Министерство образования и науки РФ
ФГБОУ ВО «Тверской государственный университет»
Факультет прикладной математики и кибернетики
Направление «Прикладная математика и информатика»
Программа магистратуры «Системное программирование»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

____
«Эвристические методы укладки графов на плоскость»

Автор:
Наймушин Алексей Владимирович

Научный руководитель:
к.ф.-м.н.
Карлов Борис Николаевич

Допущен(а) к защите:

Руководитель ООП:

____

(подпись, дата)

Заведующий кафедрой: ______________________
(наименование)

____
(подпись, дата)

Тверь 2018

Оглавление

Введение ... 3
Глава 1 Объект исследования ... 7
1.1 Задача укладки графов на плоскость ... 8
1.2 Минимизация пересечений ребер графа ... 10
1.3 Алгоритмы укладки графов на плоскость ... 16
1.3.1 Метод укладки графа на плоскость с применением физических моделей ... 17
1.3.2 Метод иерархической укладки графа ... 20
1.3.3 Критерий проверки графа на планарность ... 23
Глава 2. Программное средство «Grapher» ... 28
2.1 Эвристики уменьшения числа пересечений ... 28
2.1.1 Эвристики для метода укладки с применением физических моделей ... 28
2.1.2 Эвристики для метода иерархической укладки графов ... 31
2.1.3 Метод равномерного распределения вершин ... 34
2.2 Выбор языка программирования ... 38
2.3 Описание программного средства «Grapher» ... 38
Глава 3 Вычислительные эксперименты ... 46
3.1 Вычислительные эксперименты для метода равномерной укладки ... 46
3.2 Сравнительный анализ ... 47
Заключение ... 50
Список литературы ... 51
Приложение ... 54

Введение

Понятие графа впервые появилось в работе Леонарда Эйлера в 1736 году. В данной работе исследователь доказал отсутствие решения задачи о кенигсбергских мостах. Для решения этой задачи требовалось найти такой маршрут в городе Кенигсберг, чтобы обойти все семь мостов, побывав на каждом из них лишь один раз. Многие ученые заинтересовались подходом Эйлера на применение данного подхода в других различных областях наук: физика, биология, химия, экономика и даже теория вероятностей. Так родилась совершенно новая теория – теория графов. Было замечено, что с использованием графов можно удобно описывать огромное множество различных объектов и различных связей между ними.

Актуальность. Задача визуализации графов затрагивает достаточно обширную область применения в различных сферах науки и жизни. Например, в информатике визуализация может быть применена для разработки различных программ, анализа внутренних бизнес-процессов составляемого приложения, разработка компиляторов для различных языков программирования. В других областях визуализация графов может быть применена к анализу некоторых структур объектов (более часто иерархического вида), при создании интегральных схем, анализе социальных сетей, также применяется в различных навигационных системах.

Обычно принято изображать графы в виде точек, соединенных между собой линиями, однако могут встречаться и другие виды представлений. Например, в блок-схемах вершины графа изображаются в виде блоков различных форм. Первым этапом подхода к решению конкретной задачи визуализации того или иного графа является выбор некоторого эстетического критерия, который будет описывать некоторые параметры отображения графа. Например, для интегральных схем наилучшим эстетическим критерием служит минимизация пересечений графа. После этого, исходя из выбранного эстетического критерия, выбирается сам метод укладки графа.

По сей день популярными являются методы планарной, иерархической и физической укладки графов, о которых будет рассказано в данной работе. Затем граф, для которого решается задача визуализации, передается на вход выбранному алгоритму укладки, который находит координаты для самих вершин и ребер графа на выбранной геометрической поверхности.

В настоящее время проблеме визуализации графов посвящен ряд работ, многие из которых послужили основными источниками при выполнении данной работы. В частности, были выбраны труды таких исследователей как Варфилд Дж., Кофман Е., Сугияма К., Тарьян Р., Фари И., Феррари Д., Харари Р., Хопкрофт Дж., а также других.

В работах данных исследователей рассматриваются различные вопросы, касающиеся как фундаментальных понятий и определений теории графов, числа пересечений ребер, методов укладок графов, так и ряд других научных обобщений, на которых были построена теоретическая и методологическая база знаний, используемых в данной работе.

Цели работы. В ходе выполнения данной работы были поставлены следующие цели:

- Изучение задачи визуализации графов;
- Изучение наиболее популярных методов укладки графов на плоскость;
- Изучение проблемы минимизации пересечений;
- Изучение критериев проверки графов на планарность, в частности, изучения критерия проверки планарности, разработанный Бойером и Мирвольдой;
- Предложение некоторых эвристических методов, направленных на улучшение качества конечного изображения графа на плоскости;
- Реализация программного обеспечения с выбранными методами укладки графов на плоскость;
- Тестирование и отладка созданного программного обеспечения;

- Проведение сравнительного анализа выбранных методов укладки;
- Оформление текста работы.

В качестве выбранных методов укладки графов мною были выбраны методы физической укладки графов, метод Сугиямы, а также метод сдвигов.

При выполнении данной работы, были изучены различные методы укладок графов на плоскость. В частности были изучены такие методы, как укладка с использованием физической модели и метод иерархической укладки Сугиямы. Был изучен метод проверки графов на планарность с использованием метода добавления ребер. Данные методы были запрограммированы на языке программирования C++, с использованием кроссплатформенного фреймворка Qt.

Также в данной работе были проведены вычислительные эксперименты, направленные на изучение качества запрограммированных методов. Для эвристики равномерного распределения вершин графа, речь о которой будет идти во второй главе данной работы, эксперименты показали, что для графов с малым значением плотности время работы составляет от нескольких миллисекунд, до нескольких минут.

Кроме этого, эксперименты также были проведены с целью проведения сравнительного анализа метода укладки с физическими моделями и метода Сугиямы. Сравнительный анализ показал, что с выбранными эвристическими подходами метод Сугиямы работает немного хуже, чем другой метод.

Структура данной работы выглядит следующим образом. В первой главе будет рассмотрен сам объект исследований. Будут введены основные понятия, такие как определение графа, поставлена задача укладки графа на плоскость, определение числа пересечений ребер, а также будут приведены алгоритмы выбранных методов укладки графов на плоскость.

Во второй главе будет приведено описание реализуемого программного средства «Grapher», главной задачей которого является возможность построения изображений для графов, вводимых пользователем. Также в

данной главе будут описаны предлагаемые эвристические методы, направленные на улучшение качества конечных укладок графов.

В третьей главе будет проведен сравнительный анализ выбранных методов укладок графов на плоскость, а также приведены результаты тестирования предложенных эвристических методов.

Глава 1 Объект исследования

Теория графов – раздел дискретной математики, посвященный изучению различных свойств графов. В общем случае граф можно представить в виде некоторого множества вершин, соединенных между собой ребрами. Более формальное определение графа, будет дано ниже в данной работе.

Теория графов находит обширное применение в различных навигационных системах, при визуальном анализе различных диаграмм, а также в информатике, например при анализе бизнес-процессов.

Переформулируем некоторые базовые определения, которые были даны в [3].

Определение 1. *Граф – совокупность двух множеств: множества точек* V*, которые называются вершинами, и множества ребер* E.

Существует два фундаментальных класса графов: ориентированные и неориентированные. Определим эти классы формально.

Определение 2. *Ориентированным графом* G *называют такую пару* $G = (V, E)$*, где* V *– конечное множество, элементы которого обычно называют вершинами графа или его узлами,* E *– множество упорядоченных пар на* V. *Если* $e = (u, v) \in E$*, то говорят, что ребро* ***e*** *ведет из вершины* u *в вершину* v**.** *Это обозначается как* $u \to v$.

Определение 3. *Неориентированным графом* G *называют такую пару* $G = (V, E)$*, где* V *– конечное множество, элементы которого, как и в случае ориентированных графов, называют вершинами или узлами, а* E *– множество неупорядоченных пар на* V**.**

Определение 4. *Порядком графа называют число его вершин, т.е.* $|V|$.

Определение 5. *Путь в графе* $G = (V, E)$ *- это последовательность вершин* $v_0, v_1, \dots, v_n$ *такая, что для каждого* $0 \leq i < n$ *выполнено* $(v_i, v_{i+1}) \in E$.

1.1 Задача укладки графов на плоскость

Область применения укладки графа очень обширна. Укладка графов может применяться в различных областях информатики, таких как разработка и отладка программного обеспечения, исследование поведения внутренних бизнес-процессов ПО, а также взаимосвязь исследуемых процессов друг с другом. Кроме этого графы попросту могут быть использованы в любой сфере науки и жизни, где требуется визуально отобразить какие-либо взаимосвязи между объектами, группами объектов или их свойствами.

Сама проблема визуализации графов заключается в некотором графическом представлении графа на какой-либо геометрической поверхности. Более часто используется частный вид данной задачи – укладка графа на некоторую плоскость. Для решения данной задачи требуется найти координаты для вершин графа на заданной плоскости таким образом, чтобы они минимизировали (или максимизировали) один или несколько эстетических критериев укладки графа. В качестве такого критерия обычно выбирают один из следующих:

- Минимизация пересечений ребер графа;
- Минимизация площади конечного изображения;
- Минимизация изгибов ребер;
- Максимизация симметрии ребер графа (когда отдельно взятые части графа похожи друг на друга);
- Максимизация наименьшего угла между ребрами и др.

Стоит отметить, что хотя можно использовать сразу несколько критериев, в общем случае оптимизировать их все сразу невозможно. Это связанно с тем, что два критерия обычно противоречат друг другу. Например, нельзя минимизировать количество пересечений ребер в графе и одновременно с этим максимизировать симметрию ребер.

Определение 6. *Планарный граф – граф, который можно изобразить на двухмерной плоскости без пересечений каких-либо его ребер.*

Дадим еще несколько ключевых определений, которые используются в данной работе.

Определение 7. *Стягивание смежных вершин u и v графа G означает удаление ребра (u, v) и замена двух вершин u и v одной вершиной, которая соединяется ребрами со всеми вершинами графа G, с которыми были смежны вершины u и v.*

Определение 8. *Подразбиение ребра в графе $G = (V, E)$ — операция, состоящая из замены ребра $e = (u, v) \in E$ и добавления двух новых ребер $e_1 = (u, t)$ и $e_2 = (t, v)$, где t — новая вершина в графе G, имеющая только 2 смежные вершины u и v.*

Определение 9. *Гомоморфизм графа — это такое взаимно-однозначное непрерывное отображение*

$$\varphi : G \to G',$$

что если вершины u и v смежны в G, то вершины $\varphi(u)$ и $\varphi(v)$ смежны в G'.

При решении задачи визуализации графов важным этапом является проверка графа на его планарность. Есть несколько теорем, способных ответить на данный вопрос. Первой из них является известная теорема Куратовского, из которой следует, что граф планарен тогда и только тогда, когда он не содержит подграфов, гомеоморфных полному графу из пяти вершин или графу «домики и колодцы». Данные графы можно видеть на рисунке 1.

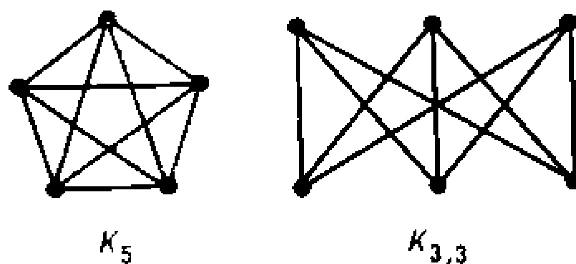

Рисунок 1 Графы Куратовского

Другим критерием проверки на планарность является теорема, доказанная Вагнером. Согласно данной теореме, граф является планарным тогда и только тогда, когда он не содержит подграфа, стягиваемого к полному графу из пяти вершин или графу «домики и колодцы». Примером такого графа является граф Петерсена, изображенный на рисунке 2. Данный граф после стягивания ребер r_1, r_2, r_3, r_4 и r_5 приводится к полному графу из пяти вершин.

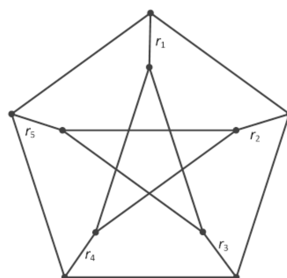

Рисунок 2 Граф Петерсена

1.2 Минимизация пересечений ребер графа

Проблема минимизации количества пересечений является комбинаторной проблемой, которая состоит в том, чтобы выбрать упорядочение вершин, с наименьшим количеством пересечений. Стоит отметить, что данная задача является NP-полной даже в случае поуровневой укладки двудольного графа. Двудольным графом называют такой граф $G = (V, E)$, для которого множество его вершин можно разбить на две части $W \cup T = V$, так, что ни одна вершина в W не соединена с другой вершиной из W, и ни одна вершина V также не соединена с вершинами в V. Более того, задача по-прежнему остается NP-полной, если точно известно такое расположение узлов на одной из долей, которое привидет к наименьшему количеству пересечений.

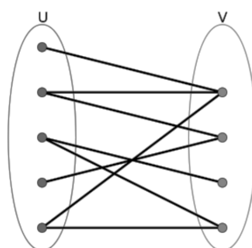

Рисунок 3 Пример двудольного графа

Подходы к решению задачи минимизации пересечений могут сильно различаться в зависимости от используемой укладки графа. Например, для поуровневой укладки графа на практике очень часто применяют эвристический метод, называемый методом барицентров. Задача поуровневой укладки графа состоит в следующем. Задан ациклический граф $G = (V, E)$. Требуется разбить множество его вершин V на несколько подмножеств $L_1, L_2, \dots, L_r$, что $L_i \cap L_j = \emptyset$, при $i \neq j$, так чтобы для любого ребра $(u, v) \in E$ выполнялось следующее условие: если $u \in L_i$, а $v \in L_j$, то $i < j$. Множества L_i называются уровнями. Также такую укладку иногда называют иерархической укладкой. В данной работе будут использованы оба термина.

Опишем метод барицентров. Данный метод был предложен Сугиямой в 1981 году. Сформулируем его. Пусть имеется некий граф G, для которого был применен один из методов поуровневой укладки. Будем обходить этот граф, начиная с верхнего слоя, просматривая сразу пару слоев. На каждом этапе алгоритма, будем фиксировать положения вершин верхнего слоя. Затем для каждой вершины нижнего слоя, найдем её новую позицию как среднее арифметическое координат соседей данной вершины из верхнего слоя:

$$avg(u) = \frac{1}{\deg(u)} \sum_{v \in N(u)} x(v),$$

где $\deg(u)$ – степень вершины, $N(u) = \{V | (u, v) \in E\}$ – множество вершин, для которых есть ребро из u, а $x(v)$ – абсцисса вершины v. Если же после определения координаты двух вершин совпадают, то к координате одной из них прибавляют небольшое значение, чтобы отдалить вершины друг от друга.

Для метода барицентров существует очень полезная теорема о том, что если для графа G возможно размещение вершин без пересечений какой-либо пары ребер, то метод его обязательно найдет.

Из недостатков данной эвристики можно назвать тот факт, что она предназначена только для иерархической укладки, поэтому использовать данный подход в любых других укладках невозможно.

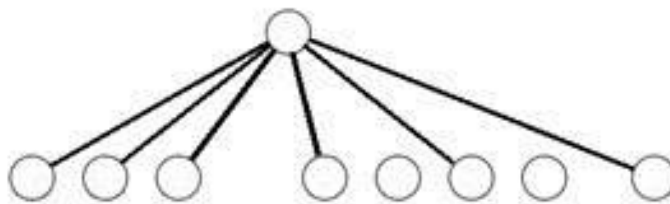

Рисунок 4 Иллюстрация метода барицентров

Кроме обычного метода барицентров также известны несколько его модификаций. Наиболее популярным из них является метод медиан. В отличие от метода барицентров каждой вершине с четной степенью присваивается среднее арифметическое значение позиций вершин соседнего уровня, а для вершин с нечетными степенями – среднее значение координат соседей.

Изучая проблему минимизации пересечений укладки графа, нельзя не затронуть такое понятие как «число пересечений».

Отправной точкой к изучению задач минимизации пересечений стала задача о кирпичной фабрике, поставленная венгерским математиком Палом Тураном еще в 1945 году. Во время Второй мировой войны математику приходилось работать на кирпичной фабрике, толкая телегу груженную кирпичами. Всякий раз телегу было толкать тяжелее на месте пересечения колей. Это и привило Турана к постановке задачи о нахождении минимального числа пересечений в рисунке графа.

Определим понятие числа пересечений графа. Числом пересечений графа G называют минимально возможное число пересечений ребер графа любой укладки. Обозначается это число как $cr(G)$.

Стоит отметить, что понятие числа пересечений является одной из характеристик степени непланарности графа.

Как было показано Гери и Джонсоном в 1983 году, проблема нахождения числа пересечений графа является NP-полной проблемой. Исследователи смогли доказать этот факт, сведя задачу об оптимальном линейном размещении графа, которая, как ими ранее было доказано, является NP-полной, к данной задаче. Сформулируем задачу оптимального линейного размещения. На вход подается граф $G = (V, E), |V| = n,$ и некоторое число $a > 0$. Вопрос заключается в следующем: можно ли построить такую биекцию $\alpha: V \rightarrow \{1, \dots, n\}$ (линейное расположение узлов), для которой выполнено:

$$\sum_{(u,v)\in E} |\alpha(u) - \alpha(v)| \leq a\,?$$

Помимо обобщенного понятия числа пересечений в литературе очень часто встречаются и вариации определений для различных укладок. Одним из наиболее часто встречающихся определений является прямолинейное число пересечений. Данное определение полностью идентично определению обычного числа пересечений, однако на него накладывается дополнительное требование в виде использования только укладок с прямолинейными ребрами. Прямолинейное значение пересечений обозначается как $\overline{cr}(G)$.

Можно сформулировать несколько полезных теорем, которые позволяют установить связь между обобщенным числом и прямолинейным. Одной из таких теорем является теорема Бьенстока и Дина. Она гласит, что если $cr(G) \leq 3$, то $\overline{cr}(G) = cr(G)$. Также ими была доказана и другая теорема, которая гласит, что для любого числа $m \geq 4$ существует такой граф G для которого $cr(G) = 4$, однако $\overline{cr}(G) \geq m$.

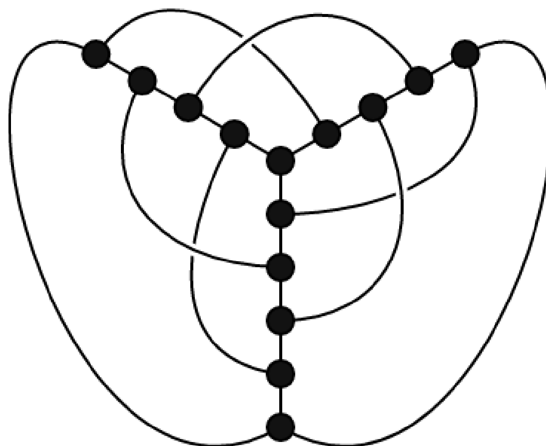

Рисунок 5 Граф Хивуда, для которого $cr(G) = 3$

Другим распространенным определением числа пересечений является линейное число пересечений. Пусть дан некий граф $G = (V, E)$. Линейным числом пересечений называют наименьшее число пересечений среди всевозможных линейных укладок графа G вдоль прямой линии. Обычно в литературе такое число обозначается как $\mu(G)$.

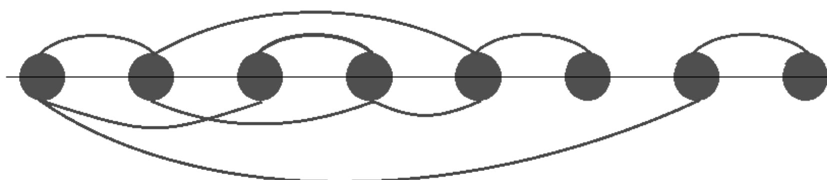

Рисунок 6 Пример линейной укладки графа на горизонтальную линию

Как и в предыдущих случаях с другими вариациями определений, проблема проверки того, что $\mu(G)$ не превосходит некоторое число k также является NP-полной проблемой.

На практике, для того чтобы найти число пересечений какого-нибудь графа, очень часто прибегают к различным эвристикам. И хотя они не позволяют найти точное значение, однако они позволяют за разумное время получить неплохое его приближение. Методика одной из эвристик заключается в следующем. Пусть дан некий граф $G = (V, E)$. Из данного графа удаляют некоторое подмножество ребер $F \subset E$ так, чтобы новый граф $G' = (V, E')$, $E' = E \setminus F$ являлся планарным. Затем из множества F выбирают ребро f и пытаются найти укладку графа $G'' = (V, E' \cup \{f\})$ с наименьшим

числом пересечений (обычно сводя данную задачу, к задаче о нахождении кратчайшего пути). Далее эта процедура повторяется до тех пор, пока множество непусто.

