Российская академия наук
Сибирское отделение
Институт систем информатики
им. А.П. Ершова

Т.А. Золотухин

VISUAL GRAPH LIBRARY: АРХИТЕКТУРА И ВОЗМОЖНОСТИ JAVA-БИБЛИОТЕКИ ДЛЯ ВИЗУАЛИЗАЦИИ ГРАФОВ

Препринт

Новосибирск 2025

В статье описывается архитектура и возможности Visual Graph Library — библиотеки для хранения, укладки и визуализации иерархических атрибутированных графов с портами. Рассматриваются ключевые модули системы и интеграция с платформой Java. Приводится анализ существующих библиотек и приложений для работы с графами.

Siberian Branch of the Russian Academy of Sciences
A.P. Ershov Institute of Informatics Systems

T.A. Zolotuhin

VISUAL GRAPH LIBRARY: ARCHITECTURE AND CAPABILITIES OF A JAVA LIBRARY FOR GRAPH VISUALIZATION

Preprint

Novosibirsk 2025

The paper describes the architecture and capabilities of the Visual Graph Library, designed for storage, layout, and visualization of hierarchical attributed graphs with ports. Key system modules and integration with the Java platform are discussed. An analysis of existing libraries and graph applications is provided.

1. ВВЕДЕНИЕ

Визуализация графов остаётся одной из ключевых задач информатики. Она необходима как для анализа программных систем и компиляторов, так и для исследования социальных сетей, биоинформатики и графовых баз данных. Графовые модели позволяют наглядно представить сложные структуры — синтаксические деревья, управляющие графы, графы вызовов, а также связи между объектами в больших информационных системах.

Однако инструменты общего назначения часто сталкиваются с ограничениями при работе со специализированными структурами — иерархическими атрибутированными графами, содержащими порты. Такие модели требуют специфических алгоритмов укладки и средств рендеринга, учитывающих вложенность и маршрутизацию ребер. Для полноценной работы необходим инструмент, обеспечивающий:

- хранение структур с атрибутами и вложенностью;
- обработку и применение алгоритмов укладки;
- визуализацию в удобной и читаемой форме;
- интеграцию с существующими форматами и экосистемами.

Существующие решения часто ориентированы либо на визуализацию без глубокой интеграции (например, настольные редакторы), либо на алгоритмы без удобного отображения (например, аналитические библиотеки). В результате возникает потребность в инструментах, которые объединяют хранение, навигацию и визуализацию в единой архитектуре.

Visual Graph Library изначально разрабатывалась в рамках диссертационного исследования алгоритма Сугиямы и методов навигации в иерархических атрибутированных графах с портами, но постепенно выросла в самостоятельный проект. Она решает задачи хранения, обработки и отображения графовых структур в рамках Java‑ экосистемы, предлагая открытую архитектуру и возможность свободного использования и развития.

Цель настоящей статьи — представить архитектуру и возможности Visual Graph Library, описать её модель хранения, алгоритмы укладки и средства визуализации, а также привести обзор существующих решений.

2. АРХИТЕКТУРА БИБЛИОТЕКИ

Архитектура Visual Graph Library построена по принципу модульности и разделения ответственности. Библиотека не является монолитом: её подсистемы взаимодействуют через четко определенные контракты, что позволяет использовать их изолированно (например, применять только алгоритмы укладки без графического интерфейса).

Структурно библиотеку можно разделить на четыре уровня:

- **Data Layer (GraphStorage):** Ядро системы. Отвечает за целостность графовой модели, хранение атрибутов и топологии в оперативной памяти.
- **I/O Layer (Decoder/Encoder):** Слой обмена данными. Обеспечивает потоковую загрузку и выгрузку моделей в стандартные форматы (GraphML, DOT, GML).
- **Layout Layer (Layouts):** Алгоритмический блок. Модифицирует атрибуты координат в хранилище, не меняя топологию графа.
- **Presentation Layer (View):** Слой визуализации. Работает поверх хранилища, отвечая за рендеринг на Canvas или в Swing-компоненты.

