Федеральное государственное автономное образовательное
учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»

На правах рукописи

УДК 512.5+519.1(075.8)

Макаровских Татьяна Анатольевна

МЕТОДЫ И АЛГОРИТМЫ РЕШЕНИЯ ЗАДАЧИ МАРШРУТИЗАЦИИ СПЕЦИАЛЬНОГО ВИДА В ПЛОСКИХ ГРАФАХ

05.13.17 – Теоретические основы информатики

Диссертация на соискание ученой степени
доктора физико-математических наук

Научный консультант

доктор физико-математических наук,

профессор В. И. Цурков

Челябинск–2017

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В современной математике, в отличие от математики начала XX века, появилось большое количество новых направлений, широко применяемых на практике [38]. К ним, например, можно отнести и дискретную математику. Данная дисциплина – это ряд математических теорий, не связанных непосредственно с концепцией предельного перехода и непрерывности. В настоящее время дискретная математика является одним из интенсивно развивающихся разделов математики. Это связано с повсеместным распространением кибернетических систем, языком описания которых она является. Кроме того, дискретная математика является теоретической базой информатики, которая все глубже проникает не только в науку и технику, но и в повседневную жизнь [8,38] в тех областях, которые так или иначе связаны с моделированием мышления, и в первую очередь в вычислительной технике и программировании.

Среди дисциплин дискретной математики видное место занимает теория графов. Данная теория родилась при решении головоломок [38] и в настоящее время играет важную роль как для теоретических исследований, так и для разнообразных приложений. Практическая роль теории графов особенно возросла во второй половине XX века в связи с проектированием различных АСУ и вычислительных устройств дискретного действия, а в начале XXI века – в связи с развитием Интернета и социальных сетей [97]. В теоретическом же плане, помимо давнишних связей с комбинаторной топологией и геометрией, наметились существенные сдвиги на стыке теории графов с алгеброй, математической логикой, лингвистикой, теорией игр, общей теорией систем [18] и др.

Дискретные математические модели получили широкое распространение в науке, технике, экономике, военном деле и т.д. Это связано с тем, что такие модели имеют большое число интерпретаций и многочисленные и разнооб-

разные дискретные задачи, как правило, могут быть описаны немногочисленными комбинаторными моделями [37]. В свою очередь, их исследование и решение прикладых дискретных задач привело к развитию теоретической информатики и существенным продвижениям в ней.

Известно, что первой работой по теории графов как математической дисциплине считается статья Эйлера (1736 г.), в которой рассматривалась задача о Кенигсбергских мостах. Эйлер показал, что нельзя обойти семь городских мостов и вернуться в исходную точку, пройдя по каждому мосту ровно один раз. Следующий импульс теория графов получила лишь спустя почти 100 лет с развитием исследований по электрическим сетям, кристаллографии, органической химии и другим наукам.

В настоящее время интенсивно развивается раздел теории графов, касающийся построения маршрутов, удовлетворяющих специальным ограничениям: эйлеровы и гамильтоновы циклы; маршруты, избегающие запрещенных переходов; самонепересекающиеся и непересекающиеся цепи; бинаправленные двойные обходы и т.д. [53].

Одной из работ по специальным вопросам эйлеровых графов является монография Г.Фляйшнера «Эйлеровы графы и смежные вопросы» [109, 118], где систематизировано и достаточно подробно рассмотрены некоторые виды эйлеровых цепей, например, цепи, не содержащие запрещенных переходов, попарно-совместимые эйлеровы цепи, A-цепи в графах.

Имеется ряд журнальных публикаций других авторов, в которых также рассматриваются задачи, посвященные эйлеровым цепям специального вида [133], например, расширение класса запрещенных переходов [157], самонепересекающиеся и непересекающиеся цепи [9, 132], бинаправленные двойные обходы [109, 118], маршруты Петри [160], прямолинейные маршруты [156], реберно-упорядоченные маршруты [113] и т.д.

Многие задачи нахождения маршрутов, удовлетворяющих определенным ограничениям, появились из конкретных практических ситуаций. Например,

в задачах раскроя листового материала **моделью раскройного плана** является **плоский граф**, а маршрут, покрывающий все ребра, определяет **траекторию движения режущего инструмента**. Ограничением является отсутствие пересечения внутренних граней любой начальной части маршрута с ребрами его оставшейся части [133, 135]. При построении систем управления манипуляторами с помощью неориентированного графа отображают всевозможные элементы траектории манипулятора. При этом возникают проблемы построения маршрутов, удовлетворяющих различным ограничениям, например: прямолинейных маршрутов [156]; маршрутов, в которых следующее ребро определяется заданным циклическим порядком на множестве ребер, инцидентных текущей вершине [117, 118]; маршрутов, в которых часть ребер следует пройти в заданном порядке [117].

Интерес к задачам маршрутизации объясняется их использованием в качестве математических моделей многих проблем управления и автоматизации проектирования.

Например, задача линейного упорядочения вершин параллельно-последовательных графов возникает в задаче размещения объектов с учетом связей между ними (проектирование расположения технологического оборудования нефтехимического предприятия). Технологическая схема производства задает порядок обработки сырья. Требуется разместить единицы оборудования таким образом, чтобы суммарная стоимость трубопроводных связей была минимальной [17].

В [6] сформулирована задача определения оптимальных путей в потоковой сети, когда элементарные требования на организацию потоков продуктов между полюсами возникают последовательно. В статье указано принципиальное отличие этой задачи от классической многопродуктовой проблемы и предложены два алгоритма решения задачи и получены вычислительные процедуры нахождения оптимальных путей. В работе [5] рассматривается моделирование двухуровневой маршрутизации в задаче последовательного

заполнения сети потоками продуктов, производится сравнение результатов работы одно- и двухуровневых алгоритмов на сетях с кластерной и стохастической топологий по таким параметрам, как время работы имитационного моделирования и суммарное количество проведенного потока.

Для планирования и оперативного управления выбора маршрута доставки решается задача, основанная на представлении совокупности типовых состояний системы в виде узлов графа, переходы которого соответствуют управляющим решениям нечеткой ситуационной сети [107].

Математическая модель выбора оптимального маршрута между различными объектами, фиксированными как вершины ориентированного графа, вообще, является одной из самых исследуемых областей [39, 101].

На основе автоматического построения обхода графа возможно исследование эффективности генерации тестов. Здесь необходимо построение маршрута, проходящего через все дуги графа [4].

С помощью модифицированного метода Дейкстры удается построить оптимальные маршруты в беспроводных эпизодических сетях [23]. Для поиска эффективных и полуэффективных решений на графах с векторными весами ребер используется метод сверток. В качестве ограничений применены критерий общей загрузки сети, показатель относительной нагрузки на канал и длина маршрута.

При решении задачи маршрутизации при распределении пассажирских и транспортных потоков [104], учитывающей специфику перемещений пассажиров в крупных городах, необходимо правильно описать поведение пассажира при выборе им пути следования. На его поведение оказывает влияние множество факторов. Для обеспечения единого информационного простанства задач в [104] предлагается использовать специальный граф, который представляет собой систему всех возможных перемещений в пределах города или граф путей сообщения (представляющего собой объединение подграфов метрополитена, железной дороги, пеших перемещений, автомобильных до-

рог и пр.). Все дуги данного графа обладают конечным жизненным циклом: каждый элемент графа характеризует момент создания и момент пометки на удаление. Такая организация хранения данных предоставляет возможность отслеживать изменения городской ситуации и генерировать варианты срезов ситуации на расчетный период времени [7].

Рациональный раскрой материалов является одним из путей решения такой сложной комплексной проблемы как экономия материалов. В 1951 г. вышло первое издание монографии [19], в которой впервые рассмотрены вопросы применения линейного программирования для оптимального гильотинного раскроя (т.е. построения раскройного плана с определением последовательности сквозных резов на гильотине). В Уфимской научной школе раскроя-упаковки была разработана промышленная система технологической подготовки процессов гильотинного раскроя [41]. Развитие автоматизации производства привело к появлению технологического оборудования с числовым программным управлением (ЧПУ), используемого для резки листовых материалов: машин газовой (кислородной), плазменной, лазерной и электроэрозионной резки материала. Новые технологии позволяют осуществлять вырезание по произвольной траектории с достаточной для практики точностью.

Снятие требования резки только сквозными прямолинейными резами позволяет существенно снизить отходы материала, в связи с этим появилось множество публикаций, посвященных вопросам негильотинного раскроя и его оптимизации в различных производствах и на разных уровнях автоматизации. Подробный обзор задач раскроя, алгоритмов и методов их решения специалистами уфимской научной школы приведен в работе А.С. Филипповой [108].

Для промышленных и проектных предприятий, связанных по роду деятельности с задачами раскроя-упаковки, возникает необходимость использования автоматизированных систем технологической подготовки процессов раскроя плоских деталей. Обычно подобные системы имеют модульную структуру. Каждый модуль такой системы позволяет автоматизировать неко-

торый этап. Учет технологических ограничений позволяет формализовать и решать задачу оптимизации маршрута движения режущего инструмента, что, в свою очередь, может привести к существенной экономии ресурсов. В зависимости от количества вырезаемых деталей, типа материала, количества точек врезки и прочих параметров, характеризующих процесс раскроя, типичный процесс вырезания может занимать от нескольких минут до нескольких часов. Так, совмещение границ контуров вырезаемых деталей позволяет сократить длину реза и количество точек врезки, перемещение инструмента между точками врезки также зависит от составленного раскройного плана и выбранной траектории движения режущего инструмента и должно быть минимизировано [114, 115].

В настоящее время имеется потенциальная возможность применения технологий, допускающих совмещение границ вырезаемых деталей. Эти технологии позволяют сократить расход материала, длину резки, и длину и количество холостых проходов. Однако разработка алгоритмов решения задачи маршрутизации для этого случая является открытой задачей.

В [25, 128] показано, что с точностью до гомеоморфизма раскройный план можно представить в виде плоского графа, поэтому задача маршрутизации в плоских графах является актуальной. Для решения данной проблемы отсутствуют соответствующая формальная постановка в терминах задачи построения маршрута в плоском графе и, как следствие, эффективные алгоритмы определения рациональных траекторий. Недостаточность исследований в области разработки алгоритмов маршрутизации инструмента по стоимостным критериям (например, по затратам электроэнергии при резке), а также по суммарному времени резки при использовании нестандартных техник резки отмечается во многих работах, в частности в [105, 114, 115].

Цели и задачи исследования

Предмет исследования – задачи маршрутизации специального вида в плоском графе. Основными **объектами исследования** являются: плоский

граф, представляющий гомеоморфный образ раскройного плана; алгоритмы построения маршрутов специального вида, удовлетворяющих технологическим ограничениям.

Методы исследования. В диссертационной работе для решения задачи маршрутизации в графе, полученном в результате абстрагирования раскройного плана использован современный аппарат теории графов.

Целью диссертационного исследования является решение проблемы маршрутизации специального вида в плоских графах, являющихся гомеоморфными образами раскройного плана.

Для достижения поставленной цели была поставлены и решались следующие **задачи**:

- определить способ представления гомеоморфного образа раскройного плана, позволяющего эффективно решать проблемы маршрутизации;
- доказать существование маршрутов, удовлетворяющих набору ограничений, для плоских графов;
- разработать методы и алгоритмы решения проблемы построения маршрутов специального вида в плоских графах;
- доказать корректность разработанных алгоритмов;
- разработать программное обеспечение для реализации представленных алгоритмов;
- получить оценки количества полученных маршрутов специального вида.

Научная новизна полученных в диссертации результатов заключается в формировании общего подхода к решению задач маршрутизации специального вида в плоских графах и состоит в следующем.

- Введен класс OE-маршрутов в плоских графах. Маршруты введенного класса удовлетворяют требованию отсутствия пересечения внутренних граней пройденной части маршрута с ребрами его непройденной части.

Доказано, что минимальное число OE-цепей, покрывающих граф, совпадает с минимальным числом цепей, покрывающих данный граф.

- Доказано, что плоские эйлеровы графы имеют эйлеровы циклы, принадлежащие классу OE.
- Введен класс AOE-маршрутов в плоских графах. Маршруты введенного класса, как и в классе OE удовлетворяют требованию отсутствия пересечения внутренних граней пройденной части маршрута с ребрами его непройденной части, но на них наложено дополнительное локальное ограничение: следующее ребро определяется заданным циклическим порядком на множестве ребер, инцидентных текущей вершине (т.е. построенная цепь является A-цепью).
- Введен класс NOE-маршрутов в плоских графах. Этот класс является расширением класса AOE и в него входят все непересекающиеся OE-цепи.
- Разработаны эффективные алгоритмы нахождения в плоском графе $G = (V, E)$ маршрутов введенных классов, имеющие полиномиальную вычислительную сложность. Данные алгоритмы позволяют минимизировать длину дополнительных переходов между концевыми вершинами цепей графа $G = (V, E)$.
- Определены оценки количества OE-цепей для фиксированной системы переходов (последовательности обхода ребер).

Теоретическая ценность. Полученные теоретические результаты позволяют расширить класс задач построения маршрутов специального вида, ориентированных на использование ресурсоберегающих технологий, и являются продолжением исследований Г. Фляйшнера, М.А. Верхотурова, Э.А. Мухачевой, А.А. Петунина и позволяют решать задачу построения маршрутов специального вида.

Практическая ценность заключается в разработке новых и совершенствовании существующих методов и средств анализа данных и управления

системами CAD/CAM, повышения эффективности надежности и качества этих систем. Разработанные алгоритмы могут быть применены в проектировании программ вырезания деталей по заданному раскройному плану с использованием ресурсосберегающих технологий. Предложенная теория дает новый импульс для построения новых методов решения задач раскроя. Появляются новые требования к раскройным планам.

Достоверность результатов, полученных в диссертационной работе, базируется на использовании апробированных научных положений, методов исследования, корректном применени математического аппарата, согласовании новых научных результатов с известными теоретическими положениями. Новизна и результативность предложенных алгоритмов подтверждены свидетельствами о регистрации программ.

Новизна и результативность предложенных алгоритмов подтверждены публикациями и свидетельствами о регистрации программ [34, 35, 80–83].

Апробация работы. Все результаты диссертационной работы, разработанные методы, алгоритмы и результаты вычислительных экспериментов докладывались и получили одобрение на следующих международных, всероссийских и региональных конференциях.

1. 8th MIM Conference, Manufacturing Modelling, Management and Control (Troyes, France, 28–30 июня, 2016).
2. 15-я Международная конференция «Системы проектирования технологической подготовки производства и управления этапами жизненного цикла промышленного продукта(CAD/CAM/PDM-2015)» (Москва, 26–28 октября, 2015).
3. Международная научно-техническая конференция «Пром-Инжиниринг» (Челябинск, 2015–2016).
4. International Conference «Information Technologies for Intelligent Decision Making Support» (2013, 2015, 2016).

5. 2nd International conference "Intelligent Technologies for Information Processing and Management" (ITIPM-2014, Уфа, 10–12 ноября, 2014).
6. 5th International conference "Optimization and Applications" (Optima-2014, Petrovac, Montenegro, Sep. 27–Oct.4, 2014).
7. Workshop on Computer Science and Information Technologies (2003, 2008, 2010, 2011, 2013).
8. Czech-Slovak International Symposium on Graph Theory, Combinatorics, Algorithms and Applications (2006, 2013).
9. Международная конференция «Дискретная оптимизация и исследование операций» (2010, 2013, 2016).
10. Третья Международная конференция «Математическое моделирование, оптимизация и информационные технологии» (Кишинев, Молдова, 2012 г.).
11. Международная конференция «Информационные технологии и системы», респ. Башкортостан, оз. Банное, (2012–2016).
12. Всероссийская конференция «Статистика. Моделирование.Оптимизация» (Челябинск, 28 ноября – 2 декабря, 2011 г.).
13. Международная научно-практическая конференция «Современные информационные технологии и ИТ-образование», Москва, МГУ им. М.В. Ломоносова, (2011–2013).
14. XIV Всероссийская конференция «Математическое программирование и приложения», Екатеринбург, (2011 и 2015).
15. 8th French Combinatorial Conference, Orsay, France (June,28–July,2, 2010).
16. X Международный семинар «Дискретная математика и ее приложения» (Москва, МГУ им. М.В. Ломоносова, 2001, 2004, 2010, 2016 г.)
17. Международная научная конференция «Дискретная математика, алгебра и их приложения», Минск, Институт математики НАН Беларуси, (2009 и 2015).

18. IV Всероссийская конференция «Проблемы оптимизации и экономические приложения» (Омск, 2003, 2009).
19. 13-th IFAC Symposium on Information Control Problems in Manufacturing, Moscow, 2009.
20. X Белорусская математическая конференция, Минск, Институт математики НАН Беларуси, 3–7 ноября 2008 г.
21. Научно-практическая конференция «Обратные задачи в приложениях», Бирск, БирГСПА, 2008.
22. Международная научная конференция «Информационно-математические технологии в экономике, технике и образовании», Екатеринбург, УГТУ-УПИ, (2007–2008).
23. International meeting «Euler and Modern Combinatorics», St. Petersburg, June 1–7, 2007.
24. 6-th International Congress on Industrial and Applied Mathematics, Zurich, 16–20 July, 2007.
25. Молодежная конференция «Проблемы теоретической и прикладной математики», Екатеринбург, ИММ, УрО РАН, (2005–2009).
26. Российская конференция «Дискретный анализ и исследование операций», Новосибирск, Институт математики им. С.Л. Соболева СО РАН, 2002, 2004).
27. XIII Международная конференция «Проблемы теоретической кибернетики», МГУ им.М.В.Ломоносова, Институт прикладной математики им.М.В.Келдыша РАН, ННГУ им. Н.И. Лобачевского, КазГУ, (1999, 2002).
28. Второй международный конгресс студентов, аспирантов и молодых ученых «Молодежь и наука – третье тысячелетие», Москва, МГТУ им. Н.Э. Баумана, 15–19 апреля, 2002 г.
29. XI Соревнование молодых ученых Европейского Союза, Греция, 19–26 сентября, 1999 г.

30. Российская молодежная инженерная выставка «Шаг в будущее», Москва, МГТУ им. Н.Э. Баумана, 9–13 марта, 1999 г.

Кроме того, результаты работы были представлены на ежегодных научно-практических конференциях Южно-Уральского государственного университета.

Публикации. По материалам проведенных исследований опубликовано 84 печатные работы, в числе которых 10 публикаций из списка ВАК [25, 31, 42, 56, 59, 67–69, 86, 138], 6 публикаций, индексируемых в SCOPUS [126, 128, 138, 141–143] (статья [141] является переводом на английский язык статьи [69]) и 6 свидетельств о регистрации программного продукта [34, 35, 80–83].

Структура и объем работы. Диссертация состоит из введения, шести глав, заключения, списка использованной литературы (160 наименований), двух приложений. Основная часть работы содержит 209 страниц машинописного текста, 66 иллюстраций.

В первой главе на основе аналитического обзора литературы, отражающего состояние проблемы применения графов в математическом моделировании, показано место решаемой в данной работе задачи относительно ранее опубликованных в научной литературе результатов. С тем, чтобы более четко очертить круг решаемых в данной работе задач и показать их место в общей теории графов, приведено краткое описание постановки задачи нахождения эйлеровых (обход по всем ребрам ровно по одному разу с возвратом в исходную вершину) маршрутов, указаны известные алгоритмы построения эйлеровых цепей и отмечено, что эти алгоритмы находят в графе произвольную эйлерову цепь, на которую не наложено никаких ограничений.

Результаты анализа ограничений различных задач маршрутизации дают возможность классифицировать их на локальные, когда следующее ребро в маршруте определяется условиями, заданными в текущей вершине или на текущем ребре (цепи, избегающие запрещенных переходов; A-цепи; прямоли-

нейные цепи), и на глобальные (маршруты Петри, бинаправленные двойные обходы и т.п.).

Анализ публикаций показал, что большинство работ посвящено алгоритмам с локальными ограничениями на порядок обхода ребер (например, запрещение левых поворотов; маршруты без поворотов; использование в каждой вершине графа заданного циклического порядка включения ребер в маршрут и т.п.). Обобщение большинства частных случаев задачи построения маршрутов с локальными ограничениями дано С. Зейдером [157]. Им предложено представлять локальные ограничения в каждой вершине v исходного графа G в виде грфа $G_{E(v)}$ возможных переходов. Множеством вершин графа $G_{E(v)}$ являются все ребра, инцидентные вершине v; смежные вершины графа $G_{E(v)}$ соответствуют разрешенным переходам.

Проблемы построения допустимого маршрута или множества маршрутов, покрывающих все ребра исходного графа, не освещены в рассмотренных источниках. Их решение приведено в диссертационном исследовании.

Среди публикаций, посвященных задачам маршрутизации с глобальными ограничениями, не удалось найти каких-либо результатов о маршрутах с запрещенными последовательностями ребер. Задача построения маршрутов в плоских графах, у которых отсутствует пересечение внутренних граней пройденной части маршрута с ребрами его непройденной части в связных и несвязных графах не рассмотрена в исследованной автором литературе, то же самое можно сказать и про задачу построения A-цепей, в которых отсутствует пересечение внутренних граней пройденной части с ребрами непройденной части.

Во второй главе проанализированы задачи построения допустимого маршрута или множества маршрутов, покрывающих все ребра исходного графа. Предполагается, что ограничения на маршруты удовлетворяют условиям теоремы С. Зейдера [157]. Отмечена связь графов переходов с понятием системы разбиения графа, используемой Г. Фляйшнером [109]. На основе

установленной связи построен алгоритм «РАЗМЕТКА» для распознавания выполнения условий теоремы С. Зейдера и алгоритм «P_G-СОВМЕСТИМЫЙ ПУТЬ» для построения допустимого пути [56, 151].

Сложность алгоритмов не превосходит величины $O(|E|)$, где E – число ребер графа G. Приведены примеры использования построенных алгоритмов, рассмотрена техника программной реализации данных алгоритмов [59, 83].

Заметим, что с помощью алгоритма нахождения P_G-совместимого пути возможно построение только простой цепи между двумя различными вершинами (т.е. цепи, в которой любая ее вершина встречается в ней ровно один раз).

Построен также алгоритм «P_G-СОВМЕСТИМЫЙ МАРШРУТ» для нахождения эйлерова покрытия графа G допустимыми маршрутами. Алгоритм является рекурсивным и имеет вычислительную сложность $O(|E| \cdot |V|)$ [56].

Алгоритмы, рассмотренные во второй главе, имеют самостоятетьный теоретический интерес, реализованы в виде программ для ЭВМ [83] и могут быть использованы для решения ряда практических задач. Однако, наложенные на порядок обхода ограничения носят локальный характер, т.е. присутствуют ограничения на последовательности из двух ребер, инцидентных общей вершине. Поэтому открытой остается задача построения маршрутов с ограничениями на использование в маршрутах более длинных последовательностей ребер. По-видимому, алгоритмы решения таких задач должны существенным образом использовать их специфику.

В третьей главе поставлена и решена задача построения в плоском графе маршрутов, удовлетворяющих условию отсутствия пересечения внутренних граней любой его начальной части с ребрами его оставшейся части. Формально такие маршруты определены как упорядоченная последовательность OE-цепей графа $G = (V, E)$ [135].

Сформулирована и доказана теорема существования OE-цикла в плоском эйлеровом графе [86, 137], OE-маршрута с минимальным по мощности

множеством цепей [142], OE-маршрута с минимальным по мощности множеством цепей и минимальной длиной этих цепей [67]. Приведены алгоритмы решения задачи для общего случая: плоского несвязного графа [27]. Доказательства данных результатов конструктивны и фактически сводятся к описанию и доказательству результативности алгоритмов построения искомых циклов (маршрутов).

Рассмотренные алгоритмы реализованы в виде компьютерных программ [82]–[81], в которых для представления заданного плоского графа G использовано задание для каждого ребра e следующих функций: $v_1(e)$, $v_2(e)$ – вершины, инцидентные ребру e; $l_1(e)$, $l_2(e)$ – ребра, следующие за e при его вращении против часовой стрелки вокруг вершин $v_1(e)$ и $v_2(e)$ соответственно. Анализ сложности построенных алгоритмов показывает, что поставленную задачу можно решить за полиномиальное время.

В четвертой главе решена задача построения A-цепей с упорядоченным охватыванием для плоского связного 4-регулярного плоского графа (AOE-цепи) [28, 32]. Также доказано, что для существующей системы переходов, которая соответствует некоторой A-цепи, можно построить OE-цепь. Введен класс NOE-маршрутов в плоских графах. Этот класс является расширением класса AOE и в него входят все непересекающиеся OE-цепи [30]. Показано, что число OE-цепей, соответствующих фиксированной системе переходов, равно удвоенному числу вершин графа, смежных внешней грани [31]. Разработан и запрограммирован алгоритм поиска AOE-цепи в плоском 4-регулярном графе [35, 36]. Показано, что алгоритм решает задачу за полиномиальное время.

В пятой главе рассмотрены разработанные программные средства для построения OE-цепей и покрытий, а также AOE-цепей для 4-регулярных графов [35, 80–82].

С помощью разработанного программного обеспечения возможно протестировать работу всех рассмотренных в главах 3 и 4 алгоритмов. Программ-

ное обеспечение позволяет вводить/выводить данные в текстовом и графическом формате, просматривать результат работы алгоритмов в динамике (анимация обхода).

Представлен алгоритм `OrderedEnclosingTest` [46], позволяющий проверить соответствие маршрута обхода плоского графа критерию упорядоченного охватывания. В случае нарушения рассмотренного критерия алгоритм определяет ребро цепи, повлекшее нарушение. Алгоритм может быть применен для повышения надежности программных комплексов, формирующих управляющие программы для станков раскроя, а так же на этапе тестирования системы технологической подготовки раскроя в ручном и автоматическом режиме.

В шестой главе отмечено, что в отличие от гильотинного раскроя, негильотинный раскройный план не дает программу вырезания деталей. Построение программы управления раскройным автоматом для реализации заданного раскройного плана является самостоятельной задачей. Приведена классификация задач маршрутизации инструмента машин листовой резки, предложенная в работах Дж. Хоэфта и У. С. Палекара [121].

Показано, что технологии ECP и ICP за счет возможности совмещения границ вырезаемых деталей позволяют сократить расход материала, длину резки, и длину и количество холостых проходов. Проблемы уменьшения отходов материала и максимального совмещения фрагментов контуров вырезаемых деталей решается на этапе составления раскройного плана.

Отмечено [27, 126], что применение технологий ECP и ICP в системе технологической подготовки процессов раскроя плоских деталей предполагает следующие этапы.

1. Составление раскройного плана, заключающееся в нахождении такого варианта размещения вырезаемых деталей на прямоугольном листе или ленте, при котором минимизируются отходы и максимизируется длина совмещенных элементов контуров вырезаемых деталей.

2. Абстрагирование раскройного плана до плоского графа. Для определения последовательности резки фрагментов раскоройного плана не используется информация о форме детали, поэтому все кривые без самопересечений и соприкосновений на плоскости, представляющие форму деталей, интерпретируются в виде ребер графа, а все точки пересечений и соприкосновений представляются в виде вершин графа. Для анализа выполнения технологических ограничений необходимо введение дополнительных функций на множестве вершин, граней и ребер полученного графа.

3. Решение задачи построения оптимальных маршрутов с ограничениями, наложенными на порядок обхода ребер. Данные ограничения непосредственно вытекают из технологических ограничений, наложенных на порядок вырезания деталей: отрезанная от листа часть не должна требовать дополнительных разрезаний, должны отсутствовать пересечения резов, необходимо оптимизировать длину холостых переходов, минимизировать количество точек врезки и т.д.

4. Составление программы управления процессом раскроя на основе маршрута, найденного с помощью алгоритма решения абстрагированной задачи маршрутизации. Выполняется обратная замена абстрактных ребер плоского графа системой команд раскройному автомату, обеспечивающей движение по кривым на плоскости, соответствующим форме вырезаемой детали.

Этапы построения раскройного плана и интерпретации найденного маршрута в терминах команд раскройному автомату являются общими для всех технологий и достаточно известны. Реализация второго и третьего этапов для технологий ECP и ICP возможна применением разработанных в работе алгоритмов построения OE-покрытий [126].

Для прямоугольного негильотинного раскройного плана разработан эффективный алгоритм его кодировки для применения алгоритмов построения

OE-маршрутов. Предложен полиномиальный алгоритм построения AOE-покрытий использующий данную кодировку [68].

В заключении перечислены основные результаты работы.

Работа выполнялась в соответствии с планами госбюджетных НИР ЮУрГУ (номер гос. регистрации 01.2001 05137) и в рамках соглашения №14.B37.21.0395 с Министерством образования и науки Российской Федерации от 06 августа 2012 года. Работа поддерживалась грантами РФФИ «Урал» (проекты 01-01-96401, 10-07-96002-р_урал_а), Грантом Президента РФ МК-2603.2008.9, Губернаторским грантом Челябинской области р2001урчел-01-04 и грантами губернатора Челябинской области для студентов, аспирантов и молодых ученых в 2002, 2003, 2013 гг.

ГЛАВА 1
ПРИМЕНЕНИЕ ГРАФОВ В ЗАДАЧАХ МАТЕМАТИЧЕСКОГО МОДЕЛИРОВАНИЯ СИСТЕМ УПРАВЛЕНИЯ

Дискретные математические модели получили широкое распространение в науке, технике, экономике, военном деле и т.д. Это связано с тем, что такие модели имеют большое число интерпретаций и многочисленные и разнообразные дискретные задачи, как правило, могут быть описаны немногочисленными комбинаторными моделями [37]. В свою очередь, их исследование и решение прикладых дискретных задач привело к развитию теории графов.

С помощью графовых моделей формализуется широкий класс задач, начиная от занимательных (задача о кенигсбергских мостах, задача четырех красок и др.), до ряда серьезных теоретических и прикладных задач электротехники, физики, химии, топологии и др. Аппарат теории графов используется для построения моделей Интернета и социальных сетей [97]. Моделью Интернета является ориентированный граф, вершинами которого служат сайты, а ребрами – ссылки. Например, с помощью графа, изображающего сеть дорог между населенными пунктами, можно определить не только маршрут проезда от одного до другого пункта, но, если таких маршрутов окажется несколько, – выбрать в определенном смысле оптимальный (самый короткий или самый безопасный, самый дешевый или путь, который требует минимум энергии и т.п.). Для каждой прикладной задачи существуют некоторые особенности, которые накладывают определенные ограничения на графы. Дополнительные ограничения, вызванные практическими требованиями прикладной задачи, могут быть наложены и на порядок обхода ребер графа.

В современной дискретной математике созданы и развиваются различные теории исчисления на графах, которые носят комбинаторный, вероятностный характер, а также ряд других задач (потоки в сетях, задачи о разрезе, о максимальном потоке и др.). При решении подобных задач используется алгебраическая техника [8].

В настоящее время теория графов активно развивается. Изучение правил и законов человеческого мышления обусловило применение методов дискретной математики в тех областях техники, которые так или иначе связаны с моделированиием мышления, и в первую очередь в вычислительной технике и программировании.

Интерес к задачам маршрутизации обусловлен тем, что такие задачи позволяют построить математические модели для многих проблем управления и автоматизации проектирования. Приведем некоторые из таких задач.

1. Задача линейного упорядочения вершин параллельно-последовательных графов возникает в задаче размещения объектов с учетом связей между ними (например, проектирование расположения технологического оборудования нефтехимического предприятия). В этом случае технологическая схема производства задает порядок обработки сырья. Требуется разместить единицы оборудования таким образом, чтобы суммарная стоимость трубопроводных связей была минимальной [17].
2. Задача, основанная на представлении совокупности типовых состояний системы в виде узлов графа, переходы которого соответствуют управляющим решениям нечеткой ситуационной сети [107], возникает при планировании и оперативном управлении выбора маршрута доставки.
3. Задача выбора оптимального маршрута между различными объектами, фиксированными как вершины ориентированного графа, является распространенной математической моделью для широкого круга исследуемых областей [39, 101].

4. Задачи автоматического построения обхода графа дают возможность исследовать эффективность генерации тестов. В этом случае необходимо построение маршрута, проходящего через все дуги графа [4].
5. Модифицированный метод Дейкстры дает возможность построить оптимальные маршруты в беспроводных эпизодических сетях [23]. Для поиска эффективных и полуэффективных решений на графах с векторными весами ребер используется метод сверток. В качестве ограничений применены критерий общей загрузки сети, показатель относительной нагрузки на канал и длина маршрута.
6. Специальные задачи, которые определяются практической деятельностью. Например, при решении задачи маршрутизации распределения пассажирских и транспортных потоков [104], учитывающей специфику перемещений пассажиров в крупных городах, необходимо правильно описать поведение пассажира при выборе им пути следования. На его поведение оказывает влияние множество факторов. Для обеспечения единого информационного пространства задач в [104] предлагается использовать специальный граф, который представляет собой систему всех возможных перемещений в пределах города или граф путей сообщения (представляющий собой объединение подграфов метрополитена, железной дороги, пеших перемещений, автомобильных дорог и пр.). Все дуги данного графа обладают конечным жизненным циклом: каждый элемент графа характеризует момент создания и момент пометки на удаление. Такая организация хранения данных предоставляет возможность отслеживать изменения городской ситуации и генерировать варианты срезов ситуации на расчетный период времени [7].

1.1 Основные понятия и определения

В дальнейшем будем использовать терминологию, введенную в монографиях [18], [109] и [118]. Для цельности изложения приведем понятия и определения, используемые в данной диссертационной работе.

Обыкновенным графом $G = (V, E)$ будем называть упорядоченную пару множеств: конечное непустое V, элементы которого называются *вершинами* графа G, и подмножество E, элементы которого называются *ребрами* графа. Ребро, соединяющее вершины x и y (или, что то же самое, y и x), будем обозначать xy. Также говорят, что ребро xy инцидентно каждой из этих вершин (и наоборот, они обе инцидентны данному ребру). Если не требуется напоминать, какие именно вершины соединяет ребро, то его можно обозначать и одной буквой (e, u и др.). Вершины $x, y \in V$ *смежны*, если ребро $xy \in E$, и несмежны, если $xy \notin E$.

Часть $G' = (V', E')$ называется *подграфом* графа G, если $E' = \{xy \in E \,|x, y \in V'\}$; иными словами, при образовании подграфа G' из графа G удаляются все вершины множества $V \setminus V'$ и только те ребра, которые инцидентны хотя бы одной удаляемой вершине. Таким образом, подграф данного графа G однозначно определяется заданием непустого подмножества вершин V' или, что равносильно, заданием строгого подмножества $W = V \setminus V' \subset V$ тех вершин, которые надо удалить; в последнем случае будем кратко писать $G' = G \setminus W$. В частности, при $V' = V$ имеем $G' = G \setminus \emptyset = G$.

Граф называется *связным*, если множество его вершин невозможно так разбить на попарно непересекающиеся непустые подмножества, чтобы никакие две вершины из разных подмножеств не были бы смежны. Несвязный же граф однозначно разбивается указанным способом на связные подграфы, называемые *компонентами связности*.

В дальнейшем, если не оговорено противное, будем рассматривать только связные графы.

Обобщением понятия обыкновенного графа является понятие *мультиграфа*. Под мультиграфом будем понимать упорядоченную тройку $G = (V, E, \phi)$, где $V \neq \emptyset$ – множество *вершин*, E – множество *ребер*, оба конечные, а $\phi : E \to V^2$ – отображение, относящее каждому ребру $e \in E$ неупорядоченную пару $\phi(e) = \tilde{x}y$ вершин $x, y \in V$, называемых *концами* этого ребра.

Возможной формой представления графов является *матрица инцидентности* «вершины/ребра». Строкам данной матрицы соответствуют вершины графа G, а столбцам – ребра. Если ребро $e = \{v_1 v_2\}$, то на пересечении столбца e и строк v_1 и v_2 будут стоять единицы. Остальные элементы столбца e – нулевые. Пространственная сложность такого представления равна $O(|V| \cdot |E|)$.

Очевидно, что для представления графа можно организовать список ребер, с указанием для каждого ребра пары инцидентных вершин. Пространственная сложность такого представления будет $O(|E| \cdot \log_2 |V|)$. Логарифм в данном выражении появляется в связи с необходимостью нумерации вершин.

Для обозначения мультиграфа $G(V, E, \phi)$ будем использовать обозначение $G = (V, E)$, если это не приводит к двусмысленности.

Топологическим графом называют такой граф $G = (V, E, \phi)$, вершинами которого служат некоторые выделенные точки трехмерного евклидова пространства, а ребрами – жордановы дуги, соединяющие эти точки; у звена обе инцидентные вершины (концевые точки) различны, у петли совпадают. Требуется еще, чтобы никакая внутренняя (неконцевая) точка ребра не совпадала ни с одной вершиной графа и ни с одной точкой другого ребра; всякое нарушение этого требования будем кратко называть *пересечением*. Таким образом, изображение абстрактного графа на рисунке само является топологическим графом, изоморфным исходному лишь при условии, что рисунок не содержит пересечений.

Всякий абстрактный граф допускает *топологическое представление*, т. е. в пространстве $\mathbf{R}^3$ существует изоморфный ему топологический граф.

Как в теории, так и в приложениях особо важную роль играют те топологические свойства графа, которые связаны с возможностью или невозможностью поместить его в плоскость. Известно, что евклидова плоскость гомеоморфна сфере, из которой удалена одна точка; соответствующее отображение можно осуществить, например, с помощью стереографического проектирования. Этим же отображением топологический граф на сфере переводится в изоморфный топологический граф на плоскости и наоборот. Граф, допускающий такие топологические представления, называется *планарным*, а его конкретное представление в плоскости – *плоским графом*.

Если G_S – топологическое представление графа G в плоскости S, то компоненты связности множества $G \setminus S$ называются *гранями* плоского графа G_S, а множество граничных точек грани – ее *краем*; теоретико-множественное объединение краев всех граней равно G_S, поэтому каждое ребро и каждая вершина принадлежат краю хотя бы одной грани, и ясно также, что никакое ребро (в отличие от вершины) не может принадлежать краям более чем двух граней.

Ровно одна из граней плоского графа G_S является *внешней* (бесконечной). Причем всегда можно так изменить расположение графа G_S в плоскости S (т. е. построить изоморфный ему граф G'_S), чтобы наперед заданная грань стала внешней: для этого достаточно сначала отобразить стереографически G_S на сферу, затем повернуть ее так, чтобы полюс N попал в образ выбранной в качестве внешней грани, и, наконец, спроектировать граф обратно на плоскость S.

Далее для плоского графа G через $E(G)$ будем обозначать множество его ребер, представляющих плоские жордановы кривые с попарно непересекающимися внутренностями, гомеоморфные отрезкам. Ребро, у которого начало и конец совпадают, называют петлей в графе. Через $V(G)$ обозначим множество граничных точек этих кривых. Топологическое представление плоского

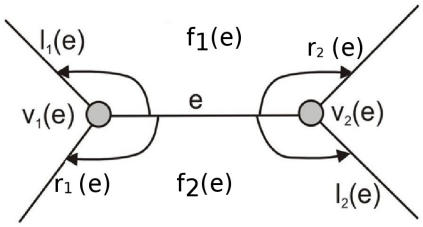


Рисунок 1.1: Функции для представления плоского графа

графа $G = (V, E)$ на плоскости S с точностью до гомеоморфизма определяется заданием для каждого ребра $e \in E$ следующих функций [86]:

- $v_k(e)$, $k = 1, 2$ – вершины, инцидентные ребру e,
- $l_k(e)$, $k = 1, 2$ – ребра, полученные вращением ребра e против часовой стрелки вокруг вершины $v(k)$,
- $r_k(e)$, $k = 1, 2$ – ребра, полученные вращением ребра e по часовой стрелке вокруг вершины $v(k)$,
- $f_k(e)$ – грань, находящаяся слева при движении по ребру e от вершины $v_k(e)$ к вершине $v_{3-k}(e)$, $k = 1, 2$.

Иллюстрация введенных функций дана на рисунке 1.1. Таким образом, пространственная сложность представления гомеоморфного образа графа $G = (V, E)$ равна $O\left(|E| \cdot \log_2 |V|\right)$.

1.2 Маршруты в графах

Пусть $G = (V, E)$ – граф. Последовательность вида

$$v_0 e_1 v_1 e_2 v_2 \dots v_{n-1} e_n v_n, \tag{1.1}$$

где $v_0, v_1, v_2, \dots, v_n \in V$, а $e_1 = \{v_0, v_1\},\ e_2 = \{v_1, v_2\}, \dots, e_n = \{v_{n-1}, v_n\} \in E$, называется *связным маршрутом* длины n из вершины v_0 в вершину v_n.

Замечание 1. Упорядоченную последовательность маршрутов вида (1.1) также будем называть маршрутом.

При $v_0 = v_n$ и $n \geq 1$ маршрут будем называть *циклическим*. Маршрут, все ребра которого различны, называется *цепью*. Маршрут, не содержащий по-

вторяющихся вершин, называется *путем*. Циклический маршрут, в котором каждое ребро встречается ровно по одному разу, называется *упорядоченным циклом* [13]. Упорядоченный цикл, который включает все ребра графа G ровно по одному разу, называется *эйлеровым циклом*, а граф, содержащий эйлеров цикл – *эйлеровым графом* [3].

Именно с задачи нахождения эйлерова цикла зародилась теория графов. Но результаты, полученные более двухсот лет назад, не только актуальны и по сей день, но активно развиваются и применяются на практике в решении самых разнообразных задач с более сложными ограничениями на порядок обхода вершин и ребер [9, 56, 67, 124, 137, 142, 157, 160]. Причем область применения алгоритмов построения маршрутов в эйлеровых графах не ограничивается классическими примерами (сбора мусора, доставки почты, чистки улиц, проверки работы линий электропередач и т.п.).

Эйлером доказана следующая теорема [109].

Теорема 1. [Эйлер] *Связный неориентированный граф G содержит эйлеров цикл (эйлерову цепь) тогда и только тогда, когда число вершин нечетной степени равно 0 (0 или 2).*

Доказательство теоремы об эйлеровых графах имеет конструктивный характер и на его основе можно построить рекурсивный алгоритм нахождения эйлерова цикла [118]. В данной работе будем рассматривать вопросы построения наборов цепей, покрывающих все ребра графа, и удовлетворяющие определенным ограничениям на порядок обхода ребер.

Несмотря на возможность нахождения эйлерова цикла за полиномиальное время, существует тесная взаимосвязь между эйлеровыми и гамильтоновыми циклами (когда требуется построить цикл, проходящий через каждую вершину ровно по одному разу) [53] и зачастую задача построения эйлерова цикла с ограничениями на порядок обхода ребер или посещения вершин сводится к известной задаче комивояжера. Это задача нахождения гамильтонова цикла с

минимальным общим весом ребер. В отличие от задачи построения эйлерова цикла без ограничений, задача комивояжера является $\mathcal{NP}$-полной. Например, в статье [119] приводятся различные взаимосвязи между эйлеровыми графами и прочими свойствами графов (как то гамильтоновость, существование нигде ненулевых потоков, существование треугольных циклов и пр.).

Представление о непосредственных применениях гамильтоновых цепей дает следующая ситуация [15]. Имеется машина и n заданий, каждое из которых она способна выполнить после соответствующей настройки. При этом необходимо затратить на переналадку t_{ij} единиц времени для того, чтобы после выполнения i-го задания выполнить j-е. В предположении, что $t_{ij} = t_{ji}$, требуется найти последовательность выполнения заданий, при которой время каждой переналадки не превосходит величины t. Если построить граф G, у которого $V = \{1, 2, \ldots n\}$, $E = \{i, j \mid t_{ij} \leq t\}$, то описанная задача сводится к отысканию гамильтоновой цепи в этом графе.

1.3 Разложения графов на цепи

Будем говорить, что набор реберно-непересекающихся цепей покрывает граф G, если каждое ребро графа G входит в одну из этих цепей. Из теоремы 1 фактически следует, что связный граф обладает открытой или замкнутой покрывающей цепью тогда и только тогда, когда он имеет не более двух вершин нечетной степени.

Пусть связный граф G содержит m вершин нечетной степени. Очевидно, что m четно, т.е. $m = 2k$. Рассмотрим граф G', полученный добавлением к G новой вершины v и ребер, соединяющих v с вершинами графа G нечетной степени. Поскольку степени всех вершин графа G' четны, то G' содержит эйлеров цикл. Если теперь выбросить v из этого цикла, то получится k цепей, содержащих все ребра графа G, т.е. *покрывающих* G. С другой стороны, граф, являющийся объединением r реберно-непересекающихся цепей, имеет

самое большее $2r$ вершин нечетной степени. Поэтому меньшим числом цепей граф G покрыть нельзя. Данный результат известен как теорема Листинга-Люка [109, 118].

Теорема 2. [Листинг, Люк] *Если связный граф содержит ровно $2k$ вершин нечетной степени, то минимальное число покрывающих его реберно-непересекающихся цепей равно k.*

В эйлеровом графе существует, как правило, несколько эйлеровых циклов. Зная один такой цикл, получить новый можно следующим простым приемом.

Пусть C – исходный эйлеров цикл, и вершина v проходится в этом цикле более одного раза. Рассмотрим часть (подцикл) цикла C, состоящую из ребер и вершин, проходимых между k-м и l-м ($k < l$) посещениями вершины v. Это будет некоторый цикл C_1. Цикл C, как и всякий эйлеров цикл, задает некоторый порядок обхода ребер графа и индуцирует порядок прохождения ребер цикла C_1. Итак, изменив указанным способом эйлеров цикл, получаем новый эйлеров цикл. Теорема Коцига [15] утверждает, что последовательности таких изменений достаточно для получения всех эйлеровых циклов из данного.

Теорема 3. [А.Коциг]. *Если C и C' – эйлеровы циклы графа G, то в G существует такая последовательность эйлеровых циклов $C = C_1, C_2, \ldots, C_k = C'$, что C_{i+1} получается из C_i путем изменения порядка обхода ребер некоторого подцикла на обратный.*

Таким образом, из теоремы Коцига можно заключить, что граф G как правило имеет много эйлеровых цепей, а, следовательно, и разложений на цепи, поэтому имеется возможность построения разложений, удовлетворяющих дополнительным ограничениям.

Большое число примеров различных типов эйлеровых цепей приведено в первом томе монографии Г. Фляйшнера «Эйлеровы графы и смежные вопро-

сы» [109], где систематизированно и достаточно подробно рассматриваются некоторые виды эйлеровых цепей, например:

- цепи, не содержащие запрещенных переходов;
- попарно-совместимые эйлеровы цепи;
- A-цепи в графах;
- самонепересекающиеся и непересекающиеся цепи;
- бинаправленные двойные обходы.

В последнее время появились публикации, посвященные новым видам маршрутов в графах [9, 24, 109, 113, 118, 123, 124, 134, 156, 157, 160], например:

- расширение класса запрещенных переходов [157];
- маршруты Петри [160];
- k-реберно-упорядоченные графы [113];
- задачи теории расписаний с логическими условиями предшествования, которым эквивалентны задачи циклических игр [1, 24];
- прямолинейные маршруты в эйлеровых графах [156] и т.п.

Однако, среди публикаций, рассматривающих задачу построения маршрутов в плоских графах, у которых отсутствует пересечение внутренних граней пройденной части маршрута с ребрами его непройденной части, можно отметить только [133]. В работе [137] автора диссертации данная задача поставлена и решена для случая плоских эйлеровых графов, предложены эффективные алгоритмы ее решения, а в работах [69, 141] автора диссертации рассмотрен случай связного неэйлерова графа и предложен алгоритм поиска допустимого эйлерова покрытия графа. В диссертационной работе обобщены решенные автором задачи маршрутизации в графах:

- задача поиска допустимого пути, избегающего запрещенных переходов [59];
- задача построения маршрутов в плоских графах, у которых отсутствует пересечение внутренних граней пройденной части маршрута с ребрами его непройденной части (для связных [67] и несвязных [44, 140] графов);

- задача построения A-цепей, в которых отсутствует пересечение внутренних граней пройденной части с ребрами непройденной части [61, 66].

Поскольку построение разложений на цепи фактически сводится к построению эйлерова цикла, рассмотрим более подробно некоторые известные классические алгоритмы решения этой задачи.

В [118] отмечено, что основу всем алгоритмам построения эйлеровых цепей составляет расщепляющий алгоритм. Он является далеко не самым быстрым. Тем не менее он служит основой для целой серии алгоритмов с полиномиально ограниченной временной сложностью. В большинстве монографий по теории графов для нахождения эйлеровой цепи изложен алгоритм Флёри, являющийся одним из самых старых алгоритмов для эйлеровых цепей [118].

Расщепляющий алгоритм последовательно строит графы H с возрастающим числом вершин степени 2, тогда как в алгоритме Флёри цепь T_i хранится отдельно, поэтому размер графа G_i строго убывает. Таким образом, алгоритм Флёри представляется более удобным с практической точки зрения, нежели расщепляющий алгоритм. Несмотря на практические недостатки расщепляющего алгоритма, из него можно вывести алгоритм построения $P(G)$-совместимых эйлеровых цепей (а, значит, и эйлеровых цепей в орграфах), или алгоритм построения непересекающихся эйлеровых цепей графа G, уложенного на некоторую поверхность. Для этого требуется ограничить подходящим образом выбор ребер.

Что касается сложности расщепляющего алгоритма (а, следовательно, и алгоритма Флёри), время его выполнения в худшем случае составляет величину $O(|V| \cdot |E|)$. Так, распознать связность графа можно за время $O(|V|)$. Поскольку операция расщепления применяется тогда и только тогда, когда $\deg(v) > 2$, и так как эта операция в вершине v уменьшает ее степень $\deg(v)$ на 2, то распознавать связность придется не более

$$\sum_{v \in V(G)} \frac{1}{2} (\deg(v) - 2) = |V| - |E|$$

раз. Это и приводит к указанной выше оценке $O(|V| \cdot |E|)$. Следовательно, время выполнения каждого полученного на основе расщепляющего алгоритма будет также не хуже $O(|V| \cdot |E|)$. Эту верхнюю оценку можно улучшить с помощью параллельных вычислений [118].

Наиболее эффективный алгоритм приведен в статье Хирхольцера [109]. Данный алгоритм работает быстрее расщепляющего алгоритма и алгоритма Флёри. Временная и емкостная сложность этого алгоритма составляет величину $O(|E|)$. Тем не менее, алгоритм Хирхольцера был сформулирован только для простых графов.

Описанные выше алгоритмы находят в графе произвольную эйлерову цепь, на которую не наложено никаких ограничений [144]. Анализ публикаций, посвященных задачам построения маршрутов специального вида, показал, что большинство работ посвящено алгоритмам с локальными ограничениями на порядок обхода ребер (например, запрещение левых поворотов; маршруты без поворотов; использование в каждой вершине графа заданного циклического порядка включения ребер в маршрут и т.п.). Как уже отмечалось выше, обобщение большинства частных случаев задачи построения маршрутов с локальными ограничениями дано С. Зейдером [157].

Ограничения на порядок обхода вершин и ребер графа можно классифицировать как

- локальные, когда следующее ребро в маршруте определяется условиями, заданными в текущей вершине или на текущем ребре [109, 117, 118, 124, 156, 157];
- глобальные (эйлеровы, гамильтоновы циклы, бинаправленные двойные обходы [118] и т.д.).

