её окрестности.

1.3.2 Алгоритм де Кастельжо

Ключевым моментом алгоритма пересечения является деление на каждом шаге кривой Безье пополам. Для осуществления этого оптимально использовать *алгоритм де Кастельжо*. Его основное предназначение суть вычисление для некоторого значения параметра t_0 координат соответствующей точки кривой Безье $B(t_0)$, однако в процессе выполнения естественным образом вычисляются координаты опорных точек кривых, получающихся делением исходной кривой точкой $B(t_0)$. Именно, рассмотрим следующие рекуррентные соотношения:

$$P_i^{(0)} = P_i, \quad i = 0, ..., n,$$

$$P_i^{(j)} = P_i^{(j-1)}(1 - t_0) + P_{i+1}^{(j-1)} t_0, \quad i = 0, ..., n - j, \quad j = 1, ...n.$$

Вычислить $B(t_0)$ можно за $\binom{n}{2}$ шагов по формуле

$$B(t_0) = P_0^{(n)},$$

причём кривые Безье, полученнные в результате деления, соответственно представляются опорными точками

$$P_0^{(0)}, P_0^{(1)}, ..., P_0^{(n)}$$

и

$$P_0^{(n)}, P_1^{(n-1)}, ..., P_n^{(0)}.$$

Алгоритм де Кастельжо является самым распространённым для решения задачи деления кривой Безье. Тем не менее, сложность алгоритма составляет $\mathcal{O}(n!)$, где n — порядок кривой, в силу чего он эффективен лишь при малых значениях n. Учитывая, что наиболее употребительными являются кубические кривые Безье (случай $n = 4$), а для представления дуг вполне достаточно сплайнов Безье, которые состоят самое большее из кубических кривых Безье, на практике ожидается приемлемая эффективность работы алгоритма.

Описанный выше алгоритм поиска пересечений кривых Безье чаще всего используется именно в связке с алгоритмом де Кастельжо.

2 Конструкторский раздел

Библиотека должна предоставлять средства для работы с кривыми Безье, анализа диаграмм, а также интеграции с существующими решениями визуализации графов. Естественным образом в проекте выделяются три модуля, где каждый последующий непосредственно зависит от предыдущего и использует его функциональность: модуль геометрических вычислений, модуль анализа диаграмм и модуль интеграции.

Для представления внутренних структур данных наиболее удобной оказывается объектно-ориентированная модель, вследствие чего описание этих структур излагается с использованием соответствующей терминологии. Использующиеся типы данных будут подробно описаны в технологическом разделе 3.

2.1 Модуль геометрических вычислений

Модуль геометрических вычислений предоставляет интерфейс для создания кривых Безье по опорным точкам, деления кривой Безье на две части точкой, соответствующей некоторому значению параметра $t \in [0, 1]$, а также пересечения кривых Безье. В практической реализации оказалось эффективным использование двух вариантов функции пересечения: одна устанавливает лишь факт его наличия, а другая возвращает все (уникальные) точки пересечения. Первая из этих функций завершает работу при обнаружении первого пересечения; во второй же необходим обход всех ветвей рекурсии, а также исполнение логики удаления дублирующихся точек, что замедляет её работу. Практическое использование функций пересечения описывается в разделе 2.2.3.

В модуле реализуются классы и методы трёх геометрических объектов: точки, прямоугольника и самой кривой Безье.

2.1.1 Точка

Точка (класс `Point`) представляет точку плоскости с полями `x` и `y`. Для удобства использования `Point` тривиально определены операторы сравнения, сложения, разности точек и умножения точки на число. Также определены

— метод `CenterWith` для вычисления центра между двумя точками;

— метод `IsInNeighborhood` для определения принадлежности одной точки достаточно малой окрестности другой точки;
— метод `Dump` для вывода информации о точке в заданный выходной поток (такой метод определён для каждого объекта, и в дальнейшем не будет явно упоминаться);

2.1.2 Прямоугольник

Прямоугольник (класс `Rectangle`) используется в качестве выпуклой оболочки кривой Безье. Фактически `Rectangle` — это наименьший прямоугольник, содержащий все опорные точки данной кривой. Выбор такого прямоугольника в качестве выпуклой оболочки имеет свои достоинства и недостатки: прямоугольник лёгок в построении, и для двух таких прямоугольников вычислительно просто определяется факт их пересечения. Однако если прямоугольник — выпуклая оболочка горизонтальной или вертикальной кривой Безье, то он вырождается в отрезок прямой, и использовать наиболее естественное — площадь прямоугольника — при определении завершаемости уже не получится.

Прямоугольник представляется своими верхней левой и нижней правой точками и хранит соответствующее поля `top_left` и `bottom_right`. У `Rectangle` определены

— метод `Perimeter` для вычисления периметра прямоугольника;
— метод `Center` для вычисления точки-центра прямоугольника;
— метод `IsOverlap` для проверки наличия пересечения двух прямоугольников;

Реализация метода `IsOverlap` основывается на простой идее: прямоугольники не пересекаются (не накладываются), если один из них находится полностью левее или выше другого; проверка соотношений соответствующих координат даёт решение.

2.1.3 Кривая Безье

Кривая Безье (класс `Curve`) представляется набором опорных точек и хранит соответствующее поле класса `points`. Для `Curve` определены

— метод `BoundingRectangle`, вычисляющий прямоугольник-выпуклую оболочку кривой;
— метод `Split`, делящий кривую на две её точкой со значением параметра t;
— метод `IsIntersect`, проверяющий наличие пересечения кривых;
— метод `Intersect`, вычисляющий приближённые точки пересечения кривых;

В методе `BoundingRectangle` определяются самые левая, правая, верхняя и нижняя координаты всех опорных точек, по которым строятся `top_left` и `bottom_right` точки конструктора `Rectangle`. Метод `Split` использует алгоритм де Кастельжо без каких-либо модификаций.

Метод `IsIntersect` реализует описанный в разделе 1.3.1 алгоритм пересечения кривых Безье. Метод принимает некоторое достаточно малое значение $\varepsilon > 0$, использующеся в условии завершения: сумма периметров прямоугольников кривых должна быть меньше ε (как было сказано в разделе 2.1.2, с использованием площади прямоугольников могут возникнуть проблемы если пересекаемые кривые вертикальные или горизонтальные).

Метод `Intersect` является модификацией `IsIntersect` с вычислением приблизительных точек пересечения. Когда кривые достаточно малы, примерная точка пересечения вычисляется как центр отрезка, соединяющего центры прямоугольников кривых. С коррекцией ε можно добиться сколько угодно малой погрешности. Чтобы не хранить дубликаты, перед добавлением каждой точки проверяется, что она не лежит ни в какой δ-окрестности найденной точки для некоторого $\delta > 0$.