Такой подход позволяет реализовать полный цикл обработки (Load → Layout → Render → Export) как в интерактивном режиме, так и в Headless-окружении на сервере.

3. ХРАНЕНИЕ ГРАФОВ

Эффективное хранение графовых структур является необходимым условием для их дальнейшей обработки и визуализации. В современных задачах используются не только простые графы, но и более сложные их разновидности — иерархические, атрибутированные, а также графы с портами. Для корректной работы с такими объектами требуется универсальная модель хранения.

Теоретический базис

Модель данных VGL опирается на объединение трех концепций:

1. **Иерархический граф.** Определяется как $\mathbf{G = (V, E, \mu)}$, где вершины могут содержать вложенные графы. Для каждой вершины $\mathbf{v \in V}$ функция вложенности $\mathbf{\mu(v)}$ задает множество вершин и ребер, непосредственно принадлежащих **v**. Таким образом, вершины бывают простыми ($\mathbf{\mu(v) = \emptyset}$) и составными ($\mathbf{\mu(v) \neq \emptyset}$), что позволяет рассматривать граф как дерево вложенных графов.
2. **Атрибутированный граф.** Граф $\mathbf{G_a = (V, E, \alpha_v, \alpha_e)}$ расширяет обычный граф за счет функций атрибутов $\mathbf{\alpha_v: V \to A_v}$ и $\mathbf{\alpha_e: E \to A_e}$, сопоставляющих вершинам и ребрам наборы свойств. Атрибуты включают семантические данные (тип, значение), визуальные характеристики (цвет, форма) и структурные параметры.
3. **Граф с портами.** Граф $\mathbf{Gp = (V, E, P, \pi, \alpha)}$ вводит множество портов **P**, разделенных на входные **P_in** и выходные **P_out**. Функция $\mathbf{\pi: P \to V}$ сопоставляет каждому порту вершину, которой он принадлежит. Ребра определяются как $\mathbf{E \subseteq P\_out \times P\_in}$, то есть соединяют выходные порты с входными. Такой формализм позволяет точно моделировать потоки данных.

Реализация модели в Visual Graph Library

Физически граф хранится в виде плоских списков записей (Records), связанных идентификаторами:

- **GraphModelRecord** — Корневой контейнер, хранящий весь иерархический граф как единый объект. Обеспечивает целостность структуры и хранит глобальные параметры (например, дефолтные настройки укладки, дефолтный цвет и форму для вершины и т.д.).
- **VertexRecord** — Запись вершины. Вершина может быть *простой* (без портов), *вершиной с портами* (содержит упорядоченные списки **P_in** и **P_out**) или *составной* (является родителем для вложенных элементов).
- **EdgeRecord** — Описывает связь. Ребро может соединять вершины напрямую либо указывать конкретные порты. Хранит атрибуты маршрутизации (точки изгиба) и ссылку на родительский контекст.
- **AttributeRecord** — Универсальное хранилище свойств «ключ–значение». Атрибут может быть прикреплен к вершине, ребру или порту. Это позволяет динамически расширять модель семантическими или визуальными данными без изменения схемы БД.

Функциональность модуля GraphStorage

Модуль GraphStorage предоставляет транзакционный API для управления данными:

- **CRUD-операции:** Создание, чтение, обновление и удаление сущностей по идентификатору. Удаление реализует каскадную логику: удаление модели или составной вершины автоматически очищает все вложенные элементы и инцидентные связи, предотвращая появление «висячих» ссылок.

- **Обход (Traversal):** Реализован метод обхода в глубину (DFS) с поддержкой составных вершин. Алгоритм корректно переходит на уровни вложенности, вызывая обработчики («визиторы») для каждой модели, вершины и ребра, что необходимо для анализа топологии.
- **Копирование:** Поддерживается глубокое копирование подграфов между моделями с сохранением структуры вложенности и атрибутов. Механизм фильтрации позволяет гибко выбирать переносимые элементы (например, копировать только выделенный фрагмент схемы).