Большинство опубликованных работ посвящено алгоритмам с локальными ограничениями на порядок обхода ребер.

Известны публикации других авторов, в которых также рассматриваются задачи, посвященные эйлеровым цепям специального вида, например, расши-

рение класса запрещенных переходов [157], самонепересекающиеся и непересекающиеся цепи, бинаправленные двойные обходы [109,118], маршруты Петри [160], прямолинейные маршруты [156], реберно-упорядоченные маршруты [113] и т.д.

Большинство задач нахождения маршрутов, удовлетворяющих определенным ограничениям, появились из конкретных практических ситуаций. Например, в упомянутых выше задачах раскроя листового материала **моделью раскройного плана** является **плоский граф**, а маршрут, покрывающий все ребра, определяет **траекторию движения режущего инструмента**. Ограничением является отсутствие пересечения внутренних граней любой начальной части маршрута с ребрами его оставшейся части [135]. При построении систем управления манипуляторами с помощью неориентированного графа отображают всевозможные элементы траектории манипулятора. При этом возникают задачи построения маршрутов, удовлетворяющих различным ограничениям, например: прямолинейных маршрутов [156]; маршрутов, в которых следующее ребро определяется заданным циклическим порядком на множестве ребер, инцидентных текущей вершине [117, 118]; маршрутов, в которых часть ребер следует пройти в заданном порядке [117].

Выводы по главе 1

Известные алгоритмы позволяют построить эйлеровы цепи или покрытия цепями без учета ограничений на порядок обхода ребер. Тем не менее, практика требует построения маршрутов, удовлетворяющих различным ограничениям на последовательность ребер. В частности, ограничения можно классифицировать на:

- локальные, когда следующее ребро в маршруте определяется условиями, заданными в текущей вершине или на текущем ребре (например, исключение запрещенных переходов);

- глобальные (отсутствие пересечения внутренности пройденной части плоского графа с ребрами его непройденной части и т.д.).

Во второй главе будут приведены новые научные результаты для маршрутов с локальными ограничениями, в третьей главе – с глобальными ограничениями, в четвертой – смешанные задачи (построение маршрутов, удовлетворяющих как локальным, так и глобальным ограничениям).

ГЛАВА 2
МАРШРУТЫ С ЛОКАЛЬНЫМИ ОГРАНИЧЕНИЯМИ

Задачи нахождения маршрутов на графах, удовлетворяющих определенным ограничениям, вызываются конкретными практическими потребностями. Как уже отмечалось выше, в настоящее время интенсивно развивается раздел теории графов, касающийся построения маршрутов, удовлетворяющих специальным ограничениям: эйлеровы и гамильтоновы циклы; маршруты, избегающие запрещенных переходов [124,156,157]; самонепересекающиеся и непересекающиеся цепи; бинаправленные двойные обходы [109, 118] и т.д.

Интерес к задачам маршрутизации объясняется их использованием в качестве математических моделей для проблем управления и автоматизации проектирования.

Рассмотрим задачу покрытия графа минимальным числом цепей, удовлетворяющих заданным локальным ограничениям в каждой вершине [157]. Решение данной задачи может быть использовано, например, при поиске маршрутов между заданными точками на карте, удовлетворяющих правилам поворотов на перекрестке либо заданной последовательности проезда по улицам.

2.1 Алгоритм построения допустимой цепи

Обобщение большинства частных случаев задачи построения простой цепи с локальными ограничениями и анализ вычислительной сложности данной проблемы даны С.Зейдером [157]. Приведем основные определения и результаты данной работы.

Ограничимся рассмотрением *конечных простых* графов. Множество вершин и множество ребер графа G будем обозначать соответственно через $V(G)$

и $E(G)$. Для вершины $v \in V(G)$ определим $E_G(v)$, множество всех ребер графа G, инцидентных вершине v. Степень вершины v будем обозначать как $\deg(v)$; для $d > 0$ положим $V_d(G) := \{v \in V(G) \,|\deg(v) = d\}$. Будем писать $H \leq G$, если H – вершинно-индуцированный подграф графа G, т.е. подграф, полученный из графа G отбрасыванием некоторого множества вершин и всех ребер, инцидентных вершинам этого множества, и только их.

Ограничения на маршруты в графе G можно сформулировать в терминах графа разрешенных переходов [55, 57].

Определение 1. *Пусть G – граф.* ***Графом переходов*** *$T_G(v)$ вершины $v \in V(G)$ будем называть граф, вершинами которого являются ребра, инцидентные вершине v, т.е. $V(T_G(v)) = E_G(v)$, а множество ребер – допустимые переходы.*

Определение 2. ***Системой разрешенных переходов*** *(или короче,* ***системой переходов****) T_G будем называть множество $\{T_G(v) \,|\, v \in V(G)\}$, где $T_G(v)$ – граф переходов в вершине v.*

Определение 3. *Путь $P = v_0e_1v_1 \ldots e_kv_k$ в графе G является T_G-****совместимым****, если $\{e_i,\ e_{i+1}\} \in E(T_G(v_i))$ для каждого i $(1 \leq i \leq k-1)$.*

Теорема 4. [С. Зейдер]. *Если все графы переходов принадлежат либо классу M полных многодольных графов, либо классу P паросочетаний, то задача построения T_G-совместимой цепи является разрешимой за время $O(|E(G)|)$. В противном случае данная задача является $\mathcal{NP}$-полной.*

Если система переходов вершины $v \in V(G)$ является паросочетанием, то задача сводится к задаче для графа

$$\begin{aligned} G' :\ V(G') &= V(G)\backslash\{v\}, \\ E(G') &= (E(G)\backslash E_G(v)) \cup \{\{v_iv_j\} :\ \{v_iv,\ vv_j\} \in E(T_G(v))\}. \end{aligned}$$

Если для любой вершины $v \in V(G)$ граф $T_G(v)$ является полным многодольным графом, то цепь можно построить с помощью следующего алгоритма [58].

Алгоритм T_G-СОВМЕСТИМЫЙ ПУТЬ

Входные данные:

- граф $G = (V, E)$;
- вершины x, y, между которыми требуется найти цепь без запрещенных переходов;
- система переходов $T_G : \ (\forall v \in V(G)) \ T_G(v) \in M$.

Выходные данные:

- последовательность ребер, определяющая T_G-совместимый путь между вершинами x и y, либо сообщение об его отсутствии.

Шаг 1. Если вершина x или вершина y является изолированной, останов: пути нет.

Шаг 2. Удалить из графа G изолированные вершины.

Шаг 3. Построить вспомогательный граф G' следующим образом (рисунок 2.1):

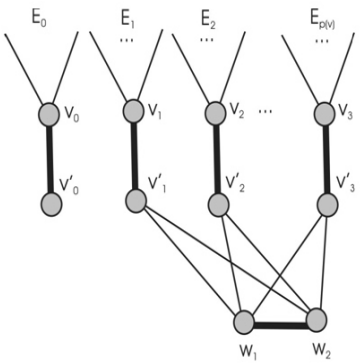


Рисунок 2.1: Иллюстрация построения вспомогательного графа G'

- каждую вершину $v \in V(G)$ расщепить на вершины $v_1, v_2, \ldots, v_{p(v)}$, где $p(v)$ – число долей графа $T_G(v)$. Вершине v_p инцидентны ребра соответствующей доли графа $T_G(v)$ и одна дополнительная вершина $v'_{p(v)}$;

- добавить две новые вершины $w_1(v)$ и $w_2(v)$, ребро $w_1(v)w_2(v)$, и ребро $v'_{p(v)}w_j(v)$ для каждой доли графа $T_G(v)$, $1 \leq j \leq 2$.

Шаг 4. Построить первоначальное паросочетание в графе G'

$$M(G') = \bigcup_{v \in V(G)} \left(\bigcup_{p=1,2,\ldots p(v)} \{v_p v'_p\} \bigcup \{w_1(v)w_2(v)\} \right).$$

Шаг 5. Искать чередующуюся последовательность между вершинами x и y, увеличивающую мощность паросочетания в графе G'. Если такую последовательность найти не удается – останов (паросочетание $M(G')$ имеет максимальную мощность, а граф не имеет T_G-совместимого пути). В противном случае все ребра данного увеличивающего пути за исключением ребер, добавленных при построении графа G', образуют T_G-совместимую цепь между вершинами x и y. Останов.

Покажем на примере графа G, представленного на рисунке 2.2, что

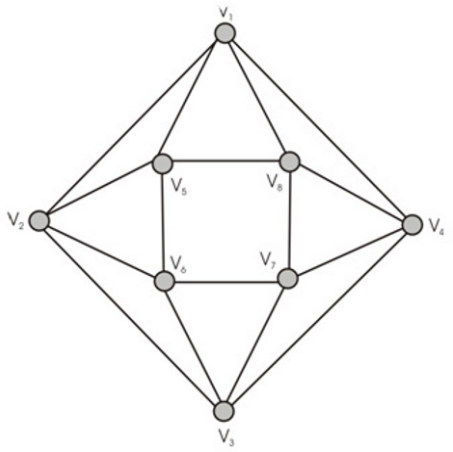


Рисунок 2.2: Пример графа

алгоритм T_G-**СОВМЕСТИМЫЙ ПУТЬ** не может быть использован для построения маршрутов, покрывающих все ребра графа G. Допустим для графа задана следующая система переходов T_G: $\{\{v_2v_1\},\{v_1v_5\}\}$, $\{\{v_6v_1\},\{v_1v_4\}\}$, $\{\{v_4v_3\},\{v_3v_7\}\}$, $\{\{v_8v_3\},\{v_3v_2\}\}$, $\{\{v_3v_2\},\{v_2v_8\}\}$, $\{\{v_5v_2\},\{v_2v_1\}\}$, $\{\{v_1v_4\},\{v_4v_6\}\}$, $\{\{v_7v_4\},\{v_4v_3\}\}$, $\{\{v_2v_5\},\{v_5v_8\}\}$, $\{\{v_2v_8\},\{v_8v_5\}\}$, $\{\{v_3v_8\},\{v_8v_7\}\}$, $\{\{v_3v_7\},\{v_7v_8\}\}$, $\{\{v_4v_7\},\{v_7v_6\}\}$, $\{\{v_4v_6\},\{v_6v_7\}\}$, $\{\{v_1v_6\},\{v_6v_5\}\}$, $\{\{v_1v_5\},\{v_5v_6\}\}$.

Граф G', необходимый для нахождения T_G-совместимого пути между вершинами v_1 и v_7, построение которого описано на **шаге 3** алгоритма, приведен на рисунке 2.3.

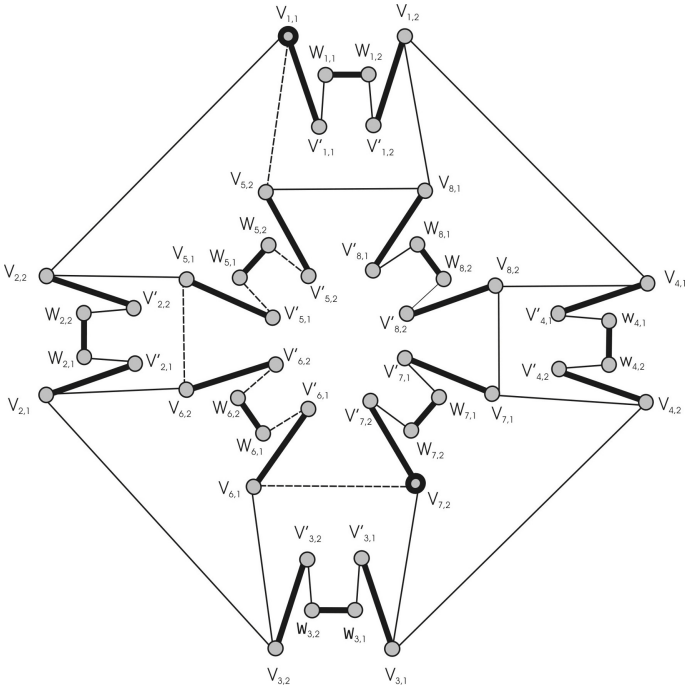


Рисунок 2.3: Граф G', полученный с помощью вспомогательных построений из графа G

Первоначальное паросочетание $M(G')$ выделено на рисунке жирными линиями. Для данного паросочетания чередующейся увеличивающей последовательностью ребер является $\{v_{1,1}v_{5,2}\}$, $\{v_{5,2}v'_{5,2}\}$, $\{v'_{5,2}w_{5,2}\}$, $\{w_{5,2}w_{5,1}\}$, $\{w_{5,1}v'_{5,1}\}$, $\{v'_{5,1}v_{5,1}\}$, $\{v_{5,1}v_{6,2}\}$, $\{v_{6,2}v'_{6,2}\}$, $\{v'_{6,2}w_{6,2}\}$, $\{w_{6,2}w_{6,1}\}$, $\{w_{6,1}v'_{6,1}\}$, $\{v'_{6,1}v_{6,1}\}$, $\{v_{6,1}v_{7,2}\}$. Ребра этой последовательности, не вошедшие в первоначальное паросочетание, изображены пунктирной линией. Эти ребра образуют множество

$$\left\{\{v_{1,1}v_{5,2}\},\ \{v'_{5,2}w_{5,2}\},\ \{w_{5,1}v'_{5,1}\},\ \{v_{5,1}v_{6,2}\},\ \{w_{6,1}v'_{6,1}\},\ \{v_{6,1}v_{7,2}\}\right\}.$$

Все ребра данного множества, принадлежащие графу G, т.е. $\{v_1v_5\}$, $\{v_5v_6\}$, $\{v_6v_7\}$, образуют T_G-совместимый путь из вершины v_1 в вершину v_7.

С помощью алгоритма T_G-**СОВМЕСТИМЫЙ ПУТЬ** возможно построение только простой цепи между двумя различными вершинами (т.е. цепи, в которых любая вершина встречается ровно один раз).

Однако в общем случае непосредственное применение данного алгоритма не позволяет решить задачу нахождения T_G-совместимого маршрута, содержащего максимальное число ребер. Действительно, паросочетание максимальной мощности в графе G' не может содержать пары ребер, образующих запрещенный переход, т.к. они инцидентны одной общей вершине графа G'. В то же время, в общем случае может существовать T_G-совместимый маршрут, содержащий такую пару ребер. Заметим, что в работе С. Зейдера [157] остался открытым вопрос распознавания многодольности графов $T_G(v)$, а также проблема построения допустимого маршрута или множества маршрутов, покрывающих все ребра исходного графа.

Например, в графе G, приведенном на рисунке 2.2, маршрут

$$\{v_2v_1\}, \{v_1v_4\}, \{v_4v_8\}, \{v_8v_1\}, \{v_1v_5\}, \{v_5v_2\}$$

принципиально не может быть получен с помощью построения паросочетания максимального веса в графе G'. Этот маршрут начинается с ребра v_2v_1, а заканчивается ребром v_5v_2, которые образуют запрещенный переход $\{v_5v_2\}, \{v_2v_1\}$, следовательно, в графе G' не существует чередующегося пути, содержащего оба эти ребра.

Таким образом, открытым остался вопрос распознавания многодольности графов $T_G(v)$, а также задача построения допустимого маршрута или множества маршрутов, покрывающих все ребра исходного графа.

2.2 Алгоритм построения допустимой эйлеровой цепи

В предыдущем разделе были сформулированы ограничения на допустимость маршрутов в терминах системы разрешенных переходов [157] и показано, что задача построения допустимого пути в графе G разрешима за полиномиальное время, если система переходов T_G содержит только паросочетания и полные многодольные графы. Распознавание принадлежности

графа разрешенных переходов классу паросочетаний тривиально. Для распознавания принадлежности графа переходов классу полных многодольных графов целесообразно использовать понятие системы разбиения [109, 124].

Понятие системы разбиения используется для определения допустимой цепи в терминах запрещенных переходов.

Определение 4. *Пусть дан граф* $G = (V, E)$. *Пусть* $P_G(v)$ – *некоторое разбиение множества* $E_G(v)$. ***Системой разбиения*** *графа* G *будем называть систему множеств* $P_G := \{P_G(v) \mid v \in V(G)\}$.

Определение 5. *Пусть* $p \in P_G(v)$, $\{e, f\} \in p$. *Цепь, не содержащую переходов* $e \to v \to f$ *и* $f \to v \to e$, *будем называть* P_G***-совместимой***, *а переходы* $e \to v \to f$ *и* $f \to v \to e$ – ***запрещенными***.

Заметим, что граф разрешенных переходов $T_G(v)$ однозначно определяет граф запрещенных переходов $\overline{T_G}(v)$, который является дополнением графа разрешенных переходов до полного графа. Таким образом, с помощью определений 1–3 можно поставить задачу с любым графом разрешенных (запрещенных) переходов.

Напротив, граф разрешенных переходов, определяемый с помощью системы разбиения P_G, не может быть произвольным, а принадлежит классу M полных многодольных графов: элементы разбиения $P_G(v)$ определяют доли графа $T_G(v) \in M$, а множество его ребер

$$E(T_G(v)) = \{e, f \in E_G(v) : (\forall p \in P_G(v))\, \{e, f\} \not\subset p\}.$$

Графом запрещенных переходов $\overline{T_G}(v)$ в данном случае будет являться набор из $|P_G(v)|$ клик, этот факт может быть использован для распознавания принадлежности $T(v) \in M$ с помощью следующего алгоритма.

Алгоритм РАЗМЕТКА

Входные данные: граф переходов $T_G(v)$.

Шаг 1. Объявить все вершины графа $T_G(v)$ непомеченными. Положить $l = 1$.

Шаг 2. Пока список непомеченных вершин не пуст, выполнять шаги 3, 4 и 5. В противном случае – останов: граф $T_G(v)$ является многодольным и вершины, принадлежащие одному элементу разбиения множества вершин, имеют одинаковые пометки.

Шаг 3. Найти некоторую непомеченную вершину v. Присвоить ей пометку l.

Шаг 4. Применить волновой алгоритм для присваивания пометки l всем вершинам, достижимым из вершины v в графе $\overline{T_G}(v)$. Очевидно, что все помеченные на данном шаге вершины будут принадлежать одной компоненте связности графа $\overline{T_G}(v)$.

Шаг 5. Если в выделенной компоненте связности любая пара вершин является смежной, то найденная компонента связности является кликой. Положить $l = l + 1$ и перейти к выполнению шага 3. В противном случае – останов: граф $T_G(v)$ не является многодольным.

Оценим сложность приведенного алгоритма. Расстановка пометок в конкретной компоненте связности T_k составляет величину $O(|E(T_k)|)$. Проверка, является ли данная компонента связности кликой, также требует не более $O(|E(T_k)|)$ операций. Таким образом, сложность алгоритма РАЗМЕТКА равна

$$O\left(\sum_{\forall k} |E(T_k)|\right) = O(|E(T)|).$$

Как было отмечено, алгоритм С. Зейдера в общем случае не позволяет строить допустимые цепи максимальной длины. Особый интерес представляют допустимые эйлеровы цепи. Необходимое и достаточное условие существования P_G-совместимых цепей дает следующая теорема [124].

Теорема 5. [А. Коциг]. *Связный эйлеров граф G имеет P_G-совместимую эйлерову цепь тогда и только тогда, когда*

$$(\forall v \in V)\,(\,\forall p \in P_G(v))\left(|p| \leq \frac{1}{2} d_G(v)\right).$$

Очевидно, что сложность проверки условия существования P_G-совместимой эйлеровой цепи не превосходит величины $O(|E(G)|)$. Ниже приведен алгоритм построения совместимой цепи [153].

Алгоритм P_G-СОВМЕСТИМАЯ ЭЙЛЕРОВА ЦЕПЬ

Входные данные:

- эйлеров граф $G = (V, E)$, заданный списком смежности для каждой вершины;
- система переходов $P_G(v)$ $\forall v \in V(G)$: в списке смежности вершины, относящиеся к одному элементу разбиения, имеют одинаковые пометки.

Выходные данные:

- допустимый эйлеров цикл G_{k+1}.

Шаг 1. Положить $k = 0$, $G_k = G$.

Шаг 2. Найти вершину v, у которой $d_{G_k}(v) > 2$.

Шаг 3. Найти элемент разбиения, который содержит максимальное число ребер:

- просмотреть список смежности текущей вершины v;
- посчитать число вхождений в этот список каждого элемента разбиения;
- выбрать тот элемент, который встречается чаще: получим класс $C_1 \in P_{G_k}(v) : |C_1| = \{\max |C| \, | C \in P_{G_k}(v)\}$.

Шаг 4. Выбрать ребра $e_1(v) \in C_1$ и $e_2(v) \in E_{G_k}(v) - C_1$. По возможности выбрать ребра e_1 и e_2, инцидентные вершинам, степень которых больше двух. Если множество $E_{G_k}(v) - C_1 = \emptyset$, останов: P_G-совместимой эйлеровой цепи не существует. В противном случае перейти на шаг 5.

Шаг 5. Построить граф G_{k+1}, отщепив от вершины v вершину $\widehat{v}$, которой инцидентны только ребра e_1 и e_2. Остальные ребра оставить инцидентными вершине v. Так как новая вершина имеет степень 2, то она не рассматривается на последующих шагах работы алгоритма.

Шаг 6. Выбрать класс $C_2 \in P_{G_k}(v)$, которому принадлежит ребро $e_2(v)$. Исключить из системы разбиения вершины v классы C_1 и C_2. Для этого нужно найти $P^-_{G_k}(v) := P_{G_k}(v) - \{C_1, C_2\}$.

Для дальнейшей модификации системы разбиений выполнить следующие действия.

Шаг 6.1. Системы разбиения, в которых отсутствует вершина v, перейдут в модифицированную систему полностью без изменений.

Шаг 6.2. Если системы C_1 и C_2 состояли из одного ребра: $|C_1| = |C_2| = 1$, то $P'_{G_{k+1}}(v) = P^-_{G_k}(v)$.

Шаг 6.3. Если $|C_1| > |C_2| = 1$, то $P'_{G_{k+1}}(v) = P^-_{G_k}(v) \cup \{C_1 - \{e_1(v)\}\}$.

Шаг 6.4. Если $|C_2| > 1$, то $P'_{G_{k+1}}(v) = P^-_{G_k}(v) \cup \{C_1 - \{e_1(v)\}, C_2 - \{e_2(v)\}\}$.

Шаг 6.5. Построить

$$P_{G_{k+1}} = \bigcup_{x \in V(G_{1,2})} P'_{G_{k+1}}(x).$$

Шаг 7. Определить значение $\sigma(G_{k+1}) = 2(|E(G_{k+1})| - |V(G_{k+1})|)$. Заметим, что количество ребер графа остается неизменным, а количество вершин на каждой итерации увеличивается на единицу.

Шаг 8. Если $\sigma(G_{k+1}) > 0$, положить $k = k + 1$, перейти на шаг 2, для графа G_{k+1}. В противном случае перейти на шаг 9.

Шаг 9. Выбрать любую вершину v и пометить все вершины достижимые из данной. Если остались непомеченные вершины перейти на шаг 10, иначе останов – построенный граф G_{k+1} является эйлеровой цепью, не содержащей запрещенных переходов.

Шаг 10. Из списка помеченных и не помеченных вершин графа G_{k+1} найти вершины v_1 и v_2, отщепленные от одной вершины v графа G_0 и объединить их в вершину $v_{1,2}$. Получим модифицированный граф $\hat{G}_{k+1}$, положим $k = k + 1$.

Шаг 11. Выбрать ребра $e_1(v_{1,2}) \in C_1$ и $e_2(v_{1,2}) \in E_{G_k}(v_{1,2}) - C_1$, так чтобы $\{e_1, e_2\} \neq E(v_1)$ и $\{e_1, e_2\} \neq E(v_2)$. Если множество $E_{G_k}(v_{1,2}) - C_1 = \emptyset$,

останов: P_G-совместимой эйлеровой цепи не существует. В противном случае построить граф G_{k+1}, отщепив от вершины $v_{1,2}$ вершину $\hat{v}_{1,2}$, которой инцидентны только ребра e_1 и e_2. Остальные ребра оставить инцидентными вершине $v_{1,2}$ и перейти на шаг 9.

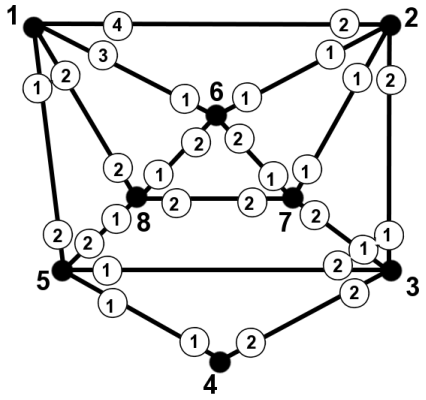
В [56] доказана следующая теорема.

Теорема 6. *Алгоритм P_G-СОВМЕСТИМАЯ ЭЙЛЕРОВА ЦЕПЬ корректно решает задачу построения $P(G)$-совместимой эйлеровой цепи.*

Доказательство. Если для некоторого k, такого что $C'' \in P'_{G_{k+1}}(v)$, выполнено неравенство $|C''| > |C_1 - \{e_1(v)\}|$, то $C'' \in P_{G_{k+1}}(v)$ и $|C_2| \leq |C''| = |C_1| \leq \frac{1}{2}d_{G_k}(v) - 1 = \frac{1}{2}d_{G_{k+1}}(v)$. На основании этого факта можно заключить, что $|C| \leq \frac{1}{2}d_{G_{k+1}}(v)$ для каждой вершины $v \in V(G_{k+1})$ и каждого класса $C \in P_{G_{k+1}}(v) \subset P_{G_{k+1}}$. При этом величина $\sigma(G_{k+1}) = |E(G_{k+1})| - |V(G_{k+1})| < \sigma(G_k)$. Если граф G_{k+1} является циклом, то число ребер в нем и число вершин совпадает, т.е. в данном случае $\sigma(G_{k+1}) = 0$. Если же граф G_{k+1} является цепью, то число вершин превышает число ребер на 2, следовательно, в данном случае $\sigma(G_{k+1}) = -2$. Если же на некотором этапе для $e_1(v) \in C_1$ не удалось найти $e_2(v) \in E_{G_k}(v) - C_1$, это значит, что $|C_1| > \deg(v)/2$, т.е. не выполнены необходимые и достаточные условия существования эйлерова цикла (теорема Коцига). Из этих фактов следует корректность выполнения алгоритма. **Теорема доказана.**

Оценим вычислительную сложность предложенного алгоритма. Выполнения шагов 2, 5, 6 и 7 можно организовать с использованием не более $O(1)$ операций (за счет специальных структур данных). Выполнения же шагов 3 и 4 можно организовать с использованием не более $O(\deg(v)_{G_k})$ операций. Цикл алгоритма будет повторен не более, чем $\sigma(G)$ раз. В итоге имеем, что алгоритм потребует число операций не более

$$O\left(\sum_{k=0,1\ldots\sigma(G)} \deg(v_k)_{G_k}\right) = O\left(|E(G)| \cdot |V(G)|\right).$$



1	(2,2)	(6,1)	(8,2)	(5,1)
2	(1,2)	(6,1)	(7,1)	(3,2)
3	(2,1)	(7,1)	(5,2)	(4,2)
4	(3,2)	(5,1)		
5	(4,1)	(3,1)	(8,2)	(1,2)
6	(1',1)	(2,1)	(7,2)	(8,2)
7	(2,2)	(3,1)	(6,1)	(8,2)
8	(1,2)	(6,1)	(7,2)	(5,2)

Рисунок 2.4: Пример графа и заданная система переходов

Таким образом, приведенные алгоритмы разрешимы за полиномиальное время и могут быть легко реализованы с помощью стандартных вычислительных средств.

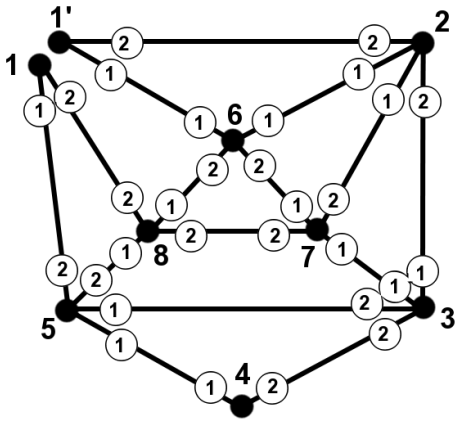
Рассмотрим построение допустимой эйлеровой цепи для графа, приведенного на рисунке 2.4 [59].

Построим допустимый эйлеров цикл, начинающийся и заканчивающийся в вершине 1. На первой итерации будет осуществлено расщепление начальной вершины. Для простоты на рисунках 2.5–2.11, иллюстрирующих пример, не показаны дополнительные построения при расщеплении вершин.

На рисунке 2.5 приведен граф, для которого начальная вершина расщеплена, а также список связности, в котором серым цветом помечены клетки с новыми или модифицированными элементами.

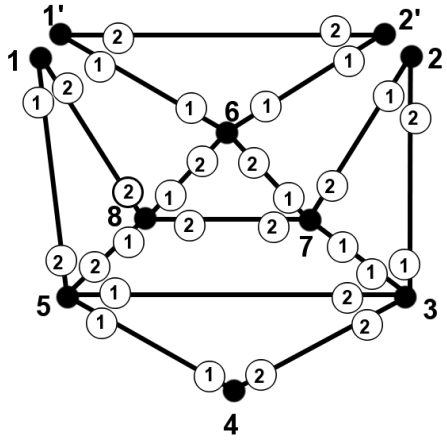
На рисунках 2.6–2.11 приведены последующие итерации работы алгоритма.

В результате граф оказывается расщеплен в простой цикл, из любой вершины которого удается построить допустимый эйлеров цикл, удовлетворяющий введенным ограничениям. Например, из вершины 1 может быть построен следующий цикл: $1 \to 2 \to 6 \to 8 \to 1 \to 5 \to 3 \to 2 \to 7 \to 3 \to 4 \to 5 \to 8 \to 7 \to 6 \to 1$.



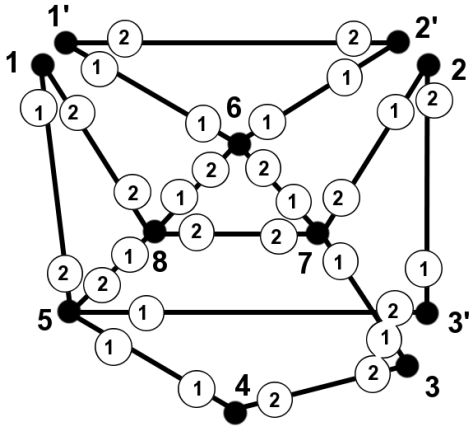
1	(8,2)	(5,1)		
2	**(1',2)**	(6,1)	(7,1)	(3,2)
3	(2,1)	(7,1)	(5,2)	(4,2)
4	(3,2)	(5,1)		
5	(4,1)	(3,1)	(8,2)	(1,2)
6	**(1',1)**	(2,1)	(7,2)	(8,2)
7	(2,2)	(3,1)	(6,1)	(8,2)
8	**(2,2)**	**(6,1)**		

Рисунок 2.5: Первая итерация алгоритма



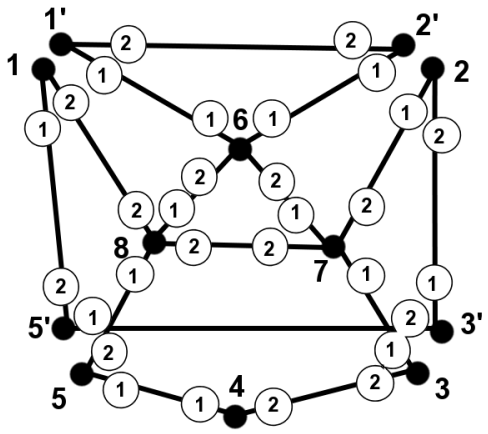
1	(8,2)	(5,1)		
2	(7,1)	(3,2)		
3	(2,1)	(7,1)	(5,2)	(4,2)
4	(3,2)	(5,1)		
5	(4,1)	(3,1)	(8,2)	(1,2)
6	(1',1)	**(2',1)**	(7,2)	(8,2)
7	(2,2)	(3,1)	(6,1)	(8,2)
8	(1,2)	(6,1)	(7,2)	(5,2)
1'	**(2',2)**	(6,1)		
2'	**(1',2)**	**(6,1)**		

Рисунок 2.6: Вторая итерация алгоритма



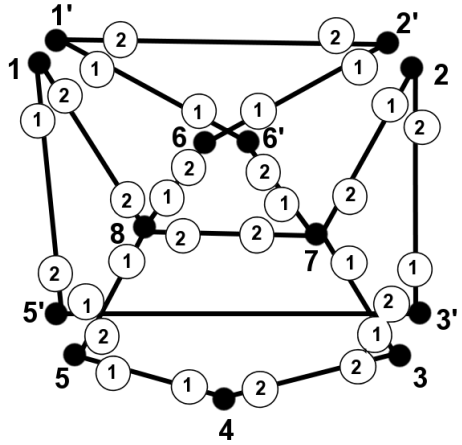
1	(8,2)	(5,1)		
2	(7,1)	**(3',2)**		
3	(7,1)	(4,2)		
4	(3,2)	(5,1)		
5	(4,1)	**(3',1)**	(8,2)	(1,2)
6	(1',1)	**(2',1)**	(7,2)	(8,2)
7	(2,2)	(3,1)	(6,1)	(8,2)
8	(1,2)	(6,1)	(7,2)	(5,2)
1'	(2',2)	(6,1)		
2'	(1',2)	(6,1)		
3'	**(2,1)**	**(5,2)**		

Рисунок 2.7: Третья итерация алгоритма



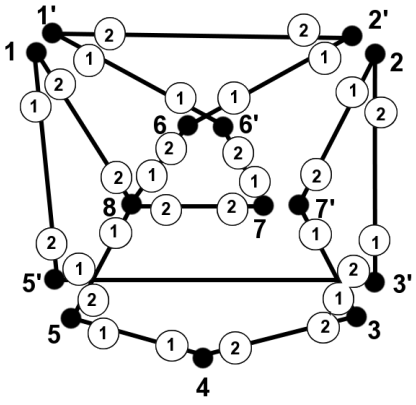
1	(8,2)	(5,1)		
2	(7,1)	(3',2)		
3	(7,1)	(4,2)		
4	(3,2)	**(5',1)**		
5	(3',1)	(1,2)		
6	(1',1)	(2',1)	(7,2)	(8,2)
7	(2,2)	(3,1)	(6,1)	(8,2)
8	(1,2)	(6,1)	(7,2)	**(5',2)**
1'	(2',2)	(6,1)		
2'	(1',2)	(6,1)		
3'	(2,1)	(5,2)		
5'	**(4,1)**	**(8,2)**		

Рисунок 2.8: Четвертая итерация алгоритма



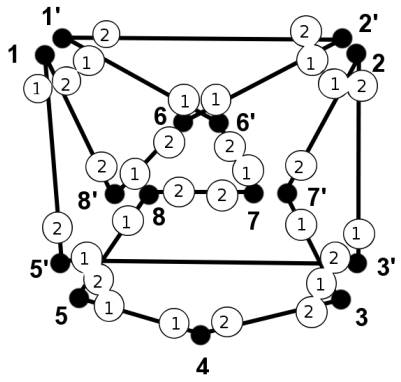
1	(8,2)	(5,1)		
2	(7,1)	(3',2)		
3	(7,1)	(4,2)		
4	(3,2)	(5',1)		
5	(3',1)	(1,2)		
6	(2',1)	(8,2)		
7	(2,2)	(3,1)	**(6',1)**	(8,2)
8	(1,2)	(6,1)	(7,2)	**(5',2)**
1'	(2',2)	**(6',1)**		
2'	(1',2)	(6,1)		
3'	(2,1)	(5,2)		
5'	(4,1)	(8,2)		
6'	**(1',1)**	**(7,2)**		

Рисунок 2.9: Пятая итерация алгоритма



1	(8,2)	(5,1)		
2	**(7',1)**	(3',2)		
3	**(7',1)**	(4,2)		
4	(3,2)	(5',1)		
5	(3',1)	(1,2)		
6	(2',1)	(8,2)		
7	(6',1)	(8,2)		
8	(1,2)	(6,1)	(7,2)	**(5',2)**
1'	(2',2)	**(6',1)**		
2'	(1',2)	(6,1)		
3'	(2,1)	(5,2)		
5'	(4,1)	(8,2)		
6'	(1',1)	(7,2)		
7'	**(2,2)**	**(3,1)**		

Рисунок 2.10: Шестая итерация алгоритма



1	**(8',2)**	(5,1)
2	(7',1)	(3',2)
3	(7',1)	(4,2)
4	(3,2)	(5',1)
5	(3',1)	(1,2)
6	(2',1)	**(8',2)**
7	(6',1)	(8,2)
8	(7,2)	(5',2)
1'	(2',2)	(6',1)
2'	(1',2)	(6,1)
3'	(2,1)	(5,2)
5'	(4,1)	(8,2)
6'	(1',1)	(7,2)
7'	(2,2)	(3,1)
8'	**(1,2)**	**(6,1)**

Рисунок 2.11: Седьмая (последняя) итерация алгоритма

2.3 Покрытие графа допустимыми цепями

Рассмотрим задачу покрытия графа допустимыми цепями. Будем считать, что система переходов T_G содержит только паросочетания и полные многодольные графы [56].

Алгоритм ПОКРЫТИЕ T_G-ДОПУСТИМЫМИ ЦЕПЯМИ

Входные данные:

- граф $G = (V, E)$,
- графы переходов $T_G(v)$ $\forall v \in V(G)$.

Выходные данные:

- набор цепей T^i, $i = 1, 2, \ldots, k$, покрывающих граф G, где $m = 2k$ – число вершин нечетной степени.

Шаг 1. Пусть $U = \{v \in V(G) \ : \ T_G(v) - \text{паросочетание}\}$. Сделать редукцию графа G до графа G':

$$V(G') = V(G) \backslash U,$$

$$E(G') = \left(E(G) \backslash \bigcup_{v \in U} E_G(v) \right) \bigcup \left\{ \bigcup_{v \in U} \{\{v_i v_j\} : \ \{v_i v, \ v v_j\} \in T_G(v)\} \right\},$$

графы $T_G(v)$ редуцировать до графов $T_{G'}(v)$ заменой всех вхождений вершин $u \in U$: $vu, wu \in E_{T_G}(u)$ вершиной w.

Шаг 2. Достроить граф G' до G^* введением дополнительной вершины v^*, смежной всем вершинам нечетной степени графа G'. Систему переходов

$T_{G'}(v)$ модифицировать до системы переходов T_{G^*} введением для всех $v \in V'(G) : \deg(v) \equiv 1 \pmod 2$ в граф переходов $T_{G^*}(v)$ вершины vv^*, смежной всем вершинам в графе $T_{G^*}(v)$.

Шаг 3. Для всех таких вершин $v \in V(G)$, что $\exists p \in P(v)$: $|p| > \deg(v)/2$, ввести $2|p| - \deg(v)$ дополнительных ребер $(vv^*)_i$, $i = 1, 2, ..., 2|p| - \deg(v)$ в граф G^*. Модифицировать граф переходов $T_{G^*}(v)$ введением вершин $(vv^*)_i$, смежных всем вершинам исходного графа $T_{G^*}(v)$ и только им.

Шаг 4. Найти в G^* T_{G^*}-совместимый эйлеров цикл T^*.

Шаг 5. Построить покрытие T' графа G' цепями, удалив из T^* ребра (vv^*).

Шаг 6. Модифицировать маршруты из T' до маршрутов из T добавлением вершин $u \in U$, удаленных на шаге 1.

Шаг 7. Останов.

Теорема 7. *Алгоритм* ***ПОКРЫТИЕ*** T_G***-ДОПУСТИМЫМИ ЦЕПЯМИ*** *корректно решает задачу минимального по мощности покрытия графа* T_G*-допустимыми цепями. Его сложность не превосходит величины* $O(|E(G)| \cdot |V(G)|)$.

Доказательство. В результате выполнения шага 1 приходим к задаче для полного многодольного графа G'. Данное преобразование возможно выполнить, используя не более $O(|E(G)|)$ операций.

В результате выполнения шага 2 получаем задачу для эйлерова графа, в каждой вершине v которого граф переходов $T_{G^*}(v)$ является полным многодольным. Введенная в граф $T_{G^*}(v)$ дополнительная вершина vv^* является отдельным элементом разбиения в $P_{G^*}(v)$.

На шаге 3 проверяется выполнение необходимых и достаточных условий существования допустимого эйлерова цикла (теорема Коцига). Во всех вершинах, где условия теоремы Коцига не выполнены, в граф G^* добавляются мультиребра $(vv^*)_i$, $i = 1, 2, ..., 2|p(v)| - \deg(v)$. Также модифицируется и

система разбиения $P_{G^*}(v)$ добавлением элемента разбиения, содержащего все ребра $(vv^*)_i$. Такие модификации также выполняются за время, не превосходящее $O(|V(G)| \cdot |E(G)|)$.

В результате проведенных модификаций граф G^* будет эйлеровым, а его система разбиения будет удовлетворять теореме Коцига.

Для построения допустимого эйлерова цикла, содержащего и дополнительные ребра, смежные v^*, на шаге 4 требуется не более $O(|V(G)| \cdot |E(G)|)$ операций.

На шаге 5 получим $l = \deg(v^*)$ простых цепей удалением ребер, инцидентных вершине v^*, которые были добавлены на шагах 2 и 3. Все полученные таким образом цепи будут $P_{G'}$-допустимыми в графе G'. Сложность этого этапа составляет величину $O(|E(G)|)$.

На шаге 6 происходит добавление удаленных на шаге 1 вершин, что также требует не более чем $O(|E(G)|)$ операций вставки.

В результате выполненных операций получим покрытие графа $l + 1$ цепями за время $O(|V(G)| \cdot |E(G)|)$. Предположение существования покрытия с меньшим числом цепей приведет к противоречию с теоремой Коцига. **Теорема доказана.**

Выводы по главе 2

1. Показана возможность распознавание системы переходов, которая позволяет решить задачу построения допустимого пути за линейное время.
2. Доказано, что с помощью разработанного алгоритма P_G-**СОВМЕСТИМАЯ ЭЙЛЕРОВА ЦЕПЬ** в эйлеровом графе G возможно построить P_G-совметимый эйлеров цикл или установить его отсутствие за время $O(|V(G)| \cdot |E(G)|)$.

3. Покрытие графа G допустимыми цепями также возможно за время $O(|V(G)| \cdot |E(G)|)$ с помощью алгоритма **ПОКРЫТИЕ T_G-ДОПУСТИМЫМИ ЦЕПЯМИ**.
4. Разработанное программное обеспечение позволяет решать задачи построения допустимой цепи и допустимого эйлерова цикла.

ГЛАВА 3
МАРШРУТЫ С УПОРЯДОЧЕННЫМ ОХВАТЫВАНИЕМ

В данной главе рассмотрена задача построения маршрутов специального вида (представляющих класс OE-маршрутов). Прикладной стороной данной задачи является задача построения маршрута движения инструмента при разрезании листового материала. Моделью раскройного листа будем считать плоскость S, моделью раскройного плана – плоский граф G с внешней гранью f_0 на плоскости S. Для любой части графа $J \subseteq G$ (части траектории движения режущего инструмента) обозначим через $\mathsf{Int}\,(J)$ теоретико-множественное объединение его внутренних граней (объединение всех связных компонент $S \setminus J$, не содержащих внешней грани). Тогда $\mathrm{Int}(J)$ можно интерпретировать как отрезанную от листа часть. Множества вершин, ребер и граней графа J будем обозначать через $V(J)$, $E(J)$ и $F(J)$ соответственно, а через $|M|$ – число элементов множества M.

Будем любой маршрут в графе G рассматривать как часть графа, содержащую все вершины и ребра, принадлежащие маршруту. Это позволяет формализовать требование к маршруту режущего инструмента как условие отсутствия пересечения внутренних граней любой начальной части маршрута в заданном плоском графе G с ребрами его оставшейся части [135]. Такие маршруты будем называть маршрутами с упорядоченным охватыванием [69,141] (или для кратости OE-маршрутами, где OE – от англ. «ordered enclosing»).

Определение 6. *Будем говорить, что цикл $C = v_1e_1v_2e_2\dots v_k$ в эйлеровом графе G имеет* ***упорядоченное охватывание*** *(является OE-циклом), если для любой его начальной части $C_i = v_1e_1v_2e_2\dots e_i$, $i \le (|E(G)|)$ выполне-*

но условие

$$\mathrm{Int}\,(C_i) \cap G = \emptyset.$$

Например, для плоского эйлерова графа, приведенного на рисунке 3.1, цикл $v_1e_1v_3e_3v_2e_2v_1e_4v_3e_5v_2e_6v_1$ удовлетворяет условию упорядочен-

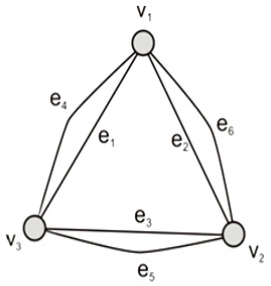


Рисунок 3.1: Пример эйлерова графа

ного охватывания, а цикл $v_1e_4v_3e_5v_2e_6v_1e_1v_3e_3v_2e_2v_1$ – не удовлетворяет, т.к. $\mathrm{Int}\,(v_1e_4v_3e_5v_2e_6v_1) \supset \{e_1, e_2, e_3\}$.

Если представлению раскройного плана соответствует плоский эйлеров граф G, то его можно представить без холостых проходов [137]. Если же соответствующий граф G не является эйлеровым и содержит $2k$ вершин нечетной степени, то с помощью алгоритма Листинга-Люка [109, 118] возможно покрыть граф k цепями. Алгоритм построения покрытия, удовлетворяющего введенным ограничениям, был предложен в [86]. Маршруты, которые реализуют построенное покрытие, содержат дополнительные ребра между концом текущей цепи и началом последующей. Однако указанные выше алгоритмы не учитывают длину дополнительных построений.

В практических задачах актуальным является сокращение длины дополнительных построений.

Ниже рассмотрим вопросы построения последовательности OE-цепей с минимальной длиной дополнительных построений.

Для сохранения целостности изложения приведем основные определения и доказанные ранее свойства эйлеровых покрытий плоского графа последовательностью OE-цепей.

3.1 Представление плоского графа

Для представления образа раскройного плана в виде плоского графа $G = (V, F, E)$ определим для каждого ребра $e \in E(G)$ функции, представленные в разделе 1.1. Это минимальная информация, необходимая для представления любого плоского графа с точностью до гомеоморфизма.

Пространственная сложность такого представления будет $O(|E| \cdot \log |V|)$.

Поскольку функции $v_k(e)$, $f_k(e)$, $l_k(e)$, $r_k(e)$, $k = 1, 2$, построенные на ребрах графа $G = (V, F, E)$, для каждого ребра определяют инцидентные вершины, инцидентные грани и смежные ребра, то справедливо следующее предложение.

Утверждение 7. *Функции $v_k(e)$, $f_k(e)$, $l_k(e)$, $r_k(e)$, $k = 1, 2$ построенные на ребрах графа $G = (V, F, E)$ определяют плоский граф $G = (V, F, E)$ с точностью до гомеоморфизма.*

Таким образом, используя известные координаты прообразов вершин графа $G = (V, F, E)$ и размещения фрагментов раскройного плана, являющихся прообразами ребер графа $G = (V, F, E)$, любой маршрут в графе $G = (V, E)$ можно интерпретировать как траекторию режущего инструмента.

Представление графа фактически задает ориентацию его ребер. Далее предполагается, что движение по ребру для определенности осуществляется от вершины $v_2(e)$ к вершине $v_1(e)$. Поскольку при задании графа G неизвестно, какое из ребер в каком направлении будет пройдено, то при выполнении алгоритма производится перестановка значений полей $v_1(e)$, $v_2(e)$ и $l_1(e)$, $l_2(e)$ некоторых ребер. В алгоритме данную процедуру выполняет функция `REPLACE` (алг.1). Функциональным назначением функции является перестановка индексов функций $v_k(e)$ и $l_k(e)$ на $3 - k$, $k = 1, 2$.

Определение 7. *[135]. Цепь $C = v_1 e_1 v_2 e_2 \dots v_k$ в плоском графе G имеет* ***упорядоченное охватывание*** *(является OE-цепью), если для любой его*

Algorithm 1 Функция REPLACE

```
1: procedure REPLACE(In: Ret.Last – ребро, для которого нужно поменять функции местами)
2:     tmp1 = v_2[Edge]; tmp2 = l_2[Edge];
3:     v_2[Edge] = v_1[Edge]; l_2[Edge] = l_1[Edge];
4:     v_1[Edge] = tmp1; l_2[Edge] = tmp2;
5: end procedure
```

начальной части $C_l = v_1 e_1 v_2 e_2 \dots e_l$, $l \leq (|E|)$ *выполнено условие* $\mathrm{Int}\,(C_l) \cap G = \emptyset$.

Определение 8. *Упорядоченная последовательность реберно-непересекающихся OE-цепей*

$$C^0 = v^0 e_1^0 v_1^0 e_2^0 \dots e_{k_0}^0 v_{k_0}^0, \quad C^1 = v^1 e_1^1 v_1^1 e_2^1 \dots e_{k_1}^1 v_{k_1}^1, \dots,$$
$$C^{n-1} = v^{n-1} e_1^{n-1} v_1^{n-1} e_2^{n-1} \dots e_{k_{n-1}}^{n-1} v_{k_{n-1}}^{n-1},$$

покрывающая граф G *и такая, что*

$$(\forall m : m < n), \quad \left(\bigcup_{l=0}^{m-1} \mathrm{Int}(C^l)\right) \cap \left(\bigcup_{l=m}^{n-1} C^l\right) = \emptyset$$

называется ***маршрутом с упорядоченным охватыванием*** *(OE-маршрутом).*

Построение OE-маршрута графа G решает поставленную задачу раскроя. Наибольший интерес представляют маршруты с минимальным числом цепей, поскольку переход от одной цепи к другой соответствует холостому проходу режущего инструмента.

Определение 9. *Маршрут, содержащий минимальную по мощности упорядоченную последовательность реберно-непересекающихся OE-цепей в плоском графе* G *будем называть* ***эйлеровым маршрутом с упорядоченным охватыванием*** *(эйлеровым OE-маршрутом), а составляющие его OE-цепи –* ***эйлеровым OE-покрытием****.*

3.2 Существование эйлеровых OE-циклов

Существование эйлеровых OE-циклов в плоских эйлеровых графах доказано в работах [135, 137]. Доказательство конструктивно и использует алгоритм, являющийся аналогом алгоритма из работы [133].

Теорема 8. *Пусть G – плоский эйлеров граф. Для любой вершины $v \in V(G)$, инцидентной границе внешней (бесконечной) грани графа G, существует эйлеров OE-цикл $C = ve_1v_1e_2v_2 \dots v_{|E|-1}e_{|E|}v$.*

Доказательство. Воспользуемся методом математической индукции по числу граней графа G.

Эйлеровы графы, содержащие две грани, являются простыми циклами. Для простых циклов справедливость утверждения теоремы очевидна.