Используя понятие числа пересечений для задачи минимизации пересечений в графе можно сформулировать задачу о числе пересечений. Пусть на вход подается некий граф G и число $k > 0$. Задача заключается в ответе на вопрос: можно ли найти такую укладку графа, при которой $cr(G) \leq k$?

Многими исследователями были предприняты многочисленные попытки оценить значения границ числа пересечений графа. Одной из известных теорем в теории графов является теорема польского математика Казимежа Заранкевича (см. [23]). Занимаясь изучением проблемы о кирпичной фабрике, он смог найти выражение, которое позволяло точно определить значение числа для любого двудольного графа. К сожалению, в его доказательстве нижней границы была найдена ошибка, однако он смог точно найти верхнюю границу числа пересечений для произвольного полного двудольного графа:

$$cr(G) \leq \left\lfloor\frac{n}{2}\right\rfloor \left\lfloor\frac{n-1}{2}\right\rfloor \left\lfloor\frac{m}{2}\right\rfloor \left\lfloor\frac{m-1}{2}\right\rfloor.$$

Немного позднее, Энтони Хилл и Джон Эрнест опубликовали свое решение в [9]. Исследователи смогли не только сформулировать саму задачу, но также высказали свою гипотезу о том, что верхняя граница числа пересечений графа равняется:

$$cr(G) \leq \frac{1}{4}\left\lfloor\frac{n}{2}\right\rfloor \left\lfloor\frac{n-1}{2}\right\rfloor \left\lfloor\frac{n-2}{2}\right\rfloor \left\lfloor\frac{n-3}{2}\right\rfloor.$$

К 2018 году, значения чисел пересечений известны лишь для небольшого числа семейств графов. Были предприняты многочисленные попытки нахождения нижних границ, однако точного результата найдено не было.

1.3 Алгоритмы укладки графов на плоскость

В виду широкой области применения графов, существует множество различных видов укладки. Обычно выделяют следующие, наиболее применяемые виды укладок:

- Прямолинейная укладка – ребра графа представляют собой отрезки;

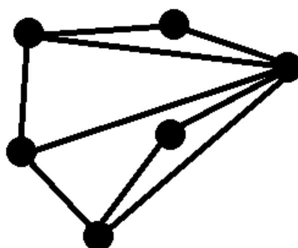

Рисунок 7 Пример прямолинейной укладки

- Укладка с использованием ломаных линий (или полигональная укладка) – аналогично прямолинейной укладке, однако ребра графа могут иметь изломы;

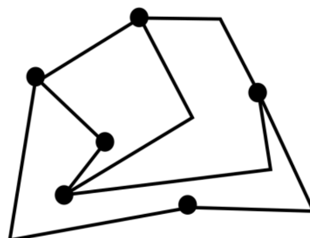

Рисунок 8 Пример полигональной укладки

- Ортогональная укладка, где звенья ребер графа представлены только горизонтальными и вертикальными ломаными линиями;

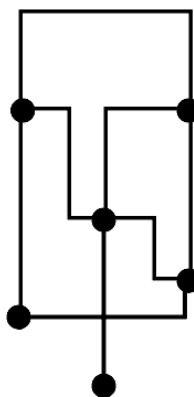

Рисунок 9 Пример ортогональной укладки

- Планарная укладка, ребра которой могут быть представлены любыми способами, однако ни одно из ребер графа не пересекает другое.

Ниже приведены визуальные примеры каждой из этих укладок:

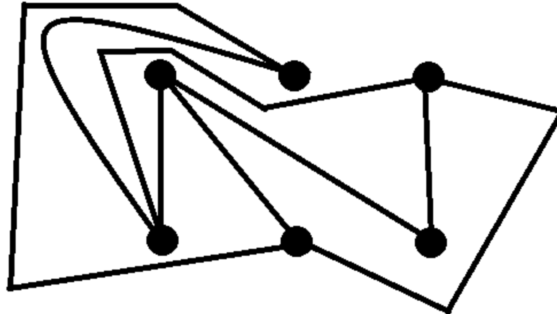

Рисунок 10 Пример планарной укладки

Наиболее популярными методами укладок графов остаются, иерархические методы укладки графов. Благодаря данным методам можно с легкостью проследить иерархию взаимоотношений между объектами или их свойствами. Частным примером применения таких методов является укладка деревьев. Рассмотрим данные методов поподробнее

1.3.1 Метод укладки графа на плоскость с применением физических моделей

Одними из наиболее применяемых методов укладок графов на плоскость являются методы с применением физических моделей.

Данные методы появились в работах Тутте в 1963 году, который показал, что графы с большим количеством узлов могут быть хорошо уложены на плоскость с применением физических моделей, добавив каждому ребру различные силы и позволив системе самой достигнуть состояния равнения.

Смысл данных методов заключается в том, что для каждого ребра или узла задаются некоторые силы, обычно согласно закону Гука, т.е. $\boldsymbol{F} = \boldsymbol{k}\Delta\boldsymbol{l}$, где $\boldsymbol{F}$ – сила упругости, $\boldsymbol{k}$ – жесткость тела, а $\Delta\boldsymbol{l}$ – деформация тела. Затем данный алгоритм пытается минимизировать энергию полученной системы для получения самой укладки.

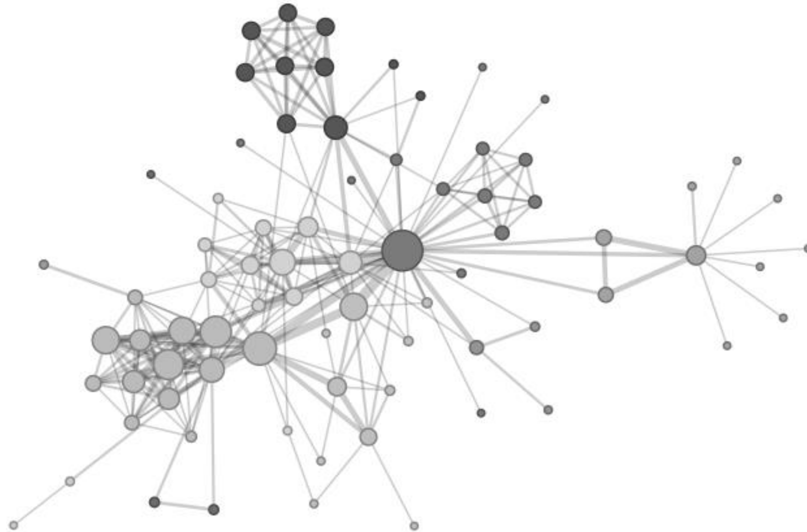

Рисунок 11 Укладка с применением физической модели

Опишем этапы алгоритма укладки с применением физической модели. Пусть на вход подается некий граф $G = (V, E)$.

1. **Инициализация**. Каждому узлу графа задается некоторая случайная точка на координатной плоскости.

2. **Итерации алгоритма**. Для каждого узла имеем векторную переменную *displ*, которая характеризирует смещение узла на данной итерации. Перед началом каждой итерации, данная переменная для всех узлов равна 0.

2.1. **Вычисление отталкивающих сил**. Для каждой пары узлов, $u, v \in V$ $(u \neq v)$, вычисляем значение смещения по следующим формулам:

$$\delta = v.pos - u.pos,$$

$$v.displ = v.displ + \left(\frac{\delta}{|\delta|}\right) f_r(|\delta|),$$

где $f_r(x) = \frac{x^2}{\sqrt{\frac{area}{|V|}}}$, $area = WL,\ W \geq 0\ , L \geq 0$ – некоторые переменные, ограничивающие площадь, в которой будет размещен граф, */x/* - оператор нормы векторной величины *x*.

2.2. **Вычисление притягивающих сил**. Для каждого ребра $e = (u, v) \in E$, произведем корректировку смещений узлов u и v по следующим правилам:

$$\delta = e.v.pos - e.u.pos,$$

$$e.v.displ = e.v.displ - \left(\frac{\delta}{|\delta|}\right) f_a(|\delta||,$$

$$e.u.displ = e.u.displ + \left(\frac{\delta}{|\delta|}\right) f_a(|\delta||,$$

где $f_a(x) = \frac{\frac{area}{|V|}}{x}$. *u.pos* – векторная переменная, определяющая позицию узла *u* на плоскости.

2.3. **Корректировка позиции узлов**. Для каждого узла $u \in V$ произведем коррекцию позиции по следующему правилу:

$$u.pos = u.pos + \frac{u.displ}{|u.displ|}.$$

2.4. Критерий останова. Если среднеквадратичное значение всех смещений достигло некоторого порогового значения, то выходим из алгоритма, иначе переходим на пункт 2.

Преимущества укладки с использованием физических моделей заключаются в том, что они способны оптимизировать сразу несколько эстетических критериев, таких как максимизация симметрии или равномерное распределение вершин. Также данные методы привлекательны своей гибкостью. Можно с легкостью применять различные эвристики, улучшающие качество укладок при этом, не изменяя самого алгоритма метода.

К недостаткам данных методов, можно отнести то, что они имеют достаточно высокое время работы: $O(I * n^3)$, где I – число итераций, n – количество вершин в графе. Однако на практике, методы данного класса сходятся достаточно быстро.

Главным недостатком данных методов является проблема локального минимума. Это объясняется тем, что данные методы стремятся уменьшить общую энергию системы и очень часто найденный минимум не является глобальным минимумом функционала энергии. Это приводит к ухудшению качества **укладки.**

1.3.2 Метод иерархической укладки графа

На практике очень часто изображения, полученные с использованием физических моделей, не всегда являются наилучшем способом отражения информации. Например, данные алгоритмы не могут явным образом отобразить иерархическую составляющую тех или иных связей, что в следствии может привести к затруднением в анализе некоторой системы, для которой и было получено изображение. В таких случаях следует использовать алгоритмы иерархической укладки графов на плоскость.

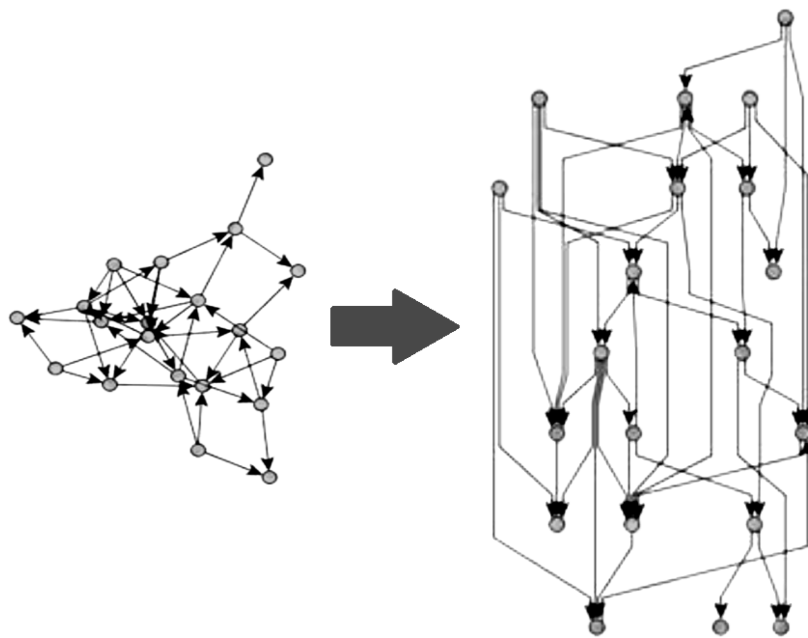

Рисунок 12 Сравнение различных методов укладки графа

Одним из наиболее популярных методов послойной укладки графа на плоскость является известный метод – метод Сугиямы, названный в честь исследователя Козо Сугиямы (Kozo Sugiyama) предложившего эту методику в 1981 году. В своих работах исследователь очень часто использовал данный подход к отображению графов сетей.

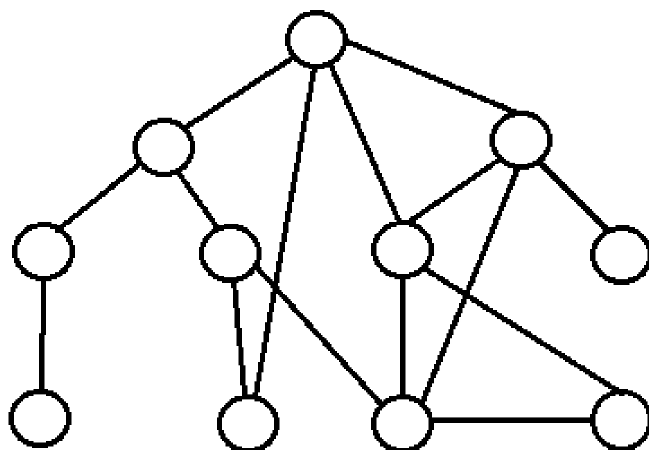

Рисунок 13 Пример послойной укладки графа

Основным преимуществом данного метода является то, что свойство иерархичности вершин очень легко проследить. Также следует отметить быстроту работы данных методов и возможность обобщения методики для случая неориентированных графов.

Из недостатков можно назвать то, что обычно алгоритмы данного класса очень трудно реализовать на практике и что данный метод очень трудно обобщить для укладки в трехмерном пространстве. Еще одним недостатком алгоритма является тот факт, что поданный на вход алгоритма граф G не должен иметь циклов (т.е. таких путей в графе, которые начинаются и заканчиваются в одной вершине). Это может привести к дополнительным затратам на вычисление. Для удаления циклов существует несколько эвристик, о которых будет сказано позднее.

Приведем описание этапов алгоритма. Пусть требуется уложить на плоскость некий ориентированный граф $G = (V, E)$. Для этого требуется выполнить следующие этапы:

1. **Проверка графа на отсутствие циклов**. Если циклы присутствуют, то их количество следует уменьшить или убрать совсем.
2. **Распределение вершин графа по слоям** таким образом, чтобы все дуги графа имели одно направление (обычно сверху-вниз).

3. **Нахождение оптимального порядка вершин для каждого слоя**, минимизирующее общее число пересечений дуг графа.
4. **Выбор координат для каждой вершины** таким образом, чтобы они отражали порядок вершин, полученный из третьего пункта алгоритма.

Стоит отметить, что наибольшую сложность в реализации алгоритма составляют первый и третий пункты алгоритма. Доказано, что нахождение такого набора ребер, который удалил из графа все циклы, а также задача на нахождение оптимального порядка узлов являются NP-полными задачами.

Задачу на нахождение набора дуг, разрезающего циклы, можно сформулировать следующим образом: пусть на вход подается ориентированный граф $G = (V, E)$. Требуется найти подмножество дуг $F \subset E$ такое, что граф $G' = (V, E \setminus F)$ не содержит циклов. Было показано, что к данной задаче можно свести задачу о вершинном покрытии, которая, как известно, является NP-полной. Этот было доказано в [12].

Сформулируем задачу о нахождении оптимального порядка вершин в слоях графа. На вход подается некий граф $G = (V, E)$, а для каждой его вершины v из V известен ее номер слоя в укладке. Требуется найти такие последовательности вершин для каждого слоя, чтобы общее число пересечений дуг в графе было минимальное.

Одно из ключевых преимуществ метода Сугиямы — это время его работы. Если принимать во внимание тот факт, что в графе $G = (V, E)$ полностью отсутствуют циклы, то оно составляет примерно $O(|V||E| \log|E|)$, однако если циклы в графах все-таки присутствуют, то суммарная сложность алгоритма стремительно возрастает до экспоненциальной.

1.3.3 Критерий проверки графа на планарность

Напомним, что планарным граф является тогда и только тогда, когда его можно изобразить на двухмерной плоскости без пересечений каких-либо его ребер. Такое изображение графа называется планарной укладкой графа на плоскость.

Задача планарной укладки является менее строгой вариацией задачи о вложении графа. Вложением графа можно называть такое представление заданного графа G на заданной поверхности σ, в котором некоторым точкам из σ сопоставлены вершины и ребра графа G следующим образом:

1. Конечным точкам ребра $e \in E$ на поверхности σ сопоставлены точно такие же значения, которые были сопоставлены вершинам ребра e.
2. Никакое ребро не содержит каких-либо других точек, сопоставленных остальным вершинам графа G,
3. Никакая пара ребер из E не пересекается во внутренних точках этих ребер.

Говоря другими словами, вложение графа в некоторую поверхность σ является укладкой графа на эту поверхность таким образом, что его ребра могу пересекаться только в конечных точках. Частным примером является то, что любой планарный граф может быть вложен в евклидово пространство R^2.

Другими словами, задачи укладки графа и вложения графа отличаются лишь тем, что решение задачи вложения графа не может содержать никаких пересечений у ребер, когда задача укладки не ограничивается данным условием.

Планарное вложение графа можно рассматривать как более удобную структуру представления его планарного изображения, поэтому большинство алгоритмов планарной укладки являются функциями отображения этой самой структуры на заданную плоскость σ.

Очевидно, что для получения корректного ответа для задачи вложения графа в двухмерное евклидово пространство R^2, этот граф должен быть планарным. Поэтому одним из важных этапов каждого алгоритма планарной укладки является проверка выполнения свойства планарности заданного графа *G*.

Существует несколько алгоритмов проверки графов на планарность, работающих за линейное время. Первый алгоритм для проверки планарности за линейное время был разработан Хопкрофтом и Тарьяном в 1974 году (см. [10]). Другим из наиболее известных методов является метод, разработанный в 2004 году исследователями Бойером и Мирвольдой. Данный алгоритм возвращает значение true, если переданный ему на вход граф является планарным или false, если в нем обнаруживается один из подграфов изображенных на рисунке 1. Кроме этого, если заданный граф является планарным, то алгоритм также может вернуть его вложение в евклидово пространство R^2. Если же граф не является планарным, то данный алгоритм сможет указать конкретный подграф, нарушающий теорему Куратовского. Еще одним преимуществом данного алгоритма является его время работы. В своей работе Бойер и Мирвольд доказали, что данный алгоритм способен проверить граф на планарность за линейное время.

Для того чтобы можно было понять суть данного алгоритма, определим некоторые термины, которые будем в дальнейшем использовать.

Цепь в графе G – путь в графе G, все ребра которого различны.

Компонентой связности графа *G = (V, E)* будем называть подграф графа *G*, порождаемый таким множеством вершин $V' \subseteq V$, что:

1. для любой пары вершин $u, v \in V'$ в графе G существует (u, v) −цепь;

2. для любой пары вершин $u, v \in V''$, $w \notin V'$ не существует (u, w) −цепи.

Данный алгоритм основывается на добавлении ребер в структуру вложения G' таким образом, чтобы поддерживать набор планарных

двусвязных компонент. Это напрямую связано с теоремой о том, что если все двусвязные компоненты в графе G являются планарными, то сам граф также является планарным. Также отметим, что при каждом вложении ребра в структуру G', может случиться так, что две (или более) промежуточные двусвязные компоненты, также находящиеся в структуре G', объединятся в одну двусвязную компоненту большего размера.

Как уже было сказано ранее, весь алгоритм основан на операции добавлении очередного ребра в структуру вложения графа G' таким образом, чтобы поддерживать в этой множество промежуточных двусвязных компонент исходного графа, которые были вычислены на предыдущих этапах алгоритма.

Перейдем к более детальному описанию алгоритма.

Пусть на вход подаем некий граф $G = (V, E)$. Результатами работы алгоритма будут булева переменная и сама структура вложения G'. Если граф не является планарным, то в булевой переменной будет находиться 0, а в структуре G' – подграф Куратовского. В противном случае, булева переменная будет равна 1.

На первом этапе алгоритма требуется построить остовое дерево для G. Это можно сделать, воспользовавшись методом обхода графа в глубину. Во время обхода графа также следует пронумеровать вершины, согласно их порядку обхода. Для удобства, будем называть данный номер DFI (от англ. Depth First Index). Во время этого будет получена некая последовательность вершин U. После этого для каждой вершины будет произведено вычисление ее низшей точки (lowpoint). Низшей точкой вершины v является ее потомок в дереве с наименьшим DFI, до которого можно дойти из v. Затем для каждой вершины создадим список ее дочерних вершин, который будет отсортирован по значениям низших точек.

На следующем этапе алгоритма, мы будем просматривать все вершины последовательности U, начиная с конца. На каждом шаге, мы будем пытаться

вложить вершину и ее детей в специальную структуру G'. Затем, для каждого ребра, ведущего непосредственно к вершине с наименьшим значением DFI, мы будем вызвать специальную функцию walkup. Данная функция нужна для нахождения в графе двусвязных компонент. После выполнения данной функции, для каждой дочерней вершины будет вызвана еще одна специальная функция walkdown, которая будет пытаться корректно встраивать ребро, ведущее непосредственно к низшей точке вершины v. В своей работе Бойер и Мирвольд доказали, что данная функция не сможет этого сделать лишь в единственном случае – когда граф не является планарным. После выполнения структура G' будет проверена на то, что ребро было успешно добавлено. Если этого не произошло, то будет вызван метод isolateKuratowskiSubgraph, хорошо описанный в самой работе Бойера и Мирвольды. Данный метод попросту находит в графе соответствующий подграф Куратовского, приводящий к нарушению свойства планарности заданного графа. После выполнения данной процедуры мы выходим из алгоритма проверки графа на планарность, предварительно разместив в булевой переменной 0.