Таким образом, модуль обеспечивает строгую основу для последующих этапов конвейера — алгоритмов укладки, навигации и рендеринга.

4. ВИЗУАЛИЗАЦИЯ

Архитектура модуля визуализации спроектирована для поддержки двух сценариев использования: встраивание интерактивных компонентов в Desktop-приложения (Swing) и генерация изображений в серверном окружении (Headless). Для реализации этих требований логика рендеринга разделена на уровни абстракции: базовый движок, слой оптимизации (кэширование) и UI-обертка.

1. Движок рендеринга: GraphViewCanvas

Фундаментом подсистемы является класс GraphViewCanvas — движок визуализации, работающий непосредственно с моделью GraphStorage. Ключевая архитектурная особенность — полная изоляция от событийной модели AWT/Swing и потока обработки событий (EDT). Это позволяет использовать класс для фоновых задач и генерации отчетов на серверах без графической подсистемы.

Функциональные возможности:

- **Рендеринг:** Отрисовка графа в BufferedImage с поддержкой произвольного масштабирования (Zoom) и трансформации координат.
- **Сетка:** Поддержка конфигурируемых слоев сетки (Background/Foreground) для позиционирования элементов.
- **Стилизация:** Управление визуальными атрибутами вершин и ребер (шрифты, цвета, формы, стили линий).
- **Программное управление:** API для управления состоянием выделения (Selection Model) и навигации, доступное даже без визуального отображения компонента.

2. Оптимизация производительности: GraphViewCachedCanvas

Для интерактивных сценариев прямого рендеринга недостаточно: перерисовка сложных графов (тысячи элементов) при навигации вызывает задержки UI. Проблема решена в классе GraphViewCachedCanvas, который наследует базовый функционал и добавляет слой кэширования в оперативной памяти.

Принцип работы:

- **Тайловое/Фрагментарное кэширование:** Канвас хранит подготовленные растровые фрагменты изображения для текущего масштаба.
- **Асинхронное обновление:** Перерисовка кэша вынесена из UI-потока, что предотвращает блокировку интерфейса.
- **Триггеры инвалидации:** Реализована логика «безопасных зон» (safe regions). Система переиспользует кэшированный кадр при малых смещениях и инициирует перерисовку только при выходе за границы буфера или изменении масштаба.

Примечание: Использование кэширующего канваса рекомендовано только для интерактивных компонентов. Для разового экспорта изображений эффективнее использовать базовый GraphViewCanvas, исключая накладные расходы на управление кэшем.

3. Интеграция в UI: GraphViewComponent

Компонент GraphViewComponent представляет собой реализацию javax.swing.JComponent и служит связующим звеном между движком рендеринга и пользователем. Он инкапсулирует GraphViewCachedCanvas и предоставляет стандартные механизмы взаимодействия.

Архитектура компонента:

- **Viewport:** Реализация интерфейса Scrollable для интеграции с JScrollPane, поддержка управления смещением (Offset).
- **Слои (Overlays):** Многослойная архитектура отрисовки. Поверх основного графа накладываются служебные слои: рамка выделения (SelectionOverlay), инструменты редактирования и визуальные подсказки.
- **Интерактивность:** Встроенные контроллеры событий мыши (MouseListeners) для выделения объектов, панорамирования и вызова контекстных меню.
- **Look-and-Feel:** Автоматическая адаптация цветовой схемы и стилей под текущую тему оформления Swing-приложения.

4. Фабрика: GraphViewFactory

Для упрощения инициализации иерархии объектов (Canvas → CachedCanvas → Component) применяется паттерн Фабрика. Класс GraphViewFactory обеспечивает корректную сборку компонентов и потокобезопасность.

Ключевая особенность: Фабрика управляет контекстом потоков. При попытке создания UI-компонента из фонового потока, метод автоматически синхронизирует вызов через SwingUtilities.invokeAndWait. Это предотвращает распространенные ошибки многопоточности, характерные для работы с библиотеками AWT/Swing.