Предположим, что любой плоский эйлеров граф с числом граней m: $2 < m < K$ имеет OE-цикл для любой его вершины $v \in V(G)$, принадлежащей внешней грани. Рассмотрим эйлеров граф, имеющий K граней. Не уменьшая общности рассуждений, будем считать, что степени всех вершин графа G больше или равны четырем.

Пусть f_0 – внешняя грань графа G, а $C(f_0) = v_1e_1v_2e_2 \dots e_iv_1$ представляет цикл из ребер графа G, инцидентных внешней грани f_0. Далее множество вершин цикла $C(f_0)$ будем обозначать через $V(C(f_0))$, а множество ребер – через $E(C(f_0))$.

Граф $\tilde{G} = G \setminus E(C(f_0))$, полученный удалением из графа G ребер, ограничивающих грань f_0, содержит не более $|V(C(f_0))|$ компонент связности. Цикл $C(f_0)$ естественным образом определяет линейный порядок L на множестве $V(C(f_0))$. Пусть $T \subset V(C(f_0))$ – семейство L-минимальных вершин-представителей всех компонент связности подграфа $\tilde{G}(t)$, $t \in T$ графа $\tilde{G}$ (т.е. $\forall v \in V(\tilde{G}(t)) \cap V(C(f_0))$ имеет место tLv). Очевидно, что $\tilde{G}(t)$, $t \in T$ – это плоские эйлеровы графы, содержащие менее K граней, у которых вершина

t принадлежит границе внешней грани. Следовательно, каждый из графов $\tilde{G}(t)$, $t \in T$ имеет эйлеров цикл с t-упорядоченным охватыванием (OE-цикл, начинающийся в вершине t).

Рассмотрим маршрут R, полученный заменой в цикле $C(f_0)$ каждой вершины $t \in T$ эйлеровым циклом с t-упорядоченным охватыванием в графе $\tilde{G}(t)$. Покажем, что R – искомый эйлеров OE-цикл.

Действительно, маршрут R есть цикл, содержащий все ребра графа G в точности по одному разу, т.е. R – эйлеров цикл. Для доказательства того факта, что R имеет упорядоченное охватывание, вновь воспользуемся математической индукцией по числу ребер множества $E(C(f_0))$ в начальной части цикла R.

Начальная часть маршрута R, не содержащая ребер множества $E(C(f_0))$, представляет обход с v_0-упорядоченным охватыванием графа $\tilde{G}(v_0)$. Пусть начальная часть цикла R, содержащая k: $0 \leq k \leq |E(C(f_0))|$ первых ребер цикла $C(f_0)$ имеет упорядоченное охватывание. При добавлении $(k+1)$-го ребра $[v_k, v_{k+1}]$ возможны два случая.

1. Если $v_{k+1} \in T$, то при движении по ребру $[v_k, v_{k+1}]$ условие упорядоченного охватывания не нарушается, т.к. не изменяется внутренность пройденной части цикла R. Дальнейшее движение производится вдоль цикла с упорядоченным охватыванием компоненты связности $\tilde{G}(v_{k+1})$. Поскольку $(\forall v:\ vLv_{k+1})\ \Big(v \notin V(\tilde{G}(v_{k+1}))\Big)$, то в рассматриваемом случае условие упорядоченного охватывания на всех частях маршрута, содержащих $k+1$ ребер из $E(C(f_0))$ также не будет нарушено.
2. Если $v_{k+1} \notin T$, то $\exists u \in T:\ uLv_{k+1},\ v_{k+1} \in V(\tilde{G}(u))$. Однако, в соответствии со способом построения $(\forall t \in T:\ v_{k+1}Lt)\, \tilde{G}(t) \cap \mathrm{Int}(C_{[v_k, v_{k+1}]}) = \emptyset$, т.е. условие упорядоченного охватывания не нарушается.

Итак, R – эйлеров OE-цикл. **Теорема доказана.**

Доказательство данной теоремы в сущности дает рекурсивный алгоритм построения эйлерова OE-цикла [47, 87, 150]. Описание алгоритма приведено в следующем подразделе.

3.3 Рекурсивный алгоритм построения эйлеровых OE-циклов

Рекурсивные алгоритмы построения таких циклов представлены в работах [78, 135]. Для реализации рекурсивного алгоритма будем использовать представление данных, описанное в подразделе 3.1.

В описании данного алгоритма и в алгоритмах, представленных ниже, используется понятие **ранга** ребра e [79].

Определение 10. *Рангом ребра $e \in E(G)$ будем называть значение функции $rank(e) : E(G) \to \mathbb{N}$, определяемое рекурсивно:*

- *пусть $E_1 = \{e \in E : \ e \subset f_0\}$ – множество ребер, ограничивающих внешнюю грань f_0 графа $G(V, E)$, тогда $(\forall e \in E_1)\,(\mathrm{rank}(e) = 1)$;*
- *пусть $E_k(G)$ – множество ребер ранга 1 графа*
$$G_k\left(V, E\backslash\left(\bigcup_{l=1}^{k-1} E_l\right)\right),$$
тогда $(\forall e \in E_k)\,(\mathrm{rank}(e) = k)$.

Псевдокод рекурсивного алгоритма определения как ранга, так и соответствующего OE-цикла представлен ниже (алг.2). Алгоритм использует структуры `FirstLast` и `Edge`. Структура `FirstLast` состоит из двух целочисленных полей `First` и `Last`, предназначенных для возврата функциями номеров первого и последнего ребер соответственно в построенных циклах. Исходный граф G задается в виде массива структур `Edge`. Отдельный элемент массива соответствует ребру графа. Поля структуры предназначены для хранения значений одноименных функций, определенных на соответствующем ребре.

Работу алгоритма `RECURSIVE_OE` можно разбить на две части.

Первая часть функции, соответствующая первому циклу `do...while`, предназначена для нахождения цикла из ребер, смежных внешней грани графа $\tilde{G}$, где $t = v_1(e_0)$. Данный цикл представляется заданием поля `Mark` для каждого его ребра.

Algorithm 2 RECURSIVE_OE (G, e_0) (Часть 1)

Require: граф $G = (V, E)$, первое рассматриваемое ребро $e_0 \in \partial f_0$;
Ensure: Очередь Mark, первое ребро в очереди Ret.First, последнее ребро в очереди Ret.Last;

1: **procedure** RECURSIVE_OE(In: $G = (V, E)$, $e_0 \in \partial f_0$ Out: $Mark$, Ret)
2: **for all** $e \in E$ **do** ▷ Инициализация. Все ребра не помечены
3: $Mark[e] = \infty$;
4: **end for**
5: $Start = e_0$; $Next = l_1[e_0]$
6: **while** $Next \neq Start$ **do** ▷ Обход ребер, смежных внешней грани
7: $Vertex = v_1[Next]$; $Next = l_1[e_0]$; $e_0 = Next$; ▷ Переход к следующему ребру
8: **if** $(Mark[Next] = \infty)$ **then** ▷ Если ребро не помечено
9: **if** $(Next = Start)$ **then** ▷ Если цепь пройдена
10: $Mark[e_0] = Next$; ▷ Пометить текущее ребро
11: break; ▷ Цикл найден. Завершение просмотра ребер
12: **end if**
13: **else** ▷ Для помеченного ребра
14: $e = l_2[Mark[Next]]$; ▷ Перейти к следующему ребру
15: **if** $(e \neq Start)$ **then** ▷ Если не достигнуто начало цепи
16: **while** $(Mark[e] \neq \infty)$ **do** ▷ Пока текущее ребро помечено
17: $e = l_2[l_1[e]]$; ▷ Выбирать следующее ребро
18: **if** $e = Start$ **then**
19: break; ▷ При достижении начала цепи, завершить выполнение цикла
20: **end if**
21: **end while**
22: $Next = e$; ▷ Переместить указатель на следующее ребро
23: **end if**
24: **end if**
25: **if** $(Vertex = v_2[Next])$ **then** ▷ Если порядок следования вершин в цепи нарушен,
26: REPLACE($Next$); ▷ переопределить функции текущего ребра
27: **end if**
28: $Mark[e_0] = Next$; ▷ Пометить текущее ребро
29: **end while**

Вторая часть, соответствующая следующему циклу `do...while`, рекурсивно вызывает алгоритм `RECURSIVE_OE` для каждого ранее непомеченного ребра, инцидентного вершинам цикла, построенного при прохождении перво-

Algorithm 3 RECURSIVE_OE (G, e_0) (Часть 2)

```
30:     Mst = 0;                                          ▷ Флаг наличия вложенного цикла
31:     while true do
32:                                  ▷ Если одной из вершин цикла инцидентно непомеченное ребро
33:         if l_2[Next] ≠ e_0 and Mark[l_2[Next]] = ∞ then
34:             if Mst = 0 then                                  ▷ Имеется вложенный цикл?
35:                 Mst = l_2[Next];                ▷ Записать в Mst его первое ребро
36:             end if
37:             if Vertex ≠ v_2[l_2[Next]] then                  ▷ Порядок на ребре нарушен,
38:                 REPLACE(l_2[Next]));                ▷ переопределить функции ребра
39:             end if
40:                                  ▷ Рекурсивный вызов алгоритма для вложенной компоненты
41:             Ret=RECURSIVE_OE(G, l_2[Next]);
42:             if Mark[e_0] ≠ ∞ then                ▷ Первое ребро компоненты помечено?
43:                 tmp = Mark[e_0];                          ▷ Запомнить пометку в tmp
44:             end if
45:                                          ▷ Если пометки вершин v_1 и v_2 совпадают, то
46:             if v_2[Mark[Ret.First]] = v_1[Mark[First]] then
47:                                  ▷ первое ребро цикла = первому ребру вложенного цикла
48:                 Mark[First] = Ret.First;
49:             else
50:                                      ▷ Первое ребро цикла = левому соседу текущего ребра
51:                 Mark[First] = l_2[Next];
52:             end if
53:             Mark[Ret.Last] = tmp;                ▷ Сохранить метку последнего ребра
54:         end if
55:         e_0 = Next; Next = Mark[e_0];                ▷ Перейти к следующей вершине
56:          Vertex = v_1[e];
57:                                  ▷ Если просмотрены все ребра, завершить выполнение цикла
58:         if Next ≠ Ret.First  and  Next ≠ Start then
59:             Break;
60:         end if
61:     end while
62:     if Mst = 0 then                          ▷ Если нет вложенных компонент связности,
63:         Ret.First = Start;                           ▷ Первое ребро найденного цикла
64:     else                                             ▷ Первое ребро найденного цикла
65:         Ret.First = Mst;
66:     end if
67:     Ret.Last = First;                            ▷ Определить последнее ребро в цепи
         return Ret;
68: end procedure
```

го цикла `do...while`. После построения обхода соответствующей компоненты связности, он включается в результирующий обход.

Таким образом, описанный алгоритм `RECURSIVE_OE` позволяет найти эйлеров OE-цикл в плоском эйлеровом графе и определяет значения рангов ребер.

3.4 Результативность рекурсивного алгоритма

Результативность работы алгоритма `RECURSIVE_OE` следует из доказательства теоремы существования эйлерова OE-цикла. Произведем оценку сложности алгоритма.

Как отмечено выше, алгоритм состоит из двух частей. При нахождении в первой части алгоритма очередной пометки `Next`, требуется просмотреть в худшем случае $\deg(v_1(e))$ инцидентных e ребер. Данный цикл с учетом рекурсии выполняется ровно $|E(G)|$ раз, следовательно, сложность выполнения цикла не превосходит величины, пропорциональной

$$|E(G)| \cdot \sum_{\forall v} \deg(v) = |E|^2,$$

то есть эта часть алгоритма имеет сложность $O(|E|^2)$.

Во второй части алгоритма осуществляется рекурсивный вызов, при этом происходит последовательный просмотр всех вершин, поэтому сложность этой части функции не превосходит $O(|V(G)|)$. Следовательно, сложность всего алгоритма составляет величину $O(|E(G)|^2)$. Таким образом, предложенный алгоритм решает задачу за полиномиальное время $O(|E(G)|^2)$.

3.5 Нерекурсивный алгоритм построения эйлерова OE-цикла

В предыдущем разделе был рассмотрен рекурсивный алгоритм построения OE-цикла в плоском эйлеровом графе. В работе [137] предложен еще один эффективный алгоритм построения OE-циклов в плоских эйлеровых графах `OECover` (алг. 4), имеющий вычислительную сложность $O(|E| \cdot \log |V|)$.

Algorithm 4 OE-Cycle

Require: $G = (V, E)$ – плоский граф;
Ensure: $first \in E$, $last \in E$, $\mathrm{mark}_1 : E \to E$;
1: **procedure** OECycle(In: $G = (V, E)$; Out:$first \in E$, $last \in E$, $\mathrm{mark}_1 : E \to E$)
2: Initiate();
3: Order();
4: $FormChain(v_0)$;
5: **end procedure**

Граф G представлен списком ребер с заданными на них функциями $v_k(e)$, $l_k(e)$, $f_k(e)$, $k = 1, 2$ (см. раздел 3.1).

При описании и анализе алгоритма будем использовать обозначения $\overline{v}_k()$, $\overline{l}_k()$, $\overline{f}_k()$, $k = 1, 2$ для функций, построенных алгоритмом, в отличие от первоначально заданных функций $v_k()$, $l_k()$, $f_k()$, $k = 1, 2$.

В теле алгоритма кроме указанных выше функций $v_k()$, $l_k()$, $f_k()$, $k = 1, 2$ формируются дополнительные функции:

1. $Stack : V \to E$: $Stack(v)$ – указатель на очередь v-списка `M2`-помеченных ребер;
2. $\mathrm{mark}_k() : E \to E$ и $\mathrm{prev}_k() : E \to E$, $k = 1, 2$ для организации двусвязных списков с целью обеспечения операций вставки/удаления за время $O(1)$.

Кроме того, функция $\mathrm{mark}_1()$ используется для представления результата выполнения алгоритма.

Алгоритм `OECycle` (алг. 4) состоит из последовательного выполнения процедур `Initiate`, `Order`, и `FormChain()`, соответствующих трем этапам: «Инициализация», «Упорядочение» и «Формрование». В алгоритме также используется описанная ранее процедура `REPLACE()` (см. раздел 3.3, алг.1).

Как уже отмечалось ранее, данная прцедура переопределяет введенные функции $v_k(e)$, $l_k(e)$, $f_k(e)$, $k = 1, 2$ таким образом, чтобы движение по ребру $e \in E$ в построенном алгоритмом цикле происходило бы от вершины $v_2(e)$ к вершине $v_1(e)$.

Этап «Инициализация». В теле процедуры `Initiate` (алг. 5) присваиваются начальные значения

$$S(v) = \emptyset, \quad v \in V, \quad \mathrm{mark}_1(e) = \infty, \quad e \in E,$$

а также определяется ребро $e_0 \in E$, принадлежащее границе внешней грани f_0. Функции на ребре e_0 переопределяются таким образом, чтобы $\overline{f_1}(e_0) = f_0$,

Algorithm 5 Процедура Initiate

```
1:  procedure INITIATE
2:      for all v ∈ V do
3:          Stack(v) = ∅;
4:      end for
5:      for all e ∈ E do
6:          mark_1(e) = mark_2(e) = ∞;
7:          prev_1(e) = prev_2(e) = 0;
8:          if (f_1(e) = f_0) or (f_2(e) = f_0) then
9:              e_0 = e;
10:         end if
11:     end for
12:     if f_2(e_0) = f_0 then
13:         REPLACE(e_0);
14:     end if
15:     first = last = e_0; v_0 = v = v_1(e_0);
16:     ne = l_1(e_0);
17: k = 1; rank(e_0) = 1;
18: end procedure
```

т.е. чтобы ребро $\overline{l_1}(e_0)$ принадлежало границе внешней грани. Также в теле данной функции инициализируется очередь `M1`-помеченных ребер (рисунок 3.2), как состоящая из единственного ребра e_0, переменные `first` и `last` используются для указания соответственно первого и последнего элементов

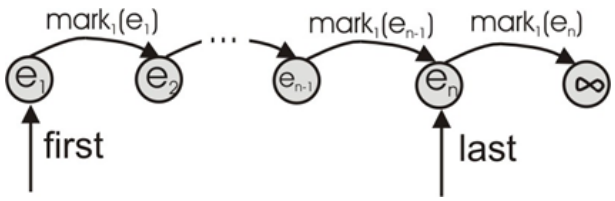


Рисунок 3.2: Организация очереди M1-помеченных ребер

очереди. Переменная ne используется для определения следующего кандидата для включения в список M1-помеченных ребер. Переменная v_0 используется для запоминания вершины, принадлежащей границе внешней грани, а переменная v – как текущая вершина для задания ориентации ребра, определяемого ne.

Этап «Упорядочение». Процедура Order (Упорядочение) представлена в алг.6. Функциональное назначение процедуры Order состоит в:

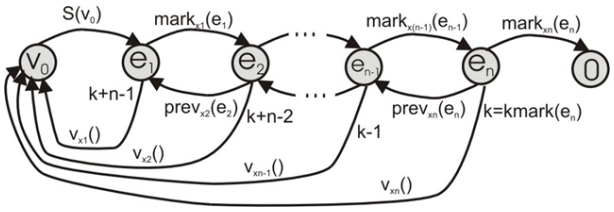
- определении на каждом ребре $e \in E$ значения $\mathrm{rank}(e)$;
- формировании для каждой вершины списка инцидентных ребер (рисунок 3.3), упорядоченных в порядке убывания значения rank().

Процедура *Order* использует переменную k как счетчик стадий и функцию $\mathrm{rank}() : E \to N$, указывающую номер стадии, на которой ребро ставится в очередь M1-помеченных ребер. Содержательный смысл функции rank заключается в том, что она определяет ранг ребра e (определение ранга ребра приведено в разделе 3.3). Заметим, что ранг любого ребра плоского графа может быть определен за время $O(|E|)$ с помощью процедуры Order.

Данная процедура выполняется следующим образом. На первой стадии в очередь M1-помеченных ребер вводятся все ребра $e \in E$, ограничивающие внешнюю грань f_0, а их ориентация задается так, чтобы $\overline{f_1}(e) = f_0$. На стадии $k+1$ каждое ребро $e \in E$, попавшее в очередь M1-помеченных на стадии k, переводится в состояние M2-помеченного и помещается в списки вершин $\overline{v_l}(e)$, $l = 1, 2$ (см. рисунок 3.3), а в очередь M1-помеченных включаются все непомеченные ребра, ограничивающие грань, общую с ребром e.

Algorithm 6 Процедура Order

```
1: procedure ORDER
2:     while first ≠ ∞ do
3:         while (mark(ne) = ∞) and (last ≠ ne) do
4:             M1:
5:             rank(ne) = k;
6:             mark_1(last) = ne;
7:             if v_2(ne) ≠ v then
8:                 REPLACE(ne);
9:             end if
10:            v = v_1(ne); last = ne; ne = l_1(ne);
11:        end while
12:        e = first; first = mark_1(first); v = v_2(e);ne = l_2(e);
13:        M2:
14:        k = rank(e) + 1; mark_1(e) = Stack(v_1(e)); mark_2(e) = Stack(v);
15:        if mark_1(e) ≠ 0 then
16:            if v_1(e) = v_1(mark_1(e)) then
17:                prev_1(mark_1(e)) = e;
18:            else
19:                prev_2(mark_1(e)) = e;
20:            end if
21:        end if
22:        if mark_2(e) ≠ 0 then          ▷ Помещение ребра в стеки вершин v_1(e) и v_2(e)
23:            if v = v_1(mark_2(e)) then
24:                prev_1(mark_2(e)) = e;
25:            else
26:                prev_2(mark_2(e)) = e;
27:            end if
28:            Stack(v) = e; Stack(v_1(e)) = e;
29:        end if
30:    end while
31: end procedure
```



$$x_k = \begin{cases} 1,\ v_1(e_k) = v_0 \\ 2,\ v_2(e_k) = v_0 \end{cases}$$

Рисунок 3.3: Организация v_0-списка M2-помеченных ребер

Для анализа результативности алгоритма положим

$$E_k = \{e \in E : \mathrm{rank}(e) = k\}, \quad E_k^* = \{e \in E : \mathrm{rank}(e) \le k\},$$
$$\overline{E_k} = E \backslash E_k^*,$$

через $G(E')$ будем обозначать плоский граф, порожденный множеством ребер $E' \subset E$.

Лемма 1. *[86] Для любого $k = 1, 2, 3, \ldots, M$, где $M = \max\limits_{e \in E} \mathrm{rank}(e)$, имеет место следующее утверждение:*

$$\mathrm{Int}(G(E_k)) \supset G(\overline{E_k}); \quad S \backslash \mathrm{Int}(G(E_k)) \supset G(E_k^*).$$

Доказательство леммы проведем методом математической индукции по k. Из описания алгоритма следует, что k принимает значение, равное 1, при выполнении процедуры `Initiate`, и значения $\mathrm{rank}() = 1$ устанавливаются только на ребрах, вводимых в очередь M1-помеченных при первом выполнении тела внешнего цикла в процедуре `Order`. В соответствии с описанием процедуры `Initiate` ребра e_0 и $e_1 = \overline{l_1}(e_0)$ ограничивают внешнюю грань f_0 графа G, значение переменной ne указывает на ребро e_1, а значение переменной $v = v_1(e_0)$ – на вершину v_1, инцидентную ребрам e_0 и e_1. Поэтому при первом выполнении тела цикла `Order` в очередь M1-помеченных будут последовательно внесены все ребра цепи

$$v_1 e_1 v_2 e_2 \ldots e_m v_{m+1},$$

удовлетворяющей условиям

$$\begin{aligned} v_2 &= e_1 = v_1,\ \overline{v_2}(e_{i+1}) = v_{i+1} = \overline{v_1}(e_i),\ i = 1, 2, \ldots, m, \\ e_{i+1} &= l_1(e_i), i = 1, 2, \ldots, m-1, \\ \overline{f_1}(e_i) &= \overline{f_1}(e_1), i = 1, 2, \ldots, m-1, \\ \overline{l_1}(e_m) &= e_0. \end{aligned}$$

Таким образом, значение $\mathrm{rank}() = 1$ будет определено на всех ребрах, ограничивающих внешнюю грань f_0 графа G. В этом случае справедливость утверждений леммы очевидна.

Предположим, что доказываемые утверждения имеют место для $k < K \leq M$. Рассмотрим множество

$$F_{K-1} = \{e \in E_{K-1} |\ l_2(e) \notin E^*_{K-1}\}.$$

Из связности графа G, а также из того, что $F_{K-1} \subset \mathsf{Int}(G(E_K))$ следует $F_{k-1} \neq \emptyset$, поэтому, в соответствии с описанием алгоритма, значение функции $\mathrm{rank}() = K$ будут определены на ребрах максимальных по включению цепей

$$C(e) = v_1 e_1 v_2 e_2 \ldots e_m v_{m+1}, \quad e \in F_{K-1},$$

удовлетворяющих условиям

$$\begin{aligned} v_1 &= \overline{v_2}(e), e_1 = \overline{l_2}(e); \\ \overline{v_2}(e_{i+1}) &= v_{i+1} = \overline{v_1}(e_i), \quad i = 1, 2, \ldots, m; \\ e_{i+1} &= \overline{l_1}(e_i), \quad \mathrm{mark}(e_{i+1}) = \infty, \quad i = 1, 2, \ldots, m-1, \\ \overline{f_1}(e_i) &= \overline{f_2}(e), i = 1, 2, \ldots, m. \end{aligned}$$

Таким образом,

$$E_K = \bigcup_{e \in F_{K-1}} E\left(C(e)\right)$$

и все цепи $C(e)$, $e \in F$ являются реберно-непересекающимися. Из связности графа G и отсутствия в нем висячих вершин следует, что в цепи $C(e)$, $e \in F_{K-1}$ ее последняя вершина v_{m+1} принадлежит $V(E_{K-1})$, где $V(E_{K-1})$ – множество вершин, инцидентных ребрам из E_{K-1}.

Если в цепи $C(e)$, $e \in F_{K-1}$ имеет место равенство $v_1 = v_{m+1}$, то $C(e)$ – цикл. Если же $v_1 \neq v_{m+1}$, то, поскольку $G^*(E_{K-1})$ – объединение непересекающихся по ребрам цепей, в цепи, содержащей ребро $\overline{l_1}(e_m)$, существует единственное ребро $e' \in F_{K-1}$: $\overline{v_2}(e') = v_{m+1}$.

Кроме того, по построению

$$\mathsf{Int}(G(E_K)) = \mathsf{Int}(G(E_{K-1}))\backslash \left(\bigcup_{e \in E_K} \overline{f_1}(e) \right), \tag{3.1}$$

$$S\backslash\mathsf{Int}(G(E_K)) = \left(\bigcup_{e \in E_K} \overline{f_1}(e) \right) \bigcup (S\backslash\mathsf{Int}(G(E_{K-1}))) . \tag{3.2}$$

Поскольку

$$\overline{E_K} \subset \overline{E_{K-1}} \subseteq \mathsf{Int}(G(E_{K-1})), \overline{E_K} \not\subset \bigcup_{e \in E_K} \overline{f_1}(e),$$

то из (3.1) следует $S\backslash\mathsf{Int}(G(E_K)) \supseteq \overline{E_K}$.

Поскольку

$$E_K^* \subseteq E_{K-1}^* \bigcup E_K, E_K \subseteq \bigcup_{e \in E_K} \overline{f_1}(e), E_{K-1}^* \subseteq S\backslash \operatorname{Int}(G(E_K)),$$

то из (3.2) следует $S\backslash\mathsf{Int}(G(E_K)) \supseteq E_K^*$. **Лемма 1 доказана.**

Лемма 2. *[86] Если* $M = \max\limits_{e \in E} \operatorname{rank}(e)$, *то* $\overline{E_M} = \emptyset$.

Доказательство. Предположим противное, то есть $\overline{E_M} \neq \emptyset$. Из леммы 1 следует $\overline{E_M} \subseteq \mathsf{Int}G(E_M)$.

Рассмотрим множество

$$F_M = \left\{ e \in E_M \,\middle|\, \overline{l_2}(e) \in \overline{E_M} \right\}.$$

Из связности графа G следует $F_M \neq \emptyset$. В соответствии с описанием алгоритма

$$(\forall e \in F_M)\,(\operatorname{rank}(e) = M + 1)\,,$$

т.е. имеем противоречие с условием леммы. **Лемма 2 доказана.**

Из леммы 2 следует, что алгоритм определит на каждом ребре $e \in E$ значение функции rank(), т.е. каждое ребро $e \in E$ будет включено в очередь `M1`-помеченных ребер. Так как после включения ребра $e \in E$ в эту очередь

$\mathrm{mark}_1(e) \neq \infty$, то такое включение возможно единственный раз. Каждое ребро, попавшее в очередь `M1`-помеченных ребер, переводится в состояние `M2`-помеченного включением его в стеки вершин $\overline{v_k}(e)$, $k = 1, 2$ в порядке, определяемом очередью, т.е. в порядке возрастания величины $\mathrm{rank}(e)$. Поэтому после завершения процедуры `Order` для каждой вершины $v \in V$ будем иметь

$$Stack(v) = \arg \max_{e \in E(v)} \mathrm{rank}(e);$$

$$\left(e' \in E(v),\ \mathrm{rank}(e') > \min_{e \in E(v)} \mathrm{rank}(e)\right) \Rightarrow$$

$$\Rightarrow \left(\exists e'' = \mathrm{mark}_x(e') \in E(v),\ \mathrm{rank}(e'') = \mathrm{rank}(e') - 1\right),$$

где

$$E(v) = \left\{e \in E :\ (v = \overline{v_2}(e)) \vee (v = \overline{v_1}(e))\right\},$$

$$x = \begin{cases} 1, \text{ если } v_1(e') = v, \\ 2, \text{ если } v_2(e') = v. \end{cases}$$

Таким образом доказана результативность процедуры `Order`.

Кроме того, из лемм 1 и 2 следует

$$E^*_M = E,\ M = \max_{e \in E(v)} \mathrm{rank}(e),$$

поэтому орграф $G^* = G^*(A^*_M)$, как ориентированный образ графа G, является связным. Так как G^* есть объединение непересекающихся по дугам орциклов, то G^* – эйлеров орграф.

Этап «Формирование». Функция `FormChain(`v`)` (см. алг.7) позволяет построить максимальную по включению цепь

$$C = v_1 e_1 v_2 e_2 v_3 \ldots e_L v_{L+1},$$

$$e_i = \arg \max_{e \in E(v_i) \setminus \{e_l | l < i\}} \mathrm{rank}(e), v_{i+1} = \overline{v_1}(e_i),\ i = 1, 2, \ldots, L,$$

которая удовлетворяет также следующим условиям:

- $v_1 = v_0$ – вершина, ограничивающая внешнюю грань, найденная на этапе инициализации (функция `Initiate`);

- для любой начальной части

$$C_l = v_1 e_1 v_2 e_2 \dots e_l,\ l \le L$$

и для любой вершины $v \in V$ имеет место неравенство

$$\min_{e \in E(v) \bigcap E(C_l)} \mathrm{rank}(e) > \max_{e \in E(v) \setminus E(C_l)} \mathrm{rank}(e).$$

Из эйлеровости орграфа G^* следует, что C является циклом. Очевидно, что цикл содержит все ребра, инцидентные вершине v_1, следовательно, и всем вершинам, принадлежащим границе внешней грани.

Algorithm 7 Процедура FormChain

```
1:  procedure FormChain(In: v_0 ∈ f_0)
2:      v = v_0; e = Stack(v); First = Last = e;        ▷ Перейти в вершину стека v
3:      while true do
4:          if v_1(e) = v then
5:              REPLACE(e);                  ▷ Установить правильный порядок индексов
6:          end if
7:          Stack(v) = mark_2(e)             ▷ Извлечение ребра из стека вершины v_2(e)
8:          if v = v_1(mark_2(e)) then
9:              prev_1(mark_2(e)) = 0;
10:         else
11:             prev_2(mark_2(e)) = 0;
12:         end if
13:         v = v_1(e);
14:         if prev_1(e) ≠ 0 then
15:             if e = mark_1(prev_1(e)) then
16:                 mark_1(prev_1(e)) = mark_1(e);
17:             else
18:                 mark_2(prev_1(e)) = mark_1(e);
19:             end if
20:         else
21:             Stack(v) = mark_1(e);
22:             if v = v_1(mark_1(e)) then
23:                 prev_1(mark_1(e)) = 0;
24:             else
25:                 prev_2(mark_1(e)) = 0;
26:             end if
27:         end if
28:         e = Stack(v);                    ▷ Извлечение ребра из стека вершины v_1(e)
29:         M3: mark_1(Last) = e; Last = e;
30:         if Last = 0 then
31:             break;
32:         end if
33:     end while
34: end procedure
```

Лемма 3. *[86] Для любых* $l = 1, 2, \ldots, L$ *и* $k = 1, 2, \ldots, M$ *имеет место равенство* $\mathrm{Int}(C_l) \bigcap G(E_k) = \emptyset$.

Для **доказательства** воспользуемся методом математической индукции по переменной k.

Ребра множества E_1 образуют цикл, ограничивающий внешнюю грань f_0 графа G, поэтому они будут включены в построенный алгоритмом цикл. Кроме того

$$(\forall H \subseteq G) \left(E_1 \bigcap \mathrm{Int}(H) = \emptyset \right).$$

Покажем, что из

$$\mathrm{Int}(C_l) \bigcap G(E_K) = \emptyset, l = 1, 2, \ldots, L, k = 1, 2, \ldots, K-1, K \leq M$$

следует

$$\mathrm{Int}(C_l) \bigcap G(E_K) = \emptyset, l = 1, 2, \ldots, L. \tag{3.3}$$

Предположим противное, пусть нашлось такое l', что $v_2(e') = v \in VC_{l'}$.

В соответствии с леммой 1

$$\mathrm{Int}G(E_K) \supseteq \overline{G(E_K)} = \{e : \ \mathrm{rank}(e) > k\}, k = 1, 2, \ldots, M$$

Из связности графа G и доказанных в лемме 1 свойств графов $G(E_K)$ и орграфов $G^*(E_K)$ следует, что множество $\mathrm{Int}(C_{l'}) \bigcap E_K$ содержит такое ребро $e' \in E$, что $v_2(e') = v \in VC_{l'}$.

Поэтому

$$\min_{e \in E(v) \bigcap EC_{l'}} \mathrm{rank}(e) < \mathrm{rank}(e') \leq \max_{e \in E(v) \setminus EC_{l'}} \mathrm{rank}(e),$$

т.е. имеем противоречие с (3.2), что доказывает справедливость равенства (3.3). Применяя принцип математической индукции, получаем утверждение леммы: $\mathrm{Int}(C_l) \bigcap G(E_K) = \emptyset, \ l = 1, 2, \ldots, L, \ k = 1, 2, \ldots, M$.

Лемма 3 доказана.

Поскольку $E = \bigcup_{k=1}^{M} E_k$, то, в соответствии с леммой 3

$$\mathrm{Int}(C_l) \bigcap G(E_K) = \emptyset, \ l = 1, 2, \ldots, L.$$

Учитывая выше изложенное, заключаем, что $L = |E|$, а цикл, построенный процедурой `FormChain`, является эйлеровым OE-циклом.

Очевидно, что вычислительная сложность этапов «Инициализация» (процедура `Initiate`) и «Формирование» (процедура `FormChain`) составляет величину $O(|E|)$. Вычислительная сложность этапа «Упорядочение» (процедура `Order`) также составляет величину $O(|E|)$, так как каждое ребро единственный раз ставится в очередь $M1$-помеченных ребер и затем переводится из нее в стеки вершин, а вычислительная сложность этих операций составляет величину $O(1)$. Таким образом, вычислительная сложность алгоритма – $O(|E|)$.

Изложенное в данном разделе обобщим в виде следующей теоремы.

Теорема 9. *Если $G(V, E)$ – плоский эйлеров граф с множеством граней F, заданными на E функциями $v_k : V \to E$, $l_k : E \to E$, $f_k : F \to E$, $k = 1, 2$, то алгоритм `OE-Cycle` находит в G эйлеров OE-цикл. При завершении алгоритма переменные $First$ и $Last$ определяют первое и последнее ребра найденного цикла, значение функции $mark(e)$ – ребро, следующее за ребром $e \in E$ в найденном цикле. Вычислительная сложность алгоритма не превосходит $O(|E|)$.*

3.6 Эффективные алгоритмы построения OE-маршрута в произвольном связном плоском графе

В случае произвольного плоского графа (который в общем случае не является эйлеровым) задачу построения OE-маршрута можно рассматривать в двух формулировках:

- как задачу построения OE-маршрута китайского почтальона;
- как задачу построения OE-покрытия графа цепями.

3.6.1 Алгоритм построения OE-маршрута китайского почтальона

Пусть дан произвольный плоский граф $G = (V, E)$. Рассмотрим задачу построения на множестве его ребер $E(G)$ маршрута C, удовлетворяющего условию упорядоченного охватывания. В общем случае граф не является эйлеровым, поэтому невозможно построить в нем замкнутый цикл. Для того чтобы получить замкнутый маршрут, необходимо некоторые ребра проходить дважды. В дальнейшем будем считать, что ребра, которые необходимо пройти дважды, дублируются ***дополнительными*** ребрами, представленными во множестве $H(G)$.

В данном разделе показано, что алгоритм построения замкнутого маршрута P в неэйлеровом графе G является алгоритмом построения эйлерова цикла C с упорядоченным охватыванием в эйлеровом графе $\tilde{G}$, представляющего модификацию графа G, в которую добавлены ребра множества $H(G)$.

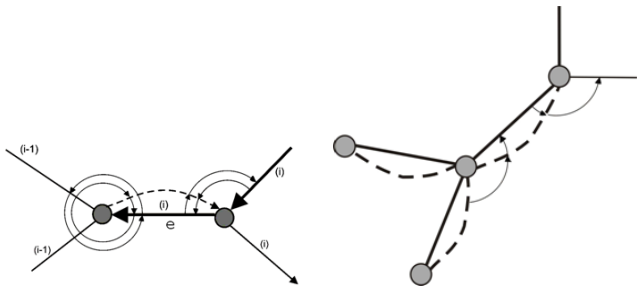
Определение 11. *Будем говорить, что* ***маршрут*** $C = v_1e_1v_2e_2\dots v_k$ *в графе* G *имеет* ***упорядоченное охватывание*** *(является* OE*-маршрутом), если для любой его начальной части* $C_i = v_1e_1v_2e_2\dots e_i$, $i \le (|E(G)| + |H(G)|)$ *выполнено условие*

$$\mathrm{Int}(C_i) \cap G(E) = \emptyset,$$

т.е. пересечение внутренних граней C_i *с множеством ребер пусто.*

В терминах задачи раскроя (для модели раскройного плана в виде плоского неэйлерова графа) ребра множества $H(G)$ интерпретируются как холостые проходы режущего инструмента.

Найти множество ребер, для которых требуются дубликаты, можно с помощью подхода, аналогичного тому, что используется для задачи китайского почтальона. Очевидно, что в этом случае вычислительная сложность алгоритма может возрасти.



а) б)

Рисунок 3.4: Модификация указателей на ребро при добавлении дополнительных построений: а) для висячей вершины; б) для моста

Здесь будем вводить дубликаты ребер таким образом, чтобы алгоритм `RECURSIVE_OE` (алг. 2) для эйлеровых графов претерпел минимум модификаций. Данную модификацию можно получить следующим образом.

В первой части алгоритма, когда требуется найти цикл из ребер подграфа, ограничивающих внешнюю грань текущего подграфа, возникает ситуация, когда поле $Mark$ помещенного в очередь ребра указывает либо на само себя (висячая вершина текущей компоненты связности), либо это ребро не совпадает с начальным для данной компоненты связности, а указывает на уже помеченные ребра (мост). В обоих случаях необходимо продублировать данные ребра. Введение дополнительных ребер повлечет за собой такое изменение указателей, как показано на рисунках 3.4.а) – для висячей вершины и 3.4.б) – для моста.

После выполнения соответствующих построений для всех ребер поле $Mark$ на первой стадии выполнения алгоритма оказывается сформированным таким образом, что остальные функции алгоритма, разработанного для эйлерова графа, не будут требовать модификации. Таким образом, после введения некоторого числа дополнительных ребер, будет получен эйлеров граф, а найденный в нем цикл будет иметь упорядоченное охватывание. В исходном же графе будем иметь маршрут, в котором ни один цикл из уже пройденных ребер не будет охватывать еще не пройденных. Описанную модификацию алгоритма 2 будем называть алгоритмом `CPP_OE` (см. алг.12) (этот алгоритм

решает OE-задачу китайского почтальона). Рекурсивный вызов функции для каждого непомеченного ребра, инцидентного вершинам цикла, полученного на предыдущем этапе, производится без изменений. После построения обхода для соответствующей компоненты связности, он включается в результирующий обход.

Обобщим все сказанное выше в виде следующей теоремы.

Теорема 10. *Маршрут, построенный с помощью алгоритма* `CPP_OE`*, является OE-маршрутом. Сложность алгоритма составляет величину $O(|E(G)| \cdot |V(G)|)$.*

Доказательство этой теоремы очевидно, т.к. на самом деле ребра, полученные в результате дополнительных построений, являются недостающими фрагментами выделяемых алгоритмом циклов. Потребуется не более чем $O(|E(G)|)$ добавлений ребер, а добавление одного ребра требует изменения шести указателей.

Для более наглядного представления последовательности выполняемых действий организуем первую часть алгоритма в качестве отдельной функции `ExternCycle` (см. алг. 8) [138]. **Входными параметрами** для данной функции являются:

- стартовое ребро графа (ребро, с которого начинается поиск ребер, ограничивающих внешнюю грань текущей компоненты связности);
- ребро, следующее за стартовым;
- начальная вершина для текущей компоненты связности (для организации правильной ориентации ребер);
- некоторые вспомогательные переменные.

Функция обращается к добавлению ребер во всех описанных выше случаях. Дублирование моста происходит непосредственно в функции `ExternCycle`, в оставшихся двух случаях вызываются различные модификации функции `Add` (см. алг. 10 и 11): для ликвидации висячей вершины текущей компо-

ненты связности и для завершения цикла в данной компоненте связности соответственно.

Algorithm 8 ExternCycle (Часть 1)

Require: $G = (V, E)$ – плоский граф; $First$ – первое рассматриваемое ребро; $Next$ – указатель на следующее ребро; $Vertex$ – текущая вершина; $Number$ – число ребер в графе;

Ensure: $NewFirst$ – номер дополнительного ребра, завершающего цикл;

```
1: procedure ExternCycle(In: G = (V,E), First, Next, Vertex, Number; Out: NewFirst)
2:     NewFirst = 0;
3:     while true do
4:         First = Next; Vertex = v1(First); Next = l1(First);
5:         if ( thenMark(Next) ≠ ∞)
6:             if ( thenNext = Start)
7:                 if ( thenv1(First) = v1(Next))
8:                     Add(G, Next, Mark(Next)); Number = Number + 1;
9:                     Mark(First) = Number; Mark(Number) = Next;
10:                    Level(First) = L; Level(Number) = L; NewFirst = Number;
11:                    return NewFirst;
12:                end if
13:                Mark(First) = Next; Level(First) = L; return NewFirst;
14:            else
15:                e = l2(Mark(Next));
16:                if e ≠ Start then
17:                    while Mark(e) ≠ ∞ do
18:                        e = l2(l1(e));
19:                        if e = Start then
20:                            break;
21:                        end if
22:                    end while
23:                end if
```

Пример работы алгоритма для плоского неэйлерова графа приведен на рисунке 3.5. Подчеркнутые фрагменты маршрута соответствуют движению по основным ребрам графа, неподчеркнутые – по дополнительным, которые на рисунке обозначены пунктиром. Предложенный алгоритм находит маршрут

$$\underline{v_1 v_3 v_7 v_{10} v_8} v_{10} \underline{v_9} v_{10} v_7 \underline{v_8 v_6 v_4} v_6 \underline{v_5} v_6 v_8 \underline{v_9 v_{11} v_{12}} v_{11} \underline{v_{13}} v_{11} v_9$$
$$\underline{v_7} v_3 \underline{v_2 v_1 v_4 v_5 v_{12} v_{13} v_2} v_3 v_1. \tag{3.4}$$

Предложенный алгоритм позволяет найти $|\tilde{V}|$ различных маршрутов для данного графа G. Здесь $\tilde{V}$ – множество вершин графа G, принадлежащих

Algorithm 9 ExternCycle (Часть 2)

```
24:                 if e ≠ First then
25:                     if Mark(Next) ≠ ∞ and Level(Next) = L then
26:                         Number = Number + 1;
27:                         v1(Number) = v2(Next); v2(Number) = v1(Next);
28:                         l1(Number) = l2(Next); r1(l2(Next)) = Number;
29:                         r1(Number) = Next; l2(Next) = Number;
30:                         if v1(r1(Next)) = v2(Number) then
31:                             l1(r1(Next)) = Number;
32:                         else
33:                             l2(r1(Next)) = Number;
34:                         end if
35:                         r2(Number) = r1(Next); r1(Next) = Number; l2(Number) = Next;
36:                         Next = Number;
37:                     end if
38:                     Next = e;
39:                 else
40:                     Number = Number + 1; Add(G, Number, First);
41:                     Next = l1(First);
42:                 end if
43:             end if
44:         end if
45:         if Vertex ≠ v2(Next) then
46:             REPLACE(Next);
47:         end if
48:         Mark(First) = Next; Level(First) = L;
49:         if Next = Start then
50:             break;
51:         end if
52:     end while
53:     return NewFirst;
54: end procedure
```

Algorithm 10 Add (Функция для добавления дополнительного ребра в случае нахождения в висячей вершине)

```
1: procedure ADD(In: G = (V, E) – плоский граф; Number – номер добавляемого ребра;
   First – ребро, приводящее в висячую вершину;)
2:     v_1(Number) = v_2(First); v_2(Number) = v_1(First);
3:     l_1(Number) = l_2(First); r_1(Number) = First;
4:     if v_1(l_2(First)) = v_2(First) then
5:         r_1(l_2(First)) = Number;
6:     else
7:         r_2(l_2(First)) = Number;
8:     end if
9:     l_2(First) = Number;
10:    r_2(Number) = First;
11:    if v_1(r_1(First)) = v_1(First) then
12:        l_1(r_1(First)) = Number;
13:    else
14:        l_2(r_1(First)) = Number;
15:    end if
16:    r_1(First) = Number; l_2(Number) = First; l_1(First) = Number;
17: end procedure
```

Algorithm 11 Add (Функция добавления дополнительного ребра для завершения цикла)

```
1: procedure ADD(In: G = (V, E) – плоский граф; Number – номер добавляемого ребра;
   Next – ребро, принадлежащее внешнему циклу;)
2:     v_1(Number) = v_2(Next); v_2(Number) = v_1(Next);
3:     l_1(Number) = Next; l_2(Next) = Number; l_2(Number) = Mark(Next);
4:     if v_1(Mark(Next)) = v_2(Number) then
5:         r_1(Mark(Next)) = Number;
6:     else
7:         r_2(Mark(Next)) = Number;
8:     end if
9:     r_2(Number) = Next; r_1(Number) = Next; r_2(Next) = Number;
10: end procedure
```

Algorithm 12 CPP_OE (OE-задача китайского почтальона, часть 1)

Require: $G = (V, E)$ – плоский граф; $Number$ – число вершин графа; $First$ – первое рассматриваемое ребро;

Ensure: Очередь $Mark$, первое ребро в очереди $Ret.First$, последнее ребро в очереди Ret.Last;

```
1: procedure CPP_OE(In: G = (V, E); Number; First; Out: Mark, Ret)
2:     for all e ∈ E do
3:         Mark(e) = ∞;
4:     end for
5:     Start = Next = First;
6:     NewFirst=ExternCycle (G, Start, Next, First, Vertex, Number);
7:     Mst = 0;
8:     while true do
9:         if l2(Next) ≠ First and Mark(l2(Next)) = ∞ then
10:            if Mst = 0 then
11:                Mst = l2(Next);
12:            end if
13:            if v ≠ v2(l2(Next)) then
14:                REPLACE(l2(Next));
15:            end if
16:            Ret=CPP_OE (G, l2(Next), Number);
17:            if Mark(First) ≠ ∞ then
18:                Mark(Ret.Last) = Mark(First);
19:                if v2(Ret.First) = v1(First) then
20:                    Mark(First) = l2(Next);
21:                else
22:                    Mark.First = l2(Next);
23:                end if
24:            end if
25:            First = Next; Next = Mark(First); Vertex = v1(e);
26:            if Next = Ret.First or Next = Start then
27:                break;
28:            end if
29:        end if
30:    end while
```

Algorithm 13 CPP_OE (*OE*-задача китайского почтальона, часть 2)

```
31:     if Mst = 0 then
32:         Ret.First = Start;
33:     else
34:         if v2(Ret.First) ≠ v1(First)and  NewFirst = 0 then
35:             Ret.First = Mst;
36:         end if
37:         if v2(Ret.First) ≠ v1(First)and  NewFirst ≠ 0 then
38:             Ret.First = Next;
39:         end if
40:     end if
41:     if NewFirst = 0 then
42:         Ret.Last = First;
43:     else
44:         Ret.Last = NewFirst;
45:     end if
46:     return Ret;
47: end procedure
```

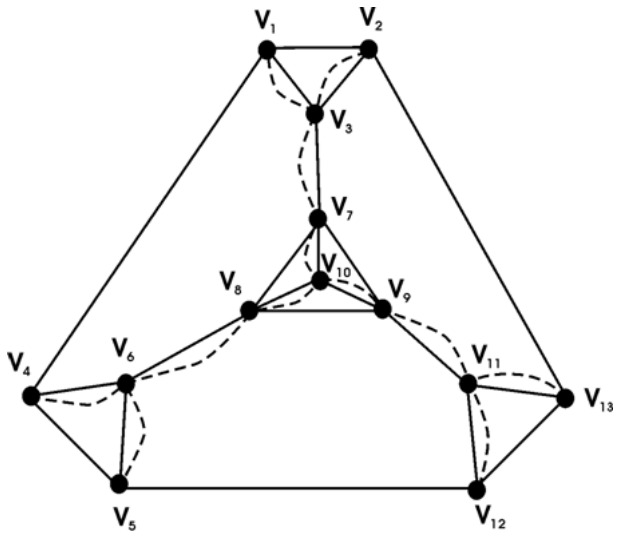


Рисунок 3.5: Пример работы алгоритма для плоского неэйлерова графа

внешней грани этого графа. На самом деле это только нижняя оценка числа решений задачи. Однако предложенный алгоритм может найти только $|\tilde{V}|$ различных решений в силу определенности выбора следующего ребра в обходе [138]. Более подробно об определении количества *OE*-цепей в графе см. главу 4.

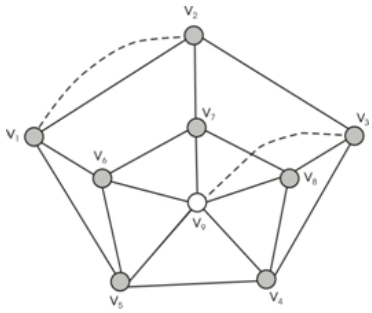


Рисунок 3.6: Пример неэйлерова графа, в котором две вершины нечетной степени смежны разным граням

3.6.2 Задача построения OE-покрытия

Легко заметить, что любой граф можно достроить до эйлерова добавлением $n/2$ ребер, где n – число вершин нечетной степени. При условии сохранения планарности данного представления графа дополнительное построение из n ребер возможно лишь в случае, когда в графе найдется такое разбиение вершин нечетной степени на пары, что они попарно будут смежны одной грани. На рисунке 3.6 этому требованию удовлетворяют вершины v_1, v_2 и v_3. Для вершины v_9 такой пары не существует, следовательно, для общего случая нужно искать другие решения.

Проблеме построения OE-маршрута в плоском графе произвольного вида посвящена работа [71], в которой разработан алгоритм построения таких маршрутов, имеющий вычислительную сложность не более $O\left(|E|^2\right)$.

Покажем, что алгоритм построения замкнутого маршрута P в неэйлеровом графе G является алгоритмом построения эйлерова OE-цикла C в эйлеровом графе $\tilde{G}$, представляющего модификацию графа G, в которую добавлены дополнительные ребра. Множество этих ребер будем обозначать через $H(G)$.

В дальнейшем будем использовать представление графа, приведенное в разделе 3.1.

В работе [45] сформулирована следующая теорема.

Теорема 11. *Пусть $G = (V, E)$ – плоский связный граф на S, не имеющий висячих вершин. Существует множество ребер $F : (F \cap S)\backslash V = \emptyset$ такое, что граф $\hat{G} = (V, E \cup F)$ – эйлеров, и в графе $\hat{G}$ существует эйлеров цикл $C = v_1e_1v_2e_2...e_nv_1$, $n = |E| + |F|$, для любой начальной части которого $C_l = v_1e_1v_2e_2...v_l$, $l \leq |E| + |F|$, выполнено условие $\mathsf{Int}(C_l) \cap G = \emptyset$.*

Доказательство теоремы дает результативность приведенного ниже алгоритма `OECover` (алг. 14) [86], который строит покрытие плоского графа G

Algorithm 14 OECover

Require: $G = (V, E)$ – плоский граф; V_{odd} – множество вершин нечетной степени;
Ensure: $first \in E$, $last \in E$, $\mathrm{mark}_1 : E \to E$;

```
1: procedure OECover(In: G = (V, E), V_odd; Out: first ∈ E, last ∈ E, mark_1 : E → E)
2:     Initiate();
3:     Order();          ▷ Сортировка списка вершин нечетной степени по убыванию ранга
   SortOdd();
4:     while V_odd ≠ ∅ do
5:         v^0 = arg max_{v∈V_odd} rank(v);
6:         V_odd = V_odd\{v^0};
7:         v = FormChain(v^0);
8:         V_odd = V_odd\{v};
9:     end while
10:    FormChain(v_0);
11: end procedure
```

последовательностью OE-цепей. Как и прежде, граф G представлен списком ребер с заданными на них функциями $v_k(e)$, $l_k(e)$, $f_k(e)$, $k = 1, 2$ (см. раздел 3.1).