2.2 Модуль анализа диаграмм

Модуль анализа диаграммы предоставляет все классы и методы, необходимые для представления топологического графа и выявления у него снижающих читаемость компонентов в смысле классов непланарности.

2.2.1 Вершина

Вершина (класс `Vertex`) графа наследуется от класса `Point`, добавляя новое опциональное поле `label`, соответствующее метке вершины. Использование наследования позволяет сократить дублирование кода; `Point` и `Vertex` взаимозаменяемы, что удовлетворяет принципу подстановки Лисков.

2.2.2 Вектор

Вектор (класс `Vector`) представляет вектор на плоскости и является небольшим вспомогательным классом, предоставляющим некоторые операции для работы со свободными векторами:

— метод `AngleWith` позволяет вычислить угол между двумя векторами;

— метод `Norm` вычисляет норму вектора;

— метод `ScalarProduct` вычисляет скалярное произведение векторов;

— метод `CollinearTo` устанавливает коллинеарность двух векторов;

В реализации методов используются базовые математические сведения. Угол φ между векторами $\vec{u} = \{x_u, y_u\}$ и $\vec{v} = \{x_v, y_v\}$ вычисляется как

$$\varphi = \arccos \frac{|(\vec{u}, \vec{v})|}{\|\vec{u}\| \|\vec{v}\|},$$

где

$$(\vec{u}, \vec{v}) = x_u x_v + y_u y_v, \quad \|\vec{u}\| = \sqrt{x_u^2 + y_u^2}, \quad \|\vec{v}\| = \sqrt{x_v^2 + y_v^2}.$$

Коллинеарность векторов равносильна выполнению равенства $x_u y_v = x_v y_u$.

2.2.3 Ребро

Ребро (класс `Edge`) топологического графа хранит в полях класса свои вершины `start` и `end`, а также сплайн Безье, представляющий ребро. Основные методы класса:

— метод `IsIntersect`, проверяющий пересечение двух рёбер;

— метод `IsStraightLine`, проверяющий возможность представления ребра отрезком прямой;

С учётом реализации определение пересечения рёбер можно переформулировать следующим образом:

> Два ребра пересекаются, если существует пара кривых Безье, являющихся участками соответствующих этим рёбрам сплайнов и имеющих общую точку пересечения, отличную от точек на концах сплайнов.

Таким образом, вычислять координаты точек пересечения необходимо только для краевых участков сплайнов Безье для проверки, что некоторая точка пересечения не совпадает с конечными точками рёбер. Для промежуточных пар кривых достаточно выявить просто факт наличия пересечения.

В соответствии с описанным реализован метод `IsIntersect`. В методе `IsStraightLine` проверяется, что все направляющие векторы сторон многоугольника Безье коллинеарны.

2.2.4 Топологический граф

Топологический граф (класс `Graph`) хранит информацию о своих вершинах и рёбрах — `edges` и `vertices` соответственно — а также предоставляет API для выявления компонентов, не удовлетворяющих классам непланарных графов.

Во всех функциях выявления используется метод `Intersections`, возвращающий информацию о пересечениях рёбер. Для каждого ребра выявляются рёбра, с которыми у него есть пересечение. Результат вызова `Intersections` незамысловато назовём `intersections`. Здесь и вообще при проведении операций над рёбрами используются не сами рёбра, а их ключи — соответствуюищие индексы в массиве `edges`, что выгодно с точек зрения и написания, и исполнения функций.

`CheckPlanar` Метод возвращает все рёбра диаграммы, пересекаемые более k раз. Для этого достаточно выбрать те рёбра графа, которым соответствует $k-1$ и более рёбер `intersections`.

`CheckQuasiPlanar` Метод возвращает все наборы k взаимно пересекающихся рёбер.

1. Определяются рёбра-кандидаты, имеющие хотя бы $k-1$ пересечение. Если на данный момент или следующих шагах алгоритма число кандидатов

станет меньше k, исходный граф не содержит k взаимно пересекающихся рёбер.

2. Обходим все рёбра-кандидаты. Выбираем очередного кандидата, и среди всех рёбер, которые он пересекает, удаляем те, что не являются кандидатами. Если кандидат теперь пересекает менее $k-1$ ребра, он более не является кандидатом и удаляется. Процесс продолжается, пока кандидаты удаляются.
3. Получено несколько систем, где k и более рёбер взаимно пересекаются. Если не стоит задача выбрать в этих системах все группы по k взаимно пересекающихся рёбер, можно просто вернуть найденные рёбра как участвующие в недопустимых компонентах, иначе работа алгоритма продолжается.
4. Заметим, что если интерпретировать полученные рёбра как вершины некоторого нового графа, а при наличии пересечения у рёбер связывать соответствующие вершины нового графа, то задача сводится к выделению максимальных m-клик для $m > k$, что, как известно, является NP-полной задачей. Можно использовать различные способы для выделения максимальных m-клик. В реализации используется рекурсивный обход по всем рёбрам с выделением максимальных замкнутых относительно пересечения групп. Таким образом, для каждой m-клики остаётся найти её сочетания по k элементов.

`CheckSkewness` Метод возвращает все сочетания из $k+1$ и более рёбер, с удалением которых граф станет планарным.

— В `intersections` выбирается очередное ребро с наибольшим числом пересечений. Если оно ни с чем не пересекается, выполняется переход к следующему шагу алгоритма. Иначе нужно удалить ребро (соответственно, и все его пересечения) в `intersections` и повторить текущий шаг.

— Если количество удалённых рёбер не более k, диаграмма принадлежит skewness-k классу. В противном случае любое сочетание из $k+1$ указанного ребра не удовлетворяет классу.

`CheckGridFree` Для заданных k, l метод возвращает все (k,l)-сетки из k и l рёбер соответственно, где каждое ребро первой группы персекает всякое ребро второй группы.

1. Выбираются рёбра-кандидаты k-групп, пересекающие хотя бы $l-1$ другое ребро.
2. Выбирается пара кандидатов (c_1, c_2) k-групп. Для неё вычисляется множество рёбер L, которые пересекают и c_1, и c_2. Если $|L| < l$, рёбра множества не могут составлять l-группу, и выбирается новая пара кандидатов. Иначе для каждого сочетания из l элементов множества L формируется связь с промежуточным множеством K кандидатов в k-группы, куда изначально входят c_1 и c_2. Если при обработке некоторой пары кандидатов (c_i, c_j) образуется новое множество L', $|L'| \geq l$, причём некоторое сочетание из l элементов L' уже встречалось до этого, то c_i, c_j добавляются в соответствующее этому сочетанию множество K, если их там не было до этого.
3. По завершении перебора из всех промежуточных K множеств таких, что $|K| \geq k$, берутся сочетания по k элементов. Для всех полученных множеств фиксируется связь со множеством L, соответствующего данному K. В результате получаем связи, описывающие все (k,l)- сетки исходного графа.