5. АЛГОРИТМЫ УКЛАДКИ

Модуль Layouts отвечает за автоматическое вычисление геометрических атрибутов графа. В отличие от модуля визуализации, который читает данные, алгоритмы укладки модифицируют состояние GraphStorage, рассчитывая координаты вершин (**x**, **y**, **width**, **height**) и траектории ребер (списки контрольных точек).

В библиотеке реализованы три фундаментальные стратегии укладки, покрывающие основные классы графовых структур.

1. Иерархическая укладка (HierarchicalLayout)

Базовый алгоритм для ориентированных графов (DAG), основанный на методе Сугиямы (Sugiyama framework). Предназначен для визуализации структур с явным направлением потока: блок-схем, диаграмм классов UML, графов зависимостей и компиляторных представлений (CFG).

Алгоритмическая реализация: Укладка выполняется рекурсивно. Алгоритм обходит граф в глубину (DFS), начиная с самых глубоких уровней вложенности. Для каждого подграфа (контейнера) выполняется классический конвейер Сугиямы:

1. **Устранение циклов (Cycle Breaking):** Временно разворачивает ребра, образующие циклы, чтобы граф стал ациклическим. Такие «обратные» связи (back-edges) при рендеринге отрисовываются пунктиром.

2. **Распределение по слоям (Layering):** Вершины распределяются по дискретным уровням (рангам). Входные порты (**P_in**) ориентируются вверх, выходные (**P_out**) — вниз, что обеспечивает читаемость «сверху-вниз».
3. **Минимизация пересечений (Crossing Reduction):** Итеративная перестановка вершин внутри слоев для уменьшения числа пересечений ребер.
4. **Присвоение координат (Coordinate Assignment):** Вычисление точных позиций **X** и **Y** с учетом размеров вложенных контейнеров.
5. **Маршрутизация ребер (Routing):** Построение траекторий связей.

Параметры конфигурации: Класс HierarchicalLayoutSettings предоставляет гранулярный контроль над этапами:

- **Стратегия:** Выбор реализации этапов (например, CUSTOM_LAYERING_ALGORITHM).
- **Геометрия:** Настройка минимальных отступов между слоями (minVSpaceBetweenRows) и объектами (minHSpaceBetweenVertices).
- **Маршрутизация:** Поддержка стилей линий: POLYLINE (ломаные), ORTHOGONAL (ортогональные) или SPLINE (сглаженные кривые).
- **Компактность:** Флаги collapseEdgeSrc/Trg позволяют агрегировать пучки ребер, исходящие из одной точки, для повышения плотности укладки.

2. Случайная укладка (RandomLayout)

Алгоритм стохастического размещения. Не выполняет топологического анализа и не учитывает иерархию или направление связей.

- **Сложность: O(V)**, где **V** — количество вершин.
- **Применение:** Используется для инициализации координат перед запуском силовых алгоритмов, для визуализации разреженных несвязных графов или для нагрузочного тестирования подсистемы рендеринга (Stress Testing) на графах большого объема.

3. Круговая укладка (CircleLayout)

Алгоритм размещает вершины на периметре окружности или дуг. Эффективен для анализа сетевых топологий, кластеров и структур без явно выраженной иерархии.

Ключевые особенности:

- **Кластеризация:** Поддержка группировки фрагментов графа. Связанные компоненты размещаются компактными секторами.
- **Оптимизация связей:** Параметр crossingReductionAlgorithm активирует эвристики (например, сортировку или группировку) для минимизации пересечений хорд внутри круга.
- **Стилизация:** Поддерживается отрисовка ребер прямыми линиями или дугами (Arc), а также скругление углов траекторий (roundingRadius) для улучшения визуального восприятия.

6. ИМПОРТ И ЭКСПОРТ ГРАФОВЫХ МОДЕЛЕЙ

Модули Decoder и Encoder обеспечивают обмен данными с внешней средой, позволяя преобразовывать графовые модели из файловых форматов во внутреннее представление GraphStorage и обратно. Архитектура подсистем построена на принципах **потоковой обработки** (Stream-based processing), что изолирует ядро библиотеки от источника данных (файл, сетевой сокет, область памяти).