После выполнения процедур `Initiate` и `Order` (см. раздел 3.5) выполняется упорядочение вершин нечетной степени $v \in V_{odd}$ в порядке возрастания их ранга с помощью процедуры `SortOdd`. За ранг вершины v принимается значение функции $\mathrm{rank}(Stack(v))$. Далее выполняется цикл `while...do` с использованием процедуры `FormChain`,в которой строится последовательность из $|V_{odd}|/2$ простых цепей между парами вершин нечетной степени.

Текст процедуры `FormChain` приведен в алг.15.

Функциональное назначение процедуры состоит в формировании OE-цепи, начинающейся в заданной вершине $w \in V_{odd}$ и заканчивающейся в

Algorithm 15 Процедура FormChain

```
1: procedure FormChain(In: w – вершина нечетной степени, из которой будет построена
   цепь; Out: v – вершина нечетной степени, завершающая построенную цепь)
2:     v = w; e = Stack(v); first = last = e;
3:     do
4:     if v_1(e) = v then
5:         REPLACE(e);
6:     end if
7:     Stack(v) = mark_2(e);
8:     if v = v_1(mark_2(e)) then
9:         prev_1(mark_2(e)) = 0;
10:    else
11:        prev_2(mark_2(e)) = 0;
12:    end if
13:    v = v_1(e);
14:    if prev_1(e) ≠ 0 then
15:        if e = mark_1(prev_1(e)) then
16:            mark_1(prev_1(e)) = mark_1(e);
17:        else
18:            mark_2(prev_1(e)) = mark_1(e);
19:        end if
20:    else
21:        Stack(v) = mark_1(e);
22:        if v = v_1(mark_1(e)) then
23:            prev_1(mark_1(e)) = 0;
24:        else
25:            prev_2(mark_1(e)) = 0;
26:        end if
27:    end if
28:    if v ∈ V_odd then
29:        mark_1(last) = 0;
30:        return v;
31:    end if
32:    e = Stack(v);
33:    M3:
34:    mark_1(last) = e; last = e;
35:    while (last ≠ 0);
36:    return v;
37: end procedure
```

некоторой вершине $v \in V_{odd}$, $v \neq w$. В результате выполнения процедуры будет построена простая цепь $C = v_0 e_1 v_1 e_2 ... e_k v_k$, в которой

$$v_1, v_2, ...v_{k-1} \notin V_{odd}, \quad v_0, v_k \in V_{odd},$$

$$e_i = \arg \max_{e \in E(v_i) \setminus \{e_l | l<i\}} \text{rank}(e),\ v_{i+1} = \overline{v_1}(e_i),\ i = 1, 2, \ldots, k,$$

кроме того, для любой начальной части $C_l = v^0 e_1 v_1 e_2 v_2 \ldots e_l$, $l \leq k$ и для любой вершины $v \in V$ имеет место неравенство

$$\min_{e \in E(v) \bigcap E(C_l)} \text{rank}(e) > \max_{e \in E(v) \setminus E(C_l)} \text{rank}(e).$$

Сначала производится инициализация списка M3-помеченных ребер (рисунок 3.7), который будет являться представлением построенной цепи. Из-

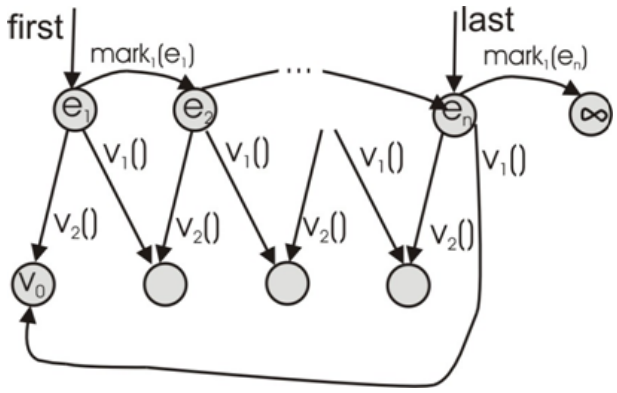


Рисунок 3.7: Организация списка M3-помеченных ребер

начально этот список состоит из ребра e, находящегося в начале w-списка M2-помеченных ребер. Вершина w определяется как текущая вершина v. В цикле `do...while` с помощью процедуры `REPLACE` устанавливается $\overline{v_2}(e) = v$, и ребро e исключается из $v_1(e)$- и $v_2(e)$-списков M2-помеченных ребер, а текущей устанавливается вершина $v_1(e)$. Если $v \notin V_{odd}$, очередь M3-помеченных ребер пополняется ребром e, в противном случае процедура возвращает текущую вершину $v \in V_{odd}$.

В результате выполнения процедуры `FormChain()` будет построена простая цепь $C = v^0 e_1 v_1 e_2 ... e_k v_k$, в которой

$$v_1, v_2, ...v_{k-1} \notin V_{odd}, \quad v^0, v_k \in V_{odd},$$

$$e_i = \arg \max_{e \in E(v_i) \setminus \{e_l | l<i\}} \text{rank}(e),\ v_{i+1} = \overline{v_1}(e_i),\ i = 1, 2, \ldots, k,$$

кроме того для любой начальной части $C_l = v^0 e_1 v_1 e_2 v_2 \dots e_l$, $l \le k$ и для любой вершины $v \in V$ имеет место неравенство

$$\min_{e \in E(v) \bigcap E(C_l)} \mathrm{rank}(e) > \max_{e \in E(v) \setminus E(C_l)} \mathrm{rank}(e).$$

Лемма 4. *Для любых* $j = 1, 2, \dots, l$ *и* $m = 1, 2, \dots, M$, *где* $M = \max\limits_{e \in E} \mathrm{rank}(e)$, *имеет место равенство* $\mathrm{Int}(C_j) \bigcap G(E_m) = \emptyset$.

Для **доказательства** воспользуемся методом математической индукции по переменной m.

Поскольку ребра множества E_1 образуют цикл, ограничивающий внешнюю грань f_0 графа G, то имеет место

$$(\forall H \subseteq G) \left(E_1 \bigcap \mathsf{Int} G(H) = \emptyset \right),$$

что доказывает справедливость леммы для $m = 1$.

Покажем, что из

$$\mathsf{Int}(C_j) \bigcap G(E_m) = \emptyset,\ j = 1, 2, ..., l,\ m = 1, 2, ..., K-1, \quad \text{где } K \le M$$

следует

$$\mathsf{Int}(C_j) \bigcap G(E_m) = \emptyset,\ l = 1, 2, ..., L. \tag{3.5}$$

Предположим противное, пусть нашлось такое l', что

$$\mathsf{Int}(C_{l'}) \bigcap G(E_m) \neq \emptyset.$$

В соответствии с леммой 1

$$\mathsf{Int}(G(E_m)) \supseteq \overline{E_m} = \{e :\ \mathrm{rank}(e) > m\},\ m = 1, 2, ..., M.$$

Из связности графа G и леммы 1 следует, что множество $\mathsf{Int}(C_{l'}) \bigcap E_m$ содержит ребро $e' \in E$ такое, что $v_2(e') = v \in V(C_{l'})$.

Поэтому

$$\min_{e \in E(v) \bigcap EC_{l'}} \mathrm{rank}(e) < \mathrm{rank}(e') \le \max_{e \in E(v) \setminus EC_{l'}} \mathrm{rank}(e),$$

т.е. имеем противоречие с (3.2), что доказывает справедливость равенства (3.5). Применяя принцип математической индукции, получаем утверждение леммы:

$$\mathsf{Int}(C_l) \bigcap G(E_m) = \emptyset, \quad l = 1, 2, ..., L, \quad m = 1, 2, ..., M.$$

Лемма 4 доказана.

Доказательство леммы 4 завершает доказательство результативности процедуры `FormChain`.

В результате выполнения цикла `while...do` алгоритм `OE-Cover` строит последовательность OE-цепей

$$C^0 = v^0 e_1^0 v_1^0 e_2^0 \dots e_{k_0}^0 v_{k_0}^0,\ C^1 = v^1 e_1^1 v_1^1 e_2^1 \dots e_{k_1}^1 v_{k_1}^1, \dots,$$
$$C^{n-1} = v^{n-1} e_1^{n-1} v_1^{n-1} e_2^{n-1} \dots e_{k_{n-1}}^{n-1} v_{k_{n-1}}^{n-1},\ \text{где } n = |V_{odd}|/2,$$

в которой

$$V_{odd} = \left\{ v^0, v_{k_0}, v^1, v_{k_1}, \dots, v^{n-1}, v_{k_{n-1}} \right\},$$

$$(\forall k < (|V_{odd}|/2) - 1,\ \forall l > k)\, \mathsf{Int}(\bigcup_{i=0}^{k} C^i) \cap C^l = \emptyset.$$

После этого выполняется процедура `FormChain` для вершины v_0, смежной внешней грани. Результатом выполнения данной процедуры будет цикл с упорядоченным охватыванием C^n, состоящий из ребер, не содержащихся в цепях C^i, $i = 0, 1, \dots,\ n-1$, где $n = |V_{odd}|/2$.

Результативность алгоритма доказана.

Легко заметить, что вычислительная сложность алгоритма `OE-Cover` не более $O\left(|E| \cdot \log |V|\right)$ операций.

Различые аспекты алгоритма `OECover` отражены в работах [69, 86, 141].

Алгоритм `OE-Cover` определяет дополнительные ребра, соединяющие конец текущей и начало последующей цепей. Эти ребра образуют множество M, существование которого утверждается в теореме 15. Они представляют собой некоторое паросочетание на множестве V_{odd}. Приведем доказательство более сильного результата, анонсированного в [56]: построение последовательностей цепей с упорядоченным охватыванием с любым множеством M, образующим паросочетание на множестве V_{odd} [65, 67, 93].

3.7 Ранжирование ребер, вершин и граней

Введенная в доказательстве функция $\mathrm{rank}(e)$ в дальнейшем будет существенным образом использоваться для решения задач маршрутизации в неэйлеровых графах, несвязных графах и в задачах построения OE-цепей с дополнительными локальными ограничениями. Поэтому целесообразно определить процедуру `Ranking(G)`, которая для любого плоского графа для любого $e \in E(G)$ вычисляет значения $\mathrm{rank}(e)$. Наряду с функцией ранга ребра в дальнейшем будут использованы функции ранга вершины $\mathrm{rank}(v)$ и ранга грани $\mathrm{rank}(f)$.

Определение 12. ***Рангом вершины*** $v \in V(G)$ *будем называть значение функции* $\mathrm{rank} : \ V(G) \to \mathrm{N}$*:* $\mathrm{rank}(v) = \max_{e \in E(v)} \mathrm{rank}(e)$, *где* $E(v)$ *– множество ребер инцидентных вершине* $v \in V$.

Определение 13. ***Рангом грани*** $f \in F(G)$ *будем называть значение функции* $\mathrm{rank} : \ F(G) \to \mathrm{Z}^{\geq 0}$*:*

$$\mathrm{rank}(f) = \begin{cases} 0, & \text{при } f = f_0, \\ \min_{e \in E(f)} \mathrm{rank}(e), & \text{в противном случае,} \end{cases}$$

где $E(f)$ *–множество ребер инцидентных грани* $f \in F$.

Из определения 10 ранга ребра следует, что его численное значение определяет удаленность этого ребра от внешней грани и показывает, какое минимальное число граней необходимо пересечь, чтобы добраться от внешней грани f_0 до этого ребра. Это позволяет для определения ранга использовать граф $G'(V, E, F)$, топологически двойственный исходному графу $G(V, E, F)$. Очевидно, что имеют место соотношения:

$$V(G') = F(G), \quad V(G) = F(G'), \quad E(G') \leftrightarrow E(G).$$

Введенная функция ранга ребра $\mathrm{rank}(e)$ фактически может быть определена как расстояние в двойственном графе от f_0 до ближайшей грани $f \in F(G)$, инцидентной ребру e. Поскольку граням графа G соответствуют

вершины в двойственном графе, то это сводится к нахождению кратчайшего пути между вершинами в двойственном графе G'. Для нахождения рангов всех вершин достаточно построить в двойственном графе G' дерево кратчайших путей с корнем в вершине f_0.

Далее будем считать, что процедуру нахождения рангов всех ребер, вершин и граней графа G реализует алгоритм `Ranking(G)`. Фактически, данный алгоритм представляет алгоритм Дейкстры и его вычислительная сложность при соответствующей организации структур данных не превосходит $O(|E| \log |E|)$. Для построения двойственного графа не нужно выполнять дополнительных операций, так как для каждого ребра e уже заданы функции $f_1(e)$ и $f_2(e)$, соответствующие номерам граней, связываемых ребром.

В качестве примера рассмотрим граф, представленный на рисунке 3.8. На рисунке круглой рамкой обведены номера граней, в соответствии с номерами вершин двойственного графа G' (рисунок 3.9). Поиск начинается с вершины, соответствующей внешней грани. Мосты в двойственном графе представляются как петли. Таким образом, для рассмотренного примера ранг 1 имеют ребра, инцидентные вершине 1, ранг 2 – все ребра, инцидентные вершине 2, а ранг 3 – ребро $3-4$, что соответствует рангам, отмеченным на рисунке 3.8 [46, 51, 99].

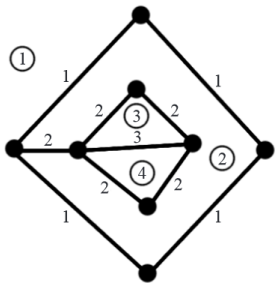


Рисунок 3.8: Исходный граф G

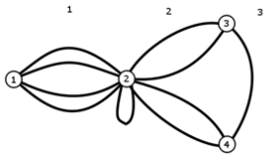


Рисунок 3.9: Ранги ребер на графе, двойственном графу G

3.8 Оптимальное покрытие плоского графа последовательностью OE-цепей

Рассмотрим алгоритм построения оптимального по длине дополнительных построений покрытия произвольного плоского связного графа без висячих вершин и мостов последовательностью цепей с упорядоченным охватыванием, а также доказательство его результативности [67, 77].

Теорема 12. *Пусть $G = (V, E)$ – плоский связный граф на S, не имеющий висячих вершин и мостов. Для любого множества M, являющегося паросочетанием на множестве V_{odd} графа G, и такого, что $(M \cap S)\backslash V = \emptyset$, существует эйлеров цикл $C = v_1e_1v_2e_2...e_nv_1$, $n = |E| + |M|$, для любой начальной части которого $C_l = v_1e_1v_2e_2...v_l$, $l \leq |E| + |M|$, выполнено условие $\mathsf{Int}(C_l) \cap G = \emptyset$.*

Доказательство этой теоремы также конструктивно и состоит в доказательстве результативности алгоритма 16.

Через f_v обозначим грань, для которой $\mathrm{rank}(f_v) = \mathrm{rank}(v)$.

Для анализа результативности алгоритма по аналогии с доказательством теоремы 9 положим

$$E_k = \{e \in E : \mathrm{rank}(e) = k\}, \quad E_k^* = \{e \in E : \mathrm{rank}(e) \leq k\},$$
$$\overline{E_k} = E\backslash E_k^*,$$

через $G(E')$ будем обозначать плоский граф, порожденный множеством ребер $E' \subset E$. Леммы 1 и 2 остаются справедливыми. Из этих лемм непосредственно следует результативность шага 1 алгоритма M-`Cover`.

Algorithm 16 Алгоритм M-`Cover`

Require: связный плоский граф G, функции $v_k(e)$, $l_k(e)$, $e \in E(G)$, $k = 1, 2$; вершина $v_0 \in V(G)$, инцидентная внешней грани; паросочетание M на множестве вершин V_{Odd} нечетной степени;

Ensure: вполне упорядоченное множество C из OE-цепей графа G, представляющее OE покрытие графа G;

Промежуточные данные:

для каждого $v \in V(G)$ очередь $Q(v)$ инцидентных вершине ребер $e \in E(V)$, упорядоченная в порядке убывания ранга;

вершины $u, v \in V(G)$; ребро $e \in E(G)$; символ конца цепи $\#$;

для каждой вершины $v \in V(G)$ пометка $\mathrm{Odd}(v)$;

$C << v_0$; ▷ Инициализация

for all $v \in V(G)$ **do**

$$\mathrm{Odd}(v) := \begin{cases} \textbf{true}, \text{ если } v \in V_{Odd}; \\ \textbf{false}, \text{ если } v \notin V_{Odd}; \end{cases}$$

end for

for all $e = \{v, w\} \in M$ **do**

$\mathrm{Flag}(e = \{v, w\}) = ((v \in \mathrm{Int}(f_w)) \vee (w \in \mathrm{Int}(f_v)))$.

end for

Ranking (G);

for all $v \in V(G)$ **do** сформировать очередь $Q(v)$;

end for

$v := v_0$; ▷ Построение

while $Q(v) \neq \emptyset$ **do**

repeat

$e << Q(v)$; ▷ Переместить из $Q(v)$ первый элемент в e

$v := u : e = \{v, u\}$; $C << e << u$; ▷ Следующие ребро и вершина

until $\mathrm{Odd}(v)$;

if $Q(v) = \emptyset$ **then**

$u = M(v)$; ▷ Вершина u является напарником вершины v

$\mathrm{Odd}(u) := \mathrm{Odd}(v) :=$ **false**; ▷ Удалить вершины u, v из V_{Odd}

$C << v << \# << u$; $v := u$; ▷ Завершение текущей цепи

else

if $(\mathrm{rank}(v) \leq \mathrm{rank}\,(M(v)) \wedge \mathrm{Flag}(\{v, M(v)\}))$ **then**

$C << v << \# << M(v)$; $v := M(v)$; ▷ Завершение цепи

end if

end if

end while

Останов

В теле алгоритма M-`Cover` организована такая последовательность применения шагов 2–7, при которой благодаря отсутствию мостов концом цепи будет либо тупиковая вершина, либо транзитная вершина, у которой напарник имеет более высокий ранг.

Для этой последовательности оказывается справедлива лемма, аналогичная лемме 4. Рассмотрим ее доказательство, приведенное в [67].

Лемма 5. *Для любых* $j = 1, 2, \ldots, |E| + |V_{odd}|/2$ *и* $k = 1, 2, \ldots, K$, *где*

$$K = \max_{e \in E} \operatorname{rank}(e),$$

имеет место равенство $\operatorname{Int}(C_j) \bigcap G(E_k) = \emptyset$, *где* C_j – *начальная часть маршрута, построенного алгоритмом* `M-Cover`.

Для **доказательства** воспользуемся методом математической индукции по переменной k.

Поскольку ребра множества E_1 образуют цикл, ограничивающий внешнюю грань f_0 графа G, то имеет место

$$(\forall H \subseteq G) \left(E_1 \bigcap \mathsf{Int}(H) = \emptyset \right),$$

что доказывает справедливость леммы для $k = 1$.

Покажем, что из

$$\mathsf{Int}(C_j) \bigcap G(E_k) = \emptyset,\ j = 1, 2, ..., l,\ k = 1, 2, ..., L-1, \quad \text{где } L \leq K$$

следует

$$\mathsf{Int}(C_j) \bigcap G(E_k) = \emptyset,\ l = 1, 2, ..., L. \tag{3.6}$$

Предположим противное, пусть нашлось такое l', что

$$\mathsf{Int}(C_{l'}) \bigcap G(E_k) \neq \emptyset.$$

В соответствии с леммой 1 имеем

$$\mathsf{Int}(G(E_k)) \supseteq \overline{E_k} = \{e :\ \operatorname{rank}(e) > k\},\ k = 1, 2, ..., K.$$

Из связности графа G, отсутствия в нем мостов и леммы 1 следует, что множество $\mathsf{Int}(C_{l'}) \bigcap G(E_k)$ содержит ребро $e' \in E$ такое, что $v_2(e') = v \in V(C_{l'})$.

Поэтому

$$\min_{e\in E(v)\bigcap EC_{l'}} \mathrm{rank}(e) < \mathrm{rank}(e') \leq \max_{e\in E(v)\setminus EC_{l'}} \mathrm{rank}(e),$$

т.е. имеем противоречие с леммой 1, что доказывает справедливость равенства (3.6). Применяя принцип математической индукции, получаем утверждение леммы:

$$\mathsf{Int}(C_j)\bigcap G(E_m) = \emptyset, \quad j = 1, 2, ..., l, \quad k = 1, 2, ..., K.$$

Лемма 5 доказана.

Из справедливости леммы следует результативность алгоритма. Очевидно, что его вычислительная сложность алгоритма M-`Cover` не превосходит величины $O\left(|E| \cdot \log |V|\right)$. **Теорема доказана.**

При наличии мостов возможна ситуация, представленная на рисунке 3.10. Если маршрут начинается в вершине v_0, то он пройдет по мостам, обойдет

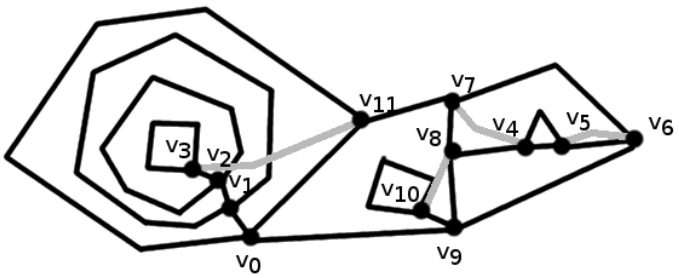


Рисунок 3.10: Пример графа с мостами, для которого алгоритм M-`Cover` не построит OE-маршрут

самую вложенную петлю, попадет в тупиковую вершину v_3 и перейдет в вершину v_{11}. При этом, при любом продолжении маршрута в соответствии с алгоритмом `M-Cover` ребра $\{v_2, v_2\}$ и $\{v_1, v_1\}$ не будут пройдены.

В графе без мостов для построения оптимального покрытия (т.е. покрытия с минимальной длиной дополнительных построений) достаточно в качестве M взять кратчайшее паросочетание на множестве V_{odd} а затем воспользоваться алгоритмом `M-Cover`. Эти действия представлены в алгоритме `OptimalCover`.

Алгоритм `OptimalCover`

Входные данные:

- плоский граф G, представленный списком ребер с заданными на них функциями $v_k(e)$, $l_k(e)$, $f_k(e)$, $k = 1, 2$.

Выходные данные: C_j, $j = 1, ..., |V_{odd}|/2$, – покрытие графа G OE-цепями.

Шаг 1. Найти кратчайшее паросочетание M на множестве V_{odd}.

Шаг 2. Выполнить алгоритм `M-Cover` для графа G и паросочетания M.

Шаг 3. Останов.

Сложность алгоритма `OptimalCover` не превосходит $O(|E| \cdot \sqrt{|V|})$ [158]. Данная сложность достигается за счет решения задачи поиска кратчайшего паросочетания на полном графе.

3.9 Построение OE-покрытия для несвязного графа

Практическую ценность представляет задача построения OE-маршрутов в несвязных графах. В этом случае задачу поиска OE-покрытия графа цепями можно свести к ряду задач меньшей размерности: строить покрытие для каждой компоненты связности в отдельности. Если полученный граф не содержит вложенных компонент, то данный подход является разумным. Однако при наличии вложенности компонент связности задача несколько усложняется и возникают следующие ограничения на порядок обхода компонент связности: компоненты связности, состоящие из ребер более высокого ранга необходимо обходить раньше компонент, состоящих из ребер более низкого ранга.

Поскольку вырезаемые по раскройному плану фрагменты являются прообразами граней, то требование к последовательности обхода граней, гарантирующие отсутствия необходимости резки отрезанных фрагментов, легко формализовать в терминах графа G', двойственного графу G.

Пусть $\preceq$ – отношение частичного порядка на $F(G)$, индуцируемое деревом $T_{G'}^{f_0}$ кратчайших путей до вершины $f_0 \in F$:

$$(f_i \preceq f_j) \Leftrightarrow \left(f_j \text{ принадлежит цепи в } T_{G'}^{f_0} \text{ между } f_i \text{ и } f_0\right).$$

Утверждение 12. *Порядок обхода граней является допустимым в том и только том случае, если он является расширением частичного порядка* $\preceq$.

Определение 14. *Под* ***рангом компоненты связности*** *будем понимать минимальный ранг ребер этой компоненты связности.*

Для задачи построения OE-покрытия в плоском графе можно построить нерекурсивный алгоритм, который находит решение за полиномиальное время [48, 89, 90].

Алгоритм `MultiComponent`

Входные данные:

- несвязный плоский граф G, заданный функциями $v_k(e)$, $f_k(e)$, $l_k(e)$, $e \in E(G)$, $k = 1, 2$;

Выходные данные:

- вполне упорядоченное множество C из OE-цепей графа G, представляющее OE покрытие графа G;

Промежуточные данные:

- множество $S(G)$ компонент связности графа;
- функции $s(v), s(e) \in S(G)$ определяющие принадлежность вершин и ребер графа компонентам связности $s \in S(G)$;
- множество вершин $v_0(s) \in V(G)$, инцидентных внешней грани для каждой компоненты связности;
- $C(s)$ – OE покрытие компоненты связности $s \in S(G)$; # – символ конца цепи;

Шаг 1. <Поиск> Выявить множество компонент связности $S(G)$.

Шаг 2. <Разметка> Определить ранги всех ребер, вершин, граней и компонент связности графа G. Определить $v_0(s)$ для каждой компоненты связности $s \in S(G)$.

Шаг 3. <Сортировка> Сформировать очередь $Q(S)$ компонент связности $s \in S(G)$ в порядке убывания ранга.

Шаг 4. Пока очередь $Q(S)$ не пуста:

$s << Q(S)$; /* переместить из очереди $Q(S)$ первый элемент в переменную s */

перейти на `Шаг 4`, иначе `Останов.`

Шаг 5. Найти покрытие $C(s)$: выполнить алгоритм OE-`Cover` для множества вершин и ребер из s.

Шаг 6. $C << C(s) << \#$. /* завершение текущей цепи и начало следующей */

Конец алгоритма `MultiComponent`

Шаги <Поиск> и <Разметка> представленного алгоритма выполняют поиск по графу в ширину, поэтому их сложность не более $O(|E(G)| \cdot \log|V(G)|)$. Число компонент связности не превосходит числа вершин графа, поэтому сложность шага <Сортировка> не превышает $O(|E(G)| \cdot \log|V(G)|)$. Сложность шагов 3–5 также не превосходит $O(|E(G)| \cdot \log|V(G)|)$, так как алгоритму OE-`Cover` последовательно передаются участки графа G без повторений. Таким образом общая вычислительная сложность алгоритма не превосходит $O(|E(G)| \cdot \log|V(G)|)$.

Полученный в результате выполнения алгоритма маршрут будет удовлетворять частичному порядоку $\preceq$ по построению, однако длина холостых переходов между компонентами связности не оптимизирована.

Рассмотрим пример работы алгоритма. На рисунке 3.11 приведен несвязный граф, в котором предложенный алгоритм связывает все компоненты в соответствии с их рангами ребрами $\{v_{12}, v_7\}$, $\{v_7, v_2\}$ и $\{v_2, v_{11}\}$, обходит по всем внутренним компонентам, строя цепи $C_1 = v_{12}v_{13}v_{14}v_{12}$ и $C_2 = v_7v_5v_6v_7$,

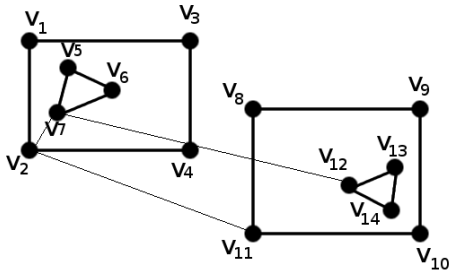


Рисунок 3.11: Пример несвязного графа, содержащего вложенные объединения, и выполненные алгоритмом `MultiComponent` дополнительные построения

и только после этого переходит к обходу внешних компонент и строит цепи $C_3 = v_2v_1v_3v_4v_2$ и $C_4 = v_{11}v_8v_9v_{10}v_{11}$. Очевидно, в данном случае не будет нарушено условие упорядоченного охватывания, однако дополнительные построения не являются оптимальными.

В рамках второго подхода можно предложить несколько методов приведения графа к связному виду.

Определение 15. *Грань $f \in F(G)$ будем называть **разделяющей**, если она инцидентна двум и более компонентам связности.*

Пусть граф $\tilde{G}$ получен из графа G добавлением в разделяющих гранях мостов между компонентами связности. Очевидно, что полученный граф $\tilde{G}$ будет плоским связным графом и для него может быть построен OE-маршрут $M(\tilde{G})$. Искомый OE-маршрут $M(G)$ может быть получен из маршрута $M(\tilde{G})$ если вершины инцидентные введенным мостам считать окончанием текущей цепи и началом следующей (т.е. введенные мосты считать холостыми перемещениями).

Рассмотрим способ построения мостов, связывающих граф G и имеющих минимальную суммарную длину.

Алгоритм `Bridging`

Входные данные: плоский несвязный граф G.

Выходные данные:

плоский связный граф $\tilde{G}$ и множество B добавленных мостов.

Шаг 1. $\tilde{G} := G$; $B = \emptyset$.

Шаг 2. Найти множество C_F всех разделяющих граней.

Шаг 3. Для каждой разделяющей грани $f \in C_F$ выполнить шаги с 3 по 6, после чего `останов.`

Шаг 4. Найти множество $S(f)$ компонент связности графа G инцидентных грани f.

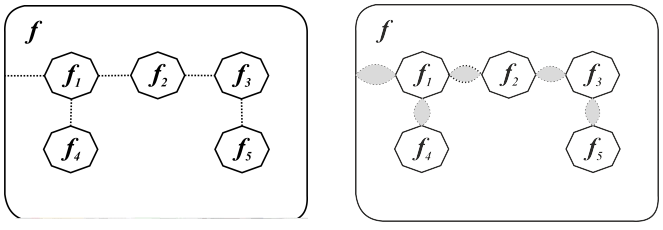
Шаг 5. Построить полный абстрактный граф $\mathcal{T}$, вершинами которого являются компоненты связности $S(f)$, а длины ребер равны расстоянию между соответствующими компонентами.

Шаг 6. Найти остовное дерево минимального веса $T(\mathcal{T})$ в $\mathcal{T}$.

Шаг 7. Добавить ребра найденного остовного дерева в граф $\tilde{G}$: $E(\tilde{G} := E(\tilde{G}) \cup E(T(\mathcal{T}))$, $B := B \cup E(T(\mathcal{T}))$.

Конец алгоритма `Bridging`.

Плоский граф $\tilde{G}$, построенный алгоритмом `Bridging`, содержит мосты, поэтому к нему можно применить только алгоритм OE-`Cover` (без оптимизации холостых переходов). Сократить длину холостых перемещений за счет использования алгоритма M-OE-`Cover` можно, например, добавив ребра остовного дерева $T(\mathcal{T})$ полученного на шаге 5 в граф дважды. Скорректированный таким образом алгоритм назовем `DoubleBridging`(рисунок 3.12(b)). Сложность алгоритмов `Bridging` и `DoubleBridging` является полиномиальной, она зависит от метода определения расстояний между компонентами связности. В случае, если расстояния заданы ее можно оценить как $O(|E(G)| \cdot \log |V(G)|)$.



(a) Bridging (b) DoubleBridging

Рисунок 3.12: Примеры объединения разделяющих граней

Теорема 13. *Если в каждой компоненте связности G_k графа G степени вершин, инцидентных разделяющим граням графа G, четны, то маршрут с минимальной длиной дополнительных построений реализуется алгоритмом* `DoubleBridging`.

Доказательство. Очевидно, что обход каждой компоненты связности должен заканчиваться на внешней границе. Если предположить, что обход компоненты связности начинается из вершины, не принадлежащей внешней границе, то завершится данный фрагмент OE-покрытия в вершине нечетной степени, не принадлежащей границе. Поскольку на границе нет вершин нечетной степени, то часть графа, содержащая внешнюю границу, останется не пройденной. В оптимальном решении компоненты будут связаны парой кратных ребер минимального веса. Причем, суммарный вес всех связывающих ребер должен быть минимален. Очевидно, что такие ребра будут являться ребрами остовного дерева минимального веса $T(\mathcal{T})$, которое строится алгоритмом `Bridging`. **Теорема доказана.**

При наличии на внешней границе некоторых компонент вершин нечетной степени можно привести примеры отсутствия оптимальности в решениях, полученных применением алгоритма `DoubleBridging`. Тем не менее, алгоритм `DoubleBridging` дает решение не хуже решений, построенных алгоритмом `MultiComponent`.

Для графа, приведенного на рисунке 3.13, покрытие, построенное алгоритмом `DoubleBridging` из вершины v_5, будет следующим (приведена последовательность цепей, состоящих из ребер исходного графа, в порядке их обхода, здесь верхний индекс – номер компоненты связности, нижний индекс – номер цепи для данной компоненты связности):

$$C_1^1 = \{v_5v_6v_7v_5\},\ C_1^2 = \{v_1v_3\},\ C_1^3 = \{v_8v_9v_{10}\},\ C_1^4 = \{v_{14}v_{13}v_{12}v_{14}\},$$
$$C_2^3 = \{v_{10}v_{11}v_8\},\ C_2^2 = \{v_3v_4v_2v_1\}.$$

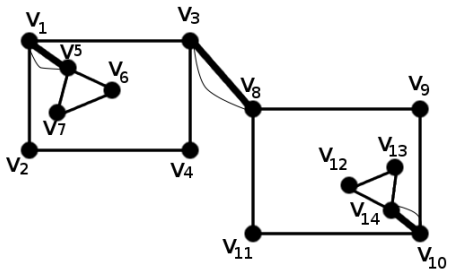


Рисунок 3.13: Добавление ребер, соединяющих компоненты связности, и ребер, делающих граф эйлеровым

Алгоритм `DoubleBridging` позволяет получить не большую длину дополнительных построений, по сравнению с разработанным ранее алгоритмом `MultiComponent` [48] (где переход по дополнительному ребру осуществлялся к ближайшей компоненте связности с внешними ребрами того же ранга). Это объясняется тем, что в данном алгоритме переход осуществляется к ближайшей возможной компоненте связности, а не к ближайшей возможной компоненте связности максимального ранга.

Выводы по главе 3

1. Введенный класс OE-маршрутов является адекватной математической моделью задачи нахождения оптимального (по длине) маршрута движения режущего инструмента при раскрое листового материала.
2. Разработанные алгоритмы решают задачу построения OE-маршрутов для плоского эйлерова графа: связный маршрут китайского почтальона и несвязный маршрут, представляющий OE-покрытие.
3. Разработанные алоритмы решают задачу построения OE-покрытия произвольного плоского (возможно, не связного) графа.
4. Разработанные алгоритмы позволяют осуществить поиск решения для произвольных плоских графов за полиномиальное время. Вычислительная сложность разработанных алгоритмов приведена в таблице 3.1.

Таблица 3.1: Сложность алгоритмов построения OE-маршрутов

Тип маршрута	Алг. сложность	Комментарий
Эйлеров OE-цикл	$O(\|V(G)\|^2)$	Рекурсивный алгоритм
Эйлеров OE-цикл	$O(\|E(G)\| \cdot \log_2 \|V(G)\|)$	Нерекурсивный алгоритм
Задача китайского почтальона	$O(\|V(G)\| \cdot \|E(G)\|)$	При использовании нерекурсивного алгоритма
Задача китайского почтальона	$O(\|V(G)\|^2)$	При использовании рекурсивного алгоритма
OE-покрытие `OE-Cover`	$O(\|E(G)\| \cdot \log_2 \|V(G)\|)$	При использовании логарифмических методов сортировки
Оптимальное OE-покрытие (`OptimalCover`)	$O(\|E(G)\| \cdot \sqrt{\|V(G)\|})$	Основные временные затраты – поиск кратчайшего паросочетания
OE-покрытие `OE-Cover` несвязного графа (`MultiComponent`)	$O(\|E(G)\| \cdot \log_2 \|V(G)\|)$	При использовании алгоритма `OptimalCover` – $O(\|E(G)\| \cdot \sqrt{\|V(G)\|})$
OE-покрытие `OE-Cover` несвязного графа (`DoubleBridging`)	$O(\|E(G)\| \cdot \log_2 \|V(G)\|)$	Если заданы расстояния между компонентами

ГЛАВА 4

ПОСТРОЕНИЕ OE-МАРШРУТОВ С ДОПОЛНИТЕЛЬНЫМИ ОГРАНИЧЕНИЯМИ

При технологической подготовке процесса раскроя имеют место различные ограничения на траекторию движения режущего инструмента. Так, выше была рассмотрена задача построения маршрута, при котором отрезанная часть листа не требовала дополнительных разрезаний. Последовательность вырезания деталей была не важна. Однако на практике это зачастую не так. Например, при огненной резке (flame cutting) требуется, чтобы вырезание продолжалось только по примыкающему контуру. Для решения такой задачи используется цепь с наложенным на нее локальным ограничением. В каждой вершине графа задан циклический порядок обхода ребер, и продолжение обхода по цепи осуществляется только в соответствии с этим циклическим порядком. Данный вид цепей подробно описан в монографии [109]. В общем случае задача поиска такой цепи в графе относится к классу $\mathcal{NP}$-полных задач, однако для некоторых частных случаев существуют эффективные алгоритмы ее решения.

В [109] Г. Фляйшнером приведено следующее определение.

Рассмотрим эйлерову цепь

$$T = v_0 e_1 v_1 \dots e_n v_n,\ v_n = v_0$$

в графе $G = (V, E)$. Предположим, что в каждой вершине $v \in V$ задан циклический порядок $O^{\pm}(v)$, определяющий систему переходов $A_G(v) \subset O^{\pm}(v)$ в этой вершине. В случае, когда для всех $v \in V(G)$ $A_G(v) = O^{\pm}(v)$, систему переходов $A_G(v)$ будем называть **полной системой переходов**.

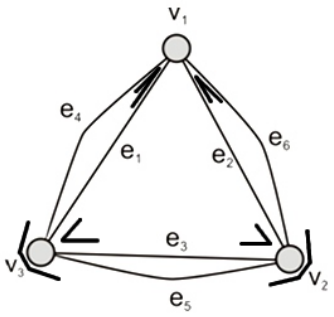


Рисунок 4.1: Пример эйлерова графа

Определение 16. *Эйлеров A_G-совместимый цикл T называют* ***A-цепью*** *[109]. Таким образом, последовательные ребра в цепи T (инцидентные вершине v) являются соседями в циклическом порядке $O^{\pm}(v)$.*

Определение 17. *Будем говорить, что цепь является AOE-цепью, если она одновременно является OE-цепью и A-цепью.*

Рассмотрим граф, приведенный на рисунке 4.1. Жирными дугами отмечены допустимые переходы в графе.

Рассмотрим цепь

$$T_1 = v_1 e_1 v_3 e_3 v_2 e_2 v_1 e_4 v_3 e_5 v_2 e_6 v_1$$

в графе на рисунке 4.1. Данная цепь не является A-цепью, но является OE-цепью. Цепь

$$T_2 = v_1 e_1 v_3 e_3 v_2 e_2 v_1 e_6 v_2 e_5 v_3 e_4 v_1$$

является AOE-цепью. Например, A-цепь

$$T_3 = v_1 e_4 v_3 e_5 v_2 e_6 v_1 e_2 v_2 e_3 v_3 e_1 v_1$$

не является OE-цепью.

В монографии Г. Фляйшнера [109] определены некоторые классы графов, для которых распознавание наличия A-цепи требует полиномиального времени. В работах данного автора приводятся также полиномиальный алгоритм для внешнеплоских графов [112] и для 4-регулярных графов [109] (т.е. графов, степень каждой вершины которых равна 4). В [109, Следствие VI.6] приводится доказательство существования A-цепи для любого связного 4-

регулярного графа на любой поверхности. Для доказательства данного факта автор использует Лемму о расщеплении ([109, Лемма III.26]). Также доказано, что существует полиномиальный алгоритм для распознавания A-цепи в 4-регулярном графе.

4.1 О существовании системы переходов, допускающей AOE-цепь

Цепь, для которой не выполнено условие упорядоченного охватывания, всегда будет содержать переход через охватывающий цикл. Этот переход несовместим с системой переходов A-цепи. С другой стороны имеет место следующая теорема.

Теорема 14. *Если в плоском графе G существует A-цепь, то существует и AOE-цепь.*

Доказательство. A-цепь в плоском эйлеровом графе представляет замкнутую жорданову кривую без самопересечений. Данную кривую можно получить следующим образом: рассмотрим граф G, в каждой вершине которого задана система переходов, соответствующая A-цепи, и граф G', полученный из графа G расщеплением вершин [14, стр.180] в соответствии с системой допустимых переходов (рисунок 4.2). Построенный таким образом граф G' представляет собой жорданову кривую без пересечений и в соответствии с теоремой Жордана разбивает плоскость на внешнюю и внутреннюю компоненты связности.

Очевидно, что на полученной кривой найдется вершина v_0, инцидентная внешней грани графа G и ребру e_n: $\mathrm{rank}(e_n) = 1$. Если принять вершину v_0 за начало AOE-цепи, а направление обхода выбрать таким образом, чтобы ребро (e_n) было концом цепи C_n, то в силу отсутствия пересечений для внутренности любой начальной части цепи $C_i = v_0e_1v_1e_2 \dots e_i$, $i \leq |E(G)|$ будет

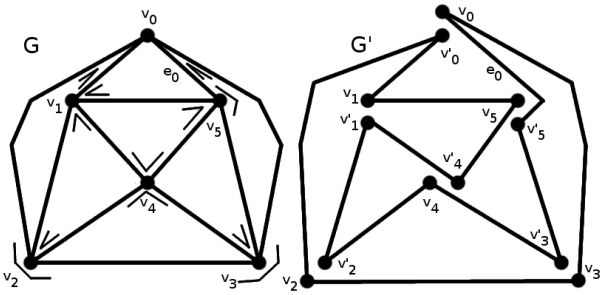


Рисунок 4.2: Плоский граф G с заданной системой переходов A-цепи и соответствующий ему граф G', являющийся жордановой кривой на плоскости

выполнено условие

$$\mathrm{Int}\,(C_i) \cap G = \emptyset,$$

т.е. такая A-цепь C_n будет являться и OE-цепью. Действительно, предположение существования ребра $e \in \mathrm{Int}\,(C_i)$ сразу приводит к противоречию с тем, что в системе переходов цепи отсутствуют пересечения. □

Теорема 15. *В плоском связном 4-регулярном графе G существует AOE-цепь.*

Доказательство. Доказательство существования A-цепи в связном 4-регулярном графе приведено в [109, Следствие VI.6]. Так как в этом графе существует A-цепь, то в силу теоремы 14 в нем существует и AOE-цепь. □

4.2 Алгоритм построения AOE-цепи

Рассмотрим алгоритм, позволяющий построить AOE-цепь в плоском связном 4-регулярном графе. Для компактности изложения будем считать, что входные и выходные данные являются глобальными переменными.

Определение 18. *Суграф G_k графа G, для которого $E(G_k) = \{e \in E(G) : \mathrm{rank(e)} \geq \mathrm{k}\}$ назовем* ***суграфом ранга*** *k.*

Приведенный далее алгоритм AOE-TRAIL накладывает достаточно жесткие ограничения на граф: требуется отсутствие точек сочленения для всех суграфов G_k.

Тем не менее, если предварительно «правильно» расщепить все точки сочленения суграфов G_k, то в результате расщепления получим граф, у которого любой суграф G_k не содержит точек сочленения. Под «правильным» будем понимать переход между дугами, соответствующими циклическому порядку и инцидентными различным парам граней (см. рисунок 4.3).

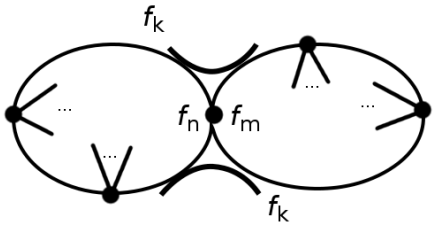


Рисунок 4.3: Верная система переходов для расщепления точки сочленения в суграфе G_k

При поиске точек сочленения используются следующие свойства 4-регулярных графов.

Предложение 1. *Вершина, инцидентная четырем ребрам, смежным внешней грани, является точкой сочленения.*

Предложение 2. *Внешняя грань суграфа G_k является объединением всех граней ранга k в графе G.*

Справедливость этих утверждений очевидна. Из них следует результативность алгоритма 17 [127].

Очевидно, что вычислительная сложность алгоритма не превосходит $O(|E(G)| \log |V(G)|)$.

Рассмотрим работу алгоритма на следующем примере (рисунок 4.4).

После выполнения алгоритма CUT-POINT-SPLITTING получим граф, представленный на рисунке 4.5.

На рисунке 4.6 приведен гомеоморфный образ графа с рисунка 4.5.

Algorithm 17 Алгоритм `CUT-POINT-SPLITTING`

Require: плоский связный 4-регулярный граф $G = (V, E)$, представленный для всех $e \in E(G)$ функциями v_s, l_s, r_s , $s = 1, 2$.

Ensure: гомеоморфный образ графа $G = (V, E)$, представленный для всех $e \in E(G)$ функциями v_s, l_s, r_s , $s = 1, 2$, в котором любой суграф G_k не имеет точек сочленения.
Промежуточные данные: $\forall v \in V(G)$: массивы point(v), rank(v), count(v);
$\forall f \in F(G)$: массив rank(f).

1: Initiate(): ▷ Инициализация
2:
3: **for all do**$v \in V(G)$ point$(v) = 0$; count$(v) = 0$;
4: **end for**
5: **Ranking**(G); ▷ //Определение ранга всех вершин, ребер и граней графа
6:
7: Finding(): ▷ Поиск
8:
9: **for all** $e \in E(G)$ **do**
10: point$(v_1(e)) = e$; point$(v_2(e)) = e$;
11: **end for**
12: **for all** $v \in V(G)$ **do**
13: $e =$ point(v); $k =$ rank(v);
14: **if** $(v = v_1(e))$ **then** $s = 1$
15: **else**$s = 2$;
16: **end if**
17: $e = l_s(e)$;
18:
19: **if** (rank$(e) = k$) **then** count$(v) =$ count$(v) + 1$, $e = l_s(e)$;
20: **end if**
21: **if** (rank$(e) = k$) **then** count$(v) =$ count$(v) + 1$, $e = l_s(e)$;
22: **end if**
23: **if** (rank$(e) = k$) **then** count$(v) =$ count$(v) + 1$,
24: **end if**
25: **if** (count$(v) = 4$) **then**
26: **if** (($f_s(e) = f_s(l_s(e))$ and $f_{3-s}(e) = f_{3-s}(l_s(e))$) or ($f_s(e) = f_{3-s}(l_s(e))$ and $f_{3-s}(e) = f_s(l_s(e))$)) **then**
27: $e^* = l_s(e)$, $l_s(e) = r_s(e)$, $r_s(r_s(e)) = e$,
28: $r_s(e^*) = l_s(e^*)$, $l_s(l_s(e^*)) = e^*$;
29: **else**
30: $e^* = r_s(e)$, $r_s(e) = l_s(e)$, $l_s(l_s(e) = e$,
31: $l_s(e^*) = r_s(e^*)$, $r_s(r_s(e^*)) = e^*$;
32: **end if**
33: **end if**
34: **end for**
Останов.

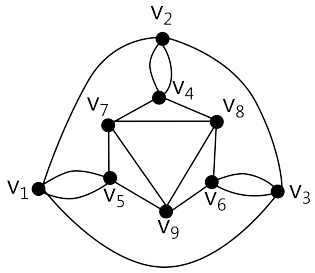


Рисунок 4.4: Пример плоского 4-регулярного графа

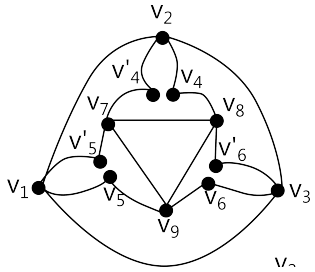


Рисунок 4.5: Граф с расщепленными точками сочленения

Процедура инициализации (`Initiate()`) заключается в инциализации значением 0 счетчика `counter` количества ребер, включенных в результирующую цепь, а также в присваивании всем ребрам пометки `true`, соответствующей непройденному ребру.

Функциональное назначение процедуры `Ranking()` – определить для каждого ребра $e \in E(G)$ графа его ранг $\mathrm{rank}(e)$. Как отмечалось ранее, различные алгоритмы вычисления значений функции $\mathrm{rank}(e)$ приведены в работах [86, 99], их вычислительная сложность не превосходит величины $O(|E| \log |V|)$.

Ниже приведено описание процедуры `Constructing()`.

При выполнении процедуры **Constructing** (e) производится взаимообмен номеров инцидентных ребру e вершин, таким образом, чтобы вершина $v_1(e)$

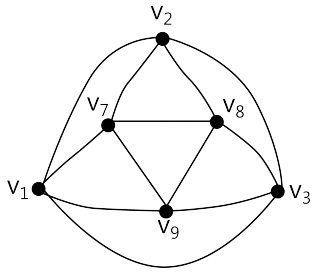


Рисунок 4.6: Гомеоморфный образ графа, представленного на рисунке 4.5

Algorithm 18 Алгоритм AOE-TRAIL

Require: плоский связный 4-регулярный граф $G = (V, E)$ без точек сочленения, представленный функциями v_k, l_k, r_k (рисунок 1.1), $k = 1, 2$;
начальная вершина $v \in V(f_0)$.
Ensure: $ATrail$ – выходной поток, содержащий построенную алгоритмом AOE-цепь.
Initiate(G, v_0); ▷ Инициализация
Ranking(G); ▷ Ранжирование
Constructing(); ▷ Построение
Конец Алгоритма

Algorithm 19 Процедура `Constructing ()`

```
1:  e = arg max_{e∈E(v)} rank(e);
2:  v = v_1(e);
3:  repeat
4:      if (v ≠ v_1(e)) then REPLACE(e);
5:      end if
6:      ATrail << v << e;
7:      mark(e) = false; counter++; v = v_2(e);
8:      if (rank(r_2(e)) ≥ rank(l_2(e))) then
9:          if mark(r_2(e)) then
10:             e = r_2(e)
11:         else
12:             e = l_2(e);
13:         end if
14:     else
15:         if (mark(l_2(e)) then
16:             e = l_2(e)
17:         else
18:             e = r_2(e);
19:         end if
20:     end if
21: until (counter > |E(G)|);
        Конец Процедуры
```

посещалась при обходе раньше вершины $v_2(e)$. Данную функцию реализует процедура `REPLACE`.

