Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

М. А. Коротков

РАЗРАБОТКА И РЕАЛИЗАЦИЯ АЛГОРИТМА УКЛАДКИ ДИАГРАММ СОСТОЯНИЙ

Бакалаврская работа

Руководитель: канд. физ.-мат. наук Новиков Ф.А.

Санкт-Петербург
2005

ОГЛАВЛЕНИЕ

1. ВВЕДЕНИЕ

Программный пакет с открытым исходным кодом *UniMod* [1] обеспечивает разработку и выполнение автоматно-ориентированных программ. Он позволяет создавать и редактировать диаграммы классов и состояний *UML* [2], которые соответствуют графу переходов и схеме связей конечного автомата [3]. После создания диаграмм существует возможность выполнить их. При этом содержимое диаграмм преобразуется в *XML*-описание, которое передается интерпретатору, также входящему в пакет *UniMod*. Разработанный пакет базируется на парадигме автоматного программирования [4].

Во многих современных редакторах для текстовых языков программирования существует возможность автоматического форматирования программного кода. В случае если программным кодом является не текст, а набор диаграмм, задача форматирования текста преобразуется в задачу укладки диаграмм. В рамках проекта *UniMod* основной интерес представляет укладка диаграмм состояний языка *UML*.

При укладке диаграммы необходимо учитывать ряд критериев, которые могут противоречить друг другу, например, такие как минимизация площади, занимаемой диаграммой, и наличие достаточного количества свободного места («воздуха»). Каждый критерий оценивает «качество» диаграммы для того или иного применения (отображения на мониторе, печати и т.д.). Такие критерии называются **эстетиками**. Задавшись некоторым набором эстетик, можно построить штрафную функцию (которая тем больше, чем больше расхождения между изображением диаграммы и критериями), и пытаться минимизировать её, перемещая элементы, или разработать алгоритм, последовательно модифицирующий исходную диаграмму так, чтобы результат соответствовал выбранным критериям.

2. ПОСТАНОВКА ЗАДАЧИ

Рассмотрим диаграмму состояний конечного автомата. Она включает в себя состояния и переходы (и у тех и у других есть дополнительные, связанные с ними, атрибуты – метки, но сейчас не стоит останавливаться на этом вопросе). Диаграмме состояний (без вложенных состояний) можно сопоставить граф [5] (при укладке диаграммы состояний не важна ориентация дуг). Каждому состоянию соответствует вершина, а переходу – ребро. При этом необходимо несколько сузить поле деятельности, зафиксировав представление элементов и тип носителя, на котором будет изображаться диаграмма.

2.1. Выбор типа носителя

Диаграмма может быть изображена на плоскости или может быть построена объемная модель [6]. Для представления на мониторе персонального компьютера наиболее естественным является выбор плоского носителя.

2.2. Выбор представления элементов

Выбор вида элементов графа называют **изобразительным соглашением** [7].

В этой работе приведен ряд наиболее популярных изобразительных соглашений. Перечислим некоторые из них:

- **полилинейное изображение** – каждое ребро изображается в виде ломаной линии;
- **прямолинейное изображение** – каждое ребро представляется с помощью отрезка прямой;
- **ортогональное изображение** – каждое ребро графа изображается в виде ломаной линии, состоящей из вертикальных и горизонтальных отрезков;

- **сетчатое изображение** - все вершины, точки пересечения и изгибы ребер имеют целочисленные координаты.

Кроме того, в изобразительном соглашении описывается представление вершины (окружность, прямоугольник, точка). В дальнейшем будем изображать вершину как прямоугольник, а ребро – как ломаную линию с конечным числом изломов (полилинейное изображение).

Выбор такого вида элементов объясняется эстетическими соображениями, а также традициями, сложившимися в области построения диаграмм.

Описанный выбор вида элементов диктует использование декартовой системы координат. Вершину принято изображать как прямоугольник со сторонами, ориентированными параллельно координатным осям. Для изображения ее достаточно знать координаты левого верхнего угла, ширину и высоту, а для изображения ребра – координаты его начала, конца и всех точек излома.

2.3. Понятие укладки

Укладкой графа $G = (V; E)$ в декартовой системе координат $(X; Y)$ назовем множество $L = (G; F_V; F_E)$, где F_V - функция из множества вершин в множество параметров, необходимых для представления вершины в выбранной системе координат (в нашем случае $F_V : V \to X \times Y \times R \times R$, где последняя пара параметров – ширина и высота прямоугольника). F_E - функция из множества ребер в множество параметров, необходимых для представления ребра в выбранной системе координат (в нашем случае $F_E : V \to (X \times Y)^n, n \in N$, где параметр n – количество изломов, вообще говоря изменяется от ребра к ребру). Для простоты будем говорить, что в уложенном графе заданы геометрические параметры каждого элемента.

Задача построения визуального изображения диаграммы по известным параметрам ее элементов в заданной системе координат (по заданной укладке) в настоящей работе не рассматривается.

Две укладки L, L' назовем **изоморфными**, если одна получается из другой путем соответствующего изменения начала координат. Далее будем называть изоморфные укладки одинаковыми. Укладки на рис. 1, например, являются различными.

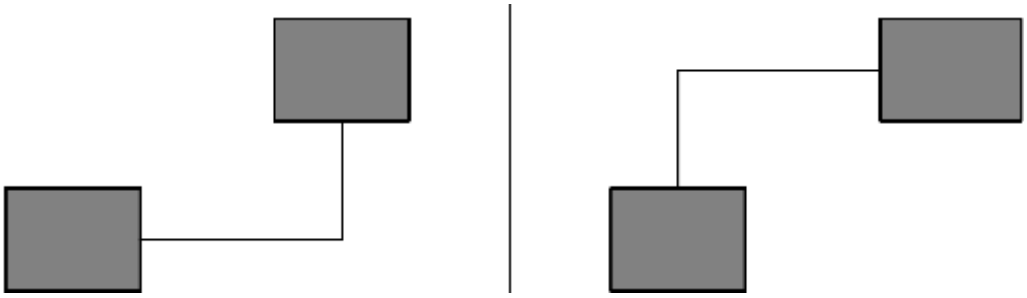

Рис. 1. Различные укладки

Алгоритм укладки диаграммы должен строить пару функций укладки $\{F_V; F_E\}$, точнее соответствующие наборы геометрических параметров для каждой вершины.

Укладку на плоскости будем называть **плоскостной.** Данный термин не является общепринятым, в литературе (например, в работе [8]) чаще говорят просто об изображении графа на плоскости. Укладку, в которой ребра представляют собой ломаные, состоящие только из горизонтальных и вертикальных отрезков, будем называть **плоскостной ортогональной** (или просто **ортогональной**). Ортогональная укладка соответствует соглашению об ортогональном изображении.

2.4. Оценка качества укладки

Теперь необходимо оценить качество укладки (точнее качество изображения, полученного по некоторой укладке). Выделим набор эстетик, по которым будет оцениваться качество укладки. Основная задача такого выделения – обеспечить «читаемость» графа (однозначность представления информации). При этом обеспечивается:

- **минимизация числа пересечений**: число пересечений ребер, прохождений ребер по вершинам и наложений вершин должно быть минимальным;
- **минимизация площади**: площадь, занимаемая выпуклой оболочкой уложенного графа (или площадь минимального прямоугольника, включающего в себя уложенный граф), должна быть минимальной;
- **ограничение «свободного места»**: доля свободного места на диаграмме не должна быть меньше некоторого предела;
- **минимизация изломов**: число изломов должно быть минимально;
- **минимизация общей длины ребер**: суммарная длина ребер должна быть минимальна;
- **минимизация коэффициента сторон**: отношение длины большей стороны к длине меньшей стороны объемлющего граф прямоугольника должно быть минимально;
- **унификация длин ребер**: минимизация различий между длинами ребер на графе.

Укладка слева на рис. 2 значительно лучше читается, чем укладка справа, поскольку укладка слева не удовлетворяет эстетикам минимизации числа пересечений, количества изломов и унификации длин ребер.

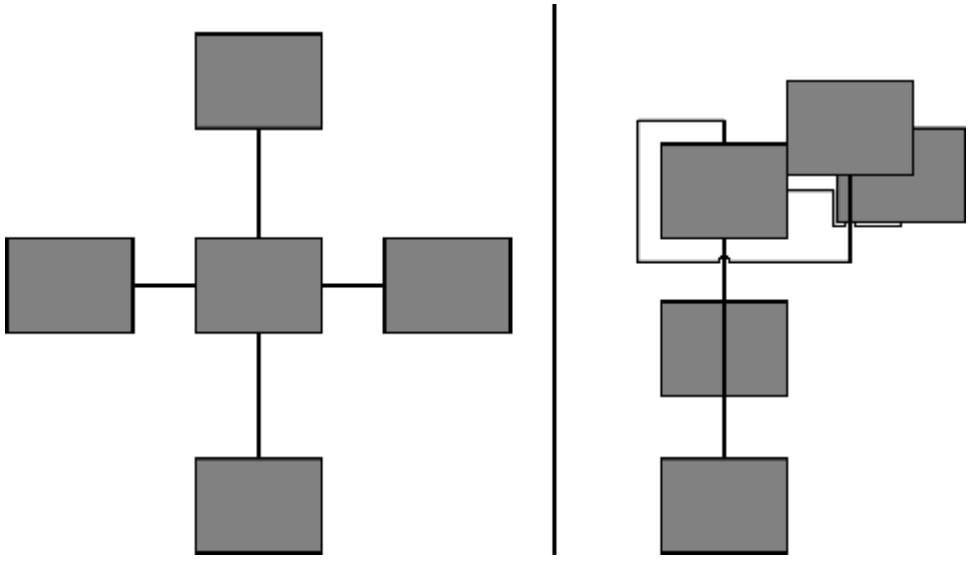

Рис. 2. Укладки графа

Граф, соответствующий диаграмме состояний, отличается от обычного графа возможностью вложения состояний друг в друга, а также выделенных начальных и конечных состояний. Для оценки качества его укладки необходимо учесть дополнительные критерии:

- **унификация размеров вершин**: размер вершин для всех простых состояний (состояний, не имеющих вложенных состояний) необходимо максимально приблизить к заданному оптимальному размеру;
- **разнесение начального и конечного состояний**: расстояние между начальным и конечным состоянием должно быть достаточно велико.

Приведенные выше критерии практически покрывают набор критериев из работы [7] (те из них, которые применимы к ортогональной укладке). Неохваченными остались критерии построения по возможности симметричного изображения и унификации количества изломов на ребре. Первый критерий сложен для оценивания и учета, учет второго не приводит к значительному улучшению читаемости.

Сравните две укладки на рис. 3. Укладка справа более предпочтительна, поскольку на ней переходы «визуально» выполняются слева направо (от начального состояния к конечному). Заметим, что приведенные эстетики не являются строгими и лишь дают понять, какими соображениями будут использоваться при оценке результатов работы программ укладки.

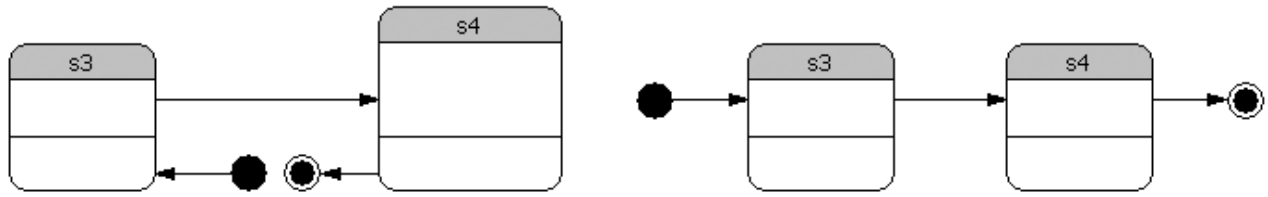

Рис. 3. Укладки диаграммы состояний

Обратим внимание на то, что принятые в диаграмме состояний *UML* начальное и конечное состояние не имеют прямоугольной формы. Для простоты заменим их объемлющими прямоугольниками.

Задача, которая ставится в настоящей работе, состоит в построении качественной ортогональной укладки диаграммы состояний. Наличие алгоритма укладки принципиально важно не только для редактора диаграмм, но и для организации импорта файлов, созданных в сторонних редакторах, так как для *UML* редакторов не существует унифицированного механизма обмена данными, сохраняющего информацию о местоположении элементов [9].

В разрабатываемых алгоритмах будем пренебрегать некоторыми из перечисленных критериев.

2.5. Типы алгоритмов укладки графов

Задача укладки графа не имеет и не может иметь универсального решения, в связи с тем, что набор критериев, применяемых для оценки качества укладки, зависит от типа диаграммы. В работе [10] приведена классификация алгоритмов, применяемых для решения данной задачи. Рассмотрим две группы алгоритмов укладки графов, принципиально отличающиеся подходом к решению:

- алгоритмы с физическим аналогом [11];
- аналитические алгоритмы [8].

2.5.1. Алгоритмы с физическим аналогом

Алгоритмы этой группы ставят в соответствие графу некоторую физическую модель, например, систему пружин, которые стремятся сжаться до некоторой заданной длины (такие алгоритмы называют «пружинным методом») или систему стержней и шарниров, с вершинами – одноименными электрическими зарядами.

Для описания физической модели вводится понятие штрафной функции, задающей потенциальную энергию системы (такие алгоритмы еще называют алгоритмами минимизации потенциала). При этом задача укладки преобразуется в задачу нахождения минимума этой функции, которая решается с помощью сдвига на некоторый вектор каждой вершины графа и проверки изменения значения штрафной функции. Сдвиг вершин выполняется в цикле, условием выхода из которого является либо достижение локального минимума, либо достижение максимума числа допустимых итераций.

Наиболее популярная подгруппа группы алгоритмов с физическим аналогом – **методы отжига**. Они выделяются тем, что «колебания» системы затухают с каждой итерацией. Название этой группы алгоритмов объясняется именно этой особенностью – затухания колебаний эквиваленты постепенному снижению температуры системы. В разделе 4 метод отжига рассматривается более подробно.

Для минимизации штрафной функции также можно использовать генетические алгоритмы. В работе [12] приводится более подробное описание такого подхода.

2.5.2. Аналитические алгоритмы

Аналитические алгоритмы, в отличие от алгоритмов с физическим аналогом представляют собой последовательность различных преобразований графа, приводящую к построению укладки. Это позволяет получать с их помощью гарантированный результат, удовлетворяющий выбранным критериям, так как критерии используются не для построения штрафной функции, а для построения самого алгоритма укладки, заметим, что на некоторых этапах аналитического алгоритма могут использоваться и псевдослучайные методы, в том числе методы минимизации функций, аналогичные методу отжига.

К недостаткам аналитических алгоритмов можно отнести сложность их построения. Кроме того, они плохо поддаются модификации, и для постановки экспериментов приходится вносить серьезные изменения в сам алгоритм, а не в набор параметров, как это можно делать в случае использования, например, алгоритмов, базирующихся на штрафных функциях.

На практике (рассмотрено три реализации: [13 – 15]) аналитические алгоритмы позволяют относительно быстро получать наиболее качественные и красивые графы. Поэтому для укладки диаграммы состояний в проекте *UniMod* было принято решение использовать эту группу алгоритмов в качестве основной.

Наиболее эффективным [16] в группе аналитических алгоритмов является алгоритм *GIOTTO* [8, 17]. В работе [18] приводится обоснование применения модификации алгоритма *GIOTTO* для решения задачи укладки диаграммы состояний. Основная часть работы [18] посвящена укладке реберных меток, однако данная проблема здесь затрагиваться не будет. В работе [8] изложены основы алгоритма *GIOTTO*. В настоящей работе рассматривается его реализация и адаптация к задаче укладки диаграммы состояний. Прежде чем перейти к собственным алгоритмам укладки, рассмотрим некоторые из существующих решений.

3. ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ

Задача укладки диаграммы состояний решается многими *CASE* средствами, поддерживающими язык *UML*. Рассмотрим наиболее популярные из них. Все примеры, приведенные в текущем разделе, построены для одной и той же диаграммы состояний.

3.1. Укладка диаграмм в *Rational Rose Professional J Edition*

В рассматриваемом продукте использован один алгоритм укладки (либо осуществляется автоматический выбор алгоритма). На рис. 4 приведена уложенная с его помощью диаграмма состояний. Эстетическое качество укладки неудовлетворительно: неоправданные пересечения переходов, прохождения переходов по состояниям, при наличии петель не удается обеспечить читаемость диаграммы. Возможно, использованный алгоритм более подходит для укладки диаграммы классов (подробнее об этом см. [19]).

3.2. Укладка диаграмм в *Borland Together Designer CE*

Данный продукт предоставляет ряд алгоритмов укладки диаграмм:

- иерархическая укладка;
- укладка деревьев;
- метод отжига;
- ортогональная укладка;
- оригинальный алгоритм «*Together*».