Если все вершины были успешно просмотрены, то, как и было сказано ранее, булеву переменную делаем равной 1 и выходим из алгоритма проверки.

Теперь приведем формальный алгоритм данной процедуры:

Алгоритм 1. Проверка планарности.

Процедура: **ПЛАНАРНОСТЬ**
Вход: Граф $G = (V, E)$ с $|V| \geq 2$ вершин и $|E| \leq 3|V| - 5$ ребер.
Выход: 1, если граф планарный и G' – некая структура для вложения графа, или 0, если граф не является планарным и подграф Куратовского в G'
(1) Выполняем обход графа в глубину, формируя список вершин U
(2) Находим низшие точки для всех вершин
(3) Для каждой вершины создаем списки ее дочерних, отсортированных по значению низших точек
(4) Для каждой вершины $v \in U$, начиная от $n - 1$ до 0
(5) Для каждой вершины c, которая является дочерней для v
(6) Добавляем ребро в (v, c) в структуру G'

(7) Для каждого ребра $e = (v, u)$, u – вершина, являющаяся низшей точкой для v

(8) ***walkup***$(\boldsymbol{G'}, \boldsymbol{v}, \boldsymbol{u})$

(9) Для каждой дочерней вершины c

(10) ***walkdown***$(\boldsymbol{G'}, \boldsymbol{v}, \boldsymbol{c})$

(11) Для каждого ребра $e \in E$, которое приводит к низшим точкам

(12) Если $(u, w) \notin G'$

(13) ***isolateKuratowskiSubgraph***$(\boldsymbol{G'}, \boldsymbol{G}, \boldsymbol{v})$

(14) Вернуть $(0, G')$

(15) Вернуть (1, G')

Глава 2. Программное средство «Grapher»

2.1 Эвристики уменьшения числа пересечений

В ходе выполнения данной работы, одной из поставленных задач являлось разработка некоторых эвристических методов, способствующих улучшению качеству конечного изображения графа, уложенного с помощью выбранных методов укладки графов. В данной главе будут описаны некоторые возможные эвристики.

2.1.1 Эвристики для метода укладки с применением физических моделей

Проблема локального минимума представляет собой одну из важнейших проблем, с которой приходится сталкиваться при использовании укладки графа с помощью физических моделей. Чтобы попытаться улучшить качество укладки графа, в данной работе были рассмотрены несколько возможных эвристик, которые были реализованы в программе. Все рисунки данного пункта были выполнены с использованием данной программы.

Идея первой эвристики была позаимствована из другой математической задачи, в которой также встречается проблема нахождения локального минимума. Этой задачей является обучение нейронной сети методом обратного распространения ошибки.

Для корректного обучения нейронных сетей в конце итерации обучения вычисляется значение среднеквадратичной ошибки, которая была допущена при распознании образов из множества обучающих примеров. Обучение нейронной сети следует остановить в том случае, когда изменение этой самой ошибки между итерациями становится меньше некоторого порогового значения. К сожалению, поверхность функционала ошибки содержит в себе очень много точек экстремума, в частности, различных локальных минимумов. Как только нейронная сеть достигает одной из таких точек, разница в изменении значения ошибки резко падает, что приводит к

тому, что обучение сети заканчивается слишком рано и, соответственно, данная сеть не сможет выдавать корректный отклик.

Для решения проблемы стабилизации функционала ошибки в локальных минимумах, одним из простейших способов является «встряска» весов (от англ. jog of weights). Идея данного метода заключается в том, что при каждой стабилизации значения ошибки, следует производить случайные модификации всех весовых коэффициентов в достаточно большой окрестности текущего значения. Это приведет к тому, что метод градиентного спуска начнет поиск минимального значения из других точек и, возможно, сможет обойти «ловушку» локального минимума, в которую он попал в прошлый раз.

Идея данного подхода была рассмотрена и в задаче нахождения укладки графа на поверхность. На каждом этапе корректировки позиции узлов графа будем также сдвигать узлы на небольшое расстояние в случайном направлении. Это позволит алгоритму укладки начать поиск оптимального расположения узлов из новых точек и, возможно, избежать «ловушки» локального минимума.

К сожалению, на практике данный подход оказал слабое влияние на качество укладки заданного графа. Лишь в нескольких случаях эвристика помогла уменьшить количество пересечений ребер конечной укладки. Кроме этого, для сходимости алгоритма потребовалось гораздо больше времени, чем для обычного алгоритма укладки.

Другая рассмотренная эвристика направлена на изменение представления ребер графа не в виде прямых линий, а в виде дуг. Смысл данной эвристики заключается в следующем. Пусть имеется некий граф $G = (E, V)$, а также пара пересекающихся ребер (u, v) и (w, t), где $u, v, w, t \in V$. Тогда выберем одно из ребер, например (u, v), разобьем его на два новых ребра (u, v') и (v', v), добавив в граф новый фиктивный узел v'. Затем, запустим процесс нахождения укладки для нового графа $G' = (V \cup \{v'\},\ E)$.

После того, как укладка нового графа будет найдена, заменим ребра графа сплайнами Безье, используя добавленные фиктивные вершины как точки характеристического многоугольника для данного ребра. Пример работы программы на графе «Домики и колодцы» приведён ниже.

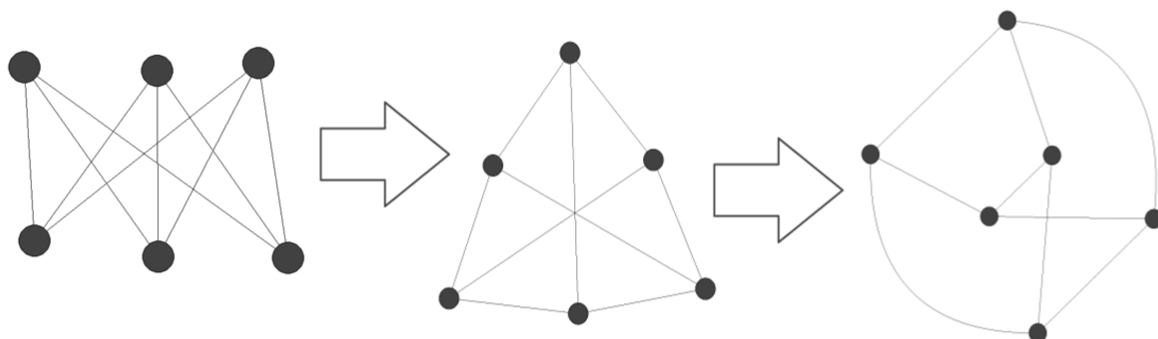

Рисунок 14 Пример конечного результата эвристики на каждом этапе

К недостаткам данного подхода можно отнести то, что для сходимости алгоритма укладки потребуется гораздо больше времени, так как граф практически полностью меняет расположение своих узлов.

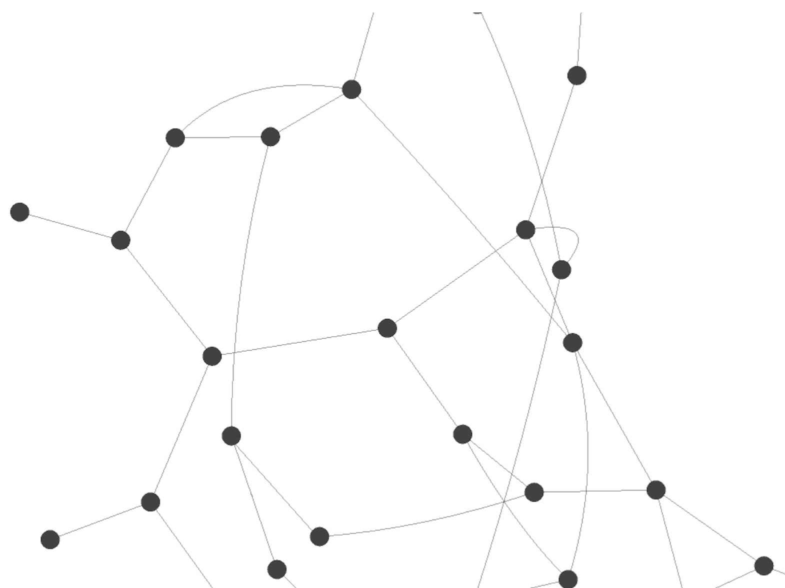

Рисунок 15 Пример неудачной укладки для некоторой части произвольного графа

Кроме этого, частое использование данной эвристики приводит к резкому возрастанию длины ребер, что также может негативно сказываться на качестве конечной укладки.

2.1.2 Эвристики для метода иерархической укладки графов

Для того чтобы граф можно было отобразить с использованием метода Сугиямы, он должен быть ацикличным (не содержать циклов). Также было сказано, что задача нахождения таких дуг, образующих циклы является NP-полной, что значительно увеличивает общее время работы алгоритма. Однако известно несколько эвристик, способные устранить эту проблему за разумное время.

Простейшей эвристикой в данном случае является использование алгоритма обхода графа в глубину. После того как на некотором шаге обхода графа выясняется, что данная дуга приводит к зацикливанию в графе, то она удаляется.

Приведем неформальное описание алгоритма обхода графа в глубину.

Пусть на вход подается некий граф $G = (V, E)$. Требуется найти и удалить из множества дуг E такие дуги, которые позволят разорвать циклические пути в графе. Тогда:

1. **Инициализация**. Помечаем все вершины в графе как новые.
2. **Для всех вершин v из множества V выполняем следующее**: если вершина помечена как новая, то вызываем для нее вспомогательную процедуру DFS.

Опишем процедуру *DFS:*

1. **Помечаем вершину** *u* как старую.
2. **Для всякой вершины, смежной с *u*** выполняем следующее: если текущая смежная вершина v помечена как старая, то $E = E \setminus \{(u, v)\}$. В противном случае рекурсивно вызываем для вершины *v* процедуру DFS.

Время работы данного алгоритма составляет $O(|V| + |E|)$.

Другая эвристика представляет собой вариацию первой, однако после того как мы приходим в уже ранее посещенную вершину, то соответствующее ребро графа следует развернуть в противоположном

направлении. Например, если обнаруживается, что дуга $(u, v) \in E$ образует цикл, то она заменяется другой дугой (v, u). Однако данный метод приводит к тому, что некоторые дуги графа будут идти «против течения». Это может сказаться на читаемости конечного изображения графа.

Перейдем к методам распределения вершин по слоям. В общем случае, выбор данной эвристики зависит от свойств рассматриваемой предметной области, для которой и требуется получение изображения графа. Если же распределение вершин по слоям не играет серьезной роли, то можно использовать следующие методы:

- Распределить вершины по уровням случайным образом. Однако это не гарантирует хорошего конечного результата.
- Использовать ранее упомянутые алгоритмы обхода графов. В качестве вершин, которые будут размещены на первом слое, следует выбрать вершины, в которые не входят дуги.

В качестве одного из наиболее популярных методов определения порядка вершин на уровнях, можно использовать уже ранее описанный метод барицентров.

В большинстве случаев после выполнения этапа нахождения порядка вершин в слоях, остаются некоторые пересечения, от которых можно легко избавиться «на глаз». Данный подход был разработан с целью уменьшения таких пересечений в конечном изображении.

Идея данной эвристики заключается в том, чтобы для каждой пары пересекающихся дуг попробовать поменять местами позиции их родителей.

Опишем этапы алгоритм данной эвристики. Для удобства положим, что все ребра в графе пронумерованы как $e_1, e_2, \dots e_n$.

Алгоритм 2. Эвристика перестановок

Процедура: switchHeuristic

(1) **for** ***i = 1*** **to** ***n do***:

(2) **for** ***j = i + 1*** **to** ***n do***:

(3) Если ребра e_i и e_j пересекаются, то поменяем местами координаты вершин родителей данных ребер.

(4) Если общее число пересечений в графе увеличилось, то возвращаем вершины на исходные позиции.

Можно заметить, что теоретическое время работы такой эвристики составляет $O(n^2)$.

Приведем пример работы данной эвристики. Пусть имеется некоторый граф, к которому был применен метод укладки Сугиямы без сортировки вершин на уровнях.

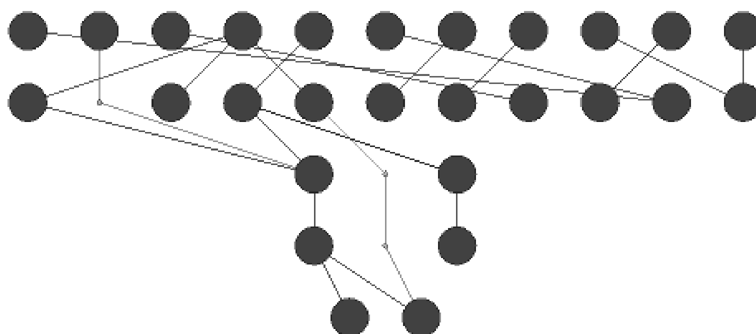

Рисунок 16 Пример изображения графа без использования эвристики

Можно посчитать, что данное изображение графа содержит 21 пересечение дуг. Применим наш эвристический метод:

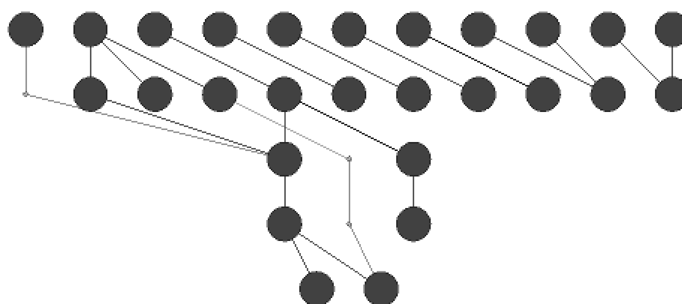

Рисунок 17 Пример работы эвристики

Как можно заметить, общее число пересечений уменьшилось до одного.

2.1.3 Метод равномерного распределения вершин

Для задачи распределения вершин по уровням было разработано множество различных методов. Примеры таких алгоритмов можно найти [4, 16]. Один из простейших алгоритмов помещает каждую вершину на уровень с номером равным её глубине в методе обхода графа в глубину (см. [1]). Преимущества данного подхода в том, что этот метод выполняется за линейное время. Недостатком является то, что он может получить укладку вершин с очень большим числом вершин на одном слое. В методе, описанном в [7] задана максимальная число вершин на уровне w, и вершины распределяются таким образом, чтобы на каждом уровне оказалось не больше заданного числа. Получившаяся укладка оказывается не очень широкой, однако число w должно быть задано заранее.

Предложенная нами эвристика заключается в равномерном распределении вершин по уровням. Равномерность заключается в том, что длины между любой парой ребер не будут сильно отличаться, а вершины на любых путях в графе будут расположены друг от друга с некоторым шагом. Для данной эвристики будем рассматривать только ациклические графы специального вида. Эти графы должны содержать в себе один источник (вершина, в которой нет входящих в нее связей) и один сток (вершина, в которой нет исходящих связей). Если граф G будет содержать несколько источников $t_1, \dots t_n$, то мы добавим новую вершину t и добавим следующие ребра: $(t, t_1), \dots, (t, t_n)$. Аналогичным образом можно поступить в случае нескольких стоков $s_1, \dots, s_m$, добавив новую вершину s и ребра $(s_1, s), \dots, (s_m, s)$.

Приведем используемые термины для данной эвристики. Глубина вершины v – наибольшая длина пути из какого-либо источника (обозначается $d(v)$). Высота вершины v – наибольшая длина пути в какой-либо сток

(обозначается $h(v)$). $l(v)$ – длина самого длинного пути, проходящий через вершину *v*. Очевидно, что $l(v) = d(v) + h(v)$.

Дадим несколько вспомогательных определений, которые используются в данном алгоритме.

Определение 10. *Пусть* $G = (V, E) -$ *ациклический ориентированный граф.* $V' \subseteq V$ – *произвольное множество вершин.* V'*–путь в графе* G *– последовательность вершин* $v_0, v_1, \dots, v_k, v_{k+1}$ *такая, что* $v_0, v_{k+1} \in V', v_i \notin V'$ для всех $1 \leq i \leq k$.

Смысл алгоритма заключается в том, чтобы вначале уложить самые длинные пути в графе *G*, а затем найти и уложить пути меньшей длины. Отметим, что согласно выбранным ограничениям на укладываемый граф, начало и конец данного пути уже имеют присвоенный номер уровня, поэтому оставшиеся вершины можно уложить с неким шагом *p*, который можно легко вычислить.

Перейдем к формальному описанию самого алгоритма и методов, которые в нем используются. В описаниях алгоритмов будут использованы следующие обозначения: $L(v)$ – номер уровня для вершины $v \in V$, $d_{V'}(v)$ – глубина вершины *v*, $h_{V'}(v)$ – высота вершины *v*.

Алгоритм 3. Укладка графа.

Процедура: EmbedGraph

Вход: ациклический ориентированный граф *G = (V, E)*

(1) Отсортировать вершины графа *G* в топологическом порядке
(2) $EmbedLongestPath(G)$;
(3) $V' = \{v \in V \mid L(v) \neq 0\}$;
(4) $\boldsymbol{while\ V' \neq V\ do}$
(5) $\quad d = CalculateDepths(G, V')$;
(6) $\quad h = CalculateHeights(G, V')$;
(7) $\quad \pi = FindLongestPath(G, V', d, h)$;
(8) $\quad$ Пусть $\pi = (v_0, v_1, \dots, v_k, v_{k+1})$.
(9) $\quad EmbedPath(G, \pi)$;
(10) $\quad V' = \{v \in V \mid L(v) \neq 0\}$;
(11) $\boldsymbol{end\ while}$

Алгоритм 4. Укладка самых длинных путей.

Процедура: EmbedLongestPaths

Вход: ациклический ориентированный граф $G = (V, E)$

(1) Вычислить $d(v)$ и $h(v)$ для каждой вершины $v\ in\ V$

(2) Пусть K – длина самого длинного пути в графе.

(3) $W = \{ v \in V \mid d(v) + h(v) = K \}$;

(4) $\boldsymbol{for\ all}\ v\ \boldsymbol{in}\ W\ \boldsymbol{do}$

(5) $L(v) = d(v) + 1$;

(6) $\boldsymbol{end\ for}$

(7) $\boldsymbol{for\ all}\ v\ in\ V \setminus W\ \boldsymbol{do}$

(8) $L(v) = 0$

(9) $\boldsymbol{end\ for}$

Алгоритм 5. Вычисление V’-глубин вершин графа.

Процедура: CalculateDepths

Вход: ациклический ориентированный граф, $G = (V, E), V' \subseteq V$

Выход: значения $d_{V'}(v)$ для $v \in V$

(1) Пусть вершины $v_1, \ldots, v_n$ – вершины графа G, расположенные в топологическом порядке.

(2) $\boldsymbol{for}\ i = 1\ \boldsymbol{to}\ n\ \boldsymbol{do}$

(3) $d(v_i) = 0$;

(4) $\boldsymbol{end\ for}$

(5) $\boldsymbol{for}\ i = 1\ \boldsymbol{to}\ n\ \boldsymbol{do}$

(6) $\boldsymbol{for\ all}\ (v_i, v_j) in\ E\ \boldsymbol{and}\ v_j \notin V'\ do$

(7) $\boldsymbol{if}\ d(v_j) < d(v_i) + 1\ \boldsymbol{then}$

(8) $d(v_j) = d(v_i) + 1$;

(9) $\boldsymbol{end\ if}$

(10) $\boldsymbol{end\ for}$

(11) $\boldsymbol{end\ for}$

(12) $\boldsymbol{return}\ d$;

Алгоритм 6. Вычисление V’-высот вершин графа.

Процедура: CalculateHeights

Вход: ациклический ориентированный граф, $G = (V, E), V' \subseteq V$

Выход: значения $h_{V'}(v)$для всех $v \in V$.

(1) $E' = \{ (u, v) | (v, u) \in E \}$;

(2) $G' = (V, E')$;

(3) $h = CalculateDepths(G', V')$;

(4) $return\ h$;

Алгоритм 7. Поиск длинных путей.

Процедура: FindLongestPath

Вход: ациклический ориентированный граф *G = (V, E),* множество V' (содержится в или равно) V, функции $d_{V'}$ и $h_{V'}$

Выход: некоторый V'-путь максимальной длины.