Algorithm 20 Процедура `REPLACE` (*Edge*)

1: $tmp1 = v_2(Edge)$; $tmp2 = l_2(Edge)$; $tmp3 = r_2(Edge)$;
2: $v_2(Edge) = v_1(Edge)$; $l_2(Edge) = l_1(Edge)$; $r_2(Edge) = r_1(Edge)$;
3: $v_1(Edge) = tmp1$; $l_2(Edge) = tmp2$; $r_2(Edge) = tmp3$;
Конец Процедуры

Справедлива следующая теорема.

Теорема 16. *Алгоритм AOE-`TRAIL` строит AOE-цепь в плоском связном 4-регулярном графе G, любой суграф G_k, $k = 1, 2, \ldots$ которого не содержит точек сочленения. Алгоритм находит решение за время*

$$O(|E(G)| \cdot \log |V(G)|).$$

Доказательство. Результативность процедур `Initiate()` и `Ranking()` очевидна, а их совокупная вычислительная сложность не превосходит $O(|E(G)| \log |V(G)|)$.

Основной цикл процедуры `Constructing()` состоит в выборе последующего непройденного ребра максимального ранга, смежного ребру e. Процедура выполняется до тех пор, пока все ребра не будут включены в результирующую цепь (их пометка будет изменена на `false`).

Докажем результативность процедуры `Constructing()`. Если для текущего ребра e вершина $v = v_2(e)$ посещается впервые, то выполнение тела цикла можно интерпретировать как расщепление вершины $v^* = v_2(e)$ в соответствии с системой переходов A-цепи. После расщепления гомеоморфный образ полученного графа уже не будет содержать вершины v^*. При повторном попадании алгоритма в вершину $v^* \in V(G)$ алгоритм продолжает формирование цепи в соответствии с гомеоморфным образом полученного ранее ребра.

Гомеоморфный образ полученного графа после выполнения тела цикла является плоским связным графом, так как ни один суграф ранга k не содержит точек сочленения.

После выполнения $|V(G)|$ расщеплений в соответствии с системой переходов A-цепи получим гомеоморфный образ графа, являющийся окружностью. В этом случае поток $ATrail$ будет содержать полученную AOE-цепь.

При выполнении расщеплений гомеоморфный образ остается связным графом, поэтому все ребра будут обработаны алгоритмом (получат пометку `false`). Процедура `Constructing()` имеет единственный цикл. Вычислительная сложность тела цикла не превосходит величины $O(\log|V(G)|)$, а число итераций этого цикла равно $|E(G)|$. Следовательно, вычислительная сложность алгоритма не превосходит величины $O(|E(G)|\log|V(G)|)$. □

AOE-цепь, начинающаяся в вершине v_1 графа, приведенного на рисунке 4.4, будет иметь вид

$$v_1v_7v_9v_8v_7v_2v_8v_3v_9v_1v_3v_2v_1,$$

что соответствует цепи

$$v_1\mathbf{v_5}v_7v_9v_8v_7\mathbf{v_4}v_2\mathbf{v_4}v_8\mathbf{v_6}v_3\mathbf{v_6}v_9\mathbf{v_5}v_1v_3v_2v_1$$

в исходном графе.

Рассмотрим произвольный плоский граф G, в котором, очевидно, нет эйлеровой цепи и, соответственно, A-цепи.

Определение 19. [32] *OE-покрытие будем называть AOE-покрытием, если каждая цепь, входящая в него, соответствует системе переходов $A_G(v) \subset O^{\pm}(v)$.*

Модификация алгоритма `AOE-TRAIL` для случая неэйлерова графа, все степени степени вершин которого не превосходят 4, будет следующей. Граф G необходимо дополнить граф до эйлерова G^* введением вспомогательных ребер, используя любой из методов, предложенных в главе 3.

Началом или концом цепей в этом покрытии будут являться вершины степени 3 и, возможно, одна вершина степени 4, смежная внешней грани [28].

- Если вершина v является началом цепи (т.е. впервые посещается с дополнительного ребра), то первым ребром новой цепи будет ребро, инцидентное вершине максимального ранга. В результате расщепления вершины число вершин нечетной степени станет на 1 меньше.
- Если в вершину v приходим по ребру e графа, то переход осуществляется по ребру (основному либо дополнительному), инцидентному вершине максимального ранга. Если этот переход был осуществлен по дополнительному ребру, то вершина v является концом текущей цепи, а ребро e – последним в текущей цепи. В результате расщепления получим 4-регулярный граф, гомеоморфный образ которого будет иметь на одну вершину меньше.

При таком построении все цепи будут A_G-совместимыми.

4.3 Алгоритм построения самонепересекающейся OE-цепи

Класс AOE-цепей не охватывает полностью всех возможных маршрутов движения режущего инструмента, при которых отсутствуют пересечения имеющихся резов. Общим случаем является решение задачи построения самонепересекающейся OE-цепи, которую будем в дальнейшем называть NOE-цепью (non-intersecting OE-trail). Следует отметить, что в работе [132] ошибочно самонепересекающейся названа A-цепь, для построения которой полиномиальный алгоритм существует только для некоторых частных случаев (например, для 4-регулярного графа). Как будет показано далее, для построения самонепересекающейся OE-цепи существует полиномиальный алгоритм. В работе [9] рассматривается алгоритм построения самонепересекающейся эйлеровой цепи, в общем случае не являющейся OE-цепью.

Определение 20. [31] *Эйлеров цикл в C плоском графе G называется самонепересекающимся, если он гомеоморфен циклическому графу $\tilde{G}$, который может быть получен из графа G с помощью применения $O(|E(G)|)$ операций расщепления вершин.*

Определение 21. [31] *Систему переходов цепи, соответствующую самонепересекающейся цепи, будем называть системой непересекающихся переходов.*

Очевидно, что для системы переходов, соответствующей самонепересекающемуся эйлерову циклу существует такая начальная вершина и такое конечное ребро, смежное внешней грани, для которых построенный цикл будет OE-циклом. Доказательство данного факта во многом будет схоже с доказательством теоремы 14 и будет представлять алгоритм построения такой цепи.

Для построения самонепересекающегося эйлерова OE-цикла в плоском эйлеровом графе, для которого не задано фиксированной системы переходов, можно поступить следующим образом [30].

На множестве вершин графа $V(G)$ определим булеву функцию

$$\text{Checked}(v) = \begin{cases} \texttt{true}, \text{ если вершина просмотрена;} \\ \texttt{false}, \text{ в противном случае.} \end{cases}$$

При выполнении функции инициализации `Initiate()` все вершины объявить непросмотренными.

Процедура `Non-intersecting` (G) (алгоритм 21) расщепляет в графе G все вершины $v \in V(G)$, $k \geq 3$ на k искусственных вершин степени 4 и вводит k искусственных ребер, инцидентных полученным после расщепления вершинам и образующим цикл.

В теле функции используется процедура `Handle` $(e,\ v_k(e),\ k)$ (алгоритм 22), которая обрабатывает каждую непросмотренную вершину графа G. Обработка заключается в расщеплении вершины $v_k(e)$ в соответствии с рис.4.7.

Algorithm 21 Функция `Non-intersecting (G)`

Require: плоский эйлеров граф G;
Ensure: плоский связный 4-регулярный граф G^*;

for all ($e \in E(G)$) **do**
  $k = 1$;
  while ($k \leq 2$) **do**
    if ($\overline{\text{Checked}(v_k(e))}$) **then**
      Handle (e, $v_k(e)$, k);
    end if
    $k++$;
  end while
end forReturn G^*;
Конец Функции

Algorithm 22 Процедура Handle (e,v,k)

▷ Проход 1: Определение степени вершины v
$e_{first} = e$;
$d = 0$; ▷ Степень вершины
repeat
  $le = l_k(e)$;
  if ($v_k(le) \neq v$) **then** REPLACE(le);
  end if
  $e = le$; $d = d + 1$;
until ($e = e_{first}$);
▷ Проход 2: Расщепление вершин, степень которых выше 4
if ($d > 4$) **then**
  $e = e_{first}$; $le = l_k(e)$; fl=new EDGE; $fle = fl$; $e_{first} = e$; $e_{next} = l_k(le)$;
  repeat
    $e = e_{next}$; $le = l_k(e)$; $fr = fl$; fl=new EDGE; $e_{next} = l_k(le)$;
    Pointers(e,le,fr,fl); ▷ Расставить указатели для ребер
  until ($l_k(le) = e_{first}$);
  Pointers(e_{first}, $l_k(e_{first})$, fle, fe); ▷ Расставить указатели для ребер
end if
Конец Процедуры

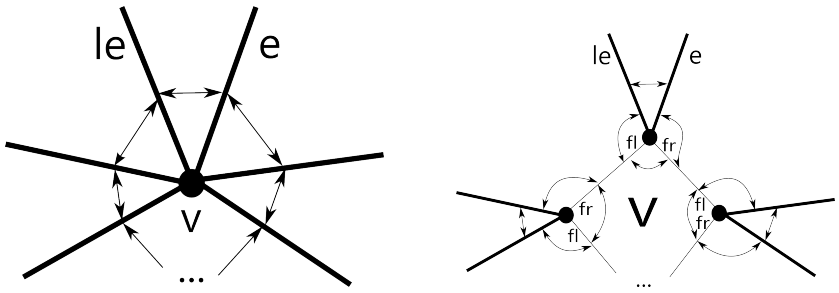


Рисунок 4.7: Расщепление вершины (жирными линиями показаны ребра графа G, тонкими линиями – дополнительные (фиктивные) ребра) и модификация указателей в соответствии с расщеплением

Введенные процедурой `Handle` $k/2$ искусственных вершин и k искусственных ребер, инцидентных этим вершинам, образуют цикл. В результате обработки всех вершин графа G получим модифицированный граф G^*, являющийся плоским связным 4-регулярным графом. Для G^* можно применить алгоритм `AOE-TRAIL()`, который построит в нем AOE-цепь T^*. Если затем в T^* все искусственные ребра и инцидентные им вершины, полученные при расщеплении вершины v, заменить на v, то получим NOE-цепь T в исходном графе G.

Отметим, что данный алгоритм строит NOE-цепь в плоском эйлеровом графе. В случае плоского неэйлерова (в общем случае не связного) графа G необходимо расщепить все вершины степени выше 4 в соответствии с алгоритмом 22. В результате получим граф, степени вершин которого равны 3 или 4. Для этого графа применим алгоритм построения AOE-покрытия. В цепях полученного покрытия удалим все искусственные ребра и стянем все расщепленные вершины. В результате получим NOE-покрытие.

Рассмотрим работу алгоритма на примере графа, приведенного на рисунке 4.8(a). Данный граф имеет две вершины нечетной степени: v_3 и v_6 – смежные внешней грани. Таким образом, построение NOE-маршрута начнем в одной из этих вершин. Пусть это будет вершина v_3.

После применения процедуры `Handle()` получим следующий граф, представленный на рисунке 4.8(b). В полученном графе все вершины имеют степени 3 или 4. На рисунке 4.8(b) отмечены ребра рангов 2 и 3. Заметим, что

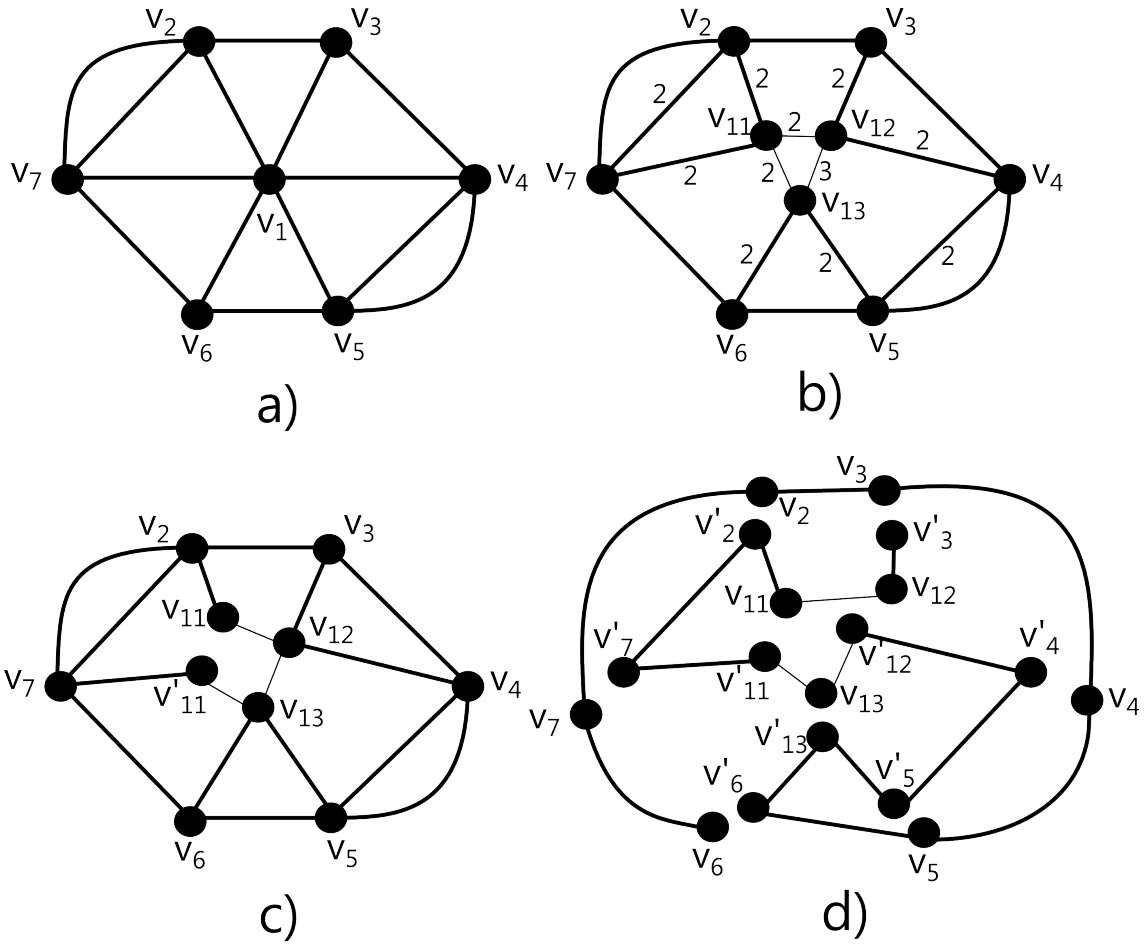


Рисунок 4.8: (a) Пример графа G, имеющего вершины степени выше 4; (b) граф G^*, полученный из графа G расщеплением вершины v_1 в соответствии с алгоритмом `Handle`; (c) граф G_1^*, полученный из графа G^* применением алгоритма `CUT-POINT-SPLITTING`;(d)граф, в котором вершины расщеплены в соответствии с системой непересекающихся переходов.

вершина v_{11} является точкой сочленения ранга 2, следовательно, ее необходимо расщепить с помощью алгоритма `CUT-POINT-SPLITTING`. В результате выполнения указанного алгоритма получим граф, представленный на рисунке 4.8(c). С помощью алгоритма `AOE-Trail` в полученном графе определим AOE-маршрут

$$T^* = v_3' v_{12} v_{11} v_2' v_7' v_{11}' v_{13} v_{12}' v_4' v_5' v_{13}' v_6' v_5 v_4 v_3 v_2 v_7 v_6,$$

которому соответструет NOE-маршрут

$$T = v_3 v_1 v_2 v_7 v_1 v_4 v_5 v_1 v_6 v_5 v_4 v_3 v_2 v_7 v_6$$

в исходном графе (на рисунке 4.8(d) представлен исходный граф с расщепленными в соответствии с системой непересекающихся переходов вершинами).

4.4 О числе эйлеровых OE-цепей для заданной системы переходов

Задача пересчета OE-цепей для плоского графа имеет большой практический интерес, т.к. ее решение позволяет, например, определить возможность раскроя по заданной траектории из различных начальных точек. Рассмотрим плоский эйлеров граф G и OE-цепь T в этом графе. Пусть $X_T(G)$ – система переходов, соответствующая цепи T, тогда верно следующе утверждение [129].

Предложение 3. *Пусть $G(V, E)$ – плоский эйлеров граф без точек сочленения и T представляет OE-цепь в графе G, которой соответствует система переходов $X_T(G)$. Тогда число OE-цепей N для системы переходов $X_T(G)$ удовлетворяет неравенству $1 \leq N \leq 2 \cdot |V(f_0)|$, $V(f_0) = \{v|\ v \in f_0\}$, причем как верхняя, так и нижняя оценки достижимы.*

Доказательство. Существование OE-цепи T доказано в работе [137], откуда следует нижняя оценка. Зафиксируем систему переходов $X_T(G)$ OE-цепи T. Для данной системы переходов все вершины множества $V(f_0)$ можно разбить на два класса: $V_1 = \{v : E(T_G(v)) \in \{e_1, e_2\} : e_1, e_2 \in f_0\}$ и $V_2 = \{v : E(T_G(v)) \in \{e_1, e_2\} : e_1, e_2 \notin f_0\}$. Система переходов для $V \in V_1$ допускает не более двух OE-цепей, стартующих с ребер, ограничивающих внешнюю грань. Если предположить, что цепь стартует с ребра, которое не принадлежит внешней грани, то она и закончится ребром, которое не принадлежит внешней грани, что не удовлетворяет требованию OE-цепи. Для вершин из множества $v \in V_2$, наоборот, построение OE-цепи возможно только при условии старта по ребру, не принадлежащему внешней грани. В противном случае при возврате в выбранную вершину v будет охвачено по крайней мере одно ребро, не смежное внешней грани. Таким образом, заданная система переходов допускает не более $2 \cdot |V(f_0)|$ OE-цепей. Покажем, что эта оцен-

ка достижима. Рассмотрим граф, приведенный на рисунке 4.9. В этом гра-

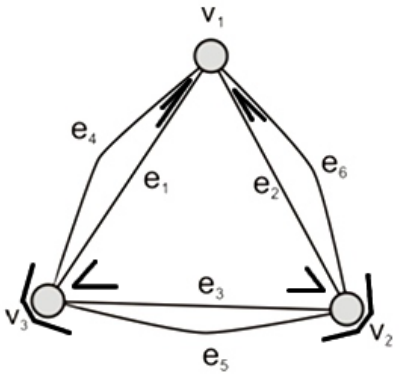


Рисунок 4.9: Пример графа с системой непересекающихся переходов

фе OE-цепь $C_{1,1} = v_1e_1v_3e_3v_2e_2v_1e_6v_2e_5v_3e_4v_1$ индуцирует систему переходов $X_{C_{1,1}}(G) = \{T_G(v_1),\ T_G(v_2),\ T_G(v_3)\}$, где

- $V(T_G(v_1)) = \{e_1, e_2, e_4, e_6\}$; $E(T_G(v_1)) = \{\{e_1, e_4\}, \{e_2, e_6\}\}$;
- $V(T_G(v_2)) = \{e_2, e_3, e_5, e_6\}$; $E(T_G(v_2)) = \{\{e_2, e_3\}, \{e_5, e_6\}\}$;
- $V(T_G(v_3)) = \{e_1, e_3, e_4, e_5\}$; $E(T_G(v_3)) = \{\{e_1, e_3\}, \{e_4, e_5\}\}$.

Для вершины v_1 существует еще одна OE-цепь

$$C_{1,2} = v_1e_2v_2e_3v_3e_1v_1e_4v_3e_5v_2e_6v_1.$$

При этом для вершины $v_2 \in f_0$ OE-цепи

$$C_{2,1} = v_2e_6v_1e_2v_2e_3v_3e_1v_1e_4v_3e_5v_2$$

и

$$C_{2,2} = v_2e_5v_3e_4v_1e_1v_3e_3v_2e_2v_1e_6v_2$$

удовлетворяют системе переходов $X_{C_{1,1}(G)}$, а для вершины v_3 данной системе удовлетворяют OE-цепи

$$C_{3,1} = v_3e_4v_1e_1v_3e_3v_2e_2v_1e_6v_2e_5v_3$$

и

$$C_{3,2} = v_3e_5v_2e_6v_1e_2v_2e_3v_3e_1v_1e_4v_3.$$

Таким образом, в графе из трех вершин имеется шесть $X_T(G)$-совместимых OE-цепей.

Рассмотрим теперь тот же граф с другой системой переходов $X_C(G)$ (рисунок 4.10). Основным отличием данной системы переходов от системы пе-

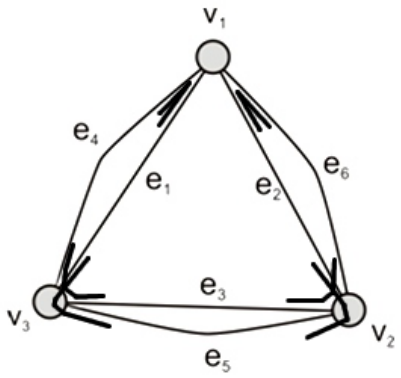


Рисунок 4.10: Пример графа, система переходов которого имеет пересечения

реходов, заданной на рисунке 4.9, является наличие пересечений переходов в вершинах v_2 и v_3.

В данном случае граф имеет единственную OE-цепь

$$C = v_2e_3v_3e_4v_1e_1v_3e_5v_2e_2v_1e_6v_2$$

для заданной $X_C(G)$. В случае выбора как вершины v_1, так и вершины v_3 получим, что цикл $v_1e_1v_3e_5v_2e_2v_1$ охватывает еще непройденное ребро e_3. Заметим, что стартовым ребром в данном случае может быть только ребро e_3. Следовательно, достижима и нижняя оценка **Предложение доказано.**

С практической точки зрения особый интерес представляют графы, для которых верхняя оценка достижима. Из доказательства предложения 3 видно, что не всякий OE-цикл индуцирует систему переходов, для которой будет достигаться верхняя оценка. Отметим также, что для нахождения подходящей системы переходов, для которой достигается верхняя оценка, недостаточно знать только начальную вершину и начальное ребро.

4.4.1 Число OE-цепей для системы переходов, соответствующей A-цепи

Определим число OE-цепей для системы переходов, соответствующей некоторой A-цепи [129]. Очевидно, что пример, приведенный на рисунке 4.9, удовлетворяет данному случаю. Для системы переходов, соответствующей A-цепи, справедливо следующее утверждение об OE-циклах [31].

Теорема 17. *Пусть плоский граф $G = (V, E)$ без разделяющих вершин имеет A-цепь T, которой соответствует система переходов $X_T(G)$. Если $V(f_0)$ – множество вершин, смежных внешней грани, то число OE-циклов для $X_T(G)$ равно $2 \cdot |V(f_0)|$.*

Доказательство. Доказательство факта, что A-цепь, начинающаяся и заканчивающаяся в вершине $v_0 \in f_0$, является OE-циклом, приведено в [79].

Подсчитаем число OE-циклов для фиксированной системы переходов. В [86] доказано, что любой OE-цикл начинается в вершине $v \in f_0$ и завершается ребром $e \in f_0$. В соответствии с условием теоремы, любая вершина $v_j \in f_0$, $j = 1, \ldots |v(f_0)|$ не является разделяющей, поэтому имеет ровно два инцидентных ей ребра, смежных внешней грани f_0. Так как система переходов соответствует A-цепи, то если по одному из этих ребер достигается вершина v_j, по другому цепь выходит из этой вершины. Если оба этих ребра используются только для достижения вершины, то не выполнено условие упорядоченного охватывания (в этом случае одно из этих входящих в вершину ребер оказывается пройдено раньше, чем были пройдены некоторые внутренние ребра). Если оба ребра используются только для покидания вершины, то в системе переходов $X_T(G)$ возникнут пересечения. Однако такая система переходов не соответствует системе переходов A-цепи.

Так как A-цепь является замкнутой последовательностью ребер и вершин, то ее начало может быть помещено в любую вершину, например, в $v_j \in f_0$.

Если v_j является последней вершиной OE-цепи, то необходимо, чтобы в последовательности $e_{j-1}v_je_j$ ребро $e_{j-1} \in f_0$. Действительно, в противном случае последнее ребро e_{j-1} OE-цепи окажется охваченным циклом из ребер, смежных внешней грани. Если за начало цепи принять некоторую вершину $v \in V(f_0)$, то в соответствии с предопределенным циклическим порядком $O^{\pm}(G)$ можно выбрать одно из двух инцидентных ребер для покидания текущей вершины. Следовательно, из произвольной вершины $v \in V(f_0)$ можно построить два OE-цикла. Так как существует $|V(f_0)|$ вершин, смежных внешней грани, число OE-циклов, соответствующих системе переходов для A-цепи, равно $2 \cdot |V(f_0)|$. **Теорема доказана.**

Если в графе $G(V, E)$ имеется несколько разделяющих вершин, то для системы $X_T(G)$, соответствующей A-цепи в данном графе, справедливо следующее утверждение [31].

Теорема 18. *Пусть плоский граф $G = (V, E)$ имеет K разделящих вершин $v_1, \dots v_K \in f_0$ и пусть в этом графе существует A-цепь T. Пусть $X_T(G)$ – система переходов, соответствующая T, а $V(f_0)$ – множество вершин, смежных внешней грани. Существует*

$$2 \cdot |V(f_0)| + \sum_{i=1}^{K} (\deg(v_i) - 2)$$

OE-циклов для $X_T(G)$.

Доказательство. В соответствии с теоремой 17 плоский граф G без разделяющих вершин имеет ровно $2 \cdot |V(f_0)|$ OE-циклов для системы переходов, соответствующей некоторой A-цепи. Пусть $v_i \in V(F_0)$ – разделяющая вершина степени $\deg(v_i) = 2 \cdot M_i$. В циклическом порядке ребер, соответствующем данной вершине, имеется ровно M_i ребер, по которым цепь достигает данную вершину и столько же ребер, по которым она покидает эту вершину. Одна пара ребер уже подсчитана в $|V(f_0)|$, но не учитывается еще $M_i - 1$ возможность начала OE-цикла. Суммируя по всем разделяющим вершинам,

получим выражение, указанное в формулировке теоремы. **Теорема доказана.**

Заметим, что если $X_T(G)$ не соответствует A-цепи, то верхняя оценка не достигается даже если цепь T является самонепересекающейся. Подтверждением данного факта является пример, приведенный на рисунке 4.11.

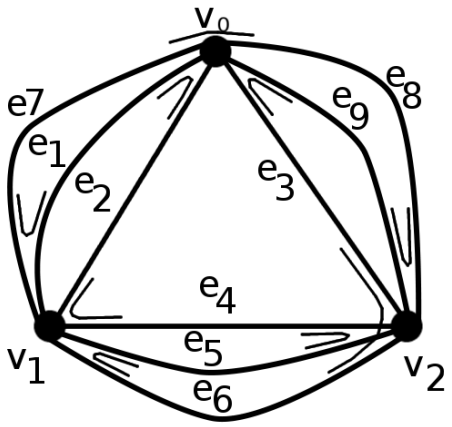


Рисунок 4.11: Пример графа с системой непересекающихся переходов, не допускающей A-цепи

В приведенном графе не существует A-цепи, однако, можно определить систему непересекающихся переходов $X_T(G)$. Для этого графа при заданной системе переходов $X_T(G)$, приведенной на рисунке 4.11, существует только пять OE-цепей, начинающиеся в разных вершинах на внешней грани:

- $C_1 = v_0e_7v_1e_1v_0e_2v_1e_4v_2e_5v_1e_6v_2e_3v_0e_9v_2e_8v_0$;
- $C_2 = v_0e_8v_2e_9v_0e_3v_2e_6v_1e_5v_2e_4v_1e_2v_0e_1v_1e_7v_0$;
- $C_3 = v_1e_1v_0e_2v_1e_4v_2e_5v_1e_6v_2e_3v_0e_9v_2e_8v_0e_7v_1$;
- $C_4 = v_2e_3v_0e_9v_2e_8v_0e_7v_1e_1v_0e_2v_1e_4v_2e_5v_1e_6v_2$;
- $C_5 = v_2e_9v_0e_3v_2e_6v_1e_5v_2e_4v_1e_2v_0e_1v_1e_7v_0e_8v_2$.

При построении цепи из вершины v_1 возможно построение цепи, начинающейся либо с ребра e_1 (в этом случае будет построена цепь C_3, последним ребром которой будет e_7), либо с ребра e_5 (в этом случае последним в цепи будет ребро e_6, однако построенная цепь

$$C_6 = v_1e_5v_2e_4v_1e_2v_0e_1v_1e_7v_0e_8v_2e_9v_0e_3v_2e_6v_1$$

не будет являться OE-цепью, т.к. ребра e_9 и e_3 к моменту их включения в цепь окажутся охваченными).

В общем случае система переходов $X_T(G)$, соответствующая любой OE-цепи, может иметь пересечения (пример цепи, соответствующей системе переходов с пересечениями, приведен на рисунке 4.10). Таким образом, в данном случае число OE-цепей лежит в интервале от 1 до $2 \cdot |V(f_0)|$.

4.4.2 Необходимое условие существования OE-цепи для заданной системы переходов

Рассмотрим частный случай, когда граф $G(V, E)$ является 4-регулярным плоским графом. Тогда в G существует эйлерова цепь T с соответствующей ей системой переходов $X_T(G)$. Выше было доказано, что если $X_T(G)$ не имеет пересечений, тогда число OE-цепей для этой системы переходов равно $2 \cdot |V(f_0)|$.

Предположим, что система переходов $X_T(G)$ имеет хотя бы один пересекающийся переход. В общем случае существование OE-цепи определяется как наличием пересечений в системе переходов, так и их расположением. Например, в графе на рисунке 4.12 приведена система переходов с единственным пересечением, для которой не существует OE-цепи.

Тем не менее, если изменить всего два перехода, то получим систему переходов, которой соответствует некоторая OE-цепь. Например, заменив всего два перехода (в вершинах v_1 и v_2), получим OE-цепь

$$v_2v_6v_7v_0v_5v_8v_0v_6v_5v_1v_4v_8v_7v_3v_2v_3v_4v_1v_2$$

(рисунок 4.13). Граф на рисунке 4.14 имеет систему переходов с тремя пересечениями, которой соответствует также одна OE-цепь. Более того, несложно найти примеры графов, имеющих до $2 \cdot |V(f_0)|$ OE-цепей для систем переходов с пересечениями.

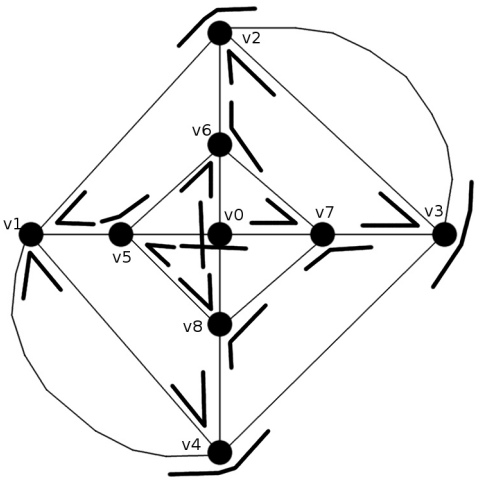


Рисунок 4.12: Пример системы переходов с единственным пересечением, которая не соответствует ни одной OE-цепи.

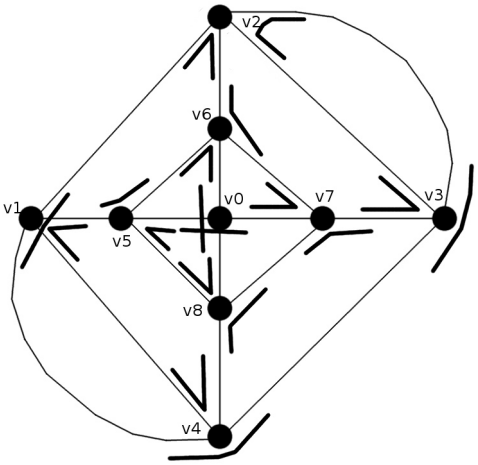


Рисунок 4.13: Пример системы переходов, соответствующей одной OE-цепи.

Прежде чем привести утверждения для 4-регулярных графов, докажем следующее.

Предложение 4. *Если система переходов $X_T(G)$ для некоторой эйлеровой цепи T 2-вершинно-связного 4-регулярного плоского графа G без разделяющих вершин имеет только пересекающиеся переходы, то $X_T(G)$ не соответствует ни одной OE-цепи в графе G.*

Доказательство. Построим модифицированный граф G^*, полученный из графа G расщеплением вершин, имеющих непересекающиеся переходы. Таким образом, если для некоторой вершины v графа G^* ее степень $\deg(v) >$

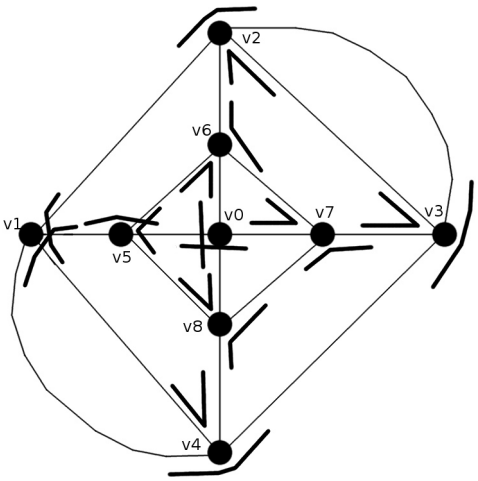


Рисунок 4.14: Еще один пример системы переходов, которой соответствует единственная OE-цепь.

2, то в этой вершине существует пересекающийся переход как в графе G^*, так и в графе G. С точностью до гомеоморфизма будем считать, что все вершины графа G^* имеют степень больше 2, следовательно, во всех вершинах графа G^* имеются пересекающиеся переходы. Предположим, что заданная в условии утверждения система переходов $X_T(G)$ соответствует некоторой OE-цепи T. Рассмотрим 2-вершинно-связный граф G^* (как было сказано выше, имеющий только вершины с пересекающимися переходами) и цепь, начинающуюся с ребра e_0 (на рисунке 4.15 представлены фрагменты такого графа). Все ребра, представленные на рис4.15, являются абстрактными и могут пред-

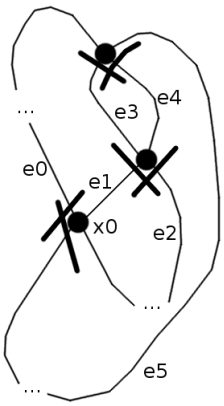


Рисунок 4.15: Некоторые фрагменты графа, имеющего только пересекающиеся переходы

ставлять различные множества ребер. В соответствии с заданной системой

переходов, после e_0 цепь проходит по «ребру» e_2. Так как вершина x_0 не является разделяющей, то цепь, начинающаяся с «ребра» e_2, должна будет пройти по «ребру» e_3, пересечься с «ребром», смежным e_0, и вернуться в вершину x_0. Непосредственно из построения следует, что e_4 в данном случае окажется охваченным циклом, следовательно, построенная цепь не удовлетворяет условию упорядоченного охватывания. Легко видеть, что подобное охватывание возникает и для цепей, начинающихся и с других «ребер». **Предложение доказано.**

Рассмотрим эйлерову цепь T, соответствующую системе переходов $X_T(G)$ 4-регулярного плоского эйлерова графа $G(V, E)$. Построим редуцированный граф $G'(V', E)$. Вершины этого графа, для которых отсутствуют пересечения переходов в системе $X_T(G)$, расщеплены на две вершины. В соответствии с предложением 4 если в графе G' найдется блок, не являющийся циклом, то в графе G для заданной системы переходов не существует OE-цепи.

С другой стороны, граф G' имеет OE-цепь только в том случае, когда каждый блок в G' имеет OE-цепь. Доказательство данного факта очевидно, т.к. все блоки редуцированного графа обходятся последовательно один за другим. Таким образом, если предположить, что существует блок, не имеющий OE-цепи, тогда этот блок, объединенный с остальными, никаким образом не будет иметь такой цепи.

Изложенное дает доказательство теоремы 19.

Теорема 19. *(Необходимое условие существования OE-цепи). Если в редуцированном графе G' существует OE-цепь, соответствующая заданной системе переходов, то в исходном графе G существует хотя бы одна OE-цепь, начинающаяся в вершинах, соответствующих разделяющим вершинам графа G'.*

К сожалению приведенное условие не является достаточным даже для 4-регулярных графов. Например, редуцированный граф G' графа G, пред-

ставленного на рисунке 4.12 является парой петель, инцидентных висячей вершине v_0. В редуцированном графе G' существует OE-цепь, тем не менее, выше было показано, что для данной системы переходов не существует OE-цепи в графе G. Вообще, вершина v_0 в рассмотренном примере не смежна внешней грани, потому из данной вершины невозможно начать построение OE-цепи. Но если начать построение цепи из любой вершины, смежной внешней грани, построить OE-цепь для заданной системы $X_T(G)$ также не удастся. Несмотря на это, в редуцированном графе G' имеется OE-цепь.

Выводы по главе 4

1. Разработанный алгоритм `AOE-TRAIL` позволяет построить AOE-цепь для любого 4-регулярного графа, любой суграф G_k, $k = 1, 2, \ldots$ которого не содержит точек сочленения. Алгоритм находит решение за время $O(|E(G)| \cdot \log |V(G)|)$. Выполнения данного алгоритма не достаточно, чтобы ответить на вопрос о существовании A-цепи в графе. Для графов, степени вершин которого превосходят 4, алгоритм может не построить A-цепь, несмотря на ее существование.
2. Разработан алгоритм `CUT-POINT-DELETING`, позволяющий зафиксировать переходы для всех точек сочленения суграфов G_k, чтобы в результате расщепления получить граф, суграф которого не содержит точек сочленения. Для полученного графа можно применить алгоритм `AOE-TRAIL`.
3. Показано, что OE-цепь можно считать последовательностью нескольких A_G-совместимых цепей.
4. Класс NOE-маршрутов в плоских графах является расширением класса AOE и в него входят все OE-цепи, имеющие непересекающиеся переходы. Разработан алгоритм `Non-intersecting` построения NOE-цепи. Его выполнение состоит в сведении исходного плоского графа к плоско-

му связному 4-регулярному графу за счет расщепления вершин степени выше 4 и дальнейшего выполнения алгоритма `AOE-TRAIL`.

5. В плоском графе G для непересекающейся системы переходов $X_T(G)$ существует не более $2 \cdot |V(f_0)|$ (где $|V(f_0)|$ – число вершин, смежных внешней грани графа) OE-цепей. Если система переходов $X_T(G)$ имеет пересечения, то число ее OE-цепей лежит в промежутке от 1 до $2 \cdot |V(f_0)|$ только тогда, когда в редуцированном графе G' существует OE-цепь. Данные результаты могут быть использованы при технологической подготовке процесса вырезания деталей, когда раскройный план представлен в виде плоского графа, траектория движения режущего инструмента является OE-цепью и требуется определить все возможные точки старта процесса вырезания при фиксированной последовательности вырезания деталей.

ГЛАВА 5
ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ДЛЯ ПОСТРОЕНИЯ OE-ЦЕПЕЙ И OE-ПОКРЫТИЙ

В данной главе привено краткое описание программного обеспечения, позволяющего решить задачи построения цепей и покрытий с упорядоченным охватыванием. Все представленное программное обеспечение разработано автором диссертационной работы, либо под ее руководством ее учениками. Целью создания данного программного обеспечения было проведение тестирования разработанных алгоритмов. Все приложения имеют простейший графический интерфейс, осуществляющий лишь ввод информации о графе либо из входного файла, либо с помощью графического редактора, который позволяет создавать множество вершин графа и соединять их ребрами. При создании ребра осуществляется проверка наличия пересечений с уже созданными ребрами графа, чтобы обеспечить корректность исходных данных (граф должен быть плоским). Программы запрещают создание кратных ребер и пересекающихся ребер. В зависимости от предпочтений пользователя программные реализации алгоритмов могут быть использованы для подключения к промышленным программам.

В дополнение к программам, иллюстрирующим работу алгоритмов, представлен алгоритм `OrderedEnclosingTest`, позволяющий проверить соответствие маршрута обхода плоского графа критерию упорядоченного охватывания. В случае нарушения рассмотренного критерия алгоритм определяет ребро цепи, повлекшее нарушение. Алгоритм может быть применен для повышения надежности программных комплексов, формирующих управляющие программы для станков раскроя, а также на этапе тестирования системы технологической подготовки раскроя в ручном или автоматическом режиме.

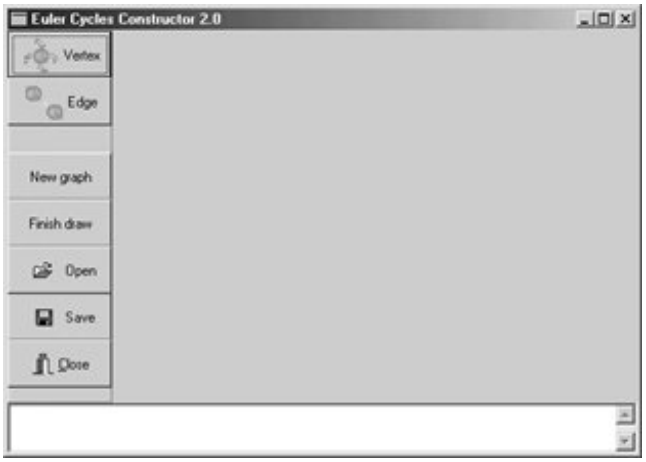


Рисунок 5.1: Внешний вид приложения `MakeCycle`

5.1 Программное обеспечение задачи построения эйлерова OE-цикла

5.1.1 Техника программной реализации рекурсивного алгоритма построения эйлерова OE-цикла

Функция `MakeCycle` [82, 88] используется для построения эйлерова OE-цикла в плоском эйлеровом графе.

Данная функция вызывается в приложении, разработанном в среде визуального программирования C++ Builder, которое представляет собой простой графический редактор, позволяющий с помощью прямых изображать различные плоские графы, сохранять и открывать уже созданные файлы графов. Окно приложения показано на рисунке 5.1.

Слева на окне расположены следующие функциональные кнопки.

1. **Vertex**. Является инструментом для создания вершин графа. Вершины задаются нажатием левой кнопки мыши в рабочей области окна в произвольном порядке.

2. **Edge**. Дает возможность соединять уже обозначенные вершины между собой. Программа не допускает нарушения планарности графа и не позволяет вновь создаваемым ребрам пересекать уже созданные.

3. **New graph**. Вызывает сообщение «Очистить экран». При положительном ответе на сообщение: «Очистить экран» поле для рисования очищается от созданного на нем ранее графа. Все переменные, содержащие информацию о графе, обнуляются.

4. **Finish Graph**. Автоматически становится активной, когда в окне уже создан граф (есть хотя бы две вершины, соединенные ребром). При нажатии на эту кнопку происходит вызов функций построения OE-маршрута.

5. **Open**. Дает возможность загрузить в область рисования ранее созданный граф. В данной версии программы разрешается загружать только один граф, причем добавление в него новых ребер разрешается, но удаление уже созданных ребер в данной версии программы не предусмотрено.

6. **Save**. Сохраняет граф, представленный в окне.

7. **Close**. Закрывает окно программы.

Данное приложение разработано с целью проведения более удобной демонстрации разработанных в диссертационной работе алгоритмов и их тестирования.

Для реализации рекурсивного алгоритма построения OE-цикла использована структура, в которой для каждого ребра заносятся значения определенных для него функций $v_k(e)$, $l_k(e)$, $f_k(e)$, $k = 1, 2$, в переменной `Level` хранится ранг ребра, а переменная `Mark` используется для сохранения пометки ребра:

```
typedef struct{
   int Vertex1, Vertex2;
   int LEdge1, LEdge2;
   int REdge1, REdge2;
   int Mark;
   int Level;
}Edges;
```

Функция `CYCLE`, реализующая выполнение рекурсивного алгоритма, в качестве входных данных принимает указатель на массив структур (граф), номер первого ребра и информацию о количестве ребер графа. Функция выглядит следующим образом.

```
FirstLast CYCLE(Edges *G,int First,int *Number){
```

```
int Vertex, Start, Next, MNext, Edge, Last,MSt;
FirstLast ret;
Start=Next=First;
int NewFirst=ExternCycle(G,Start,&Next,&First,&Vertex,Number);
MSt=0;
do {
   int tmp;
   if(((MNext=G[Next].LEdge2)!=First)
        &&(G[MNext].Mark==Infty)) {
       if (!MSt) MSt=MNext;
       if (Vertex!=G[MNext].Vertex2)
            REPLACE(&G[MNext]);
       L++;
       ret=CYCLE(G,MNext,Number);
       if (G[First].Mark!=Infty)
          tmp=G[First].Mark;
       if (G[ret.First].Vertex2==G[First].Vertex1)
          G[First].Mark=ret.First;
       else
          G[First].Mark=MNext;
       G[ret.Last].Mark=tmp;
       }
    First=Next;
     Next=G[First].Mark;
    Vertex=G[First].Vertex1;
}while(Next!=ret.First&&Next!=Start);
if (!MSt)
   ret.First=Start;
else{
   if (G[ret.First].Vertex2!=G[First].Vertex1&&NewFirst==0)
      ret.First=MSt;
   if (NewFirst!=0&&G[ret.First].Vertex2!=G[First].Vertex1)
      ret.First=Next;
}
if (NewFirst==0)
   ret.Last=First;
else
   ret.Last=NewFirst;
return ret;
}
```

Здесь функция `ExternCycle`, соответствующая первой стадии работы рекурсивного алгоритма, определяет ребра, смежные внешней грани текущего подграфа, а функция `REPLACE` меняет у всех функций текущего ребра индекс k на $3 - k$. Функция полностью соответствует алгоритму, приведенному в разделе 3.3.

Приведем некоторые примеры эйлеровых графов, для которых с помощью рекурсивного алгоритма (см. раздел 3.3) были получены OE-маршруты.

Сначала рассмотрим простейший случай графа с двумя гранями (см. рисунок 5.2).

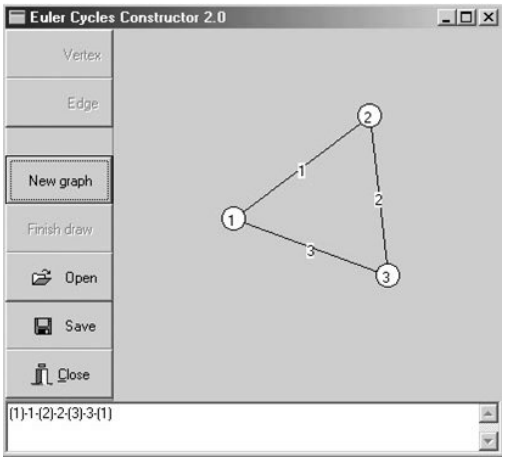


Рисунок 5.2: Граф с двумя гранями

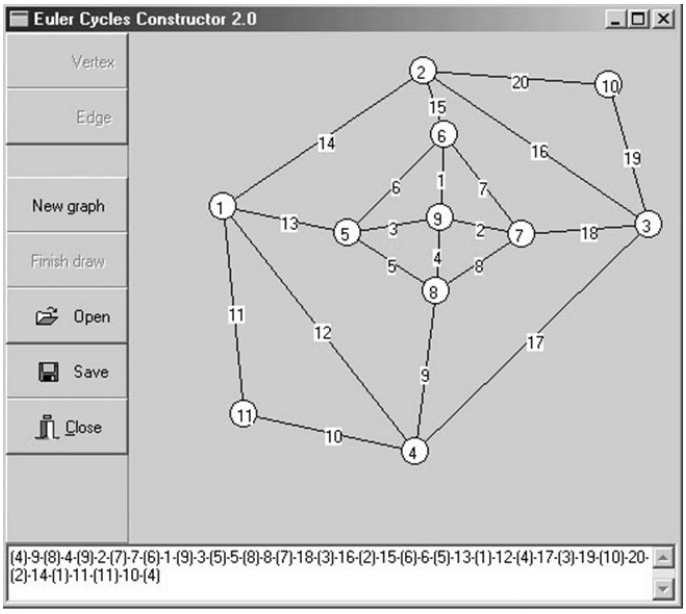


Рисунок 5.3: Граф с ребрами разных рангов

В нижней части экрана рабочего окна приведена последовательность обхода ребер, где в скобках указываются номера вершин. Начав обход в вершине 1, пройдем по ребру 1 в вершину 2, далее по ребру 2 – в вершину 3, а по ребру 3 возвращаемся в исходную вершину, получив, тем самым, цикл, удовлетворяющий условию упорядоченного охватывания.

Рассмотрим более общий пример: граф, имеющий ребра разных рангов (рисунок 5.3).

В данном случае полученный ответ не так очевиден, как в предыдущем примере. Прокомментируем его и покажем, что выполняется условие упоря-

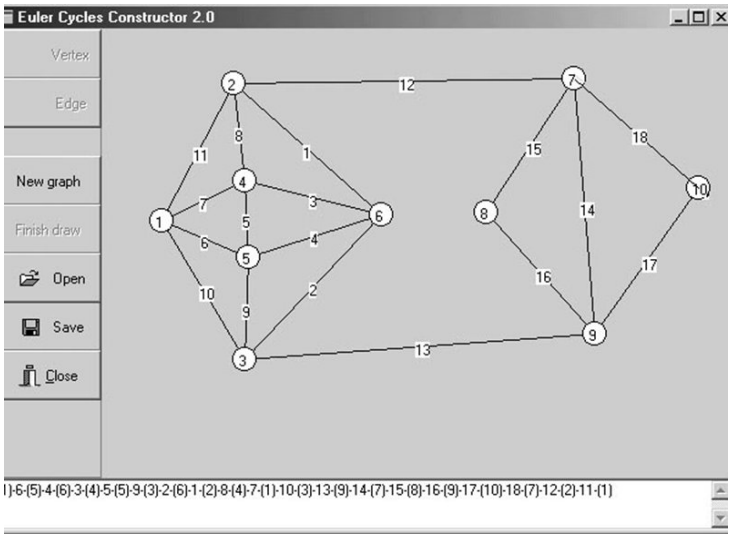


Рисунок 5.4: Граф, в котором внешний цикл охватывает несколько компонент связности

доченного охватывания. В качестве начальной выберем вершину с номером 4 и начинем обход идущего из нее вложенного цикла, проходя по ребру 9. Попав в вершину 8, находим инцидентные ей ребра более высокого ранга (в данном случае – ранга 2), поэтому найдем вложенный цикл из ребер 4, 2, 7, 1, 3, 5, который, как видно из рисунка, и обходится в первую очередь. После обхода самого внутреннего цикла (ранга 3), завершается обход цикла из ребер 9, 8, 18, 16, 15, 6, 13, 12 и лишь после этого обходится цикл из ребер, смежных внешней грани (имеющих ранг 1). Из этого примера легко видеть, что полученный обход снова обладает свойством упорядоченного охватывания.

Приведем еще один пример, когда внешний цикл охватывает несколько компонент связности (рисунок 5.4).

Обход цикла в данном случае также построен по тем же принципам, что и обход из предыдущего примера. Как видно из результатов работы программы две компоненты связности из вложенных циклов, полученные после удаления ребер, смежных внешней грани, обходились в положенном им порядке.

Рассмотрение случаев, наиболее часто встречающихся на практике, показывает корректность программной реализации алгоритма, построенного в работе.

5.1.2 Программа для построения OE-маршрута китайского почтальона

Для реализации данного алгоритма использовалась графическая оболочка, описанная в предыдущем разделе.