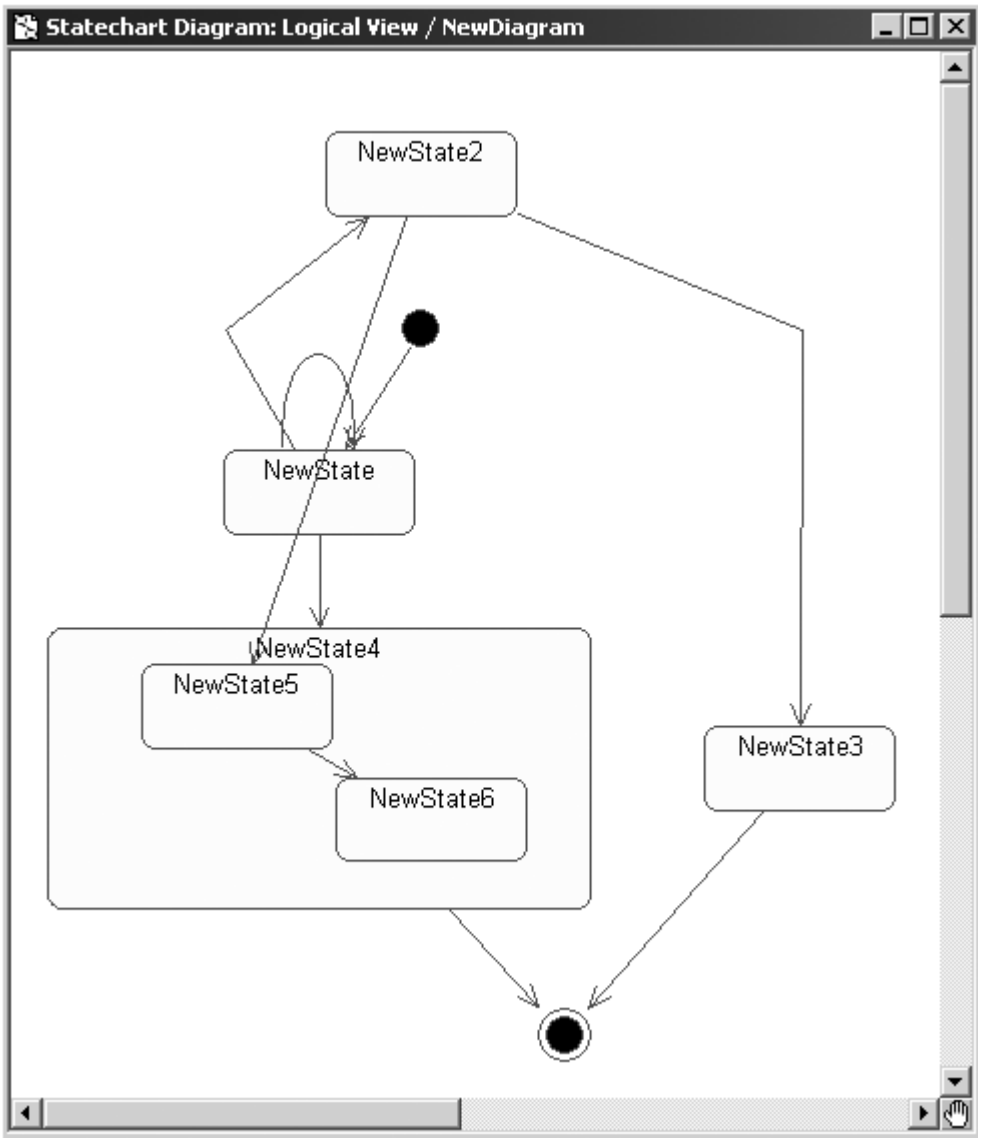

Рис. 4. Укладка в *Rational Rose*

Алгоритмы иерархической укладки и укладки деревьев могут быть отброшены без рассмотрения, как неподходящие для применения к диаграмме состояний. Рассмотрим оставшиеся алгоритмы.

Метод отжига, реализованный в рассматриваемом программном продукте, не представляет особого интереса (достаточно посмотреть на рис. 5). Укладка занимает неоправданно много места.

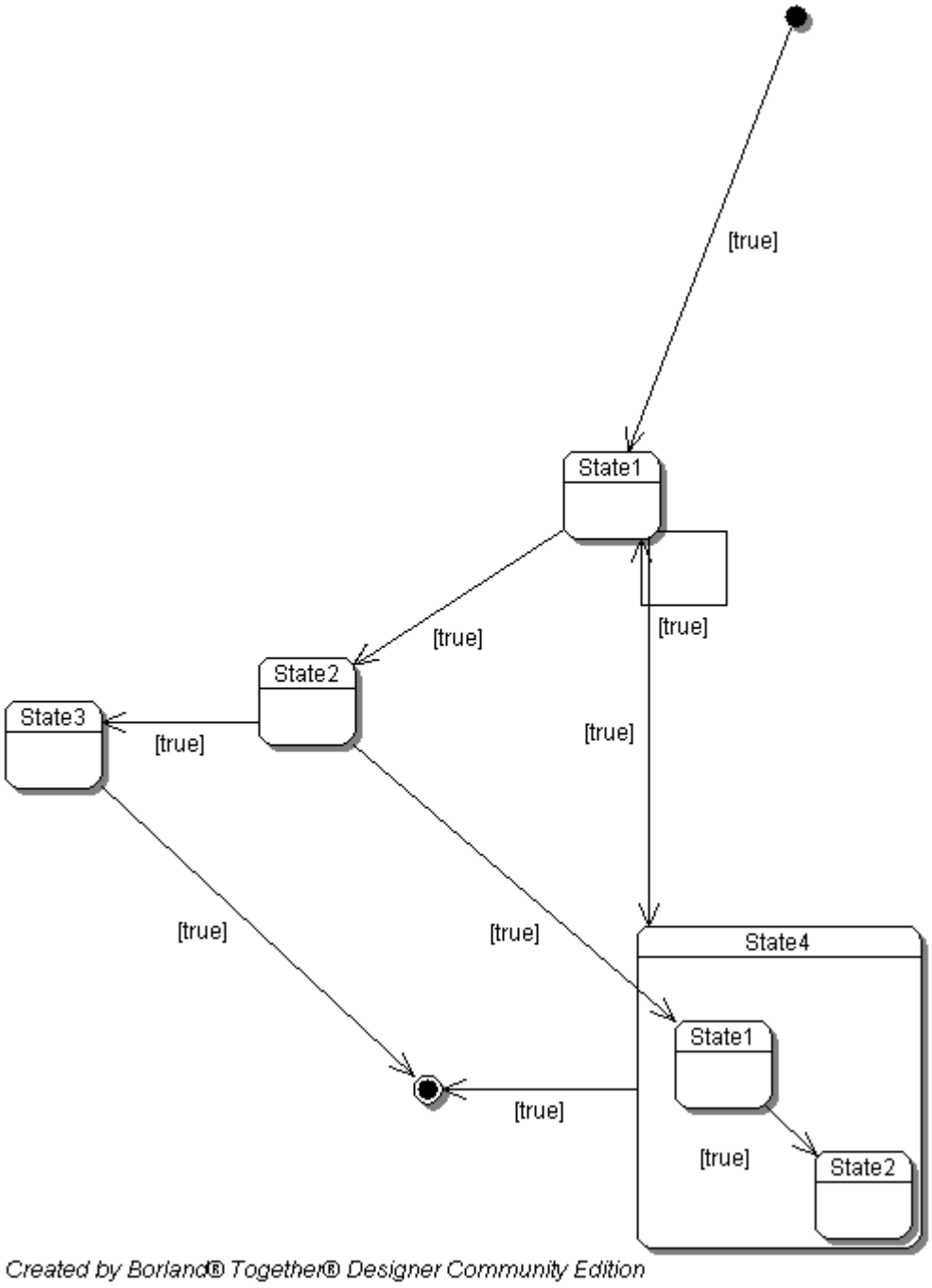

Рис. 5. Метод отжига

Алгоритм ортогональной укладки, вероятно, является модификацией простого алгоритма ортогонализации, притом достаточно неудачной. Переходы во вложенные состояния не удовлетворяют требования ортогональной укладки. Количество пересечений неоправданно велико (рис. 6).

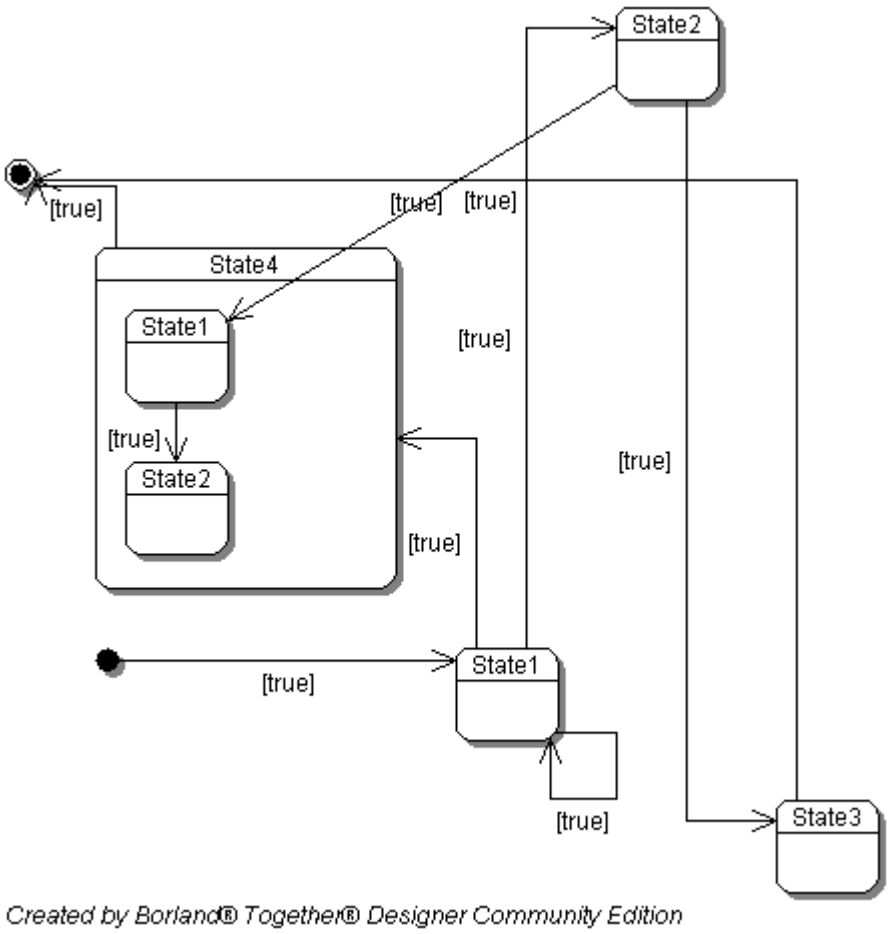

Рис. 6. Ортогональная укладка

На рис. 7 приведен результат укладки с использованием оригинального алгоритма *Together*. Последний осуществляет неортогональную укладку, достигая высокой компактности диаграммы. Эстетически укладки, построенные им, кажутся достаточно качественными. Информация о том, на базе каких идей построен алгоритм, компанией *Borland* не разглашается.

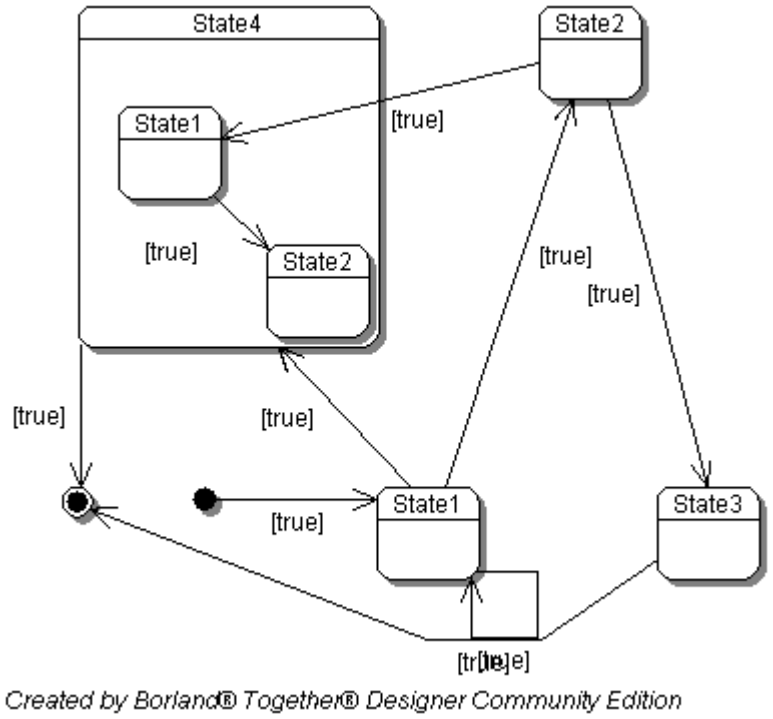

Рис. 7. Алгоритм "*Together*"

3.3. Укладка в *Sun Java Studio Enterprise 7*

В текущей версии данного программного продукта (версия “*2004 Q4*”) для диаграммы состояний доступен алгоритм ортогональной укладка (рис. 8), в отношении которого справедливо все сказанное выше про ортогональный алгоритм укладки в *Borland Together Designer CE.*

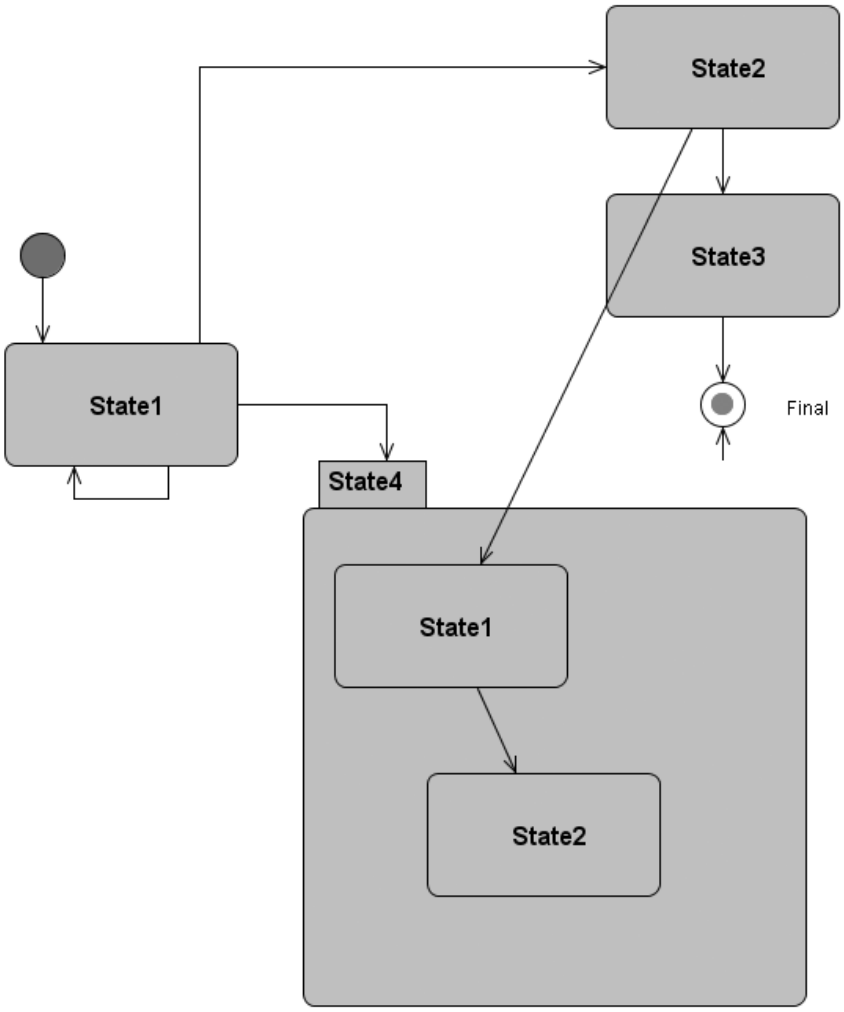

Рис. 8. Укладка в *Sun Java Studio Enterprise 7*

3.4. Укладка с помощью пакета *yFiles*

Комплекс алгоритмов укладки графов *yFiles* применен, например, в *UML*-редакторе *Poseidon for UML*. К сожалению, в некоммерческой версии этого продукта укладка диаграмм недоступна. Поэтому на рис. 9 приведена укладка графа, соответствующего диаграмме состояний использованной в остальных примерах, выполненная с помощью редактора графов *yEd*. Использованный алгоритм базируется на *GIOTTO* и модифицирован для наличия выделенных групп вершин (соответствуют суперсостояниям на диаграмме состояний). Компания-разработчик не разглашает идей, на которых основана такая модификация. Обратим внимание на то, что граф уложен без пересечений и укладка является ортогональной.