1. Импорт данных (Decoder)

Подсистема декодирования отвечает за парсинг входного потока, валидацию синтаксиса и реконструкцию топологии графа.

Центральным элементом архитектуры является абстрактный класс GraphDecoder, определяющий контракт для всех парсеров:

- **Входные данные:** Принимает имя графа, InputStream и целевой экземпляр GraphStorage.
- **Результат:** Возвращает идентификатор (ID) корневой модели графа после успешного разбора.
- **Валидация:** В случае нарушения структуры данных выбрасывается исключение GraphDecoderException. Оно содержит детальный контекст ошибки (номер строки, некорректный токен), что критически важно для отладки входных файлов.

Поддерживаемые форматы: Выбор реализации осуществляется через фабрику GraphDecoderFactory.

- **GraphML (GraphMLDecoder):** Основной XML-формат. Обеспечивает полную поддержку вложенных графов, типизированных атрибутов и метаданных топологии.
- **GML (GMLDecoder):** Легковесный текстовый формат (Graph Modelling Language), удобный для ручного редактирования простых структур.
- **DOT / GV (DotDecoder):** Формат пакета Graphviz. Поддержка обеспечивает совместимость с широким спектром существующих утилит генерации графов.
- **ZIP (ZipDecoder):** Прозрачная работа со сжатыми архивами. Позволяет загружать модели, упакованные в ZIP-контейнеры, без предварительной распаковки.

2. Экспорт данных (Encoder)

Подсистема кодирования выполняет сериализацию состояния GraphStorage во внешний формат. Важно отметить, что экспортируется не только топология (вершины и ребра), но и **метаданные визуализации**: вычисленные координаты, размеры элементов и стили. Это гарантирует сохранение состояния (state-preservation) — граф будет восстановлен в точности таким, каким его видел пользователь.

Базовый класс GraphEncoder унифицирует процесс:

- **Входные данные:** Принимает ID модели, источник данных GraphStorage и целевой OutputStream.
- **Процесс:** Выполняет итеративный обход структуры и запись в поток.
- **Надежность:** Ошибки IO или нарушения целостности данных оборачиваются в GraphEncoderException.

Реализация и расширяемость: В текущей версии реализован экспорт в формат **GraphML** (GraphMLEncoder), так как он наиболее полно покрывает возможности модели данных библиотеки. Архитектура допускает расширение списка форматов (например, добавление JSON или SVG) путем наследования от GraphEncoder и регистрации новой реализации в фабрике, без модификации ядра библиотеки.

7. АНАЛИЗ СУЩЕСТВУЮЩИХ БИБЛИОТЕК И ПРИЛОЖЕНИЙ ДЛЯ ГРАФОВ

В этой главе проведен обзор графовых решений: от низкоуровневых алгоритмических библиотек до полноценных экосистем визуализации. Цель анализа — определить место разрабатываемой библиотеки среди существующих инструментов, выявить их сильные и слабые стороны, а также обозначить сценарии их применения.

Все решения можно разделить на две большие категории:

- **Чистые библиотечные решения:** инструменты (JGraphT, NetworkX, D3.js), предоставляющие API для использования внутри программного кода. Их фокус — алгоритмы и гибкость интеграции.
- **Экосистемы:** решения, выросшие вокруг GUI-редакторов (yFiles, Gephi, Graphviz). Они часто предоставляют и визуальный инструмент, и библиотеку для разработчиков, но могут навязывать собственные форматы и архитектурные паттерны.