(1) $m = max\{d_{V'}(v) + h_{V'}(v) | v \in V \setminus V'\}$;
(2) $W = \{v \in V \setminus V' | d_{V'}(v) + h_{V'}(v) = m\}$;
(3) $F_1 = \{ (u, v) \in E | u \in V', d_{V'}(v) + h_{V'}(v) = m\}$;
(4) $F_2 = \{ (u, v) \in E \cap (W \times W) | d_{V'}(v) = d_{V'}(u) + 1\}$;
(5) $F_3 = \{ (u, v) \in E | d_{V'}(u) + h_{V'}(u) = m, v \in V' \}$
(6) Выбрать вершины v_0, v_1 такие, что $(v_0, v_1) \in F_1$.
(7) $i = 1$;
(8) $\boldsymbol{while}$ существует ребро $(v_i, u) \in F_2 do$
(9) $v_{i+1} = u$; $i = i + 1$;
(10) $\boldsymbol{end\ while}$
(11) Выбрать вершину $v_i + 1$ такую, что $(v_i, v_{i+1}) \in F_3$.
(12) $\boldsymbol{return}\ (v_0, v_1, \dots, v_i, v_{i+1})$;

Алгоритм 8. Укладка пути $v_1, \dots, v_k$.

Процедура: EmbedPath

Вход: ациклический ориентированный граф *G = (V, E),* путь $v_0, \dots, v_{k+1}$, в котором $L(v_0) \neq 0, L(v_{k+1}) \neq 0, L(v_m) = 0$; для $1 \leq m \leq k$.

(1) $i = L(v_0); j = L(v_{k+1})$;
(2) $\boldsymbol{for}\ m = 1\ \boldsymbol{to}\ k\ \boldsymbol{do}$
(3) $\boldsymbol{if}\ j - 1$ делится на $k + 1\ \boldsymbol{then}$
(4) $p = \frac{j-1}{k+1}; n_1 = i + mp$;
(5) $\boldsymbol{else}$
(6) $p = \left[\frac{j-i}{k+1}\right]; x = \left(\left[\frac{j-i}{k+1}\right] + 1\right)(k + 1) + i - j$;
(7) $\boldsymbol{if}\ m \leq x\ \boldsymbol{then}$
(8) $n_1 = i + mp$;
(9) $\boldsymbol{else}$
(10) $n_1 = i + xp + (m - x)(p + 1)$;
(11) $\boldsymbol{end\ if}$
(12) $\boldsymbol{end\ if}$
(13) $n_2 := \min\{L(u) | L(u) \neq 0, (v_m, u) \in E$;
(14) $L(v_m) = min\{n1. n2 - 1\}$;
(15) $\boldsymbol{for}\ r = m\ \boldsymbol{to}\ 2\ \boldsymbol{step} - 1\ \boldsymbol{do}$
(16) $\boldsymbol{if}\ L(v_r) \leq L(v_{r-1})\ \boldsymbol{do}$
(17) $L(v_{r-1}) = L(V_r) - 1$;
(18) $\boldsymbol{end\ if}$
(19) $\boldsymbol{end\ for}$

2.2 Выбор языка программирования

В качестве инструментария для разработки, была выбрана кроссплатформенная библиотека Qt, предназначенная для эффективного создания приложений с графическим интерфейсом на объектно-ориентированном языке программирования C++ (11-го стандарта). Выбор данного языка был связан с тем, что скомпилированный на нем исходный код программы, по своей скорости сравним с исходным кодом написанном на языке ассемблера.

Язык C++, возникший в 1980-х годах, широко используется для разработки различных программ, поэтому он остается одним из наиболее популярных языков программирования.

Выбор фреймворка Qt связан с тем, что он позволяет запускать созданное на его основе программное обеспечение на современных операционных систем, путем перекомпиляции исходного кода для каждой системы без каких-либо изменений самого кода.

2.3 Описание программного средства «Grapher»

Данная программа состоит из некоторого множества различных классов, каждый выполняющий отведенную ему роль. Опишем действия каждого класса.

Класс MainWindow, унаследованный от базового класса Qt QMainWindow, представляет собой описание главного окна реализованной программы. В нем описан основной функционал, такой как сохранение и загрузка состояния приложения, сохранения изображения текущего графа в файл PNG-формата, запуск процедуры проверки планарности и др. Для вызова функций текущего класса, использован базовый механизм Qt, именуемый сигналами и слотами. Для каждого сигнала (нажатие на кнопку, произошло передвижение курсора и так далее), можно задать набор определенных слотов, которые будут вызваны сразу после того, как сработает сигнал.

Аналогичным образом построен класс LayoutSelectDialog. Данный класс также унаследован от базового класс Qt – QDialog, используемый для описания диалоговых окон приложения. Ключевым методом данного класса является метод createSelectedLayout, который на основе выбранных пользователем данных при помощи шаблона проектирования фабрика создает требуемый метод укладки. Впоследствии данный метод укладки вызывается в классе MainWindow, при закрытии окна LayoutSelectDialog щелчком по кнопке «Ок».

Сама модель граф представлена 3 классами: CGraph, CNode и CEdge, описывающие граф, вершину и ребро соответственно. В классе CGraph реализованы такие базовые возможности, как добавление, удаление и получение ребер и вершин, а также более специфичные методы, например, сохранение и загрузка графа при помощи массива бинарных данных QByteArray, используемые в сохранении и загрузки состояния приложения.

Представление модели графа описано классами CGraphView, CGraphicsNode и CGraphicsEdge. Данные классы являются наследниками базовых классов Qt для работы с графическими примитивами. CGraphView является наследником класса QGraphicsView отображающий переданные ему графические примитивы на экране компьютера. Функции CGraphView нужны для создания и редактирования графа. Например функция wheelEvent добавляет возможность масштабирования всего графического представления, путем прокрутки колеса мыши. Функционал добавления вершин и ребер определен в функции eventFilter, отлавливающая события данного виджета. Наконец, в данном классе также содержится функционал сохранения и загрузки позиций вершин, используемый в классе MainWIndow.

Классы CGraphicsNode и CGraphicsEdge описывают каким образом следует отрисовывать вершины и ребра графа на графическом представлении CGraphView. Класс CGraphicsEdge содержит в себе функцию adjust, пересчитывающий конечные точки каждого ребра.

Класс CPlanarTesting используется для описания процедуры проверки графа на планарность, описанной в пунке 1.3.3 данной работы. Ключевой функцией данного класса является функция testPlanarity, возвращающая true, если текущий граф является планарным и false, если нет. Данная функция вызывает соответствующие этапы алгоритма Бойера-Мирвольды, такие как walkup и walkdown. Для данного алгоритма, также были реализованы вспомогательные классы CBlock и CDFSTree. CBlock представляет структуру вложения графа в плоскость, реализованный как некоторый блок двусвязных компонент графа. CDFSTree описывает древовидную структуру, построенную в ходе обхода графа в глубину. Данная структура содержит такие поля, как дочерние узлы дерева обхода графа в глубину, индекс низшей точки вершины и глубину текущего узла.

Классы CForceDirectedLayout и CSigyamaLayout описывают сами процедуры укладок графов с применением физических моделей и иерархическую укладку соответственно. Данные классы являются наследником базового класс AGraphLayout, представляющий собой абстрактный интерфейс укладки графа. В данном интерфейсе присутствует лишь одна функция – doLayout, вызывающая саму процедуру укладки.

Функционал класса CForceDirectedLayout описывает этапы укладки с применением физических моделей, такие как вычисление сил и применение вычисленных сдвигов. Данные функции вызываются на каждой итерации цикла, находящегося в функции doLayout.

Класс CSigyamaLayout, как уже было сказано выше, представляет собой метод иерархической укладки Сугиямы. Этапы метода Сугиямы были разделенены на несколько классов, что позволяет с легкостью добавлять и менять эвристические методы данной укладки.

Класс CTrivialCycleRemover реализует абстрактный интерфейс эвристики удаления циклов из графов с использованием метода обхода графа в глубину.

Класс CTrivialLayerAssigner является реализаций эвристического подхода к задаче распределения вершин с использованием обхода графа в глубину.

Класс CEqualDistributionHeuristic является реализацией метода равномерного распределения вершин, описанного в пункте 2.1.2 данной работы.

Класс CTrivialNodeOrderer представляет собой эвристический подход к сортированию вершин на уровнях, который также был описан в пункте 2.1.2 данной работы.

Последний класс CTrivialCoordinateAssigner описывает функционал определения координат вершин графа, для непостредственной отрисовке на графическом представлении.

Перейдем к непосредственному описанию самого программного средства. При запуске приложения, пользователь может наблюдать главное окно программы, которое показано на рисунке 18.

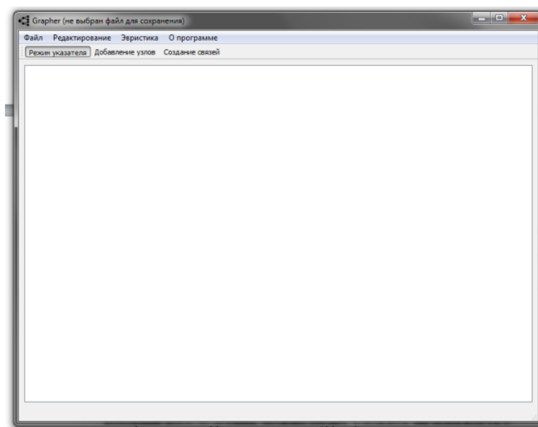

Рисунок 18 Главное окно программы

Перейдем к описанию функционала данного окна. Щелкнув по пункту меню «Файл», пользователю отобразится выпадающий список, показанный на следующем рисунке. В данном списке можно выбрать один из основных функционалов приложения. Опишем действия каждого пункта подробно.

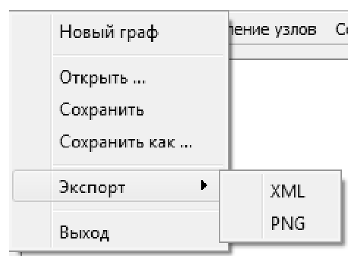

Рисунок 19 Выпадающее меню «Файл»

- «Новый граф» - служит для создания в приложении нового графа, который впоследствии можно будет уложить на плоскость с использованием одно из реализованных методов укладки;
- Пункт списка «Открыть» позволяет пользователю открыть ранее сохраненное состояние приложения для возобновления работы с графом. После того как пользователь щелкнет по данному пункту, ему будет показан стандартное диалоговое окно операционной системы, в котором можно будет открыть файл с сохраненными результатами работы;
- Пункты списка «Сохранить» и «Сохранить как …» позволяют пользователю сохранить состояние приложения и, соответственно сам граф и его укладку в некий файл с расширением *.sav* на диске.
- Пункт списка «Экспорт» позволяет пользователю экспортировать данное изображение графа либо в XML-файл для дальнейшей работы с ним в другом приложении, либо в изображение с расширением PNG;
- Щелкнув по пункту списка «Выход», пользователь может выйти из данного приложения.

Перейдем к описанию пункта меню «Редактирование».

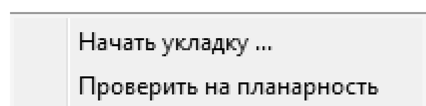

Рисунок 20 Меню «Редактирование»

- Пункт списка «Начать укладку …» открывает вспомогательное диалоговое окно «Выбор укладки», где пользователь может выбрать

и настроить один из реализованных методов укладки для графа. Подробнее об этом окне будет рассказано ниже;

- Пункт списка «Проверить на планарность» позволяет проверить текущий граф с использованием метода добавления ребер, описанного выше в данной работе. После выполнения данного метода, на экране пользователя появится сообщение о том, является ли введенный граф планарным.

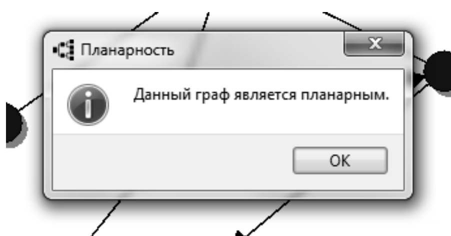

Рисунок 21 Результат работы метода добавления ребер

Пункт меню «Эвристика» служит для запуска эвристического метода добавления узлов, описанная в пункте 2.1.1 данной работы.

Последний пункт меню «О программе» позволяет пользователю узнать текущую версию приложения, а также версию библиотек Qt, с которыми приложение было собрано.

Сразу под главным меню приложения пользователю предоставлено небольшое окно инструментов, на котором находятся 3 кнопки: «Режим указателя», «Добавление узлов» и «Создание связей». Пользователь может использовать данные кнопки, чтобы переключать режим работы с графическим представлением графа, о котором будет рассказано ниже.

Большую часть главного окна занимает объект графического представления графа. В нем пользователь может создавать граф, используя для этого мышь. Работа с данным представлением напрямую зависит от выбранного режима работы.

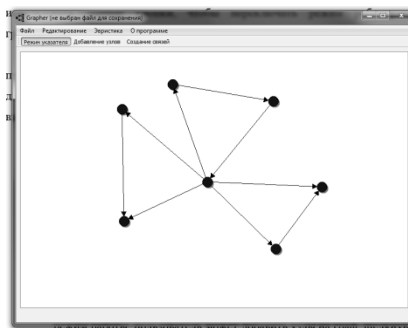

Рисунок 22 Главное окно приложения с некоторым нарисованным на нем графом

- Режим работы «Режим указателя» - при выборе данного режима, с помощью левой кнопки мыши пользователь может перемещать окно просмотра графического представления в любом направлении. Кроме этого, пользователь может передвигать уже созданные узлы графа, щелкнув и зажав на этом узле левую кнопку мыши.
- Режим работы «Добавление узлов» - при использовании данного режим работы, пользователь может добавить узлы на граф, щелкнув в любом месте графического представления левой кнопкой.
- Режим работы «Создание связей» служит для того, чтобы добавлять ребра на граф. Для этого пользователю достаточно выбрать на необходимые узлы графа левой кнопкой мыши. Первый выбранный узел будет являться родительским для данного ребра. Создание ребра можно отменить, переключившись на другой режим работы с графическим представлением.

Теперь перейдем к описанию диалогового окна приложения «Выбор укладки».

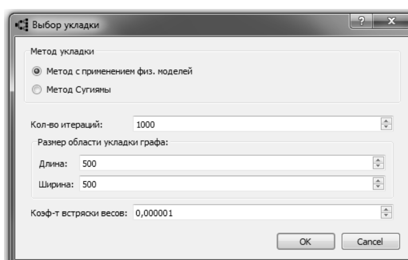

Рисунок 23 Диалоговое окно «Выбор укладки» с выбранным методом укладки с использованием физических моделей

При помощи данного окна пользователь может выбрать либо метод укладки с применением физических моделей, либо метод Сугиямы. При выборе одного из методов укладки пользователю будет предложено выбрать конкретные значения параметров, применяемые к тому или иному методу, а также выбор эвристических методов, реализованных в данном приложении.

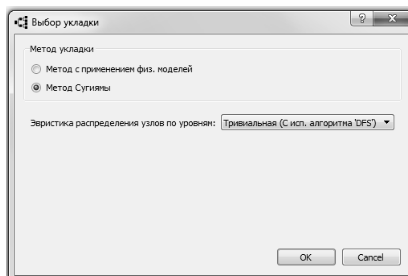

Рисунок 24 Диалоговое окно «Выбор укладки» с выбранным методом укладки Сугиямы

После щелчка левой кнопкой мыши по кнопке «Ок» выбранный метод укладки запустит свою работу на введенном графе. Результат работы алгоритма будет отображен на графическом представлении графа.

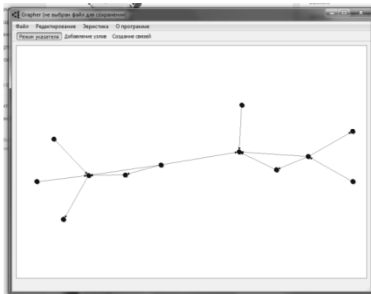

Рисунок 25 Результат работы метода укладки с применением физической модели для некоторого графа

Глава 3 Вычислительные эксперименты

3.1 Вычислительные эксперименты для метода равномерной укладки

Отметим, что все вычислительные эксперименты проводились на компьютере со следующими характеристиками: Intel® Core(TM) i5-2400-CPU, 3.10 Ghz, 4 ядра, ОЗУ 32ГБ, операционная система Windows 7-64Bit.

Далее в работе используется определение плотности графа. Определим его.

Определение 11. *Плотностью графа называют число, определяемое по следующей формуле:*

$$D = \frac{2|E|}{|V|(|V|-1)}.$$

Приведем эксперименты для метода равномерного распределения вершин. На рисунке 26 можно увидеть две иерархические укладки одного и того же графа.

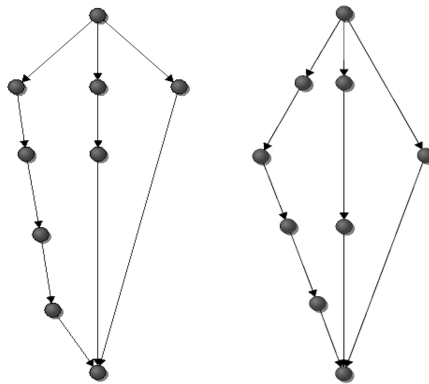

Рисунок 26 Сравнение методов

Правая укладка была получена с помощью метода равномерного распределения вершин. Левая была получена с использованием эвристики распределения вершин по методу обхода графа в глубину. Можно заметить, что данные метод распределил вершины укладываемого графа более равномерно, а разница в длинах ребер является незначительной, по сравнению с правым рисунком.

Результаты вычислительного эксперимента можно увидеть в таблице 1.

Число вершин	Число ребер	Время (с)	Число ребер	Время (с)	Число ребер	Время (с)
100	300	1	300	1	3000	1
1000	3000	2	25000	4	250000	28
2000	6000	15	100000	44	1000000	403
4000	12000	62	400000	444	4000000	6146
6000	18000	654	1000000	1830	15000000	> 8 часов

Таблица 1 Результаты работы алгоритма

3.2 Сравнительный анализ

Хотя метод Сугиямы и метод с применением физических моделей являются двумя совершенно разными подходами к решению задачи укладки графов на плоскость, оптимизируя различные эстетические критерии качества укладки, проведем их сравнение по нескольким критериям. В качестве критериев для сравнения, мною были выбраны следующие:

1. Время работы;
2. Трудности в реализации;
3. Размер занимаемой области уложенного графа, без ухудшения читаемости;
4. Число пересечений;

Под критерием размера занимаемой области подразумевается то, насколько близко можно разместить вершины друг к другу (предполагая, что размер самой вершины фиксирован), не ухудшая его читаемость (можно проследить, куда ведут ребра графа).

Время работы. Для определения времени работ алгоритма были случайным образом сгенерированы графы на 10, 100, 1000, 10000 и 10000 вершин. Отметим, что конечный результат был усреднен.

Ниже приведены графики зависимости времени работ алгоритмов от сложности структуры графов. Вертикальная ось представляет собой время в минутах. Горизонтальная ось представляет собой количество вершин графа.

Вычисление времени работы проводилось с использованием эвристических методов, указанных в главе 2 данной работы.

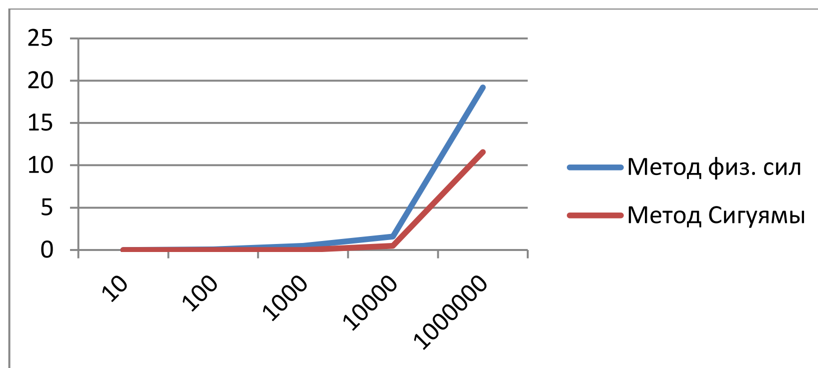

Рисунок 27 График времени затраченного на получения изображений с использованием метода физических сил и метода Сугиямы

Опираясь на полученные результаты, можно сделать вывод о том, что метод Сугиямы работает намного быстрее метода физических сил. Данный факт можно объяснить тем, что метод физических сил является итеративным методом и для его сходимости может потребоваться достаточно большое число итераций. Кроме этого, время одной итерации в худшем случае занимает $O(n^2)$. Алгоритм с использованием физических сил можно сильно ускорить, добавив дополнительное ограничение на число итераций алгоритма. Однако от этого может существенно ухудшиться качество конечной укладки.