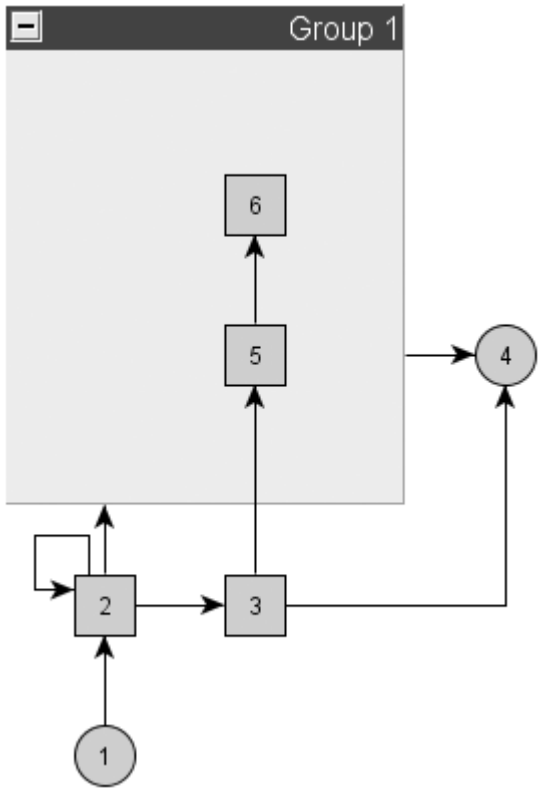

Рис. 9. Укладка с помощью *yFiles*

3.5. Укладка графа с помощью пакета *AGD*

Для укладки графов (без группировки вершин) разработан ряд пакетов, демонстрирующих возможности современных алгоритмов. На рис. 10 приведен результат укладки графа в программе *AGD* [13]. Укладка производилась с использованием алгоритма *GIOTTO*.

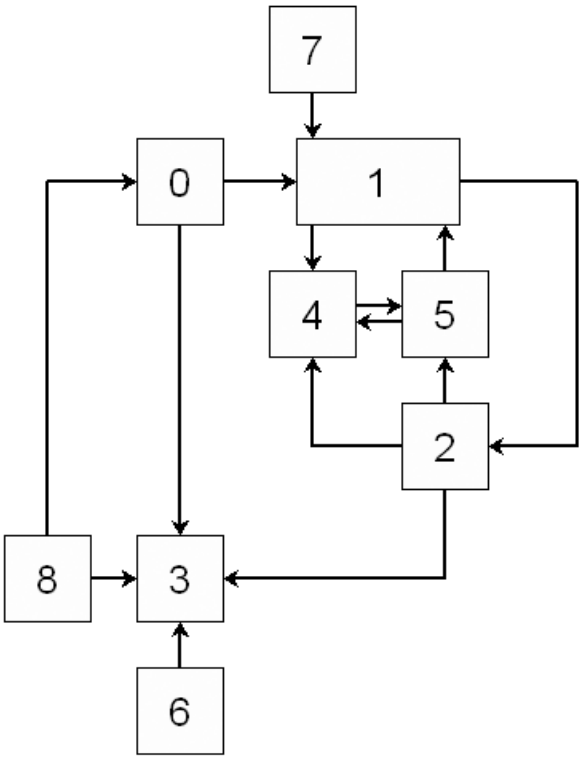

Рис. 10. Укладка с помощью *AGD*

3.6. Результаты обзора

Проведенный обзор показывает, что наиболее популярные на данный момент средства редактирования *UML* диаграмм не используют качественных алгоритмов укладки диаграмм состояний (и, если такие алгоритмы существуют, то они сами недостаточно популярны). С другой стороны, для укладки графов общего вида разработан целый ряд алгоритмов, которые строят хорошие укладки. Это означает, что адаптация алгоритмов укладки графов для решения задачи укладки диаграмм состояний является актуальной и практически значимой работой.

4. МЕТОД ОТЖИГА

Первый разработанный и реализованный автором алгоритм укладки является модификацией метода отжига с применением ортогонализации. Проходя по состояниям, этот алгоритм пытается сдвинуть каждое из них. Если производится сдвиг простого состояния, вложенного в составное состояние, то запрещается выход за пределы объемлющего составного состояния. После сдвига каждого состояния вычисляется новое значение штрафной функции. Если штрафная функция уменьшилась – новое расположение вершин фиксируется и производится сдвиг следующего состояния, если увеличилась – перемещение вершины отклоняется. Колебания системы затухают – расстояния, на которые сдвигаются состояния, уменьшаются на каждом шаге.

4.1. Построение штрафной функции

Для описания штрафной функции определим **диаметр графа** (*D*) – максимальный из его линейных размеров. У вершины на диаграмме задан размер по умолчанию (ширина и высота вершины). Обозначим максимальный из линейных размеров вершины по умолчанию как S_m. Каждая вершины при этом, имеет размер, возможно, превышающий размер по умолчанию. Большой размер имеют как вершины с большой инцидентностью, так и составные вершины (вершины, соответствующие составным состояниям).

В алгоритме используется вспомогательная координатная сетка с крупными ячейками (порядка минимального размера вершины). Вершины располагаются в точках, имеющих целочисленные координаты в крупной сетке (это улучшает качество диаграммы).

Штрафная функция составлена из слагаемых, приведенных в табл. 1.

Таблица 1. Слагаемые штрафной функции

Название	Метод вычисления слагаемого
Пересечение переходов	Каждое пересечение переходов учитывается с весом D*INTERSECT_MODIFIER. Наложение переходов также считается пересечением. Вес пропорционален диаметру графа, так как с ростом линейных размеров графа растет сумма штрафов за отклонения длин ребер и расстояний между состояниями от оптимальных значений. Таким образом, умножение на диаметр позволяет избежать конфликта между штрафом за пересечение ребер и штрафом за отклонение длины ребра от оптимальной.
Отклонение от оптимальной длины перехода	Отклонение длины перехода от заданной оптимальной длины (OPTIMAL_LENGTH) учитывается с весом Δ*LONGER_MODIFIER в случае превышения оптимальной длины и с весом Δ*SHORTER_MODIFIER в противном случае.
Пересечение вершин	Наложение вершин (кроме наложения простых вершин на составные, в которые они вложены) учитывается с весом D*OVERLAY_MODIFIER.
Отклонение от оптимального расстояния между	Отклонение расстояния между вершинами от заданного оптимального (OPTIMAL_DISTANCE) учитывается с весом Δ*FARTHER_MODIFIER в

вершинами	случае превышения и с весом Δ*CLOSER_MODIFIER в противном случае. Каждая пара вершин учитывается один раз.
Компактность диаграммы	Превышение расстояния до центра масс графа над корнем из количества вершин ($\sqrt{N}$) учитывается с весом $\Delta * \sqrt{N} * S_m$ *CENTER_MODIFIER. Данный критерий позволяет предотвратить «расползание» элементов диаграммы.
Прохождение перехода по вершине	Прохождение перехода по вершине учитывается с весом D*OL_MODIFIER*Angle. Так как в процессе ортогонализации переходы, которые исходно являлись наклонными, будут состоять из вертикальных и горизонтальных отрезков, то учитывается отклонение от вертикали или горизонтали (Angle) – нормированное на единицу угловое расстояние от наиболее близкой из координатных осей. Таким образом, предотвращается прохождение горизонтальных и вертикальных переходов по вершинам.

Зафиксируем оптимальные расстояния между вершинами и длины ребра (табл. 2).

Таблица 2. Значения оптимальных расстояний

Константа	Значение
OPTIMAL_DISTANCE	50
OPTIMAL_LENGTH	50

В ходе вычислительного эксперимента (производилась укладка диаграмм состояний, взятых из ряда проектов, в которых применялся пакет *UniMod*) были выбраны значения множителей при компонентах штрафной функции, приведенные в табл. 3. Эксперимент проводился следующим образом: производилась укладка диаграммы, взятой из проекта, в котором применялся пакет *UniMod*, затем визуально оценивался результат укладки и, в случае если результат оказывался неудовлетворительным, изучалось влияние различных констант, а затем производились соответствующие изменения их значений.

Таблица 3. Значения констант

Константа	Значение
INTERSECT_MODIFIER	1
OL_MODIFIER	5
OVERLAY_MODIFIER	50
SHORTER_MODIFIER	2
LONGER_MODIFIER	1
CLOSER_MODIFIER	0.5
FARTHER_MODIFIER	0.25
CENTER_MODIFIER	1

OPTIMAL_DISTANCE	50
OPTIMAL_LENGTH	50

При выборе и вычислении штрафной функции будем полагать, что ребро является отрезком, а не ломаной линией – используется другое геометрическое представление элементов. В результате требуется построить ортогональную укладку. Одно из решений – такое преобразование штрафной функции, чтобы она учитывала представление ребер. Проанализируем это решение: в ортогональной укладке есть дополнительные элементы – изломы ребер (эти точки имеют координаты), дополнительное жесткое ограничение (только горизонтальные и вертикальные отрезки) и, что не менее важно, параметр ребра – количество изломов. Кроме того, ребра начинаются в конкретных точках вершины (портах). Учет этих параметров в ходе минимизации штрафной функции может привести как к значительному увеличению вычислительной сложности алгоритма, так и к неоправданному усложнению укладки (например, к увеличению количества изломов). Например, укладки на рис. 11, отличаются лишь расположением портов.

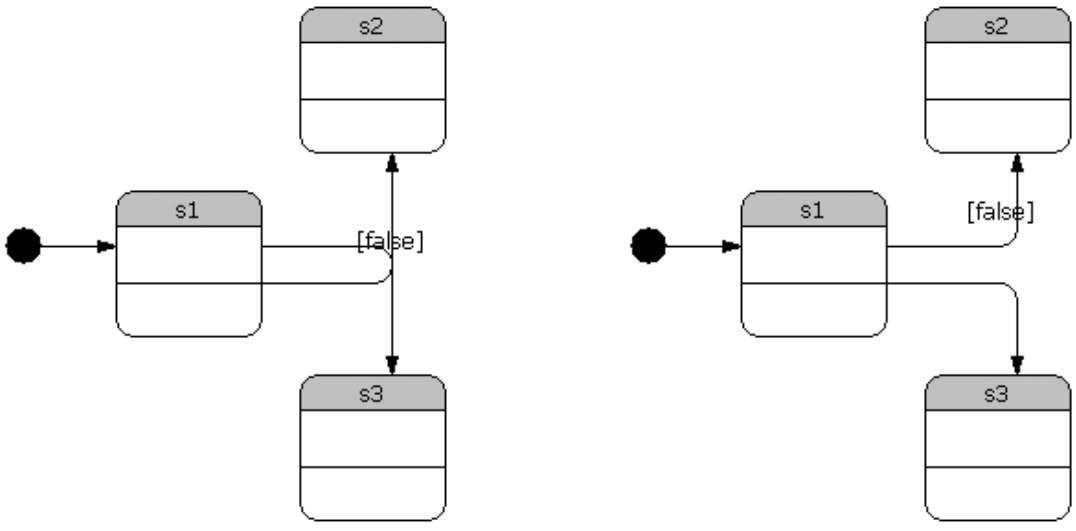

Рис. 11. Расположение портов

Рис. 11 является иллюстрацией того, что расположение портов (и изломов на ребрах) на диаграмме не менее важно, чем расположение вершин и ребер. Автором выбран следующий подход: построим первоначальную укладку с помощью алгоритма отжига, а затем ортогонализуем ее (вообще говоря, с помощью аналитического алгоритма). Получившийся алгоритм можно назвать смешанным. При этом в процессе ортогонализации могут появиться дополнительные пересечения ребер, не учтенные алгоритмом отжига, но их количество, как показывает практика, невелико.

4.2. Ортогонализация укладки

После нахождения некоторого расположения вершин с помощью метода отжига ребра укладываются так, чтобы каждое состояло только из горизонтальных и вертикальных отрезков. В укладке, которая строится в настоящей работе, ребро будет иметь не более одного излома. Алгоритм ортогонализации разработан автором и базируется на идеях, изложенных в [7, 8, 10].

Для нахождения окончательного расположения ребер автором разработан оригинальный алгоритм ортогонализации, состоящий из следующих шагов:

- распределение ребер по сторонам вершин;
- распространение петель по сторонам вершин;
- выделение **портов** (точек входа ребер в вершину) на сторонах вершин.

Опишем предложенный алгоритм. На вход алгоритм получает граф, в котором уже известны геометрические координаты вершин, но не известны ни координаты начала и конца каждого ребра (представлением вершины является прямоугольник, поэтому соответствующие координаты невозможно получить автоматически), ни координаты изломов на ребрах. На выходе алгоритма – укладка графа.

4.2.1. Распределение ребер по сторонам вершин

Если две вершины можно соединить горизонтальным или вертикальным отрезком, то в окончательной укладке их будет соединять прямое ребро (стороны, в которые оно входит, определяются однозначно). В противном случае необходимо выбрать «маршрут» для ребра. Для каждого (непрямого) ребра возможно не более двух вариантов его проведения (маршрутов), как показано на рис. 12 (именно ради упрощения этой части алгоритма было введено ограничение на количество изломов на ребре).

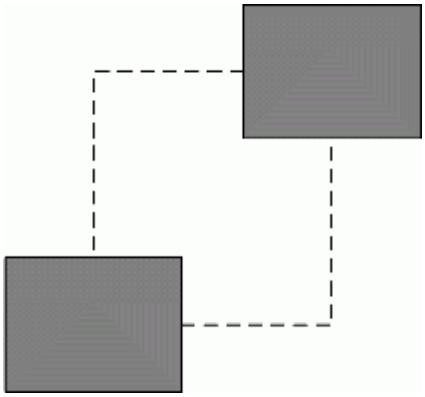

Рис. 12. Варианты проведения ребра

Будем действовать следующим образом:

- определим ребра, для которых известно, какой стороне они принадлежат (какой стороне принадлежит соответствующий порт). Это ребра, маршрут которых уже определен при обработке других вершин, и ребра, соединяющие вершины, лежащие на одной линии;
- определим текущую инцидентность каждой стороны (количество ребер, входящих в эту сторону или исходящих из нее);
- переберем все ребра, для которых есть два варианта их расположения (рис. 12). Каждое из них проводим с той стороны вершины, для которой текущая инцидентность минимальна.

Применим описанный алгоритм к вершинам (обходя их в порядке убывания инцидентности). Таким образом, для каждой вершины определим, с какой ее стороны находятся точки выхода ребер.

4.2.2. Распределение петель по сторонам вершин

Распределим петли по сторонам с минимальной набранной инцидентностью (петлю можно поместить на любой стороне).

4.2.3. Выделение портов

Теперь отсортируем ребра, инцидентные каждой из сторон вершины так, чтобы, по возможности, уменьшить количество пересечений. Рассмотрим множество таких ребер, инцидентных левой стороне вершины *A*. В нижней части рассматриваемой стороны вершины будут расположены порты ребер, ответные вершины которых располагаются ниже вершины *A*. Затем – порты горизонтальных ребер (разд. 4.2.1), а затем – порты таких ребер, что их ответные вершины находятся слева вверху от вершины *A*.

Рассмотрим последнюю группу ребер. Чем ближе по горизонтали находится ответная вершина к вершине *A*, тем выше будет расположен соответствующий порт. Для других относительных расположений рассматриваемой вершины и ответной вершины алгоритм аналогичен (рис. 13).

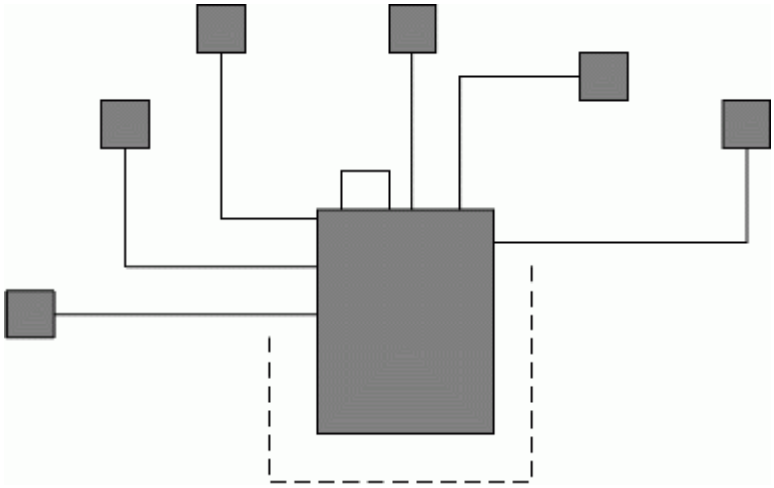

Рис. 13. Сортировка ребер.

Пусть задана диаграмма состояний (рис. 14). На рис. 15 приведена диаграмма состояний, полученная из нее с помощью алгоритма отжига с последующей ортогонализацией, а на рис. 16 та же диаграмма, преобразованная вручную.