Выполнение дополнительных построений (дублирование ребер, инцидентных вершинам нечетных степеней) осуществляется только если для исходного графа были найдены вершины нечетной степени. Данная процедура формирует массив степеней вершин и проверяет четность каждого элемента сформированного массива. Функция дублирования ребра приведена ниже. Параметрами функции является пара инцидентных дублируемому ребру вершин. Функция просматривает список ребер, пока не будет найдено дублируемое ребро (в силу отсутствия кратных ребер такое ребро единственно) и добавляет в соответствии с описанным в разделе 3.6.1 построением информацию о дополнительном ребре и увеличивает значение счетчика дополнительных ребер `numenew`.

```
void AddEdge(int v1, int v2){
      //Найдем ребро, которое дублируем
      for (int i=0;i<nume;i++){
          if ((ed[i].v[0].num==v1&&ed[i].v[1].num==v2)||
                (ed[i].v[1].num==v1&&ed[i].v[0].num==v2)){
              if (ed[i].v[0].num!=v1){
                 int t=v1;
                 v1=v2;
                 v2=t;
              }
              ed[nume+numenew].num=nume+numenew;
              ed[nume+numenew].v[0].num=v1;
              ed[nume+numenew].v[0].x=ver[v1].x;
              ed[nume+numenew].v[0].y=ver[v1].y;
              ed[nume+numenew].v[1].num=v2;
              ed[nume+numenew].v[1].x=ver[v2].x;
              ed[nume+numenew].v[1].y=ver[v2].y;
              for (int l=0;l<2;l++)
                 if (ed[ed[i].el[0]].v[l].num==v1)
                    ed[ed[i].el[0]].er[l]=nume+numenew;
              for (int l=0;l<2;l++)
                 if (ed[ed[i].er[1]].v[l].num==v2)
                    ed[ed[i].er[1]].el[l]=nume+numenew;
              ed[nume+numenew].el[0]=ed[i].el[0];
              ed[nume+numenew].el[1]=ed[i].num;
              ed[nume+numenew].er[0]=ed[i].num;
              ed[nume+numenew].er[1]=ed[i].er[1];
```

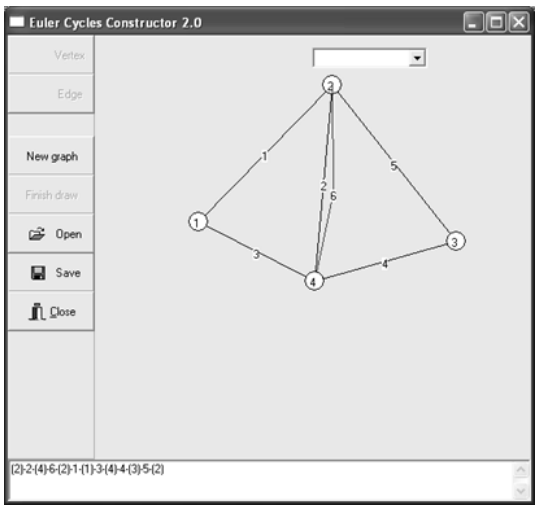


Рисунок 5.5: Один из простейших примеров плоского неэйлерова графа

```
                ed[i].el[0]=nume+numenew;
                ed[i].er[1]=nume+numenew;
            }
        }

            numenew++;
}
```

После выполнения дополнительных построений используется функция построения OE-цикла, описанная в предыдущем подразделе.

Приведем несколько примеров, демонстрирующих работу программы с неэйлеровыми графами.

На рисунке 5.5 приведен простейший пример, когда граф содержит только две вершины нечетной степени (вершины 2 и 4) и они смежны одной грани.

Дополнительные ребра достраиваются ломаной линией. Смысл, который вкладывается в дополнительные ребра, заложен на этапе конструирования модели (в терминах задачи раскроя они понимаются, например, как холостые проходы режущего инструмента), а не на этапе отыскания решения. В нижней части окна, как и в случае эйлерова графа, выводится найденный маршрут с заключенными в скобки номерами вершин, так как маршрут в данном случае содержит и дополнительные ребра.

Для графа с рисунка 5.5 в процессе выполнения алгоритма достраивается дополнительное ребро с номером 6, соединяющее вершины 2 и 4.

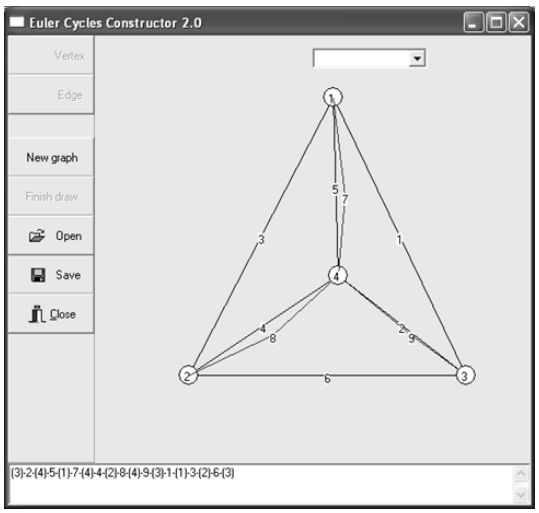


Рисунок 5.6: Граф, не имеющий ни одной вершины четной степени

Рассмотрим теперь пример графа, не имеющего ни одной вершины четной степени (см. рисунок 5.6).

Очевидно, что после пометки внешнего цикла из ребер 1, 3 и 6, остается подграф с тремя висячими вершинами. Чтобы получить внутренний цикл, необходимо достроить три дополнительных ребра 7, 8 и 9. Такое вспомогательное построение не является оптимальным по числу введенных ребер, т.к. для графа с четырьмя вершинами нечетной степени достаточно ввести два дополнительных ребра, для преобразования графа до эйлерова. Тем не менее, проведенное построение позволяет свести граф к эйлерову в процессе работы алгоритма `CPP_OE`.

Приведем более сложный пример (см. рисунок 5.7), для которого необходимо на втором уровне вложенности дублировать ребра-мосты.

Заметим, что условие упорядоченного охватывания выполняется, так как дополнительные ребра являются недостающими звеньями во вложенных циклах, а т.к. полученный эйлеров граф имеет OE-цикл, то и маршрут, в котором некоторые ребра заменяются фиктивным переходом из вершины в вершину, будет иметь упорядоченное охватывание.

Итак, мы рассмотрели несколько часто встречающихся на практике случаев с произвольными плоскими графами. Во всех этих случаях произведена

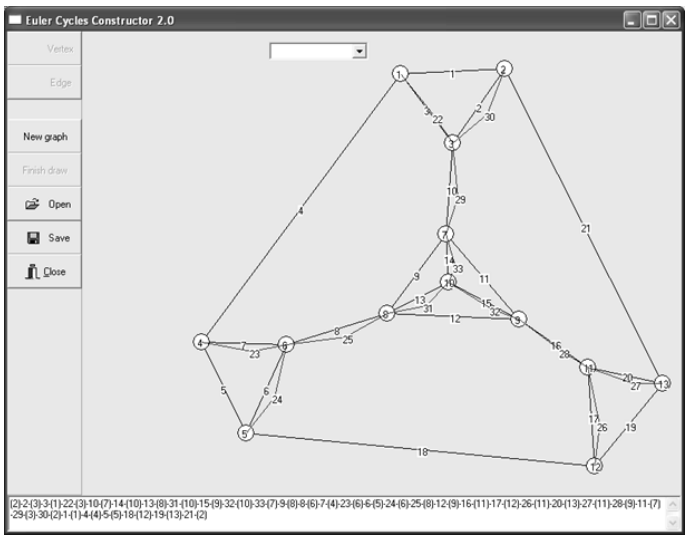


Рисунок 5.7: Плоский неэйлеров граф, для которого не существует тривиальной достройки

оптимальная достройка исходного графа до эйлерова с помощью описанных выше функций и найден маршрут в исходном графе, удовлетворяющий условию упорядоченного охватывания.

5.1.3 Техника программной реализации эффективного алгоритма построения OE-покрытия

Описанный алгоритм также реализован в виде функции и включен в качестве функции в разработанное ранее для реализации рекурсивного алгоритма GUI-приложение [80].

Здесь осуществлена модификация используемых структур данных и граф представлен в виде класса `EG`, содержащего все используемые переменные (свойства) и прототипы функций (методы класса).

```
class EG{
public:
 int EuNumber, EuVertNumb;
 int NFace;
 int First, Last;
 unsigned * Vertex1, *Vertex2;
 unsigned * LEdge1, *LEdge2;
 int * Stack, *Mark1, *Mark2, *prev1, *prev2;
  int *ExtEdge,Enum;
 bool *nech;
 int *NV;
 int NechNum;
 int* VerArray;
 int *kmark;
```

```
 int *kmark_v;
 int KM;
 int edge, NextEdge, FirstEdge;
 int vertex, i;
 //Методы*************************************************************
  int* EuLoop(char *a, char *b, int g, int*nv, int k,double **rasst,int t);
 void WriteToFile(char *OutFile);
 void ReadFromFile (char *InFile);
 void Form(int v);
 int FormNech(int V_Max);
 void Making();
 void Initialisation(int f);
 void REPLACE(int edge);
 void SortNech ();
 void DelVer(int what);
  void GetFromStacks(int vertex);
};
```

Приведем функцию построения OE-покрытия. После считывания данных из файла, выполняется инициализация всех переменных и далее выполняется построение OE-покрытия с помощью алгоритма `OECover`.

```
int* EG::EuLoop(char *InFile, char *OutFile, int first, int*nv,
                int k,double **rasst){
        ReadFromFile(InFile);
        NV=nv;
        NechNum=k;
        VerArray=new int [EuVertNumb+k/2];
        Initialisation(first);
        Making();
       SortNech();
        int v=Vertex1[FirstEdge];
        while(NechNum!=0){
                int q=NV[NechNum];
            DelVer(q);
            v=FormNech(q);
            WriteToFile(OutFile);
            DelVer(v);
        }
        Form(v);
        WriteToFile(OutFile);
        return VerArray;
}
```

Отметим, что данный алгоритм находит только допустимое OE-покрытие плоского графа, так как он не использует поиск кратчайшего паросочетания между парами вершин нечетной степени, а сортирует эти вершины в соответствии с их рангами с помощью функции `SortNech()`. В данной реализации функция `SortNech()` использует пузырьковую сортировку, однако, программная реализация допускает ее замену другой функцией, использующей более эффективные методы сортировки. Программный код функций

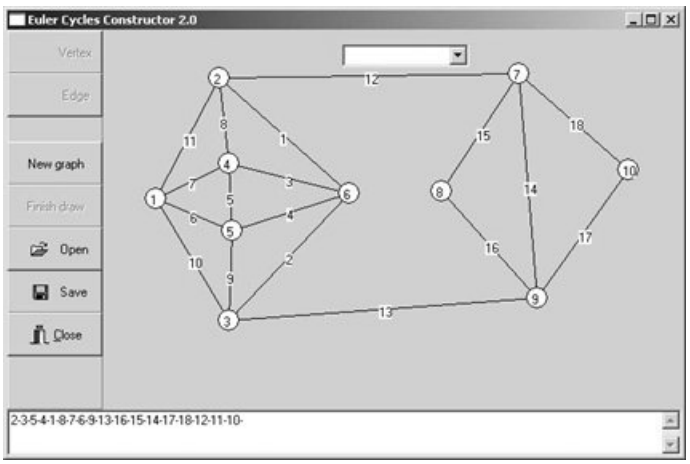


Рисунок 5.8: Работа алгоритма `OECover` для эйлерова графа, в котором цикл из ребер, смежных внешней грани, охватывает несколько компонент связности

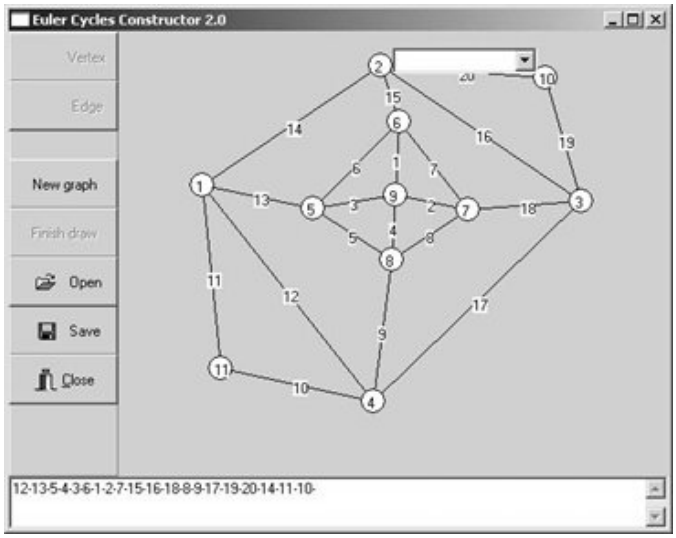


Рисунок 5.9: Работа алгоритма `OECover` для эйлерова графа

`Initialisation()`, `Making()`, `FormNech()` и `Form()` приводить не будем, так как он полностью соответствует описанному в разделе 3.6.2 на псевдокоде алгоритму. Полный текст перечисленных функций приведен в Приложении 1.

Рассмотрим работу алгоритма `OECover` для случая эйлеровых графов, представленных на рисунке 5.8 и 5.9.

Как видно из полученной последовательности ребер, в первую очередь проходятся ребра, имеющие более высокий ранг. В отличие от рекурсивного алгоритма `RECURSIVE_OE`, для нерекурсивного алгоритма `OECover` существуют примеры (рисунок 5.9), когда он не проходит по циклам ребер одного ранга, а выбирает последовательность пройденных ребер по принципу «отре-

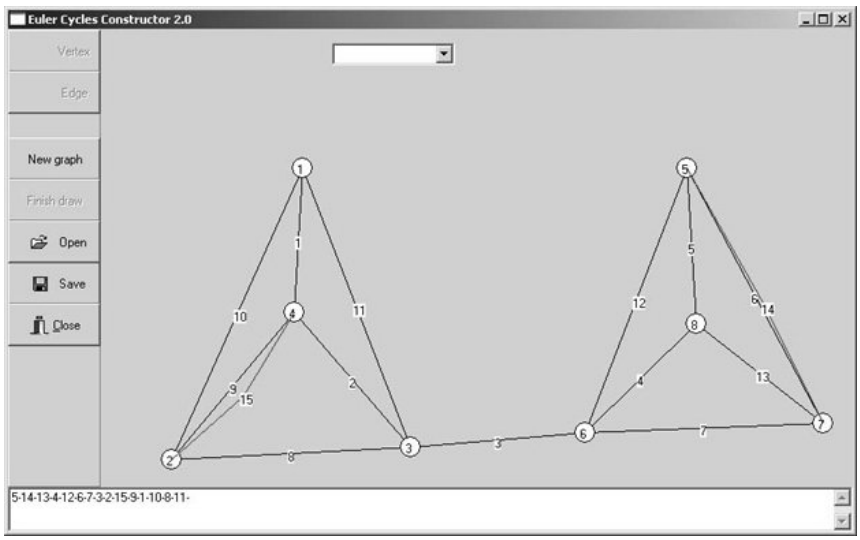


Рисунок 5.10: Работа алгоритма `OECover` для графа с мостом, принадлежащим внешней грани

зания» циклов из ребер разного ранга. Несмотря на такой порядок обхода ребер, условие упорядоченного охватывания сохраняется, а сложность работы алгоритма `OECover` на порядок меньше сложности алгоритма `RECURSIVE_OE`, как это было показано в главе 3.

В случае плоских неэйлеровых графов алгоритм `OECover` выполняет построение эйлеровой цепи с помощью $|V_{odd}|/2$ дополнительных построений, где $|V_{odd}|$ – число вершин нечетной степени. Дополнительные ребра отображаются красным цветом (см. рисунок 5.10–5.12). После полученных дополнительных построений не обязательно модифицированный граф останется плоским (например, см. рисунок 5.12). Легко видеть, что полученные с помощью алгоритма `OECover` эйлеровы цепи удовлетворяют условию упорядоченного охватывания.

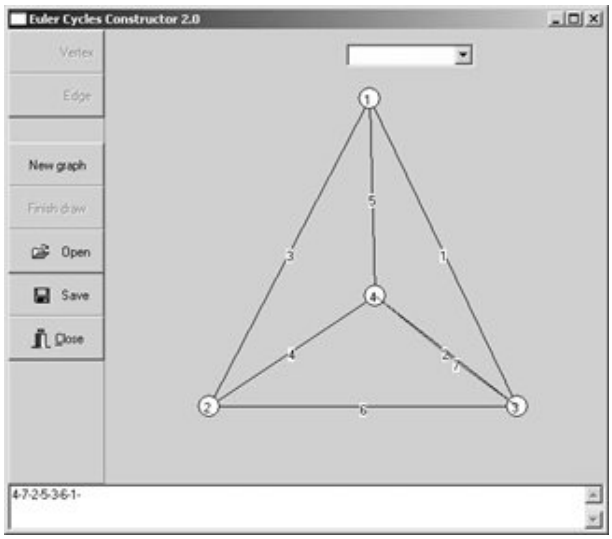


Рисунок 5.11: Работа алгоритма `OECover` для графа, не имеющего вершин четной степени

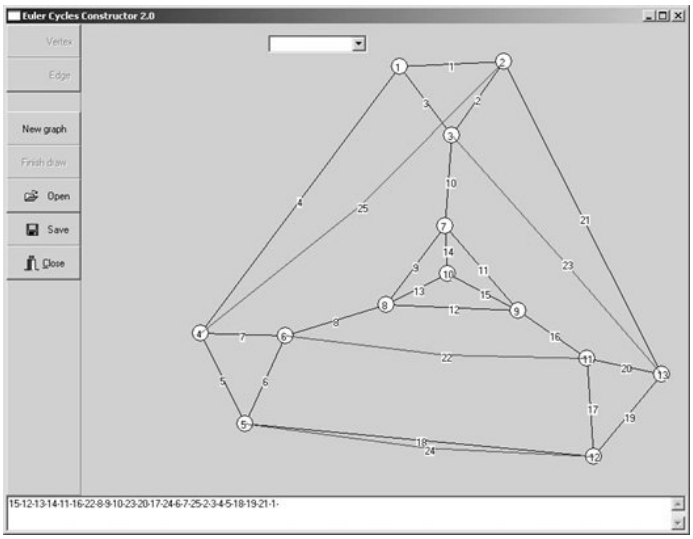


Рисунок 5.12: Работа алгоритма `OECover` для графа

5.2 Программное обеспечение для построения оптимального OE-покрытия плоского связного графа и допустимого OE-покрытия несвязного графа

На основе приведенного в разделе 3.9 алгоритма `MultiComponent` разработана программа «Graph Editor». Она представляет собой простейший графический редактор, позволяющий изображать планарные графы, сохранять информацию о сконструированном с помощью инструментов вставки вершин и ребер графе в файле, загружать, отображать сохраненные программой графы, масштабировать их [42, 81, 85, 152].

Данная программа разрабатывалась с целью отладки алгоритма построения оптимального OE-покрытия для произвольного плоского графа. Пользователь может как решить задачу за один этап (выбрав в главном меню соответствующую команду), так и разбить процесс решения задачи на несколько этапов:

- добавлять и удалять дополнительные ребра;
- помечать компоненты связности;
- находить дополнительные построения минимальной длины между компонентами связности (для этого используется алгоритм Краскала);
- находить OE-покрытие для изображенного в рабочей области экрана графа.

Для представления графа в памяти компьютера в соответствии с указанным в главе 3 способом, а также для задания вспомогательных свойств графа сконструирован следующий класс:

```
class EulerWayMaker {
private:
   struct Vert {
      int x,y; //Координаты вершины
      int selected; //выделена ли вершина
                    //(вспомогательная переменная для поиска мостов)
      int kmark; //уровень вложенности вершины
```

```
   int deg; //Степень вершины
   int odd_go; //вершина нечетной степени оптимальная для перехода
   bool can_go; //true - can make dop edge
   int max_kmark;
   bool mark; //mark for isWay function
};
struct Edge { //Ребро графа
   int v1, v2; //id вершин в векторе V
   int x1, y1, x2, y2; //Координаты начала и конца ребра
   int type;  /*тип ребра {
               0 - внутреннее;
               1 - внешнее;
               2 - дополнительное в маршруте;
               3 - дополнительное длЯ свЯзи компонент свЯзности;}*/
   int bridge; //явлЯетсЯ ли мостом
   double left_angle,right_angle; //Угол в радианах для поиска смежных ребер
   int left_edge, right_edge; //Ближайшее по повороту против часовой стрелки
                              //ребро длЯ первой и второй вершин
   int left_face, right_face; //Номера граней разделенных ребром
   int kmark; //Уровень вложенности ребра
   bool free; //not in way
};
//Вспомогательные переменные
int* out_e; //длЯ вывода ребер
int* out_v; //длЯ вывода вершин
vector<Vert> V; //Вершины графа
vector<Edge> E; //Ребра графа
vector<Edge> dop; //additional edges
vector<int> F; //Грани графа
vector<int> way; //Маршрут в графе (хранит id вектора E)
int hasBridges; //В графе присутствуют мосты
//Функции построениЯ обхода
void Init(); //Собрать информацию о графе
void Order(); //УпорЯдочение (вложенность ребер, списки инцедентных ребер)
void Form();  //сформировать путь
//Вспомогательные функции
int formWay(int s_v,int last_e); //переход из начаьной вершины в очередную
                              //вершину нечетной степени, возвращает end_vert
int getNextWayVert(int n_v, int n_e); //возвращает следующую вершину пути
int getNextWayEdge(int n_v);//возвращает следующее ребро для текущей вершины
int getNextWayEdge(int n_v, int n_e); //возвращает следующее ребро
int maxVertKmark(int n); //Определение максимального ранга
int minVertKmark(int n_v); //Определение минимального ранга
bool hasFreeEdges();  //Проверка непройденных ребер
bool isWay(int start_v, int end_v); //есть ли маршрут из start_v в end_v
void addToWay(int n); //Добавление ребра в маршрут
void addToWayDop(int v1, int v2); //добавлеие дополнительного ребра
int getVertId(int x, int y); //возвращает номер вершины в векторе V или -1
                             // (если вершина не найдена)
void findBridges(); //Найти все мосты графа
int isBridge(int id); //ЯвлЯетсЯ ли ребро id мостом
void countAngles(); //Рассчитать углы ребра
void countSmej(); //Найти ближайшие по повороту против часовой стрелки ребра
                  //для каждой вершины всех рЮбер
void countSmej(int id); //Найти ближайшие по повороту против часовой стрелки
                        //ребра для ребра id
int findBorderEdge(); //Ребро внешней грани длЯ обхода внешней грани
```

```
    void findBorder(); //Отметить внешнюю грань
    void markFace(int fn, int e, int v); //Обойти грань, отметить все ее ребра
    void findFaces(); //Отметить все грани
    void countEdgeKmark(); //Определить уровни вложенности рЮбер
    void findOddPairs(); //Найти вершины нечетной степени

public:
    EulerWayMaker();
    virtual ~EulerWayMaker();

    //Поиск пути
    void findWay(); //Найти обход графа

    //Заполнение графа
    void addEdge(int x1, int y1, int x2, int y2); //Добавить ребро
    //Вывод информации
    int getEdgeCount(); //Количество ребер
    int getWayLength(); //Длина полученного пути
    int getVertCount(); //Боличество вершин
    int* getEdge(int id); //Вернуть вектор ребер {x1,y1,x2,y2,type}
    int* getWayEdge(int id); //Вернуть ребро шага в пути
    int* getVert(int id); //Вернуть вершину {x,y}
};
```

Полный текст всех описанных методов класса `EulerWayMaker` приведен в Приложении 2.

При построении покрытия можно либо использовать оптимизацию (искать решение с помощью алгоритма `MultiComponent`), либо ограничиться поиском решения без оптимизации длины дополнительных построений (в этом случае будут построены дополнительные ребра оптимальной длины между компонентами связности, а для полученного односвязного графа используется алгоритм лексикографического (по возрастанию порядковых номеров вершин) упорядочения дополнительных построений).

Пользователь имеет возможность анимировать полученное решение и просмотреть процесс обхода ребер.

На рисунке 5.13 приведен пример несвязного графа, для которого с помощью разработанного программного обеспечения построено OE-покрытие.

Ребра 6 и 10 построены на этапе связывания компонент. Ребра 18, 21, 23, 28 – дополнительные ребра, имеющие минимальную суммарную длину, по которым осуществляется переход между цепями в покрытии. Все ребра прону-

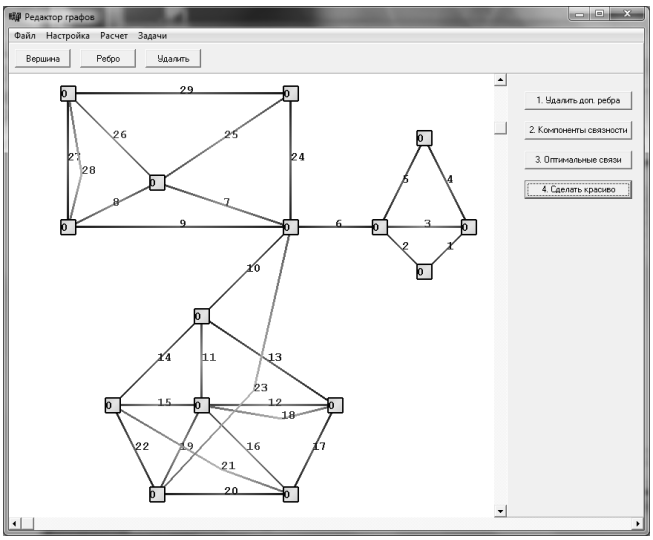


Рисунок 5.13: Работа алгоритма `MultiComponent` для графа с тремя не вложенными компонентами

мерованы в порядке осуществления обхода. Нетрудно видеть, что полученное покрытие имеет упорядоченное охватывание.

5.3 Верификация результатов работы алгоритмов

Рассмотрим алгоритм `OrderedEnclosingTest`, который для заданного графа и маршрута его обхода устанавливает, соответствует ли маршрут условию упорядоченного охватывания [46].

В процессе работы алгоритма построим граневый двойственный граф G^* к графу G, представляющему карту раскроя. Для построения двойственного графа не требуются дополнительные вычислительные операции, так как для каждого ребра e определены грани $f_k(e)$, $k = 1, 2$, смежные этому ребру. Двойственный граф имеет одну компоненту связности, так как исходный граф имеет общую внешнюю грань. Вершину, соответствующую внешней грани графа обозначим как f_0. Приведем алгоритм `OrderedEnclosingTest` проверки маршрута обхода плоского графа на соответствие критерию упорядоченного охватывания.

Алгоритм `OrderedEnclosingTest`

Входные данные:

- Плоский граф $G(V, E)$;
- Маршрут обхода $C = e_1 e_2 \dots e_n$.

Выходные данные: номер ребра в маршруте обхода, нарушающего упорядоченное охватывание (0 – если условие соблюдается)

Шаг 1. Отметить все рёбра графа как не пройденные.

Шаг 2. Выбрать $e = e_1 \in C$ первое ребро обхода.

Шаг 3. Отметить ребро e как пройденное. Выполнить шаги 4–5 для $i = 1, 2$.

Шаг 4. Если в двойственном графе нет пути от $f_i(e)$ до f_0, проходящего только по не пройденным ребрам – перейти к шагу 5.

Шаг 5. Если существует e', не пройденное ребро, инцидентное $f_i(e)$, то условие упорядоченного охватывания нарушается ребром e (перейти к шагу 7).

Шаг 6. Выбрать e – следующее ребро маршрута C, перейти к шагу 3.

Шаг 7. Конец алгоритма. Если e не найдено – вывод 0, иначе вывести e'.

Алгоритм имеет сложность $O(|E|^2 \cdot \log |E|)$ за счет цикла из шагов 3–6, выполняемых для каждого ребра цепи и поиска пути в графе, выполняемого внутри этого цикла на шаге 4.

На каждой итерации цикла (шаги 3–6) рассматривается только грань, включенная в $\mathrm{Int}(C)$ последним добавленным ребром. Чтобы доказать справедливость алгоритма рассмотрим лемму.

Лемма 6. *Пусть* $C = v_1 e_1 v_2 e_2 \dots v_k$ *– маршрут обхода графа, а* $C_i = v_1 e_1 v_2 e_2 \dots e_i$ *– начальная часть маршрута, являющаяся* OE*-цепью, тогда если* $\mathrm{Int}(C_{i+1} \setminus \mathrm{Int}(C_i)) \cap E = \emptyset$*, то* C_{i+1} *– тоже является* OE*-цепью.*

Доказательство. Чтобы C_{i+1} была OE-цепью, по определению необходимо выполнение следующего тождества

$$\mathrm{Int}(C_{i+1}) \cap E = \emptyset.$$

Это тождество можно преобразовать, разбив его на два подмножества $\mathrm{Int}(C_i)$ и $(\mathrm{Int}(C_{i+1})\backslash\mathrm{Int}(C_i))$. Получим

$$(\mathrm{Int}(C_i) \cap \mathrm{Int}(C_{i+1}))\backslash\mathrm{Int}(C_i)) \cap E =$$

$$= \mathrm{Int}(C_i) \cap E \cup (\mathrm{Int}(C_{i+1})\backslash\mathrm{Int}(C_i)) \cap E = \emptyset.$$

Так как $\mathrm{Int}(C_i) \cap E = \emptyset$ по определению, то, чтобы тождество было верным, необходимо $(\mathrm{Int}(C_{i+1})\backslash\mathrm{Int}(C_i))$. **Лемма 6 доказана.**

5.4 Программа построения AOE-цепи

Программа обеспечивает определение последовательности ребер в плоском 4-регулярном графе без точек сочленения ранга $k = 1, 2, \ldots$, удовлетворяющей двум ограничениям на порядок обхода [35]:

- цикл из пройденных ребер не должен охватывать еще не пройденных (условие упорядоченного охватывания);
- в качестве следующего ребра цепи выбирается правый либо левый сосед текущего ребра (A-цепь).

Программа является реализацией представленного в главе 4 алгоритма и может быть использована для демонстрации разработанных алгоритмов поиска решения указанной задачи [33].

Каждое ребро графа представляется списком инцидентных ему вершин и левых и правых соседних ребер, инцдентных каждой из вершин и значением ранга каждого ребра. Например, для графа, представленного на рисунке 5.14, данные закодированы следующим образом:

```
16
1 2 4 12 2 10 1 1
1 6 1 13 3 12 0 2
1 5 2 5 4 13 0 2
1 4 3 7 1 5 1 1
4 5 4 14 6 3 0 2
```

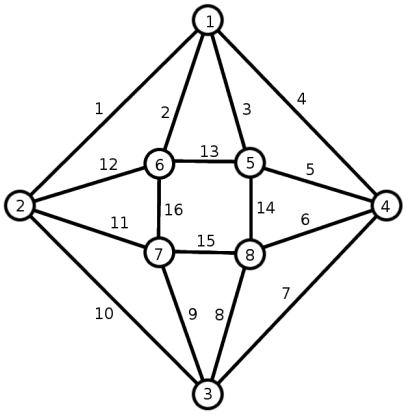


Рисунок 5.14: Пример 4-ругулярного графа с помеченными вершинами и ребрами

```
4 8 5 8 7 14 0 2
4 3 6 10 4 8 1 1
3 8 7 15 9 6 0 2
3 7 8 11 10 15 0 2
3 2 9 1 7 11 1 1
2 7 10 16 12 9 0 2
2 6 11 2 1 16 0 2
6 5 16 3 2 14 0 3
5 8 13 6 5 15 0 3
8 7 14 9 8 16 0 3
7 6 15 12 11 13 0 3
```

Для представления графа в памяти компьютера используются следующие классы:

```
struct GraphEdge{
   int v1,v2; //Инцидентные вершины
   int l1,l2; //Соседние ребра при вращении против часовой стрелки
   int r1,r2; //Соседние ребра при вращении по часовой стрелке
   bool f0; //Флаг смежности ребра внешней грани
   int rank; //ранг ребра
   void REPLACE();
   GraphEdge(){
      };
};

class Graph{
public:
   GraphEdge *E; //Набор ребер графа
   int EdgeNum; //Число ребер графа
   Graph(int N){ //Конструктор графа из N ребер
      EdgeNum=N+1;
      E=new GraphEdge[N+1];
   };
   Graph(){};
   void WriteData(int *ATrail); //Запись ответа
   int *FindATrail(int v0); //Поиск A-цепи
   int Deg(int vertex);//Определение степени вершины
};
```

Приведем текст функции FindATrail, реализующей поиск AOE-цепи согласно заданным данным. Функция реализует работу этапа «ПОСТРОЕНИЕ» алгоритма AOE-TRAIL [36].

```
int *Graph::FindATrail(int v0){
   //NextEdge - текущее ребро, curV - ее конец
   //Выделим память под массив номеров ребер результирующей цепи
   int *ATrail=new int [EdgeNum];
   //Инициализируем элементы массива
   for (int i=1;i<=EdgeNum;i++) ATrail[i]=INT_MAX;
   int m=0;
   //В качестве текущей выберем первую вершину
   int curV=v0;
   //Считаем, что следующее ребро не определено
   int NextEdge=INT_MAX;
   //k -- счетчик ребер в построенной A-цепи
   int k=1;
   //Отыщем первое ребро и вершину, в которую оно приводит.
   //В силу 4-регулярности графа, это ребро будет иметь ранг 2
   //либо 1 (если начальной выбрана вершина степени 2)
   //В дальнейшем все ребра нумеруются с 1
   for (int i=1;i<=EdgeNum;i++){
      if ((E[i].v1==curV||E[i].v2==curV)&&(E[i].rank==2||Deg(curV)==2)){
         NextEdge=i;
         ATrail[k]=NextEdge;
         //В качестве текущей должна быть задана v1 ребра.
         //Если это не так, меняем индексы местами
         if (E[i].v1!=curV) E[i].REPLACE();
         curV=E[i].v2;
         break;
         }
   }//for
   //Цикл для построения A-цепи
   do{
      k++;
      //Если ребро задано наоборот, поменять индексы местами
      if (E[NextEdge].v1!=curV)
         E[NextEdge].REPLACE();
      //Ищем самое вложенное непройденное ребро, делаем его текущим,
      //и находим его концевую вершину с учетом возможности задания
      //ребра наоборот
      //Попав в тупик, выводим сообщение об отсутствии решения
      if (E[E[NextEdge].l1].rank > E[E[NextEdge].r1].rank){
         if (!Included(ATrail,EdgeNum,E[NextEdge].l1)){
            NextEdge=E[NextEdge].l1;
            if (curV==E[NextEdge].v2) curV=E[NextEdge].v1;
               else curV=E[NextEdge].v2;
         }
         else{
            if (!Included(ATrail,EdgeNum,E[NextEdge].r1)){
               NextEdge=E[NextEdge].r1;
               if (curV==E[NextEdge].v1) curV=E[NextEdge].v2;
                  else curV=E[NextEdge].v1;
            }else{
               cout<<"There is no OE-A-trail!\n"; break;
            }
         }
```

```
      }else{
         if (!Included(ATrail,EdgeNum,E[NextEdge].r1)) {
            NextEdge=E[NextEdge].r1;
            if (curV==E[NextEdge].v2) curV=E[NextEdge].v1;
               else curV=E[NextEdge].v2;
         }
         else{
            if (!Included(ATrail,EdgeNum,E[NextEdge].l1)){
               NextEdge=E[NextEdge].l1;
               if (curV==E[NextEdge].v1) curV=E[NextEdge].v2;
                  else curV=E[NextEdge].v1;
            }else{
               cout<<"There is no OE-A-trail!\n"; break;
            }
         }
      }
      //Заносим найденное ребро в результирующий массив
      ATrail[k]=NextEdge;
   }while (k!=EdgeNum-1); //Продолжаем цикл до тех пор,
                          //пока не перебрали все ребра
   return ATrail; //Возвращаем результирующий массив
}
//*******************************************
bool Included (int *ATrail, int N, int K){
   //Проверка, пройдено ли ребро. Если ребро пройдено,
   //то оно находится в результирующем массиве
   for (int i=1;i<=N;i++)
      if (ATrail[i]==K) return true;
   return false;
}
```

Разработанная программа в зависимости от номера начальной вершины определяет AOE-цепь в графе, либо выводит сообщение, что задача не имеет решения. Для рассмотренного примера при указании вершины с номером 1 в качестве начальной программа находит цепь:

$$2-13-14-15-16-12-11-9-87-6-5-3-4-7-20-1.$$

В перспективе предполагается разработать графическую оболочку, позволяющую демонстрировать полученную цепь в динамике.

Выводы по главе 5

1. С помощью разработанного программного обеспечения возможно протестировать работу всех рассмотренных в разделах 3 и 4 алгоритмов. Программное обеспечение позволяет вводить/выводить данные в тек-

стовом и графическом формате, просматривать результат работы алгоритмов в динамике (анимация обхода).

2. С помощью программной реализацици алгоритма `OrderedEnclosingTest` возможно провести верификацию результатов работы программного обеспечения. Алгоритм может использоваться для повышения надежности разработанных программ.

ГЛАВА 6
ПРИМЕНЕНИЕ АЛГОРИТМОВ МАРШРУТИЗАЦИИ В САПР ТЕХНОЛОГИЧЕСКОЙ ПОДГОТОВКИ ПРОЦЕССОВ РАСКРОЯ

В настоящее время применение ресурсосберегающих технологий является актуальным. На количество отходов, образующихся в процессе раскроя, влияют:

- технологические допуски на кромку;
- резы и перемычки между отдельными заготовками;
- сочетание конфигураций взаимно прилегающих заготовок;
- некратность размеров заготовки и размеров материала (особенно ощутим при раскрое крупных заготовок).

Меры борьбы за уменьшение потерь при раскрое:

- утилизация отходов;
- ужесточение технологических допусков;
- совмещение резов;
- сокращение времени холостых переходов при вырезании.

В 1949 г. за рубежом появились первые публикации по линейному программированию. В 1951 г. вышло первое издание монографии [19], в которой впервые рассмотрены вопросы применения линейного программирования для оптимального гильотинного раскроя (т.е. построения раскройного плана с определением последовательности сквозных резов на гильотине).

Развитие автоматизации производства привело к появлению технологического оборудования с числовым программным управлением (ЧПУ), используемого для резки листовых материалов: машин газовой (кислородной), плазменной, лазерной и электроэрозионной резки материала. Новые технологии

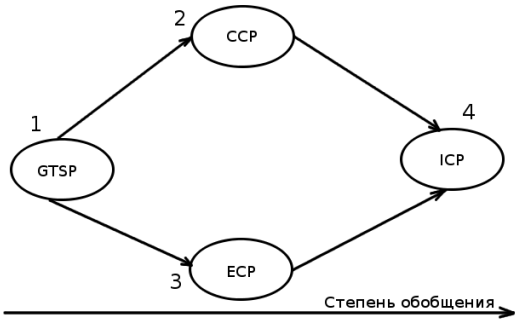


Рисунок 6.1: Классификация задач вырезания деталей

позволяют осуществлять вырезание по произвольной траектории с достаточной для практики точностью. Снятие требования резки только сквозными прямолинейными резами позволяет существенно снизить отходы материала. В связи с этим появилось множество публикаций, например, [41], [40], [20], [116], посвященных вопросам негильотинного раскроя и его оптимизации в различных производствах и на разных уровнях автоматизации.

В отличие от гильотинного раскроя, негильотинный раскройный план не дает программу вырезания деталей. Построение программы управления раскройным автоматом для реализации заданного раскройного плана является самостоятельной задачей. В работе [114] предложена следующая классификация задач маршрутизации инструмента машин листовой резки (рисунок 6.1):

1. **Обобщенная задача коммивояжера (GTSP)** (Generalized Travelling Salesman Problem): режущий инструмент последовательно проходит по контуру каждой детали. Возможные точки врезки в каждый контур заданы. Данная технология не допускает совмещения фрагментов контуров вырезаемых деталей. Оптимальным маршрутом является решение обобщенной задачи коммивояжера на множестве точек врезки с ограничениями предшествования, учитывающими вложенность одних контуров во внутренность других (рисунок 6.2.а).
2. **Задача последовательной резки (CCP)** (Continuous Cutting Problem): детали вырезаются последовательно. Точка врезки может на-

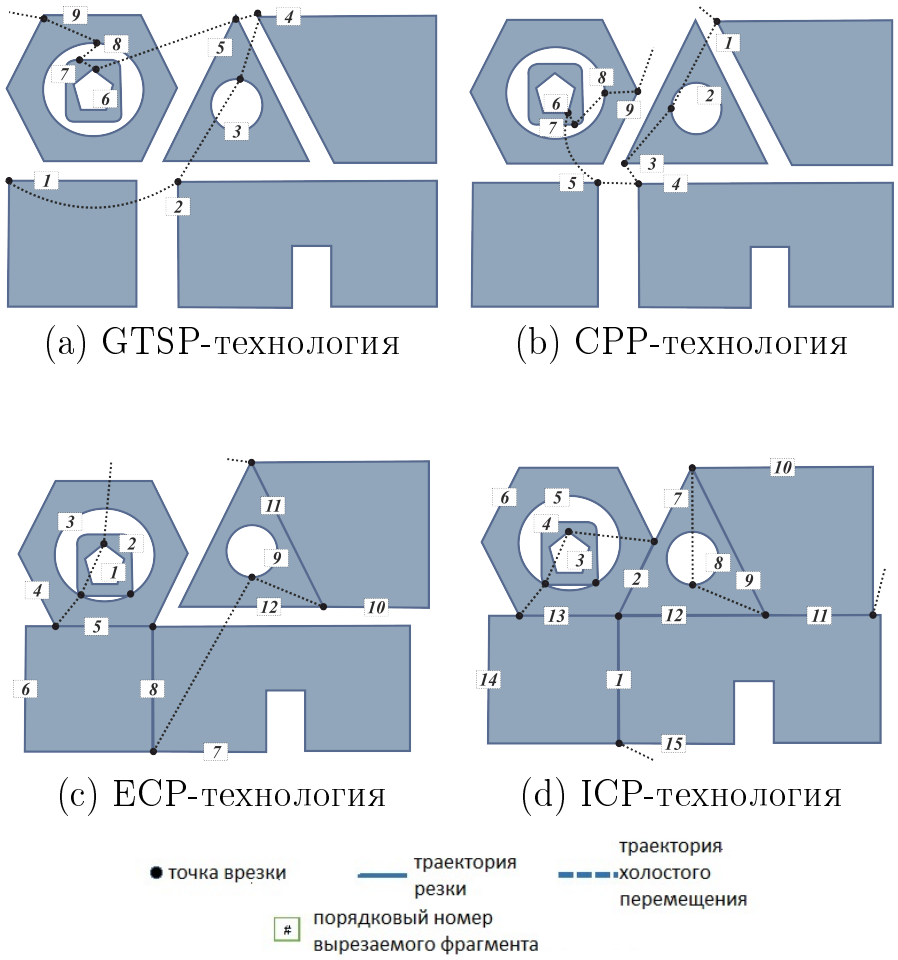


Рисунок 6.2: Примеры раскройных планов с решением задачи маршрутизации

ходиться в любой части контура, переход к другому контуру осуществляется только после окончания вырезания текущего. Данная технология, как и GTSP, не допускает совмещения фрагментов контуров вырезаемых деталей. Оптимальным маршрутом является решение обобщенной задачи коммивояжера на множестве выбранных алгоритмом точек врезки с ограничениями предшествования, учитывающими вложенность одних контуров во внутренность других (рисунок 6.2.b).

3. **Задача с фиксированными точками врезки (ECP)** (Endpoint Cutting Problem): инструмент осуществляет врезку и переходит к другому фрагменту раскройного плана в заданных точках на границе. Допускается совмещение контуров вырезаемых деталей, что приводит к вырезанию контура отдельных деталей по частям (рисунок 6.2.c).

4. **Задача прерывистого раскроя (ICP)** (Intermittent Cutting Problem): общий случай задачи раскроя, когда допускается совмещение контуров вырезаемых деталей, и нет ограничений на выбор точек врезки (рисунок 6.2.d).

Технологии GTSP и CCP различаются размерами и значениями элементов матриц расстояний между контурами. Данные технологии предполагают только оптимизацию холостых перемещений, а программы вырезания контуров транслируются с точностью до точки врезки. При этом длина резки равна сумме длин вырезаемых контуров, а также необходим зазор между вырезаемыми контурами, что приводит к дополнительному расходу материала. Технология CCP в сравнении с технологией GTSP позволяет сократить только время на холостые проходы между точками врезки. Имеется множество эвристических алгоритмов построения маршрутов для данных технологий, например [159]–[12]. В работе [94] рассмотрены способы повышения эффективности точного алгоритма построения маршрутов для данных технологий.

Технологии ECP и ICP за счет возможности совмещения границ вырезаемых деталей позволяют сократить расход материала, длину резки, количество и длину холостых проходов (см., например, рисунок 6.2). Однако, это существенно усложняет процесс составления программы вырезания: (1) последовательные фрагменты контуров детали не всегда являются последовательными элементами траектории режущего инструмента, (2) нетривиальной становится проблема нахождения такой последовательности вырезания фрагментов, чтобы отрезанная от листа часть не требовала дополнительных разрезаний. В работе [121] сделана попытка оценить эффект от совмещения фрагментов границ вырезаемых деталей, в этой же работе констатируется отсутствие эффективных алгоритмов нахождения маршрутов резки при использовании технологий ECP и ICP, в частности отмечается, что предложенный в работе [120] подход, требующий решения задачи сельских почтальонов, является трудно реализуемым. Применение технологий ECP и ICP в системе

технологической подготовки процессов раскроя плоских деталей предполагает следующие этапы:

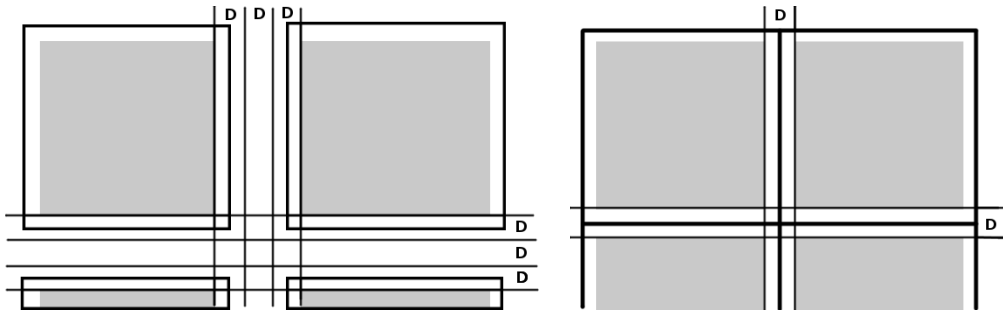
1. **Составление раскройного плана,** заключающееся в нахождении такого варианта размещения вырезаемых деталей на прямоугольном листе, при котором минимизируются отходы и максимизируется длина совмещенных элементов контуров вырезаемых деталей. Решению данной задачи отражено в публикациях [41], [40], [20], [116].
2. **Абстрагирование раскройного плана до плоского графа.** Для определения последовательности резки фрагментов раскройного плана не используется информация о форме детали, поэтому все кривые без самопересечений и соприкосновений на плоскости, представляющие форму деталей, интерпретируются в виде ребер графа, а все точки пересечений и соприкосновений представляются в виде вершин графа. Для анализа выполнения технологических ограничений необходимо введение дополнительных функций на множестве вершин, граней и ребер полученного графа.
3. **Решение задачи построения оптимальных маршрутов** с ограничениями, наложенными на порядок обхода ребер. Данные ограничения непосредственно вытекают из технологических ограничений, наложенных на порядок вырезания деталей: отрезанная от листа часть не должна требовать дополнительных разрезаний, должны отсутствовать пересечения резов, необходимо оптимизировать длину холостых переходов, минимизировать количество точек врезки [11] и т.д.
4. **Составление программы управления** процессом раскроя на основе маршрута, найденного с помощью алгоритма решения абстрагированной задачи маршрутизации. Здесь выполняется обратная замена абстрактных ребер плоского графа системой команд раскройному автомату, обеспечивающей движение по кривым на плоскости, соответствующим форме вырезаемой детали.

Этапы построения раскройного плана и интерпретации найденного маршрута в терминах команд раскройному автомату являются общими для всех технологий и достаточно известны. На международной конференции CAD/-CAM/PDM – 2015 [26, 27] рассмотрены особенности реализации второго и третьего этапов для ресурсосберегающих технологий ECP и ICP. Рассмотрим подробнее применение результатов работы, описанных в предыдущих главах, для решения возникающих проблем [128].

6.1 Особенности и различия составления раскройных планов для различных технологий

Рассмотрим более подробно особенности и различия составления раскройного плана для различных технологий.

Составление раскройного плана для технологии GTSP предполагает поконтурное вырезание деталей, поэтому если D – ширина реза, то при отсутствии совмещения резов детали должны находиться на расстоянии не менее $3D$ (рисунок 6.3.a)). Напротив, при совмещении резов реальные границы деталей должны находиться на расстоянии D (рисунок 6.3.b)).



(a) Технология GTSP (b) Технология ICP

Рисунок 6.3: Определение нименьшего расстояния между деталями (a) без совмещения резов и (b) при наличии совмещения

Рассмотрим экстремальный случай, когда требуется разместить $m{\cdot}n$ квадратных заготовок размера s [131]. На рисунке 6.4 приведена оптимальная упаковка таких заготовок.

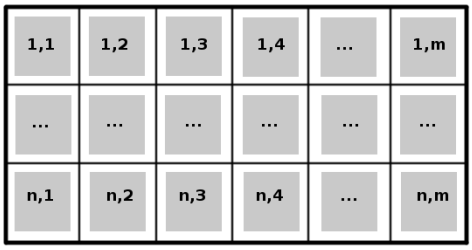


Рисунок 6.4: Раскройный план, состоящий из $m \cdot n$ одинаковых квадратных заготовок

В этом случае при отсутствии совмещения резов **непродуктивный расход материала** составит величину

$$3D(n-1) + 3D(m-1) = 3D(n+m-2).$$

При совмещении резов непродуктивный расход составит величину

$$D(n-1) + D(m-1) = D(n+m-2).$$

Таким образом, непродуктивный расход материала за счет совмещения резов может быть сокращен в данном случае в 3 раза.

Рассчитаем **длину реза** L без учета холостых перемещений и ширины реза [131]. При отсутствии совмещения резов будет равна произведению периметра прямоугольника на общее число таких прямоугольников

$$L \geq (4 \cdot s) \cdot (m \cdot n).$$

При совмещении резов получим величину

$L = (m-1) \cdot (n-1) \cdot 2 \cdot s + 2 \cdot m + 2 \cdot n = 2 \cdot s \cdot n \cdot m + 2 \cdot s = 2 \cdot s \cdot (m \cdot n + 1)$.