Трудности в реализации. Основное преимущество метода укладки с использованием физических моделей является его интуитивность и простота описания самого алгоритма. Алгоритм достаточно просто реализовать и использовать как для использования различных эвристических методов, так и для построения изображений графов, например, с целью анализа данных. Метод Сугиямы в этом плане проигрывает, так как для его реализации требуется протестировать множество эвристических подходов для каждого этапа алгоритма, прежде чем алгоритм начнет выдавать адекватные результаты и за разумное время. Однако эти трудности полностью

оправдывают себя, так как конечный результат получается намного нагляднее и информативнее.

Размер занимаемой области. В отличие от метода Сугиямы, метод с использованием физических сил имеет параметры, регулирующие размеры области необходимой для построения изображения укладки графа. Для метода Сугиямы можно попытаться уменьшить конечную область, путем использования более жадных методов определения координат для вершин, которые будут стараться размещать вершины как можно ближе друг к другу. Однако от этого может сильно ухудшится читаемость графа.

Число пересечений. Для сравнения числа пересеченый, случайным образом были сгенерированы графы различной сложности на 100, 1000 и 2000 вершин. После этого было взято среднее число пересечений для каждого случая. Данный способ позволяет дать некоторую среднюю оценку для числа пересечений, которые могут быть допущены при использовании метода укладки с применением физических сил и метода Сугиямы.

Резульаты вычислительного эксперимента можно увидеть в таблице 2.

Кол-во вершин	Метод физ. сил	Метод Сугиямы
100	21512	15665
1000	2967512	4346823
2000	19254424	21243176

Таблица 2 Среднее число допущенных пересечений

Как можно заметить, число пересечений для метода Сугиямы с выбранным набором эвристических методов, указанных в четвертом пункте данной работы, работает несколько хуже, чем метод укладки с применением физических моделей. Данный результат можно улучшить, попробовав применить другие эвристические подходы для сортировки вершин на слоях.

Заключение

При выполнении данной работы, были изучены метод укладки с использованием физической модели, метод иерархической укладки Сугиямы, а также метод проверки графов на планарность с использованием метода добавления ребер. Было реализовано программное обеспечение, в котором применяются указанные методы. Также были рассмотрены некоторые возможные эвристики улучшения качества конечных изображений графов, уложенных на плоскость с использованием выбранных методов.

Для эвристики равномерного распределения вершин графа, вычислительные эксперименты показали, что для графов с малым значением плотности время работы составляет от нескольких миллисекунд, до нескольких минут.

Сравнение метода укладки с физическими моделями и метода Сугиямы показало, что с выбранными эвристическими подходами метод Сугиямы работает немного хуже, чем другой метод.

В дальнейшем можно изучить возможность параллельных вычислений для методов укладок графов и предложенных эвристических методов.

Список литературы

1. Алгоритмы: Построение и анализ, 2-е издание [Текст] / Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. // пер с англ. М.: Издательский дом «Вильямс», 2005. 1296 с.
2. Касьянов В. Н., Евстигнеев В.А. Графы в программировании: обработка, визуализация и применение [Текст] / В. Н. Касьянов, В.А Евстигнеев. // СПб., БХВ-Петербург, 2003 – 1104 с. ил.
3. Харари Ф. Теория графов [Текст] / Пер. с англ. И предисл. В. П. Козырева. Под ред. Г. П. Гаврилова. Изд. 2-е. // М.: Едиториал УРСС, 2003. – 296 с.
4. Bastert O., Matuszewski C. Layered drawings of digraphs / O. Bastert, C. Matuszewski // In Kaufmann M., Wagner D. Drawing Graphs: Methods and Models. Lecture Notes in Computer Science, vol. 2025, Springer-Verlag, 2001, pp. 87–120. https://doi.org/10.1007/3-540-44969-8_5
5. Bondy J.A., Murty U.S.R., Graph Theory with Applications / J.A. Bondy, U.S.R. Murty // North Holland, Amsterdam, 1976
6. Boyer J., Myrvold W., Simplified Planarity / J. Boyer, W. Myrvold // JGAA, 8(3) 241–273 (2004) 242
7. Coffman E. G., Graham R.L. Optimal scheduling for two processor systems / E. G. Coffman, R.L. Graham // Acta Informatica. 1972, vol. 1. pp 200-213. https://doi.org/10.1007/BF00288685
8. Eades P., Complexity Issues in Drawing Directed Graphs, Proc. / P. Eades // Int. Workshop on Discrete Algorithms and Complexity, pp. 9{15, Fukuoka, Japan, 1989.
9. Ferrari D., Mezzalira L., On Drawing a Graph with the Minimum Number of Crossings / D. Ferrari, L. Mezzalira // Technical Report n. 69-11, Istituto di Elettrotecnica ed Elettronica, Politecnico di Milano, 1969.
10. Guy R. K. A combinatorial problem, Nabla / R. K. Guy // Bulletin of the Malayan Mathematical Society). 7: 68-72, 1960.

11. Hopcroft J., Tarjan R. Efficient planarity testing / J. Hopcroft, R. Tarjan // Journal of the Association for Computing Machinery, 24 (4): 549-568, 1974.
12. Johnson D.S., The NP-Completeness Column: an Ongoing Guide, / D.S. Johnson // J. of Algorithms, vol. 3, no. 1, pp. 89-99, 1982.
13. Johnson D.S., The NP-Completeness Column: an Ongoing Guide / D.S. Johnson // J. of Algorithms, vol. 5, no. 2, pp. 147-160, 1984.
14. Lin X., Analysis of Algorithms for Drawing Graphs, PhD thesis / X. Lin // Department of Computer Science, University of Queensland, 1992.
15. Sander G. Graph layout for applications in compiler construction / G. Sander // Theoretical Computer Science. 1999, vol. 217, no. 2, pp. 175–214. https://doi.org/10.1016/S0304-3975(98)00270-9
16. Sugiyama K., Tagawa S., Toda M., Methods for Visual Understanding of Hierarchical Systems / K. Sugiyama, S. Tagawa, M. Toda // IEEE Trans. on Systems, Man, and Cybernetics, vol. SMC-11, no. 2, pp. 109- 125, 1981.
17. Sugiyama K., Toda M., Structuring Information for Understanding Complex Systems: A Basis for Decision Making / K. Sugiyama, M. Toda // FUJITSU Scientific and Technical Journal, vol. 21, no. 2, pp. 144-164, 1985.
18. A Technique for Drawing Directed Graphs / E.R. Gansner, E. Koutsofos, S.C. North, K.P. Vo // IEEE Trans. on Software Engineering, vol. 19, no. 3, pp. 214-230, 1993.
19. Thomassen C., Planarity and Duality of Finite and Infinite Planar Graphs / C. Thomassen // J. Combinatorial Theory, Series B, vol. 29, pp. 244-271, 1980.
20. Vaucher J., Pretty Printing of Trees / J. Vaucher // Software Practice and Experience, vol. 10, no. 7, pp. 553-561, 1980.
21. Wetherell C., Shannon A., Tidy Drawing of Trees / C. Wetherell, A. Shannon // IEEE Trans. on Software Engineering, vol. SE-5, no. 5, pp. 514-520, 1979.
22. Woods D., Drawing Planar Graphs, Ph.D. dissertation / D. Woods, // Technical Report STAN-CS-82-943), Computer Science Dept., Stanford Univ., 1982.

23. Zarankiewicz K. On a Problem of P. Turan Concerning Graphs. / K. Zarankiewicz // Fundamenta Mathematicae vol. 41, pp. 137-145, 1954