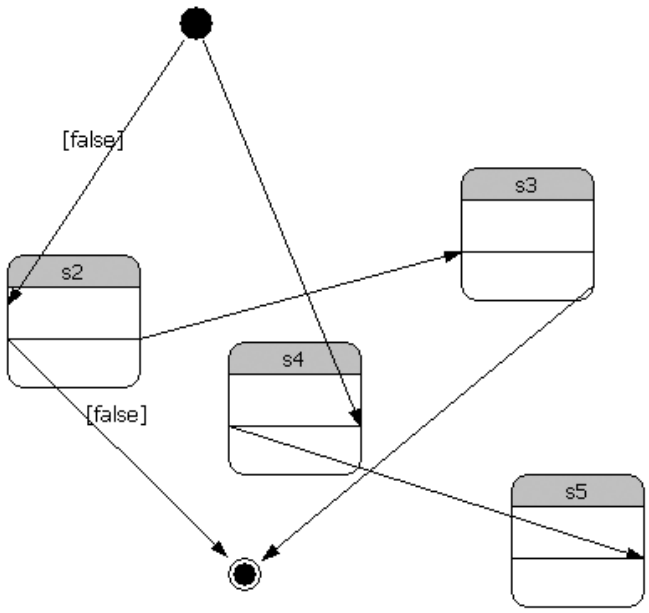

Рис. 14. Исходная диаграмма

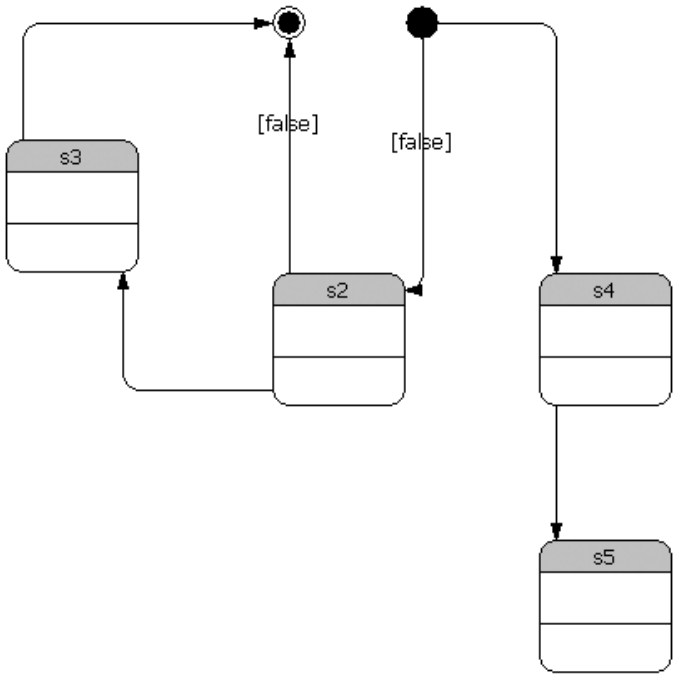

Рис. 15. Уложенная диаграмма

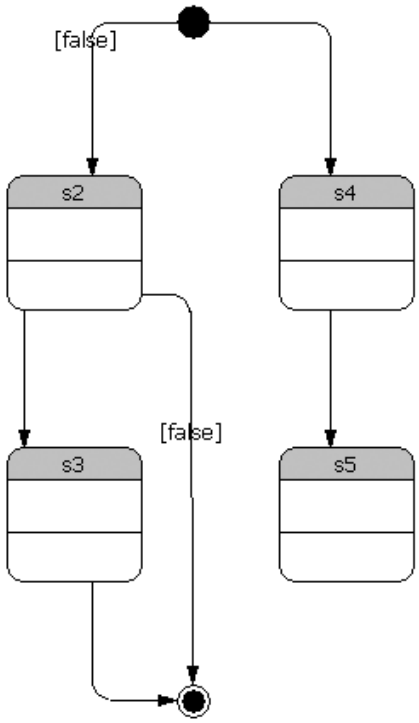

Рис. 16. Диаграмма, преобразованная вручную

Обратим внимание, что на рис. 15 начальная и конечная вершины находятся на небольшом расстоянии друг от друга, что противоречит одному из критериев, выдвинутых для диаграммы состояний. Вызвано несоответствие тем, что сейчас этот критерий не учитывается в штрафной функции. Учет всех критериев обычно невозможен, поскольку приводит к конфликтам между ними и требует неоправданного усложнения алгоритма [7].

4.3. Реализация

В рамках проекта *UniMod* автором реализован изложенный выше метод отжига с применением ортогонализации. Эта реализация вошла в *UniMod Release 1 Build 07*. Затем от данного алгоритма отказались в пользу одного из аналитических алгоритмов. Причины отказа изложены в разделе 4.4.

4.4. Выводы

Штрафная функция в данной ситуации вычисляется за время асимптотически не медленнее, чем $O(E^2)$, где E – количество ребер в графе (константа при оценке велика, так как последовательно вычисляется несколько различных слагаемых штрафной функции). Для графов среднего размера (порядка 50 вершин) каждая вершина, как показала экспериментальная проверка (разд. 4.1), должна испытать не менее 15-30 возмущений для достижения приемлемого качества укладки, что дает время работы алгоритма $O(VE^2)$, но с большой константой при асимптотической оценке. На больших графах время работы алгоритма неудовлетворительно.

Другая проблема заключается в сильной неопределенности результата (выражается в том, что качество укладки значительно изменяется от запуска к запуску алгоритма). Последнее обстоятельство вызывается наличием у штрафной функции большого количества локальных минимумов.

Из изложенного следует (и этот результат подтверждается экспериментально), что алгоритм, основанный на методе отжига, хорошо работает лишь на несложных диаграммах, с ростом же количества вершин снижается качество укладки (вследствие попадания штрафной функции в окрестность одного из локальных минимумов) и увеличивается время работы алгоритма.

5. МОДИФИЦИРОВАННЫЙ АЛГОРИТМ *GIOTTO*

Следующий разработанный и реализованный алгоритм ортогональной укладки диаграмм состояний *UML* является аналитическим и базируется на алгоритме *GIOTTO* [8]. Перечислим основные шаги алгоритма:

1. Преобразование графа в связный (разд. 5.1);
2. Преобразование графа в двусвязный (разд. 5.2);
3. Выделение планарного подграфа (разд. 5.3);
4. Построение *st*-графа (разд. 5.4);
5. Построение планарного «надграфа» (разд. 5.5);
6. Разбиение вершин инцидентностью больше четырех на цепочки (разд. 5.6);
7. Построение ортогонального представления (разд. 5.8);
8. Минимизация площади, занимаемой уложенным графом (разд. 5.9);
9. Удаление фиктивных элементов, созданных в процессе предыдущих шагов;
10. Восстановление информации об ориентации ребер (информация «теряется» на четвертом шаге).

Опишем подробно каждый шаг алгоритма.

5.1. Построение связного графа

Алгоритм укладки работает только с двусвязными графами (переход к двусвязному графу будет описан в разделе 5.2). **Двусвязный граф** (**дограф**) остается связным после удаления любого ребра. Для того чтобы сделать граф связным, найдем его компоненты связности [20] и соединим любые их вершины мостами, как показано на рис. 18. В исходном графе на рис. 17 три компоненты связности. Добавленные в ходе работы алгоритма ребра выделены на рис. 18 пунктиром. Для графа с *N* компонентами связности будет добавлено *N-1* фиктивное ребро.

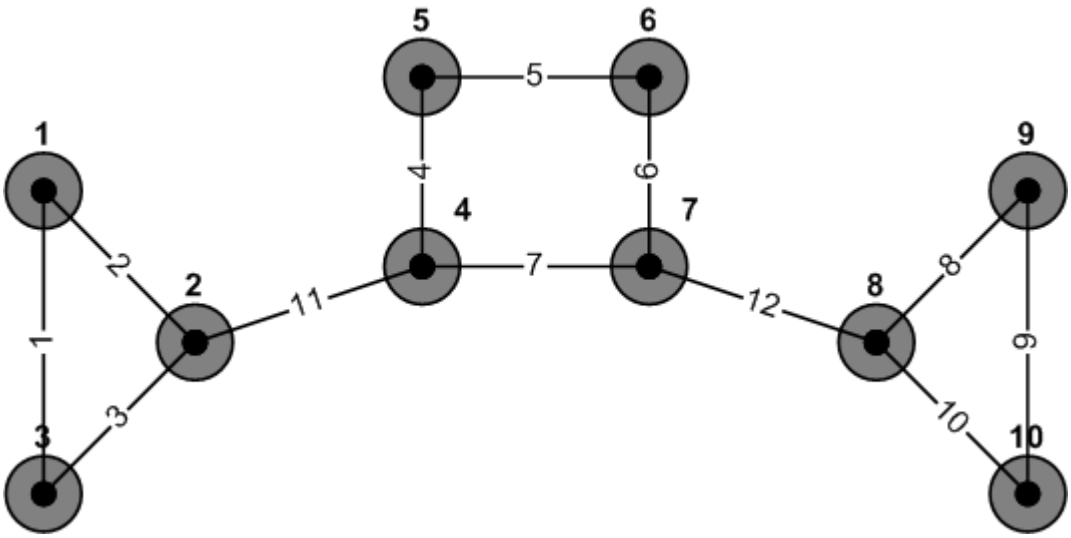

Рис. 17. Исходный граф

В рамках рассматриваемой задачи, можно также учесть, что граф переходов конечного автомата не может являться несвязным (с другой стороны, реализация этой части алгоритма необходима для обобщения его на другие задачи).

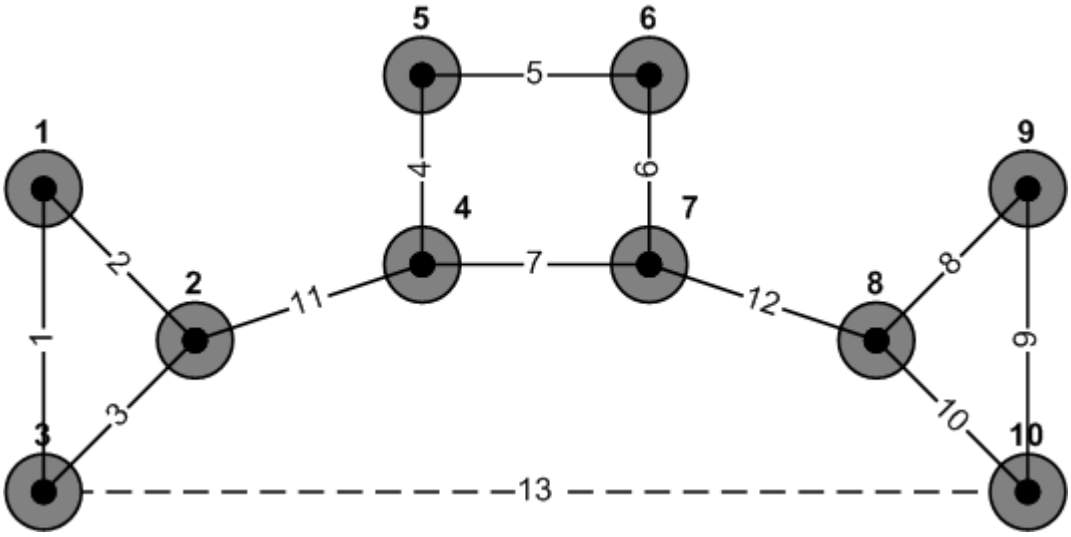

Рис. 18. Построение связного графа

5.2. Построение двусвязного графа

В работе [21] описан ряд алгоритмов построения двусвязных графов. Воспользуемся следующим алгоритмом: выделим компоненты двусвязности, построим дерево таких компонент и прошьем его крону циклом (рассматривая двусвязные компоненты как отдельные вершины). Приведем более строгое описание этого алгоритма:

- возьмем связный граф G;
- выделим в нем двусвязные компоненты (**блоки**), точки сочленения и мосты. В графе на рис. 18 вершины v_2, v_4, v_7, v_8 – точки сочленения, а ребра e_{11}, e_{12} – мосты. Граф содержит три блока: $x_1 = \{v_1, v_2, v_3\}$, $x_2 = \{v_4, v_5, v_6, v_7\}$, $x_3 = \{v_8, v_9, v_{10}\}$;

- построим **граф блоков** *B(G)*. Вершины этого графа соответствуют блокам, мостам и точкам сочленения исходного графа. Ребра соединяют блоки принадлежащими им точками сочленения или мостами, вершины которых принадлежат блоку. Иными словами, в графе блоков существует ребро $e = (v, v')$, если:

  v соответствует точке v_c сочленения в графе G, v' – блоку x, содержащему данную точку сочленения;

  v соответствует точке v_c сочленения в графе G, v' – мосту e, причем v_c – одна из вершин e.

Граф блоков для рассматриваемого примера приведен на рис. 19.

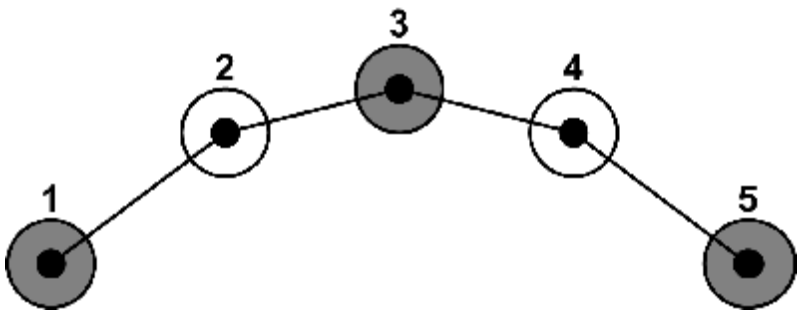

Рис. 19. Граф блоков

Вершины v_1, v_3, v_5 соответствуют блокам x_1, x_2, x_3, *а* v_2, v_4 – мостам e_{11}, e_{12}. Заметим, что граф блоков всегда является свободным деревом [22]. Выделим все висячие вершины и соединим их в цепочку (как и в алгоритме построения связного графа, для N вершин понадобится $N-1$ ребро). В рассматриваемом примере будет добавлено одно ребро (рис. 20, на этом рисунке фиктивное ребро выделено пунктиром).

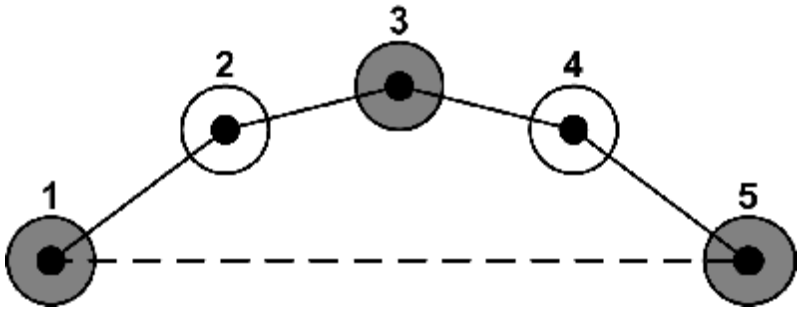

Рис. 20. Граф блоков после преобразования

Вершинам степени 1 всегда соответствуют блоки в исходном графе, причем в каждом таком блоке найдется хотя бы одна вершина, не являющаяся точкой сочленения. Обозначим такую вершину $B^{-1}(v)$, где v – вершина в графе блоков. Для каждого фиктивного ребра $e = (v_1, v_2)$, добавленного в граф блоков, добавим ребро $e' = (B^{-1}(v_1), B^{-1}(v_2))$ в исходный граф. В результате получим граф, изображенный на рис. 21. Пунктиром выделено ребро, добавленное на этапе построения двусвязного графа.

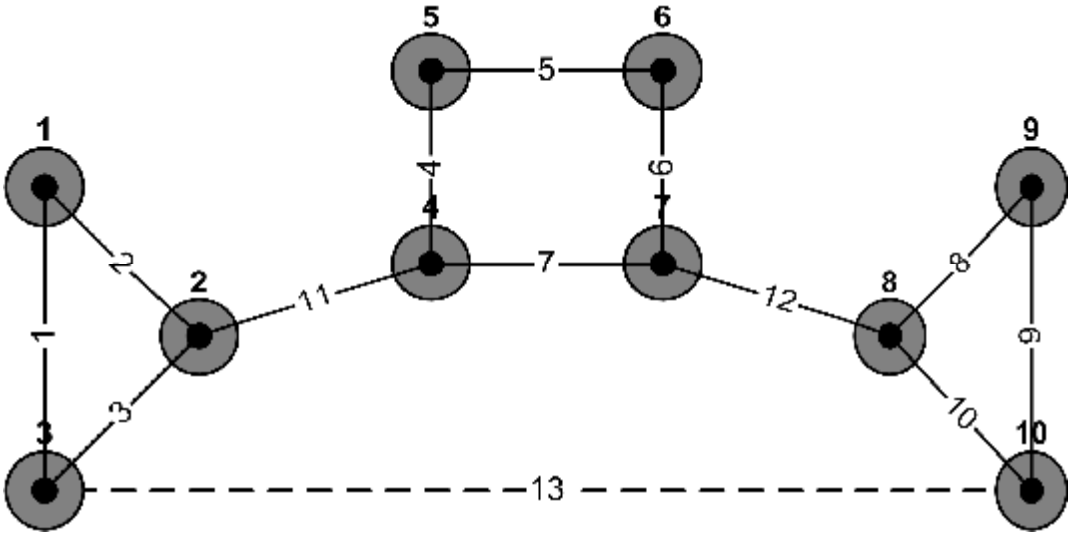

Рис. 21. Двусвязный граф

5.3. Выделение планарного подграфа

Для выделения планарного подграфа воспользуемся алгоритмом, приведенным в работе [23]. Далее введем некоторые термины и опишем алгоритм тестирования планарности, поскольку он важен для понимания ключевых понятий укладывания – понятий **грани** и **протограни**. Граф с выделенными гранями показан на рис. 22.