`CheckRAC`, `CheckACE`, `CheckACL` В геометрическом графе метод `CheckRAC` возвращает пары рёбер, пересекающиеся под углом, отличным от $\frac{\pi}{2}$; метод `CheckACE` возвращает пары рёбер, пересекающиеся под углом, не равным заданному α, а метод `CheckACL` возвращает пары рёбер, пересекающиеся под углом меньшим заданного α. Все три указанных метода реализуют по сути единый алгоритм; различие возникает лишь в условии, которому должен удовлетворять угол между двумя пересекающимися рёбрами. Более того, вызов `CheckRAC` тривиально сводится к параметризованному значением угла вызову `CheckACE`.

Упомянутый общий алгоритм реализуется вспомогательным методом `CheckAC`. Его вызов имеет смысл лишь применительно к геометрическому графу. Для выяснения является ли граф геометрическим нужно проверить, что все его

рёбра представляются отрезками прямых — это выполняется с использованием метода `IsStraightLine` класса `Edge`.

В реализации метода `CheckAC` для каждого ребра вычисляется его направляющий вектор, представляемый классом `Vector`. Затем выявляются пары векторов, угол между которыми не удовлетворяет определённому условию. Метод возвращает все пары соответствующих рёбер.

2.3 Модуль интеграции

Предназначение модуля интеграции — автоматический ввод диаграмм, генерируемых различными средствами визуализации графов. Сегодня подобные программы используются повсеместно, и возможность непосредственного ввода результата работы некоторого визуализатора значительно было упростила процесс использования библоитеки.

2.3.1 Graphviz

Программное обеспечение Graphviz [3] — наиболее распространённое средство визуализации графов. Для описания графов повсеместно используется язык DOT [4]; Graphviz принимает описание графа на этом языке и генерирует соответствующую диаграмму.

Диаграмму необходимо привести к удобному для анализа виду. Утилита dot2tex [5] позволяет представить диаграмму Graphviz последовательностью команд TikZ окружения макрорасширения LaTeX [6] системы вёрстки TeX. Окружение TikZ используется, в частности, для ручного построения диаграмм, когда явно задаются координаты каждой вершины, а рёбра представляются последовательностью соединённых участков прямых и квадратичных, кубических кривых Безье (также с явно заданными коодинатами контрольных точек). Таким образом, становится возможным анализ генерируемого LaTeX файла с автоматическим представлением диаграммы внутренними структурами данных.

В библиотеке это осуществляется статическим методом `Graphviz` класса `Graph`, принимающим на вход путь к нужному `DOT`-файлу с описанием графа. Также предусмотрена метода, принимающий на вход непосредственно описание на языке DOT.

1. Вызовом утилиты `dot2tex` с флагами `-tmath` и `-ftikz` порождается LaTeX-файл строго определённого формата.
2. Строки файла сканируются в поиске начала окружения `TikZ`.
3. Считывается непрерывный блок с определениями вершин графа, описываемых `TikZ`-командой `\node`. В префиксе каждого определения считываются метка вершины и её координаты в единицах измерения `bp` (большой пункт, `1bp` $\approx$ `0.35mm`).
4. Считывается непрерывный блок определений рёбер, описываемых `TikZ`-командой `\draw`. Средствами `dot2tex` каждая дуга представляется сплайном Безье, состоящим из кубических кривых Безье и только них. Начальная и конечная точка всякого сплайна есть метка какой-либо прочитанной до этого вершины; промежуточные точки представляются непосредственно в координатах (в единицах измерения `bp`). Если в `DOT`-описании ребра для него указывался атрибут, непосредственно за данной командой `\draw` следует ещё одна команда этого типа, в которой указывается значение атрибута и координаты точки его отображения. Таким образом, во внутренних структурах можно сохранить достаточно много информации о диаграмме.
5. Считывается конец `TikZ` окружения. Промежуточный файла удаляется, возвращается построенная диаграмма.

3 Технологический раздел

Языком реализации был выбран C++17. Язык предоставляет все необходимые средства для работы в объектно-ориентированной парадигме, гибкие инструменты представления объектов в памяти и управления их временем жизни, обильную стандартную библиотеку, предлагающую эффективно реализованные алгоритмы и структуры данных и, наконец, возможность применения множества оптимизаций к итоговой программе.

3.1 Руководство администратора

Наряду со стандартной используется библиотека Boost [7], предлагающая различные экспериментальные и специализированные инструменты, ещё не закреплённые в стандарте языка. Поэтому пользователям Linux необходимо предварительно установить библиотеку `libboost-all-dev`.

Как будет описано в разделе тестирования 4, для проверки работоспособности написаны end-to-end и unit-тесты, использующие библиотеки Google Test и Google Mock [8]. Для их запуска пользователям Linux необходимо установить библиотеки `libgtest-dev` и `libgmock-dev`.

Помимо того, предварительно необходимо установить утилиту dot2tex.

Итоговый проект представляется заголовочной библиотекой (header-only library) — это удобный способ подключения сторонних C++ библиотек, когда объявления и определения всех функций, классов и других объектов располагаются в едином файле, и для использования библиотеки достаточно подключить файл директивой `#include`.

Во время разработки для удобства и уменьшения время компиляции библиотека декомпозировалась на описанные в конструкторском разделе 2 модули. Для сборки проекта использовалась система CMake [9]. Чтобы использовать эту версию библиотеки, нужно в корневой директории проекта выполнить команду

```
cmake -S . -B . -DBUILD_TYPE=Release && cmake --build build.
```

Это сгенерирует конфигурационные файлы проекта и скомпилирует результат.

3.2 API

При проектировании интерфейса прикладного программирования библиотеки в приоритете было использование современных средств языка C++.

— функции, вычисление и применение результата которых возможно на этапе компиляции, помечаются ключевым словом `constexpr`;

— функции, не выбрасывающие исключения, помечаются `noexcept`, что позволяет компилятору порождать оптимизированный ассемблерный код;

— методы, возвращающие ссылки на внутреннее состояние объекта, аннотированы `lvalue`-ссылками для предотвращения появления *висячих ссылок*;

— функции при необходимости перегружаются для `lvalue` и `rvalue` категорий значений одних и тех же входных параметров, что позволяет эффективно использовать временные объекты;

— все неизменяемые параметры функции обязательно помечаются `const`;

При описании API относительно небольшие фрагменты кода будут представлены на месте, остальные же вынесены в приложение.