Приложение

```
#ifndef AGRAPHLAYOUT_H
#define AGRAPHLAYOUT_H

#include "cgraph.h"

#include <qmap.h>
#include <qpoint.h>

/* Базовый класс укладки
 */
class AGraphLayout
{

public:

    AGraphLayout() {}

    virtual ~AGraphLayout() {}

    virtual QMap<CNode*, QPoint > doLayout(CGraph* g) = 0;
};

#endif // AGRAPHLAYOUT_H
#ifndef CEDGE_H
#define CEDGE_H

#include <qgraphicsitem.h>

class CGraphicsNode;

class CGraphicsEdge : public QGraphicsItem
{
public:

    CGraphicsEdge(CGraphicsNode* parent, CGraphicsNode* child);

    CGraphicsNode* parent() const { return parent_; }
    CGraphicsNode* child() const { return child_; }

    void adjust();

protected:

    QPainterPath shape() const;

    QRectF boundingRect() const;

    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget);
```

```
protected:

    CGraphicsNode* parent_;
    CGraphicsNode* child_;

    QPointF sourcePoint_;
    QPointF destPoint_;
    qreal arrowSize_;
};

#endif // CEDGE_H
#ifndef CEQUALDISTRIBUTIONHEURISTIC_H
#define CEQUALDISTRIBUTIONHEURISTIC_H

#include "csigyamalayout.h"

class CEqualDistributionHeuristic : public ALayerAssigner
{

public:

    CEqualDistributionHeuristic() {}

    QMap<int, QList<SSigNode*> > assignLayers(QList<SSigNode*>& nodes);

private:

    QHash<CNode*, int> embedGraph(CGraph* CGraph);

    void embedLongestPath(CGraph* g);

    QHash<CNode*, int> calculateDepths(const QList<CNode *> &sorted, const
QList<CNode*>& v_);

    QHash<CNode*, int> calculateHeights(CGraph* g, const QList<CNode*>& v_);

    QList<CNode *> findLongestPath(CGraph *g, const QList<CNode *> &v, const
QHash<CNode *, int> &d, const QHash<CNode *, int> &h);

    void embedPath(CGraph* g, const QList<CNode*>& path);

    QList<CNode*> calcVSet();

private:

    QHash<CNode*, int> layers_;

    CGraph* inverted_;

    QList<CNode*> graphSorted_;
```

```
    QList<CNode*> invertedGraphSorted_;
};

#endif // CEQUALDISTRIBUTIONHEURISTIC_H
#ifndef CFORCEDIRECTEDLAYOUT_H
#define CFORCEDIRECTEDLAYOUT_H

#include "agraphlayout.h"

#include <qvector.h>

/* Укладка с использованием физ. моделей
 */
class CForceDirectedLayout : public AGraphLayout
{

public:

    CForceDirectedLayout(qreal w, qreal h, int iters)
        : AGraphLayout(), w_(w), h_(h), area_(w * h)
        , iters_(iters) { }

    QMap<CNode*, QPoint> doLayout(CGraph* g) override;

private:

    float f_a(float x) const;
    float f_r(float x) const;
    float norm(const QPointF& p) const;

    void calcRepulsiveForces();
    void calcAttractiveForces(CGraph* g);
    void applyCorrections();

    // Создаем необходимую структуру
    struct SNode
    {
        CNode* node_;
        QPointF pos_;

        SNode() : node_(0), pos_() { }
    };

private:

    qreal w_;
    qreal h_;
    qreal area_;
    int iters_;

    QMap<CNode*, QPointF> disp_;
    QVector<SNode> nodes_;
```

```
};

#endif // CFORCEDIRECTEDLAYOUT_H
#ifndef CGRAPH_H
#define CGRAPH_H

#include <qobject.h>
#include <qlist.h>
#include <qset.h>

class CGraph;

/*
 */
class CNode
{
   friend class CGraph;

public:

   CNode() : graph_(NULL) {}

   int childCount() const { return children_.size(); }
   CNode* child(int i) const { return children_.at(i); }

   int parentCount() const { return parents_.size(); }
   CNode* parent(int i) const { return parents_.at(i); }

   CGraph* graph() const { return graph_; }

private:

   CGraph* graph_;

   QList<CNode*> parents_;
   QList<CNode*> children_;
};



/*
 */
class CEdge
{
   friend class CGraph;

public:

   CEdge(CNode* parent, CNode* child)
       : parent_(parent), child_(child) {}
```

```
    CNode* parent() const { return parent_; }
    CNode* child() const { return child_; }

    void setParent(CNode* p) { parent_ = p; }
    void setChild(CNode* c) { child_ = c; }

private:

    CGraph* graph_;
    CNode* parent_;
    CNode* child_;
};



/*
 */
class CGraph : public QObject
{
    Q_OBJECT

public:

    CGraph();
    ~CGraph();

    CNode* addNode();
    CNode* node(int i) const { return nodes_.at(i); }
    int nodeCount() const { return nodes_.size(); }
    int nodeIndex(CNode* n) const { return nodes_.indexOf(n); }

    CEdge* addEdge(int i, int j);
    int edgeCount() const { return edges_.size(); }
    CEdge* edge(int i) const { return edges_.at(i); }
    CEdge* edge(CNode* u, CNode* v) const;

    // Топологическая сортировка
    QList<CNode *> topologicalSorted() const;

    static CGraph *generateAccyclicGraph(int nodes, double density);

    QByteArray serialize() const;

    void deserialize(const QByteArray& arr);

signals:

    void nodeAdded();

    void edgeAdded();

private:
```

```
    QList<CNode*> nodes_;
    QList<CEdge*> edges_;
};

#endif // CGRAPH_H
#ifndef CGRAPHICSNODE_H
#define CGRAPHICSNODE_H

#include <qgraphicsitem.h>

class CGraphicsEdge;

class CGraphicsNode : public QGraphicsItem
{

public:

    CGraphicsNode();

    void addEdge(CGraphicsEdge* edge);
    QList<CGraphicsEdge*> edges() const { return edges_; }

    QRectF boundingRect() const;
    QPainterPath shape() const;

    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget);

protected:

    QVariant itemChange(GraphicsItemChange change, const QVariant &value);

    void mousePressEvent(QGraphicsSceneMouseEvent *event);
    void mouseReleaseEvent(QGraphicsSceneMouseEvent *event);

private:

    QList<CGraphicsEdge*> edges_;
};

#endif // CGRAPHICSNODE_H
#ifndef CGRAPHVIEW_H
#define CGRAPHVIEW_H

#include <qevent.h>
#include <qgraphicsview.h>
#include <qgraphicsscene.h>
#include "cgraph.h"

class CGraphicsNode;
class CGraphicsEdge;
```

```
/*
 */
class CGraphView : public QGraphicsView
{
    Q_OBJECT

public:

    enum EPointerMode {ePointer = 0, eNodeAdding, eEdgeAdding};

    CGraphView(QWidget* parent = 0);

    void setGraph(CGraph* g);

    CGraph* graph() const { return graph_.data(); }

    void saveToFile(const QString& filename);

    void wheelEvent(QWheelEvent *event);

    bool eventFilter(QObject *pObj, QEvent *pEv);

    void setPointerMode(EPointerMode mode);

    CGraphicsNode* node(int i) const { return nodes_.at(i); }

    QByteArray serialize() const;

    void deserialize(const QByteArray &arr);

    QList<CGraphicsEdge*> edges() const {return edges_;}

    void setEdgeParent(CGraphicsEdge* e, CNode* p);

public slots:

    CNode* addNode(const QPointF& pos);

    void addEdge(int parentIdx, int childIdx);

private:

    template<typename T>
    T* getItemAt(const QPointF& pos) const
    {
        QList<QGraphicsItem*> list = scene()->items(pos);
        for(int i = 0; i < list.size(); ++i) {
            T* itm = dynamic_cast<T*>(list.at(i));
            if (itm) return itm;
        }
        return NULL;
    }
```

```
private:

    QScopedPointer<CGraph> graph_;

    QList<CGraphicsNode*> nodes_;

    QList<CGraphicsEdge*> edges_;

    EPointerMode mode_;

    CGraphicsNode* firstSelectedNode_;
};

#endif // CGRAPHVIEW_H
#ifndef CPLANARTESTING_H
#define CPLANARTESTING_H

#include "cgraph.h"

#include <qpair.h>
#include <qstack.h>
#include <qmap.h>
#include <qset.h>

/* Специальная структура для метода добавления ребер
 */
class CDFSTree
{
public:

    CDFSTree();

    ~CDFSTree() { qDeleteAll(children_); children_.clear(); }

    QList<CDFSTree*> orderedList() const;

    inline QList<CDFSTree*> pathTo(CDFSTree* w) {
        QStack<CDFSTree*> stack;
        stack.push_back(this);
        QList<CDFSTree*> path;
        pathToUtil(w, stack, path);
        return path;
    }

    inline void setDfi(int dfi) { dfi_ = dfi; }
    int dfi() const { return dfi_; }

    int lowpoint() const { return lowpoint_; }

    CDFSTree* leastAncestor() const { return leastAncestor_; }
```

```
   void addChild(CDFSTree* c) { children_.push_back(c); dfsChildren_.push_back(c); }

   QList<CDFSTree*> children() const { return children_; }

   QList<CDFSTree*> dfsChildren() const { return dfsChildren_; }

   QList<CDFSTree*> backedges() const { return backedges_; }

   void calcLeastAncestor();

   //static CDFSTree* formDFSTree(CGraph* g, QMap<CNode*, CDFSTree*>& mapped);
   static inline bool lowpointLessThan(CDFSTree* l, CDFSTree* r) {
      return l->lowpoint_ < r->lowpoint_;
   }

   static CDFSTree* createDfsTree(CNode *v, int depth, QSet<CNode *> &visited,
QMap<CNode *, CDFSTree *> &mapped);

private:

   bool pathToUtil(CDFSTree* w, QStack<CDFSTree*>& stack, QList<CDFSTree*>& path);

   //static void visit(CNode* v, int depth, int& dfi, QSet<CNode*>& visited, QMap<CNode*,
CDFSTree*>& mapped);

private:

   CNode* node_;
   CDFSTree* parent_;
   CDFSTree* leastAncestor_;

   int depth_;
   int lowpoint_;
   int dfi_;

   QList<CDFSTree*> dfsChildren_;
   QList<CDFSTree*> children_;
   QList<CDFSTree*> backedges_;
};



/* Специальная структура для метода добавления ребер
 */
class CBlock
{

public:

   CBlock();

   ~CBlock() {}
```

```
    void setExternalFace(CDFSTree* v, int dir);

    inline void addVertex(CDFSTree* v, bool flag) {
        vertices_.push_back(v);
        if (flag) bound_.push_back(v);
        if (root_ == NULL || v->dfi() < root_->dfi()) root_ = v;
    }

    CDFSTree* root() { return root_; }
    inline void setRoot(CDFSTree* r) { root_ = r; }

    QList<CDFSTree*> vertices() { return vertices_; }
    inline void setVertices(const QList<CDFSTree*>& list) { vertices_ = list; }

    inline void addEdge(const QPair<CDFSTree*, CDFSTree*>& edge) {
edges_.push_back(edge); }
    QList<QPair<CDFSTree*, CDFSTree*>> edges() { return edges_; }
    inline void setEdges(const QList<QPair<CDFSTree*, CDFSTree*>>& e) { edges_ = e; }

    QList<CDFSTree*> boundaryVertices() { return bound_; }
    inline void setBoundaryVertices(const QList<CDFSTree*>& l) { bound_ = l; }

private:

    CDFSTree* root_;

    QList<CDFSTree*> vertices_;
    QList<QPair<CDFSTree*, CDFSTree*>> edges_;
    QList<CDFSTree*> bound_;
};


/* Метод добавления ребер
 */
class CPlanarTester
{

public:

    CPlanarTester() {}

    ~CPlanarTester() { delete dfsTree_; qDeleteAll(blocks_); blocks_.clear(); }

    bool testPlanar(CGraph* g);

private:

    bool task(CDFSTree* v);

    void walkup(CDFSTree* v, const QPair<CDFSTree*, CDFSTree*>& backEdge);
```

```
    bool walkdown(CDFSTree* v, const QPair<CDFSTree*, CDFSTree*>& backEdge
        , QList<QPair<CDFSTree*, CDFSTree*>>& backEdges);

    void createBlock(CDFSTree* src, CDFSTree* dest);

    CBlock* merge(QList<CBlock*> blocksToBeJoined);

    inline bool isExternallyActive(CDFSTree* w, CDFSTree* v) {
        CDFSTree* la = w->leastAncestor();
        return (la != NULL && la->dfi() < v->dfi()) ||
            (!mapChildrenList_[w].isEmpty() && mapChildrenList_[w].front()->dfi() < v->dfi());
    }

    QList<CDFSTree*> externalyActive(CDFSTree* v);
    QList<CBlock*> blockWith(CDFSTree* v);
    QList<CDFSTree*> outsideFace();
    QList<QPair<CDFSTree*, CDFSTree*>> externalFace();

private:

    CGraph* graph_;
    CDFSTree* dfsTree_;

    QList<CBlock*> blocks_;
    QList<CDFSTree*> ordered_;
    QList<CDFSTree*> endpoins_;
    QList<CDFSTree*> extAct_;

    QMap<CDFSTree*, int> mapLowpoint_;
    QMap<CDFSTree*, QList<CBlock*>> mapBlocks_;
    QMap<CDFSTree*, QList<CDFSTree*>> mapChildrenList_;
    QMap<QPair<CDFSTree*, CDFSTree*>, QMap<CDFSTree*, CBlock*>> mapPertinent_;
};

#endif // CPLANARTESTING_H
#ifndef CSIGYAMALAYOUT_H
#define CSIGYAMALAYOUT_H

#include <qpointer.h>
#include <qstack.h>
#include <qset.h>

#include "agraphlayout.h"

/* Структура узла для алгоритма Сигуямы
 */
struct SSigNode
{
    CNode* node_;
    QList<SSigNode*> children_;

    QList<SSigNode*> parents_;
```

```
   SSigNode(CNode* n) : node_(n), children_() {}
};

/* Базовый класс эвристики удаления циклов
 */
class ACycleRemover
{
public:
   ACycleRemover() {}
   virtual ~ACycleRemover() {}

   virtual void removeCycles(QList<SSigNode*>& nodes) = 0;
};

class CTrivialCycleRemover : public ACycleRemover
{
public:

   CTrivialCycleRemover() : ACycleRemover() {}
   ~CTrivialCycleRemover() {}

   bool removeCyclesUtil(SSigNode* v, QSet<SSigNode*>& visited, QSet<SSigNode*>&
recSet)
   {
      if(visited.contains(v) == false)
      {
         visited.insert(v);
         recSet.insert(v);

         for(int i = 0; i < v->children_.size(); ++i)
         {
            SSigNode* c = v->children_.at(i);
            if ( !visited.contains(c) )
               removeCyclesUtil(c, visited, recSet);
            else if (recSet.contains(c)) {
               v->children_.removeOne(c);
               c->parents_.removeOne(v);
               --i;
            }
         }

      }

      recSet.remove(v);
      return false;
   }

   void removeCycles(QList<SSigNode*>& nodes)
   {
      QSet<SSigNode*> visited;
      QSet<SSigNode*> recSet;
```

```
      for(int i = 0; i < nodes.size(); i++)
         removeCyclesUtil(nodes.at(i), visited, recSet);
   }
};

/* Базовый класс эвристики распределения вершин по уровням
 */
class ALayerAssigner
{
public:

   ALayerAssigner() {}
   virtual ~ALayerAssigner() {}

   virtual QMap<int, QList<SSigNode*> > assignLayers(QList<SSigNode*>& nodes) = 0;
};

class CTrivialLayerAssigner : public ALayerAssigner
{
public:
   CTrivialLayerAssigner() : ALayerAssigner() {}
   ~CTrivialLayerAssigner() {}

   void assignLayersUtil(SSigNode* v, QSet<SSigNode*>& visited, QSet<SSigNode*>& 
recSet, QMap<SSigNode*, int>& nodeToLayer, int layer)
   {
      if(visited.contains(v) == false)
      {
         visited.insert(v);
         recSet.insert(v);
         nodeToLayer[v] = layer;

         for(int i = 0; i < v->children_.size(); ++i)
         {
            SSigNode* c = v->children_.at(i);
            if ( !visited.contains(c) )
               assignLayersUtil(c, visited, recSet, nodeToLayer, layer + 1);
            else if (recSet.contains(c)) {
               nodeToLayer[c] = qMax(nodeToLayer[c], layer);
            }
         }
      }

      recSet.remove(v);
   }

   QMap<int, QList<SSigNode*> > assignLayers(QList<SSigNode*>& nodes)
   {
      QMap<SSigNode*, int> nodeToLayer;

      QList<SSigNode*> ordered;
      for(int i = 0; i < nodes.size(); ++i) {
```

```
            if (nodes.at(i)->parents_.size() == 0) ordered.push_front(nodes.at(i));
            else ordered.push_back(nodes.at(i));
        }

        QSet<SSigNode*> visited;
        QSet<SSigNode*> recSet;
        for(int i = 0; i < nodes.size(); i++)
            assignLayersUtil(nodes.at(i), visited, recSet, nodeToLayer, 0);

        QMap<int, QList<SSigNode*> > levels;
        for(auto it = nodeToLayer.begin(); it != nodeToLayer.end(); ++it)
            levels[it.value()].push_back(it.key());

        return levels;
    }
};

/* Базовый класс сортировки узлов на уровнях
 */
class ANodeOrderer
{
public:

    ANodeOrderer() {}
    virtual ~ANodeOrderer() {}

    virtual void orderNodes(QMap<int, QList<SSigNode*> >& levels) = 0;
};

class CTrivialNodeOrderer : public ANodeOrderer
{
public:

    CTrivialNodeOrderer() : ANodeOrderer() {}
    ~CTrivialNodeOrderer() {}

    void orderNodes(QMap<int, QList<SSigNode*> >& levels)
    {
        auto iterCur = levels.end();
        --iterCur;
        if (iterCur == levels.begin()) return;

        auto iterPrev = iterCur;
        --iterPrev;

        while(true)
        {
            for(int i = 0; i < iterPrev->size(); ++i) {
                SSigNode* node = iterPrev->at(i);
                int new_pos = 0;
                int edgeCount = 0;
                for(int j = 0; j < node->children_.size(); ++j) {
```

```
                SSigNode* ch = node->children_.at(j);
                if (iterCur->indexOf(ch) == -1)
                    continue;

                ++edgeCount;
                new_pos += iterCur->indexOf(ch);
            }
            new_pos = (edgeCount) ? new_pos / edgeCount : i;
            new_pos = qBound(0, new_pos, iterPrev->size() - 1);

            iterPrev->removeOne(node);
            iterPrev->insert(new_pos, node);
        }

        if (iterPrev == levels.begin()) break;
        else --iterPrev;
        --iterCur;
    }
  }
};

/* Базовый класс задания координат
 */
class ACoordinateAssigner
{
public:
    ACoordinateAssigner() { }
    virtual ~ACoordinateAssigner() { }

    virtual QMap<CNode*, QPoint > assignCoordinates(const QMap<int, QList<SSigNode*>
>& levels) = 0;
};

class CTrivialCoordinateAssigner : public ACoordinateAssigner
{
public:
    CTrivialCoordinateAssigner() : ACoordinateAssigner() { }
    ~CTrivialCoordinateAssigner() { }

    QMap<CNode*, QPoint > assignCoordinates(const QMap<int, QList<SSigNode*> >&
levels)
    {
        QMap<CNode*, QPoint > pos;

        qreal y = 0;
        for(auto it = levels.begin(); it != levels.end(); ++it) {
            qreal x = 0;
            qreal cen = 100 * it->size() * 0.5f;
            for(auto jt = it->begin(); jt != it->end(); ++jt) {
                pos[(*jt)->node_] = QPoint(-cen + x * 100, y * 100);
                ++x;
            }
```

```
        ++y;
    }

    return pos;
  }
};

/* Класс уклакди методом Сигуямы
 */
class CSigyamaLayout : public AGraphLayout
{

public:

  CSigyamaLayout(ACycleRemover* r, ALayerAssigner* a, ANodeOrderer* o,
                 ACoordinateAssigner* ca)
    : AGraphLayout()
    , remover_(r)
    , assigner_(a)
    , orderer_(o)
    , coordAssigner_(ca)
  {}

  QMap<CNode*, QPoint > doLayout(CGraph* g);

private:

  // Эвристики алгоритма
  QScopedPointer<ACycleRemover> remover_;
  QScopedPointer<ALayerAssigner> assigner_;
  QScopedPointer<ANodeOrderer> orderer_;
  QScopedPointer<ACoordinateAssigner> coordAssigner_;
};

#endif // CSIGYAMALAYOUT_H
#ifndef LAYOUTSELECTDIALOG_H
#define LAYOUTSELECTDIALOG_H

#include <QDialog>

#include "agraphlayout.h"

namespace Ui {
class LayoutSelectDialog;
}

class LayoutSelectDialog : public QDialog
{
  Q_OBJECT

public:
```

```
    explicit LayoutSelectDialog(QWidget *parent = 0);
    ~LayoutSelectDialog();

    AGraphLayout* createSelectedLayout(CGraph *g);

private slots:

    void updateCurrentPage();

private:
    Ui::LayoutSelectDialog *ui;
};

#endif // LAYOUTSELECTDIALOG_H
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:

    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

public slots:

    void exportToPng();

    void save();

    void saveAs();

    void aboutQt();

    void startLayout();

    void newGraph();

    void load();

    void testPlanarity();

    void addNodesInCrossings();
```

```
private slots:

    void onActionGroupTriggered(QAction* act);

private:

    Ui::MainWindow *ui;

    QString fileName_;
};

#endif // MAINWINDOW_H
#include "cedge.h"

#include "cgraphicsnode.h"

#include <qpen.h>
#include <qbrush.h>
#include <qpainter.h>

static const double Pi = 3.14159265358979323846264338327950288419717;
static double TwoPi = 2.0 * Pi;

CGraphicsEdge::CGraphicsEdge(CGraphicsNode *parent, CGraphicsNode *child)
    : QGraphicsItem()
    , parent_(parent)
    , child_(child)
    , arrowSize_(10)
{
    if (parent && child) {
        parent->addEdge(this);
        child->addEdge(this);
        adjust();
    }
}

void CGraphicsEdge::adjust()
{
    if (!parent_ || !child_)
        return;

    QLineF line(mapFromItem(parent_, 0, 0), mapFromItem(child_, 0, 0));
    qreal length = line.length();

    prepareGeometryChange();

    if (length > qreal(20.)) {
        QPointF edgeOffset((line.dx() * 10) / length, (line.dy() * 10) / length);
        sourcePoint_ = line.p1() + edgeOffset;
        destPoint_ = line.p2() - edgeOffset;
    } else {
        sourcePoint_ = destPoint_ = line.p1();
```

```
    }
}

QPainterPath CGraphicsEdge::shape() const
{
    QPainterPath path;
    path.moveTo(sourcePoint_);
    path.lineTo(destPoint_);
    return path;
}

QRectF CGraphicsEdge::boundingRect() const
{
    if (!parent_ || !child_)
        return QRectF();

    qreal penWidth = 1;
    qreal extra = (penWidth + arrowSize_) / 2.0;

    return QRectF(sourcePoint_, QSizeF(destPoint_.x() - sourcePoint_.x(), destPoint_.y() -
sourcePoint_.y()))
        .normalized()
        .adjusted(-extra, -extra, extra, extra);
}

void CGraphicsEdge::paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
QWidget *widget)
{
    Q_UNUSED(option);
    Q_UNUSED(widget);
    if (!parent_ || !child_)
        return;

    QLineF line(sourcePoint_, destPoint_);
    if (qFuzzyCompare(line.length(), qreal(0.)))
        return;

    // Draw the line itself
    painter->setPen(QPen(Qt::black, 1, Qt::SolidLine, Qt::RoundCap, Qt::RoundJoin));
    painter->drawLine(line);

    // Draw the arrows
    double angle = ::acos(line.dx() / line.length());
    if (line.dy() >= 0)
        angle = TwoPi - angle;

    QPointF destArrowP1 = destPoint_ + QPointF(sin(angle - Pi / 3) * arrowSize_,
                                    cos(angle - Pi / 3) * arrowSize_);
    QPointF destArrowP2 = destPoint_ + QPointF(sin(angle - Pi + Pi / 3) * arrowSize_,
                                    cos(angle - Pi + Pi / 3) * arrowSize_);

    painter->setBrush(Qt::black);
```

```
    painter->drawPolygon(QPolygonF() << line.p2() << destArrowP1 << destArrowP2);
}
#include "cequaldistributionheuristic.h"

#include <qvector.h>

#include <math.h>

QMap<int, QList<SSigNode*> >
CEqualDistributionHeuristic::assignLayers(QList<SSigNode*>& nodes)
{
    if (nodes.isEmpty()) return QMap<int, QList<SSigNode*> >();

    // Создаем граф, аналогичный графу составленному из SSigNode'ов (но он без циклов)
    CGraph* graph = new CGraph();
    for(int i = 0; i < nodes.size(); ++i)
        graph->addNode();

    for(int i = 0; i < nodes.size(); ++i) {
        for(int j = 0; j < nodes.at(i)->children_.size(); ++j)
            graph->addEdge(i, nodes.indexOf(nodes.at(i)->children_.at(j)));
    }

    // Создаем инвертированный граф, для вычисления высот вершин
    inverted_ = new CGraph();
    for(int i = 0; i < graph->nodeCount(); ++i)
        inverted_->addNode();

    for(int i = 0; i < graph->nodeCount(); ++i) {
        CNode* vi = graph->node(i);
        for(int j = 0; j < vi->childCount(); ++j) {