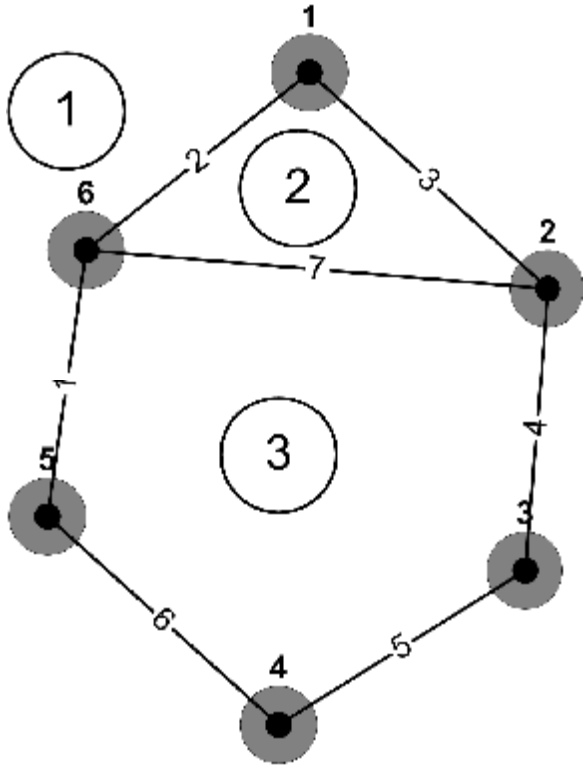

Рис. 22. Выделение граней

Большинство терминов, используемых в работе, являются общепринятыми [24], но нам потребуется ввести ряд понятий, специфических для решаемой задачи.

Гранью планарного уложенного графа будем называть максимальное множество точек плоскости, каждая пара которых может быть соединена непрерывной линицей, не пересекающей ребра графа. Грань, содержащую бесконечную по площади область, будем называть **внешней**.

Минимальный подграф, ребра которого ограничивают грань, называется **протогранью** (грань порождается протогранью). К примеру, на рис. 22, подграф ($\{v_1, v_2, v_6\}; \{e_2, e_3, e_7\}$) является протогранью грани 2. При построении укладки строится грани по протограням. На этапе выделения протограней координаты вершин неизвестны, но выделение всех протограней задает их относительное расположение.

Рассмотрим произвольный подграф G' двусвязного графа G. Этот подграф позволяет разбить весь граф (кроме ребер самого G' и, возможно, некоторых его вершин) на подграфы – **сегменты** (другое популярное название – **части**), удовлетворяющие следующим условиям:

- для каждого сегмента S любые две его вершины v_1, v_2 соединяет простой путь, не проходящий по вершинам подграфа G' (кроме, возможно, самих v_1, v_2);
- для любых двух сегментов S_1, S_2 и различных вершин $v_1 \in S_1$, $v_2 \in S_2$ не существует простого пути, не проходящего по вершинам G';
- каждый сегмент включает как минимум одно ребро.

На рис. 23 показан граф, с выделенным подграфом G' (его ребра обозначены пунктиром).

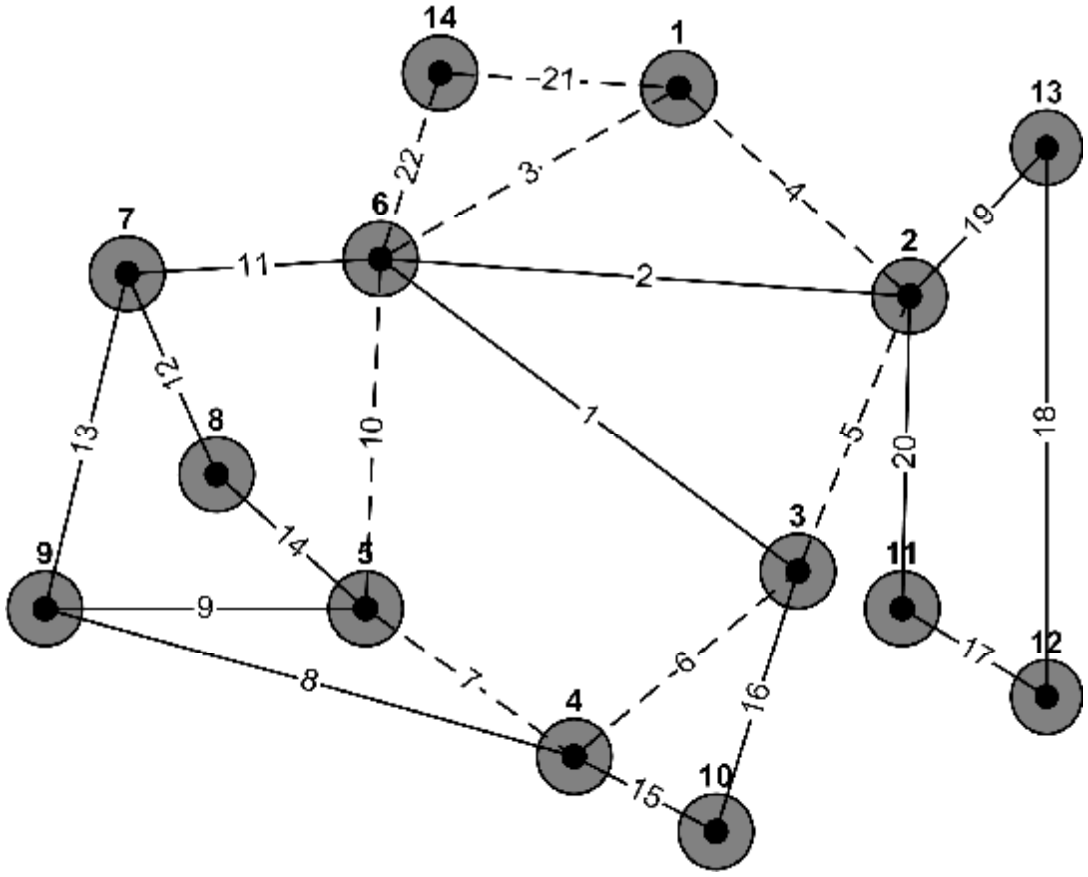

Рис. 23. Выделение сегментов

Этот подграф порождает пять сегментов:

- $S_1 = (\{v_2, v_6\}; \{e_2\})$;
- $S_2 = (\{v_3, v_6\}; \{e_1\})$;
- $S_3 = (\{v_3, v_4, v_{10}\}; \{e_{15}, e_{16}\})$;
- $S_4 = (\{v_4, v_5, v_6, v_7, v_8, v_9\}; \{e_8, e_9, e_{11}, e_{12}, e_{13}, e_{14}\})$;
- $S_5 = (\{v_2, v_{11}, v_{12}, v_{13}\}; \{e_{17}, e_{18}, e_{19}, e_{20}\})$.

Сегменты бывают двух видов:

- являющиеся графом K_2 – состоящие из двух вершин подграфа G' и одного не принадлежащего G' ребра (в нашем примере – S_1, S_2);

- включающие в себя как вершины подграфа *G'*, так и вершины, не принадлежащие ему (в нашем примере – S_3, S_4, S_5).

Вершины из сегмента, принадлежащие подграфу, назовем **контактными** (иногда их еще называют **внешними** по отношению к сегменту). Все вершины сегмента S_1 являются контактными, только вершина v_2 является контактной в сегменте S_5.

Подграф *G'* назовем **разделительным** для сегмента, если он содержит как минимум две контактные вершины этого сегмента. В нашем примере *G'* является разделительным для всех сегментов, кроме S_5.

5.3.1. Алгоритм тестирования планарности

Общая идея алгоритма тестирования планарности (его часто называют **гамма-алгоритмом** [25]) следующая: выберем первую протогрань (произвольный цикл). На каждом шаге имеем подграф *G'*, протограни в котором уже выделены. Добавим к нему путь, проходящий по *G**G'* и соединяющий две вершины из *G'*.

Рассмотрим граф *G* и его подграф *G'*, для которого уже выделены протограни.

На рис. 24 для подграфа *G'* (его ребра проведены пунктиром) выделены три протограни (обозначим их F_1, F_2, F_3). Сегменты относительно *G'*:

- $S_1 = (\{v_2, v_3, v_4\}; \{e_2, e_3\});$
- $S_2 = (\{v_1, v_2, v_4, v_5\}; \{e_5, e_6, e_8\}).$

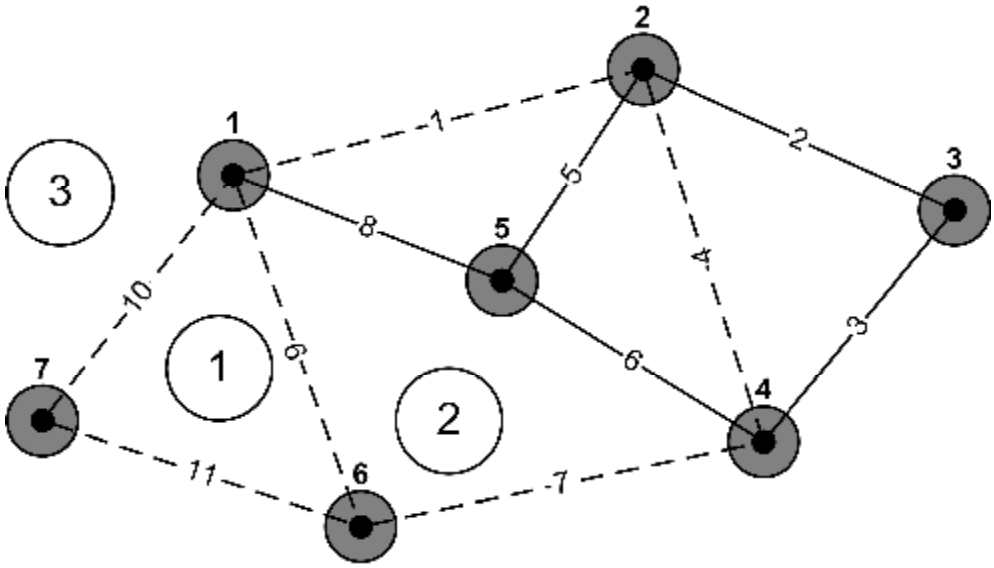

Рис. 24. Допустимые грани

Протограпь назовем **допустимой** для сегмента, если все его контактные вершины ей принадлежат. На рис. 24 для обоих сегментов допустимы протограни F_2 и F_3 (внешняя).

Приведем алгоритм выделения протограней:

- инициализация: выделим произвольный цикл, добавим его в подграф G' и назначим первой протогранью;
- шаг алгоритма: найдем все сегменты относительно подграфа G'. Если найдется сегмент с пустым множеством допустимых протограней, то граф не планарный. Если найдется сегмент ровно с одной допустимой протогранью, то выделим a-цепь и добавим ее в эту протограпь. В противном случае выделим a-цепь в любом из сегментов и добавим в одну из допустимых протограней. Добавляемая a-цепь разбивает протограпь на две;
- завершение алгоритма: $G = G'$.

В графе на рис. 24 нет сегмента ровно с одной допустимой гранью. Возьмем *a* - **цепь** (простой путь в сегменте между двумя различными контактными вершинами), проходящую по вершинам v_1, v_5, v_4 и добавим ее в протограни F_2, разделив последнюю на две протограни. В результате протограни F_2 изменилась и появилась протограни F_4 (рис. 25).

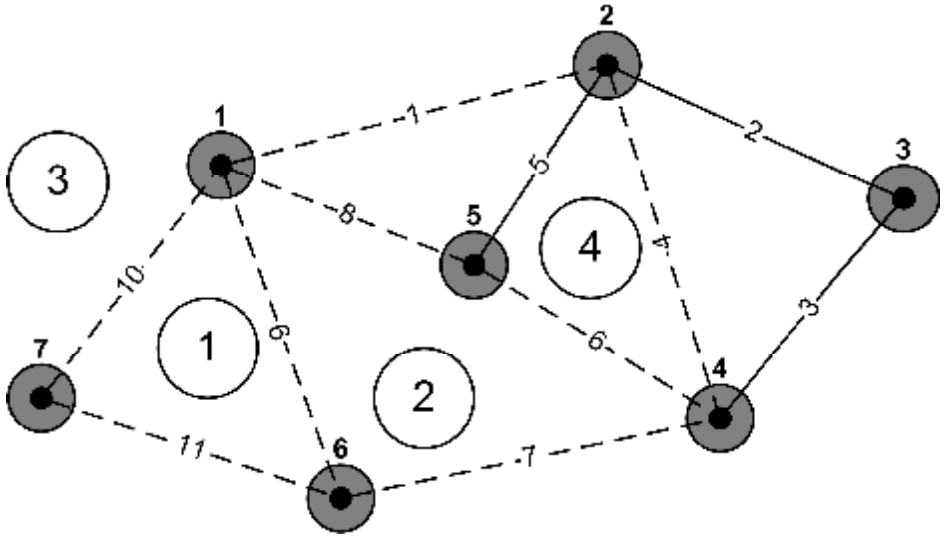

Рис. 25. Выделение протограней

Для сегмента $S' = (\{v_3, v_5\}, \{e_5\})$ ровно одна протограни является допустимой (F_4), кроме того весь этот сегмент представляет собой *a* -цепь. Таким образом, следующий шаг алгоритма приводит к результату, показанному на рис. 26.

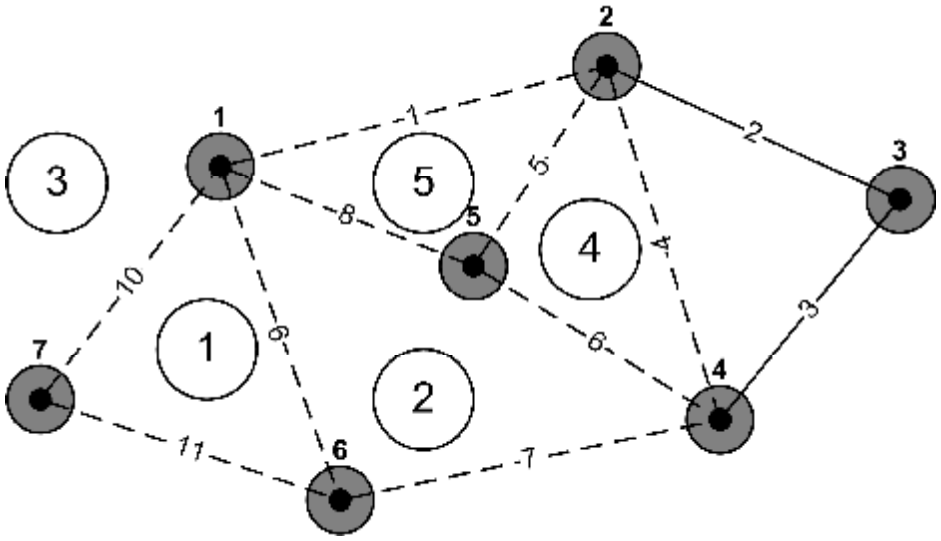

Рис. 26. Выделение протограней. Шаг 2

В графе на рис. 26 существует единственный сегмент $S_1 = (\{v_2, v_3, v_4\}; \{e_2, e_3\})$, для которого допустимы протограни F_3, F_4. Весь этот сегмент является a -цепью. Его добавление (например, в протограни F_4) завершает работу алгоритма (рис. 27).