3.2.1 Структура `Point`

Поскольку для объектов `Point` не требуется выполнения нетривиальных инвариантов, тип представляется *структурой* языка C++, т.е. поля `Point::x` и `Point::y` по умолчанию публичные. В листинге 1 представлен API `Point`.

Поскольку для типа определён виртуальный деструктор, от `Point` можно наследоваться. Вместе с тем метод `Point::Dump`, использующийся для вывода информации о даной точке в объект `std::ostream`, явно помечен `virtual` как подлежащий переопределению наследником.

Параметр `eps` метода `Point::IsInNeighborhood` соответствует величине окрестности при приблизительном сравнении точек. На практике, как правило, достаточно значения в `1e-3`.

Листинг 1: Публичные методы `Point`

```
namespace bezier {

struct Point {
  double x, y;

  Point(const double x, const double y) noexcept;
  virtual ~Point(){};

  bool operator==(const Point &rhs) const noexcept;
  Point operator+(const Point &rhs) const noexcept;
  Point operator-(const Point &rhs) const noexcept;
  Point operator*(const double rhs) const noexcept;

  Point CenterWith(const Point &p) const noexcept;
  bool IsInNeighborhood(const Point &p, const double eps) const noexcept;
  virtual void Dump(std::ostream &os) const;
};

}  // namespace bezier
```

Листинг 2: Публичные методы `Rectangle`

```
namespace bezier {

class Rectangle final {
 public:
  Rectangle(const Point &top_left, const Point &bottom_right);

  const Point &get_top_left() const &noexcept;
  const Point &get_bottom_right() const &noexcept;

  double Perimeter() const noexcept;
  bool IsOverlap(const Rectangle &r) const noexcept;
  Point Center() const noexcept;
  void Dump(std::ostream& os) const;

  // ... (private section)
};

}  // namespace bezier
```

3.2.2 Класс `Rectangle`

В объектах типа `Rectangle` (листинг 2) необходимо выполнение корректного соотношения между координатами верхней левой и нижней правой его точек. Для сохранения инварианта установка точек происходит лишь в конструкторе `Rectangle`; при нарушении соотношения координат выбрасывается соответствующее исключение, и объект не создаётся. Доступ к точкам осуществляется вызовом методов `Rectangle::get_top_left` и `Rectangle::get_bottom_right`.

3.2.3 Класс `Curve`

API класса `Curve` представлен в листинге 7. Возможность создавать кривые Безье порядка 0 или 1 на практике по сути бесполезна — применение к ним алгоритмов для работы с кривыми Безье скорее всего породит бессмысленный результат. Поэтому конструктор кривой принимает массив контрольных точек размера хотя бы 2 — это инвариант класса `Curve`; в противном случае конструктор выбрасывает исключение.

Конструктор принимает как `const &` (lvalue), так и `&&` (rvalue) ссылки на `std::vector` опорных точек. Как было сказано ранее, это позволяет эффективнее использовать временные объекты, и содержимое `std::vector` может быть не скопировано, а *перемещено* при инициализации поля класса `Curve` в соответствии с `move`-семантикой языка C++.

В реализации библиотеки не используется непосредственное выделение и освобождение динамической памяти при создании и удалении объектов. Для этого применяются *умные указатели* `std::unique_ptr` и `std::shared_ptr`, предоставляющие единоличное и совместное владение объектом соответственно, что безопаснее в использовании и в целом идиоматичнее для языка C++. Поэтому, в частности, объекты `Curve` часто представляются посредством `std::unique_ptr`.

Параметр `t` метода `Curve::Split` должен принимать значение в промежутке $[0, 1]$ по определению кривой Безье.

3.2.4 Стуктура `Vertex`

API структуры `Vertex` представлен в листинге 3.

Листинг 3: Публичные методы `Vertex`

```
namespace graph {

struct Vertex;

using VertexSptrConst = std::shared_ptr<const Vertex>;

struct Vertex final : public bezier::Point {
  std::string label;

  Vertex(const double x, const double y, const std::string& label)
      noexcept;

  void Dump(std::ostream& os) const override;
};

}  // namespace graph
```

3.2.5 Структура `Vector`

API структуры `Vector` представлен в листинге 4.

Листинг 4: Публичные методы `Vector`

```
namespace utils {

struct Vector final {
  double x, y;

  Vector(const bezier::Point &p) noexcept;
  Vector(const graph::Edge &e);

  double AngleWith(const Vector &v) const;
  double Norm() const;
  double ScalarProduct(const Vector &v) const noexcept;
  bool CollinearTo(const Vector &v) const noexcept;
};

}  // namespace utils
```

3.2.6 Класс `Edge`

API класса `Edge` представлен в листинге 5. Точка `start` ребра обязана совпадать с первой точкой его сплайна Безье; симметрично, `end` совпадает с конечной точкой сплайна.

Конструктор `Edge` принимает вершины и может принимать или не принимать соответствующий сплайн Безье. В последнем случае сплайн автоматически создаётся из единственной линейной кривой Безье, т.е. ребро описывается отрезком прямой.

Листинг 5: Публичные методы `Edge`

```
namespace graph {

class Edge;

using EdgeSptrConst = std::shared_ptr<const Edge>;

class Edge final {
 public:
  Edge(VertexSptrConst start, VertexSptrConst end);
  Edge(VertexSptrConst start, VertexSptrConst end,
       std::vector<bezier::CurveUptrConst> &&curves);
  Edge(VertexSptrConst start, VertexSptrConst end,
       const std::vector<bezier::CurveUptrConst> &curves);

  const VertexSptrConst &get_start() const &noexcept;
  const VertexSptrConst &get_end() const &noexcept;
  const std::vector<bezier::CurveUptrConst> &get_curves() const &noexcept;

  bool IsIntersect(const Edge &e) const;
  bool IsStraightLine() const;

  // ... (private section)
};

}  // namespace graph
```

3.2.7 Класс `Graph`

В листинге 8 представлены основные методы создания и наполнения графа, в листинге 9 — методы выявления его компонентов, не удовлетворяющих некоторым классам непланарных графов.

Конструктор `Graph` может принимать как просто контрольные точки, так и точки вместе соответствующими рёбрами типа `std::shared_ptr`. Во втором случае необходимо, чтобы все объекты были согласованно связаны. В целом, для хранения вершин и рёбер в `Graph` используется `std::shared_ptr`, чтобы можно было возвращать пользователю указатель на один и тот же объект и продлевать время жизни объекта, пока им пользуются.