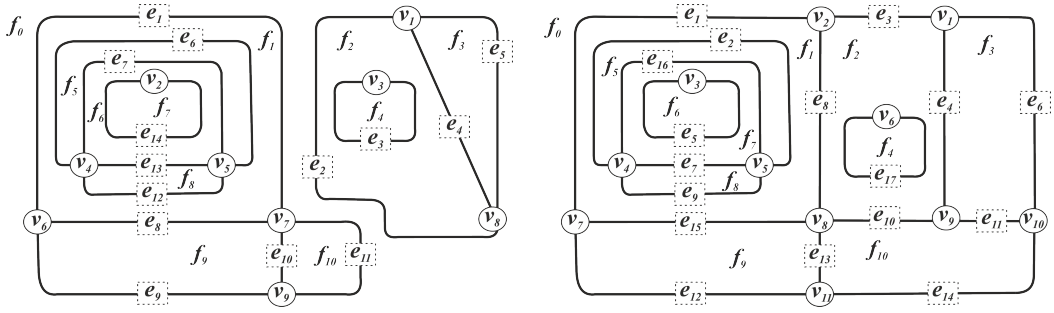
То есть длина реза при совмещении может быть сокращена почти в два раза.

При отсутствии совмещения резов **количество точек врезки** $|V_{odd}| = m \cdot n$, т.е. совпадает с числом прямоугольников. При совмещении резов все точки врезки (являющиеся в гомеоморфном образе раскройного плана вершинами нечетной степени) находятся на внешней границе раскройного плана и на каждой из четырех границ таких точек на единицу меньше числа прямоугольников в ряду (столбце), то есть $|V_{odd}| = 2 \cdot (n-1) + 2 \cdot (m-1)$. Следовательно, при совмещении резов количество точек врезки на порядок меньше, нежели при отсутствии совмещения.

Таким образом, показано, что технология ICP является ресурсосберегающей по таким важным критериям как непродуктивный расход материала, длина горячей резки и количество точек врезки, – по отношению к технологии GTSP, используемой современными CAD/CAM системами.

6.2 Абстрагирование раскройного плана до плоского графа

Моделью раскройного листа будем считать плоскость S, моделью раскройного плана – плоский граф [18] G с внешней гранью f_0 на плоскости S. Множество вершин компонент связности графа G негомеоморфных окружности будем считать точки соприкосновения трех и более фрагментов раскройного плана, а соответствующие фрагменты ребрами, инцидентными данной вершине. Компоненту связности гомеоморфную окружности будем считать петлей. На рисунке 6.5 представлены плоские графы, являющиеся гомеоморфными образами ESP и ICP раскройных планов, приведенных на рисунке 6.2. Для любой части графа $J \subseteq G$ обозначим через $\mathsf{Int}\,(J)$ теоретико-



(a) ECP-граф (b) ICP-граф

Рисунок 6.5: Примеры абстрагирования раскройных планов до плоских графов

множественное объединение его внутренних граней (объединение всех связных компонент $S \setminus J$, не содержащих внешней грани). Если считать, что режущий инструмент прошел по всем фрагментам графа J, то $\mathrm{Int}(J)$ можно интерпретировать как отрезанную от листа часть. Множества вершин, ребер

и граней графа J будем обозначать через $V(J)$, $E(J)$ и $F(J)$ соответственно, а через $|M|$ – число элементов множества M.

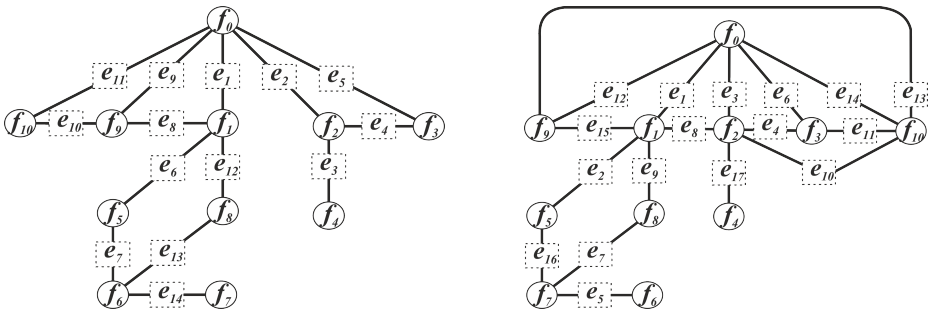
Для представления образа раскройного плана в виде плоского графа $G = (V, F, E)$ определим для каждого ребра $e \in E(G)$ функции, введенные в разделе 1 (рисунок 1.1). В таблице 6.2 дано представление графов, изображенных на рисунке 6.5.

Таким образом, используя известные координаты прообразов вершин графа $G = (V, F, E)$ и размещения фрагментов раскройного плана, являющихся прообразами ребер графа $G = (V, F, E)$, любой маршрут в графе $G = (V, E)$ можно интерпретировать как траекторию режущего инструмента.

6.3 Ранжирование ребер плоского графа

Ранжирование заключается в присваивании рангов всем ребрам графа.

Ранг ребра определяет его удаленность от внешней грани и показывает, какое минимальное число граней необходимо пересечь, чтобы добраться от внешней грани f_0 до этого ребра. Это позволяет для определения ранга использовать граф $G'(F, V, E)$, топологически двойственный исходному графу $G(V, F, E)$: множеством вершин графа G' является множество F граней графа G, а ребрам графа G' соответствует наличие между двумя гранями границы ненулевой длины, т.е. ребра $e \in E(G)$. Изображение графов, двойственных представленным в таблице 6.2, приведено на рисунке 6.6 [25].



(a) двойственный ECP-граф (b) двойственный ICP-граф

Рисунок 6.6: Двойственные графы

Таблица 6.1: Примеры педставления графов

	Представление ECP-графа									Представление ICP-графа								
e	$v_1(e)$	$v_2(e)$	$f_1(e)$	$f_2(e)$	$l_1(e)$	$l_2(e)$	$r_1(e)$	$r_2(e)$	rank	$v_1(e)$	$v_2(e)$	$f_1(e)$	$f_2(e)$	$l_1(e)$	$l_2(e)$	$r_1(e)$	$r_2(e)$	rank
e_1	v_6	v_7	f_1	f_0	e_9	e_8	e_8	$e_1 1$	1	v_2	v_7	f_0	f_1	e_8	e_{12}	e_3	e_{15}	1
e_2	v_1	v_8	f_0	f_2	e_4	e_5	e_5	e_4	1	v_4	v_5	f_5	f_1	e_9	e_{16}	e_{16}	e_9	2
e_3	v_3	v_3	f_2	f_4	e_3	e_3	e_3	e_3	2	v_2	v_1	f_2	f_0	e_1	e_4	e_8	e_6	1
e_4	v_8	v_1	f_3	f_2	e_2	e_5	e_5	e_2	2	v_1	v_9	f_2	f_3	e_6	e_{10}	e_3	e_{11}	2
e_5	v_1	v_8	f_3	f_0	e_2	e_4	e_4	e_2	1	v_3	v_3	f_6	f_7	e_5	e_5	e_5	e_5	4
e_6	v_4	v_5	f_5	f_1	e_{12}	e_7	e_7	e_{12}	2	v_{10}	v_1	f_0	f_0	e_{11}	e_3	e_{14}	e_4	1
e_7	v_5	v_4	f_5	f_6	e_{13}	e_6	e_6	e_{13}	3	v_4	v_5	f_8	f_7	e_{16}	e_9	e_9	e_{16}	3
e_8	v_7	v_6	f_1	f_9	e_{10}	e_1	e_1	e_9	2	v_8	v_2	f_2	f_1	e_{15}	e_3	e_{10}	e_1	2
e_9	v_6	v_9	f_0	f_9	e_8	e_{11}	e_1	e_{10}	1	v_4	v_5	f_1	f_8	e_7	e_1	e_2	e_7	2
e_{10}	v_9	v_7	f_{10}	f_9	e_9	e_{11}	e_{11}	e_8	2	v_8	v_9	f_{10}	f_2	e_8	e_{11}	e_{13}	e_4	2
e_{11}	v_9	v_7	f_0	f_{10}	e_{10}	e_1	e_9	e_{10}	1	v_9	v_{10}	f_{10}	f_3	e_4	e_{14}	e_{10}	e_6	2
e_{12}	v_4	v_5	f_1	f_8	e_{13}	e_6	e_6	e_{13}	2	v_{11}	v_7	f_9	f_0	e_{14}	e_{15}	e_{13}	e_1	1
e_{13}	v_4	v_5	f_8	f_6	e_7	e_{12}	e_{12}	e_7	3	v_8	v_{11}	f_9	f_{10}	e_{10}	e_{12}	e_{15}	e_{14}	2
e_{14}	v_2	v_2	f_6	f_7	e_{14}	e_{14}	e_{14}	e_{14}	4	v_{11}	v_{10}	f_0	f_{10}	e_{13}	e_6	e_{12}	e_{11}	1
e_{15}	—	—	—	—	—	—	—	—	—	v_7	v_8	f_9	f_1	e_1	e_{13}	e_{12}	e_8	2
e_{16}	—	—	—	—	—	—	—	—	—	v_4	v_5	f_7	f_5	e_2	e_7	e_7	e_2	3
e_{17}	—	—	—	—	—	—	—	—	—	v_6	v_6	f_2	f_4	e_{17}	e_{17}	e_{17}	e_{17}	2

Для всех $f \in F(G)$ расстояние в графе G' между f и внешней гранью f_0 можно определить, построив в графе G' дерево $T_{G'}^{f_0}$ кратчайших путей до вершины $f_0 \in F$. Наличие в представлении графа G функций $l_k : E(G) \to E(G),\ k = 1, 2$ позволяет найти функции ранга за время не превосходящее величины $O(|E| \log_2 |V|)$ [137]. В таблице 6.2 приведены ранги ребер графов, изображенных на рисунке 6.5.

6.4 Добавление дополнительных ребер и построение маршрута

Добавление дополнительных ребер подразумевает построение кратчайшего паросочетания M на множестве вершин нечетной степени. Под длиной ребра при поиске дополнительных построений (холостых переходов) будем понимать кратчайший путь по прямой между двумя точками на плоскости (соответствующими точкам врезки на раскройном плане). Для графов, соответствующих технологиям ICP и ECP, такие переходы показаны на рисунке 6.2 пунктирными линиями.

Для **построения маршрута** можно воспользоваться любым, приемлемым для рассматриваемого связного графа, алгоритмом построения OE-маршрута. Построенные маршруты приведены на рисунке 6.2.

6.5 Задача прямоугольного раскроя и OE-покрытия

Рассмотрим задачу прямоугольного раскроя на полубесконечной полосе. Особенностью этой задачи является то, что плоский граф, соответствующий раскройному плану, содержит только вершины степеней 2, 3 или 4. Каждой вершиной такого графа является точка соприкосновения (возможно, малая окрестность) нескольких деталей на плоскости [73].

Для такого графа возможно применение как любого из алгоритмов построения OE-покрытия в плоском графе [68], так и алгоритма построения AOE-покрытия [32].

Схема кодирования графа для решения задачи раскроя-упаковки не совпадает со схемой кодирования графа для задачи построения OE-покрытия, в которой осуществляется поиск траектории движения режущего инструмента с определенным ограничением.

Кодирование раскройного плана предполагает указание для каждого прямоугольника декартовых координат его верхней левой и нижней правой вершин. Примером кодировния прямоугольной укладки, представленной на рисунке 6.7 является матрица

$$A = \left(\begin{array}{c|cccc} \text{№ прямоугольника} & x_{\text{верх}} & y_{\text{верх}} & x_{\text{нижн}} & y_{\text{нижн}} \\ \hline 1 & 0 & 0 & 3 & 2 \\ 2 & 1 & 2 & 2 & 4 \\ 3 & 3 & 1 & 5 & 4 \\ 4 & 0 & 4 & 4 & 6 \end{array}\right).$$

В общем случае количество строк матрицы A равно числу прямоугольников.

Рассмотрим алгоритм преобразования раскройного плана в граф и определения функций для каждого ребра на примере укладки, приведенной на рисунке 6.7.

Заметим, что функции $f_1(e)$ и $f_2(e)$ определяются автоматически с помощью программы, разработанной для построения OE-покрытий [81], поэтому достаточно определить только функции $l_k(e)$, $k = 1, 2$ и $v_k(e)$, $k = 1, 2$.

Очевидно, что вершинами графа с учетом кратности будут все вершины прямоугольников и только они. Вертикальные и горизонтальные отрезки, соединяющие пары соседних вершин, являются ребрами графа. Можно уменьшить число ребер, удалив вершины степени два и объединив инцидентные

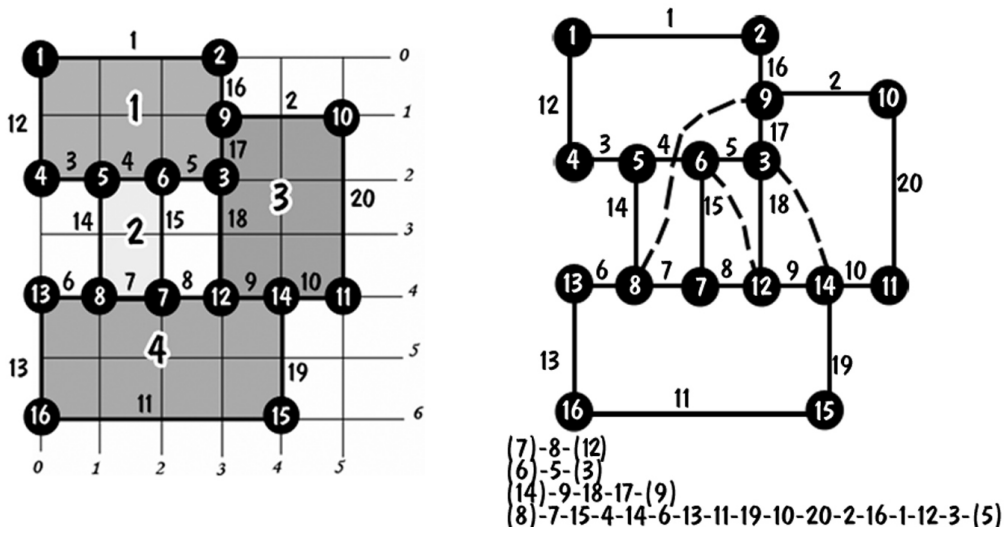


Рисунок 6.7: (а) Пример прямоугольной укладки. (б) Пример покрытия раскройного плана, состоящего из прямоугольников, OE-цепями

этим вершинам ребра. Однако полученный гомеоморфный граф представляет лишь теоретический интерес.

6.5.1 Алгоритмы перекодирования раскройного плана

Алгоритм перекодирования данных о прямоугольном раскройном плане в кодировку для ребер графа, принятую в $[68, 75, 142, 143, 145]$, можно представить следующим образом.

Алгоритм `RECODE`

Входные данные: раскройный план, представленный в виде матрицы A.

Выходные данные: представление графа с помощью функций $v_k(e)$, $l_k(e)$, $k = 1, 2$.

Шаг 1. Найти максимальные значения координат раскройного плана $x_{max} = \max\{x_{3,i} : \ i = 1, \ldots, N_A\}$ и $y_{max} = \max\{y_{4,j} : \ j = 1, \ldots, N_A\}$.

Шаг 2. Для каждого $y_k \in [0, y_{max}]$ найти все горизонтальные ребра графа. Для этого просмотреть все узлы сетки от $(0, y_k)$ до (x_{max}, y_k). Если отрезок $[(x_i, y_k), (x_j, y_k)]$, $i < j$ принадлежит границе хотя бы одного из прямоугольников, то $[(x_i, y_k), (x_j, y_k)]$ является ребром графа. Положить $v_1(e) = (x_i, y_k)$, $v_2(e) = (x_j, y_k)$.

Шаг 3. Для каждого $x_k \in [0, x_{max}]$ найти все вертикальные ребра графа: просмотреть все узлы сетки от $(x_k, 0)$ до (x_k, y_{max}) и найти отрезки $[(x_k, y_i), (x_k, y_j)]$, $i < j$, принадлежащие границе хотя бы одного из прямоугольников. Тогда $[(x_k, y_i), (x_k, y_j)]$ является ребром графа. Положить $v_1(e) = (x_k, y_i)$, $v_2(e) = (x_k, y_j)$.

Шаг 4. Для каждого ребра e выполнить следующие действия.

Шаг 4.1. Перейти в вершину $v_1(e)$. Среди ребер, инцидентных этой вершине, выбрать ребро $\tilde{e}_1$, образующее с ребром e минимальный угол (угол между ребрами отсчитывается как на рисунке 4.12, т.е. против часовой стрелки). Положить $l_1(e) = \tilde{e}_1$.

Шаг 4.2. Перейти в вершину $v_2(e)$. Среди ребер, инцидентных этой вершине, выбрать ребро $\tilde{e}_2$, образующее с ребром e минимальный угол. Положить $l_2(e) = \tilde{e}_2$.

Шаг 5. Конец алгоритма.

Для нахождения покрытия соответствующего раскройному плану плоского графа OE-цепями необходимо загрузить закодированный файл в программу «Eulerian Cover Constructor» [81]. Эта программа позволяет найти покрытие OE-цепями для любого плоского графа без висячих вершин. В частности, для укладки, представленной на рисунке 6.7(а), будет найдена последовательность цепей, приведенная на рисунке 6.7(б). В скобках указаны вершины, из которых начинается построение цепи и в которых оно завершается. Первые три цепи соединяют вершины нечетной степени, четвертая же является эйлеровой цепью суграфа, содержащего все оставшиеся ребра.

Модифицируем алгоритм `RECODE` таким образом, чтобы с его помощью можно было находить кодировку заданного графа в терминах функций $v_k(e)$, $l_k(e)$, $f_k(e)$, $k = 1, 2$;

Для решения задачи выделим все прямые, содержащие стороны прямоугольников. Для каждой такой прямой рассмотрим отрезки (стороны прямоугольников), лежащие на ней. Отсортируем полученные отрезки по возрас-

танию координаты начала отрезка. Для решения задачи достаточно рассмотреть отрезки на текущей прямой, их области пересечения будут соответствовать ребрам, не смежным внешней грани, а остальные – ребрам, смежным внешней грани f_0. Заметим, что, благодаря специфике задачи (раскройный план состоит из прямоугольных деталей), пересекаться могут не более двух отрезков. Кроме того пересекающиеся отрезки будут соседними элементами в списке, так как отрезки отсортированы.

Допустим, что отрезки, находящиеся на одной прямой, отсортированы по возрастанию координаты начала отрезка. Список отрезков представлен в виде очереди M длиной n. Текущая точка pm – координата, которую алгоритм рассматривает в данный момент. Начало отрезка i будем обозначать $M_{i,0}$, а конец – $M_{i,1}$. Описанный выше процесс можно обобщить и представить в качестве алгоритма `STRAIGHT LINE` [54], осуществляющего проход вдоль прямой.

Алгоритм `STRAIGHT LINE`

Входные данные: массив M, содержащий отрезки, находящиеся на одной прямой.

Выходные данные: список ребер, смежных внешней грани.

Шаг 1. Перейти в начало отсчета: $i = 0$, $j = 0$, текущая рассматриваемая точка $pm = M[0]$.

Шаг 2. Если рассмотрены не все отрезки ($j < n$) перейти к **шагу 3**, иначе – к **шагу 6**.

Шаг 3. Рассмотреть отрезки $M_{i,k}$ и $M_{j,l}$ $(k, l = 0, 1)$. Если $M_{i,1} > M_{j,1}$, а $pm \neq M_{j,0}$, то отрезок $[pm; M_{j,0}]$ смежен внешней грани, а ребро $[M_{j,0}; M_{j,1}]$ — нет. Если $M_{i,1} \neq M_{j,1}$, то переместить указатель на текущую координату pm в $M_{j,1}$ и положить $j = j + 1$, в противном случае $i = j + 1$, $j = j + 2$, а в качестве pm будет рассматриваться $M_{i,0}$. Перейти к **шагу 2**.

Шаг 4. Если $M_{i,1} < M_{j,0}$, то отрезок $[pm; M_{i,1}]$ смежен внешней грани. Переместить указатель pm в точку $M_{j,0}$. Положить $i = j$, $j = j + 1$. Перейти к **шагу 2**.

Шаг 5. Если $M_{i,1} < M_{j,1}$, то в случае, когда рт $\neq M_{j,0}$ отрезок $[pm; M_{j,0}]$ смежен внешней грани, а когда $M_{j,0} \neq M_{i,1}$, то отрезок $[M_{j,0}; M_{i,1}]$ не смежен внешней грани. Положить рт $= M_{i,1}$, $i = j$, $j = j + 1$ и перейти к **шагу 2**.

Шаг 6. Останов.

Сложность алгоритма `STRAIGHT LINE` составляет величину $O(|V|)$. Используется только один цикл и в каждой итерации параметр цикла увеличивается хотя бы на 1.

Приведем алгоритм `RECODE2` [54], осуществляющий представление множества графов в рассмотренной выше кодировке.

Алгоритм `RECODE2`

Входные данные: n раскройных планов, представленных в виде матрицы A_i, $i = 1, \ldots, n$.

Выходные данные: представление графов G_i, $i = 1, \ldots, n$ с помощью функций $v_k(e)$, $l_k(e)$, $k = 1, 2$.

Вертикальные прямые

Шаг 1. Выделить все вертикальные прямые V_i, $i = 1, \ldots, n$ (сложность данного шага составляет величину $O(|V|)$).

Шаг 2. Удалить повторения (сложность – $O(|V| \log_2 |V|)$).

Шаг 3. Отсортировать прямоугольники по возрастанию координаты Y (сложность – $O(|V| \log_2 |V|)$). Для каждой вертикальной прямой выполнить следующие действия.

Шаг 3.1.Выбрать отрезки фигуры принадлежащие этой прямой (сложность – $O(|V|)$).

Шаг 3.2. Применить алгоритм `STRAIGHT LINE` прохода вдоль прямой для вертикальных отрезков (сложность – $O(|V|)$).

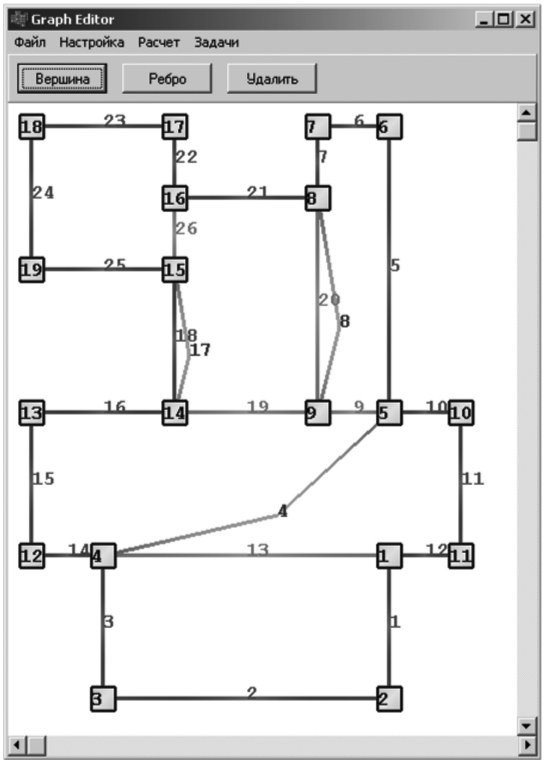


Рисунок 6.8: Решение задачи построения маршрута для прямоугольного раскройного плана с помощью разработанного программного обеспечения

Горизонтальные прямые

Шаг 4. Выделить все горизонтальные прямые H_j, $j = 1, \ldots, n$ (сложность – $O(|V|)$).

Шаг 5. Удалить повторения (сложность $O(|V| \log_2 |V|)$).

Шаг 6. Отсортировать прямоугольники по возрастанию координаты X (сложность – $O(|V| \log_2 |V|)$). Для каждой горизонтальной прямой выполнить следующие действия.

Шаг 6.1.Выбрать отрезки фигуры принадлежащие этой прямой (сложность – $O(|V|)$).

Шаг 6.2. Применить алгоритм `STRAIGHT LINE` прохода вдоль прямой для отрезков, принадлежащих H_j (сложность – $O(|V|)$).

Конец алгоритма.

Легко видеть, что общая сложность алгоритма $O(|V| \log_2 |V|)$. Для нахождения покрытия полученного плоского графа OE-цепями необходимо загрузить закодированный файл в программу «Eulerian Cover Constructor» [81].

Эта программа позволяет найти покрытие OE-цепями для любого плоского графа без висячих вершин.

6.5.2 Выбор оптимальной укладки деталей

Полученную с помощью рассмотренного алгоритма кодировку можно использовать при решении следующей задачи оптимизации [62]. Предположим, что для некоторого набора прямоугольных деталей известно несколько оптимальных упаковок. Требуется найти множество упаковок, для которых покрытие OE-цепями было бы оптимальным.

Рассмотрим возможные критерии оптимальности. Стоимость раскроя зависит в основном от трех факторов: длины пути холостого хода, длины пути реза и количества холостых проходов (т.е. точек врезки) [63]. Перечисленные параметры не являются независимыми.

Количество холостых проходов (число точек врезки). Задача их минимизации тривиальна. В данном случае необходимо рассмотреть все имеющиеся в качестве исходных данных упаковки и выбрать те, для которых в соответствующем им плоском графе число вершин нечетной степени будет минимально, и построить для них покрытия OE-цепями. Такая задача решается за линейное время.

Суммарная длина пути реза. Эта задача также может быть решена за линейное время еще на этапе кодирования графа.

Длина пути холостого хода. Эта задача не так тривиальна. В частности, алгоритм для задачи китайского почтальона не сможет решить эту задачу, т.к. в данном случае при построении маршрута уровень вложенности каждой вершины определяет допустимый порядок обхода. С целью уменьшения длины маршрута можно рекомендовать идти в ближайшую непомеченную вершину $v \in V_{odd}$ с максимальным рангом (т.е. жадный алгоритм). Вычислительный эксперимент показывает, что построенный таким образом маршрут

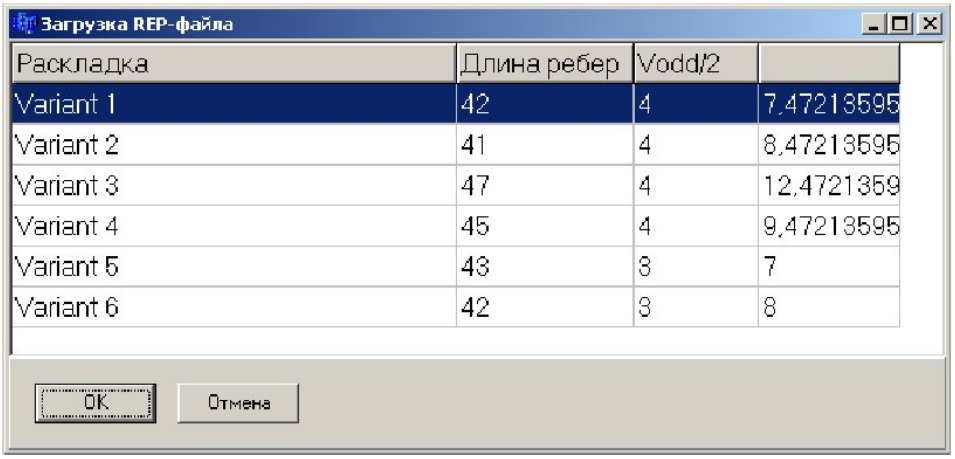


Рисунок 6.9: Найденные оптимальные решения

имеет длину не больше маршрута найденного с помощью алгоритма [135], когда очередная вершина $v \in V_{odd}$ для врезки выбиралась лексикографически. Результаты тестирования алгоритма для раскройного плана, представленного на рисунке 6.8, представлены на рисунке 6.9.

Выводы по главе 6

1. Рассмотренные алгоритмы могут быть применены в проектировании программ вырезания деталей с использованием ресурсосберегающих ECP и ICP технологий.
2. В случае прямоугольного раскройного плана возможна эффективная инкапсуляция программ построения AOE-маршрутов для 4-регулярных графов и OE-маршрутов в систему технологической подготовки процессов прямоугольного раскроя. Это позволяют осуществить разработанные алгоритмы `RECODE` и `RECODE2` перекодировки раскройных планов.

ВЫВОДЫ И ОСНОВНЫЕ РЕЗУЛЬТАТЫ

1. Введен класс маршрутов с упорядоченным охватыванием (OE-маршрутов). В общем случае такие маршруты представляют покрытие графа упорядоченной последовательностью цепей. Доказаны теоремы существования OE-маршрутов в связном графе с числом цепей, равным k, где $2k$ – число вершин нечетной степени в гомеоморфном образе раскройного плана.
2. Разработаны алгоритмы решения задачи для разных случаев: плоский эйлеров граф, произвольный плоский связный граф (задача китайского почтальона и задача построения OE-покрытия), произвольный несвязный граф.
3. Разработаны алгоритмы поиска оптимального решения для произвольных плоских графов. Оптимальным решением в данном случае считается OE-покрытие с минимальной длиной дополнительных построений и минимальным числом цепей.
4. Показано, что все разработанные алгоритмы имеют полиномиальную сложность.
5. Введен класс AOE-цепей, в котором на цепь наложено локальное ограничение: смежные ребра цепи соответствуют системе переходов A-цепи. Разработан алгоритм `AOE-TRAIL`, который позволяет построить AOE-цепь для плоского связного 4-регулярного графа. Алгоритм находит решение за время $O(|E(G)| \cdot \log |V(G)|)$.
6. Введен класс NOE-маршрутов в плоских графах. Этот класс является расширением класса AOE и в него входят все OE-цепи, имеющие непересекающиеся переходы. Разработан алгоритм `Non-intersecting` построения NOE-цепи. Его выполнение состоит в сведении исходного плоского графа к плоскому связному 4-регулярному графу за счет

расщепления вершин степени выше 4 и дальнейшего выполнения алгоритма `AOE-TRAIL`.

7. Определены оценки количества OE-цепей в эйлеровом графе для фиксированной системы переходов. Решение данной задачи может быть полезно при генерации допустимых вариантов маршрутизации.
8. Рассмотренные алгоритмы могут быть применены в проектировании CAD/CAM систем технологической подготовки процессов раскроя, ориентированных на применение ресурсосберегающих ECP и ICP технологий.

Список использованных источников

1. *Алифанов Д.В., Лебедев В.Н., Цурков В.И.* Оптимизация расписаний с логическими условиями предшествования // Известия Российской академии наук. Теория и системы управления. 2009. № 6. С. 88–93.

2. *Алферов И.О., Панюкова Т.А.* Техника программной реализации алгоритма построения допустимой эйлеровой цепи // Информационные технологии и системы: материалы Первой междунар.конф. (Банное, Россия, 28.02–4.03.2012) / отв.ред. В.А. Мельников. Челябинск: Изд-во Челяб.гос.ун-та, 2012. С. 32–34.

3. *Андерсон Дж.А.* Дискретная математика и комбинаторика: Пер. с англ. – М.: Издательский дом «Вильямс», 2004. 960 с.

4. *Арутюнян А.Р.* Сравнение эффективности обходчиков UniTESK // Труды института системного программирования РАН. 2006. Т. 10. С. 167–180.

5. *Афанасьев А.П., Гринберг Я.Р., Курочкин И.И., Корх А.В.* Моделирование двухуровневой маршрутизации в задаче последовательного заполнения сети потоками продуктов // Труды Института системного анализа Российской академии наук. 2013. Т. 63. № 4. С. 25–34.

6. *Афанасьев А.П., Гринберг Я.Р., Курочкин И.И.* Равномерные алгоритмы последовательного заполнения потоковой сети потоками продуктов // Труды института системного анализа Российской академии наук. 2005. Т. 14. С. 118–140.

7. *Баламирзоев А.Г., Султанахмедов М.А.* Математическое моделирование транспортных потоков // Современные проблемы математики и смежные вопросы: Материалы Межд. конф. «Мухтаровские чтения». Махачкала, 2008. С. 53–56.

8. *Белоусов А.И.* Дискретная математика / Под ред. В.С. Зарубина, А.П. Крищенко. – 2-е изд., стереотип. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2002. – 744 с.

9. *Белый С.Б.* О самонепересекающихся и непересекающихся цепях // Математические заметки, 1983. Т. 34. № 4. С. 625–628.

10. *Болл У., Коксетер Г.* Математические эссе и развлечения: Пер. с англ. Под ред. с предисл. и примеч. И.М.Яглома. – М.: Мир, 1986. 474 с.

11. *Верхотуров М.А., Тарасенко П.С.* Математическое обеспечение задачи оптимизации пути режущего инструмента при плоском фигурном раскрое на основе цепной резки // Вестник УГАТУ, «Управление, вычислительная техника и информатика». 2008. Т. 10, № 2(27). С. 123–130.

12. Ганелина Н. Д., Фроловский В. Д. Исследование методов построения кратчайшего пути обхода отрезков на плоскости. // Сиб. журн. вычисл. матем. 2006. Том 9(3). С. 241–252.

13. *Гэри М., Джонсон Дж.* Вычислительные машины и труднорешаемые задачи: Пер. с англ. – М.: Мир, 1982. 416 с., ил.

14. *Евстигнеев В.А., Касьянов В.Н.* Толковый словарь по теории графов в информатике и программировании. – Новосибирск: Наука. Сиб. предприятие РАН. 1999. – 291 с.

15. *Емеличев В.А., Зверович И.Э., Мельников О.И., Сарванов В.И., Тышкевич Р.И.* Теория графов в задачах и упражнениях: Более 200 задач с подробными решениями. – М.: Книжный дом «ЛИБРОКОМ», 2013. 416 с.

16. *Ермаченко, А.И.* Рекурсивный метод для решения задачи гильотинного раскроя / *А.И. Ермаченко, Т.М. Сираэтдинов* // Принятие решений в условиях неопределенности. Сб. научн. статей. УГАТУ. – 2000. – С. 35–39.

17. *Забудский Г.Г.* О задаче линейного упорядочения вершин параллельно-последовательных графов // Дискретный анализ и исследование операций. 2000. Т. 7, № 1. С. 61–64.

18. *Зыков А.А.* Основы теории графов. – М.: Вузовская книга, 2004. 664 с.

19. *Канторович Л.В., Залгаллер В.А.* Рациональный раскрой промышленных материалов. – СПб.: Невский Диалект, 2012. 304 с., ил., табл.

20. *Картак В.М.* Задача упаковки прямоугольников: точный алгоритм на базе матричного представления // Вестник УГАТУ, сер. «Управление, вычислительная техника и информатика». 2007. Т. 9. № 4(22), С. 104–110.

21. *Кочетов Ю.А.* Вероятностные методы локального поиска для задач дискретной оптимизации // Дискретная математика и ее приложения. Сборник лекций молодежных и научных школ по дискретной математике и ее приложениям. М.: Изд-во центра прикладных исследований при мех.-матем. ф-те МГУ. 2000. С. 87–117.

22. *Кристофидес Н.* Теория графов. – М.: Мир, 1978. 432 с.

23. *Кулаков Ю.О., Воротников В.В.* Формирование оптимальных маршрутов в мобильных сетях на основе модифицированного алгоритма Дейкстры // Вісник Національного технічного університету України «Київський політехнічний інститут». Серія: Інформатика, управління та обчислювальна техніка. 2012. № 56. С. 13–19.

24. *Лебедев В.Н., Цурков В.И.* Эффективные алгоритмы для игр с запретами и их приложения // Известия Российской академии наук. Теория и системы управления. 2007. № 3. С. 54–58.

25. *Макаровских Т.А., Савицкий Е.А.* Абстрагирование раскройного плана до плоского графа для эффективного решения задачи вырезания деталей // Вестник УГАТУ. 2015. Т.19, №3(69). С 190–196.

26. *Макаровских Т.А., Панюков А.В., Савицкий Е.А.* Алгоритмы маршрутизации для систем технологической подготовки процессов раскроя // Системы проектирования технологической подготовки производства и управления этапами жизненного цикла промышленного продукта(CAD/CAM/PDM-2015). Тезисы 15-й международной конференции. Под ред. А.В. Толока. – М.: ООО «Аналитик». - С. 65–66.

27. *Макаровских Т.А., Панюков А.В., Савицкий Е.А.* Алгоритмы маршрутизации для систем технологической подготовки процессов раскроя // Си-

стемы проектирования технологической подготовки производства и управления этапами жизненного цикла промышленного продукта(CAD/CAM/PDM-2015). Труды 15-й международной конференции. Под ред. А.В. Толока. – М.: ООО «Аналитик». – 2015. С. 182–186.

28. *Макаровских Т.А.* О построении эйлерова AOE-покрытия в плоском графе // Информационный бюллютень №13, XV Всероссийская конф. «Математическое программирование и приложения». 2015. С. 96–97.

29. *Макаровских Т.А.* Определение траектории движения режущего инструмента для прямоугольного раскройного плана, избегающей пересечения имеющихся резов // Proceedings of the 3-rd International Conference «Information Technologies for Intelligent Decision Making Support». 2015. Vol. 1. P. 39–43.

30. *Макаровских Т.А., Панюков А.В.* Определение траектории режущего инструмента с отсутствием пересечения резов // Системы проектирования технологической подготовки производства и управления этапами жизненного цикла промышленного продукта(CAD/CAM/PDM-2016). Труды 16-й международной конференции. Под ред. А.В. Толока. – М.: ООО «Аналитик». - С. 138–142.

31. *Макаровских Т.А.* О числе OE-цепей для заданной системы переходов // Вестник Южно-Уральского государственного университета. Серия «Математика. Механика. Физика». 2016. Т.8. №1. С. 5–12.

32. *Макаровских Т.А.* Покрытие графа для прямоугольного раскройного плана AOE-цепями // Информационные технологии и системы: тр. Четвертой междунар. науч. конф., Банное, Россия, 25 февр. – 1 марта 2015 г. (ИТиС–2015): науч. электр. изд. / отв. ред. Ю.С. Попков, А.В. Мельников. Челябинск: Изд-во Челяб. гос. ун-та, 2015. – С.17–18.

33. *Макаровских Т.А., Савицкий Е.А.* Построение AOE-цепи в плоском графе// Дискретная математика, алгебра и их приложения: Тез. докл. Междунар.науч. конф. Минск, 14–18 сентября 2015 г. – С. 44–45.

34. *Макаровских Т.А., Панюков А.В., Савицкий Е.А.* Программа для оценки укладки деталей прямоугольной формы на плоскости по критерию плотности упаковки и ограничениям маршрута их вырезания / Свидетельство о регистрации программы ЭВМ и базы данных №2016662723, 21.11.2016

35. *Макаровских Т.А.* Программа построения A-цепи с упорядоченным охватыванием в плоском 4-регулярном в графе // Программы для ЭВМ. Базы данных. Топологии интегральных микросхем. – Официальный бюллетень Российского агентства по патентам и товарным знакам. 2015 г. М.: ФИПС. – 2015. – Рег. №2014663188.

36. *Макаровских Т.А.* Техника программной реализации заачи построения AOE-цепи в плоском 4-регулярном графе // тр. Пятой Междунар. науч. конф., Банное, Россия, 24–28 февр. 2016 г. (ИТиС — 2016): науч. электрон. изд. (1 файл 8,9 Мб) / отв. ред. Ю. С. Попков, А. В. Мельников. Челябинск: Изд-во Челяб. гос. ун-та, 2016. – С. 17–20.

37. *Мельников О.И.* Обучение дискретной математике. – М.: Издательство ЛКИ, 2008. 224 с.

38. *Мельников О.И.* Теория графов в занимательных задачах. Изд. 3-е, испр. и доп. – М.: Книжный дом «ЛИБРОКОМ», 2009. 232 с.

39. *Моргунов И.Б.* Разработка математической модели выбора оптимального маршрута // Вестник УГАТУ. Сер. «Математическое моделирование, численные методы и комплексы программ». 2008. Т. 11, № 1(28). С. 194–199.

40. *Мурзакаев Р. Т., Шилов В. С., Бурылов А. В.* Применение метаэвристических алгоритмов для минимизации длины холостого хода режущего инструмента. // Вестник ПНИПУ. Электротехника, информационные технологии, системы управления. 2015. Вып. 14. С. 123-136.

41. *Мухачева Э.А.* Рациональный раскрой промышленных материалов. Применение АСУ. – М.: Машиностроение. 1984. – 176 с.

42. *Панюкова Т.А., Савицкий Е.А.* The Software for Algorithms of Ordered Enclosing Covering Constructing for Plane Graphs // Вестник УГАТУ. 2013. Vol. 17, no. 6 (59). P. 39–44.

43. *Панюкова Т.А.* Алгоритм покрытия плоского графа последовательностью цепей с упорядоченным охватыванием // Материалы международной конференции «Дискретная оптимизация и исследование операций». Новосибирск: Изд-во ин-та математики, 2013. С. 110.

44. *Панюкова Т.А.* Алгоритм построения оптимального эйлерова покрытия для многосвязного графа // Современные информационные технологии и ИТ-образование. Сборник избранных трудов VII Международной научно-практической конференции. Под ред. проф. В.А. Сухомлина. М.: ИНТУИТ.РУ, 2012. – С. 706–713.

45. *Панюкова Т.А.* Алгоритм построения эйлеровых циклов специального вида // Проблемы теоретической кибернетики. Тезисы докладов XIII Международной конференции (Казань, 27–31 мая 2002 г.). Часть II. Под ред. О. Б. Лупанова. М.: Изд-во механико-математического факультета МГУ, 2002. С. 142.

46. *Панюкова Т.А., Савицкий Е.А.* Алгоритм проверки маршрута реза в раскройном плане на сответствие условию упорядоченного охватывания // XII Всероссийское совещание по проблемам управления ВСПУ-2014. Институт проблем управления им. В.А. Трапезникова РАН. М.: Институт проблем управления им. В.А. Трапезникова РАН, 2014. С. 9315–9318.

47. *Панюкова Т.А.* Алгоритмы построения обходов с упорядоченным охватыванием в плоских эйлеровых графах //Материалы Всероссийской конференции «Проблемы оптимизации и экономические приложения» (Омск, 1–5 июля 2003 г.). Омский филиал Института Математики СО РАН. Омск: Изд-во Наследие. Диалог Сибирь, 2003. С. 188.

48. *Панюкова Т.А., Савицкий Е.А.* Допустимые эйлеровы покрытия с упорядоченным охватыванием для многосвязного графа // Статистика. Мо-

делирование. Оптимизация: сб. тр. Всероссийской конференции (Челябинск, 28 ноября – 3 декабря 2011 г.). Челябинск: Издательский центр ЮУрГУ, 2011. С. 154–158.

49. *Панюкова Т.А.* Информационная и технологическая поддержка процессов раскроя одежды // Научные труды молодых исследователей программы «Шаг в будущее». М.: НТА «Актуальные проблемы фундаментальных наук», 1999. Т. 2. С. 168.

50. *Панюкова Т.А.* Информационная и технологическая поддержка процессов раскроя одежды // Сборник материалов Российской молодежной научной и инженерной выставки «Шаг в будущее» (Москва, 9–13 марта, 1999). М.: НТА «Актуальные проблемы фундаментальных наук», 1999. С. 48.

51. *Панюкова Т.А., Савицкий Е.А.* Использование двойственного граневого графа при построении маршрута режущего инструмента автомата раскроя// Информационные технологии интеллектуальной поддержки принятия решений (Proceedings of the 2nd International Conference «Information Technologies for Intelligent Decision Making Support» and the Intended International Workshop «Robots and Robotic Systems»). 2014. С. 284-287.

52. *Панюкова Т.А.* К задаче об эйлеровых циклах с упорядоченным охватыванием/ Т.А. Панюкова//Тезисы Второго Международного конгресса студентов, молодых ученых и специалистов «Молодежь и наука – третье тысячелетие»/YSTM'02 (15–19 апреля 2002 г.). Часть 2. – М.: Издание региональной общественной организации научно-технической ассоциации «Актуальные проблемы фундаментальных наук», 2002. С. 33–34.

53. *Панюкова Т.А.* Комбинаторика и теория графов: Учебное пособие. М.: Книжный дом «ЛЕНАНД», 2014. 208 с.

54. *Панюкова Т.А., Савицкий Е.А.* Композиция интерфейсов при технологической подготовке процессов раскроя // Труды 40-й Региональной молодежной конференции «Проблемы теоретической и прикладной математики». Екатеринбург: УрО РАН, 2009. С. 367–372.

55. *Панюкова Т.А.* Маршруты манипулятора, не содержащие запрещенных переходов // Труды XXXIV Уральского семинара «Механика и процессы управления». Т. 2. Миасс, 2004. С. 556–564.

56. *Панюкова Т.А.* Маршруты с локальными ограничениями // Вестник Южно-Уральского государственного университета. Серия: Математическое моделирование и программирование. 2010. Вып. 5. № 16(192). С. 58–67.

57. *Панюкова Т.А.* Маршруты с локальными ограничениями // Труды 37-й Региональной молодежной конференции «Проблемы теоретической и прикладной математики». Екатеринбург: УрО РАН, 2006. С. 66–70.

58. *Панюкова Т.А.* Маршруты с локальными ограничениями // Информационно-математические технологии в экономике, технике и образовании: сборник тезисов Международной научной конференции. Екатеринбург: УГТУ-УПИ, 2007. С. 303–305.

59. *Панюкова Т.А., Алферов И.О.* Маршруты с локальными ограничениями: алгоритмы и программная реализация // Прикладная информатика. 2013. № 1(43). С. 80–90.

60. *Панюкова Т.А.* Маршруты с упорядоченным охватыванием // Труды VII международной конференции «Дискретные модели в теории управляющих систем» (Покровское, 4–6 марта, 2006 г.). М.: МАКС Пресс, 2006. С. 265–271.

61. *Панюкова, Т.А.* Модель движения режущего инструмента при условии вырезаний только соседних деталей // «Информационно-телекоммуникационные системы и технологии» (ИТСиТ-2014) Материалы Всероссийской научно-практической конференции. Кемерово, 2014. С. 411–412.

62. *Панюкова Т.А.* Некоторые критерии оценки раскройных планов // IV Всероссийская конференция «Проблемы оптимизации и экономические приложения». Материалы конференции (Омск, 29 июня–4 июля, 2009). Омский филиал Института математики им. С.Л. Соболева СО РАН. Омск: Полиграфический центр КАН, 2009. С. 238.

63. *Панюкова Т.А., Савицкий Е.А.* О некоторых критериях оценки покрытий с упорядоченным охватыванием // Материалы X Международного семинара «Дискретная математика и ее приложения» (1–6 февраля, 2010 г.). М.: Изд-во механико-математического факультета МГУ, 2010. С. 444.

64. *Панюкова Т.А., Алферов И.О., Сахаров И.А.* Об алгоритме построения совместимых путей в графе // Информационный бюллетень №12 Ассоциации математического программирования. XIV Всероссийская конференция «Математическое программирование и приложения». Екатеринбург, 2011. С. 120–121.

65. *Панюкова Т.А.* Об оптимальном эйлеровом покрытии плоских графов // Информационные технологии и системы: тр. Второй междунар. науч. конф. (Банное, Россия, 27 февр.–3 марта 2013 г.): науч. электр. изд. / отв. ред. А. В. Мельников. Челябинск: Изд-во Челяб.гос.ун-та, 2013. С. 52–54.

66. *Панюкова Т.А.* О построении маршрута движения режущего инструмента при условии вырезания только соседних деталей // Информационные технологии и системы: тр. Третьей междунар. науч. конф. (Банное, Россия, 26 февр.–2 марта 2014 г.): науч. электр. изд. Отв. ред. Ю.С. Попков, А.В. Мельников. Челябинск: Изд-во Челяб. гос. ун-та, 2014. С. 41–42.

67. *Панюкова Т.А.* Оптимальные эйлеровы покрытия с упорядоченным охватыванием для плоских графов // Дискретный анализ и исследование операций. 2011. Том 18, № 2. С. 64–74.

68. *Панюкова Т.А.* Оптимизация использования ресурсов при технологической подготовке процессов раскроя // Прикладная информатика. 2012. № 3(39). С. 20–32.

69. *Панюкова Т.А.* Последовательности цепей с упорядоченным охватыванием // Известия Академии наук. Теория и системы управления. 2007. № 1. С. 88–97.

70. *Панюкова Т.А.* Построение маршрута для оптимального хода режущего инструмента // Информационно-математические технологии в эконо-

мике, технике и образовании. Тезисы докладов 3-й Международной научной конференции. Екатеринбург: УГТУ-УПИ, 2008. Ч. 2. С. 12–13.

71. *Панюкова Т.А.* Построение маршрутов с упорядоченным охватыванием в плоских графах // Труды 36-й Региональной молодежной конференции «Проблемы теоретической и прикладной математики». Екатеринбург: УрО РАН, 2005. С. 61–66.

72. *Панюкова Т.А., Мирасов В.Ф.* Построение совместимых цепей в графах // Проблемы теоретической и прикладной математики: тр. 39-й Регион. молодеж. конф. Екатеринбург, 2008. С. 38–43.

73. *Панюкова Т.А.* Построение эйлерова покрытия с упорядоченным охватыванием для плоского графа с прямоугольными гранями // X Белорусская математическая конференция: Тез. докл. междунар. науч. конф. (Минск, 3–7 ноября 2008 г.). Мн.: Институт математики НАН Беларуси, 2008. Ч. 5. С. 98.

74. *Панюкова Т.А., Мухачева Э.А.* Проблема рационального использования промышленных материалов: оптимизация обратного хода раскроя // Обратные задачи в приложениях. Сборник статей научно-практической конференции. Бирск: БирГСПА, 2008. С. 270–276.

75. *Панюкова Т.А.* Программное обеспечение для построения последовательностей цепей с упорядоченным охватыванием // Проблемы теоретической и прикладной математики: труды 39-й Региональной молодежной конференции. Екатеринбург: УрО РАН, 2008. С. 393–398.

76. *Панюкова Т.А.* Построение эйлеровых циклов специального вида // Научные труды молодых исследователей программы «Шаг в будущее». М.: НТА «Актуальные проблемы фундаментальных наук», 1999. Т. 1. С. 108.

77. *Панюкова Т.А.* Покрытия с упорядоченным охватыванием с минимальной длиной дополнительных построений // Российская конференция «Дискретная оптимизация и исследование операций»: Материалы конферен-

ции (Алтай, 27 июня – 3 июля, 2010). Новосибирск: Изд-во Ин-та математики. 2010. С. 98.

78. *Панюкова Т.А.* Построение эйлеровых циклов специального вида в планарном графе // Материалы VII Международного семинара «Дискретная математика и ее приложения»; Ч. II. Под ред. О. Б. Лупанова. М.: Изд-во центра прикладных исследований при механико-математическом факультете МГУ, 2001. С. 149.

79. *Панюкова Т.А.* Построение эйлеровых циклов с упорядоченным охватыванием как математическая модель решения задачи раскроя // Современные информационные технологии и ИТ-образование /Сборник избранных трудов VIII Международной научно-практической конференции. Под ред. проф. В.А. Сухомлина. М.: ИНТУИТ.РУ, 2013. С. 706–713.

80. *Панюкова Т.А.* Свидетельство Роспатента о государственной регистрации программы построения маршрутов с упорядоченным охватыванием (Ordered Routes Constructor) №2005612413 от 22 сентября 2005, правообладатель: Панюкова Т.А.

81. *Панюкова Т.А., Савицкий Е.А.* Свидетельство Роспатента о государственной регистрации программы построения оптимальных покрытий с упорядоченным охватыванием для многосвязных графов №2011617777 от 09 августа 2011, правообладатель: ФГБОУ ВПО «ЮУрГУ» (НИУ).

82. *Панюкова Т.А.* Свидетельство Роспатента о государственной регистрации программы построения эйлеровых циклов с упорядоченным охватыванием (Euler Cycles Constructor) №2004610785 от 3 февраля 2004, правообладатель: Панюкова Т.А.