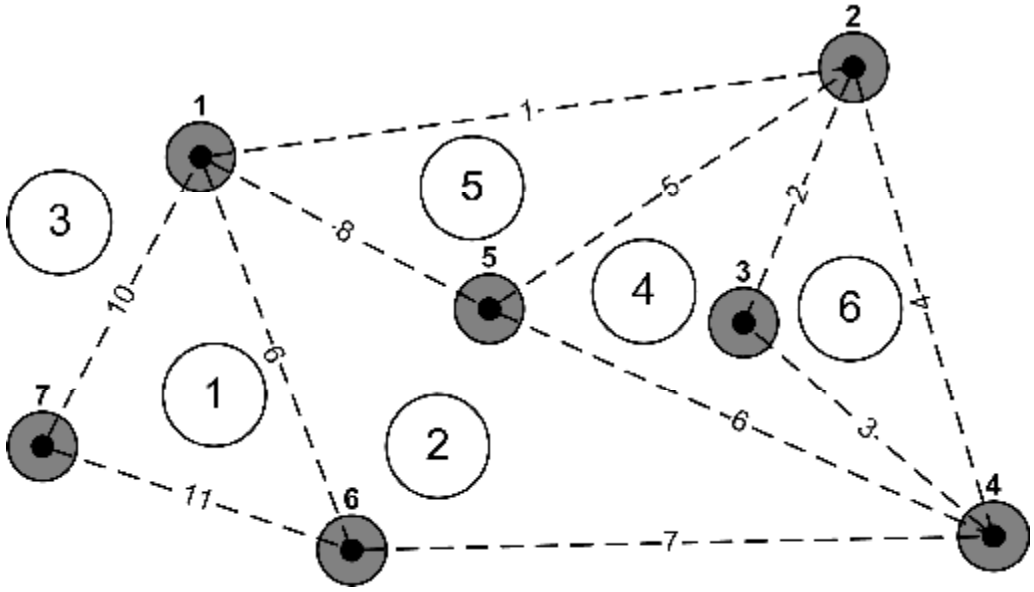

Рис. 27. Выделение протограней. Шаг 3

Обратим внимание на то, что на этом этапе вершинам графа не присвоены координаты, и выделение задает лишь границы граней, но не их расположение (обратите внимание на положение вершины v_3 на рис. 26, рис. 27).

В случае если граф не является планарным, выделим его планарный подграф, по возможности близкий к максимальному. Для этого существует ряд алгоритмов [23].

5.4. Построение *ST*-графа

Алгоритм первоначального расположения вершин на плоскости с помощью представления видимости работает только с *st*-графами. ***St*-граф** – это ориентированный дираф, в котором для выделенного стока *t* (*target*) существуют только входящие ребра, для выделенного источника *s (source)* – только исходящие, а для остальных вершин – обязательно существуют входящие и исходящие ребра. Необходимо в полученном на вход диграфе переориентировать ребра так, чтобы он стал *st*-графом (*st*-ориентировать дираф).

Эта задача эквивалентна задаче о построении *st*-нумерации. *St*-нумерацией называется такая нумерация вершин, что любая вершина с номером не равным максимальному или минимальному имеет инцидентные вершины с большими и меньшими номерами, номера вершин совпадать не могут. Для решения этой задачи применим алгоритм, описанный в работе [26].

Пример *st*-графа приведен на рис. 28.

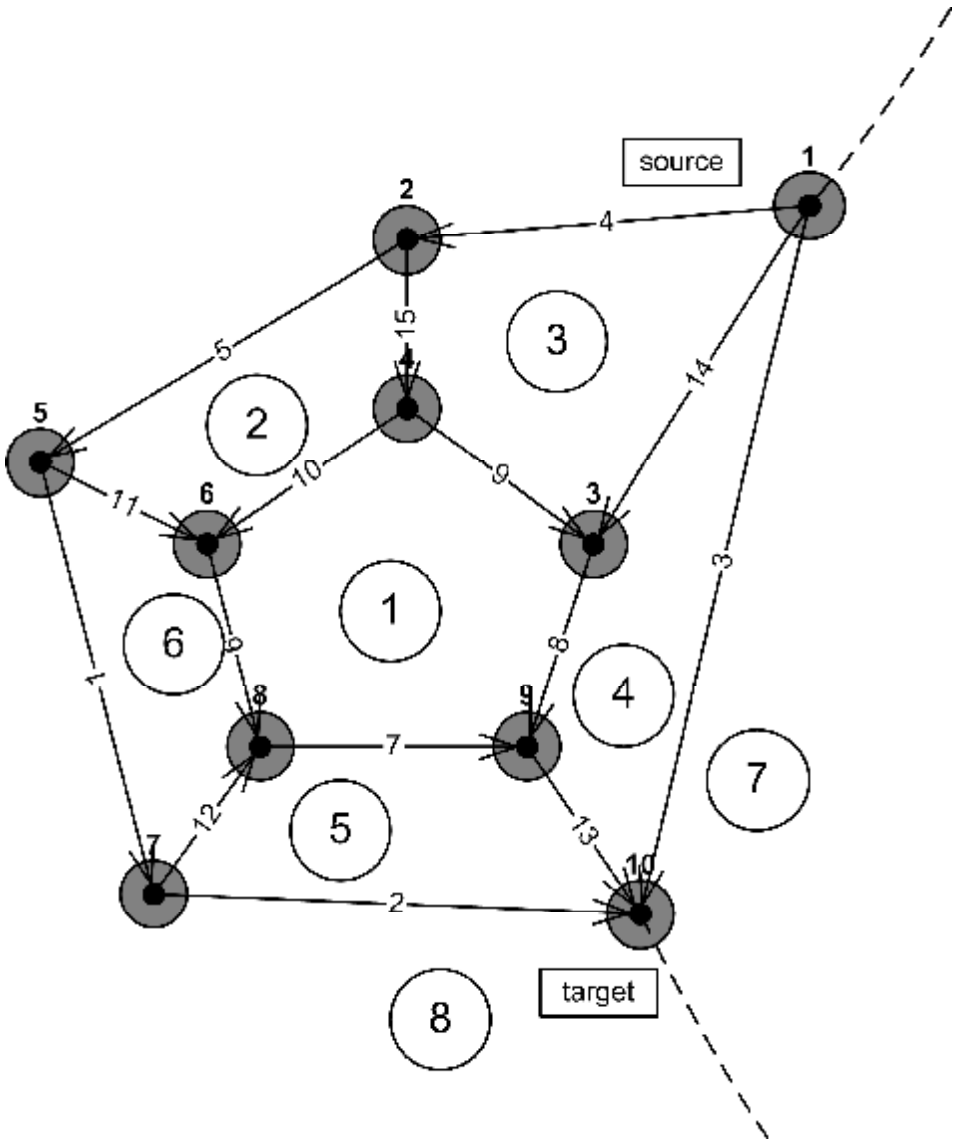

Рис. 28. St-граф.

5.4.1. Свойства *left, right, orig, dest*.

Каждая протогрань *st*-графа является *st*-графом (см. [26]). В *st*-графе каждой вершине *v*, ребру *e* и грани *F* сопоставлены четыре элемента: левая и правая протограни ***left*** и ***right***, локальные источник и сток ***orig*** и ***dest***.

Внешняя протогрань *st*-графа разделяется на две фиктивные протограни, порожденные двумя путями от источника к стоку, проходящими по внешней протограни. Назовем их левой и правой внешними протогранями. Заметим, что эти протограни порождают одну грань в изображении графа. Опишем алгоритм выделения этих свойств:

- для ребра e выделим $left(e)$ и $right(e)$ – протограни, находящиеся по левую и правую сторону[1] от ребра соответственно. На рис. 28 для ребра e_1 левой протогранью является грань F_6, а правой – F_8;
- для протограни F найдем источник $orig(F)$ и сток $dest(F)$ – вершины с нулевой входящей и исходящей инцидентностью (рассматриваются только ребра этой грани). Локальными источником и стоком внешних протограней, являются вершины s и t (глобальные источник и сток). Для протограни F_1 вершина v_{10} – локальный источник, а v_8 – локальный сток. Доказательство того, что для st-графа локальные источники и стоки протограней определяются однозначно [8];
- для вершины v положим $orig(v) = dest(v) = v$;
- для грани F положи $left(F) = right(F) = F$;
- для вершины v элементы $left(v)$ и $right(v)$ – правая и левая протограни относительно вершины, причем такие, что среди их ребер, инцидентных рассматриваемой вершине, есть как входящие, так и исходящие (рис. 29).

---

1 На этом этапе координаты вершинам не присвоены, но выделение протограней уже произведено, поэтому понятия «левый» и «правый» отвечают за относительное расположение и рассматриваются с точностью до переобозначения.

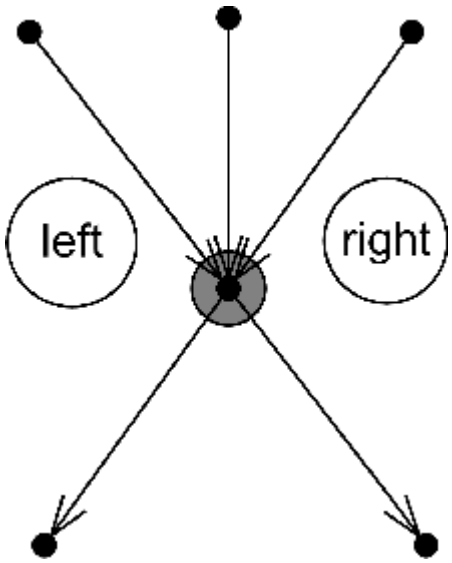

Рис. 29. Левая и правая протограни вершины

5.5. Построение планарного «надграфа»

После построения планарного подграфа и ориентации в нем ребер (так, чтобы он стал *st*-графом) воспользуемся алгоритмом, описанным в работе [8], для добавления оставшихся ребер и фиктивных вершин так, чтобы свойство планарности не нарушилось.

Опишем построение **ассоциированного графа**. Каждой протограни исходного графа *G* в ассоциированном графе *AG* соответствует вершина, для каждого ребра $e \neq (s,t)$ из исходного графа в *AG* существует ребро *e* = (left(e),right(e))*. Пример графа и его ассоциированного диграфа приведен на рис. 30.

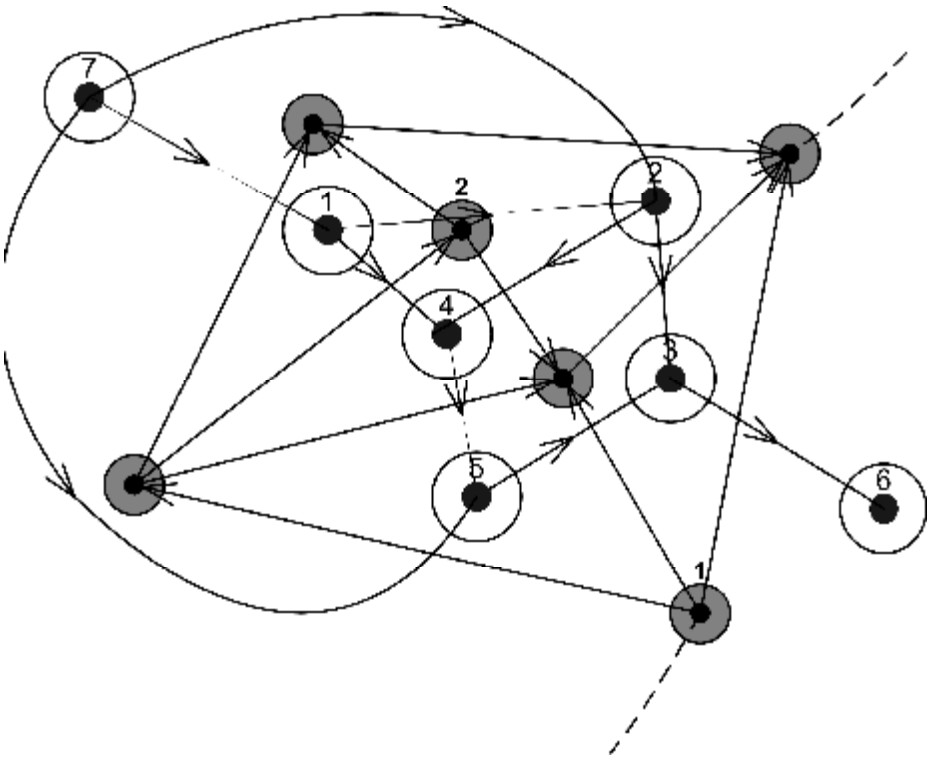

Рис. 30. Ассоциированный граф

Построим ассоциированный граф *AG'* для планарного подграфа *G'*. Выберем ребро *e*, не вошедшее в планарный подграф. Это ребро необходимо добавить. Допустим, что необходимо добавить ребро, соединяющее вершины v_1 и v_2 (рис. 30).

Определим множества протограней, которым принадлежит каждая вершина. Для этих множеств найдем сопоставленные им множества протовершин в ассоциированном графе (пусть это будут множества $P(v_1)$ и $P(v_2)$). В рассматриваемом примере $P(v_1) = \{F_3, F_5, F_6, F_7\}$, *а* $P(v_2) = \{F_1, F_2, F_4\}$.

Найдем кратчайший путь p' из $P(v_1)$ в $P(v_2)$ в ассоциированном графе. Каждой вершине из p' соответствует протогрань в исходном графе. Проходя по пути p' рассматриваем пары протограней, соответствующих соседним вершинам в ассоциированном графе. Эти протограни имеют общие ребра. Создаем фиктивные вершины на этих ребрах и «разделяем» протограни. В нашем случае один из кратчайших путей состоит из единственного ребра, соединяющего F_4 и F_5, а другой – F_2 и F_3. Таким образом, ребро $e = (v_1, v_2)$ было добавлено (рис. 31).

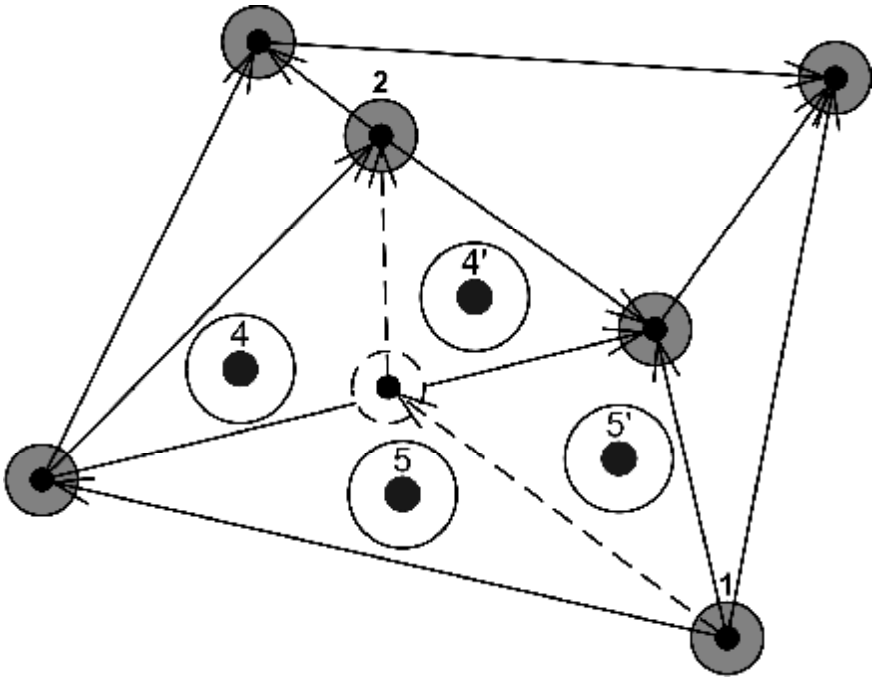

Рис. 31. Добавление непланарных ребер

В граф добавилась одна фиктивная вершина (которая станет точкой пересечения ребер), два ребра (будут заменены на одно), протограни F_4 и F_5 были разделены на пары протограней (добавлены протограни $F_{4'}$ и $F_{5'}$). Свойство *st*-ориентированности при таком добавлении вершин и ребер не нарушается [8].

5.6. Разбиение вершин инцидентностью больше четырех на цепочки

Каждая вершина инцидентности больше четырех в исходном графе представляется как цепочка вершин инцидентностью четыре. На рис. 32 приведен пример такого преобразования. Фиктивные вершины выделены серым цветом.