Статический метод `Graph::GraphvizFromFile` принимает путь к файлу с описанием графа на языке DOT и конструирует диаграмму посредством визуализатора `Graphviz`. В методе автоматически создаётся и удаляется временный файл. Метод `Graph::GraphvizFromText` непосредственно принимает описание графа на языке DOT.

Методы `Graph::AddVertex` и `Graph::AddVertices` используются для добавления одной или нескольких вершин соответственно. Аналогично методы `Graph::AddEdge` и `Graph::AddEdges` добавляют ребро, связывающее существующие вершины с индексами `start` и `end`; если ребо — отрезок прямой, можно воспользоваться специальными методами `AddSLEdge` и `AddSLEdges`.

Метод `Graph::CheckPlanar` принимает параметр целочисленного неотрицательного типа `std::size_t k` ≥ 1 и возвращает в векторе (контейнере C++) `std::vector<EdgeSptrConst>` все рёбра диаграммы, пересекаемые более `k` раз. При некорректном значении параметра функция (здесь и далее) выбрасывает исключение.

Метод `Graph::CheckQuasiPlanar` принимает параметр `k` ≥ 3 и возвращает в векторе `std::vector<std::vector<EdgeSptrConst>>` все сочетания из `k` взаимнопересекающихся рёбер графа.

Метод `Graph::CheckSkewness` принимает параметр `k` ≥ 1 и возвращает пустой вектор, если удаление не более `k` рёбер позволяет сделать диаграмму планарной, и вектор `std::vector<std::vector<EdgeSptrConst>>` всех сочетаний из `k + 1` рёбер диаграммы, не удовлетворяющих требованиям класса skewness-k.

Метод Graph::CheckGridFree принимает параметры k $\geq$ 1, l $\geq$ 1, и возвращает вектор всех (k, l)-сеток графа. (k, l)-сетка представляется вспомогательной структурой KLGrid с полями k_group и l_group типа std::vector<EdgeConstSptr>.

Методы Graph::CheckACE и Graph::CheckACL принимают угол alpha в радианах и возвращают вектор всех пар рёбер std::vector<std::pair<EdgeConstSptr, EdgeConstSptr>>, пересекающихся под углом не равным alpha или меньшим alpha соответственно. Вызов метода Graph::CheckRAC сводится к вызову метода Graph::CheckACE, параметризованного углом alpha $= \frac{\pi}{2}$. Три описанных метода выбрасывают исключение, если диаграмма не является геометрическим графом.

4 Тестирование

Во время написания библиотеки активно применялось `end-to-end` и `unit`-тестирование. Техника `end-to-end` используется для проверки работоспособности целых модулей программы с задействованием всей промежуточной функциональности, используемой в модуле. Unit-тесты используются для проверки работоспособности отдельных функций и изолированных участков кода. При написании тестов учтывались различные случаи представления входных данных.

Указаные end-to-end и unit-тесты реализовывались с помощью C++ библиотек Google Test и Google Mock [8]. Примеры end-to-end тестов представлены в листингах 13 и 14, unit-тестов — в листингах 10 и 11.

4.1 Анализ диаграммы графа K_{10}

Целью данного теста является демонстрация выявления недопустимых компонентов автоматически созданной диаграммы Graphviz. На рисунке 3 изображена диаграмма графа K_{10}, т.е. полного графа с 10 вершинами. Описание графа на языке DOT приведено в листинге 12.

Пусть `g` — объект класса `Graph` — представляет эту диаграмму. Вызов `g.CheckPlanar(6)` выявляет рёбра `b -- f`, `b -- h`, `c -- i`, `d -- j`. При вызове `g.CheckPlanar(7)` неудовлетворяющих рёбер не обнаруживается.

Вызов `g.ChekQuasiPlanar(3)` выявляет следующие недопустимые сочетания рёбер:

— {`c -- g`, `d -- e`, `d -- f`};
— {`a -- e`, `c -- i`, `c -- j`};
— {`a -- f`, `c -- i`, `c -- j`};
— {`a -- f`, `c -- h`, `e -- j`};
— {`a -- f`, `c -- i`, `e -- j`};
— {`b -- i`, `d -- j`, `g -- j`};
— {`b -- h`, `d -- j`, `g -- j`};
— {`a -- h`, `c -- i`, `e -- j`};
— {`a -- i`, `b -- j`, `g -- j`};
— {`a -- g`, `b -- i`, `d -- j`};