            inverted_->addEdge(graph->nodeIndex(vi->child(j)), i);
        }
    }

    // Распределяем по-уровням
    QHash<CNode*, int> layers = embedGraph(graph);

    delete inverted_;
    inverted_ = NULL;

    invertedGraphSorted_.clear();

    QMap<int, QList<SSigNode*> > info;
    for(auto it = layers.begin(); it != layers.end(); ++it) {
        int nodeIdx = graph->nodeIndex(it.key());
        info[it.value()].push_back(nodes[nodeIdx]);
    }

    delete graph;

    return info;
```

```
}

QHash<CNode *, int> CEqualDistributionHeuristic::embedGraph(CGraph *g)
{
   // Обнуляем переменные
   layers_.clear();

   // Кэшируем топологический порядок вершин обычного графа и инвертированного
   graphSorted_ = g->topologicalSorted();
   invertedGraphSorted_ = inverted_->topologicalSorted();

   embedLongestPath(g);

   QList<CNode*> v_ = calcVSet();
   while(v_.size() != g->nodeCount()) {
      QHash<CNode*, int> d = calculateDepths(graphSorted_, v_);

      QHash<CNode*, int> h = calculateHeights(g, v_);

      QList<CNode*> path = findLongestPath(g, v_, d, h);

      embedPath(g, path);
      v_ = calcVSet();
   }

   return layers_;
}

void CEqualDistributionHeuristic::embedLongestPath(CGraph *g)
{
   QHash<CNode*, int> d = calculateDepths(graphSorted_, QList<CNode*>());
   QHash<CNode*, int> h = calculateHeights(g, QList<CNode*>());

   int k = 0;
   for(auto it = d.begin(); it != d.end(); ++it)
      k = qMax(k, d[it.key()] + h[it.key()]);

   for(int i = 0; i < g->nodeCount(); ++i) {
      CNode* v = g->node(i);
      int d_ = d[v], h_ = h[v];
      layers_[v] = (d_ + h_ == k) ? d_ + 1 : 0;
   }
}

QHash<CNode *, int> CEqualDistributionHeuristic::calculateDepths(const QList<CNode*>&
sorted, const QList<CNode *> &v_)
{
   QHash<CNode*, int> d_;

   for(int i = 0; i < sorted.size(); ++i)
      d_[sorted.at(i)] = 0;
```

```
    for(int i = 0; i < sorted.size(); ++i) {
        CNode* vi = sorted.at(i);
        for(int j = 0; j < vi->childCount(); ++j) {
            CNode* vj = vi->child(j);
            if (v_.indexOf(vj) != -1) continue;

            if (d_[vj] < d_[vi] + 1)
                d_[vj] = d_[vi] + 1;
        }
    }

    return d_;
}

QHash<CNode *, int> CEqualDistributionHeuristic::calculateHeights(CGraph *g, const
QList<CNode *> &v_)
{
    // Создаем CDFSTree*' для инвертированного графа, аналогичный v' граф g
    QList<CNode*> _v;
    for(int i = 0; i < v_.size(); ++i)
        _v.push_back(inverted_->node(g->nodeIndex(v_.at(i))));

    QHash<CNode*, int> h = calculateDepths(invertedGraphSorted_, _v);
    QHash<CNode*, int> h_;
    for(auto it = h.begin(); it != h.end(); ++it)
        h_[g->node(inverted_->nodeIndex(it.key()))] = it.value();

    return h_;
}

CNode* findWithParentInList(CNode* n, const QList<CEdge*>& l)
{
    for(int i = 0; i < l.size(); ++i)
        if (l.at(i)->parent() == n)
            return l.at(i)->child();

    return NULL;
}

QList<CNode*> CEqualDistributionHeuristic::findLongestPath(CGraph* g, const
QList<CNode*>& v_, const QHash<CNode*, int>& d, const QHash<CNode*, int>& h)
{
    QList<CNode*> v_sub_vprime;
    for(int i = 0; i < g->nodeCount(); ++i) {
        if (v_.contains(g->node(i))) continue;
        v_sub_vprime.push_back(g->node(i));
    }

    int m = 0;
    for(int i = 0; i < v_sub_vprime.size(); ++i)
        m = qMax(m, d[v_sub_vprime.at(i)] + h[v_sub_vprime.at(i)]);
```

```
QList<CNode*> w;
for(int i = 0; i < v_sub_vprime.size(); ++i) {
   if ((d[v_sub_vprime.at(i)] + h[v_sub_vprime.at(i)]) == m)
      w.push_back(v_sub_vprime.at(i));
}

QList<CEdge*> f1;
for(int i = 0; i < v_.size(); ++i) {
   CNode* u = v_.at(i);
   for(int j = 0; j < u->childCount(); ++j) {
      CNode* v = u->child(j);
      if (d[v] + h[v] == m)
         f1.push_back(g->edge(u, v));
   }
}

QList<CEdge*> f2;
for(int i = 0; i < w.size(); ++i){
   for(int j = 0; j < w.size(); ++j) {
      CEdge* e = g->edge(w.at(i), w.at(j));
      if (!e) continue;
      if (d[e->child()] == d[e->parent()] + 1)
         f2.push_back(e);
   }
}

QList<CEdge*> f3;
for(int i = 0; i < w.size(); ++i) {
   CNode* u = w.at(i);
   for(int j = 0; j < u->childCount(); ++j) {
      CNode* v = u->child(j);
      if (v_.indexOf(v) == -1) continue;
      if (d[u] + h[u] == m)
         f3.push_back(g->edge(u, v));
   }
}

QList<CNode*> path;

path.push_back(f1.front()->parent());
path.push_back(f1.front()->child());

CNode* v = path.back();
while (true) {
   v = findWithParentInList(v, f2);
   if (v != NULL)
      path.push_back(v);
   else
      break;
}

for(int i = 0; i < f3.size(); ++i) {
```

```
        if (f3.at(i)->parent() == path.back()) {
            path.push_back(f3.at(i)->child());
            break;
        }
    }

    return path;
}

void CEqualDistributionHeuristic::embedPath(CGraph *g, const QList<CNode *> &path)
{
    Q_UNUSED(g);

    if (path.isEmpty()) return;

    int i = layers_[path.front()];
    int j = layers_[path.back()];

    Q_ASSERT(i != 0);
    Q_ASSERT(j != 0);

    int k_1 = path.size() - 1;

    for(int m = 1; m < k_1; ++m) {
        CNode* v = path.at(m);

        int n1;
        if ( (j - i) % k_1 == 0 ) {
            int p = (j - i) / k_1;
            n1 = i + m * p;
        }
        else {
            int p = (j - i) / k_1;
            int x = (p + 1) * k_1 + i - j;

            if (m <= x)
                n1 = i + m * p;
            else {
                n1 = i + x * p + (m - x) * (p + 1);
                if (n1 == i) n1 = i + m * (p + 1);
            }
        }

        int n2 = INT_MAX;
        for(int jj = 0; jj < v->childCount(); ++jj) {
            if (layers_[v->child(jj)] != 0)
                n2 = qMin(n2, layers_[v->child(jj)]);
        }

        layers_[v] = qMin(n1, n2 - 1);

        for(int r = m; r >= 2; --r) {
```

```
            if (layers_[path.at(r)] > layers_[path.at(r - 1)])
                layers_[path.at(r - 1)] = layers_[path.at(r)] - 1;
        }
    }
}

QList<CNode *> CEqualDistributionHeuristic::calcVSet()
{
    QList<CNode*> v_;
    for(auto it = layers_.begin(); it != layers_.end(); ++it)
        if (it.value() != 0) v_.push_back(it.key());
    return v_;
}
#include "cforcedirectedlayout.h"

#include <math.h>

float CForceDirectedLayout::f_a(float x) const
{
    float k = sqrt(area_ / nodes_.size());
    return x*x/k;
}

float CForceDirectedLayout::f_r(float x) const
{
    if (!nodes_.size()) return 0;

    float k = sqrt(area_ / nodes_.size());
    return k*k/x;
}

float CForceDirectedLayout::norm(const QPointF& p) const
{
    return p.isNull() ? 1 : sqrt(p.x() * p.x() + p.y() * p.y());
}

void CForceDirectedLayout::calcRepulsiveForces()
{
    for(int i = 0; i < nodes_.size(); ++i)
    {
        SNode* v = &nodes_[i];
        disp_[v->node_] = QPointF();

        for(int j = 0; j < nodes_.size(); ++j)
        {
            if ( i == j ) continue;

            SNode* u = &nodes_[j];
            QPointF d = v->pos_ - u->pos_;

            float dn = norm(d);
            disp_[v->node_] += (d/dn) * f_r(dn);
```

```
        }
    }
}

void CForceDirectedLayout::calcAttractiveForces(CGraph* g)
{
    for(int i = 0; i < g->edgeCount(); ++i)
    {
        CEdge* e = g->edge(i);
        int parentIdx = g->nodeIndex(e->parent());
        int childIdx = g->nodeIndex(e->child());

        QPointF d = nodes_.at(parentIdx).pos_ - nodes_.at(childIdx).pos_;
        float dn = norm(d);
        disp_[e->parent()] -= (d/dn) * f_a(dn);
        disp_[e->child()] += (d/dn) * f_a(dn);
    }
}

void CForceDirectedLayout::applyCorrections()
{
    // Корректируем позиции
    for(int i = 0; i < nodes_.size(); ++i)
    {
        SNode* v = &nodes_[i];

        QPointF p = v->pos_ + disp_[v->node_] / norm(disp_[v->node_]);
        /*p.setX(qMin(w_/2, qMax(-w_/2, p.x())));
        p.setY(qMin(h_/2, qMax(-h_/2, p.y())));*/
        v->pos_ = p;
    }
}

QMap<CNode *, QPoint> CForceDirectedLayout::doLayout(CGraph* g)
{
    disp_.clear();

    if (!g) return QMap<CNode*, QPoint>();

    // Заполняем вектор узлов
    nodes_ = QVector<SNode>(g->nodeCount());
    for(int i = 0; i < nodes_.size(); ++i) {
        nodes_[i].node_ = g->node(i);
        nodes_[i].pos_.rx() = rand() % (int)w_;
        nodes_[i].pos_.ry() = rand() % (int)h_;
    }

    // Алгоритм укладки
    for(int iter = 0; iter < iters_; ++iter)
    {
        calcRepulsiveForces();
        calcAttractiveForces(g);
```

```
        applyCorrections();
    }

    // Выдаем конечный результат
    QMap<CNode*, QPoint > res;
    for(int i = 0; i < nodes_.size(); ++i)
        res[nodes_.at(i).node_] = nodes_.at(i).pos_.toPoint();
    return res;
}
#include "cgraph.h"

#include <qdatastream.h>

CGraph::CGraph()
    : QObject()
{

}

CGraph::~CGraph()
{
    qDeleteAll(nodes_);
    qDeleteAll(edges_);
    nodes_.clear();
    edges_.clear();
}

CNode *CGraph::addNode()
{
    CNode* node = new CNode();
    node->graph_ = this;
    nodes_.push_back(node);
    emit nodeAdded();
    return node;
}

CEdge *CGraph::addEdge(int i, int j)
{
    CNode* p = nodes_.at(i);
    CNode* c = nodes_.at(j);

    CEdge* e = new CEdge(p, c);
    p->children_.push_back(c);
    c->parents_.push_back(p);

    e->graph_ = this;
    edges_.push_back(e);
    emit edgeAdded();
    return e;
}

CEdge *CGraph::edge(CNode *u, CNode *v) const
```

```
{
   if (u->children_.contains(v) == false) return NULL;

   // TODO: Рефактор
   for(int i = 0; i < edges_.size(); ++i)
      if (edges_.at(i)->parent() == u && edges_.at(i)->child() == v) return edges_.at(i);
   return NULL;
}

void visit(QList<CNode*>& l, QSet<CNode*>& marked, CNode* v)
{
   if (marked.contains(v)) return;
   marked.insert(v);
   for(int i = 0; i < v->childCount(); ++i)
      visit(l, marked, v->child(i));
   l.push_front(v);
}

QList<CNode *> CGraph::topologicalSorted() const
{
   QList<CNode*> l;
   QSet<CNode*> marked;

   // Сначала пройдем вершины без родителей
   for(int i = 0; i < nodes_.size(); ++i)
      if (nodes_.at(i)->parentCount() == 0 && marked.contains(nodes_.at(i)) == false)
         visit(l, marked, nodes_.at(i));

   // Потом остальные
   for(int i = 0; i < nodes_.size(); ++i) {
      if (marked.contains(nodes_.at(i))) continue;
      visit(l, marked, nodes_.at(i));
   }

   return l;
}

CGraph *CGraph::generateAccyclicGraph(int nodes, double density)
{
   CGraph* g = new CGraph();

   for(int i = 0; i < nodes; ++i)
      g->addNode();

   for(int i = 0; i < nodes; ++i)
      for(int j = 0; j < i; ++j)
         g->addEdge(i, j);
   return g;
}

QByteArray CGraph::serialize() const
{
```

```
    QByteArray arr;
    QDataStream stream(&arr, QIODevice::WriteOnly);

    stream << nodes_.size();
    stream << edges_.size();
    for(int i = 0; i < edges_.size(); ++i) {
        stream << nodes_.indexOf(edges_.at(i)->parent());
        stream << nodes_.indexOf(edges_.at(i)->child());
    }

    return qCompress(arr, 9);
}

void CGraph::deserialize(const QByteArray &arr)
{
    qDeleteAll(nodes_);
    qDeleteAll(edges_);
    nodes_.clear();
    edges_.clear();

    QByteArray uncomp = qUncompress(arr);
    QDataStream stream(&uncomp, QIODevice::ReadOnly);

    int nodeCnt; stream >> nodeCnt;
    int edgeCnt; stream >> edgeCnt;

    for(int i = 0; i < nodeCnt; ++i)
        addNode();

    for(int i = 0; i < edgeCnt; ++i) {
        int p; stream >> p;
        int c; stream >> c;

        addEdge(p, c);
    }
}

#include "cgraphicsnode.h"

#include "cedge.h"

#include <qpainter.h>
#include <qpainterpath.h>
#include <qstyleoption.h>

CGraphicsNode::CGraphicsNode()
    : QGraphicsItem()
{
    setFlag(ItemIsMovable);
    setFlag(ItemSendsGeometryChanges);
    setCacheMode(DeviceCoordinateCache);
    setZValue(-1);
```

```
}

void CGraphicsNode::addEdge(CGraphicsEdge* edge)
{
   edges_ << edge;
   edge->adjust();
}

QRectF CGraphicsNode::boundingRect() const
{
   qreal adjust = 2;
   return QRectF( -10 - adjust, -10 - adjust,
      23 + adjust, 23 + adjust);
}

QPainterPath CGraphicsNode::shape() const
{
   QPainterPath path;
   path.addEllipse(-10, -10, 20, 20);
   return path;
}

void CGraphicsNode::paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
QWidget *widget)
{
   painter->setPen(Qt::NoPen);
   painter->setBrush(Qt::darkGray);
   painter->drawEllipse(-7, -7, 20, 20);

   //QRadialGradient gradient(-3, -3, 10);
   //if (option->state & QStyle::State_Sunken) {
   //    gradient.setCenter(3, 3);
   //    gradient.setFocalPoint(3, 3);
   //    gradient.setColorAt(1, QColor(Qt::blue).light(120));
   //    gradient.setColorAt(0, QColor(Qt::darkYellow).light(120));
   //} else {
   //    gradient.setColorAt(0, Qt::yellow);
   //    gradient.setColorAt(1, Qt::darkYellow);
   //}
   painter->setBrush(Qt::blue);

   painter->setPen(QPen(Qt::black, 0));
   painter->drawEllipse(-10, -10, 20, 20);
}

QVariant CGraphicsNode::itemChange(QGraphicsItem::GraphicsItemChange change, const
QVariant &value)
{
   switch (change) {
   case ItemPositionHasChanged:
      foreach (CGraphicsEdge* edge, edges_)
         edge->adjust();
```

```
      break;
   default:
      break;
   };

   return QGraphicsItem::itemChange(change, value);
}

void CGraphicsNode::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
   update();
   QGraphicsItem::mousePressEvent(event);
}

void CGraphicsNode::mouseReleaseEvent(QGraphicsSceneMouseEvent *event)
{
   update();
   QGraphicsItem::mouseReleaseEvent(event);
}

#include "cgraphview.h"

#include "cgraphicsnode.h"
#include "cedge.h"

#include <qpen.h>
#include <qbrush.h>

#include <qgraphicssceneevent.h>
#include <qgraphicsitem.h>
#include <qimage.h>

#include <qdebug.h>

CGraphView::CGraphView(QWidget *parent)
   : QGraphicsView(parent)
   , firstSelectedNode_(0)
{
   setScene(new QGraphicsScene(this));
   scene()->installEventFilter(this);
   setSceneRect(-INT_MAX / 2, -INT_MAX / 2, INT_MAX, INT_MAX);

   setDragMode(QGraphicsView::ScrollHandDrag);
   setTransformationAnchor(QGraphicsView::AnchorUnderMouse);

   setHorizontalScrollBarPolicy( Qt::ScrollBarAlwaysOff );
   setVerticalScrollBarPolicy( Qt::ScrollBarAlwaysOff );
}

void CGraphView::setGraph(CGraph *g)
{
   qDeleteAll(nodes_);
```

```
    qDeleteAll(edges_);
    nodes_.clear();
    edges_.clear();

    graph_.reset(g);

    if (graph_.isNull()) return;

    for(int i = 0; i < graph_->nodeCount(); ++i) {
        nodes_.push_back(new CGraphicsNode());
        scene()->addItem(nodes_.back());
    }

    for(int i = 0; i < graph_->edgeCount(); ++i) {
        int parentIdx = graph_->nodeIndex(graph_->edge(i)->parent());
        int childIdx = graph_->nodeIndex(graph_->edge(i)->child());
        edges_.push_back(new CGraphicsEdge(nodes_.at(parentIdx), nodes_.at(childIdx)));
        scene()->addItem(edges_.back());
    }
}

void CGraphView::saveToFile(const QString &filename)
{
    if (!scene()) return;

    QImage image(sceneRect().size().toSize(), QImage::Format_ARGB32);
    QPainter p(&image);
    scene()->render(&p);
    image.save(filename);
}

void CGraphView::wheelEvent(QWheelEvent *event)
{
    // Scale the view / do the zoom
    const double scaleFactor = 1.15;

    if( event->delta() > 0 ) // Zoom in
        scale(scaleFactor, scaleFactor);
    else // Zooming out
        scale(1.0 / scaleFactor, 1.0 / scaleFactor);
}

bool CGraphView::eventFilter(QObject *pObj, QEvent *pEv)
{
    if ( pObj != scene() )
        return QGraphicsView::eventFilter(pObj, pEv);

    QGraphicsSceneMouseEvent* pMouseEv = dynamic_cast<QGraphicsSceneMouseEvent*>(
pEv );
    if ( !pMouseEv )
        return QGraphicsView::eventFilter(pObj, pEv);
```

```
    if (pMouseEv->type() == QEvent::GraphicsSceneMousePress) {
        if (mode_ == EPointerMode::eNodeAdding) {
            addNode(pMouseEv->scenePos());
        } else if (mode_ == EPointerMode::eEdgeAdding) {
            if (firstSelectedNode_ != NULL) {
                CGraphicsNode* secondNode = getItemAt<CGraphicsNode>(pMouseEv-
>scenePos());
                if (secondNode == NULL) return QGraphicsView::eventFilter(pObj, pEv);

                CGraphicsNode* firstNode = firstSelectedNode_;
                firstSelectedNode_ = NULL;

                if (firstNode && secondNode) {
                    int parentIdx = nodes_.indexOf(firstNode);
                    int childIdx = nodes_.indexOf(secondNode);
                    if (parentIdx != childIdx) addEdge(parentIdx, childIdx);
                }

                return QGraphicsView::eventFilter(pObj, pEv);
            }

            firstSelectedNode_ = getItemAt<CGraphicsNode>(pMouseEv->scenePos());
        }
    }

    return QGraphicsView::eventFilter(pObj, pEv);
}

void CGraphView::setPointerMode(CGraphView::EPointerMode mode)
{
    mode_ = mode;
    firstSelectedNode_ = NULL;
}

QByteArray CGraphView::serialize() const
{
    QByteArray arr;
    QDataStream stream(&arr, QIODevice::WriteOnly);

    stream << nodes_.size();
    for(int i = 0; i < nodes_.size(); ++i)
        stream << nodes_.at(i)->scenePos();

    return qCompress(arr, 9);
}

void CGraphView::deserialize(const QByteArray& arr)
{
    QByteArray uncomp = qUncompress(arr);
    QDataStream stream(&uncomp, QIODevice::ReadOnly);

    int nodesSize; stream >> nodesSize;
```

```
    if (nodes_.size() != nodesSize) return;
    for(int i = 0; i < nodes_.size(); ++i) {
        QPointF p; stream >> p;
        nodes_.at(i)->setPos(mapFromScene(p));
    }
}

void CGraphView::setEdgeParent(CGraphicsEdge *e, CNode *p)
{
    if (graph_.isNull()) return;

    CNode* oldP = graph_->node(nodes_.indexOf(e->parent()));
    CNode* oldC = graph_->node(nodes_.indexOf(e->child()));

    CEdge* edge = graph_->edge(oldP, oldC);
}

CNode* CGraphView::addNode(const QPointF &pos)
{
    if (graph_.isNull()) return NULL;

    CNode* n = graph_->addNode();
    nodes_.push_back(new CGraphicsNode());
    nodes_.back()->setPos(pos);
    nodes_.back()->setFlag(QGraphicsItem::ItemIsSelectable);
    nodes_.back()->setFlag(QGraphicsItem::ItemIsMovable);

    scene()->addItem(nodes_.back());
    return n;
}

void CGraphView::addEdge(int parentIdx, int childIdx)
{
    if (graph_.isNull()) return;

    CEdge* edge = graph_->addEdge(parentIdx, childIdx);

    edges_.push_back(new CGraphicsEdge(nodes_.at(parentIdx), nodes_.at(childIdx)));
    //edges_.back()->setFlag(QGraphicsItem::ItemIsSelectable);

    scene()->addItem(edges_.back());
}

#include "cplanartesting.h"

/*
 */
CDFSTree::CDFSTree() : node_(0), parent_(0) , leastAncestor_(0)
    , lowpoint_(0), dfi_(0), dfsChildren_(), children_(), backedges_()
{
}
```

```
QList<CDFSTree*> CDFSTree::orderedList() const
{
   QList<CDFSTree*> ret;
   for (int i = 0; i < dfsChildren_.size(); ++i)
      ret.append(dfsChildren_.at(i)->orderedList());
   ret.push_back(const_cast<CDFSTree*>(this));
   return ret;
}

/*CDFSTree* CDFSTree::formDFSTree(CGraph* g, QMap<CNode*, CDFSTree*>& mapped)
{
   if (!g || g->nodeCount() == 0) return NULL;

   QList<CNode*> ordered;
   for(int i = 0; i < g->nodeCount(); ++i) {
      if (g->node(i)->parentCount() == 0)
         ordered.push_front(g->node(i));
      else
         ordered.push_back(g->node(i));
   }

   CDFSTree* root = new CDFSTree();

   QSet<CNode*> visited;
   for (int i = 0; i < ordered.size(); ++i) {
      CNode* v = ordered.at(i);
      if (visited.contains(v))
         continue;

      int dfi = 0;
      visit(v, 1, dfi, visited, mapped);
   }

   for (auto it = mapped.begin(); it != mapped.end(); ++it) {
      if ((*it)->parent_ != NULL) continue;
      root->children_.push_back(*it);
      (*it)->parent_ = root;
   }

   return root;
}*/

CDFSTree* CDFSTree::createDfsTree(CNode* v, int depth, QSet<CNode*>& visited,
QMap<CNode*, CDFSTree*>& mapped)
{
   visited.insert(v);

   CDFSTree* node = new CDFSTree();
   node->node_ = v;
   node->depth_ = depth;
   node->lowpoint_ = depth;
   mapped[v] = node;
```

```
    for (int i = 0; i < v->childCount(); ++i) {
        CNode* c = v->child(i);
        if (!visited.contains(c)) {
            // parent[ni] = i;
            CDFSTree* treeChild = createDfsTree(c, depth + 1, visited, mapped);
            node->lowpoint_ = qMin(node->lowpoint_, treeChild->lowpoint_);
            treeChild->parent_ = node;
            node->children_.push_back(treeChild);
            node->dfsChildren_.push_back(treeChild);
        }
        else {
            CDFSTree* treeChild = mapped[c];
            if (node->lowpoint_ > treeChild->depth_) {
                node->lowpoint_ = treeChild->depth_;
                node->backedges_.push_back(treeChild);
                node->children_.push_back(treeChild);
            }
        }
    }

    return node;
}

/*void CDFSTree::visit(CNode* v, int depth, int& dfi, QSet<CNode*>& visited,
QMap<CNode*, CDFSTree*>& mapped)
{
    CDFSTree* tree = new CDFSTree();
    tree->node_ = v;
    tree->depth_ = depth;
    tree->lowpoint_ = depth;
    mapped[v] = tree;
    visited.insert(v);

    for (int i = 0; i < v->childCount(); ++i) {
        CNode* c = v->child(i);
        if (visited.contains(c)) {
            CDFSTree* childTree = mapped[c];

            if (tree->lowpoint_ > childTree->depth_) {
                tree->lowpoint_ = childTree->depth_;
                tree->backedges_.push_back(childTree);
                tree->leastAncestor_ = childTree;
            }

            continue;
        }

        visit(c, depth + 1, dfi, visited, mapped);

        CDFSTree* childTree = mapped[c];
        childTree->parent_ = tree;
```

```
    childTree->dfi_ = dfi;
    ++dfi;

    tree->lowpoint_ = qMin(tree->lowpoint_, childTree->lowpoint_);
    tree->children_.push_back(childTree);
  }
}*/

void CDFSTree::calcLeastAncestor()
{
  for (int i = 0; i < backedges_.size(); ++i) {
    CDFSTree* other = backedges_.at(i);
    if (dfi_ > other->dfi_ && (leastAncestor_ == NULL || other->dfi_ < leastAncestor_->dfi_))
      leastAncestor_ = other;
  }
}

bool CDFSTree::pathToUtil(CDFSTree* w, QStack<CDFSTree*>& stack,
QList<CDFSTree*>& path)
{
  if (stack.top() == w) {
    while (stack.isEmpty() == false)
      path.push_front(stack.pop());