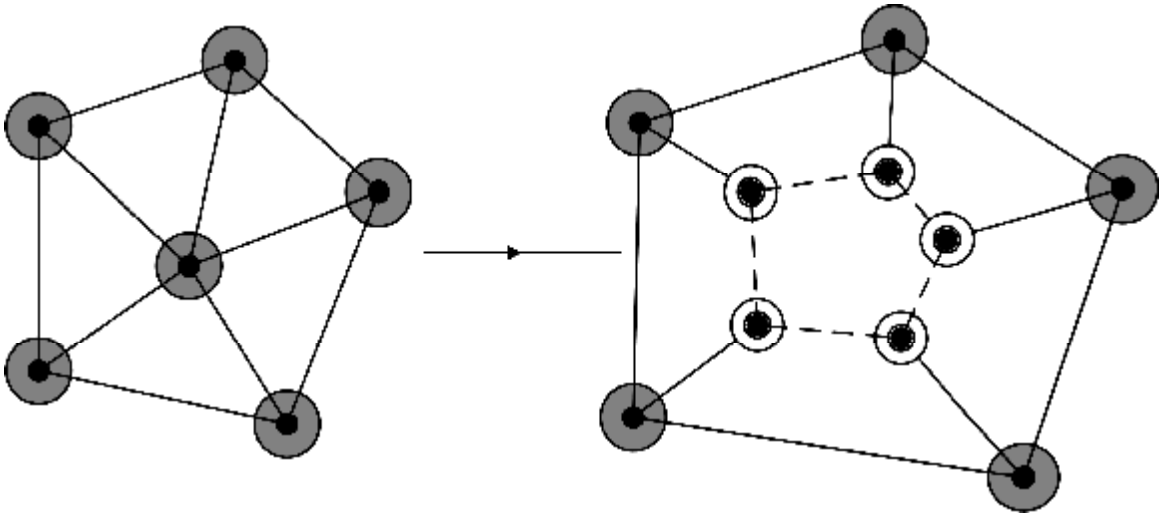

Рис. 32. Преобразование вершин большой инцидентности.

5.7. Представления видимости

После осуществления описанных выше шагов алгоритма получен st-ориентированный планарный диграф с уже выделенными гранями. Пусть степень вершины не превышает четырех (как этого достичь, сказано ниже). Представление графа, в котором вершина является горизонтальным отрезком, а ребро – вертикальным, называется **представлением видимости**. Для построения такого представления нам понадобится алгоритм построения топологической нумерации.

5.7.1. Топологическая нумерация

Топологической нумерацией называют такую нумерацию, при которой каждой вершине присвоен номер, причем $\exists(u,v) \to N(v) > N(u)$. Топологическая нумерация называется **взвешенной**, если $N(v) \geq N(u) + w(u,v)$ и **оптимальной**, если $\max_v N(v) - \min_u N(u) = \min$. В топологической нумерации допустимы вершины с равными номерами, а требование оптимальности есть требование минимизации разброса номеров. Для построения топологической нумерации используем алгоритм из работы [20].

5.7.2. Простое представление видимости

Построим представление видимости:

- построим ассоциированный граф *AG*;
- для обоих графов построим взвешенную оптимальную топологическую нумерацию (*Y* - для самого графа, *X* - для ассоциированного), присвоив ребрам единичный вес;
- для каждой вершины *v* храним абсциссы ее левой (x_L) и правой (x_R) точек в представлении видимости и ее ординату. Положим

  y(v) = Y(v);

  x_L *(v) = X(left(v));*

  x_R *(v) = X(right(v)) – 1;*

- для каждого ребра *e* храним ординаты его нижней (y_B) и верхней (y_T) точек в представлении видимости и его абсциссу. Положим

  x(e) = X(left(e));

  y_B *(e) = Y(orig(e));*

$y_T(e) = Y(dest(e))$.

Построим, в качестве примера, представление видимости для графа с рис. 28. Ребра на рис. 33 направлены сверху вниз, вершины обозначены номерами, находящимися у левого края соответствующего горизонтального отрезка.

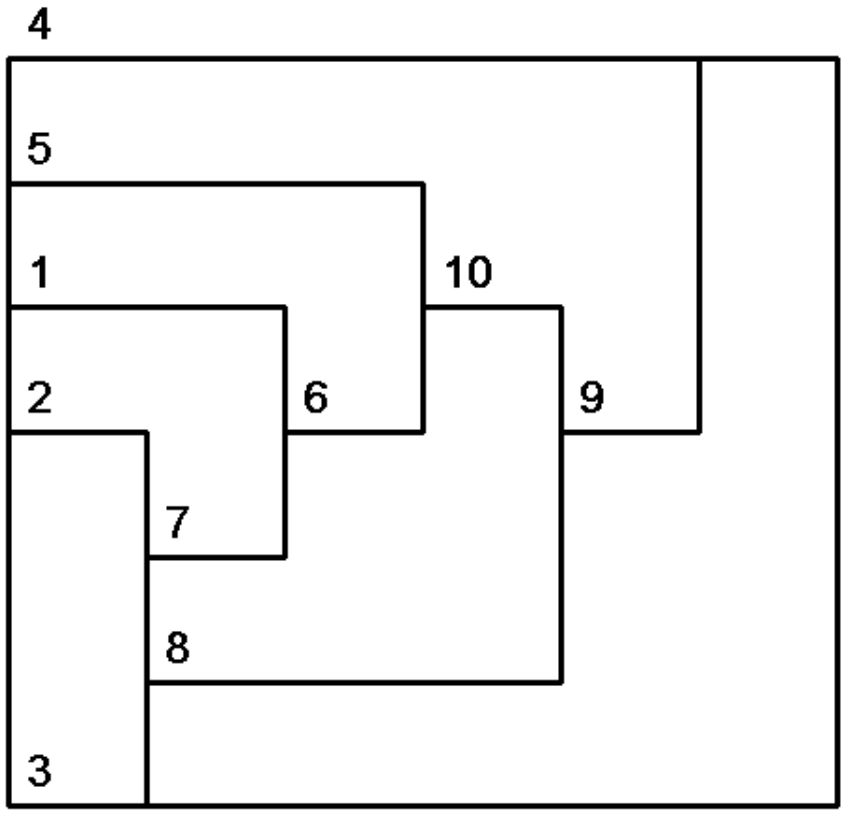

Рис. 33. Представление видимости

5.7.3. Усиленное представление видимости

Простое представление видимости дает результаты, которые порождают укладку со слишком большим количеством изломов на ребрах [8], поэтому построим улучшенный вариант (**усиленное представление видимости**), обеспечивающий выстраивание выделенных путей по вертикальным линиям.

Назовем пути **непересекающимися**, если они не имеют ни общих ребер, ни «пересечений по вершинам». Поясним отличие последнего термина от дизъюнктности путей: пути p_1, p_2 являются непересекающимися, если не существует такой вершины v в графе, что ребра $e_1 \ldots e_2$ (нумерованные вокруг вершины по часовой стрелке) таковы, что $e_1, e_3 \in p_1$ и $e_2, e_4 \in p_2$.

На рис. 34 пути по ребрам e_1, e_3 и e_2, e_4 являются пересекающимися, а по ребрам e_1, e_4 и e_2, e_3 – непересекающимися.

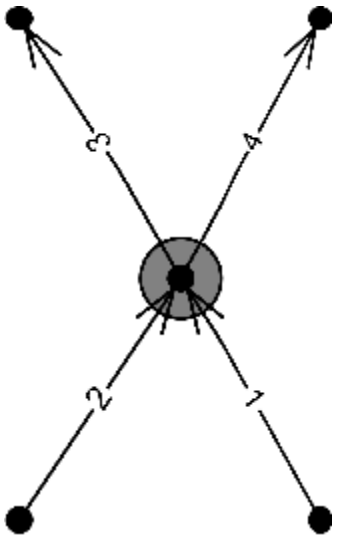

Рис. 34. Пересекающиеся пути

Рассмотрим множество путей P и построим двудольный граф G_P, вершинами в котором будут протограни исходного графа (первая доля) и непересекающиеся пути (вторая доля). Ребра в этом графе строятся по следующему правилу:

- *(f,p)*, если *f = left(e)* для некоторого ребра из пути;
- *(p,g)*, если *g = right(e)* для некоторого ребра из пути.

Построим оптимальную взвешенную топологическую нумерацию Y графа G с единичными весами, и оптимальную взвешенную топологическую нумерацию X графа G_P с весами $1/2$, положив номер источника $X(s) = -1/2$.

- Для каждого пути p в множестве непересекающихся путей и для каждого ребра e из этого пути:

$$x(e) = X(p);$$

$$y_B(e) = Y(orig(e));$$

$$y_T(e) = Y(dest(e)).$$

- Для каждой вершины v в графе:

$$y(v) = Y(v);$$

$x_L(v) = min\ X(p)$ по всем путям p, содержащим вершину v;

$x_R(v) = max\ X(p)$ по всем путям p, содержащим вершину v.

5.7.4. Построение непересекающихся путей

Входящей (исходящей) **медианой** вершины называется среднее из входящих (исходящих) ребер. Относительное расположение ребер на данном этапе уже определено, так как построено разбиение графа на грани.

В st-графе набор непересекающихся путей строится следующим образом: с начала с каждой вершиной v, кроме источника и стока, ассоциирован состоящий из двух ребер (e',e'') путь:

- если в v входит ровно два ребра, то e' – левое из них, а e'' – правое;
- в других случаях (одно или три входящих ребра, так как четыре ребра может входить только в сток) e' – входящая медиана, а e'' – исходящая медиана.

Соединив пути, имеющие общие ребра, получим набор непересекающихся путей (доказательство этого факта приведено в работе [8]). Все те ребра, которые не принадлежат ни одному из путей, добавим в список, как пути, состоящие ровно из одного ребра. В отличие от простого представления видимости (рис. 33) в усиленном (рис. 35) ребра, соответствующие пути по вершинам v_5, v_{10}, v_9, v_8 выстроены в прямую линию.

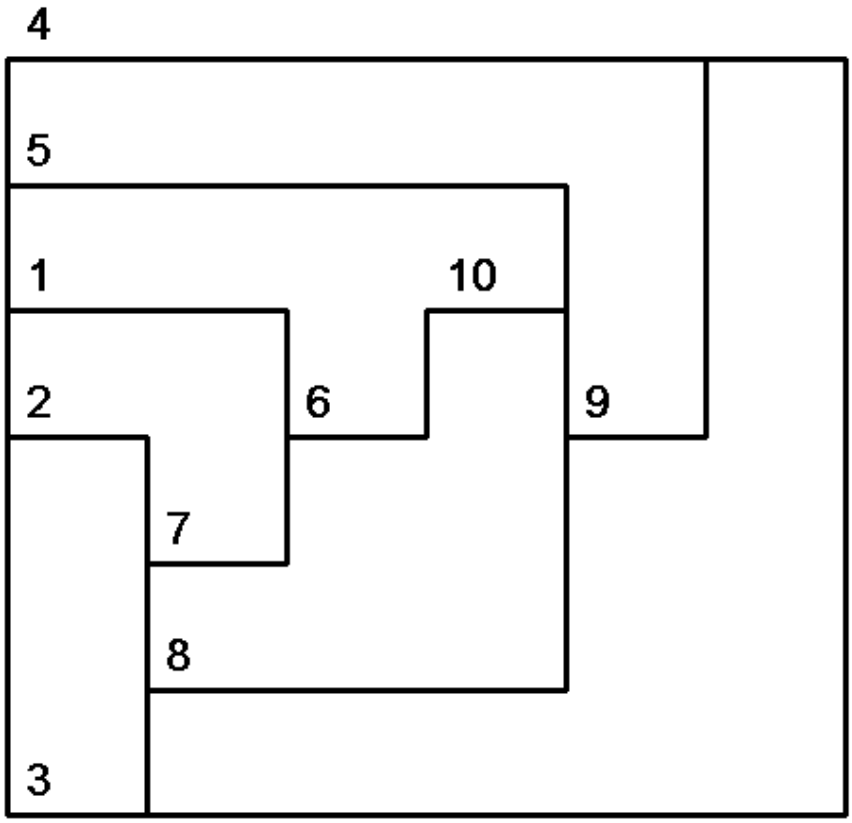

Рис. 35. Усиленное представление видимости

5.8. Ортогональное представление

На базе усиленного представления видимости создадим ортогональное представление, являющееся наиболее важным.

Каждую вершину (кроме источника и стока) изобразим как пересечение ее горизонтального отрезка и вертикальных отрезков ребер на пути *p(v)* (искомое пересечение есть точка).

Источник и сток изобразим как пересечение горизонтального отрезка вершины с вертикальным отрезком ребра медианы.

Каждое ребро, не инцидентное ни источнику, ни стоку изобразим в виде ломаной проходящей через начальную вершину, пересечения отрезков ребра и соответствующих вершин и конечную вершину.

Ортогональное представление является ортогональной укладкой и позволяет изобразить граф на плоскости (на рис. 36 приведено ортогональное представление графа с рис. 32).

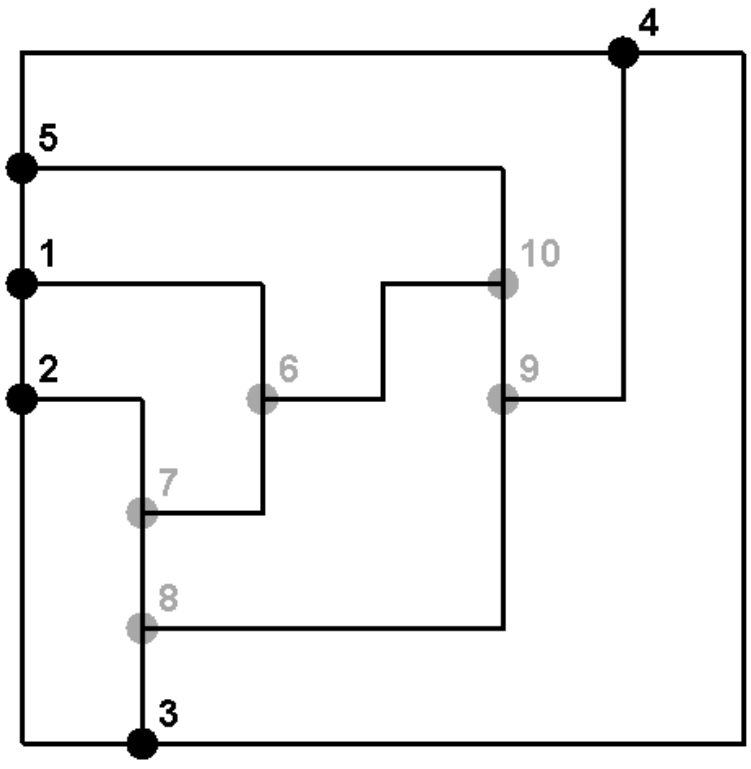

Рис. 36. Ортогональное представление

Здесь представлением вершины является точка. Однако все вершины имеют целочисленные координаты. Поэтому можно задать им ограничивающие прямоугольники со сторонами длиной, например, *0.25*.

5.9. Минимизация площади

Теперь попытаемся минимизировать площадь изображения графа. Рассмотрим задачу в применении к укладке только с прямоугольными гранями (о том, как построить такие грани, сказано ниже). В ходе минимизации площади будем обрабатывать точку излома на ребре также как и вершину графа. Соответственно ребра будут представлять собой только вертикальные и горизонтальные отрезки. Направления ребер графа на этапе минимизации площади не учитываются.

5.9.1. Построение сетей

Сеть N_{VER} содержит по одной вершине для каждой грани[2] и дополнительные вершины – источник и сток (для «левой» и «правой» граней). Дуги в сети соответствуют вертикальным ребрам, общим для двух граней, и направлены слева направо. Дуга допускает минимальный поток 1 и имеет неограниченную пропускную способность. Аналогично построим сеть N_{HOR}. Пример сети N_{VER} приведен на рис. 37.

[2] Здесь можно говорить о гранях, а не о протогранях, так как координаты вершин уже заданы при построении ортогонального представления.

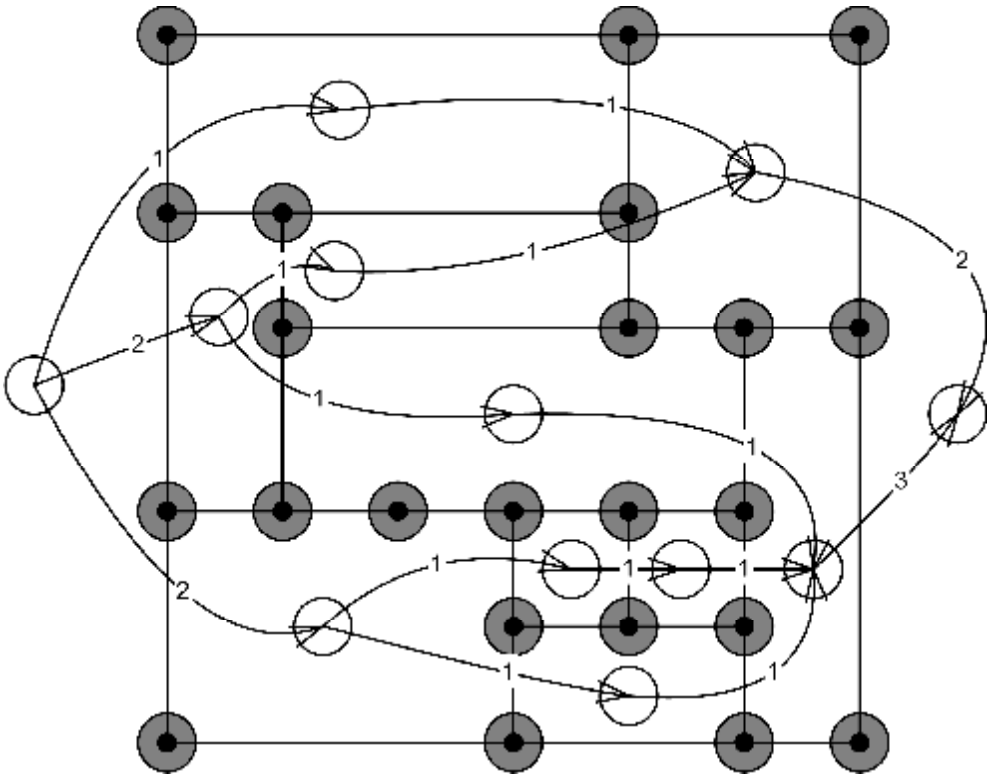

Рис. 37. Пример сети

Минимизировав поток в сети (один из алгоритмов приведен в работе [27]), выполним минимизацию занимаемой площади и суммарной длины ребер. (Длина каждого ребра устанавливается равной потоку через пересекающую его дугу в сети.) На рис. 37 на дугах обозначены потоки, соответствующие минимальному потоку в сети. Окончательный вид графа приведен на рис. 38.