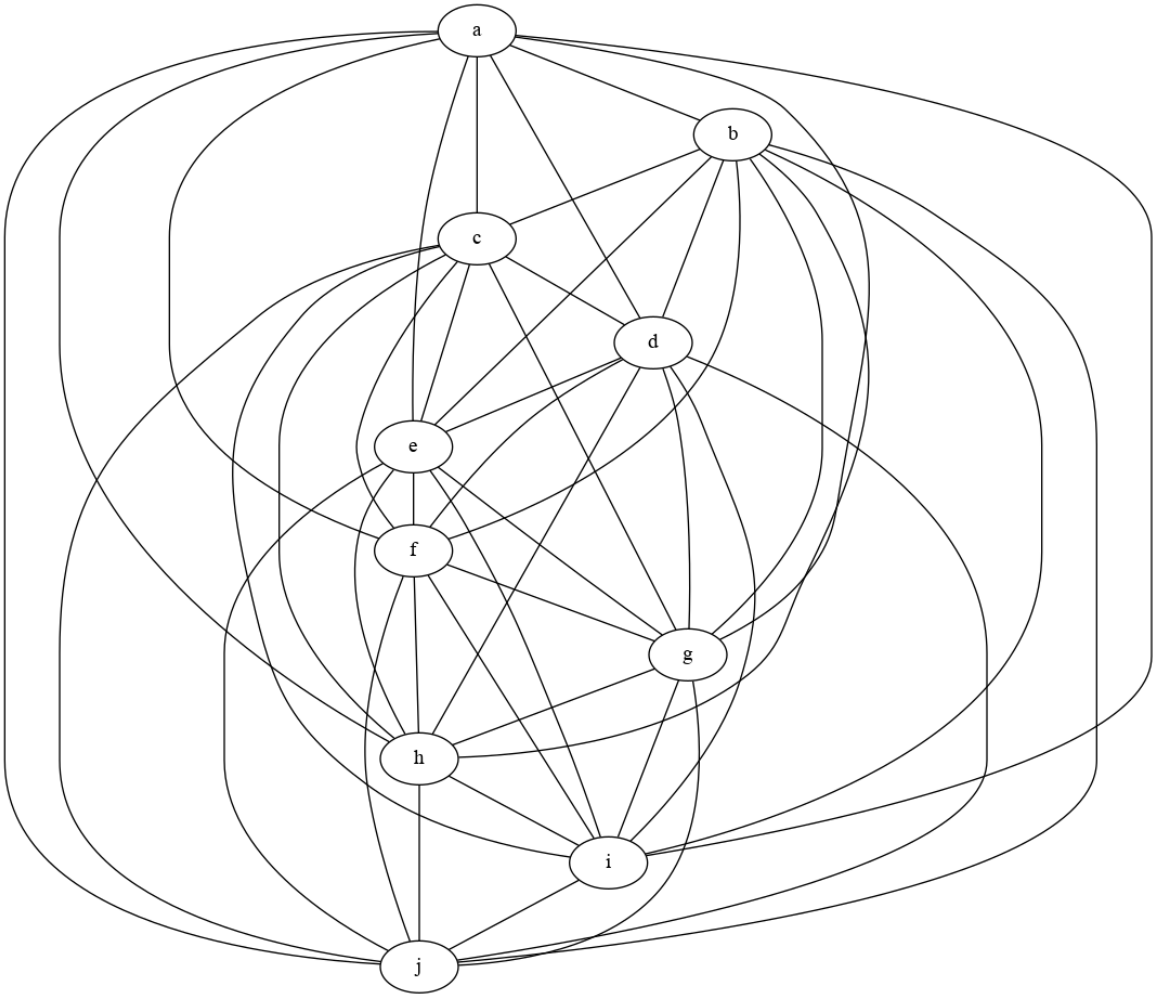

Рисунок 3 — Диаграмма Graphviz графа K_{10}

— {b -- f, c -- g, d -- h};
— {b -- h, d -- i, g -- j};
— {a -- i, d -- j, g -- j};
— {b -- f, d -- h, e -- g};
— {a -- g, b -- h, d -- j};
— {a -- g, b -- h, d -- i};
— {d -- h, e -- i, f -- g};
— {b -- f, d -- h, e -- i};
— {a -- h, c -- i, c -- j};
— {a -- h, c -- i, f -- j};

Вызов `g.CheckQuasiPlanar(4)` не выявляет никаких сочетаний из 4 взаимно пересекающихся рёбер.

Вызов `g.CheckSkewness(20)` выявляет единственное сочетание из 21 ребра: {`f -- i, c -- g, b -- g, a -- i, a -- d, c -- h, d -- f, b -- h, d -- h, c -- i, d -- j, e -- h, a -- e, e -- i, a -- f, b -- f, g -- j, a -- g, a -- h, c -- f, b -- e`}. Вызов `g.CheckSkewness(21)` не выявляет ни одного сочетания рёбер.

Вызов `g.CheckGridFree(3, 2)` выявляет следующие $(3, 2)$-сетки:

— {`b -- j, d -- i, d -- j`}, {`a -- g, g -- j`};
— {`c -- h, c -- i, c -- j`}, {`a -- e, a -- f`};
— {`b -- i, d -- i, d -- j`}, {`a -- g, g -- j`};
— {`b -- i, b -- j, d -- j`}, {`a -- g, g -- j`};
— {`b -- i, b -- j, d -- i`}, {`a -- g, g -- j`};
— {`b -- h, d -- i, d -- j`}, {`a -- g, g -- j`};
— {`c -- f, c -- h, c -- i`}, {`a -- e, e -- j`};
— {`a -- e, a -- f, a -- h`}, {`c -- i, c -- j`};
— {`a -- f, a -- h, c -- i`}, {`c -- j, e -- j`};
— {`a -- g, b -- g, b -- h`}, {`d -- i, d -- j`};
— {`b -- h, b -- i, d -- i`}, {`a -- g, g -- j`};
— {`b -- h, b -- j, d -- j`}, {`a -- g, g -- j`};
— {`b -- f, d -- f, d -- h`}, {`c -- g, e -- i`};
— {`b -- f, d -- f, d -- h`}, {`c -- g, e -- g`};
— {`b -- f, d -- f, d -- h`}, {`e -- g, e -- i`};
— {`b -- f, b -- g, b -- h`}, {`d -- i, d -- j`};
— {`a -- g, b -- f, b -- g`}, {`d -- i, d -- j`};
— {`a -- i, b -- h, b -- i`}, {`d -- j, g -- j`};
— {`a -- h, c -- h, c -- i`}, {`e -- j, f -- j`};
— {`b -- h, b -- i, b -- j`}, {`a -- g, g -- j`};
— {`b -- h, b -- i, d -- j`}, {`a -- g, g -- j`};
— {`a -- g, b -- f, b -- h`}, {`d -- i, d -- j`};
— {`b -- h, b -- j, d -- i`}, {`a -- g, g -- j`};

Сетки большей размерности данный граф не содержит.

Вызовы методов `Graph::CheckRAC`, `Graph::CheckACE` и `Graph::CheckACL` для данного графа по определению недопустимы.

4.2 Анализ геометрического графа

Целью данного теста является демонстрация ручного ввода диаграммы, а также автоматического анализа углов, образуемых пересечением его рёбер.

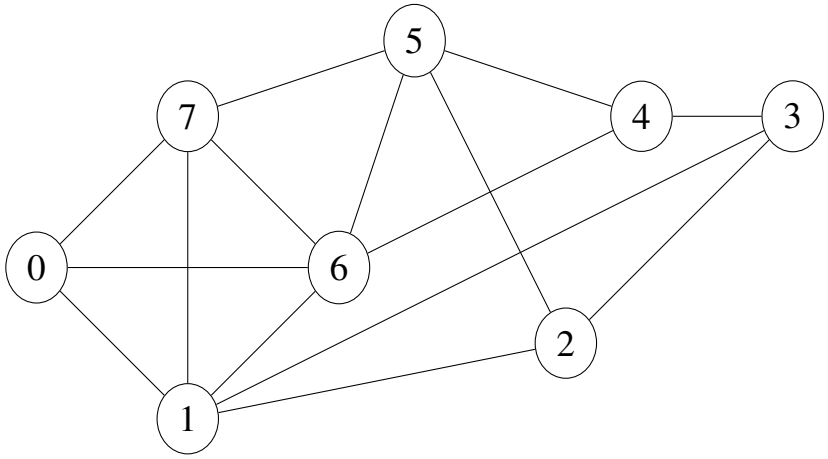

Рисунок 4 — Геометрический граф G

В листинге 6 представлен тест, написанный с использованием средств библиотеки Google Test, демонстрирующий ввод небольшого геометрического графа G (рисунок 4), и проверяющий его принадлежность классу RAC.