Категория	Представители	Особенности
Чистые библиотечные решения	JGraphT, NetworkX, igraph, Boost Graph Library, OGDF, JGraphX, GraphStream, Prefuse, Sigma.js, G6.js, D3.js, Vis.js, Graph-tool	Основной фокус — API, алгоритмы, укладки и визуализация; используются в коде и интеграциях
Экосистемы (инструменты + библиотеки)	yEd ↔ yFiles, Gephi ↔ Gephi Toolkit, Tulip ↔ Tulip Library, Cytoscape Desktop ↔ Cytoscape.js, Graphviz ↔ GVEdit/xdot/ZGRViewer, Higres ↔ внутренние модули	Предоставляют и GUI- редактор, и библиотечные решения для интеграции; часто включают собственные форматы и укладчики

Для систематизации данные сгруппированы в три таблицы по критериям: базовые сведения, функциональность ядра и возможности интеграции.

7.1. Базовые сведения и жизнеспособность

Первый этап анализа касается технологического стека, условий лицензирования и поддерживаемого масштаба данных. Эти параметры определяют возможность внедрения инструмента в производственную среду.

Библиотека	Статус	Языки интеграции	Лицензия	Масштабируемость (практический предел)
yFiles	развивается	JS, Java, .NET	Коммерческая	>1 млн вершин/дуг
Gephi Toolkit	развивается	Java	GPL/CDDL	~100k–300k
Tulip Library	развивается	C++, Python	LGPL	~1 млн

Cytoscape.js	активно развивается	JS	MIT	~50k
Graphviz	развивается	C, Python	EPL/CPL	~100k
Higres	не развивается	C/C++	Академическая	~10k–50k
JGraphT	активно развивается	Java	EPL‑ 2.0	~100k (алгоритмы, без визуализации)
NetworkX	активно развивается	Python	BSD	~100k (алгоритмы, без визуализации)
igraph	развивается	C, C++, Python, R	GPL	>1 млн
Boost Graph Library	развивается	C++	Boost	>1 млн (алгоритмы, без визуализации)
OGDF	развивается	C++	GPL	~100k–500k
JGraphX (mxGraph)	не развивается	Java	BSD	~10k–20k
GraphStream	не развивается	Java	LGPL	~50k–100k
Prefuse	не развивается	Java	BSD	~10k–20k
Sigma.js	развивается	JS	MIT	~50k–100k
G6.js (AntV)	активно развивается	JS/TS	MIT	~100k
D3.js	развивается	JS	ISC	~10k–50k

Vis.js	развивается	JS	MIT/Apache	~10k–50k
Graph-tool	развивается	Python (C++ backend)	LGPL	>1 млн

Наблюдается четкая сегментация инструментов по языковым платформам и статусу поддержки:

- **Enterprise-сектор:** Безусловным лидером является **yFiles**. Библиотека поддерживает все основные платформы (Java, .NET, JS) и масштабируется до миллионов элементов, но коммерческая лицензия ограничивает её применение в Open Source и бюджетных проектах.
- **Java-сегмент:** Наблюдается дисбаланс. Существует мощная алгоритмическая база в лице **JGraphT**, однако сегмент визуализации представлен либо устаревшими решениями (**JGraphX**, **Prefuse**), либо библиотеками со сложной интеграцией (**Gephi Toolkit**).
- **Web-сегмент:** Активно развиваются JS-библиотеки (**Cytoscape.js**, **G6**, **D3.js**). Их использование в Java Desktop приложениях возможно только через компонент WebView, что усложняет архитектуру и увеличивает потребление ресурсов.
- **Научный сектор:** Python и C++ (**NetworkX**, **igraph**) доминируют в задачах анализа данных, но практически не предлагают встраиваемых инструментов визуализации.

7.2. Функциональность ядра

Оценка пригодности библиотеки для инженерных задач требует анализа ее функционального ядра. Критически важными являются три аспекта:

1. **Поддержка форматов:** Работа со стандартами GraphML, DOT, GML для обмена данными.
2. **Модель данных:** Поддержка атрибутов и вложенности (nested graphs). Игнорирование вложенности делает библиотеку непригодной для визуализации иерархических систем (UML, блок-схемы).
3. **Алгоритмы укладки:** Наличие встроенных механизмов автоматического размещения вершин (Layouts). Это ключевой параметр для инструментов визуализации, так как ручная расстановка элементов в больших графах невозможна.