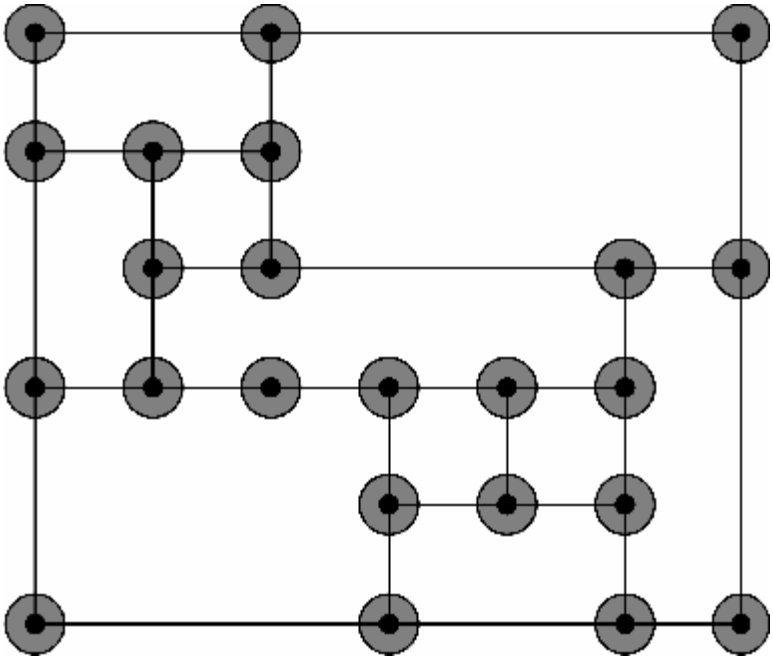

Рис. 38. Минимизация площади

Откажемся от минимизации суммарной длины ребер и применим более простой (но и более быстродействующий) алгоритм, не требующий минимизации потока в сети.

5.9.2. Быстрый алгоритм минимизации площади

Построим графы N_h и N_v. Ребрами графа N_h являются горизонтально ориентированные ребра исходного графа, а вершинами – максимальные связные вертикальные пути. В графе N_v – вертикально ориентированные ребра и максимальные связные горизонтальные пути, соответственно. Данные графы представляют собой *st*-графы. Для каждого из них построим оптимальную взвешенную топологическую нумерацию (X и Y для N_h и N_v соответственно). Положим абсциссу вершины исходного графа равной топологическому номеру соответствующей вершины в X, а ординату – номеру в Y. Пример графа N_v для графа с рис. 37 приведен на рис. 39.

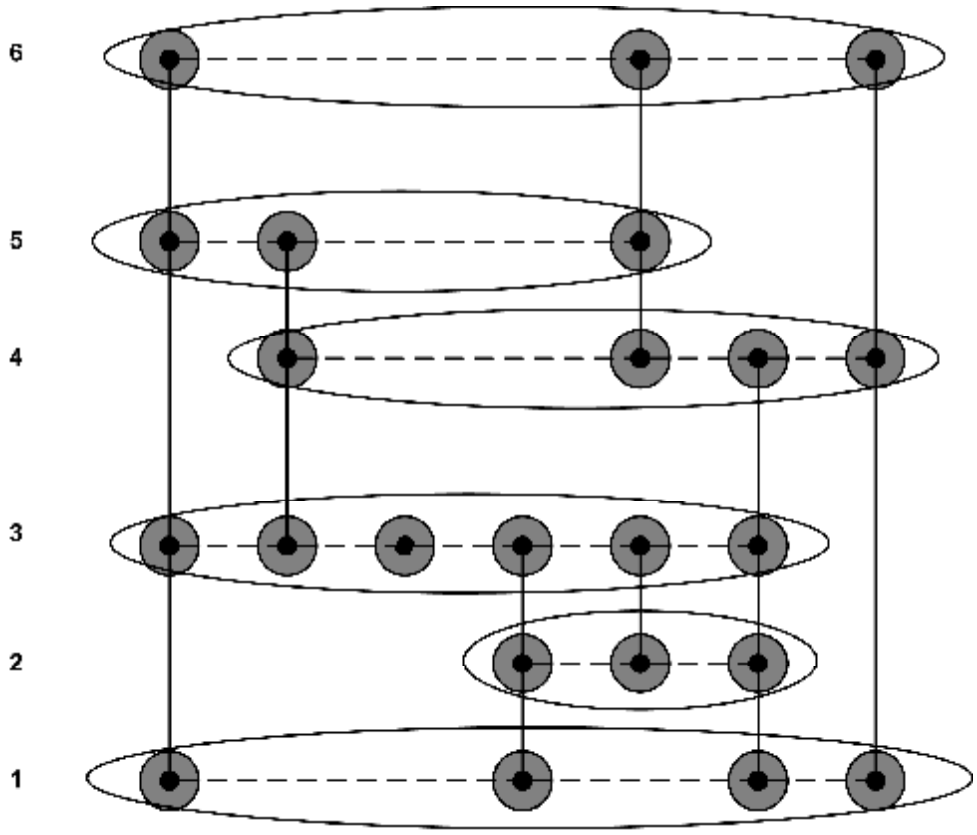

Рис. 39. Граф N_v

Максимальные связные горизонтальные пути выделены эллипсами. Номера в топологической нумерации подписаны в левом столбце.

Из изложенного следует, что более простой алгоритм не приводит к минимизации суммарной длины ребер, но дает достаточно близкий результат. Единственное ребро, дающее увеличение суммарной длины ребер, выделено на рис. 40 пунктиром. Алгоритм, обеспечивающий частичную минимизацию суммарной длины ребер, описан в работе [28].

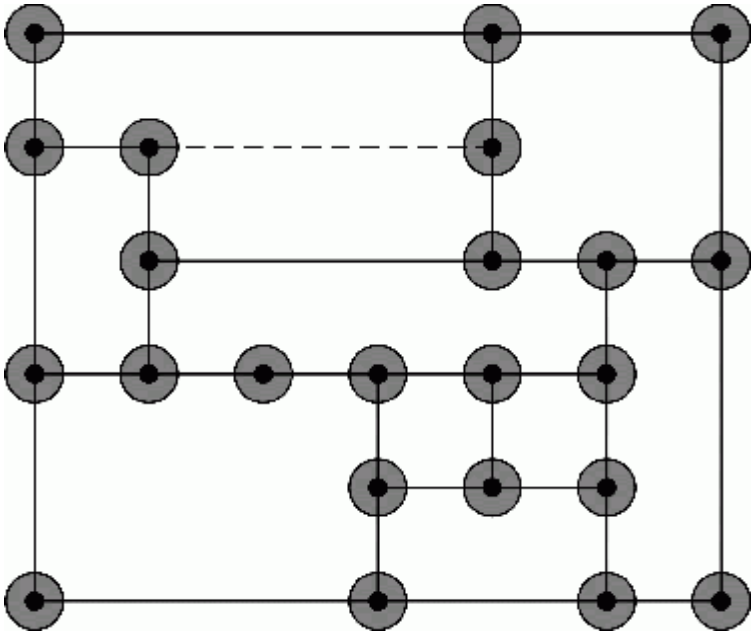

Рис. 40. Минимизация площади. Быстрый алгоритм

Обратим внимание на то, что алгоритм работает только с прямоугольными гранями. Для приведения всех граней к прямоугольному виду используется алгоритм, базирующийся на описанном в работе [8]. Для ортогонального представления с рис. 36 результат выделения прямоугольных граней показан на рис. 41. Для этого графа не удастся уменьшить занимаемую площадь.

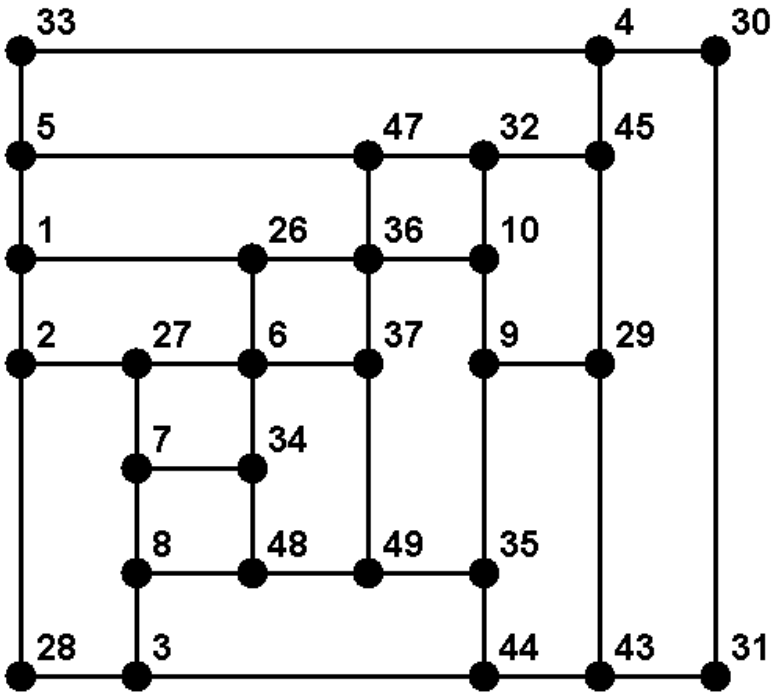

Рис. 41. Прямоугольные грани

5.10. Реализация

В рамках проекта *UniMod* автором реализован алгоритм *GIOTTO,* модифицированный для укладки диаграмм состояний. На данный момент эта реализация не используется для укладки, поскольку не завершена интеграция в среду визуализации диаграмм. Пакет укладки диаграмм *Glayout* входит в дистрибутив *UniMod*, доступный на сайте [1], начиная с версии *UniMod Release 1 Build 11*.

5.11. Выводы

В работе [8] приведена оценка быстродействия алгоритма построения ортогональной укладки без учета времени, затраченного на выделение граней. Асимптотическая оценка – $O(V^2 logV)$, наиболее сложным этапом является минимизация занимаемой графом площади (именно на этом этапе и достигается такая оценка). На графах среднего размера (порядка 50 вершин) рассмотренные реализации алгоритма ([13 – 15]) работали достаточно быстро. Кроме того, необходимо обратить внимание на детерминированность алгоритма, а, значит, и на неизменность укладки для выбранного графа.

На данный момент алгоритм *GIOTTO*, адаптированный к укладке диаграммы состояний реализован, но не используется в *UniMod*, но результаты исследования ряда библиотек, реализующих его для укладки графов, позволяют предполагать, что этот алгоритм позволит строить качественную укладку за небольшое время.

6. ЗАКЛЮЧЕНИЕ

В настоящей работе исследована проблема укладки диаграмм состояний *UML*. Произведено исследование существующих продуктов, как предназначенных для редактирования диаграмм и предоставляющих возможность укладки, так и специализирующихся на укладке графов (разд. 3). Результаты исследования показали, что задача укладки диаграмм состояний является актуальной и практически значимой. Ее решение может найти применение в целом ряде программных продуктов.

Автором разработан смешанный алгоритм укладки диаграмм состояний, который строит начальную укладку элементов с применением метода отжига, и ортогонализует ее с помощью оригинального аналитический алгоритма ортогонализации (разд. 4). Этот алгоритм реализован в продукте *UniMod*.

На базе алгоритма укладки графа *GIOTTO*, являющегося на данный момент наиболее эффективным, разработана его модификация (разд. 5), позволяющая уложить произвольную диаграмму состояний языка *UML*. Для решения ряда возникших подзадач, таких как выделение прямоугольных граней, начальная укладка элементов, разработаны оригинальные алгоритмы. В процессе разработки этих алгоритмов учитывались специфические особенности рассматриваемых диаграмм (разд. 2.4).

Применение качественных алгоритмов укладки делает работу с диаграммами значительно более комфортной, что увеличивает производительной разработчика.

СПИСОК ЛИТЕРАТУРЫ

1. *Гуров В.С., Мазин М.А.* Веб-сайт проекта *UniMod*. http://unimod.sourceforge.net/
2. *Rumbaugh J., Jacobson I., Booch G.* The Unified Modelling Language Reference Manual, Second Edition. MA: Addison-Wesley, 2004.
3. *Шалыто А.А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
4. *Шалыто А.А., Туккель Н.И.* Танки и автоматы // BYTE/Россия. 2003. № 2, с. 69-73. http://is.ifmo.ru/ (раздел «Статьи»).
5. *Новиков Ф.А.* Дискретная математика для программистов. СПб.: Питер, 2004.
6. *Tamassia R.* Advances in the Theory and Practice of Graph Drawing. http://www.cs.brown.edu/people/rt/papers/ordal96/ordal96.html
7. *Касьянов В.Н., Евстигнеев В.А.* Графы в программировании: обработка, визуализация, применение. СПб.: БХВ-Петербург, 2003.
8. *Battista G., Eades P., Tamassia R., Tollis I.* Graph Drawing. Algorithms for the Visualization of Graphs. New Jersey: Prentice Hall, 1999.
9. *Frankel D.* Model Driven Architecture. Applying MDA to Enterprise Computing. Indianapolis: Web Publishing Inc., 2003.
10. *Sugiyama K.* Graph Drawing and Applications for Software and Knowledge Engineers. Singapore: Mainland Press, 2002.
11. *Frutcherman T., Reingold E.* Graph Drawing by Force-directed Placement // Software - Practice and Experience, 1991, vol. 21, pp. 1129-1164.
12. *Makinen E., Seiranta M.* Genetic algorithms for drawing bipartite graphs // International Journal of Computer Mathematics, 1994, vol. 53, No 3, pp. 157 – 166.
13. Веб-сайт проекта AGD (Algorithms for Graph Drawing). http://www.ads.tuwien.ac.at/AGD/

14. Веб-сайт проекта GDT (Graph Drawing Toolkit). http://www.dia.uniroma3.it/~gdt
15. Веб-сайт проекта Graph Layout Toolkit компании Tom Sawyer Software http://www.tomsawyer.com/tsl/tsl.java.php
16. *Battista G., Garg A., Liotta G., Tamassia R., Tassinari E., Vargiu F.* An experimental comparison of four graph drawing algorithms // Computational Geometry, 1997, 7(5-6), pp. 303 - 325.
17. *Tamassia R., Battista G., Batini C.* Automatic graph drawing and readability of diagrams // IEEE Transactions on Systems Man Cybernetics, 1998, 18(1), pp. 61-79.
18. *Klaw G., Mutzel P.* Automatic layout and labeling of state diagrams / Materials of Graph Drawing conference, 1997.
19. *Степанян К.* Автоматическое представление модели UML. СПбГПУ. Дипломная работа. 2003.
20. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999.
21. *Hsu T., Ramachandran V.* On finding a smallest augmentation to biconnect a graph // SIAM Journal on Computing, 1993, 22, pp. 889-912.
22. *Ueno S., Kajitani Y., Wada H.* Minimum augmentation of a tree to a k-edge connected graph // Networks, 1988, 18, pp. 19-25.
23. *Смирнова О.М.* Алгоритм автоматической раскладки диаграммы классов CASE-средства Real. СПбГУ. Дипломная работа. 2000.
24. *Оре О.* Теория графов. М.: Наука, 1980.
25. *Емеличев Р., Мельников О., Сарванов В., Тышкевич Р.* Лекции по теории графов. М.: Наука, 1990.
26. *Harris J.* JGraphEd – A Java Graph Editor and Graph Drawing Framework. Carleton University, School of computer science. Дипломная работа. 2004.

27. *Klein M.* A Primal Method for Minimal Cost Flows with Application to the Assignment and Transportation Problems // Management Science, 1967, 14, pp. 205-220.

28. *Maon Y., Schieber B., Vishkin U.* Parallel ear decomposition search and st-numbering in graphs. Courant Institute of Mathematical Sciences / Ultracomputer note № 102, Computer Science Department Technical Report № 222, 1986.