    return true;
  }

  CDFSTree* t = stack.top();
  for (int i = 0; i < t->children_.size(); ++i) {
    stack.push(t->children_.at(i));
    if (pathToUtil(w, stack, path))
      return true;

    stack.pop();
  }

  return false;
}



/*
 */
CBlock::CBlock() : root_(0), vertices_(), edges_(), bound_() {}

void CBlock::setExternalFace(CDFSTree* v, int dir)
{
  QList<CDFSTree*> list = bound_;
  int idx = bound_.indexOf(v);

  bound_.clear();
  bound_.push_back(list.front());
```

```
    if (dir == 1) {
        for (int i = 1; i <= idx; ++i)
            bound_.push_back(list.at(i));
        return;
    }

    for (int i = list.size() - 1; i >= idx; --i)
        bound_.push_back(list.at(i));
}



/*
 */
bool CPlanarTester::testPlanar(CGraph* g)
{
    QMap<CNode*, CDFSTree*> mapped;
    //dfsTree_ = CDFSTree::formDFSTree(g, mapped);
    QSet<CNode*> visited;

    dfsTree_ = new CDFSTree();
    for(int i = 0; i < g->nodeCount(); ++i) {
        if (visited.contains(g->node(i)) == false)
            dfsTree_->addChild(CDFSTree::createDfsTree(g->node(i), 0, visited, mapped));
    }

    for (int i = 0; i < g->nodeCount(); ++i) {
        CDFSTree* v = mapped[g->node(i)];
        QList<CDFSTree*> separatedDFSChildList;
        auto children = v->children();
        for (int j = 0; j < children.size(); ++j){
            CDFSTree* u = children.at(j);
            separatedDFSChildList.push_back(u);
            mapLowpoint_[u] = u->lowpoint();
        }

        qSort(separatedDFSChildList.begin(), separatedDFSChildList.end(),
&CDFSTree::lowpointLessThan);
        mapChildrenList_[v] = separatedDFSChildList;
    }

    ordered_ = dfsTree_->orderedList();
    for (int i = 0; i < ordered_.size(); ++i) {
        ordered_.at(i)->calcLeastAncestor();
    }

    for (int i = 0; i < ordered_.size(); ++i){
        CDFSTree* v = ordered_.at(i);
        auto children = v->children();
        for (int j = 0; j < children.size(); ++j)
            createBlock(v, children.at(j));
```

```
        if (!task(v)) return false;
    }

    return true;
}

bool CPlanarTester::task(CDFSTree* v)
{
    auto children = v->children();
    auto backedges = v->backedges();
    if (children.isEmpty() && backedges.isEmpty()) return true;

    for (int i = 0; i < children.size(); ++i) {
        CDFSTree* c = children.at(i);
        QList<QPair<CDFSTree*, CDFSTree*>> edges;
        for (int j = 0; j < backedges.size(); ++j) {
            CDFSTree* dest = backedges.at(j);
            if (c->orderedList().contains(dest))
                edges.push_back(qMakePair(v, dest));
        }

        if (edges.isEmpty()) continue;

        endpoins_.clear();
        extAct_ = externalyActive(v);

        for (int j = 0; j < edges.size(); ++j)
            endpoins_.push_back(edges.at(j).second);

        for (int j = 0; j < edges.size(); ++j)
            walkup(v, edges.at(j));

        while (edges.isEmpty() == false) {
            if (!walkdown(v, edges.front(), edges))
                return false;

            edges.pop_front();
        }
    }

    return true;
}

void CPlanarTester::walkup(CDFSTree* v, const QPair<CDFSTree*, CDFSTree*>& backedge)
{
    QList<CDFSTree*> path = v->pathTo(backedge.first);

    QMap<CDFSTree*, CBlock*> pertinentBlocksMap;
    for (int i = 0; i < path.size(); ++i) {
        CDFSTree* vv = path.at(i);
        QList<CBlock*>& blocks = mapBlocks_[vv];
```

```
        /*if (blocks.isEmpty() || vv != path.front())
            continue;*/

        for (int j = 0; j < blocks.size(); ++j) {
            CBlock* b = blocks.at(j);

            bool flag = false;
            for (int k = 0; k < vv->children().size(); ++k) {
                CDFSTree* child = vv->children().at(k);
                if (path.indexOf(child) != -1 && b->vertices().indexOf(child) != -1) {
                    flag = true;
                    break;
                }
            }

            if (!flag) continue;

            pertinentBlocksMap[vv] = b;
            break;
        }
    }

    mapPertinent_[backedge] = pertinentBlocksMap;
}

bool CPlanarTester::walkdown(CDFSTree *v, const QPair<CDFSTree *, CDFSTree *>
&backEdge,
                     QList<QPair<CDFSTree *, CDFSTree *> > &backEdges)
{
    CBlock* cur = mapPertinent_[backEdge][v];
    Q_ASSERT(cur != NULL);

    int curDir = 1;

    bool done = false;

    CDFSTree* endpoint = backEdge.second;
    QList<CDFSTree*> vertList;
    QList<CBlock*> blocksWithPoints = blockWith(endpoint);
    QList<CBlock*> blocks;

    while (!done) {
        CBlock* block = NULL;
        vertList = QList<CDFSTree*>();

        blocks.push_back(cur);

        int idx = 0;
        if (curDir == 0) idx = cur->boundaryVertices().size() - 1;

        bool flip = false;
        for(;;) {
```

```
    CDFSTree* curBound = cur->boundaryVertices().at(idx);
    vertList.push_back(curBound);

    if (blocksWithPoints.indexOf(cur) == -1) {
        if (mapPertinent_[backEdge][curBound] != NULL &&
                mapPertinent_[backEdge][curBound] != cur){
            block = mapPertinent_[backEdge][curBound];
            break;
        }
    }

    if (curBound == endpoint) {
        done = true;
        break;
    }

    if (extAct_.indexOf(curBound) != -1) {
        flip = true;
        break;
    }

    if (curDir == 1 && (++idx == cur->boundaryVertices().size()))
        break;
    else if (--idx == 0) break;
}

if (flip) {
    curDir = !curDir;

    idx = 1;
    if (curDir == 0)
        idx = cur->boundaryVertices().size() -1;

    vertList.clear();

    for(;;) {
        CDFSTree* curBound = cur->boundaryVertices().at(idx);

        vertList.push_back(curBound);
        if (!blocksWithPoints.contains(cur) &&
                mapPertinent_[backEdge].contains(curBound) &&
                mapPertinent_[backEdge][curBound] != cur) {
            block = mapPertinent_[backEdge][curBound];
            break;
        }

        if (curBound == endpoint) {
            done = true;
            break;
        }

        if (extAct_.contains(curBound))
```

```
            return false;

        if (curDir == 1 && (++idx == cur->boundaryVertices().size()))
            break;
        else if (--idx == 0)
            break;
    }
}

CDFSTree* l = vertList.at(vertList.size() - 1);

if (flip) cur->setExternalFace(l, !curDir);
else {
    int endIdx = cur->boundaryVertices().
            indexOf(vertList.at(vertList.size() - 1));

    bool extActive = false;
    bool toEmbed = false;
    bool pertinent = false;

    if (curDir == 1) {
        for (int i = cur->boundaryVertices().size() - 1; i > endIdx; --i) {
            CDFSTree* vert = cur->boundaryVertices().at(i);
            if (extAct_.indexOf(vert) != -1){
                extActive = true;
                break;
            }

            if (endpoins_.indexOf(vert) != -1){
                toEmbed = true;
                break;
            }

            pertinent = (mapBlocks_.contains(vert) &&
                    mapBlocks_[vert].size() > 0);
        }
    } else {
        for (int i = 1; i < endIdx; --i) {
            CDFSTree* vert = cur->boundaryVertices().at(i);
            if (extAct_.contains(vert)){
                extActive = true;
                break;
            }

            if (endpoins_.contains(vert)) {
                toEmbed = true;
                break;
            }

            pertinent = (mapBlocks_.contains(vert) &&
                    mapBlocks_[vert].size() > 0);
        }
```

```
            }

            flip = (extActive || toEmbed /*&& pertinent*/);

            if (!flip){
                bool traversedPertinent = false;
                for(CDFSTree* vert : vertList){
                    if (vert == l)
                        break;
                    if (mapBlocks_.contains(vert) && mapBlocks_[vert].size() > 0){
                        traversedPertinent = true;
                        break;
                    }
                }
                if (!traversedPertinent && pertinent) flip = true;
            }

            int toSet = curDir;
            if (flip) toSet = !curDir;
            cur->setExternalFace(l, toSet);
        }

        cur = block;
    }

    if (blocks.size() > 1) {
        CBlock* newBlock = merge(blocks);
        newBlock->addEdge(backEdge);
        endpoins_.removeOne(endpoint);

        for (int j = 0; j < backEdges.size(); ++j) {
            QPair<CDFSTree*, CDFSTree*> edge = backEdges.at(j);
            QMap<CDFSTree*, CBlock*> pertinent = mapPertinent_[edge];
            auto entries = pertinent.begin();
            auto end = (--pertinent.end());
            bool replaced = false;
            while (entries != end){
                auto entry = entries;
                ++entry;

                if (blocks.contains(entry.value())) {
                    pertinent.erase(entries);
                    replaced = true;
                }
            }

            if (replaced) pertinent[newBlock->root()] = newBlock;
        }
    }

    mapPertinent_.remove(backEdge);
    backEdges.removeOne(backEdge);
```

```
    return true;
}

QList<CBlock*> CPlanarTester::blockWith(CDFSTree* v)
{
    QList<CBlock*> ret;
    for (int i = 0; i < blocks_.size(); ++i) {
        if (blocks_.at(i)->vertices().indexOf(v) != -1)
            ret.push_back(blocks_.at(i));
    }

    return ret;
}

QList<CDFSTree*> CPlanarTester::outsideFace()
{
    QList<CDFSTree*> ret;
    for (int i = 0; i < blocks_.size(); ++i) {
        for (int j = 0; j < blocks_.at(i)->boundaryVertices().size(); ++j) {
            CDFSTree* vb = blocks_.at(i)->boundaryVertices().at(j);
            if (ret.indexOf(vb) == -1) ret.push_back(vb);
        }
    }
    return ret;
}

QList<CDFSTree*> CPlanarTester::externalyActive(CDFSTree* v)
{
    QList<CDFSTree*> ret;

    auto allChildren = v->orderedList();
    for (int i = 0; i < allChildren.size(); ++i) {
        CDFSTree* w = allChildren.at(i);
        if (isExternallyActive(w, v)) ret.push_back(w);
    }

    return ret;
}

void CPlanarTester::createBlock(CDFSTree* u, CDFSTree* v)
{
    blocks_.push_back(new CBlock());
    CBlock* b = blocks_.back();

    if (u->dfi() < v->dfi()) {
        b->addVertex(u, true);
        b->addVertex(v, true);
    } else {
        b->addVertex(v, true);
        b->addVertex(u, true);
    }
```

```
    b->addEdge(qMakePair(u, v));

    mapBlocks_[b->root()].push_back(b);
}

CBlock* CPlanarTester::merge(QList<CBlock*> blocksToBeJoined)
{
    CBlock* res = new CBlock();

    for (int i = 0; i < blocksToBeJoined.size(); ++i) {
        CBlock* block = blocksToBeJoined.at(i);
        if (res->root() == NULL)
            res->setRoot(block->root());

        for (int j = 0; j < block->vertices().size(); ++j) {
            CDFSTree* v = block->vertices().at(j);
            if (!res->vertices().contains(v))
                res->addVertex(v, false);
        }

        for (int j = 0; j < block->boundaryVertices().size(); ++j) {
            CDFSTree* v = block->boundaryVertices().at(j);
            bool addBoundary = true;
            for (int k = 0; k < blocksToBeJoined.size(); ++k) {
                CBlock* otherBlock = blocksToBeJoined.at(k);
                if (otherBlock->vertices().contains(v) && !otherBlock-
>boundaryVertices().contains(v)){
                    addBoundary = false;
                    break;
                }
                if (addBoundary)
                    res->boundaryVertices().push_back(v);
            }
        }

        res->edges().append(block->edges());

        blocks_.removeOne(block);
        mapBlocks_[block->root()].removeOne(block);
    }

    blocks_.push_back(res);
    mapBlocks_[res->root()].push_back(res);

    for (int j = 0; j < res->vertices().size(); ++j) {
        CDFSTree* v = res->vertices().at(j);

        QList<CDFSTree*> separatedDFSChildList = mapChildrenList_[v];
        auto it = separatedDFSChildList.begin();
        while (it != separatedDFSChildList.end()) {
            CDFSTree* next = (*it);
            if (res->vertices().contains(next))
```

```
            it = separatedDFSChildList.erase(it);
        }

        mapChildrenList_[v] = separatedDFSChildList;
    }

    return res;
}

QList<QPair<CDFSTree*, CDFSTree*>> CPlanarTester::externalFace()
{
    QList<CDFSTree*> faces = outsideFace();
    QList<QPair<CDFSTree*, CDFSTree*>> face;
    for (int i = 0; i < faces.size(); i++){
        CDFSTree* v1 = faces.at(i);
        CDFSTree* v2 = NULL;
        if (i == faces.size() - 1)
            v2 = faces.front();
        else
            v2 = faces.at(i + 1);
        face.push_back(qMakePair(v1, v2));
    }
    return face;
}
#include "csigyamalayout.h"


QMap<CNode *, QPoint> CSigyamaLayout::doLayout(CGraph *g)
{
    // Создаем необходимую структуру для алгоритма
    QList<SSigNode*> nodes;
    for(int i = 0; i < g->nodeCount(); ++i)
        nodes.push_back(new SSigNode(g->node(i)));
    // Заполняем дочерними узлами
    for(int i = 0; i < nodes.size(); ++i) {
        CNode* v = nodes.at(i)->node_;
        for(int j = 0; j < v->childCount(); ++j) {
            int childIdx = g->nodeIndex(v->child(j));
            nodes.at(i)->children_.push_back( nodes[childIdx] );
            nodes.at(i)->children_.back()->parents_.push_back(nodes.at(i));
        }
    }

    remover_->removeCycles(nodes);
    auto layers = assigner_->assignLayers(nodes);
    orderer_->orderNodes(layers);
    QMap<CNode*, QPoint > coords = coordAssigner_->assignCoordinates(layers);

    // Очищаем ненужные переменные
    qDeleteAll(nodes);
    return coords;
}
```

```
#include "layoutselectdialog.h"
#include "ui_layoutselectdialog.h"

#include "cforcedirectedlayout.h"
#include "csigyamalayout.h"
#include "cequaldistributionheuristic.h"

LayoutSelectDialog::LayoutSelectDialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::LayoutSelectDialog)
{
    ui->setupUi(this);

    connect(ui->rbForce, SIGNAL(clicked(bool)), this, SLOT(updateCurrentPage()));
    connect(ui->rbSigyama, SIGNAL(clicked(bool)), this, SLOT(updateCurrentPage()));
    updateCurrentPage();
}

LayoutSelectDialog::~LayoutSelectDialog()
{
    delete ui;
}

AGraphLayout *LayoutSelectDialog::createSelectedLayout(CGraph* g)
{
    // Фабрика на основе выбранных параметров

    // Force-Directed Layout
    if (ui->stackedWidget->currentIndex() == 0) {
        // Force-Directed Layout

        return new CForceDirectedLayout(ui->spinWidth->value(), ui->spinHeight->value(),
                                        ui->spinIterations->value());
    } else if (ui->stackedWidget->currentIndex() == 1) {
        // Sigyama

        ALayerAssigner* assigner;
        if (ui->cboxLayoutOrderHeuristic->currentIndex() == 0)
            assigner = new CTrivialLayerAssigner();
        else {

            int parentCnt(0), childCnt(0);
            for(int i = 0; i < g->nodeCount(); ++i) {
                if (g->node(i)->parentCount() == 0) ++parentCnt;
                if (g->node(i)->childCount() == 0) ++childCnt;
            }

            if (parentCnt > 1) throw std::runtime_error("В графе больше чем 1 источник. Алгоритм равномерного распределения требует, чтобы граф содержал лишь 1 источник и 1 сток.");
            if (childCnt > 1) throw std::runtime_error("В графе больче чем 1 сток. Алгоритм равномерного распределения требует, чтобы граф содержал лишь 1 источник и 1 сток.");
```

```
            assigner = new CEqualDistributionHeuristic();
        }

        return new CSigyamaLayout(new CTrivialCycleRemover(), assigner,
                              new CTrivialNodeOrderer(), new CTrivialCoordinateAssigner());
    }

    // Planar
    return NULL;
}

void LayoutSelectDialog::updateCurrentPage()
{
    if (ui->rbForce->isChecked()) ui->stackedWidget->setCurrentIndex(0);
    else if (ui->rbSigyama->isChecked()) ui->stackedWidget->setCurrentIndex(1);
    else ui->stackedWidget->setCurrentIndex(2);
}
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
#include "mainwindow.h"
#include "ui_mainwindow.h"

#include "layoutselectdialog.h"
#include "cgraphicsnode.h"
#include "cplanartesting.h"
#include "cedge.h"

#include <qgraphicsitem.h>
#include <qfiledialog.h>
#include <qimage.h>
#include <qpainter.h>
#include <qmessagebox.h>
#include <qfile.h>
#include <qdatastream.h>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow),
    fileName_()
{
    ui->setupUi(this);
```

```
    connect(ui->actionSave, SIGNAL(triggered(bool)), this, SLOT(save()));
    connect(ui->actionSaveAs, SIGNAL(triggered(bool)), this, SLOT(saveAs()));
    connect(ui->actionStartLayout, SIGNAL(triggered(bool)), this, SLOT(startLayout()));
    connect(ui->action_Qt, SIGNAL(triggered(bool)), this, SLOT(aboutQt()));
    connect(ui->actionNew, SIGNAL(triggered(bool)), this, SLOT(newGraph()));
    connect(ui->actionLoad, SIGNAL(triggered(bool)), this, SLOT(load()));
    connect(ui->actionPlanarity, SIGNAL(triggered(bool)), this, SLOT(testPlanarity()));
    connect(ui->actionAdd_nodes, SIGNAL(triggered(bool)), this,
SLOT(addNodesInCrossings()));

    ui->graphicsView->setGraph(new CGraph());

    // Настройка действий
    ui->actionMouseMode1->setCheckable(true);
    ui->actionMouseMode1->setChecked(true);

    ui->actionNodeAdding->setCheckable(true);
    ui->actionNodeAdding->setChecked(false);

    ui->actionEdgeAdding->setCheckable(true);
    ui->actionEdgeAdding->setChecked(false);

    QActionGroup* gr = new QActionGroup(this);
    gr->addAction(ui->actionMouseMode1);
    gr->addAction(ui->actionNodeAdding);
    gr->addAction(ui->actionEdgeAdding);

    connect(gr, SIGNAL(triggered(QAction*)), this,
SLOT(onActionGroupTriggered(QAction*)));
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::exportToPng()
{
    QString path = QFileDialog::getSaveFileName(this, "Укажите файл для сохранения",
QString(), "PNG-Файлы (*.png)");
    if (path.isEmpty()) return;
}

void MainWindow::save()
{
    if (fileName_.isEmpty()) {
        saveAs();
        return;
    }

    QFile file(fileName_);
    if (!file.open(QIODevice::WriteOnly)) return;
```

```
    QDataStream stream(&file);

    stream << ui->graphicsView->graph()->serialize();
    stream << ui->graphicsView->serialize();

    file.close();

    setWindowTitle(QString("Grapher (%1)").arg(fileName_));
}

void MainWindow::saveAs()
{
    fileName_ = QFileDialog::getSaveFileName(this, "Укажите файл для сохранения",
QString(), "Файл состояния приложения (*.sav)");
    if (fileName_.isEmpty()) return;

    QFile file(fileName_);
    if (!file.open(QIODevice::WriteOnly)) return;

    QDataStream stream(&file);

    stream << ui->graphicsView->graph()->serialize();
    stream << ui->graphicsView->serialize();

    file.close();

    setWindowTitle(QString("Grapher (%1)").arg(fileName_));
}

void MainWindow::aboutQt()
{
    QMessageBox::aboutQt(this, "О Qt");
}

void MainWindow::startLayout()
{
    LayoutSelectDialog d;
    if (d.exec() == QDialog::Rejected) return;

    QScopedPointer<AGraphLayout> graphLayout;

    try {
        graphLayout.reset(d.createSelectedLayout(ui->graphicsView->graph()));
    }
    catch(const std::runtime_error& e) {
        QMessageBox::warning(this, "Ошибка требований графа", QString(e.what()));
        return;
    }

    QMap<CNode*, QPoint> positions = graphLayout->doLayout(ui->graphicsView->graph());
```

```
    QPointF center;
    for(auto it = positions.begin(); it != positions.end(); ++it) {
        int nodeIdx = ui->graphicsView->graph()->nodeIndex(it.key());

        ui->graphicsView->node(nodeIdx)->setPos(it.value());

        center += QPointF(it.value().x() / positions.size(), it.value().y() / positions.size());
    }

    ui->graphicsView->centerOn(center);
}

void MainWindow::newGraph()
{
    setWindowTitle("Grapher (не выбран файл для сохранения)");
    fileName_ = QString();
    ui->graphicsView->setGraph(new CGraph());
}

void MainWindow::load()
{
    fileName_ = QFileDialog::getOpenFileName(this, "Укажите файл для загрузки", QString(),
"Файл состояния приложения (*.sav)");
    if (fileName_.isEmpty()) return;

    QFile file(fileName_);
    if (!file.open(QIODevice::ReadOnly)) return;
    QDataStream stream(&file);
    QByteArray gr;
    stream >> gr;
    QByteArray positions;
    stream >> positions;
    file.close();

    CGraph* graph = new CGraph();
    graph->deserialize(gr);
    ui->graphicsView->setGraph(graph);
    ui->graphicsView->deserialize(positions);
}

void MainWindow::testPlanarity()
{
    CPlanarTester t;
    if ( t.testPlanar(ui->graphicsView->graph()) )
        QMessageBox::information(this, "Планарность", "Данный граф является планарным.");
    else
        QMessageBox::warning(this, "Планарность", "Данный граф не является планарным.");
}

#include <qdebug.h>
void MainWindow::addNodesInCrossings()
{
```

```
    QList<QPair<CGraphicsEdge*, CGraphicsEdge*>> collided;

    auto edges = ui->graphicsView->edges();
    for(int i = 0; i < edges.size(); ++i) {
        for(int j = i + 1; j < edges.size(); ++j) {
            if (edges.at(i)->collidesWithItem(edges.at(j)))
                collided << qMakePair(edges.at(i), edges.at(j));
        }
    }

    QList<CNode*> resolvers;
    for(int i = 0; i < collided.size(); ++i)
        resolvers.push_back(ui->graphicsView->addNode(QPointF(0, 0)));

    // Перенаправляем связи
    for(int i = 0; i < collided.size(); ++i) {
        int pIdx = ui->graphicsView->graph()->nodeIndex(resolvers.at(i)));
        int cIdx = ui->graphicsView->graph()->nodeIndex(collided.at(i).first->child());

        collided.at(i).first->setChild(resolvers.at(i));
        CEdge* ui->graphicsView->addEdge(pIdx, cIdx);
    }
}

void MainWindow::onActionGroupTriggered(QAction *act)
{
    if (act == ui->actionMouseMode1) {
        ui->graphicsView->setPointerMode(CGraphView::ePointer);
    } else if (act == ui->actionNodeAdding) {
        ui->graphicsView->setPointerMode(CGraphView::eNodeAdding);
    } else {
        ui->graphicsView->setPointerMode(CGraphView::eEdgeAdding);
    }
}
```