Библиотека	**Форматы (GraphML/DOT/GML)**	**Атрибуты**	**Nested**	**Алгоритмы укладки**
yFiles	GraphML, GML	Да	Да	Иерархические, силовые, круговые, ортогональные, спец.

Gephi Toolkit	GraphML, DOT, GML	Да	Нет	Иерархические, силовые, круговые
Tulip Library	GraphML, GML	Да	Подграфы (ограниченно)	Иерархические, силовые, круговые, спец.
Cytoscape.js	GraphML, GML (плагины)	Да	Нет	Иерархические, силовые, круговые
Graphviz	DOT	Да (через атрибуты DOT)	Да (subgraph)	Иерархические, силовые, круговые, радиальные
Higres	частично GraphML	Да	Да	Иерархические, вложенные
JGraphT	GraphML, DOT, GML	Да	Нет	Иерархические, силовые, круговые
NetworkX	GraphML, DOT, GML	Да	Нет	Иерархические, силовые, круговые
igraph	GraphML, GML	Ограниченно	Нет	Силовые, круговые
Boost Graph Library	—	Ограниченно (property maps)	Нет	Нет встроенных
OGDF	GraphML, GML	Да	Нет	Иерархические, ортогональные, круговые

JGraphX (mxGraph)	—	Да	Группировка (но не nested)	Иерархические, силовые, круговые
GraphStream	GraphML	Да	Частично	Силовые (динамические), иерархические
Prefuse	—	Да	Нет	Иерархические, силовые, круговые
Sigma.js	GraphML, GML (конвертеры)	Да	Нет	Силовые, круговые
G6.js (AntV)	GraphML, GML (плагины)	Да	Нет	Иерархические, силовые, круговые, спец.
D3.js	—	Да (через data binding)	Нет	Иерархические, силовые, круговые
Vis.js	—	Да	Нет	Иерархические, силовые
Graph-tool	GraphML, GML	Да	Нет	Силовые, круговые, радиальные

Таблица демонстрирует существенные архитектурные различия между инструментами общего назначения и специализированными средствами визуализации:

- **Алгоритмический базис (Force-directed vs Layered):** Большинство Open Source библиотек реализуют силовые алгоритмы. Они эффективны для визуализации сетей, но не гарантируют сохранение структурной иерархии, необходимой для инженерных диаграмм. Качественные уровневые (иерархические) алгоритмы, решающие задачу минимизации пересечений, полноценно представлены только в **yFiles**, **Graphviz** и десктопных редакторах.
- **Топология и вложенность:** Большинство библиотек (JGraphT, NetworkX) оперируют классической моделью «плоского» графа. Отсутствие нативной поддержки составных графов (compound graphs) приводит к тому, что алгоритмы укладки

разрушают визуальную целостность контейнеров, размещая дочерние узлы вне границ родительских.

- **Маршрутизация ребер:** В инженерной графике важна ортогональная трассировка и привязка ребер к портам. Библиотеки общего назначения чаще всего используют центроидную трассировку (прямая линия от центра к центру), что снижает читаемость схем.

7.3. Интеграция и дополнительные возможности

Завершающий этап сравнения фокусируется на возможностях, выходящих за рамки базового рендеринга: экспорт результатов, поддержка динамических изменений структуры и встроенная аналитика.

Библиотека	Алгоритмы анализа	Экспорт (изображения)	Экспорт (графовые форматы)	Динамические графы
yFiles	Базовые + расширенные	PNG, SVG, PDF, HTML	GraphML, GML, JSON	Да
Gephi Toolkit	Базовые + расширенные	PNG, SVG, PDF	GraphML, GML, CSV	Нет
Tulip Library	Базовые + расширенные	PNG, SVG, PDF	GraphML, GML	Да
Cytoscape.js	Базовые	HTML, SVG	JSON, GraphML (плагины)	Нет
Graphviz	Нет	PNG, SVG	—	Нет
Higres	Базовые	PNG	частично GraphML	Да
JGraphT	Базовые + расширенные	—	GraphML, DOT, GML	Нет
NetworkX	Базовые + расширенные	—	GraphML, GML, adjacency list	Нет
igraph	Базовые + расширенные	—	GraphML, GML, Pajek, adjacency list	Нет