83. *Панюкова Т.А., Алферов И.О.* Свидетельство Роспатента о государственной регистрации программы построения эйлеровой цепи с запрещенными переходами в графе №2013661312 от 08 октября 2013, правообладатель: ФГБОУ ВПО «ЮУрГУ» (НИУ).

84. *Панюкова Т.А., Савицкий Е.А.* Композиция интерфейсов при технологической подготовке процессов раскроя // Труды 40-й Региональной молодежной конференции «Проблемы теоретической и прикладной математики». Екатеринбург: УрО РАН, 2009. С. 367–372.

85. *Панюкова Т.А., Савицкий Е.А.* Программное обеспечение для построения покрытия с упорядоченным охватыванием для многосвязных плоских графов // Вестник Южно-Уральского государственного университета: Серия "Вычислительная математика и информатика". 2013. Т. 2, № 2. С. 111–117.

86. *Панюкова Т.А.* Обходы с упорядоченным охватыванием в плоских графах // Дискретный анализ и исследование операций. Сер. 2. 2006. Т. 13, № 2. С. 31–43.

87. *Панюкова Т.А.* Рекурсивный алгоритм построения обходов с упорядоченным охватыванием в плоских неэйлеровых графах // Материалы VIII Международного семинара «Дискретная математика и ее приложения» (Москва, 2–6 февраля, 2004 г.). М.: Изд-во механико-математического факультета МГУ, 2004. С. 444.

88. *Панюкова Т.А.* Синтез программ управления процессом раскроя // Обозрение прикладной и промышленной математики. Под ред. Прохорова Ю.В. М.: Научное изд-во «ТВП», 2001. Т. 8, Вып. 2. С. 664.

89. *Панюкова Т.А.* Эйлерово покрытие с упорядоченным охватыванием для многосвязного графа // Математическое моделирование, оптимизация и информационные технологии: сборник научных трудов 3-й Международной конференции. Кишинев, 2012. С. 429–437.

90. *Панюкова Т.А.* Эйлерово покрытие с упорядоченным охватыванием для многосвязного плоского графа // Материалы V Всероссийской конференции (Омск, 2–6 июля, 2012 г.). 2012. С. 154.

91. *Панюкова Т.А.* Эйлеровы циклы специального вида // Российская конференция «Дискретный анализ и исследование операций». Материалы

конференции (Новосибирск, 28 июня–2 июля, 2004 г.). Новосибирск: Изд-во Ин-та математики, 2004. С. 147.

92. *Панюкова Т.А., Панюков А.В.* Эйлеровы циклы с упорядоченным охватыванием // Проблемы теоретической кибернетики. Тезисы докладов XII Международной конференции (Нижний Новгород, 17–22 мая, 1999 г.). Под ред. Лупанова О.Б. М.: Изд-во механико-математического факультета МГУ, 1999. Ч. II. С. 148.

93. *Панюкова Т.А., Панюков А.В.* Decreasing of Length for Routes with Ordered Enclosing // Дискретная математика, алгебра и их приложения. Тез. Докл. Междунар. науч. конф. (Минск, 19–22 октября, 2009 г.). Мн.: Институт математики НАН Беларуси, 2009. С. 155–157.

94. *Петунин А.А., Ченцов А.Г., Ченцов П.А.* К вопросу о маршрутизации движения инструмента в машинах листовой резки с числовым программным управлением / Научно-технические ведомости Санкт-Петербургского государственного политехнического университета. Сер. «Информатика. Телекоммуникации. Управление». 2013. № 169. С. 103–111.

95. *Петунин А.А.* О некоторых эвристических стратегиях формирования маршрута инструмента при разработке управляющих программ для машин термической резки материала // Вестник УГАТУ, «Управление, вычислительная техника и информатика». 2009. Т. 13. № 2(35), С. 280–286.

96. *Петунин, А. А.* Две задачи маршрутизации режущего инструмента для машин фигурной листовой резки с ЧПУ/А. А. Петунин//Интеллектуальные технологии обработки информации и управления: тр. Второй Междунар. конф. Уфа, 2014. С. 215–220.

97. *Райгородский А.М.* Экстремальные задачи теории графов и Интернет. – Долгопрудный: Издательский дом «Интеллект», 2012. 104 с.

98. *Романовский, И.В.* Алгоритмы решения экстремальных задач. – М.: Наука. 1977. – 352 С.

99. *Савицкий Е. А.* Использование алгоритма поиска в ширину для определения уровней вложенности ребер плоского графа // Информационные технологии и системы: тр. Третьей междунар. науч. конф. (Банное, Россия, 26 февр. – 2 марта 2014 г.): науч. электр. изд. / отв. ред. Ю.С. Попков, А.В. Мельников. Челябинск: Изд-во Челяб. гос. ун-та, 2014. С. 43–45.

100. *Свами М., Тхуласираман К.* Графы, сети и алгоритмы: Пер. с англ/ М.: Мир, 1984. 455 с., ил.

101. *Сергеев А.С.* Разработка алгоритма формирования сети трасс и его применение для определения семейства маршрутов в ориентированных графах и сетях // Вестник РГУПС. 2001. № 1. С. 45–48.

102. *Сесекин А.Н., Шолохов А.Е.* Эвристические алгоритмы в задачах маршрутизации перемещений // Информационные технологии и системы: тр. Четвертой междунар. науч. конф. (Банное, Россия, 25 февр. – 1 марта 2015 г.): науч. электр. изд. / отв. ред. Ю.С. Попков, А.В. Мельников. Челябинск: Изд-во Челяб. гос. ун-та, 2015. – С.34–35.

103. *Стоян Ю.Г., Яковлев С.В.* Математические модели и оптимизационные методы геометрического проектирования. Киев. – Наукова думка. 1986. – 268 с.

104. *Султанахмедов М.А.* Управление городскими пассажиропотоками на основе графовых моделей // Вестник Астраханского государственного технического университета. Серия: Управление, вычислительная техника и информатика. 2010. № 2. С. 55–60.

105. *Таваева А.Ф., Петунин А.А.* К вопросу о разработке алгоритмов маршрутизации инструмента лазерных машин листовой резки с числовым программным управление при использовании «цепной» техники резки // Информационные технологии и системы: тр. Третьей междунар. науч. конф. (Банное, Россия, 26 февр.–2 марта 2014 г.): науч. электр. изд./ отв. ред. Ю.С. Попков, А.В. Мельников. Челябинск: Изд-во Челяб. гос. ун-та, 2014. С. 48–51.

106. *Таваева А.Ф., Петунин А.А.* Об одном способе минимизации пути режущего инструмента для машин термической резки // Инженерная мысль машиностроения будущего: материалы всерос. молодеж. науч.-практ. конф. с междунар. участием. Екатеринбург, 2013. С. 365–373.

107. *Фараонов А.В.* Разработка ситуационной модели задачи маршрутизации при необходимости изменения опорного плана на основе нечеткой ситуационной сети / А.В. Фараонов // XII Всероссийское совещание по проблемам управления ВСПУ-2014. Институт проблем управления им. В.А. Трапезникова РАН. Москва, Институт проблем управления им. В.А. Трапезникова РАН, 2014. С. 5101–5113.

108. *Филиппова А.С.* Обзор методов решения задач раскроя-упаковки уфимской научной школы Э.А. Мухачевой // Статистика. Моделирование. Оптимизация/ сборник трудов Всероссийской конференции (Челябинск, 28 ноября–3 декабря 2011 г.). – Челябинск: Издательский центр ЮУрГУ, 2011. С. 73–85.

109. *Фляйшнер Г.* Эйлеровы графы и смежные вопросы. М.: Мир, 2002. 335 с., ил.

110. *Фроловский В.Д.* Автоматизация проектирования управляющих программ тепловой резки металла на оборудовании с ЧПУ // Информационные технологии в проектировании и производстве. - Вып. 4, 2005. – С. 63–67.

111. *Фроловский В.Д., Забелин С.Л.* Разработка и исследование моделей, методов и алгоритмов для синтеза и анализа решений задач геометрического покрытия // Вестник СибГУТИ. 2013. № 2(22). С. 42–53.

112. *Andersen L.D., Fleischner H., Regner S.* Algorithms and outerplanar conditions for A-trails in plane Eulerian graphs. // Discrete Applied Mathematics, 1998. no. 85. P. 99–112.

113. *Chebikin D.* On k-edge-ordered graphs // Discrete Mathematics, 2004. № 281. P. 115–128.

114. *Dewil, R., Vansteenwegen, P., Cattrysse, D., Laguna, M., Vossen, T.* An improvement heuristic framework for the laser cutting tool path problem // International Journal of Production Research, 2015. Volume 53, Issue 6, P. 1761–1776.

115. *Dewil, R., Vansteenwegen, P., Cattrysse, D.* A review of cutting algorithms for laser cutters // International Journal of Manufacturing Technologies, 2016. Volume 87, P. 1865–1884.

116. EURO Special Interest Group on Cutting and Packing // http://www.fe.up.pt/esicup

117. *H. Fleischner, L.W. Beineke, R.J. Wilson* Eulerian Graphs // Selected Topics in Graph Theory 2, Academic Press, London-NewYork, 1983. P. 17–53.

118. *Fleischner H.* Eulerian Graphs and Related Topics. Part 1, Vol.2. Ann. Discrete Mathematics, 1991. № 50.

119. *Fleischner H.* (Some of) the many uses of Eulerian graphs in graph theory (plus some applications) // Discrete Mathematics. 2001. № 230. P. 23–43.

120. *Garfinkel R. S., Webb I. R.* On crossings, the Crossing Postman Problem, and the Rural Postman Problem. // Networks. 1999. Vol. 34(3). P. 173–180.

121. *Hoeft J., Palekar U.S.* Heuristics for the plate-cutting travelling salesman problem // IIE Transactions, 1997, no.29(9), P. 719–731.

122. *Jing Y., Zhige C.* An Optimized Algorithm of Numberical Cutting-Path Control in Garment Manufacturing. // Advanced Materials Research. 2013. Vol. 796. P. 454-457.

123. *Kerivin H.L.M., Lacroix M., Mahjoub A.R.* On the complexity of the Eulerian closed walk with procedence path constraints problem// Electronic Notes in Discrete Mathematics. 2010. № 36. P. 899–906.

124. *Kotzig A.* Moves Without Forbidden Transitions in a Graph // Mat.-Fiz. Casopis 18, 1968. № 1. P. 76–80.

125. Lee M. K., Kwon K. B. Cutting path optimization in CNC cutting processes using a two-step genetic algorithm. // International Journal of Production Research. 2006. Vol. 44(24). P 5307-5326.

126. *Makarovskikh T., Savitskiy E.* Algorithms for Constructing Resource-Saving Cutting Machines// Procedia Engineering. 2015. Vol. 129. P. 781–786.

127. *Makarovskikh T.A., Panyukov A.V.* AOE-Trails Constructing for a Plane Connected 4-Regular Graph. // CEUR Workshop Proceedings. Vol 1623. Pp. 62–71. Online: htpp://ceur-ws.org/Vol-1623

128. *Makarovskikh T.A., Panyukov A.V., Savitsky E.A.* Mathematical Models and Routing Algorithms for CAM of Technological Support of Cutting Processes // ScienceDirect IFAC-PapersOnLine 49-12 (2016) 821–826. Available online at http://www.sciencedirect.com/science/article/pii/S2405896316311624

129. *Makarovskikh T.A.* On the number of starting points for a fixed cutting plan and fixed cutter trajectory // Proceedings of the 2-nd International Conference «Inteligent Technologies for Information Processing and Management». Ufa: USATU, 2014. Vol. 1. P. 239–244.

130. *Makarovskikh T.A.* The Algorithm for Constructing of Cutter Optimal Path // Journal of Computational and Engineering Mathematics. 2014. Vol. 1, №2. P. 52–61.

131. *Makarovskikh T.A., Savitskiy E.A.* The features of cutting plans design for different cutting technologies// Information Technologies for Intelligent Decision Making Support (ITIDS'2016) Proceedings of the 4th International Conference. 2016. C. 98–103.

132. *Manber U., Bent S.W.* On Non-intersecting Eulerian Circuits// Discrete Applied Mathematics, Vol. 18, 1987. P. 87–94.

133. *Manber U., Israni S.* Pierce Point Minimization and Optimal Torch Path Determination in Flame Cutting// Journal of Manufacturing Systems, Vol. 3, No.1, 1984. P. 81–89.

134. *Mao-Cheng Cai* An algorithm for an Eulerian trail travrsing specified edges in given order// Discrete Applied Mathematics, 1994. № 55. P. 233–239.

135. *Panioukova T.A., Panyukov A.V.* Algorithms for Construction of Ordered Enclosing Traces in Planar Eulerian Graphs // The International Workshop on Computer Science and Information Technologies' 2003, Proceedings of Workshop (Ufa, September 16–18, 2003). Ufa: USATU, 2003. Vol. 1. P. 134–138.

136. *Panyukova T.* Eulerian Cover with Ordered Enclosing for Flat Graphs// Abstracts of Sixth Czech-Slovak International Symposium on Combinatorics, Graph Theory, Algorithms and Applications. Prague, Charles University, 2006. P. 103–104.

137. *Panioukova T.A., Panyukov A.V.* The Algorithm for Tracing of Flat Euler Cycles with Ordered Enclosing // Известия Челябинского научного центра УрО РАН. 2000. № 4(9). P. 18–22.

138. *Panyukova T.A.* Constructing of OE-postman Path for a Planar Graph // Вестник Южно-Уральского государственного университета. Серия: Математическое моделирование и программирование. 2014. Т. 7, № 4. С. 90–101.

139. *Panyukova T.* Covering with ordered enclosing for a disconnected graph // Proceedings of the Workshop on Computer Science and Information Technologies (Sheffield, England, September 16–22, 2014). Ufa: USATU, 2014. Vol. 1. P. 132–137.

140. *Panyukova T.* Covering with ordered enclosing for a multiconnected graph // Abstracts of the Seventh Czech-Slovak International Symposium on Graph Theory, Combinatorics, Algorithms and Applications (Kosice, Slovakia, 7–13 July, 2013), 2013. P. 65.

141. *Panyukova T.* Chain Sequences with Ordered Enclosing // Journal of Computer and System Sciences International. 2007. Vol. 46, No. 1. P. 83–92.

142. *Panyukova T.* Eulerian Cover with Ordered Enclosing for Flat Graphs// Electronic Notes in Discrete Mathematics. 2007. No. 28. P. 17–24.

143. *Panyukova T., Mukhacheva E.A.* Mathematical Models for Cutting Process Design // IFAC Proceedings Volumes. 13th IFAC Symposium on Information Control Problems in Manufacturing, INCOM'09. Cep. «Proceedings of the 13th IFAC Symposium on Information Control Problems in Manufacturing, INCOM'09» sponsors: Honeywell. Moscow, 2009. P. 1085–1090.

144. *Panyukova T.* On Algorithms for Eulerian Trails Constructing // Proceedings of the Workshop on Computer Science and Information Technologies (Vienna-Budapest-Bratislava, September 15–21, 2013). Ufa: USATU, 2013. Vol. 1. P. 176–181.

145. *Panyukova T., Savitskiy E.* Optimization of Resources Usage for Technological Support of Cutting Processes // Proceedings of the Workshop on Computer Science and Information Technologies (Russia, Moscow–St.Petersburg, September 13–19, 2010). Ufa State Technical University, 2010. Vol. 1. P. 66–70.

146. *Panyukova T.* Ordered Enclosing Covers with Minimal Length of Additional Edges// Abstracts of 8th French Combinatorial Conference (Orsay, France, June,28–July,2, 2010). 2010. Abstract № 18.

147. *Panyukova T.* Pattern Maker Project – from Doll to Reality. Informative and Technological Support for Clothes Cutting Process// 11th European Union Contest for Young Scientists (19–26 September 1999, Thessaloniki, Greece). Documents of the Contest: European Commission, Brussels, Belgium, 1999. P. 130.

148. *Panyukova T.* Routing problems for cutting processes // The International Workshop on Computer Science and Information Technologies' 2008. Proceedings of Workshop (Turkey, Antalya, September 15–17, 2008). Ufa: Ufa State Technical University, 2008. Vol. 2. P. 100–106.

149. *Panyukova T.* Special Routes in Flat Graphs// Short abstracts of the international meeting "Euler and Modern Combinatorics" (St. Petersburg, June 1–7, 2007), 2007. P. 34–35.

150. *Panyukova T., Panyukov A.* The Algorithm for Construction of Euler Cycles with Ordered Enclosing// Российская конференция «Дискретный анализ и исследование операций»: Материалы конференции (Новосибирск, 24–28 июня, 2002). Новосибирск: Издательство Института математики СО РАН, 2002. С. 147.

151. *Panyukova T.* The Covering of Graphs by Trails with Local Restrictions// Proceedings oh the 13-th International Workshop on Computer Science and Information Technologies. Уфа: Издательство УГАТУ, 2011. Т. 1. С. 202–207.

152. *Panyukova T., Savitskiy E.* The Software for Algorithms of Ordered Enclosing Covering Constructing for Plane Graphs // The Proceedings of the International Conference "Information Technologies for Intelligent Decision Making Support" (2013, May 21–25, Ufa, Russia). Уфа: изд-во УГАТУ, 2013. С. 122–125.

153. *Panyukova T., Alferov I.* The Software for Constructing Trails with Local Restrictions in Graphs // Open Journal of Discrete Mathematics. 2013. No. 3, Vol. 2. P. 86–92. DOI: 10.4236/ojdm.2013.32017.

154. *Panyukova T.* The Special Routes in Plane Graphs// Abstracts for 6-th International Congress on Industrial and Applied Mathematics (Zurich, 16–20 July, 2007). 2007. P. 449–450.

155. *Panyukova T.* The special routes in plane graphs // PAMM. Special Issue: Sixth International Congress on Industrial Applied Mathematics (ICIAM07) and GAMM Annual Meeting, Zurich, 2007, Published Online: 12 Dec 2008. Vol. 7. Issue 1. P. 2070015–2070016. DOI: 10.1002/pamm.200701066.

156. *Pisanski T., Tucker T.W., Zitnik A.* Straight-ahead walks in Eulerian graphs // Discrete Mathematics, 2004. №. 281. P. 237–246.

157. *Szeider S.* Finding Paths in Graphs Avoiding Forbidden Transitions // Discrete Applied Mathematics, 2003. № 126. P. 261–273.

158. *Vazirani V. V.* A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{V}E)$ general graph maximum matching algorithm // Combinatorica, 14(1):71–109, 1994.

159. *Xie S. Q., et al.* Optimal process planning for compound laser cutting and punch using Genetic Algorithms. // International Journal of Mechatronics and Manufacturing Systems. 2009. Vol. 2 (1/2). P 20-38.

160. *Zitnik A.* Plane graphs with Eulerian Petrie walks // Discrete Mathematics. 2002. Vol. 244. P. 539–549.

ПРИЛОЖЕНИЯ

Приложение 1. Определения функций построения OE-покрытия

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop
#include <fstream.h>
#include "Unit3.h"
bool wasopened;
int I;
int ADD;

//===========================================================================
void EG::REPLACE(int edge){
int rv, rl;
int mt, pt;
   mt=Mark1[edge];
   Mark1[edge]=Mark2[edge];
   Mark2[edge]=mt;
   rv=Vertex2[edge];
   rl=LEdge2[edge];
   Vertex2[edge]=Vertex1[edge];
   LEdge2[edge]=LEdge1[edge];
   Vertex1[edge]=rv;
   LEdge1[edge]=rl;
   pt=prev1[edge];
   prev1[edge]=prev2[edge];
   prev2[edge]=pt;

}

//===========================================================================
void EG::Initialisation(int first){
int Ege;
Stack=new int[EuVertNumb+1];
Mark1=new int[EuNumber+1];
Mark2=new int[EuNumber+1];
prev1=new int[EuNumber+1];
prev2=new int[EuNumber+1];
kmark=new int[EuNumber+1];
nech=new bool[EuVertNumb+1];
for (vertex=1;vertex<=EuVertNumb;vertex++){
    nech[vertex]=false;
    Stack[vertex]=0;
    for (int l=0;l<NechNum;l++){
       if (vertex==NV[l]){
          nech[vertex]=true;
          break;
       }
    }
}
FirstEdge=first;
for (Ege=1; Ege<=EuNumber;Ege++){
    Mark1[Ege]=Mark2[Ege]=N;
    prev1[Ege]=prev2[Ege]=0;
}
First=FirstEdge;
Last=FirstEdge;
vertex=Vertex1[FirstEdge];
NextEdge=LEdge1[FirstEdge];
kmark[FirstEdge]=1;
KM=1;
wasopened=false;
I=0; ADD=EuNumber;

}
//===========================================================================
void EG::Making(){
   while (First!=N){
      while (Mark1[NextEdge]==N&&Last!=NextEdge){
         kmark[NextEdge]=KM;
         Mark1[Last]=NextEdge;
         if (Vertex2[NextEdge]!=vertex) REPLACE(NextEdge);
         vertex=Vertex1[NextEdge];
         Last=NextEdge;
         NextEdge=LEdge1[NextEdge];
      }
      edge=First;
      First=Mark1[First];
      vertex=Vertex2[edge];
      NextEdge=LEdge2[edge];
```

```
      Mark1[edge]=Stack[Vertex1[edge]];
      Mark2[edge]=Stack[vertex];
      if (Mark1[edge]!=0){
         if (Vertex1[edge]==Vertex1[Mark1[edge]])
              prev1[Mark1[edge]]=edge;
         else prev2[Mark1[edge]]=edge;
      }
      if (Mark2[edge]!=0){
         if (vertex==Vertex1[Mark2[edge]])
               prev1[Mark2[edge]]=edge;
         else prev2[Mark2[edge]]=edge;
      }
      Stack[vertex]=edge;
      Stack[Vertex1[edge]]=edge;
      KM=kmark[edge]+1;
   }
}

//=========================================================================
void EG::Form(int vertex){
   edge=Stack[vertex];
   First=edge;
   Last=edge;
         do{
          if (Vertex1[edge]==vertex){
               REPLACE(edge);
            }
               Stack[vertex]=Mark2[edge];
               if (vertex==Vertex1[Mark2[edge]])
                  prev1[Mark2[edge]]=0;
               else
                  prev2[Mark2[edge]]=0;
               vertex=Vertex1[edge];
               int pprev=prev1[edge];
               if (pprev!=0){   //Edge - внутри стека vertex
                  if (edge==Mark1[pprev])
                     Mark1[pprev]=Mark1[edge];
                  else
                     Mark2[pprev]=Mark1[edge];
               }else {
                     //Edge - в вершине стека vertex
                     Stack[vertex]=Mark1[edge];
                     if (vertex==Vertex1[Mark1[edge]])
                        prev1[Mark1[edge]]=0;
                     else
                        prev2[Mark1[edge]]=0;
               }
            edge=Stack[vertex];
            Mark1[Last]=edge;
         //   KS[Last]=K;
            Last=edge;
         }while (Last!=0);
}

//=========================================================================
void EG::ReadFromFile (char *InFile){
    edge=1;
    ifstream inf;
    inf.open(InFile);
    if (inf){
    inf>>EuNumber>>EuVertNumb;
    Vertex1=new unsigned[EuNumber+1];
    Vertex2=new unsigned[EuNumber+1];
    LEdge1=new unsigned[EuNumber+1];
    LEdge2=new unsigned[EuNumber+1];
//    KS=new int [EuNumber+1];
    for (edge=1;edge<=EuNumber;edge++){
        inf>>Vertex1[edge]>>Vertex2[edge]>>LEdge1[edge]>>LEdge2[edge];
     //   KS[edge]=0;
    }
    inf.close();
   }else{
      ShowMessage("Проблемы с файлом!!!");
   }
}

//=========================================================================
void EG::WriteToFile(char *OutFile){
    ofstream outf;
    if (!wasopened){
       outf.open(OutFile);
       wasopened=true;
    }else
       outf.open(OutFile,ios::app);
    do{
       outf<<First<<"-> ";
       VerArray[I]=First;
       First=Mark1[First];
       I++;
    }while (First!=0);
    outf<<"\n";
    if (NechNum>2){
       ADD++;
```

```
       VerArray[I]=ADD;
       I++;
    }else{
       if (NechNum==0) VerArray[I]=INT_MAX;
    }
    outf<<"\n";
    outf.close();
}
//===========================================================================
int Nearest(double **rasst,int k){
}
//===========================================================================
int* EG::EuLoop(char *InFile, char *OutFile, int first, int*nv, int k,double **rasst){
        ReadFromFile(InFile);// { code1.dat }
        NV=nv;
        NechNum=k;
        VerArray=new int [EuVertNumb+k/2];
        Initialisation(first);
        Making();
       SortNech();
        int v=Vertex1[FirstEdge];
//        int q=NV[NechNum];
        while(NechNum!=0){
                int q=NV[NechNum];
          /*  int v2=INT_MAX;
            for(int i=1;i<=k;i++)
               if (NV[i]==q)
                  v2=i;*/
            DelVer(q);
            v=FormNech(q);
            WriteToFile(OutFile);
            //**********
            /* int v1=INT_MAX;
            for(int i=1;i<=k;i++)
               if (NV[i]==v) v1=i;
           for (int i=1;i<=k;i++){
               rasst[v1][i]=INT_MAX;
               rasst[i][v1]=INT_MAX;
               rasst[v2][i]=INT_MAX;
               rasst[i][v2]=INT_MAX;
            }
            int z=INT_MAX;
           for (int i=0;i<k;i++){
               if (z>rasst[v1][i]) z=rasst[v1][i];
               q=NV[i+1];
            }*/
            //**********
            DelVer(v);
        }
        Form(v);
        //Route(OutFile);
        WriteToFile(OutFile); //{output.dat}
        return VerArray;
}
//===========================================================================
void EG::SortNech(){
   bool fl;
   int n=NechNum-1;
   do{
      fl=false;
      for (int i=1;i<=n;i++){
         if (kmark[Stack[NV[i]]]>kmark[Stack[NV[i+1]]]){
            int t=NV[i];
            NV[i]=NV[i+1];
            NV[i+1]=t;
            fl=true;
         }
      }
      n--;
   }while (fl);
}
//===========================================================================
void EG::DelVer(int what){
int i=1;
   while (NV[i]!=what)i++;
for (int j=i;j<(NechNum);j++){
   NV[j]=NV[j+1];
}
NV[NechNum]=INT_MAX;
nech[what]=false;
NechNum--;
}
//===========================================================================
int EG::FormNech(int V_Max){
   vertex=V_Max;
   edge=Stack[vertex];
   First=edge;
   Last=edge;
         do{
           if (Vertex1[edge]==vertex){
               REPLACE(edge);
            }
               Stack[vertex]=Mark2[edge];
               if (vertex==Vertex1[Mark2[edge]])
                  prev1[Mark2[edge]]=0;
               else
                  prev2[Mark2[edge]]=0;
```

```
               vertex=Vertex1[edge];
               int pprev=prev1[edge];
               if (pprev!=0){   //Edge - внутри стека vertex
                  if (edge==Mark1[pprev])
                     Mark1[pprev]=Mark1[edge];
                  else
                     Mark2[pprev]=Mark1[edge];
               }else {
                     //Edge - в вершине стека vertex
                     Stack[vertex]=Mark1[edge];
                     if (vertex==Vertex1[Mark1[edge]])
                        prev1[Mark1[edge]]=0;
                     else
                        prev2[Mark1[edge]]=0;
               }
            if (nech[vertex]){
               Mark1[Last]=0;
             //  KS[Last]=K;
               return vertex;
            }
            edge=Stack[vertex];
            Mark1[Last]=edge;
          //  KS[Last]=K;
            Last=edge;
         }while (Last!=0);
return vertex;
}

//============================================================================
#pragma package(smart_init)
```

Приложение 2. Определение методов класса EulerWayMaker

```
#include "EulerWayMaker.h"

EulerWayMaker::EulerWayMaker() {
    out_e = new int[5];
    out_v = new int[2];
    hasBridges = 0;
}

EulerWayMaker::~EulerWayMaker() {
    delete[] out_e;
    delete[] out_v;
}

void EulerWayMaker::findWay() { //Найти путь в графе
    Init();
    Order();
    Form();
}

void EulerWayMaker::Init() { //Собрать информацию о графе
    //Начальные значения
    for (int i = 0; i < V.size(); i++) {
        V[i].kmark = 0;
        V[i].odd_go = -1;
    }
    for (int i = 0; i < E.size(); i++) {
        E[i].kmark = 0;
        E[i].free = true;
        E[i].left_face = -1;
        E[i].right_face = -1;
    }
    Edge e;
    dop.push_back(e);
    F.push_back(0); //Определили внешнюю грань
    findBridges(); //Найти мосты
    countSmej(); //Определить смежные вершины
    findBorder(); //Найти внешнюю грань
    findFaces(); //Отметить все грани
}

void EulerWayMaker::Order() { //Упорядочение (вложенность ребер, списки инцедентных ребер)
    countEdgeKmark(); //Определить уровни вложенности ребер и вершин
    findOddPairs(); //Найти вершины нечетной степени
}

void EulerWayMaker::Form() { //сформировать путь
    int start_v = 0; //start vertex
    for (int i = 0; i < V.size(); i++) {
        V[i].max_kmark = maxVertKmark(i);
        V[i].kmark = minVertKmark(i);
        if (V[i].odd_go != -1)
            V[i].can_go = true;
        else
            V[i].can_go = false;
    }
    for (int i = 0; i < E.size(); i++) {
        if (E[i].kmark == 1) {
            start_v = E[i].v1;
            break;
        }
    } //for (first vertex)
    for (int i = 0; i < E.size(); i++) {
        if (E[i].kmark == 1) {
            if (V[E[i].v1].max_kmark >= V[start_v].max_kmark
                    && V[E[i].v1].can_go)
                start_v = E[i].v1;
            if (V[E[i].v2].max_kmark >= V[start_v].max_kmark
                    && V[E[i].v2].can_go)
                start_v = E[i].v2;
        }
    }

    if (V[start_v].kmark <= V[V[start_v].odd_go].kmark) {
        if (V[start_v].can_go) {
            V[start_v].can_go = false;
            V[V[start_v].odd_go].can_go = false;
            start_v = V[start_v].odd_go;
        }
    }

    bool was_dop = true;
    while (V[start_v].deg > 0) { //(hasFreeEdges()) //while has edges
        if (V[start_v].can_go == false && V[start_v].odd_go != -1)
            was_dop = true;
        if (was_dop) {
            start_v = formWay(start_v, -1);
            was_dop = false;
```

```
        } else
            start_v = formWay(start_v, way[way.size() - 1]);

        if (V[start_v].odd_go != -1) { //is this vertex odd
            if (V[start_v].deg == 0) { //no more way
                V[start_v].can_go = false;  //make dop
                V[V[start_v].odd_go].can_go = false;
                addToWayDop(start_v, V[start_v].odd_go);
                start_v = V[start_v].odd_go;
            } else if (V[start_v].kmark <= V[V[start_v].odd_go].kmark) { //if kmark to go not less
                if (V[start_v].can_go) { //if can make dop edge
                    if (isWay(start_v, V[start_v].odd_go)) { // can go back
                        V[start_v].can_go = false;
                        V[V[start_v].odd_go].can_go = false;
                        addToWayDop(start_v, V[start_v].odd_go);
                        start_v = V[start_v].odd_go;
                        //was_dop = true;
                    }
                }
            }
        }
    }
}

int EulerWayMaker::formWay(int s_v, int last_e) { //goes from start vert to next odd vert
    int next_v, next_e;
//find first edge
    if (last_e == -1) {
        next_e = getNextWayEdge(s_v);
        addToWay(next_e);
        next_v = getNextWayVert(s_v, next_e);
    } else {
        next_e = getNextWayEdge(s_v, last_e);
        addToWay(next_e);
        next_v = getNextWayVert(s_v, next_e);
    }
//while not odd of deg == 0 next vert goes on
    while ((V[next_v].odd_go == -1) && (V[next_v].deg > 0)) {
        next_e = getNextWayEdge(next_v, next_e);
        if (E[next_e].bridge == 1)
            next_e = getNextWayEdge(next_v);
        addToWay(next_e);
        next_v = getNextWayVert(next_v, next_e);
    }
    return next_v; // returns end_vert
}

int EulerWayMaker::getNextWayVert(int last_v, int next_e) { //returns next vert by last vert and edge
    if (E[next_e].v1 == last_v)
        return E[next_e].v2;
    else
        return E[next_e].v1;
}

int EulerWayMaker::getNextWayEdge(int last_v) { //returns next edge by curent vert
    int imax, maxe = 0, max = 0;
    for (int i = 0; i < E.size(); i++)
        if (E[i].free && (E[i].v1 == last_v || E[i].v2 == last_v)
                && E[i].kmark > maxe)
            maxe = E[i].kmark;
    for (int i = 0; i < E.size(); i++) {
        if (E[i].free && E[i].kmark == maxe) {
            if (E[i].v1 == last_v && F[E[i].left_face] > max) {
                max = F[E[i].left_face];
                imax = i;
            } else if (E[i].v2 == last_v && F[E[i].right_face] > max) {
                max = F[E[i].right_face];
                imax = i;
            }
        }
    }
    return imax;
}

int EulerWayMaker::getNextWayEdge(int last_v, int last_e) { //returns next edge by current vert and last edge
    int max_v_kmark = maxVertKmark(last_v);
//init next_e
    int next_e = last_e;
    while (!E[next_e].free) {
        if (E[next_e].v1 == last_v)
            next_e = E[next_e].left_edge;
        else
            next_e = E[next_e].right_edge;
    }
//if has biger kmark => goes there
    while ((E[next_e].kmark < max_v_kmark) || !E[next_e].free) {
        if (E[next_e].v1 == last_v)
            next_e = E[next_e].left_edge;
        else
            next_e = E[next_e].right_edge;
    }
    return next_e;
}

int EulerWayMaker::maxVertKmark(int n_v) { //find max kmark
```

```
    int max = 0;
    for (int i = 0; i < E.size(); i++) {
        if (E[i].free && (E[i].v1 == n_v || E[i].v2 == n_v)
                && E[i].kmark > max) {
            max = E[i].kmark;
        }
    }
    return max;
}

int EulerWayMaker::minVertKmark(int n_v) { //find min kmark
    int min = 100;
    for (int i = 0; i < E.size(); i++) {
        if (E[i].free && (E[i].v1 == n_v || E[i].v2 == n_v)
                && E[i].kmark < min) {
            min = E[i].kmark;
        }
    }
    return min;
}

bool EulerWayMaker::hasFreeEdges() {  //has edges not in way
    for (int i = 0; i < E.size(); i++) {
        if (E[i].free)
            return true;
    }
    return false;
}

bool EulerWayMaker::isWay(int start_v, int end_v) { //is way from start_v to end_v
    for (int i = 0; i < V.size(); i++) //init mark
        V[i].mark = false;
    bool f = true; //vertex was marked
    V[start_v].mark = true;
    while (f) {
        f = false;
        for (int i = 0; i < E.size(); i++) { //for all edges
            if (E[i].free) { //can go this edge
                if (V[E[i].v1].mark && !V[E[i].v2].mark) {
                    V[E[i].v2].mark = true;
                    f = true;
                }
                if (!V[E[i].v1].mark && V[E[i].v2].mark) {
                    V[E[i].v1].mark = true;
                    f = true;
                }
            }
        }
        if (V[end_v].mark)
            return true;
    }
    return false;
}

void EulerWayMaker::addToWay(int n) { //add edge to way
    way.push_back(n);
    V[E[n].v1].deg--;
    V[E[n].v2].deg--;
    E[n].free = false;
}

void EulerWayMaker::addToWayDop(int v1, int v2) { //add edge to way as dop
    Edge e;
    e.bridge = 0;
    e.free = false;
    e.kmark = -1;
    e.type = 2;
    e.v1 = v1;
    e.x1 = V[v1].x;
    e.y1 = V[v1].y;
    e.v2 = v2;
    e.x2 = V[v2].x;
    e.y2 = V[v2].y;
    dop.push_back(e); //TODO
    way.push_back(-(dop.size() - 1));
}

int EulerWayMaker::getVertId(int x, int y) { //возвращает номер вершины в векторе V или -1 (если вершина не найдена)
    for (int i = 0; i < V.size(); i++) {
        if (V[i].x == x && V[i].y == y)
            return i;
    }
    return -1;
}

void EulerWayMaker::findBridges() { //Найти все мосты графа
//Найти все мосты
    int i, res = 0;
    for (i = 0; i < E.size(); i++) {
        E[i].bridge = isBridge(i);
        if (E[i].bridge)
            res = 1;
    }
    for (i = 0; i < V.size(); i++)
        V[i].selected = 0; //снимаем выделение со всех вершин
    hasBridges = res;
}
```

```
int EulerWayMaker::isBridge(int id) { //Является ли ребро id мостом
    int i;
    for (i = 0; i < V.size(); i++)
        V[i].selected = 0; //снимаем выделение со всех вершин
//Если дошли из одной вершины ребра до второй не проходя через него - не является
    V[E[id].v1].selected = 1; //выделяем первую вершину
    int f = 0; //флаг завершения
    while (!f) {
        f = 1; //нет изменений
        for (i = 0; i < E.size(); i++) {
            if (i != id && V[E[i].v1].selected && !V[E[i].v2].selected) {
                V[E[i].v2].selected = 1;
                f = 0;
            }
            if (i != id && V[E[i].v2].selected && !V[E[i].v1].selected) {
                V[E[i].v1].selected = 1;
                f = 0;
            }
        }
    }
    if (V[E[id].v2].selected)
        return 0;
    else
        return 1;
}

void EulerWayMaker::countSmej() { //Найти ближайшие по повороту против часовой стрелки ребра для
                                  //каждой вершины всех ребер
    countAngles(); //Найти угол наклона каждого ребра
    for (int i = 0; i < E.size(); i++)
        countSmej(i);
}

void EulerWayMaker::countAngles() { //Рассчитать углы ребра
    for (int i = 0; i < E.size(); i++) {
        //Рассчитываем угол ребра
        if (E[i].x1 == E[i].x2) {
            if (E[i].y1 < E[i].y2) {
                E[i].left_angle = M_PI / 2;
                E[i].right_angle = 3 * M_PI / 2;
            }
            if (E[i].y2 < E[i].y1) {
                E[i].left_angle = 3 * M_PI / 2;
                E[i].right_angle = M_PI / 2;
            }
        }
        if (E[i].x1 < E[i].x2) {
            if (E[i].y1 < E[i].y2) {
                E[i].left_angle = atan(
                        (double) (E[i].y2 - E[i].y1) / (E[i].x2 - E[i].x1));
                E[i].right_angle = E[i].left_angle + M_PI;
            }
            if (E[i].y1 > E[i].y2) {
                E[i].left_angle = atan(
                        (double) (E[i].y2 - E[i].y1)
                                / (E[i].x2 - E[i].x1))+2*M_PI;
                E[i].right_angle = E[i].left_angle - M_PI;
            }
            if (E[i].y1 == E[i].y2) {
                E[i].left_angle = 0;
                E[i].right_angle = M_PI;
            }
        }
    }
}

void EulerWayMaker::countSmej(int id) { //Найти ближайшие по повороту против часовой стрелки ребра для ребра id
//считаем
    struct angl {
        int num;
        double a;
    };
    angl *M = new angl[E.size()]; //Создаем массив длЯ углов
    int count = 0; //Число выбраных углов
//Выбираем все инцедентные 1-й вершине ребра
    for (int i = 0; i < E.size(); i++) {
        if (i != id) {
            if ((E[i].v1 == E[id].v1) && (E[i].v2 != E[id].v2)) {
                M[count].num = i;
                M[count].a = E[i].left_angle;
                count++;
            }
            if ((E[i].v2 == E[id].v1) && (E[i].v1 != E[id].v2)) {
                M[count].num = i;
                M[count].a = E[i].right_angle;
                count++;
            }
        }
    }
//Находим ближайшее по повороту против часовой стрелки
    int imin = 0;
//Добавляем 2пи
    for (int i = 0; i < count; i++)
        M[i].a += 2 * M_PI;
//,ычитаем угол
```

```
    for (int i = 0; i < count; i++)
        M[i].a -= E[id].left_angle;
//Находим остаток от деления на 2пи
    for (int i = 0; i < count; i++)
        M[i].a = fmod(M[i].a, 2 * M_PI);
//сам минимум
    for (int i = 0; i < count; i++)
        if (M[i].a > M[imin].a)
            imin = i;
    E[id].left_edge = M[imin].num; //Запоминаем номер
    delete[] M;

    /*То же длЯ 2й вершины */

    M = new angl[E.size()]; //Создаем массив длЯ углов
    count = 0; //Число выбраных углов
//Выбираем все инцедентные 2-й вершине ребра
    for (int i = 0; i < E.size(); i++) {
        if (i != id) {
            if ((E[i].v1 == E[id].v2) && (E[i].v2 != E[id].v1)) {
                M[count].num = i;
                M[count].a = E[i].left_angle;
                count++;
            }
            if ((E[i].v2 == E[id].v2) && (E[i].v1 != E[id].v1)) {
                M[count].num = i;
                M[count].a = E[i].right_angle;
                count++;
            }
        }
    }
//Находим ближайшее по повороту против часовой стрелки
    imin = 0;
//Добавляем 2пи
    for (int i = 0; i < count; i++)
        M[i].a += 2 * M_PI;
//Вычитаем угол
    for (int i = 0; i < count; i++)
        M[i].a -= E[id].right_angle;
//Находим остаток от деления на 2пи
    for (int i = 0; i < count; i++)
        M[i].a = fmod(M[i].a, 2 * M_PI);
//сам минимум
    for (int i = 0; i < count; i++)
        if (M[i].a > M[imin].a)
            imin = i;
    E[id].right_edge = M[imin].num; //Напоминаем номер
    delete[] M;
}

int EulerWayMaker::findBorderEdge() { //Ребро внешней грани для обхода внешней грани
    int n_v = 0;
    int n_r = 0;
    int r_x;
//Найдем верхнюю вершину графа
    for (int i = 0; i < V.size(); i++) {
        if (V[i].y < V[n_v].y)
            n_v = i;
    } //for i
//Найдем ребро, связывающее эту вершину с другой высокой вершиной
    for (int j = 0; j < E.size(); j++) {
        if (E[j].y1 == V[n_v].y && E[j].x1 == V[n_v].x) {
            n_r = j;
            r_x = E[j].x2;
            break;
        }
        if (E[j].y2 == V[n_v].y && E[j].x2 == V[n_v].x) {
            n_r = j;
            r_x = E[j].x1;
            break;
        }
    } //for j
    for (int j = 0; j < E.size(); j++) {
        if (E[j].y1 == V[n_v].y && E[j].x1 == V[n_v].x)
            if (E[j].x2 < r_x) {
                r_x = E[j].x2;
                n_r = j;
            }
        if (E[j].y2 == V[n_v].y && E[j].x2 == V[n_v].x)
            if (E[j].x1 < r_x) {
                r_x = E[j].x1;
                n_r = j;
            }
    } //for j
//Напоминаем внешнее ребро
    return n_r;
}

void EulerWayMaker::findBorder() { //Отметить внешнюю грань
    for (int i = 0; i < E.size(); i++)
        E[i].type = 0;
    int n_r = findBorderEdge();
    int first = n_r; //запоминаем с какой начали

    E[n_r].type = 1;
    int last = n_r;
```

```
    if (E[n_r].y2 > E[n_r].y1) {
        E[n_r].right_face = 0; //внешняя грань
        n_r = E[n_r].right_edge;
    } else {
        E[n_r].left_face = 0; //внешняя грань
        n_r = E[n_r].left_edge;
    }
    int f = 0;
    while (!f) {
        int next = -1;
        E[n_r].type = 1;
        if (E[last].v1 == E[n_r].v1 || E[last].v2 == E[n_r].v1) {
            E[n_r].right_face = 0;
            next = E[n_r].right_edge;
        } //if v1
        if (E[last].v1 == E[n_r].v2 || E[last].v2 == E[n_r].v2) {
            E[n_r].left_face = 0;
            next = E[n_r].left_edge;
        } //if v2
        last = n_r;
        n_r = next;
        if (first == next)
            f = 1; //Дошли до конца
    } //while
}

void EulerWayMaker::markFace(int fn, int e, int v) { //Обойти грань, отметить все ее ребра
    int next_e;
    int next_v;
    if (E[e].v1 == v) {
        E[e].left_face = fn;
        next_e = E[e].left_edge;
    } else if (E[e].v2 == v) {
        E[e].right_face = fn;
        next_e = E[e].right_edge;
    }
    if (E[next_e].v1 == v)
        next_v = E[next_e].v2;
    else if (E[next_e].v2 == v)
        next_v = E[next_e].v1;
    while ((next_e != e) && (next_v != v)) {
        if (E[next_e].v1 == next_v) {
            E[next_e].left_face = fn;
            next_e = E[next_e].left_edge;
        } else if (E[next_e].v2 == next_v) {
            E[next_e].right_face = fn;
            next_e = E[next_e].right_edge;
        }
        if (E[next_e].v1 == next_v)
            next_v = E[next_e].v2;
        else if (E[next_e].v2 == next_v)
            next_v = E[next_e].v1;
    }
}

void EulerWayMaker::findFaces() { //Обозначить все грани графа
    bool f = true;
    while (f) {
        f = false;
        for (int i = 0; i < E.size(); i++) {
            if (E[i].left_face == -1) {
                F.push_back(0);
                markFace(F.size() - 1, i, E[i].v1);
                f = true;
                break;
            }
            if (E[i].right_face == -1) {
                F.push_back(0);
                markFace(F.size() - 1, i, E[i].v2);
                f = true;
                break;
            }
        } //for
    } //while
}

void EulerWayMaker::countEdgeKmark() { //Определить уровни вложенности ребер
//Начиная от face 0
    int klevel = 1;
    F[0] = 1; //Начинаем отмечать с внешней грани
    bool f = true;
    while (f) {
        f = false; //Если ничего не отметили - конец
        for (int i = 0; i < E.size(); i++) {
            if (E[i].kmark == 0) { //еще не определили
                if ((F[E[i].left_face] > 0) || (F[E[i].right_face] > 0)) {
                    E[i].kmark = klevel;
                    V[E[i].v1].kmark = klevel;
                    V[E[i].v2].kmark = klevel;
                    if (F[E[i].left_face] == 0)
                        F[E[i].left_face] = -2;
                    if (F[E[i].right_face] == 0)
                        F[E[i].right_face] = -2;
                    f = true;
                }
            }
        } //for
```

```
        for (int i = 0; i < F.size(); i++)
            if (F[i] == -2)
                F[i] = klevel + 1;
        klevel++;
    }
}

void EulerWayMaker::findOddPairs() { //Найти вершины нечетной степени TODO оптимизировать
//Досчитать вершины нечетной степени
    vector<int> Vert;
    for (int i = 0; i < V.size(); i++)
        if (V[i].deg % 2 == 1)
            Vert.push_back(i);
    int n = Vert.size();
//Достроить матрицу расстояний
    double* r;
    double d1, d2, l;
    r = new double[n * n]; //Расстояния между нечетными вершинами
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++) {
            d1 = V[Vert[i]].x - V[Vert[j]].x;
            d2 = V[Vert[i]].y - V[Vert[j]].y;
            l = sqrt(d1 * d1 + d2 * d2);
            r[i * n + j] = l;
            r[j * n + i] = l;
        }
    for (int i = 0; i < n; i++)
        r[i * n + i] = -1;
//Найти опорное решение
    int* w; //матрица смежности длЯ паросочетания
    w = new int[n * n];
    for (int i = 0; i < n; i++)    //инициализация
        for (int j = 0; j < n; j++)
            w[i * n + j] = 0;
    double min;
    int i_min, j_min;
    int* use = new int[n]; //использовались ли вершины
    for (int i = 0; i < n; i++)
        use[i] = -1; //Не использовались изначально
    for (int t = 0; t < n / 2; t++) {
        //найти начальные min, i_min, j_min
        for (int i = 0; i < n; i++) //находим i_min начальный
            if (use[i] == -1) {
                i_min = i;
                break;
            }
        for (int i = i_min + 1; i < n; i++) //находим j_min начальный
            if (use[i] == -1) {
                j_min = i;
                break;
            }
        min = r[i_min * n + j_min]; //
        //найти минимум
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j < n; j++)
                if ((r[i * n + j] < min) && use[i] == -1 && use[j] == -1) { //минимум+i и j еще не использовались
                    min = r[i * n + j];
                    i_min = i;
                    j_min = j;
                }
        w[i_min * n + j_min] = 1;
        w[j_min * n + i_min] = 1;
        use[i_min] = j_min;
        use[j_min] = i_min;
    }
//Улучшить опорное решение до оптимального   TODO
//Сохранить результат
    for (int i = 0; i < n; i++)
        V[Vert[i]].odd_go = Vert[use[i]];
//очистить память
    delete[] r;
    delete[] w;
    delete[] use;
}

void EulerWayMaker::addEdge(int x1, int y1, int x2, int y2) { //Добавить ребро
    Edge e;
    if (x2 < x1) {
        e.x1 = x1;
        e.y1 = y1;
        x1 = x2;
        y1 = y2;
        x2 = e.x1;
        y2 = e.y1;
    }
    e.v1 = getVertId(x1, y1);
    if (e.v1 == -1) {
        Vert v;
        v.x = x1;
        v.y = y1;
        v.deg = 0;
        V.push_back(v);
        e.v1 = V.size() - 1;
    }
    V[e.v1].deg++;
```

```
    e.v2 = getVertId(x2, y2);
    if (e.v2 == -1) {
        Vert v;
        v.x = x2;
        v.y = y2;
        v.deg = 0;
        V.push_back(v);
        e.v2 = V.size() - 1;
    }
    V[e.v2].deg++;
    e.x1 = x1;
    e.y1 = y1;
    e.x2 = x2;
    e.y2 = y2;
    E.push_back(e);
}

int EulerWayMaker::getEdgeCount() { //Количество ребер
    return E.size();
}

int EulerWayMaker::getWayLength() { //Длина полученного пути
    return way.size();
}

int EulerWayMaker::getVertCount() { //Количество вершин
    return V.size();
}

int* EulerWayMaker::getEdge(int id) { //Вернуть вектор ребер
    out_e[0] = E[id].x1;
    out_e[1] = E[id].y1;
    out_e[2] = E[id].x2;
    out_e[3] = E[id].y2;
    out_e[4] = E[id].kmark;
    return out_e;
}

int* EulerWayMaker::getWayEdge(int id) { //Вернуть вектор ребер
    if (way[id] >= 0) {
        out_e[0] = E[way[id]].x1;
        out_e[1] = E[way[id]].y1;
        out_e[2] = E[way[id]].x2;
        out_e[3] = E[way[id]].y2;
        out_e[4] = E[way[id]].type;

    } else {
        out_e[0] = dop[-way[id]].x1;
        out_e[1] = dop[-way[id]].y1;
        out_e[2] = dop[-way[id]].x2;
        out_e[3] = dop[-way[id]].y2;
        out_e[4] = dop[-way[id]].type;
    }
    return out_e;
}

int* EulerWayMaker::getVert(int id) { //Вернуть вершину
    out_v[0] = V[id].x;
    out_v[1] = V[id].y;
    return out_v;
}
```