Листинг 6: Ручной ввод и анализ небольшого графа

```
namespace graph {

namespace {

TEST(Graph, ChechRAC) {
  Graph g({
      {2, 3, "0"}, {4, 1, "1"}, {9, 2, "2"}, {12, 5, "3"},
      {10, 5, "4"}, {7, 6, "5"}, {6, 3, "6"}, {4, 5, "7"}});

  g.AddSLEdges({
    {0, 1}, {1, 2}, {1, 3}, {1, 6}, {1, 7}, {2, 3}, {2, 5}, {3, 4},
    {4, 5}, {5, 6}, {5, 7}, {6, 0}, {6, 4}, {7, 0}, {7, 6}});

  const auto unsat_rac = g.CheckRAC();
  ASSERT_TRUE(unsat_rac.empty());
}

}  // namespace

}  // namespace graph
```

5 Заключение

В результате работы над проектом поставленные цель и задачи во многом были достигнуты. Использование библиотеки действительно позволяет выявлять снижающие читаемость компоненты диаграмм многих классов непланарных графов. Автоматический ввод Graphviz-диаграмм по описанию графа на языке DOT упрощает взаимодействие с библиотекой.

Тем не менее, теоретическая сложность реализованных алгоритмов очень высока, поэтому полные графы и графы относительно больших размеров обрабатываются продолжительное время. Это связано и с особенностями реализации, и высокой сложностью поставленных задач в принципе.

С другой стороны, выстроенная архитектура проекта масштабируема и допускает добавление методов обработки новых классов непланарных графов. Более того, с использованием полиморфизма, предоставляемого языком C++, нетрудно построить новые представления рёбер другими объектами компьютерной графики, что может быть эффективнее в той или иной ситуации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. A Survey on Graph Drawing Beyond Planarity. — URL: https://arxiv.org/abs/1804.07257.
2. A Primer on Bézier Curves. — URL: https://pomax.github.io/bezierinfo/.
3. Документация Graphviz. — URL: https://graphviz.org/.
4. Документация языка DOT. — URL: https://graphviz.org/doc/info/lang.html.
5. Документация утилиты dot2tex. — URL: https://dot2tex.readthedocs.io/en/latest/.
6. Документация системы вёрстки LaTeX. — URL: https://www.latex-project.org/.
7. Документация библиотеки Boost. — URL: https://www.boost.org/.
8. Документация библиотек Google Test и Google Mock. — URL: http://google.github.io/googletest/.
9. Документация системы сборки CMake. — URL: https://cmake.org/.

ПРИЛОЖЕНИЕ

Листинг 7: Публичные методы Curve

```
namespace bezier {

class Curve;

using CurveUptrConst = std::unique_ptr<const Curve>;

class Curve final {
 public:
  Curve(const std::vector<Point> &ps);
  Curve(std::vector<Point> &&ps);

  const std::vector<Point> &get_points() const &noexcept;

  Rectangle BoundingRectangle() const;
  std::pair<std::unique_ptr<Curve>, std::unique_ptr<Curve>> Split(
      const double t) const;
  bool IsIntersect(const Curve &c, const double eps) const;
  std::vector<Point> Intersect(const Curve &c, const double eps) const;
  void Dump(std::ostream &os) const;

  // ... (private section)
};

}  // namespace bezier
```

Листинг 8: Публичные методы Graph (1)

```
namespace graph {

struct KLGrid;
class Graph;

using GraphUptr = std::unique_ptr<Graph>;

class Graph final {
 public:
  Graph(const std::vector<Vertex> &vs);
  Graph(std::vector<Vertex> &&vs);
  Graph(std::vector<VertexSptrConst> &&vs,
        std::vector<EdgeSptrConst> &&es);

  const std::vector<VertexSptrConst> &get_vertices() const &noexcept;
  const std::vector<EdgeSptrConst> &get_edges() const &noexcept;

  static GraphUptr GraphvizFromFile(const std::string &path);
  static GraphUptr GraphvizFromText(const std::string &dot);

  void AddVertex(const Vertex &v);
  void AddVertex(Vertex &&v);

  void AddVertices(const std::vector<Vertex> &vs);
  void AddVertices(std::vector<Vertex> &&vs);

  void AddEdge(const std::size_t start, const std::size_t end,
               EdgeSptrConst e);
  void AddEdges(
      const std::vector<std::pair<std::size_t, std::size_t>> &vertex_pairs,
      const std::vector<EdgeSptrConst>& es);

  void AddSLEdge(const std::size_t start, const std::size_t end);
  void AddSLEdges(
      const std::vector<std::pair<std::size_t, std::size_t>>
      &vertex_pairs);

  bool IsStraightLine() const;

  // ... (public, private sections)
};

}  // namespace graph
```

Листинг 9: Публичные методы Graph (2)

```
namespace graph {

class Graph final {
 public:
  // ... (public section)

  std::vector<EdgeSptrConst> CheckPlanar(const std::size_t k) const;

  std::vector<std::vector<EdgeSptrConst>> CheckQuasiPlanar(
      const std::size_t k) const;

  std::vector<std::vector<EdgeSptrConst>> CheckSkewness(
      const std::size_t k) const;

  std::vector<KLGrid> CheckGridFree(const std::size_t k,
                                    const std::size_t l) const;

  std::vector<std::pair<EdgeSptrConst, EdgeSptrConst>> CheckRAC() const;

  std::vector<std::pair<EdgeSptrConst, EdgeSptrConst>> CheckACE(
      const double alpha) const;

  std::vector<std::pair<EdgeSptrConst, EdgeSptrConst>> CheckACL(
      const double alpha) const;

  // ... (private section)
};

struct KLGrid {
  std::vector<EdgeSptrConst> k_group;
  std::vector<EdgeSptrConst> l_group;

  KLGrid(const std::vector<EdgeSptrConst> &k_group,
         const std::vector<EdgeSptrConst> &l_group)
      : k_group(k_group), l_group(l_group) {}
};

}  // namespace graph
```

Листинг 10: Unit-тест вспомогательной функции `Combinations`

```
namespace utils {

namespace {

TEST(Utils, Combinations) {
  EXPECT_ANY_THROW(Combinations(std::vector<std::size_t>{0}, 2));

  {
    const auto result
        = Combinations(std::vector<std::size_t>{0, 1, 2}, 3);

    ASSERT_EQ(result.size(), 1);

    EXPECT_THAT(result[0], testing::ElementsAre(0, 1, 2));
  }

  {
    const auto result
        = Combinations(std::vector<std::size_t>{0, 1, 2, 3}, 3);

    ASSERT_EQ(result.size(), 4);

    EXPECT_THAT(result[0], testing::ElementsAre(0, 1, 2));
    EXPECT_THAT(result[1], testing::ElementsAre(0, 1, 3));
    EXPECT_THAT(result[2], testing::ElementsAre(0, 2, 3));
    EXPECT_THAT(result[3], testing::ElementsAre(1, 2, 3));
  }
}

}  // namespace

}  // namespace utils
```