Boost Graph Library	Базовые	—	DOT (адаптеры)	Нет
OGDF	Базовые	PNG	GraphML, GML	Нет
JGraphX (mxGraph)	Базовые	PNG, SVG, PDF	XML (mxGraph формат)	Нет
GraphStream	Базовые + расширенные	PNG	DGS, GraphML	Да
Prefuse	Базовые	PNG, SVG	CSV	Нет
Sigma.js	Базовые	HTML, Canvas	JSON	Да
G6.js (AntV)	Базовые	HTML, Canvas, SVG	JSON	Да
D3.js	Базовые	SVG, HTML	JSON, CSV	Да
Vis.js	Базовые	HTML, Canvas	JSON	Да
Graph-tool	Базовые + расширенные	—	GraphML, GML	Нет

На основе собранных данных можно классифицировать существующие решения на три группы:

1. **Зрелые промышленные стандарты (yFiles, Graphviz, Cytoscape):** Обладают полным набором функций (укладка, экспорт, форматы). Главные недостатки — либо высокая стоимость (yFiles), либо сложность встраивания в Java-код (Graphviz требует запуска внешнего процесса).
2. **Алгоритмические библиотеки (JGraphT, NetworkX, igraph):** Эффективны для вычислений и анализа топологии, но не предоставляют средств визуализации «из коробки», требуя подключения сторонних рендереров.
3. **Нишевые и устаревшие решения (JGraphX, GraphStream, Prefuse):** Либо прекратили развитие, либо решают узкие задачи (например, только динамические графы), что создает риски при их использовании в новых проектах.

8. ЗАКЛЮЧЕНИЕ

В статье была рассмотрена проблема инструментальной поддержки визуализации **иерархических атрибутированных графов с портами**. Как было отмечено во введении, существующие решения часто поляризованы: они предлагают либо мощный алгоритмический аппарат без средств отображения, либо развитый графический интерфейс без возможностей гибкой интеграции.

Visual Graph Library (VGL) предлагает подход, объединяющий модель хранения сложных структур, алгоритмы автоматической укладки и подсистему рендеринга в единой модульной архитектуре. Анализ существующих библиотек показывает, что VGL не дублирует функциональность теоретико-графовых пакетов (таких как JGraphT) или Enterprise-платформ (yFiles), а ориентирована на прикладные инженерные задачи, требующие работы с вложенностью и семантикой портов в среде Java.

Основные результаты:

1. **Универсальная модель хранения:** Реализован механизм GraphStorage, поддерживающий глубокую иерархию и типизированные атрибуты, что позволяет корректно моделировать потоки данных и управляющие графы.
2. **Гибридная визуализация:** Архитектура разделяет логику рендеринга и UI, обеспечивая работу как в интерактивных Swing-приложениях, так и в серверных (Headless) сценариях.
3. **Алгоритмическая база:** Адаптация метода Сугиямы для работы с контейнерами и портами позволяет автоматически строить читаемые схемы для сложных инженерных моделей.

Направления дальнейших исследований и разработки: Развитие библиотеки сфокусировано на проблемах масштабируемости и обработки графов сверхбольшого размера:

- **Структурная навигация:** Разработка компонента «Graph Explorer» для древовидного отображения иерархии (аналог Project View в IDE) с возможностью частичной загрузки подграфов.
- **Оптимизация хранения:** Переход к гибридной модели хранения (Memory/Disk) для работы с графами, содержащими миллионы вершин.
- **Расширение алгоритмов:** Оптимизация эвристик укладки для плотных графов и реализация физических (Force-Directed) алгоритмов для анализа топологии.

Библиотека распространяется с открытым исходным кодом, что позволяет использовать её как базу для создания специализированных средств визуализации и анализа программных систем.