Листинг 11: Unit-тест вспомогательной функции `Intersect`

```
namespace utils {

namespace {

TEST(Utils, Intersection) {
  EXPECT_THAT(Intersect<std::size_t>({}, {}), testing::ElementsAre());

  EXPECT_THAT(Intersect<std::size_t>({1, 2}, {}), testing::ElementsAre());

  EXPECT_THAT(Intersect<std::size_t>({1, 2}, {3, 4, 5}),
              testing::ElementsAre());

  EXPECT_THAT(Intersect<std::size_t>({1, 3, 5, 6}, {2, 4}),
              testing::ElementsAre());

  EXPECT_THAT(Intersect<std::size_t>({1, 3, 4, 6}, {2, 4, 5}),
              testing::ElementsAre(4));

  EXPECT_THAT(Intersect<std::size_t>({1, 2, 4}, {2, 3, 4, 5}),
              testing::ElementsAre(2, 4));

  EXPECT_THAT(Intersect<std::size_t>({1, 2, 4, 5, 6}, {2, 3, 4, 5}),
              testing::ElementsAre(2, 4, 5));

  EXPECT_THAT(Intersect<std::size_t>({1, 2, 3}, {1, 2, 3}),
              testing::ElementsAre(1, 2, 3));
}

}  // namespace

}  // namespace utils
```

Листинг 12: Описание K_{10} графа на языке DOT

```
graph {
  a -- b;
  a -- c;
  a -- d;
  a -- e;
  a -- f;
  a -- g;
  a -- h;
  a -- i;
  a -- j;
  b -- c;
  b -- d;
  b -- e;
  b -- f;
  b -- g;
  b -- h;
  b -- i;
  b -- j;
  c -- d;
  c -- e;
  c -- f;
  c -- g;
  c -- h;
  c -- i;
  c -- j;
  d -- e;
  d -- f;
  d -- g;
  d -- h;
  d -- i;
  d -- j;
  e -- f;
  e -- g;
  e -- h;
  e -- i;
  e -- j;
  f -- g;
  f -- h;
  f -- i;
  f -- j;
  g -- h;
  g -- i;
  g -- j;
  h -- i;
  h -- j;
  i -- j;
}
```

Листинг 13: End-to-end тест 4-quasi-planar диаграммы

```
namespace graph {

namespace {

std::function<bool(const std::vector<EdgeSptrConst>&)> RefComp(
    const std::vector<EdgeSptrConst>& es1) {
  return [&es1](const std::vector<EdgeSptrConst>& es2) {
    for (const auto& e : es1) {
      if (std::find_if(es2.begin(), es2.end(), RefComp(e)) == es2.end()) {
        return false;
      }
    }
    return true;
  };
}

TEST(Graph, 4QuasiPlanar) {
  Graph g({{1, 2, "0"},
           {5, 2, "1"},
           {1, 5, "2"},
           {5, 5, "3"},
           {3, 1, "4"},
           {3, 7, "5"},
           {4, 6, "6"},
           {1, 1, "7"}});
  g.AddSLEdges({{0, 1}, {0, 2}, {1, 3}, {2, 3}, {4, 5}, {6, 7}});
  const auto& es = g.get_edges();

  const auto unsat_3qp = g.CheckQuasiPlanar(3);
  ASSERT_EQ(unsat_3qp.size(), 2);

  const std::vector<std::vector<EdgeSptrConst>> expected_unsat_3qp{
      {es[0], es[4], es[5]}, {es[3], es[4], es[5]}};

  for (const auto& es : expected_unsat_3qp) {
    EXPECT_NE(
        std::find_if(unsat_3qp.begin(), unsat_3qp.end(), RefComp(es)),
        unsat_3qp.end());
  }

  const auto unsat_4qp = g.CheckQuasiPlanar(4);
  ASSERT_EQ(unsat_4qp.size(), 0);
}

}  // namespace

}  // namespace graph
```

Листинг 14: End-to-end тест ACEα и ACLα диаграмм

```
namespace graph {

namespace {

std::function<bool(const std::pair<EdgeSptrConst, EdgeSptrConst>&)>
RefComp(
    const std::pair<EdgeSptrConst, EdgeSptrConst>& p1) {
  return [&p1](const std::pair<EdgeSptrConst, EdgeSptrConst>& p2) {
    return &*p1.first == &*p2.first && &*p1.second == &*p2.second ||
           &*p1.first == &*p2.second && &*p1.second == &*p2.first;
  };
}

TEST(Graph, CheckAC) {
  Graph g({{1, 1, "0"}, {3, 1, "1"}, {5, 2, "2"},
           {7, 4, "3"}, {5, 4, "4"}, {3, 4, "5"}});
  g.AddSLEdges({{0, 3}, {1, 5}, {2, 4}, {4, 5}});

  const auto alpha = std::acos(1 / std::sqrt(5));
  auto unsat_ace = g.CheckACE(alpha);
  ASSERT_EQ(unsat_ace.size(), 0);

  g.AddSLEdges({{0, 2}, {1, 4}});
  const auto& es = g.get_edges();
  unsat_ace = g.CheckACE(alpha);
  ASSERT_EQ(unsat_ace.size(), 3);

  const std::vector<std::pair<EdgeSptrConst, EdgeSptrConst>>
  expected_unsat_ace{{es[1], es[4]}, {es[4], es[5]}, {es[0], es[5]}};
  for (const auto& p : expected_unsat_ace) {
    EXPECT_NE(
        std::find_if(unsat_ace.begin(), unsat_ace.end(), RefComp(p)),
        unsat_ace.end());
  }

  const auto unsat_acl = g.CheckACL(alpha);
  ASSERT_EQ(unsat_acl.size(), 2);
  const std::vector<std::pair<EdgeSptrConst, EdgeSptrConst>>
  expected_unsat_acl{{es[4], es[5]}, {es[0], es[5]}};
  for (const auto& p : expected_unsat_acl) {
    EXPECT_NE(
        std::find_if(unsat_ace.begin(), unsat_ace.end(), RefComp(p)),
        unsat_ace.end());
  }
}

}  // namespace

}  // namespace graph
```